**Wasan Alabed**

**CS-320**

**December 9, 2024**

**CS 320 Project Two**

**Summary**

The testing method checks the software requirements by verifying the main functions of the Task Service class. It confirms that the Singleton pattern is used correctly, ensuring only one instance exists. The tests for adding, deleting, and updating tasks show that these actions can be performed in the database, meeting the basic needs. While the main requirements are addressed, adding tests for edge cases would further improve the software's performance.

A screenshot of a computer program

Description automatically generated

The tests in my Contact Service Test class look at the key functions of managing contacts: adding, deleting, and updating. I confirmed that contacts are added correctly and that duplicates are not allowed. I also checked that I could delete existing and nonexistent contacts and that updates are applied accurately.

A computer screen shot of a computer code

Description automatically generated

Unit tests for appointments Check how well the app creates and manages appointment objects. These tests ensure that appointment IDs are unique, dates are in the future, and descriptions are not empty. They also test invalid input, like past dates or duplicate IDs, to ensure the app properly manages errors.

So, the JUnit tests for the `contact service` and `task service` cover important functions like adding, deleting, and updating tasks and contacts. They also handle some edge cases, such as preventing duplicates and ensuring that tasks or contacts cannot be updated or deleted if they do not exist.

While these tests cover many key scenarios, we can improve them by adding more edge cases. For example, we should test invalid inputs like empty fields or incorrect phone numbers and check the limits on field lengths. Additionally, looking at how exceptions are handled would improve coverage.

I am confident that the `contact service test` works well because it tests all the main functions, like adding, deleting, and updating contacts. The tests also check for special cases, such as preventing duplicate contacts and ensuring attempts to update non-existent contacts fail.

In the task service test, there may be issues with how the assertions are written, which could lead to test failures. I need to ensure that the database updates correctly after each task operation, whether it involves adding, deleting, or updating tasks. If the tasks do not update as expected, I will review the logic in the Task Service and examine the error messages from JUnit to resolve any problems. After a couple of checks, I fixed the issues and confirmed that the Task Service test is now functioning correctly.

I focused on writing simple tests to ensure my code was efficient. In the `task service test`, I used straightforward assertions. By testing the main actions directly and using minimal code, I kept the tests clear and focused on confirming the functionality efficiently.

**Reflection:**

Throughout the project, I mainly used unit, integration, and exception-handling testing to ensure each component worked properly and interacted well with others. I performed unit testing with JUnit to check functions for adding, updating, and deleting appointments, tasks, and contacts. Integration testing verified that services and in-memory data stores worked together smoothly, ensuring tasks and contacts were managed correctly. Exception handling testing confirmed the application could handle invalid inputs or duplicate entries by showing the right error messages.

Several software testing methods were not utilized during the project milestones. **System testing**, which evaluates the entire application in a production-like environment to ensure all components work together, was not conducted because the focus was on isolated features rather than complete workflows. **Regression testing**, involving re-running existing tests to verify that new changes do not introduce issues, was skipped since the milestones primarily introduced new features without revisiting earlier implementations. Similarly, **performance testing**, which assesses the application's speed, scalability, and resource usage under different loads, was not performed as the milestones prioritized functionality over performance metrics. Lastly, **user acceptance testing (UAT)**, where real users validate that the application meets their needs and is ready for deployment, was omitted because the milestones concentrated on development and functional accuracy rather than user feedback. The milestones emphasized building and validating new features rather than comprehensive testing.

I worked on this project with care and attention to detail. My goal was to test all functions thoroughly and consider edge cases. As a software tester, it was important to be cautious to avoid missing potential errors that could disrupt the application. I tested a variety of inputs—both valid and invalid—to make sure the application worked correctly in different situations.

For example, when testing the Contacts feature, I checked how the system handled invalid input, like a contact ID that was too long or a missing first name. This helped ensure that the system triggered the right error messages. Taking these precautions allowed me to find problems that could cause errors or unexpected behavior in a live environment. I also tested valid inputs to ensure the application worked as expected, reducing the chance of hidden bugs.

I used an objective testing method to reduce bias in my code review, focusing on the application's behavior rather than my personal opinions. I created test cases that included challenging scenarios to identify potential problems.

As a software engineer, it's crucial to write and test code carefully to ensure quality. Taking shortcuts can lead to technical debt, which builds up and causes problems in maintenance and performance.

In this project, I focused on writing tests for various scenarios, including edge cases. I ensured to test complex features, like the interactions between `Contact Service` and `Task Service, ` covering adding, updating, and deleting tasks, as well as handling duplicates. Skipping these tests could result in bugs that would worsen as the application grows.

To reduce technical debt, I will follow good testing practices. This includes writing unit tests and doing thorough code reviews. I will also simplify complex code and follow the DRY (Don't Repeat Yourself) principle. This approach will help prevent unnecessary technical debt and make future development easier.

References:

GeeksforGeeks. (2024b, June 13). *Regression testing - software engineering*. <https://www.geeksforgeeks.org/software-engineering-regression-testing/>

GeeksforGeeks. (2024a, February 22). *Don’t repeat yourself(dry) in software development*. <https://www.geeksforgeeks.org/dont-repeat-yourselfdry-in-software-development/>