Patricia — Software Engineer, Grand Strand Systems

Unit Testing Approach

Unit testing was implemented across three core features of the Project One mobile application: Contact Service, Task Service, and Appointment Service. The approach followed test-driven development (TDD) principles, validating each method against expected behavior before integration.

- Contact Service: Create, read, update, and delete operations were assessed, along with boundary conditions such as name length and duplicate entries. For instance, tests confirmed the system rejected empty or invalid contact names.

- Task Service: Validation included deadlines, task status transitions, and null handling. One test confirmed that marking a task as completely correct updated its status.

- Appointment Service: Time conflicts, date formatting, and user associations were validated. A key test ensured that overlapping appointments were flagged as conflicts.

Alignment to Requirements

The testing strategy mapped each software requirement to specific test cases. For example, multiple tests rejected invalid phone number formats to satisfy the requirement for valid contact information. Boundary and equivalence partitioning techniques were also used to efficiently ensure field constraints.

Quality of JUnit Tests

I achieved over 85% of test coverage, verified using JaCoCo. This high coverage ensured that most logical paths were exercised and that tests caught edge cases and regressions during refactoring. For example, a refactor flagged a null pointer issue via failed tests.

Experience Writing JUnit Tests

Writing JUnit tests was both methodical and rewarding. I used setup methods to initialize test data before each test and grouped related tests for clarity. For example, I created a reusable setup that added a sample contact before each test case.

To ensure technical soundness, I used assertions to validate outcomes and mocking tools to isolate dependencies. Efficiency was achieved by minimizing redundant setup and utilizing parameterized tests to verify multiple input values within a single test method. For instance, I assessed several invalid name formats in one test using a loop structure.

Reflection

Testing Techniques Used

I employed the following techniques:

- Unit Testing: Focused on individual methods and logic paths.

- Boundary Testing: Checked limits of input fields, such as maximum name length.

- Equivalence Partitioning: Grouped inputs into valid and invalid categories.

- Regression Testing: Re-ran tests after code changes to catch unintended effects.

These techniques ensured robustness and maintainability. For example, boundary tests caught an off-by-one error in the appointment time validator.

Techniques Not Used

I did not use:

- Integration Testing: Since the project focused on isolated services, I deferred integration tests to the next phase.

- System Testing: Full end-to-end testing was outside the scope of this unit-level assignment.

-Exploratory Testing: I relied on structured test cases rather than ad hoc exploration.

Each technique has its place. Integration testing is vital for validating interactions between modules, while exploratory testing is useful in UI-heavy applications where user behavior is unpredictable.

Mindset and Caution

I adopted a cautious and analytical mindset, treating each service as a potential source of failure. I appreciated the complexity of interrelated data, especially in appointment scheduling, where overlapping times and user conflicts required careful validation. For example, I created a test suite specifically for edge cases involving daylight saving time transitions.

Limiting Bias

To limit bias, I reviewed my code with a hostile tester mindset and enlisted peer reviews for my test cases. Developers can miss issues due to familiarity; for instance, I initially overlooked a null check in the task update method.

Commitment to Quality

Discipline in testing is non-negotiable. Cutting corners leads to technical debt, which compounds over time. I plan to avoid this by:

- Maintaining high test coverage

- Using continuous integration tools for automated testing

- Documenting assumptions and edge cases

- Refactoring with tests in place

For example, I created a checklist for each service to ensure all requirements were tested before submission.

References

Beizer, B. (1990). Software Testing Techniques (2nd ed.). Van Nostrand Reinhold.

Jorgensen, P. C. (2013). Software Testing: A Craftsman's Approach (4th ed.). CRC Press.