# **Summary and Reflections Report**

CS-320 Project Two

Lamberto Nunez

9/13/2025

## **Summary**

### **Unit Testing Approach and Alignment**

My unit testing approach for the contact, task, and appointment services was to develop a comprehensive and methodical test suite directly aligned with the specified software requirements. For each of the three services, the core functionality centered on performing CRUD (Create, Read, Update, Delete) operations while enforcing a set of validation rules on the data. My testing methodology was to create granular JUnit tests that would validate each requirement individually, covering both the successful execution (positive cases) and the graceful failure (negative cases) of these operations.

This approach was highly aligned to the software requirements because every test case was a direct validation of a specific functional or non-functional requirement. For example, for the Contact Service, a test method named testAddContact() was created to directly support the requirement for contact creation. Similarly, tests were designed to validate the field constraints, such as ensuring that the firstName field was no more than ten characters. This systematic validation provided concrete evidence that the code adhered to the project's specifications. For the Task Service, tests like testUpdateTaskDescription() ensured that only valid updates were applied, thus aligning the tests directly with the specified business rules.

### **Overall Quality of JUnit Tests**

The overall quality of my JUnit tests was high, as evidenced by the high code coverage percentages achieved and the robustness of the test cases. My tests for the Contact Service reached approximately 95% coverage, while those for the Task Service achieved around 92%. This level of coverage is significant because it indicates that a vast majority of the codebase, including various code paths and conditions, was executed by the test suite. This provides a strong level of confidence that the services behave as expected and that the risk of undetected bugs is low.

The effectiveness of these tests was further amplified by the inclusion of negative test cases. By deliberately providing invalid inputs such as null values, oversized strings, and non-existent IDs, the tests verified that the system would handle these exceptions gracefully and predictably. This robust approach to testing minimized the risk of undetected defects that could otherwise manifest in a production environment.

### **Experience Writing JUnit Tests**

My experience writing the JUnit tests was one of applying a disciplined and thoughtful approach to ensure both technical soundness and efficiency.

To ensure the code was **technically sound**, I focused on creating modular, well-structured test methods with clear naming conventions that communicated the purpose of each test. The use of precise JUnit assertions was critical for validating expected outcomes. As cited in my reflections, an assertion like assertEquals("John Doe", contactRepository.getContactById(1). getName()); was used to guarantee that the retrieved contact's name matched the expected value after an operation. This approach provided definitive proof of correctness. Another example, assertTrue(taskRepository.getAllTasks(). contains(task));, confirmed that a newly added task was correctly present in the repository, thereby validating both the insertion logic and the retrieval mechanism.

To ensure the code was **efficient**, I avoided redundant test code and leveraged key features of both the JUnit framework and the Java language. I made extensive use of the @BeforeEach annotation to initialize common test data before each test method, preventing repetitive setup code. Furthermore, my choice to use a HashMap for in-memory storage of contacts and tasks allowed for an efficient O(1) lookup time, ensuring that the tests could run quickly even as data sets grew. Finally, the use of stream operations for verification, as seen in the code assertTrue(taskRepository.getAllTasks().stream().anyMatch(t -> t.getId().equals(task.getId())));, provided a concise and performant way to check for the existence of an object within a collection.

## **Reflection**

### **Testing Techniques**

Throughout this project, the primary software testing techniques I employed were **Unit Testing**, **Boundary Testing**, and **Negative Testing**.

* **Unit Testing:** This was the foundational technique. It is characterized by testing individual methods and classes in isolation, providing rapid feedback to the developer. It is an essential practice in modern development, enabling continuous integration and providing a safety net for regression testing.
* **Boundary Testing:** This technique involves testing input values at the limits of acceptable ranges. It is particularly effective at identifying "off-by-one" errors and other validation bugs that might occur at the edge of a data domain.
* **Negative Testing:** This technique focuses on deliberately providing invalid, unexpected, or malicious inputs to ensure the system handles errors gracefully. It is a critical practice for verifying the robustness of a system and preventing security vulnerabilities.

Other software testing techniques that were not used for this project include **Integration Testing**, **System Testing**, and **Acceptance Testing**.

* **Integration Testing:** This technique focuses on verifying the interactions and data flow between different modules or components. Its practical use is in validating APIs or microservices architectures, where multiple services must communicate effectively.
* **System Testing:** This involves testing the complete, integrated system to verify that it meets all specified requirements, including non-functional aspects like performance and security. It is crucial for mission-critical applications where failure could have severe consequences.
* **Acceptance Testing:** This is a final validation step to determine if the system meets business requirements and is ready for deployment. It is often performed by end-users or stakeholders and is vital for custom software development to ensure the client’s needs have been met.

### **Mindset**

#### **Caution**

In acting as a software tester, I employed a mindset of caution that was heavily influenced by my personal experiences and a "what could go wrong" attitude. This perspective, honed by my horror novel hobby, led me to think about failure scenarios and edge cases that a developer might overlook. It was important to appreciate the complexity and interrelationships of the code, even in a small project, because an issue in one method could cause a cascading failure elsewhere. The London Ambulance Service computer-aided dispatch system failure serves as a powerful reminder of this. That system's defects—including memory leaks and poor error handling—led to a catastrophic breakdown, resulting in delays that were linked to dozens of deaths. This example underscores the necessity of testing software under realistic conditions and with a cautious mindset to prevent catastrophic failures.

#### **Bias**

I tried to limit bias in my review of the code by systematically checking against predefined requirements and test cases rather than relying on intuition. As a software developer, I can imagine that bias would be a major concern if I were responsible for testing my own code. A developer might assume certain conditions "would never happen" or be blind to flaws in their own work. For instance, a developer might not test a method with a null input because they assume a user interface will prevent it. This bias, however, would lead to a critical failure if the API were called from another system that did not have the same input validation, creating a serious vulnerability.

#### **Discipline**

The importance of being disciplined in a commitment to quality as a software engineering professional cannot be overstated. It is critical not to cut corners when writing or testing code because doing so creates technical debt that will need to be addressed later at a much higher cost. The experience of accumulated technical debt in my personal projects has taught me how much more difficult and time-consuming it is to fix problems after they have been deployed. A lack of discipline leads to a flawed product, customer dissatisfaction, and potential security vulnerabilities. To avoid this as a practitioner, I plan to leverage formal testing practices like Test-Driven Development (TDD) and continuous integration with automated testing. These practices embed a commitment to quality directly into the development process, ensuring that every code change is thoroughly validated before it is merged and deployed.

## **References**

Beynon-Davies, P. (1999). Human error and information systems failure: The case of the London ambulance service computer-aided despatch system project. *Interacting with Computers, 11*(6), 699–720.

Finkelstein, A., & Dowell, J. (1996). A comedy of errors: the London Ambulance Service case study. In *Proceedings of the 8th International Workshop on Software Specification and Design* (pp. 2-4). IEEE Computer Society Press.

García, B. (2017). *Mastering software testing with JUnit 5*. Packt Publishing.

*Wired*. (2009, October 26). *Oct. 26, 1992: Computer crash kills dozens*. <https://www.wired.com/2009/10/1026london-ambulance-computer-meltdown>