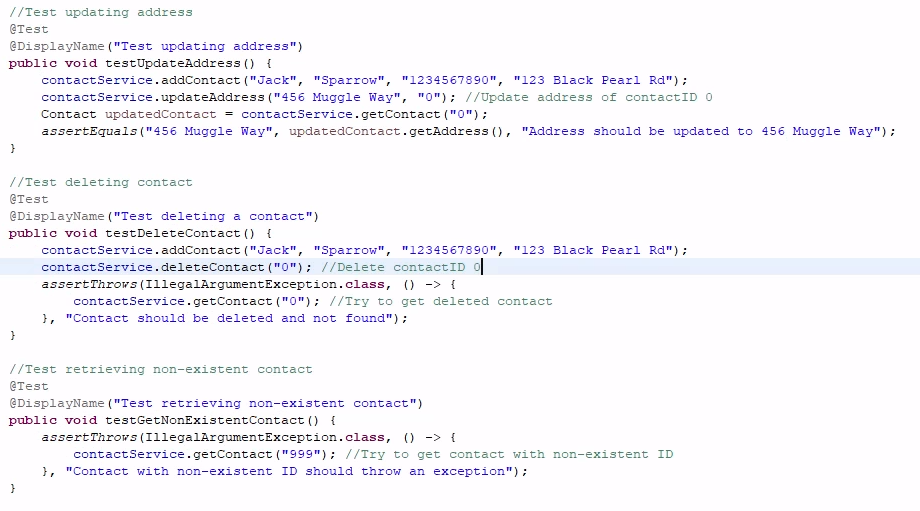
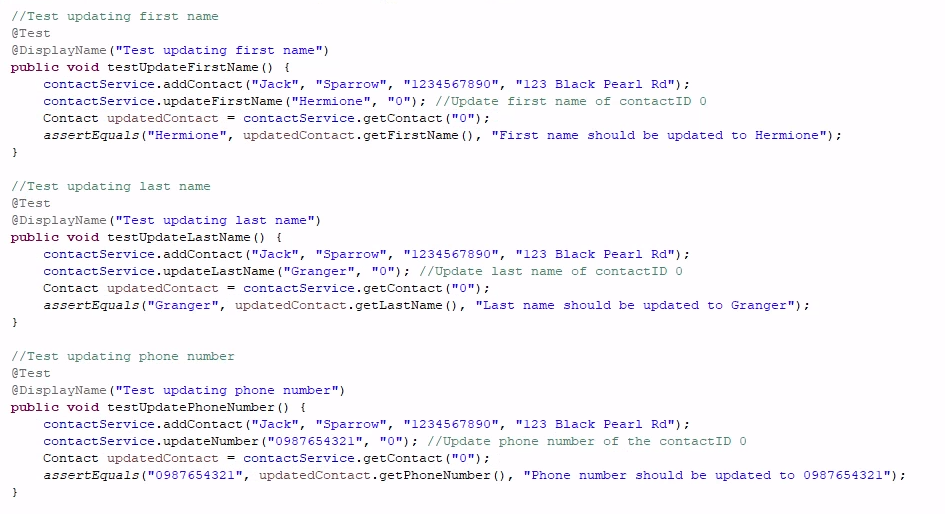
Jordan Mitchell

