CS320

Professor David Handlos

Jermaic Fuentes

October 26, 2025  
 Grand Strand Systems   
 Working on the contact, task, and appointment services taught me how important it is for testing to begin with the requirements rather than the code itself. Before writing any JUnit tests, I broke down each requirement into a specific behavior that needed to be proven or rejected. That mindset helped me see the test suite as more than a way to “check” correctness it became a way to confirm that invalid data could never enter the system in the first place. This gave me much more confidence in the integrity of each class as it evolved.

For the Contact service, I verified correctness by beginning with a valid object and then targeting each rule with intentional failure cases. For example, the requirement that phone numbers must be exactly ten digits was enforced using a test like: assertThrows(IllegalArgumentException.class, () -> new Contact("ID", "John", "Doe", "12345", "Sunshine Rd"));.

This test directly reflects the rule, and because the exception is thrown from the constructor, it confirms that invalid data is stopped at creation time. I applied the same approach for first and last name length limits and the non-null requirement for all fields. Every constraint was paired with a corresponding test to ensure the behavior matched the specification.

The Task service followed the same pattern of validating a proper object first and then challenging the boundaries. For example, since the name field was limited to twenty characters, I passed in a twenty-one-character string to verify that the system rejected it. This made sure that the validation logic was precise, not just generally correct. The Appointment service brought in a slightly different kind of rule of time instead of string length. To support that, I used a helper method to generate a valid future date and then created a past date for the negative test to confirm that outdated appointments are rejected. This protected the logic against edge-case errors that could break functionality later.

At the service layer, I tested not only that methods returned true or false, but also that the underlying object state changed (or remained unchanged) as expected. For example, when updating a contact’s first name, I verified both the Boolean return value and the stored value in the object. This ensured that I was not just testing mechanics but confirming actual behavior. I also tested attempts to update or delete non-existent records and attempts to add duplicates. These cases helped verify that the system behaved safely even when used incorrectly.

Although I did not run an automated coverage report, the structure of the suite ensures high branch coverage and complete requirement coverage. Every path that could lead to acceptance or rejection of data was triggered intentionally. Constructors were tested for invalid inputs, setters were tested to confirm runtime validation, and services were tested for missing IDs and invalid updates. Both positive and negative outcomes were covered, which is the same principle an automated coverage report would measure.

Writing the tests strengthened my discipline for ensuring technical soundness. I made sure that invalid data could not silently slip through by using assertThrows to confirm that exceptions were raised where they should be. At the same time, I confirmed that service methods delegated validation to the model rather than duplicating it, which was reinforced with tests like: assertThrows(IllegalArgumentException.class, () -> svc.updatePhone("4", "bad"));.

That verified not only that the logic worked, but also that it was located in the right layer of the architecture. Efficiency in the JUnit code came from keeping each test self-contained while reusing object instances when appropriate. The helper method used for appointment dates also prevented repeated boilerplate code and reduced opportunities for inconsistency.

The most important testing techniques used in this project were unit testing, negative testing, and boundary testing. Unit testing allowed me to isolate each method and each validation rule so I could clearly identify where an issue would originate. Negative testing confirmed that invalid data is actively rejected, not just ignored. Boundary testing helped verify that the limits were enforced exactly as written in the requirements. These techniques were appropriate because the main risk in this project was incorrect or unvalidated data, not system integration or performance.

Techniques such as integration testing, system testing, user acceptance testing, and performance testing were not required here because the services run in memory and do not yet interact with external systems or real user workflows. If this application were to expand into a full mobile platform with a user interface or a persistent storage layer, I would introduce system-level and integration tests to verify workflows and user expectations. But at this stage, unit-level validation provides the greatest value.

The mindset I used while testing focused heavily on caution. I did not assume that simple code could not fail, and I treated setter validation with the same seriousness as constructor validation. This was especially important because a class can appear valid when first created, but later become invalid through poorly guarded updates. Thinking this way helped me avoid overlooking parts of the code that might otherwise seem harmless. I also tried to limit personal bias by writing negative tests early, since it is easy for a developer to focus only on proof of success. By treating invalid input as a core part of the design, I avoided assuming that code would only ever be used correctly.

This project also reinforced the importance of discipline in software development. Testing is not something that happens after code is finished; it is a major tool for shaping the design and preventing technical debt. If I had skipped even a few boundary or null checks, those issues could grow into failures that appear much later in development, where they would be more expensive and less obvious to fix. Treating every requirement as a contract helped keep the test suite organized, and it also created a foundation I can rely on as the code evolves. As I continue growing professionally, I plan to maintain requirement-driven testing and use comprehensive negative testing to protect data integrity and long-term maintainability.