**Todd Johnson**

**06/22/2025**

**CS 320 Project Two**

**1. Summary**

**A. Alignment of software requirements**

My unit testing approach closely aligned with the functional requirements provided for each class—Contact, Task, and Appointment. For instance, the ContactService required that each contact have a unique ID and valid non-null fields. I created tests such as testAddContact\_validInput\_contactAdded() to verify these constraints. Similarly, for the TaskService, I made sure that the title and description adhered to length and null constraints through tests like testUpdateTask\_withValidChanges\_taskUpdated(). The AppointmentService included date validation to reject past dates, which I verified using testAddAppointment\_withPastDate\_appointmentRejected(). These tests were derived directly from the requirements and enforced through assertions, ensuring full traceability between the code and the documented constraints. However I did miss several key changes that were made for requirements, I was able to correct with feed back

**B. Effective tests**

I tried to stick with test-driven development (TDD), writing the tests before the logic. It helped me stay focused and build a solid structure, though it took some getting used to at first. All tests were run individually to ensure proper isolation. Coverage tools confirmed that my code had nearly 100% coverage, with all branches and paths tested at least once. This validates that the JUnit tests were effective in covering edge cases, invalid input scenarios, and valid use paths.

**c. Technically sound code**

To ensure my tests were technically sound, I validated critical constraints using robust assertions and appropriate test isolation. For example, in TaskServiceTest, I wrote:

A computer screen shot of words

AI-generated content may be incorrect.

This line ensures invalid input handling is enforced. In AppointmentServiceTest, the date validation logic was exercised with:

A screenshot of a computer program

AI-generated content may be incorrect.

These code snippets confirm that the logic is reliable and meets professional software testing standards.

**D. Efficient code**

Efficiency was prioritized using constants and utility methods to reduce code duplication. For example, frequently used objects like:



were declared at the class level, making tests cleaner and more readable. Also, update methods in ContactService were consolidated into a single method to avoid redundancy and reduce code size. These efforts made the test code both efficient and maintainable.

**2. Reflection**

**A. Testing techniques**

**Techniques employed**  
I employed black box testing and unit testing techniques using JUnit 5. Black box testing allowed me to focus on input-output behavior without needing to understand internal implementation, which was good for validating user-facing constraints. I also used exception testing to verify that constraints like null values, length violations, and invalid dates were enforced.

**Techniques not used**  
I did not use integration testing or boundary value analysis in a formalized way. Integration testing wasn't applicable because each service (Contact, Task, Appointment) was designed to operate independently

**Practical uses and Implications**  
Blackbox testing is especially useful when the tester is different from the developer or when working with API contracts. Unit testing, particularly with tdd, supports fast feedback loops and early bug detection, reducing technical debt. Exception testing is vital for enforcing fail-fast principles in safety-critical applications. Lack of integration testing, however, would be a concern in microservices or layered architecture systems where component interaction is crucial.

**b. Mindset**

**Caution**  
I tried to be careful and double-check the requirements before starting each test, but I still missed some things that were posted in announcements, which I learned from the hard way.

**Bias**  
To limit personal bias, I used a break the code mindset, designing test cases that challenged the success path. I also separated test logic from implementation logic and imagined myself as a different developer when reviewing the code. This mental shift helped me uncover edge cases like deleting nonexistent IDs or updating with null fields.

**Discipline**  
Discipline was crucial in maintaining high test quality. I learned to avoid shortcuts like skipping assertion checks or lumping multiple assertions into a single test. Each test followed the Arrange Act Assert pattern and used clear naming conventions by the end of the project with aid from the teacher

0 ‘s’ corrections. Using testDeleteAppointment\_withInvalidId\_shouldThrowException(). As a future developer, I plan to manage technical debt by using code reviews, enforcing unit test requirements, and treating test coverage as a quality gate.

**Conclusion**

In completing Project One, I applied a range of effective software testing strategies using JUnit. My experience affirmed the importance of aligning tests to software requirements, focusing on quality through structured patterns, and reflecting on my own development mindset. The testing strategies and habits I formed in this project will contribute to code quality and reliability in the future. Though I still need to figure out how to write all these code references in Microsoft word without making the spell check make everything look like a Xmas tree.