**Summary and Reflections Report**

**Summary**

When I worked on Project One, I built specific unit tests for each of the three services, contact, task, and appointment, which mirrored the customer requirements closely. For example, in the contact service, I tested that contact IDs were unique, not null, and limited to ten characters, and that first names, last names, phone numbers, and addresses adhered to the given constraints. My tests used assertions like assertThrows(IllegalArgumentException.class, () -> new Contact(null, "John", "Doe", "1234567890", "Address")) to verify that invalid input was handled correctly. For the task service, I tested name and description length restrictions and that invalid inputs triggered exceptions. For the appointment service, I validated that appointment IDs met length requirements and that appointment dates were not null or set in the past. Each test was structured to confirm that the system rejected incorrect inputs gracefully and accepted valid ones without issue.

I believe my testing approach aligned fully with the software requirements because each validation rule defined in the project prompt had a corresponding test. Coverage metrics supported that confidence by showing that nearly every branch, edge case, and exception path was exercised during testing. That gave clear evidence that the tests were effective and meaningful. In real development, this type of alignment is vital because missing even one requirement can lead to costly rework, defect leakage, or failure once the product is in production.

Writing these JUnit tests taught me a lot. I strove to ensure technical soundness by using well-structured assertions, relying on meaningful error messages, and grouping test setup and assertions clearly. I followed the Arrange-Act-Assert model in each test, making them easy to read and maintain, as recommended in recent best practices (Microsoft, 2025). I also focused on efficiency by consolidating similar test scenarios into parameterized tests where possible, reducing redundancy while maintaining full coverage. These practices not only made my tests reliable but also made them easier to extend when requirements changed, which is an important skill in real projects where requirements often evolve.

**Reflection**

In this project, I primarily used black-box testing and boundary-value analysis. Black-box testing let me treat each service from the user’s perspective, focusing purely on whether the observed behavior matched the requirements. Boundary-value analysis helped me catch off-by-one errors and edge scenarios that are often sources of bugs. For instance, when testing the maximum string length for a first name, I made sure to test not only a name that was exactly ten characters but also one that was eleven. If I had not performed this boundary testing, I might have assumed the code handled it correctly and missed a potential defect. This type of thinking shows why structured techniques are necessary. Many bugs live at the edges of requirements, not in the middle.

I did not use white-box testing, which examines internal logic and paths within the code. While that was not necessary for these simple services, I know that in larger applications, white-box techniques are valuable for uncovering hidden inefficiencies or logical gaps (Software Testing Magazine, 2024). Integrating both black-box and white-box testing is generally the most robust approach when developing complex systems. It ensures not only that the software behaves correctly on the surface but also that its internal logic is efficient, maintainable, and consistent with industry standards.

Exploratory testing also played a role, though informally. I deliberately tried unexpected or invalid inputs like null values, overly long strings, and duplicate IDs to see how the system responded. This kind of dynamic, unscripted testing can help identify issues that structured tests might miss. Exploratory testing has been recognized as essential for uncovering hidden issues, especially in agile or evolving environments (BrowserStack, 2025). Without it, subtle defects might slip through scripted testing and only emerge later in production, where fixing them is much more expensive.

I approached testing with caution, understanding that even a small oversight could produce a larger issue later. That led me to test not only normal use cases but also failure and edge cases. For example, I tested adding duplicate appointments or using boundary values for string lengths to ensure that my services failed safely and predictably. That mindset is important because many real-world defects come from developers assuming that their code is “good enough” without actually validating how it behaves under unusual conditions.

I was aware of the bias that comes from testing one’s own code. To reduce that bias, I consciously wrote tests targeting failure scenarios and tried to anticipate common mistakes I might make as a developer. Having the tests throw out invalid input helped me see parts of my code from a different perspective and feel more confident in its reliability. In industry, teams often split development and testing roles for this very reason, since testers tend to assume there are defects and search for them more aggressively than developers. I tried to adopt that same mindset even though I was testing my own work.

Discipline in testing proved critical. Cutting corners might have saved time, but it would also have introduced technical debt. To avoid that, I committed to rigorous testing from day one. I intend to keep using coverage tools, maintain clear and focused test writing patterns, and regularly refactor both code and tests to prevent future issues and keep the codebase clean and maintainable. Software history is full of examples where poor testing discipline led to catastrophic results, such as large-scale security breaches, data losses, or public recalls. These examples emphasize why disciplined testing is not optional but central to the work of a responsible developer.

**References**

BrowserStack. (2025). Exploratory testing (part of *Different Types of Testing in Software*). Retrieved from <https://www.browserstack.com/guide/types-of-testing>

Microsoft. (2025, April). Best practices for writing unit tests (part of *.NET Documentation*). Retrieved from <https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices>

Software Testing Magazine. (2024). Black-box vs white-box testing: Choosing the right balance. Retrieved from <https://www.softwaretestingmagazine.com/>