CS-320

Eddy Kwon

Jun 20, 2025

CS 320 Project 2 Review

Throughout the development of the contact, task, and appointment features in Project One, I used a white-box unit testing approach for each module. I focused on testing the internal logic of my classes by writing detailed JUnit test cases. My goal was to ensure that every method behaved correctly based on the requirements. For example, the appointment feature required that the ID not exceed ten characters and must not be null. I handled this by writing conditional logic in the constructor and setter methods to throw exceptions if these rules were broken. I applied this same logic pattern to the contact and task features as well. In the Contact class, I used IllegalArgumentException checks for null or over-length inputs, and I verified phone number patterns using regular expressions. I stayed close to the requirements provided and made sure the functionality worked as expected before moving on to the test classes.

My JUnit tests were mostly successful. I now realize I initially missed the importance of writing tests for failure scenarios, not just for correct cases. I would estimate that my JUnit coverage was around 50% at first, but after adding test cases to validate exceptions, such as duplicate IDs or invalid deletions, the quality significantly improved. The JUnit test results showed all green for the task and task service classes, indicating they passed. For example, I tested scenarios like deleting a task that didn’t exist, and it correctly threw an exception.

**This screenshot shows how validation was handled directly in the class logic.**



To keep my code technically sound, I ensured that each function did what it was meant to do and nothing more. My use of assert methods like assertThrows, assertEquals, and assertNull confirmed whether my functions returned the expected results. One example is from my task service test class, where I checked that attempting to delete an invalid task ID threw the right error. In terms of efficiency, I reused a @BeforeEach method to initialize the same setup across multiple tests. This saved time and ensured consistency.

**This screenshot highlights that all tests passed and shows coverage of both valid and invalid conditions.**

A screenshot of a computer

AI-generated content may be incorrect.

I used unit testing and exception-based validation as my primary software testing techniques. Unit testing focuses on testing individual pieces of logic like methods and constructors. Except testing checks for negative inputs and system robustness. I did not use integration testing in this project, which would normally test how multiple parts work together, or system testing, which verifies end-to-end functionality. While those are more advanced, they would be valuable in future stages of development.

Throughout the project, I adopted a cautious mindset. I was very aware of the connections between the service and model classes. For example, the TaskService depended entirely on correctly written logic in the Task class. If that class had faulty validation, my service would break or pass the wrong values. I took time to test both regular and edge cases. One example is making sure that phone numbers in the Contact class were exactly 10 digits, not more or less.

I tried to avoid bias by stepping back from my code and asking, “What would break this?” That helped me write better tests and think from the user's perspective. If I were the only one testing my own code, I might overlook certain edge cases. Testing for errors like null values and overly long strings helped remove that bias. For instance, I made sure that setting a task name to more than 20 characters would correctly throw an exception.

Being disciplined about quality matters in this profession. Taking shortcuts during testing might not hurt in a classroom assignment, but in real-life software, even one overlooked bug can lead to serious issues. I want to avoid technical debt by consistently writing full unit tests and documenting every function. Going forward, I plan to include exception handling, comment blocks, and full testing coverage as part of my workflow. That way, even if the code is small, it will be reliable and scalable.