Valeria Duharte

Project Two

CS320

During the development of the mobile application for the customer, I designed and implemented unit tests for three core services: Contact Service, Task Service, and Appointment Service. Each service required specific validation and manipulation functionalities that I rigorously verified using the JUnit 5 testing framework to ensure robustness, correctness, and full compliance with the predefined software requirements.

**Contact Service:**  
My unit testing approach for the Contact Service focused on validating the creation, updating, and deletion of contact entries. The ContactTest class specifically enforced constraints such as maximum string length, non-null fields, and correct phone number format. For example, the method testConstructorInvalidPhone() verified that improperly formatted phone numbers triggered exceptions by using the assertThrows(IllegalArgumentException.class, ...) construct. Additionally, update methods were thoroughly tested to ensure data integrity when modifying first names, last names, addresses, or phone numbers.

**Task Service:**  
For the Task Service, testing centered on ensuring that tasks were created with valid IDs, names, and descriptions, according to length and null constraints. The TaskTest class validated scenarios such as testInvalidConstructorTooLongDescription(), ensuring that the constructor correctly rejected descriptions exceeding 50 characters. Update methods such as setName and setDescription were similarly verified to reject null or overly long inputs. The TaskServiceTest class ensured that tasks could be accurately added, updated, and deleted within the collection through methods such as testAddTask().

**Appointment Service:**  
The Appointment Service required handling date-specific logic to prevent null or past appointment dates from being entered. Within the AppointmentTest class, testDateInPast() confirmed that creating appointments with past dates resulted in appropriate exceptions. String length constraints for descriptions were also verified. The AppointmentServiceTest validated the addition and removal of appointments, ensuring IDs remained unique and dynamically retrieved within the testing process.

My unit testing approach was tightly aligned with the software requirements specified for each feature. The requirement that Contact IDs remain immutable, phone numbers be exactly 10 digits, and address lengths not exceed 30 characters were explicitly tested and verified. For the Task and Appointment Services, description length constraints, null handling, and ID uniqueness were validated through comprehensive unit tests such as TaskTest.validConstructor() and AppointmentTest.testValidAppointment(). The Appointment Service's requirement that appointment dates cannot be in the past was specifically tested using the utility method getPastDate(int daysAgo) within AppointmentTest.testDateInPast().

I am confident in the overall quality of my JUnit tests due to their comprehensive coverage and effectiveness in validating both positive and negative scenarios for every public method. All constructors and setters underwent testing under standard and edge conditions, ensuring resilience against invalid data inputs. My tests adhered to JUnit best practices—remaining atomic, independent, and repeatable. The inclusion of edge cases, such as null values and boundary-length checks, further bolstered the reliability and completeness of the test suite.

To ensure technical soundness, I explicitly tested exception handling and input validation mechanisms. Efficiency was maintained by isolating tests to single-purpose methods and avoiding redundant code. Utility methods such as getPastDate(int daysAgo) in AppointmentTest minimized code repetition and improved maintainability.

The following software testing techniques were employed in this project:

1. Boundary Value Testing: Minimum and maximum constraints were validated, such as enforcing the exact 50-character limit for descriptions in the Appointment Service.
2. Negative Testing: Invalid inputs, such as null values or overly long strings, were used to provoke exceptions and validate error handling.
3. Equivalence Partitioning: Input domains were divided into valid and invalid classes, such as phone numbers with exactly 10 digits versus incorrect lengths.

Testing Techniques Not Used

1. Integration Testing: I did not perform tests combining the Contact, Task, and Appointment Services to verify their interactions as a unified system. Integration testing identifies interface-related issues but was beyond the scope of this project’s unit-level focus.
2. Performance Testing: I did not evaluate execution time or resource usage under stress conditions. Performance testing is essential for high-load applications but was not pertinent to this development phase.

Boundary Value and Negative Testing are crucial for validating individual method robustness and ensuring that edge cases are handled gracefully in unit-level testing. Equivalence Partitioning helps optimize test coverage without redundant cases. In contrast, Integration Testing and Performance Testing are better suited for later stages of development, particularly in complex or high-traffic applications.

Throughout this project, I maintained a cautious and skeptical mindset, if invalid or unexpected inputs could occur. This proactive attitude led to thorough validation, such as deliberately simulating past dates in AppointmentTest.testDateInPast() to ensure exception handling. Appreciating the complexity and interdependencies of each service class ensured that state management and data integrity remained intact, avoiding issues like ID overwriting or invalid object states. To limit bias during code review and testing, I consciously approached the code from an end-user perspective, expecting misuse or unforeseen input scenarios. This unbiased mindset prevented assumptions of perfect input and improved test completeness. As a developer, self-testing poses risks of oversight due to familiarity; thus, maintaining an outsider's critical eye was essential.

Discipline and a quality-focused approach are vital in reducing long-term maintenance costs and avoiding technical debt. Cutting corners in validation or testing could result in hidden bugs and expensive refactoring later. Moving forward, I plan to implement continuous testing alongside feature development, avoid hardcoded assumptions, and regularly refactor test suites to align with evolving codebases. This project enhanced my ability to design and execute effective unit tests that ensure both functional correctness and robustness against unexpected scenarios.