CS 320 Project Two: Summary and Reflections Report

Rafael Salazar

This project was focused on developing and testing three core services for a mobile application: the Contact Service, Task Service, and Appointment Service. Each of these components had specific requirements provided by the customer, and my responsibility was to build the classes and create JUnit tests to verify that those requirements were fully met. This project helped me strengthen my skills in test-driven development, while also showing how even small errors in logic or data handling can lead to larger issues if not properly tested. My approach throughout this process was to test thoroughly, think critically about edge cases, and learn how to maintain both accuracy and efficiency in my code.

## Summary

### Unit Testing Approach

My unit testing approach centered on verifying that each class, the Contact, Task, and Appointment, followed the requirements exactly as described. I started with small, isolated tests that validated field constraints and then expanded to confirm that update and delete methods in the service classes behaved correctly. The goal was to make sure that any invalid data was caught early and that no updates could occur on restricted fields like IDs.

For example, in the Contact class, I tested that the contact ID could not be null, longer than 10 characters, or changed after creation. I used assertions like `assertThrows(IllegalArgumentException.class, ...)` to make sure the constructor handled bad input correctly. In the ContactService class, I verified that only contacts with unique IDs could be added and that trying to delete or update a non-existent contact returned an error. Similar methods were applied to the Task and Appointment services. Each test directly reflected a requirement, ensuring a one-to-one link between what was expected and what was verified.

For the Task service, I tested boundaries for name and description lengths and ensured updates could only modify valid fields. The Appointment service required additional logic since it involved date validation. Using `new Date()` and comparison logic like `before(new Date())`, I confirmed that appointments could not be created in the past. Each of these checks reinforced real-world behavior that users would expect from a professional scheduling application.

Across all files, my testing strategy focused on both positive and negative scenarios. Positive tests confirmed that valid inputs worked correctly, while negative tests verified that invalid or unexpected inputs were handled gracefully. The structure of the tests followed a clear pattern: setup, execution, assertion, and teardown. Using JUnit 5’s `@BeforeEach` annotation helped ensure a clean environment for each test run.

### Alignment with Software Requirements

My approach was closely aligned with the software requirements provided in the project outline. Each test was written specifically to match a rule defined in the customer’s expectations—for instance, name and ID length limits or required non-null fields. For every method, I included at least one test to check successful behavior and another to test failure conditions. This way, the unit tests acted as both a quality check and a form of living documentation that showed what each class was supposed to do.

After running my tests in Eclipse through Codio, I confirmed that my coverage consistently reached over 80%. The JUnit coverage tool showed green bars across nearly all methods, which indicated that most logical paths were tested. This gave me confidence that the services behaved as intended, even when handling edge cases.

### Experience Writing JUnit Tests

Writing the JUnit tests helped me understand how to think like both a developer and a tester. Early on, I realized that my first instinct was to test only what I thought would work—but effective testing means trying to break your own code. For example, when testing the Task class, I initially focused only on valid inputs, but I later added tests for null names and overly long descriptions. Those additions quickly revealed small gaps in my validation logic.

I also learned how valuable assertions can be. Methods like `assertEquals`, `assertNotNull`, and `assertThrows` made my tests both readable and reliable. For efficiency, I created reusable helper methods within my test classes to avoid repeating setup code. For example, in the ContactServiceTest, I had a helper method to quickly create and return a sample Contact object. This cut down on repetitive code and made it easier to modify tests as requirements changed.

Overall, this experience made me more confident in using JUnit 5 and showed me the importance of building small, testable components rather than large, complex methods that are harder to verify.

## Reflection

### Testing Techniques

Throughout this project, I used several software testing techniques, mainly unit testing, boundary testing, and negative testing. Unit testing allowed me to focus on small, isolated pieces of code and confirm their behavior in a controlled environment. Boundary testing was useful for verifying that inputs like name lengths and description limits stayed within defined boundaries. Negative testing, on the other hand, focused on how the system handled invalid or unexpected input.

For example, I used boundary tests in the Task class to verify that a 20-character name passed validation but a 21-character one did not. I also used negative testing in the Appointment class by attempting to create an appointment with a past date, which correctly triggered an exception. These techniques gave me confidence that the code was resilient under both normal and extreme conditions.

Other techniques that weren’t part of this project include integration testing, system testing, and regression testing. Integration testing would focus on verifying how different components work together, while system testing would assess the entire application as a whole. Regression testing would be valuable for ensuring that new code changes don’t break existing functionality. Although these weren’t required here, I can see how they would be essential in larger, collaborative projects.

### Mindset and Professional Growth

This project taught me to approach testing with patience and curiosity. Writing the code was only half the job; the real work came in making sure it could handle anything thrown at it. I learned to test my assumptions and treat every unexpected outcome as an opportunity to improve. It made me realize that testing isn’t just about proving code works, it’s about proving it won’t fail.

Working on my own code also made me think more critically about bias. It’s easy to assume your own logic is flawless because you understand it, but that’s exactly where blind spots form. To limit bias, I followed the requirements list directly instead of relying on memory or assumptions. I also reran tests after making even small code changes to avoid missing regressions. This mindset helped me stay focused on quality rather than speed.

Finally, I learned that discipline in software testing pays off. Cutting corners might make things faster in the short term, but it usually leads to technical debt later. Going forward, I plan to continue writing clear, maintainable tests and using coverage tools to keep myself accountable. Testing may not always feel exciting, but it’s what separates a functioning program from a reliable one.

## Conclusion

Completing this project gave me a better understanding of how testing fits into the overall software development life cycle. I learned how to approach coding with a tester’s mindset, plan for edge cases, and structure my work to make debugging easier. Through building and testing the Contact, Task, and Appointment services, I developed a new appreciation for precision and consistency in software development. Going forward, I’ll carry these lessons with me into future projects, especially the importance of testing early, testing often, and never assuming that something works just because it looks like it should.