Throughout the development of the Contact Service, Task Service, and Appointment Service for Project One, I adopted a structured, requirements-driven approach to unit testing. From the outset, I designed each test to reflect a specific customer requirement, validating field lengths, enforcing non-null constraints, and confirming correct behavior when inputs fall outside expected ranges. In the Contact Service, tests verified that contact IDs could not exceed ten characters, first and last names were capped at ten characters, and phone numbers were exactly ten digits. Each positive test was paired with a corresponding negative test—such as providing an eleven-character ID or a nine-digit phone number—to ensure the custom InvalidContactDataException was thrown when invalid data was supplied. In the Task and Appointment Services, the same pattern held: Task tests confirmed that names and descriptions met their twenty- and fifty-character limits, respectively, while Appointment tests enforced that dates were never in the past and descriptions remained under fifty characters. By aligning tests so directly with requirements, I ensured that each customer specification was both exercised and documented in code.

My testing approach mapped each requirement to one or more JUnit test methods. For example, the requirement that “the appointment date cannot be in the past” became a test that deliberately created a date before today and asserted an InvalidAppointmentDataException. Conversely, creating a date in the future and verifying successful object construction confirmed the positive scenario. This one-to-one correspondence between requirements and tests made the suite a living requirements document. It also facilitated easy maintenance: if a requirement changed, I simply updated or added the corresponding test.

To assess the overall quality of my JUnit tests, I used a coverage tool integrated in the IDE. Coverage metrics showed that each of the six source files—three model classes and three service classes—had at least eighty percent line coverage. Whenever any class fell below that threshold, I examined uncovered branches and wrote new tests to cover those conditions. For instance, if a setter's null check was not exercised, I added a test passing null to trigger that exception. This process, guided by coverage results, ensured that all logical branches were tested under both success and failure conditions.

Writing the tests taught me to balance thoroughness with readability. I began by listing valid and invalid inputs for each field, then translated those lists into test methods. Over time, I grouped related assertions—such as checking both maximum-length and just-over-limit values—into single boundary-test methods. This approach reduced duplication and kept the suite maintainable. Descriptive method names like testSetLastNameTooLongThrowsException() made it clear which requirement each test validated without requiring detailed comments.

Ensuring technical soundness meant using custom exception classes—such as DuplicateContactException—and matching those exceptions precisely in my tests. For example, in ContactServiceTest I wrote:

```
assertThrows(ContactService.DuplicateContactException.class,
    () -> service.addContact(existingContact));
```

This precise matching made failures obvious and guided me to write code that failed clearly when given bad inputs. It also followed best practices from the JUnit 5 User Guide,

which recommends clear, focused tests using its assertThrows API to verify exception handling(junit.org).

I improved efficiency by choosing HashMap to store objects in each service, ensuring constant-time lookup, addition, and deletion. My tests implicitly confirmed this choice by performing rapid add-and-remove loops without measurable slowdown, something that would have been problematic if I had used a list with linear search. In the test suite itself, I avoided redundant code by employing helper methods to create valid objects, keeping individual test methods concise.

Beyond unit, boundary, and negative tests, I reflected on other testing techniques not used in Project One. Integration testing would verify interactions among services, system testing would validate end-to-end behavior, acceptance testing would confirm business requirements with stakeholders, and performance testing would assess scalability under load. Each of these techniques plays a distinct role in a comprehensive testing strategy, but for isolated backend services, unit tests provided the fastest feedback loop and the clearest mapping to requirements.

Adopting the mindset of a software tester required caution and skepticism. I assumed every requirement could fail and wrote tests even for trivial getters and setters to guard against future changes. To limit bias, I wrote tests before implementing features in a mini test-driven-development style: tests failed first, then I wrote code to make them pass. This approach prevented confirmation bias, which might otherwise lead me to only see the scenarios for which I had already written code.

Finally, discipline in testing is vital to avoid technical debt. Cutting corners by skipping tests for edge cases can leave hidden defects that surface later. To prevent this, I enforced my

eighty-percent coverage standard and plan to integrate continuous integration tools that run the full test suite on every code push. Maintaining a comprehensive, up-to-date test suite will allow confident refactoring and extension of the codebase, ensuring quality over the long term.

Sources:

Stefan Bechtold, S. B. (n.d.). *JUnit 5 User Guide*. JUnit 5 user guide.

https://junit.org/junit5/docs/current/user-guide/?utm_source