Kyle Marinaro

8/16/2025

CS 320

Project Two

During Project One, I undertook the implementation and thorough validation of the contact, task, and appointment services for a mobile application, with a particular emphasis on rigorous unit testing. Each service was examined independently to ensure strict adherence to the documented requirements.

For the contact service, I developed a comprehensive suite of JUnit tests to verify that each contact entity was instantiated with a unique identifier. I also enforced stringent input validation for all fields: names were constrained to a maximum of ten characters, addresses to thirty characters, and phone numbers to exactly ten digits. Boundary value analysis was employed to test cases at the permitted limits and just beyond, to confirm that constraint violations resulted in the appropriate exceptions. Furthermore, I verified that duplicate contact IDs were systematically rejected, thereby upholding the uniqueness requirement.

The task service was subjected to similarly methodical testing. My unit tests validated the creation, updating, and deletion of tasks by the specified constraints. Descriptions were restricted to under fifty characters, and due dates were mandated to be non-null. The test cases included both valid scenarios and those designed to elicit errors, such as attempts to create tasks with excessively long descriptions or to remove non-existent tasks. These tests confirmed not only functional correctness but also the system’s robustness in handling invalid operations and preserving data integrity.

Special attention was given to the appointment service, particularly due to its handling of date-related logic. Here, I ensured that appointment IDs were unique, descriptions were limited to fifty characters, and all scheduled dates lay in the future. The test suite included both valid and invalid cases, such as the creation of appointments with duplicate IDs, past dates, or descriptions exceeding the maximum length. For instance, I utilized assertThrows to verify that attempting to set an appointment in the past appropriately triggered an exception. My testing methodology was fundamentally requirements-driven. Each JUnit test directly corresponded to a documented functional or non-functional requirement for the application. This ensured that the tests were not arbitrary but rather provided targeted verification of both code behavior and business logic. When field length constraints were defined, I leveraged boundary testing; when uniqueness was required, I explicitly tested for duplicate insertions. This systematic approach substantiates the alignment of my testing with project needs.

Lastly, the breadth and depth of my JUnit tests were validated through coverage analysis. I confirmed that a significant majority of service methods and code branches were exercised by the test suite. For example, in testing the contact service’s addContact functionality, both successful and failing scenarios were assessed. The use of assertEquals and assertThrows allowed for precise validation of both expected outcomes and error conditions. Overall, this approach provided reliable assurance of software quality and conformance to requirements.

Efficiency was a major priority for me during test development. Honestly, there’s no bigger time-waster than duplicating the same setup all over the place, so I made sure each test method focused on a unique requirement. Redundant work? Not on my watch. For example, when I had several negative cases to test, I bundled them into a single helper method, cutting down on repeat code and making everything way easier to maintain. I also kept test data lean. Every test was targeted: one behavior, nothing extra to muddy the waters.

The techniques I relied on were tried and true unit testing, boundary value analysis, and equivalence partitioning. Unit tests let me zero in on each method by itself, confirming it worked as intended. Boundary value analysis was clutch for checking edge cases: names at exactly ten characters, addresses at thirty, that sort of thing. Equivalence partitioning? Lifesaver. Instead of getting lost in a sea of inputs, I picked representative good and bad examples. It’s a comprehensive balance, but not overkill. I didn’t touch integration, system, or performance testing for this project; those are better suited for checking how everything works together or under stress, which wasn’t in scope here.

When it came to mindset, I leaned into skepticism. My goal wasn’t to confirm the code was perfect, but to dig up anything that could go wrong. I checked not just the obvious invalid cases, but also weird stuff, nulls, whitespace, you name it. The contact, task, and appointment services all shared similar validation rules, so a gap in one could show up everywhere. That meant thorough testing across the board.

To keep myself from missing things, I deliberately wrote tests expecting failure. It’s easy to let your guard down when you’ve written the code yourself, but I assumed things would break like the appointment service letting past dates slip through. Writing the test showed the logic was mostly solid, though I did spot a need for clearer exception handling. That kind of critical perspective is invaluable.

Also, I refused to cut corners. Skipping tests or ignoring edge cases just racks up technical debt; minor problems snowball into serious issues down the line. So I stuck to the requirements, kept documentation on my test cases, and updated them whenever requirements changed. For instance, if international phone numbers became a requirement, I’d update the contact tests accordingly and make sure nothing else broke. That discipline is key to keeping code reliable and maintainable.

In summary, my unit tests for contact, task, and appointment services were comprehensive and requirements-driven. I validated positive, negative, and boundary cases, using proven techniques and steering clear of unnecessary ones. By staying skeptical, minimizing bias, and committing to best practices, I ensured the code was technically sound and efficient. These lessons aren’t going anywhere; they’ll keep shaping how I approach testing in every future project.