Burak Tutkavul  
06/16/2025  
CS 320 Project Two

**Summary and Reflections Report**

1. **Summary**

As a software engineer for Grand Strand Systems, I adopted a systematic and requirement-driven approach to unit testing while developing the contact, task, and appointment services in Project One. My unit testing approach was grounded in black-box testing and boundary-value analysis to validate service behavior against user-defined constraints and functionality.

For the Contact Service, unit tests focused on validating the creation, update, and deletion of contact objects. Each test ensured that no field exceeded maximum character limits and that IDs were immutable. For instance, the test method testUpdateFirstName() confirmed that names could be updated within specified constraints, while testUpdateFirstNameTooLong() ensured the application rejected overly long inputs. These tests directly aligned with the requirements defined in the project specifications, which mandated strict field length constraints and immutability for contact IDs.

In the Task Service, unit tests verified similar constraints, including task creation with unique IDs and valid title and description fields. The test testAddTaskWithEmptyDescription() ensured that empty inputs were handled gracefully, maintaining data integrity and aligning with non-null input requirements.

The Appointment Service required an additional focus on date validation. For example, in the test testAddAppointmentWithPastDate(), I ensured the system rejected any appointment scheduled in the past, as per project requirements. This alignment between tests and functional requirements guaranteed that edge cases and real-world user inputs were considered.

The quality of the JUnit tests was validated using coverage analysis. My test suite achieved over 84% code coverage across the services, confirming that most branches, conditions, and paths were exercised.   
  
This high coverage provided confidence in the robustness of the application.My experience writing the JUnit tests was rewarding. I ensured technical soundness by using assertions such as assertThrows() and assertEquals() to cover both positive and negative paths. For example:

assertThrows(IllegalArgumentException.class, () -> contactService.updateFirstName("12345", "VeryLongNameExceedingLimit"));

This line demonstrates validation of a constraint using exception handling.

To keep the tests efficient, I avoided redundancy by using setup methods and tested only one behavior per method to ensure single responsibility. For example:

@BeforeEach

void setUp() {

contact = new Contact("12345", "John", "Doe", "1234567890", "123 Street");

}

This code snippet reduced duplication and improved test clarity.

1. **Reflection**

The primary software testing techniques used in this project were unit testing and boundary-value analysis. Unit testing isolated each service method and validated it independently, while boundary-value analysis ensured edge cases like maximum string length or invalid dates were handled appropriately.

Techniques not used include integration testing and system testing. Integration testing, which focuses on verifying interactions between multiple modules, would be useful when connecting these services to the mobile front-end. System testing, which validates the end-to-end behavior of the application, would be ideal once the complete app is deployed.

Unit testing is most practical during early development, providing fast feedback and reducing the cost of fixing bugs. Integration testing is vital for verifying contracts between modules, and system testing ensures the entire application meets business requirements.

While working on this project, I adopted a mindset of caution and thoroughness. I treated each service independently and ensured complete test coverage to catch hidden bugs. Understanding the interdependencies of fields (e.g., ID immutability affecting update logic) helped me anticipate how seemingly minor changes could impact other parts of the code.

To limit bias, I reviewed test cases critically and tested both expected and unexpected inputs. As a developer, it is easy to assume correctness in one’s code. To combat this, I wrote tests that deliberately tried to break the logic. For instance, by testing date validation using LocalDate.now().minusDays(1), I simulated invalid input even though I initially believed the function would handle it.

Being disciplined about quality is essential in software engineering. Cutting corners leads to technical debt, which slows down future development. I plan to avoid technical debt by writing meaningful tests during development, performing regular refactoring, and using tools like static code analyzers and coverage tools to maintain code health. For example, I plan to integrate SonarLint into my IDE to continuously monitor code quality.

In summary, this project strengthened my understanding of effective testing strategies and highlighted the importance of thoughtful, thorough, and unbiased validation in software development.  
  
**References**

Jorgensen, P. C. (2013). *Software testing: A craftsman's approach* (4th ed.). CRC Press.  
 Retrieved from :   
<https://user.ceng.metu.edu.tr/~e1451970/SQA/books/___software-testing-a-craftmans-approach-4-edition.pdf>

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing* (3rd ed.). John Wiley & Sons.  
Retrieved from : <https://user.ceng.metu.edu.tr/~e1451970/SQA/books/___The_Art_of_Software_Testing_3rd_Edition.pdf>