**CS320 Module Seven Project 2**

Jermaine Wiggins

Southern New Hampshire University

CS320: Software Test, Automation

Prof. Joseph Martinez

April 20, 2025

**CS320 Module Seven Project 2**

**Summary**

My testing approach aligned with the Grand Strands system requirements by using the six classes requirements as my test cases. For example, the contact class required the ID, first name and last name to be no longer than 10 characters and not null. When building the class, if a requirement wasn’t met, an exception would be thrown. So, when I created the test, I made sure it checked that an exception was thrown for invalid input, and that valid input matched the original input and was accepted. Therefore, if each test passes then I know my code covers all the requirements. For project one all my tests passed, and I believe the quality and coverage percentage of my JUnit tests are high. A good majority if not all my code’s logic and branches were tested through assertions that were directly tied to the expected outcomes. Additionally, I installed a JaCoCo plugin, and the results were 97% for missed instructions and 90% branch coverage. With further investigation this was due to the added helper methods and main not being tested.

To ensure my code was technically sound and efficient, I focused on writing effective JUnit tests that verified expected behavior based on requirements, that handled unexpected inputs gracefully and followed best practices like clear naming conventions and meaningful comments. For example, the deleteContactTest clearly indicates its purpose and checks both successful deletion and that retrieving a deleted contact throws the expected IllegalArgumentException. To improve efficiency and reduce redundancy, I used the @BeforeEach annotation to initialize shared objects like the contact and contact service, rather than recreating them in every test. I also refactored repeated logic, such as checking if a contact exists, into a private helper method, which streamlined the code and made it easier to maintain and understand. Examples below.

A screen shot of a computer

AI-generated content may be incorrect.

A computer screen shot of a program code

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

**Reflection**

The software techniques used to complete this project were unit testing, integration testing, and white box testing. Unit testing tests individual components to ensure they’re functioning as intended. This helps catch bugs early in the SDLC and reduces the need for extensive debugging later, but the drawback is it doesn’t test the entire system (GeeksforGeeks, 2025). Integration testing checks whether different components work correctly together and helps reduce the need for extensive debugging later (GeeksforGeeks, 2025). However, it is more complex and time-consuming than unit testing. White box testing was used to ensure the code logic and branches execute as intended, which was useful in this project because the assertions wouldn’t work if the branches don’t execute properly (GeeksforGeeks, 2025).

The software techniques not used were black box testing and system testing. Black box testing is evaluating the output of the code without examining its internal structure (GeeksforGeeks, 2025). This can be useful when verifying that the application meets user requirements, as it focuses on how the software behaves from an external perspective. System testing is testing the entire application to ensure all components work together as expected (GeeksforGeeks, 2025). It checks both functional and non-functional requirements such as performance, reliability, and security. Skipping system testing could mean missing issues that only appear when different parts of the software interact in real-world scenarios.

While working on this project, I took a careful and focused approach. As a software tester, I made sure to check that the code worked the way it was supposed to, but I also tested for situations where it might fail. For example, when testing the deleteContact method, I didn’t just confirm that the contact was removed, I also tested what would happen if I tried to retrieve that same contact afterward. The test confirmed that it threw an IllegalArgumentException, which is what I expected. That helped prove the system was handling errors correctly.

It was important to understand how various parts of the code were connected. Adding, updating, and deleting all used the same data structure, so a bug in one method could easily affect the others. I used @BeforeEach to set up shared test data before each test ran. That made it easier to keep the tests consistent and spot issues that could affect more than one part of the system.

To avoid bias, I didn’t assume my code was perfect. I tested for things like duplicate IDs or missing info, like a blank phone number. Even though I wrote the code, I still treated it like it could have problems. Bias can be a concern when you’re testing your own work, because it’s easy to only check the paths you think will work. That’s why I made sure to try inputs and situations that would challenge the logic of the code, not just confirm it.

As a developer, I know it’s important to stay committed to quality. Cutting corners may seem faster at first, but it often leads to technical debt, which can slow things down later. For example, skipping proper testing or writing messy code might save time now, but fixing it later takes more effort. To avoid that, I plan to stick to clean coding habits like writing tests for each feature, reusing code when it makes sense, and making sure everything is easy to read and maintain. That way, my future self (or someone else working on the code) won’t have to spend extra time cleaning it up.

**References**

GeeksforGeeks. (2025c, March 1). *Software testing techniques*. https://www.geeksforgeeks.org/software-testing-techniques/