Project Two

For each feature—Contact, Task, and Appointment services—I ensured my unit testing approach directly aligned with the software requirements by developing test cases based on user stories and acceptance criteria. For example, the Contact service required validation for non-null first names and phone numbers limited to 10 digits. I created unit tests to confirm the system threw exceptions when these constraints were violated, demonstrating adherence to the defined requirements. As well as verifying that many of the different objects that are created are going to be unique in their nature.

A screen shot of a computer code

AI-generated content may be incorrect.

This is what the JUnit tests looked like for adding a contact

A screen shot of a computer

AI-generated content may be incorrect.

This verifies that only 1 contact with a specific unique ID can be had.

I measured the effectiveness of my JUnit tests with the test coverage analysis feature of IntelliJ. I found this to be incredibly useful in making sure that I was getting test coverage for every method of each class being tested overall. I had 100% coverage of all methods across the 6 different classes created, with line coverage being on the low end with 88% coverage, and the high end being 100% coverage. This indicates that a majority of the code paths were tested , including edge cases and exception handling. Assertions such as assertThrows() and assertEquals() confirmed that the code behaved as expected, and maintained the expected functionality.

A screenshot of a program

AI-generated content may be incorrect.

I ensured that my code was technically sound by creating modular test cases that focused on specific behavior. This code inside of task service was made to reduce redundancy in the overall code, and was adapted to fit the other service classes as well. A computer screen shot of a program code

AI-generated content may be incorrect.

This is a modular test case that made sure that the code would show that the task should not exist if it was not created.

A computer screen shot of a computer

AI-generated content may be incorrect.

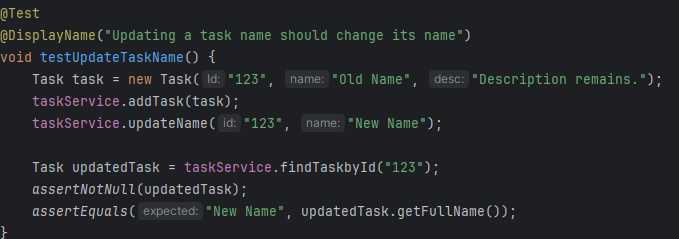
Before each test was ran we made the testing efficient by developing a void setup() that allows us to create the list of objects before passing any data, which reduces the amount of redundant code that was written. Overall this allows us to reuse setup logic and avoids excess testing.

A screen shot of a computer code

AI-generated content may be incorrect.

In my code I used both black box testing, and boundary value analysis. The Black box testing allowed me to focus on the input/output behavior which is perfect for validating our business logic. Boundary Value analysis is used for fields like the phone number length, and ensuring that other fields aren’t going over the specified character limit. I did use White-box testing, which requires internal knowledge of the logic paths, and is better suited to analyze systems, especially when running unit tests. I also did not use Equivalence partitioning, which divide input data into valid and invalid groups to minimize the number of test cases.

Black box testing is suitable for user facing features to ensure compliance with our overall requirements. Boundary Value analysis is perfect for validation logic, like input length or format. White box testing is best for testing that every statement is covered, using statement coverage.

I adopted a cautious mindset by thoroughly testing typical and edge cases. I had considered that when calling different functions to update certain variables between the three sets of classes, that theirs a possibility of duplicate IDs or null values, or variables containing the wrong types of information. 

This ensures that everything is functioning appropriately, but also checks for different things that might not be very obvious bugs in the code. To limit bias overall, I decided to approach testing as if I was a new user. My tests contained information that would intentionally break the functions I had created in order to observe that the error catching was valid. This helps ensure that our testing is reliable in its nature to ensure that all the code functions as intended. To remain disciplined in my commitment to quality as a software engineering professional, I kept consistent naming conventions (“updateTask”, “addContact”,etc.), added descriptive comments that explained logic, and ensured that the code was thought out instead of looking as if it was rushed. I avoided technical debt by writing clean and reusable tests using methods to setup @BeforeEach, and ensured that constantly called fucntions were set up properly to be reusable. Overall you should never cut corners when coding, otherwise the code is going to break in an unexpected way, and that will upset the client it was prepared for.

References

* Bose, S. (2023, May 25). *Software Testing Techniques: Explained with Examples*. BrowserStack. https://www.browserstack.com/guide/software-testing-techniques