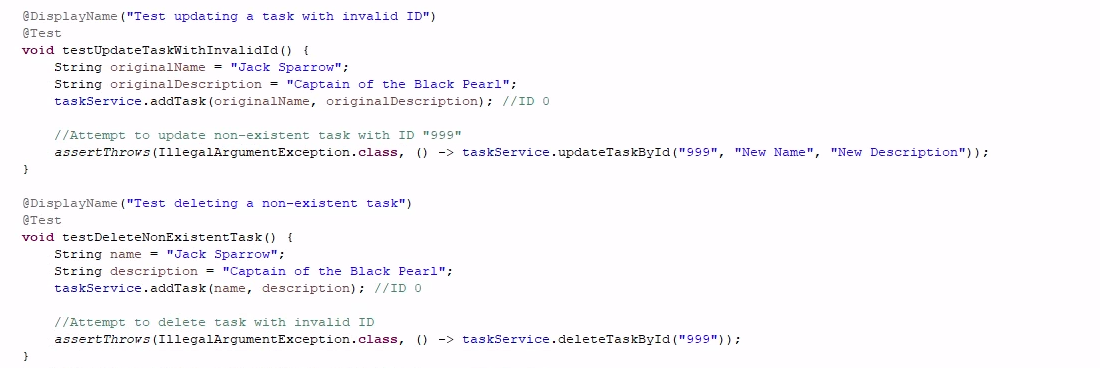
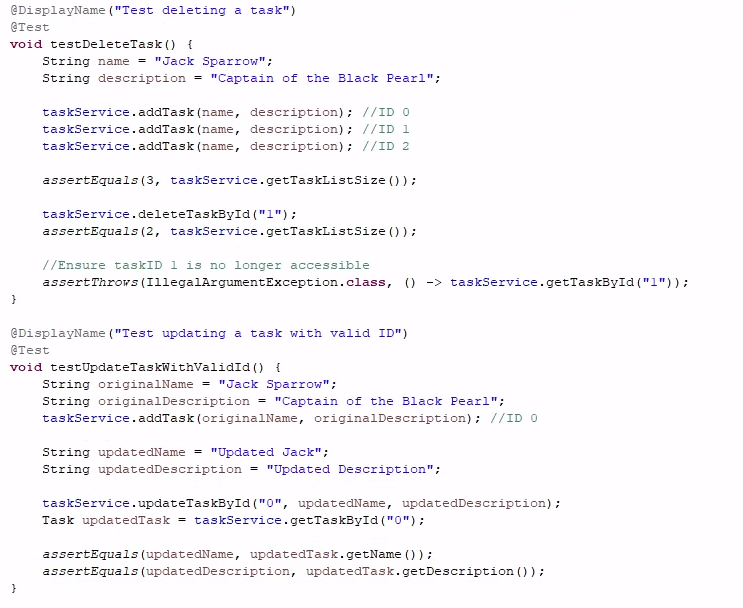
9/28/2024

Professor Farley

Journal: Unit Testing Approach and Writing JUnit Tests

The Contact Service and Task Service assignments from Module Three and Module Four were focused on creating systems to manage contacts and tasks, ensuring that each entry adhered to validation rules. These services had functionality to add, update, and delete records, and used JUnit tests to verify their correctness. My approach to unit testing aimed to cover various input scenarios, ensuring the services met all requirements. By writing comprehensive JUnit tests, I was able to confirm the reliability of the code and its fulfillment of project specifications. I will examine how the testing aligned with the requirements, ensured coverage, and maintained the technical integrity and efficiency of the code.

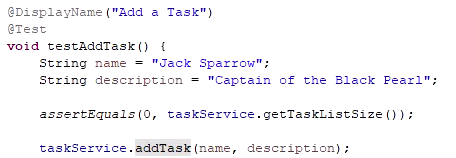
For the Contact Service, my testing approach was aligned with the set requirements. I created unit tests that specifically checked for the addition, updating, and deletion of contacts. For example, I ensured that contacts could be added with valid information and verified that the details were correct after addition. Additionally, the tests covered essential update operations, like changing the first name, last name, phone number, and address, as specified in the requirements. Finally, I wrote a test to verify that a contact could not be retrieved once it was deleted. These tests ensured that all core functionality of the Contact Service operated as expected.

For the Task Service, my testing approach also effectively met the software requirements. I wrote comprehensive tests to ensure tasks could be added, updated, and deleted correctly, with validation checks for task ID, name, and description length as outlined in the rubric. For instance, I tested that adding a task correctly generates a unique ID, and verified that tasks were properly stored and retrievable by their IDs. Additionally, I implemented tests to validate that task names and descriptions were updated only when valid inputs were provided, ensuring I adhered to the requirement of character limits for each field. I also included negative tests to check for edge cases, such as attempts to update or delete tasks that do not exist. These tests ensured that invalid operations were handled properly, throwing the appropriate exceptions. By covering both positive and negative scenarios, I ensured that the Task Service met functional requirements and handled a range of possible input conditions, solidifying the strength of the code.  For both the Contact Service and Task Service, I ensured that my JUnit tests followed best practices and covered various scenarios, such as valid and invalid inputs. In the Contact Service tests, I verified that the contact was added successfully and that updates to fields like the first name, last name, phone number, and address worked as expected. The tests also confirmed that the system correctly handled edge cases like null inputs and overly long strings. Similarly, for the Task Service, I tested the creation of tasks with valid and invalid data, ensuring that invalid tasks (like those with excessively long descriptions or null values) were handled with relevant exceptions. Additionally, the test coverage extended to updating and deleting tasks, confirming that the system could manage tasks efficiently while ensuring data integrity.

