Karina Aronov

Professor Cross

CS320

October 20, 2024

**7-2 Project Two Submission**

**Summary**

I carefully considered the requirements for each of the three features before creating my unit testing strategy. I made sure that every test I wrote addressed the necessary functions outlined in the project guidelines. For the Contact Service, I verified that the contact ID couldn’t be null or changed, which was a key requirement. I followed the same pattern for the Task Service, ensuring that tasks could be added, updated, and deleted as required. A good example of this is the testUpdateTaskName() in the TaskServiceTest, which checked that the task name could be updated, directly aligning with the software requirements. By writing tests specifically designed for these functionalities, I made sure that my approach was consistent with what the project asked for.

I focused on writing thorough tests with a high coverage percentage for my JUnit tests to make sure that the majority of the code was correctly tested. tests to consider potential errors like incorrect input lengths or handling null values. For example, in the TaskServiceTest, the testInvalidTaskId() method ensured that invalid task IDs were handled correctly, whereas the testAddTask() method verified the successful addition of a task. This reassured me that the tests were valid and would find any flaws quickly. The quality of my tests was further improved by the inclusion of edge cases, such in ContactServiceTest, where I tested for things like null or extremely long names.

It was interesting to write the JUnit tests since it made me consider all the possible ways that each feature may malfunction or break. In order to ensure the structural quality of my code, I verified expected outcomes using explicit assertions. In the TaskServiceTest, for instance, the line assertEquals("Updated Name", service.getTask(taskId).getName()) verified that the task name was updated appropriately, demonstrating that the method achieved its intended purpose. Additionally, the ContactServiceTest verified that contacts were being added accurately with the line assertNotNull(service.getContact(contactId)). Carefully validating the code helped ensure the code was functioning as needed.

I also tried to keep the tests straightforward and uncomplicated in order to optimize my code. For example, I tested the update functionality directly in the TaskServiceTest by using service.updateTaskName(taskId, "New Name"), eliminating the need for extra steps. This made the code readable and clear. I also tried to avoid duplicating code by not utilizing functions like service.addTask() in different tests. Because this method followed the DRY (Don't Repeat Yourself) philosophy, the tests were easier to maintain and more effective.

**Reflection**

For this project, I mostly used unit testing to confirm that every single system component functioned as intended when left alone. Testing separate, small-scale code segments, such as particular functions or procedures, is the main purpose of unit testing. I found it easy to find and repair problems by dividing the code into small, manageable portions and testing each one separately. This approach was especially useful for the Contact, Task, and Appointment services, where each service had distinct methods for adding, updating, or deleting objects, and unit testing allowed me to confirm that these methods worked independently of one another.

There are other testing methods that I didn’t use in this project but could have been helpful in a more complex scenario. Integration testing, for instance, would have been helpful in determining how various services worked together, ensuring that they were in sync and did not conflict with one another. System testing, which tests the entire program, might have been used to confirm that, in the real-world environment, every component of the system functioned as intended. Finally, acceptance testing would be necessary to guarantee that the application met the user's needs and expectations and operated as needed in a real-world environment (GeeksforGeeks, 2024a).

There is a place for each of these testing methods for different kinds of software projects. Unit testing is ideal for smaller, more modular systems, such as the services I worked on, as it focuses on checking that each tiny component functions independently (GeeksforGeeks, 2024b). However, system and integration testing become important in larger, more integrated systems to guarantee that all components function as a whole. When providing a finished product to a client, acceptance testing is essential to make sure the program functions exactly as the client expects.

**Conclusion**

In conclusion, I learned a lot about bringing unit testing to useful functionalities like Contact, Task, and Appointment Services through this project. I was able to confirm that every feature performed as planned and met with the client's requirements by using JUnit tests. For this project, I concentrated on unit testing, but I also recognize that system and integration testing are important methods of testing, especially for larger applications. Moving forward, I’ll stay committed to being thorough and careful when testing code, making sure to maintain a high standard of quality throughout the process.

**Resources:**

GeeksforGeeks. (2024a, September 26). *Types of software testing*. GeeksforGeeks. https://www.geeksforgeeks.org/types-software-testing/#other-types-of-testing

GeeksforGeeks. (2024b, October 7). *Unit testing software testing*. GeeksforGeeks. https://www.geeksforgeeks.org/unit-testing-software-testing/