Ryan Orton

CS320

08/15/25

**Project Two**

**Summary**

For the Contact Service, I used a positive and negative path testing approach to verify that all CRUD operations worked as expected and that invalid inputs triggered the correct exceptions. For example, I tested valid contact creation with all fields populated and invalid cases with null or overlength strings.

For the Task Service, I applied boundary value testing to verify that constraints such as maximum task length and due date formats were enforced. I also included unit tests for updating and deleting tasks, ensuring that data persisted as expected.

For the Appointment Service, I used equivalence partitioning to group valid and invalid appointment date/time combinations. I tested valid scheduling within working hours and invalid cases where appointments overlapped or were scheduled in the past.

My testing approach was directly tied to the requirements provided for each service. For example, the Contact Service requirement that first and last names be no longer than 10 characters led to the following unit test:

assertThrows(IllegalArgumentException.class, () -> {

new Contact("12345", "FirstnameTooLong", "Smith", "1234567890", "123 Street");

});

This ensured compliance with input length constraints defined in the functional requirements.

JUnit coverage reports showed over 90% line coverage for all three services, indicating comprehensive testing. This high coverage, combined with the variety of test cases, confirmed the effectiveness of the tests. Coverage metrics included both branches of conditional logic, ensuring that exceptions and success paths were exercised.

Writing the JUnit tests required careful attention to detail, especially when simulating both success and failure cases. For example, in the Task Service tests, I ensured that adding a task beyond the allowed length triggered the proper exception:

assertThrows(IllegalArgumentException.class, () -> {

new Task("1", "A".repeat(51), "Description");

});

This demonstrated that I accounted for edge cases and ensured technical soundness by aligning the tests with constructor and mutator validation logic.

To keep the tests efficient, I reused setup logic with @BeforeEach to initialize service objects rather than creating them in every test method:

@BeforeEach

void setUp() {

contactService = new ContactService();

}

This reduced repetition, made the code easier to maintain, and improved readability.

**Reflection**  
I primarily employed the following testing techniques:

* **Black Box Testing** – focusing on inputs/outputs without inspecting internal logic, which matched the role of a service API tester.
* **Boundary Value Testing** – testing inputs at the edge of allowed ranges, ensuring constraint enforcement.
* **Equivalence Partitioning** – grouping test cases to reduce redundancy while covering logical input categories.

Some testing techniques that I did not use were:

* **White Box Testing** – examining internal logic; not used because requirements emphasized functional behavior over implementation details.
* **Exploratory Testing** – informal, unscripted testing; not used due to the formal unit test deliverable requirements.
* **Stress/Load Testing** – measuring performance under heavy usage; not applicable for these small-scale services.

Boundary value and equivalence partitioning are well-suited for systems with strict validation rules, such as banking apps or healthcare scheduling systems. White box testing would be more critical in safety-critical or performance-sensitive systems like aviation control software.

I approached the project with a cautious and methodical mindset, double-checking that each test matched a specific requirement. This was important because small errors in the services could cascade into major application failures, especially since Contact, Task, and Appointment Services interacted indirectly in the larger application.

For example, I made sure that deleting a contact that didn’t exist raised an exception rather than failing silently, because this behavior could have downstream effects on appointment records tied to that contact.

Bias was reduced by writing tests that deliberately tried to break my own code. For instance, in the Appointment Service, I intentionally scheduled appointments with invalid past dates to confirm that my constructor validation caught them. If I only wrote tests I expected to pass, I could have overlooked error handling flaws.

Cutting corners in testing might save time in the short term but creates technical debt. I avoided this by writing complete test suites before finalizing the services. For example, I resisted skipping null-input tests even though they were repetitive, because null handling is a common source of runtime errors. In professional practice, I plan to maintain this discipline by integrating automated unit tests into CI/CD pipelines so that regressions are caught early.