Title: Project Two – Summary and Reflections on Unit Testing

Gary Travis

SNHU CS320

Summary

Unit testing approach

For this project, I used unit tests to verify the behavior of the three main service classes: ContactService, TaskService, and AppointmentService. Each test class targets core operations such as adding, deleting, and updating objects, as well as enforcing uniqueness and error handling, with each test method focused on a specific scenario (Qase, 2024).

In ContactServiceTest, I wrote tests to confirm that contacts can be added, deleted, and updated correctly across fields like first name, last name, phone number, and address. In TaskServiceTest, the tests verify adding and deleting tasks, updating all fields at once, and updating a single field while leaving others unchanged. In AppointmentServiceTest, the tests cover adding appointments with valid data, preventing duplicate IDs, and deleting existing appointments while throwing exceptions when operations are attempted on non-existent IDs.

Alignment to software requirements

My unit tests were designed directly from the requirements for each service. For contacts, the requirements state that the service must add new contacts, prevent duplicate contact IDs, allow updates to each field, and delete contacts by ID. That is reflected in tests like testAddContact, which asserts that a newly added contact can be retrieved with the expected first name, and testAddContactDuplicateId, which asserts that adding another contact with the same ID throws an IllegalArgumentException. The delete and update requirements are covered by tests such as

testDeleteContact and the update tests for first name, last name, phone number, and address, which verify that only the intended field changes.

The task requirements similarly call for adding tasks, preventing duplicate IDs, updating task details, and deleting tasks. TaskServiceTest enforces these through tests like testAddTask, testAddTaskDuplicateId, testDeleteTask, and testUpdateTaskAllFields. The testUpdateTaskSingleField method is especially aligned with partial-update behavior, because it updates only the task name while passing null for the description and then asserts that the original description is preserved. For appointments, the requirements to add appointments, prevent duplicate IDs, and support deletion are implemented through tests such as testAddAppointment, testAddAppointmentDuplicateId, testDeleteAppointment, and testDeleteAppointmentDoesNotExist.

Effectiveness and coverage

I used the built-in code coverage tools in the Eclipse IDE to judge how effective my JUnit tests were for each service class. Eclipse reported that I had full coverage of the service methods: 12 out of 12 methods covered in ContactService, 7 out of 7 in TaskService, and 4 out of 4 in AppointmentService. Every service method was executed by at least one test, including the methods that implement add, update, delete, and checks for invalid operations.

Coverage by itself does not prove that the code is perfect, but reaching 12/12, 7/7, and 4/4 methods covered means that no service method is completely untested, which reduces the chance of hidden logic going unnoticed (Qase, 2024). I also combined this structural coverage with behavior-focused tests that verify normal and exceptional cases. For example, in each service I have “happy path” tests (like testAddContact, testAddTask, and testAddAppointment) and

negative tests that assert exceptions for duplicate or non-existent IDs, which makes the suite reasonably effective at catching regressions.

Writing technically sound and efficient tests

I followed an arrange–act–assert pattern to keep my tests technically sound and easy to read (SemaphoreCI, 2025). In testAddContact, I arrange by creating a ContactService and a Contact with valid data, act by calling service.addContact(contact), and assert by checking that service.getContact("1") is not null and that its first name is "John". The same structure appears in tests like testAddTask and testAddAppointment, which improves consistency across the test suite.

I also wrote explicit tests for error conditions using assertThrows(IllegalArgumentException.class, ...), which is a best practice for verifying that the system fails in the expected way when given invalid input (Qase, 2024). Examples include duplicate-ID tests and the “does not exist” delete and update tests in all three services. The tests are efficient because they are small, focused, and free from external dependencies like files or databases; each test creates a fresh instance of the corresponding service so there is no shared state (Qase, 2024). In AppointmentServiceTest, I use a helper method to generate a future date so that all “valid” tests consistently use dates that satisfy the business rule about not scheduling in the past.

Reflection

Testing techniques employed and not used

The primary testing technique I employed was unit testing with JUnit. Each test class targets one service layer and checks that its public methods behave as expected under various conditions. Unit tests validate small units of functionality in isolation, which helps catch defects early and makes it easier to reason about individual components (Qase, 2024). I also used a black-box testing perspective by focusing on public APIs and requirements instead of internal implementation details, which makes the tests more resilient to refactoring as long as the external behavior stays the same.

Several other testing techniques were not directly used in this project. Integration testing, which verifies how multiple components interact, was not needed because the services are self-contained and use in-memory collections. System and acceptance testing, which validate complete end-to-end workflows and user requirements, were also outside the scope of this back-end focused project (Qase, 2024). I did not perform performance, load, or security testing, which are more appropriate for larger systems that are closer to deployment.

Practical uses and implications

Unit and black-box tests like the ones I wrote are especially useful in everyday development because they are fast and can be run frequently. As projects grow, running unit tests after each change helps catch regressions before they reach later stages of testing or production, and existing tests quickly reveal if changes to methods like updateTask or deleteContact break expected behavior (Qase, 2024).

Other techniques such as integration, system, and acceptance testing are better suited to verifying how the system behaves as a whole. In a more complete version of this application, integration tests might ensure that the services correctly interact with a database, while system and acceptance tests might validate that a user can successfully create, update, and delete contacts, tasks, and appointments through the interface. Choosing the right mix of techniques depends on project size, risk, and how the software is deployed (Qase, 2024).

Mindset: caution, bias, and discipline

Working through these tests required a cautious mindset because mistakes in the service logic could easily lead to bad data or unexpected behavior. If I did not carefully test duplicate IDs and "does not exist" cases across the services, they might allow conflicting records or silently ignore invalid operations. Verifying partial updates, not just full replacements, also reminded me that small changes in one part of the code can have a ripple effect on the rest of the system.

To limit bias, I intentionally wrote both positive and negative tests instead of only checking scenarios I expected to succeed. For every operation, I asked what would happen if it were used incorrectly and added tests around those cases, such as invalid IDs and partial updates. Thinking like a different or even "hostile" user helped me challenge my assumptions and design more realistic test scenarios.

This project also reinforced the importance of discipline and a long-term commitment to quality. It can be tempting to skip tests for "simple" methods or edge cases to move faster, but that creates technical debt, the implied cost of rework caused by choosing quick solutions instead of thorough ones (IBM, 2025). By writing systematic tests for adding, deleting, updating, and handling invalid operations in each service, and by ensuring full method coverage in Eclipse, I

invested in a foundation that will make future changes safer. Going forward, I plan to avoid technical debt by writing tests alongside new features, keeping them updated when requirements change, and using patterns like arrange–act–assert to keep tests clear and maintainable (SemaphoreCI, 2025; IBM, 2025).

References

IBM. (2025, March 26). *What is technical debt?* IBM Think. https://www.ibm.com/think/topics/technical-debt

Qase. (2024, September 11). *Unit testing: The what, why, and how.* Qase Blog. https://qase.io/blog/unit-testing/

SemaphoreCI. (2025, January 16). *The arrange, act, and assert (AAA) pattern in unit test automation.* Semaphore Blog. https://semaphore.io/blog/aaa-pattern-test-automation