**Module Seven Journal**

Naim Lindsay

Southern New Hampshire University

CS 319

Professor David Handlos

June 19, 2025

### **Summary and Reflections Report: Contact, Task, and Appointment Services**

For Project One, I worked on building and testing three key back-end features: Contact Service, Task Service, and Appointment Service. Each service had its own requirements and structure, so I took a test-driven approach and wrote JUnit tests to make sure the code behaved the way it was supposed to. In this reflection, I’ll go over how I wrote my tests, how they matched the requirements, and what I learned from the process. I’ll also explain the testing techniques I used and how my mindset around software quality evolved throughout the project.

### **Unit Testing Approach and Alignment to Requirements**

I started each part of this project by reading the requirements carefully and turning each rule into one or more test cases. For the Contact Service, I tested that the contact ID could not be changed after creation, that first and last names were no longer than 10 characters, and that the phone number had to be exactly 10 digits. I created test methods that checked both valid and invalid inputs. For example, I wrote a test where I tried to set a phone number like "123" and confirmed that it threw an exception.

In the Task Service, I followed the same pattern. I wrote tests to make sure the task ID was no more than 10 characters and that names and descriptions followed the specified limits. I also wrote tests for edge cases, such as inputs that were exactly the maximum allowed length. For the Appointment Service, I tested both formatting and logical rules. I confirmed that the system would reject null values and past dates, which helped verify that scheduling worked as expected. All of my tests were based directly on the requirements, which made it easier to see the connection between what I was testing and what the software was supposed to do.

### **Effectiveness of My Tests**

I didn’t use a test coverage tool like JaCoCo, but I focused on strong functional coverage. Every public method had at least one test, and most had several to cover valid input, invalid input, null values, and edge cases. For example, in the Appointment Service, I created a test that accepted a future date and another that triggered an exception with a past date. Even without formal metrics, I felt confident my tests were effective because they helped catch logic flaws and protected against unexpected inputs. If I ever made a change and something broke, the tests would let me know immediately. That’s one of the best parts of unit testing, it builds a layer of safety for future development.

### **Experience Writing JUnit Tests and Code Quality**

Writing the tests was one of the most useful parts of this project. I used @BeforeEach in my test classes to create reusable setup code that made everything cleaner. For example, in ContactServiceTest, I created a contact with ID "001" before each test so I didn’t have to repeat that in every method. I also focused on writing small, specific tests. That way, if a test failed, I’d know exactly which part of the functionality was broken. For example, I had one test just for checking phone number length and another just for handling null values. This made debugging and maintenance easier and helped keep my code organized.

### **Testing Techniques I Used**

The main testing technique I used was unit testing with JUnit. Since I was working directly with the logic inside the service classes, this was a white-box approach. I used assertions like assertEquals to check expected results and assertThrows to confirm that invalid input caused exceptions. I also applied input partitioning by testing across a range of acceptable and unacceptable values. For example, in the Task Service, I tested valid task names, names that were too long, and null names. These tests helped confirm that the system could handle all types of input correctly.

### **Techniques I Did Not Use**

I didn’t use integration testing, since the services weren’t interacting with each other or with external systems like a database or a front-end. I also didn’t use regression testing or exploratory testing. In a larger or more connected system, those would have been useful. Integration testing would help confirm that services can work together without issues. Regression testing would ensure that updates don’t break older functionality. Exploratory testing is great for UI-heavy systems where users might interact with the software in unexpected ways.

### **Practical Uses and Implications**

The techniques I used in this project have a lot of practical value beyond the classroom. In real software projects, writing thorough unit tests is key to preventing bugs, making safe changes, and improving team communication. For example, having good unit tests means I can refactor code with confidence because I’ll know right away if something breaks. Integration testing becomes more important when APIs or modules need to communicate reliably. Knowing when and how to apply different testing methods is something I plan to carry with me in future work.

### **Mindset: Caution, Bias, and Discipline**

### One of the biggest takeaways from this project was learning to test with caution. I couldn’t assume that just because the code looked correct, it would actually work. For example, I initially forgot to write a test for a null last name in the Contact Service. When I added that test and it failed, I realized my code didn’t catch that case. That mistake helped me see how important it is to be careful and think like a tester, not just a developer.

I also worked to avoid bias in my testing. It’s easy to only write tests for what I expect users to do, but users make mistakes. That’s why I wrote tests with bad inputs, empty strings, and other edge cases. Testing only the happy path doesn’t give you a full picture of how solid your code really is.

Discipline played a big role too. It would have been easy to write one or two tests per service and call it done, but I stuck to a structure that helped me cover every requirement. That discipline saves time later and keeps technical debt from piling up. Going forward, I plan to write tests before adding new features, avoid hardcoding, and make sure validation is always in place.

### **Conclusion**

This project helped me grow both as a developer and as a tester. I learned how to turn software requirements into working test cases, how to write code that handles edge cases, and how to build reusable tests that will last over time. It also helped me understand how important testing is for software quality, and how having the right mindset can make a big difference. These lessons will stick with me as I continue my journey in software development.

### **References**

FreeCodeCamp. (2022, March 10). How to write unit tests in Java. <https://www.freecodecamp.org/news/java-unit-testing/>

Simform. (2021, November 15). Unit testing best practices: 10 ways to get the most out of your tests. <https://www.simform.com/blog/unit-testing-best-practices/>

Testim. (2025, February 20). Testing best practices you must implement today. <https://www.testim.io/blog/testing-best-practices-you-must-implement/>