Summary and Reflections Report  
*Harsh Patel*  
Grand Strand Systems  
06/22/2025

Summary

Unit Testing Approach

For Project One, I developed and tested three core services—ContactService, TaskService, and AppointmentService—to support the mobile application for our customer. My unit testing approach was methodical and consistent across all services. I began by identifying the key requirements for each service and then wrote JUnit test cases to validate individual behaviors based on those requirements.

* For ContactService, tests focused on ensuring that new contacts could be added with unique IDs, edited properly, and deleted from the internal data store.
* For TaskService, test cases checked that tasks were assigned correct attributes (ID, name, description), and that updates and deletions worked as expected.
* For AppointmentService, I validated that appointments were assigned future dates, had correct formatting, and allowed for proper updating and deletion.

Alignment to Software Requirements

My testing approach was strongly aligned with the documented requirements. For example, the ContactService required that no two contacts could have the same ID. I wrote a test case (testAddDuplicateContact) to specifically verify that an IllegalArgumentException would be thrown if a duplicate ID was added. Similarly, for TaskService, I enforced that the task name could not exceed 20 characters and validated this in the testUpdateTaskNameTooLong() method.

Quality of JUnit Tests

I ensured high-quality JUnit tests by aiming for 95%+ code coverage across all services. I used IntelliJ’s coverage report to verify that all logical branches—such as if-else conditions and exception handling—were exercised. The effectiveness of the tests was evident from their ability to catch edge cases early during development. For instance, testDeleteNonexistentAppointment() helped identify a missed null-check in the AppointmentService logic, which was corrected before deployment.

Writing JUnit Tests

The experience of writing JUnit tests was both challenging and rewarding. I used assertions such as assertEquals(), assertThrows(), and assertNotNull() to test expected behavior and exceptions. For example, in ContactServiceTest, I wrote:

java

assertThrows(IllegalArgumentException.class, () -> {

service.addContact(new Contact("123", "John", "Doe", "1234567890", "123 Main St"));

service.addContact(new Contact("123", "Jane", "Smith", "9876543210", "456 Elm St"));

});

To ensure efficiency, I avoided redundant code by using reusable setup methods annotated with @BeforeEach, and I grouped logically similar assertions together to make test results more readable and maintainable.

Reflection

Testing Techniques Employed

For this project, I primarily used unit testing and boundary testing. Unit testing ensured that individual methods worked in isolation. Boundary testing helped verify edge cases such as input length limits (e.g., task names and contact phone numbers). These techniques allowed me to detect both functional and input-related issues early in development.

Testing Techniques Not Used

I did not employ integration testing or system testing in this project. Integration testing would have involved testing how the ContactService interacts with the TaskService or AppointmentService, which was outside the scope. System testing would involve testing the complete mobile application with a UI and backend interactions, typically requiring tools like Selenium or Appium. These are better suited for end-to-end validation and would be more appropriate during final system validation stages.

Practical Uses of Techniques

* Unit testing is essential for agile, test-driven development (TDD) environments where immediate feedback is needed during coding.
* Boundary testing is useful in validating constraints set by the business (e.g., max field length).
* Integration testing would be valuable in microservices architectures to ensure components communicate correctly.
* System testing ensures the overall software meets client expectations and is suitable for pre-release phases.

Mindset as a Software Tester

While working on this project, I approached testing with a mindset rooted in caution, curiosity, and completeness. I was cautious not to assume that “simple” functions would behave correctly without testing. For instance, when implementing the deleteContact() method, I accounted for the possibility of trying to delete a non-existent contact, which helped me avoid a potential NullPointerException.

Understanding the interrelationship between services was also crucial. While the services were tested independently, I maintained an awareness of how they might be composed in a real application, which encouraged me to write clean, well-encapsulated code.

Limiting Bias in Code Review

To limit bias, I separated the roles of developer and tester in my workflow. After writing the services, I reviewed them the next day with a “tester’s mindset,” challenging assumptions I had made during development. I also used descriptive test names like testUpdateContactPhoneInvalidFormat() to keep the focus on behavior, not implementation.

Bias becomes a real concern when developers test their own code and assume it works because they wrote it. For example, I initially skipped writing a test for invalid appointment dates assuming the date parsing logic was reliable, only to later find that certain formats were being silently ignored. This reinforced the importance of testing every path, even ones I believed were foolproof.

Commitment to Quality and Avoiding Technical Debt

Being disciplined about writing and maintaining good test coverage is key to avoiding technical debt. Rushing to release without comprehensive tests often leads to bugs that are harder to trace and fix later. For example, if I had skipped validating task descriptions’ max length, it could have caused truncation issues or unexpected UI behavior later on.

As a future software practitioner, I plan to incorporate:

* Continuous testing in CI/CD pipelines,
* Strict adherence to TDD principles,
* Code reviews with a focus on test completeness, and
* Maintaining a robust suite of regression tests.

This discipline will ensure quality code delivery and maintain trust with clients and users alike.

References

JUnit 5 User Guide. (n.d.). Retrieved from https://junit.org/junit5/docs/current/user-guide/

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