Corey Gaspar

9/24/2025

**Module 4 Journal**

**Testing Approach**

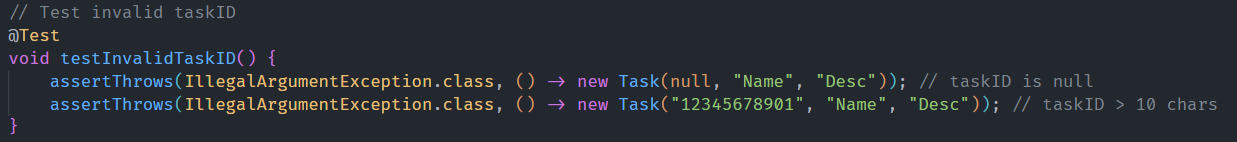
I would say that my testing approach was closely aligned with the software requirements. I ensured that each task had an ID (max 10 characters), a name (max 20 characters), and a description (max 50 characters). In TaskTest, I began by testing a valid task using assertEquals and made sure that all invalid inputs, such as null values and overly long strings, were throwing exceptions. Within TaskServiceTest, I confirmed that creating duplicate tasks, updating a task with a non-existent ID, or deleting a non-existent task would throw an exception. The same goes for the contact service, where I verified that each contact had a valid ID, first and last name, phone number, and address. I also confirmed that duplicate contacts, updating non-existent contacts, or deleting non-existent contacts triggered exceptions. By testing valid and invalid scenarios, I was able to verify that my code met the requirements.

**Quality of JUnit Tests**

The quality of my JUnit tests for both the contact and task service was high because they provided coverage of all functional requirements and edge cases. For the contact service, I tested adding, updating, and deleting contacts, including the use of invalid inputs such as long names, invalid phone numbers, or non-existent IDs. Same goes for the task service, where I tested creating, deleting, and updating tasks, both valid and invalid. During testing, I made sure to test all public methods and edge cases to ensure that the majority of the code was executed. These tests verified that both services behaved as intended.

**Technically Sound Code**

I ensured my code was technically sound by validating all functional requirements, handling as many edge cases as possible, and ensuring that all invalid inputs were rejected. A specific example of this would be in my TaskTest file, where I verified that invalid IDs, names, and descriptions were properly rejected.

The code in testInvalidTaskID will throw an exception if the task ID is null or greater than 10 characters.

A colorful text on a black background

AI-generated content may be incorrect.The code in testInvalidName will throw an exception if there is no name entered or if the name is longer than 20 characters.

A computer code on a black background

AI-generated content may be incorrect.

Finally, testInvalidDescription will throw exceptions if the description is null or if it’s longer than 50 characters. I also made sure to test my setters to ensure that they correctly updated the values for the task name and description. My code for the task service closely aligns with the code for the contact service, since they function similarly to each other, but with different data.

A screen shot of a computer code

AI-generated content may be incorrect.

**Efficient Code**

I ensured that my code was efficient by designing my services to perform its necessary functions and avoid redundant processing. One way I did this was in the TaskServiceTest file, where I used the task ID as a key, which was useful in preventing duplicates. The key is used similarly when adding and deleting tasks. Instead of repeatedly searching through the list, it will first check for the existence of the ID, and if it exists, it will perform the requested action. This works similarly to the contact service. **(Code snippet on next page).**

A screen shot of a computer program

AI-generated content may be incorrect.