Throughout my work, I made sure to test both the core functionalities and edge cases, ensuring that my code could handle real-world scenarios effectively. For example, in ContactServiceTest.java, I verified that new contacts could be added with accurate details by calling the addContact method, as seen here:



Similarly, in TaskServiceTest, I used the addTask method to create tasks:



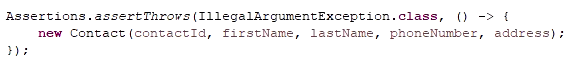
This demonstrates the ability to generate new objects while maintaining unique IDs. Additionally, I included update methods to ensure information could be modified when necessary. For example, in ContactServiceTest, I tested updating a contact's first name:

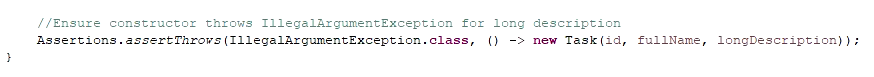


Here is an example from TaskServiceTest:



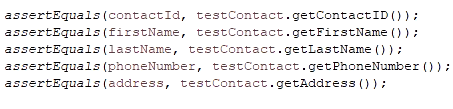
This confirmed that my code handled updates as expected. Another key aspect of my testing was validating input and managing edge cases. In ContactTest, I ensured that invalid inputs, such as phone numbers that were too short, triggered exceptions:



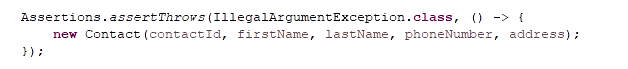
I performed a similar process for TaskTest when the length of a description exceeded the length limitation:  
  
These tests helped me confirm the effectiveness of the input validation in both services. Lastly, I confirmed the delete functionality worked by removing contacts and tasks:

and 

To ensure the soundness of my code, I structured each of my unit tests to cover a range of valid and invalid inputs, using assertions to validate expected outcomes. In ContactTest, the test method testGoodConstructor() validates that the constructor assigns the correct values for each field in a Contact object. The test checks for each field like this:



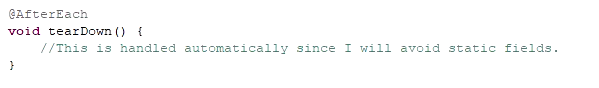
This ensures that the constructor correctly initializes the object with valid inputs. Additionally, I included tests for edge cases, such as invalid input lengths. For example, in testBadConstructor(), I check for an invalid phone number length and ensure that the program responds appropriately by throwing an IllegalArgumentException:



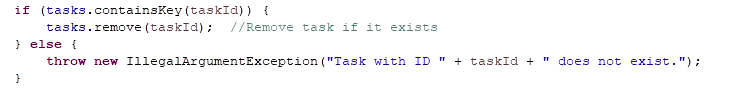
This shows the technical sturdiness of my code by handling improper inputs correctly, maintaining the integrity of the contact creation process. The same principle is applied in TaskTest.java where methods like testInvalidTaskIDConstructor() and testInvalidLongDescriptionConstructor() enforce the constraints on task ID and description lengths, ensuring the program remains within constraints. Each test is structured to handle both positive and negative scenarios, covering both technical and functional requirements effectively.

To ensure that my code was efficient, I used clear and concise logic in the test cases to quickly validate the functionality without unnecessary clutter, reducing redundancy and maximizing reusability. For example, in TaskServiceTest, I used the setUp() method with the @BeforeEach annotation to initialize the TaskService object before each test. This avoids duplicating code across the test cases and promotes code reusability:

Additionally, in both ContactServiceTest and TaskServiceTest, I made sure to avoid static fields to make sure that my test data was cleared between tests. This is so that each test runs independently and does not interfere with others, improving the overall efficiency:



Efficient error handling was also a priority for me. In TaskService, I used the method deleteTaskById() to check for the existence of a task before attempting to delete it, preventing unnecessary operations and improving performance by avoiding redundant code execution:

This combination of systematic testing and efficient error handling ensures that both services perform efficiently while maintaining their functionality.