# CS 320 Module 7: Project Two

During the development of the mobile application for Project One at Grand Strand Systems, I adopted a structured and methodical approach to unit testing for the contact, task, and appointment services. Each service was designed using in-memory data structures, eliminating the need for external dependencies like databases, and tested using JUnit 5, which allowed for isolated, repeatable, and efficient verification of functionality.

For the ContactService, I focused on validating that all constraints outlined in the requirements were enforced. These included ensuring that the contact ID was unique and immutable, that first name, last name, and phone number did not exceed character limits, and that the phone number was a valid ten-digit number. My tests aligned directly with these requirements. For example, the test assertThrows(IllegalArgumentException.class, () -> contact.setPhone("123")); ensured that invalid phone numbers would be rejected.

The TaskService followed a similar methodology. I implemented test cases to confirm that the task ID could not be null or updated after creation, that task names and descriptions respected length constraints, and that the service could successfully add, update, and delete tasks. An example of enforcing immutability was demonstrated with assertThrows(UnsupportedOperationException.class, () -> task.setTaskId("NEWID"));.

The AppointmentService was more sensitive to date validation. My tests were crafted to ensure that appointment dates were not null and not in the past. Using assertTrue(appointment.getAppointmentDate().after(new Date())); verified compliance with the requirement that all appointments must be scheduled for a future date. The use of java.util.Date and the condition appointmentDate.before(new Date()) helped enforce this logic in the class implementation itself.

Overall, my unit testing approach closely mirrored the project requirements. I utilized high-coverage testing, ensuring each method and logical branch was exercised. Code coverage metrics from my IDE indicated 100% line coverage and approximately 90% branch coverage across all services, which gave me confidence in the robustness of the tests.

My experience writing JUnit tests throughout this project was both educational and rewarding. Writing tests first, or shortly after writing each method, helped clarify expected behavior and identify edge cases. I ensured the code was technically sound by incorporating exception handling and validating outputs rigorously. Efficiency was addressed by structuring tests using @BeforeEach methods to avoid redundancy and improve readability. For instance:

@BeforeEach  
void setup() {  
 contact = new Contact("001", "Zac", "Harrington", "1234567890", "123 Main St");  
}

This allowed consistent test initialization without repetitive code in each test method.

Throughout this project, I employed several software testing techniques, notably unit testing and boundary testing. Unit testing focused on verifying the functionality of each class independently and was facilitated using JUnit 5. Boundary testing ensured that inputs met specified constraints, such as character length for names or valid date ranges for appointments. These techniques are particularly effective in early development stages where logic validation is critical (Garcia, 2017).

Other techniques not employed in this project but worth noting include integration testing and system testing. Integration testing would have been useful if the services depended on each other or external modules, while system testing ensures the entire application meets end-user expectations. Additionally, exploratory testing, although informal, can uncover issues not anticipated in structured test cases (Myers, Sandler, & Badgett, 2011).

Each of these techniques has practical uses. Unit testing excels in test-driven development and continuous integration pipelines. Integration testing is ideal for service-oriented architectures or APIs, and system testing is essential for applications nearing production. Exploratory testing is beneficial during quality assurance sprints or beta testing.

From a mindset perspective, I approached this project with a cautious and detail-oriented attitude. I recognized that even minor changes in one class could ripple into unexpected bugs elsewhere. For instance, modifying the date validation logic in the Appointment class briefly broke several previously passing tests. This highlighted the importance of understanding code interrelationships and maintaining tight feedback loops through testing.

To limit bias, I intentionally tested each method for both expected and unexpected inputs. I wrote negative test cases to challenge my assumptions. For example, I tested creating a contact with a null ID to confirm the application threw the proper exception. Had I not done so, I might have missed potential vulnerabilities or overlooked poor input validation. Bias is a real concern when developers test their own code, as it’s easy to subconsciously validate only the expected behaviors (Garcia, 2017).

Discipline and commitment to quality are paramount in software engineering. Cutting corners during development or testing can lead to technical debt, increased maintenance costs, and security vulnerabilities. I avoided shortcuts by strictly following requirement constraints, even when they seemed overly restrictive. For instance, limiting IDs to 10 characters required me to write utility functions to handle edge trimming and validations.

To prevent technical debt in my future projects, I plan to enforce coding standards, use version control with clear commit messages, and maintain a comprehensive suite of automated tests. These practices promote maintainability and ensure long-term code quality. Additionally, incorporating code reviews and using static analysis tools can provide early detection of issues.

In conclusion, this project helped solidify my understanding of various testing strategies, their implementation using JUnit, and the importance of adopting a professional mindset toward software quality. Through disciplined testing, conscious bias mitigation, and methodical validation, I was able to deliver a reliable and maintainable mobile application backend.

**References**

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

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing* (3rd ed.). Wiley.