My approach to creating the software built for testing was entirely aligned with the requirements of the customer’s request. The requirements asked for objects that could be linked to a unique ID and updated. The application also asked for services that could add, update, and delete contact objects. The objects had to be a certain length of characters and could not be null. These requirements specifically drove my implementation of the software I designed.

The first step in ensuring full coverage was to use different kinds of tests. Part of the code used if-else decision branches to ensure the objects met the character length requirement and were not null. The tests required for this piece of code ensured that each possible branch (if-else decision) was tested, known as branch coverage. To ensure 100% coverage, I needed to test both the If and Else branches.

Before this class, I had no experience writing JUnit tests. Up until then, the only testing technique I employed was manual testing. I used what is known as black box testing. Essentially, I would provide the input for the application and test it manually. If it returned the expected output, I would proceed with another set of inputs. If the input failed during the testing, I would go back and debug the code.

To ensure that my code was technically sound, I focused on validating the functionality of each method in the Contact class. The tests check both the valid and invalid inputs and the expected behaviors for the attributes of the Contact object. Here is a specific example of how I accomplished this. The first test, testValidContact(), ensures that a Contact object is created with correct values. The test checks that the attributes match the expected values:

assertEquals("C12345678", validContact.getContactID()); // Test valid contact ID

To ensure the code was efficient, I focused on the logic within the Contact class. It was simple and direct, and I made sure that the tests were focused on important use cases rather than testing every possible scenario. For example, my test below ensures that a valid phone number is correctly updated without overcomplicating the test by testing multiple formats or unnecessary variations:

validContact.setPhone("9876543210"); // Set new phone number

assertEquals("9876543210", validContact.getPhone()); // Check if the phone number changed correctly

In my tests, I used functional testing, boundary testing, input validation testing, regression testing, edge case testing, and negative testing to achieve the most effective coverage percentage. One example from the tests I used is negative testing. The methods testSetInvalidFirstName(), testSetInvalidPhone(), and testSetInvalidAddress() check that the system rejects invalid inputs. If the user inputs first names with more than 10 characters, short phone numbers, or long addresses, the negative tests ensure that the invalid input does not affect the valid data. The practical use of negative testing is common in ensuring users cannot input invalid responses.

I did not use any integration testing in my development. Integration testing involves testing how different components of the system work together. In this case, it would verify how the Contact class interacts with other classes, modules, or services in the system. I had a Contact class and a ContactService class that depended on one another. However, I did not run any tests to verify that their interaction was functional. A practical use would be if the code I was using employed inheritance, then I would want to test that the classes interact with each other properly.

The mindset I walked away with is that I recognized testing isn’t just about finding faults but also about verifying that the software behaves in a robust, predictable manner. I had to employ critical thinking and caution when implementing the JUnit tests for my code. In the ContactServiceTest, I tested adding contacts with invalid IDs, ensuring that the system rejects invalid data. These scenarios reflect caution because handling edge cases ensures that the application behaves correctly even when the input deviates from the expected responses.

Since I was testing the code I wrote, biases such as blind spots and overconfidence were bound to happen. A blind spot can emerge if I assumed that my code worked as intended without thoroughly questioning whether it handled every edge case. This can lead to a lack of comprehensive testing. This is also where overconfidence comes into play. Since I wrote the code, I didn’t want to see flaws in it. This bias can prevent me from thoroughly testing all possible situations, especially those that I didn't explicitly consider.

Not cutting corners can be difficult in testing because it’s easy to overlook simple code, thinking it isn’t necessary to test an if-else statement or something similarly straightforward. In unit testing, being disciplined meant testing not only the expected outputs but also thoroughly validating edge cases. This ensures that the application can handle all user inputs correctly, even under unusual conditions.

Technical debt refers to the long-term cost incurred when shortcuts are taken during the development process. Essentially, if you skip over the “easy” tests, then there’s a good chance that as the product scales, errors will become more likely. I can avoid this by being a more disciplined developer. I will make sure to keep testing all areas of the code, even the ones that seem simple or obvious, because those are often the ones that get overlooked and cause issues later on.

**RESOURCES**

Bechtold, S., Brannen, S., Link, J., Merdes, M., Philipp, M., de Rancourt, J., & Stein, C. (n.d.). *JUnit 5 User Guide*. Junit.org. https://junit.org/junit5/docs/current/user-guide/

*Software Testing | Manual Testing*. (2019, May 7). GeeksforGeeks. <https://www.geeksforgeeks.org/software-testing-manual-testing/>

*Ten cognitive biases to look out for as a developer*. (2019). Codecentric AG. https://www.codecentric.de/knowledge-hub/blog/ten-cognitive-biases-to-look-out-for-as-a-developer

Testlio. (2025, February 14). *The Ultimate Guide to Negative Testing*. Testlio. <https://testlio.com/blog/negative-testing-in-software-testing/>

*What is Technical Debt in Software Development and How to Manage It?* (2020, December 10). GeeksforGeeks. https://www.geeksforgeeks.org/what-is-technical-debt-in-software-development-and-how-to-manage-it/

‌

‌

‌