**Project One: Summary and Reflections Report**

Aleyna Morales

CS-320: Software Test, Automation QA

Kevin Tepas

October 18, 2025

**Summary**

Throughout the development of the Contact, Task, and Appointment services, I used a structured unit testing approach based on the software requirements. Each class was tested with a corresponding JUnit test class that validated object construction, field constraints, and service operations such as adding, updating, and deleting records. For example, in the Contact class, the JUnit tests ensured that no field exceeded its required length and that null or invalid data raised an IllegalArgumentException. These tests were directly aligned with the system’s functional requirements that specified validation rules such as contact IDs being unique and not longer than 10 characters, phone numbers containing exactly 10 digits, and address fields not exceeding 30 characters.  
  
In the TaskService tests, my approach focused on verifying CRUD functionality, ensuring that a new task could only be added once per unique ID and that duplicate entries were prevented. Assertions such as assertThrows() and assertEquals() provided validation that methods behaved as expected. Similarly, in AppointmentService, JUnit tests confirmed that appointments could not be scheduled in the past and that the description field respected its maximum length of 50 characters. These targeted tests demonstrated strong alignment between the test cases and the given project requirements.  
  
To evaluate quality, I examined the coverage of my JUnit tests. Each service’s test suite covered nearly all possible input conditions; valid, invalid, and boundary cases, indicating a high coverage percentage. Since the tests included both positive and negative scenarios, I am confident they effectively validated the functional behavior of the application components.  
  
Writing the JUnit tests gave me a hands-on understanding of how to verify code logic systematically. I ensured that the code was technically sound by using explicit assertions for every test condition. For instance, I used assertThrows(IllegalArgumentException.class, () -> new Contact(null, "Alice", "Brown", "1234567890", "123 Main Street")) to confirm that null IDs were properly rejected. This approach provided confidence that the validation logic within constructors and service methods worked as designed.   
  
Efficiency in my tests came from organizing reusable test objects and minimizing redundant code. For example, creating one valid Task object and reusing it across multiple methods avoided repetition. Tests were isolated to specific behaviors, such as testing only the updateName() or deleteContact() functionality this ensurred fast execution and clear test outcomes. Each class’s service-level tests validated the integrity of in-memory HashMap operations, ensuring constant-time retrieval and modification performance.

**Reflection**

The main software testing techniques I employed were unit testing, boundary value testing, and black-box testing. Unit testing isolated each method to ensure it functioned correctly, while boundary value testing checked limits such as the maximum allowed characters for IDs, names, and descriptions. Black-box testing was used to evaluate how the methods handled expected versus unexpected inputs without examining the internal logic, focusing purely on input-output behavior.  
  
Techniques not used in this project included integration testing, system testing, and regression testing. Integration testing would combine multiple modules to evaluate how they interact as a whole, while system testing would validate the complete end-to-end workflow of the application. Regression testing ensures that previously working features still function after new code changes. While these methods were beyond the project’s scope, they are vital in larger software systems to verify stability and consistency.  
  
In practical terms, unit and boundary testing are most useful during early development stages when validating core logic. Integration and regression testing become essential later when the system grows or when updates risk affecting existing functionality.  
  
Throughout this project, I adopted a cautious and methodical mindset as both a developer and tester. I recognized the complexity of ensuring that all validation rules worked correctly together. When testing the Appointment class, it was crucial to verify that both the date and description validation worked independently and in combination. Appreciating these two helped prevent subtle bugs that could emerge from overlapping conditions.  
  
To reduce bias while reviewing my own code, I approached testing with the mindset of an external quality assurance engineer rather than the developer who wrote it. I deliberately tried to “break” my code by entering invalid data or unexpected sequences of operations. For instance, I attempted to add multiple contacts with the same ID to ensure that the ContactService correctly rejected duplicates. This impartial testing helped identify weaknesses that I might otherwise overlook if I assumed my code was correct.  
  
As a software engineering professional, maintaining discipline and commitment to quality is essential. Cutting corners in testing could result in unreliable software, user frustration, or technical debt. To avoid technical debt in the future, I plan to continue writing thorough unit tests for all new code, performing code reviews with peers, and applying continuous integration tools that run tests automatically with every update. This disciplined approach ensures long term maintainability, scalability, and trust in my codebase.  
  
Completing Project One reinforced my understanding of software testing as both an analytical and creative process. Writing effective JUnit tests not only validated my code but also strengthened my ability to think critically about design, logic, and edge cases. These skills will continue to guide my growth in the future, helping me focus on delivering dependable, high-quality applications.