**7-2 Project Two**

Sharif Ayesh

SNHU

CS 320

Professor Bermudez

04/15/2025

**CS 320 Project Two: Testing Summary and Reflection**

**Unit Testing Summary**

In Project One, I developed three core Java services: Contact, Task, and Appointment. These services were designed to handle CRUD (Create, Read, Update, Delete) operations while enforcing strict input validation. I then wrote JUnit tests for each class to verify these services' accuracy, stability, and robustness under various conditions. This project helped me understand not only how to write unit tests but also how to test with intentionality and purpose.

I implemented validation logic for the Contact service to ensure that each field met specific length and null-value requirements. For instance, contact names over ten characters were truncated, and missing fields were set to a placeholder value like "NULL". Phone numbers were validated for exactly 10 digits, defaulting to "5555555555" otherwise. My unit tests confirmed that these rules were enforced correctly. In ContactTest.java, I verified that none of the fields could exceed their character limits or be null. In ContactServiceTest.java, I tested updates and deletions based on auto-generated contact IDs using methods like updateFirstName and deleteContact. I used JUnit features such as @Order, @DisplayName, and assertions like assertEquals and assertNotNull to structure my tests.

The Task service followed a similar pattern. I set constraints for the task name (20 characters max) and description (50 characters max), with null or empty inputs being replaced with "NULL". My test classes ensured that both constraints and CRUD operations functioned as expected. For instance, TaskServiceTest.java checked that task names and descriptions were correctly updated and that tasks were successfully added and deleted from the list. I also included logic in TaskTest.java to test edge cases, including when names or descriptions exceeded the allowed lengths or were null. I felt confident in the service’s ability to handle invalid input due to these tests gracefully.

For the Appointment service, I added a slightly more complex validation layer. Appointment dates were required to be set in the future, and if not, a fallback default date (January 1, 2022) was used. Descriptions had a strict 50-character limit. To maintain immutability, I used a cloning strategy cloneWithNewDate and cloneWithNewDescription to return updated appointment instances without altering the original. My test logic validated that these immutable updates functioned correctly.

**Testing Techniques and Practices**

I primarily relied on black-boxtesting to validate outputs without knowing the internal state of objects. I simulated real-world usage scenarios and focused on results rather than implementation. For example, I tested whether a task was correctly added to a list and whether its attributes matched the expected values. I also practiced equivalence partitioning by testing valid and invalid inputs in various boundary ranges, such as phone numbers of varying lengths, overly long descriptions, and null fields.

One area where I could have improved is implementing boundary testing more deliberately. While writing test cases for over-limit inputs, I didn’t always test values inside the valid range (e.g., nine vs. 10 characters for phone numbers). Additionally, I didn’t use mocking frameworks like Mockito, since the services didn’t rely on external systems like databases or APIs. However, I recognize the importance of mocking in more complex applications and plan to incorporate it in future projects.

The structure and organization of the JUnit tests made a big difference in keeping the code maintainable. I adopted consistent naming conventions, used the @TestMethodOrder annotation to ensure tests executed in a logical sequence, and regularly commented on each test case. This helped me keep track of what was being validated and why. As emphasized by Ammann & Offutt (2016), well-organized and well-documented unit tests are critical for scalable software projects.

**Testing Mindset and Reflection**

Developing a testing mindset was one of the most valuable parts of this course. In the past, I often viewed testing as an afterthought, something to do once the code was “done.” This project shifted my thinking toward test-driven development (TDD) principles, where I wrote or planned tests in tandem with the application logic. Writing tests early helped clarify how each method should behave and made my code more intentional.

I also became more aware of confirmation bias. Early in development, I assumed the constructors and setters were handling validations properly and skipped over some tests. When I went back and tested edge cases like null inputs or overly long descriptions, I discovered a few missed issues. This reinforced the importance of disciplined and unbiased testing. As Martin (2008) argues in Clean Code, effective testing requires a mindset that is skeptical of the code’s correctness until proven otherwise.

In the future, I aim to introduce regression testing practices into my development workflow. While my tests were practical for initial implementation, I need a framework for re-running them whenever code changes occur. Automating regression tests would help ensure that new updates don’t break existing functionality.

**Conclusion**

Overall, Project One taught me more than just how to write tests; it taught me how to think like a tester. Through JUnit, I learned to rigorously validate data, explore edge cases, and test with intention. I also gained hands-on experience with essential software engineering principles like immutability, input validation, and modularity. By adopting a cautious, disciplined, and unbiased mindset, I uncovered hidden issues and improved code quality significantly. I will continue to use these practices in future projects and in my professional career.

**References:**

Ammann, P., & Offutt, J. (2016). Introduction to Software Testing (2nd ed.). Cambridge University Press.

Martin, R. C. (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.