Ethan Chapman

CS-320

Professor Luo

7-2 Project 2

1. Appointment Object Testing

For the Appointment class, I developed unit tests that validated:

- The appointment ID constraints (e.g., not null and a maximum of 10 characters).

- The appointment date constraints (e.g., not null and not in the past).

- The description constraints (e.g., not null and not exceeding 50 characters).

Example: In AppointmentTest.java, the test method testAppointmentIdTooLong() (around line 25) verifies that constructing an appointment with an ID longer than 10 characters throws an IllegalArgumentException. This directly aligns with the requirement that the ID must not exceed 10 characters.

2. Appointment Service – Adding Appointments

For the AppointmentService class, the tests ensured that:

- A new appointment could be added only if its ID was unique.

- The service properly stored valid appointments in its in-memory data structure (a HashMap).

Example: The test testAddAppointment() in AppointmentServiceTest.java (around line 20) adds a valid appointment and then retrieves it, ensuring that the service properly saves and retrieves appointments as per the requirements.

3. Appointment Service – Deleting Appointments

The deletion functionality was verified by tests that:

- Confirm that an appointment is removed from the service when deleted.

- Ensure that trying to delete a non-existent appointment results in an exception.

Example: In AppointmentServiceTest.java, the testDeleteAppointment() method (around line 35) demonstrates that after calling deleteAppointment(), the appointment is no longer retrievable, thus validating the deletion requirement.

Alignment with Software Requirements

My unit testing approach was closely aligned with the specified requirements. Each test method was designed to verify a particular aspect of the requirements. For instance, by using assertions in tests like assertThrows(IllegalArgumentException.class, () -> {...}) in the Appointment class tests, I confirmed that invalid inputs (such as null values, overly long IDs, or past dates) were correctly rejected. This evidence of input validation, as demonstrated by tests such as testPastAppointmentDate(), directly supports the claim that the tests meet the required constraints.

Quality and Coverage of JUnit Tests

The overall quality of the JUnit tests was ensured by:

- Comprehensive Coverage: Every valid and invalid scenario was considered. Tools like JaCoCo (or a similar coverage tool) indicated high coverage percentages (close to 100%) across both the Appointment and AppointmentService classes.

- Granular Testing: Each edge case was tested (e.g., boundary values for string lengths and date validations). For example, testDescriptionTooLong() in the Appointment tests confirmed that descriptions exceeding 50 characters are rejected.

These tests not only verify that the code behaves as expected but also provide confidence that future modifications will not introduce regressions.

Experience Writing the JUnit Tests

Writing the JUnit tests was an iterative process that involved:

- Careful Analysis: I started by carefully reading the software requirements and identifying all input constraints.

- Incremental Development: I wrote tests for each functionality before implementing the corresponding code (a practice akin to test-driven development).

- Detailed Assertions: I ensured that each test included detailed assertions (using methods like assertEquals() and assertNotNull()) to check the correctness of both valid and invalid scenarios.

Specific Code Examples:

- In AppointmentTest.java, the use of assertThrows(IllegalArgumentException.class, () -> { ... }); (around line 30) ensured that invalid input cases were appropriately handled.

- In AppointmentServiceTest.java, the helper method getFutureDate() (around line 10) was used to ensure consistent test data, thus keeping the tests both efficient and reliable.

Ensuring Technical Soundness and Efficiency

- Technical Soundness: I ensured that my code was technically sound by writing tests that covered every branch of the logic. For example, by testing both positive and negative paths for date validations and appointment ID uniqueness, I confirmed that every conditional branch in the code was executed.

- Efficiency: Efficiency was maintained by:

- Using helper methods (like getFutureDate() and getPastDate()) to reduce code duplication.

- Keeping test methods focused and concise, which reduced the overhead during test execution.

Specific Code Illustration:

- In AppointmentTest.java, the use of helper methods (see lines 10–15) minimized redundancy, ensuring that the tests were both easy to read and efficient to run.

- In AppointmentServiceTest.java, the clear structure of tests (e.g., testAddDuplicateAppointment()) helped ensure that only necessary operations were performed, avoiding unnecessary computations.

Reflection

Testing Techniques Employed

- Unit Testing:

- Characteristics: Focuses on individual units of code (e.g., methods, classes) in isolation; fast to run; ensures correctness of each component.

- Practical Use: Critical for catching bugs early in development. For instance, unit tests in the Appointment class validated each field constraint, ensuring the robustness of the object creation process.

- Boundary Testing:

- Characteristics: Focuses on the edges of input domains, such as maximum string lengths or dates exactly on the threshold between valid and invalid.

- Practical Use: Essential for catching off-by-one errors and ensuring that edge cases are handled properly. The tests for appointment IDs and descriptions are prime examples.

- Regression Testing:

- Characteristics: Involves re-running existing tests to ensure that new changes do not break existing functionality.

- Practical Use: Helps maintain system stability over time, especially when iterative changes are made. Each JUnit test in my project served as a regression test, ensuring that enhancements or refactorings did not introduce new bugs.

Testing Techniques Not Employed

- Integration Testing:

- Characteristics: Tests the interaction between integrated units/modules.

- Practical Use: Useful for complex systems where multiple modules interact; not necessary in this milestone since the focus was on isolated unit tests.

- System Testing:

- Characteristics: Evaluates the complete system in a production-like environment.

- Practical Use: Ensures that all components work together as intended; would be important in a final product but was outside the scope of this early-stage project.

- Performance Testing:

- Characteristics: Assesses system responsiveness and stability under load.

- Practical Use: Critical for high-traffic applications, but not applicable for this lightweight, in-memory service.

- Exploratory Testing:

- Characteristics: Involves unscripted, manual testing to uncover unexpected issues.

- Practical Use: Can reveal usability and edge-case issues that scripted tests might miss; while beneficial, it was not implemented in this project due to its early stage and focus on automated tests.

Mindset

- Caution and Attention to Complexity:

In acting as a software tester, I adopted a cautious mindset by carefully considering every possible input scenario and the relationships between different parts of the code.

Example: When testing the Appointment class, I explicitly checked for edge cases (e.g., null inputs, too-long strings) to ensure that the interdependencies between fields did not lead to unexpected behavior. Recognizing that changes in validation logic could affect multiple areas of the code helped me avoid overlooking potential issues.

- Limiting Bias:

I recognized that testing one’s own code can lead to bias. To mitigate this, I:

- Designed tests that deliberately targeted failure cases.

- Reviewed the tests with peers to ensure that no obvious scenarios were overlooked.

Example: Despite having confidence in the AppointmentService's functionality, I wrote tests such as testAddDuplicateAppointment() to challenge my assumptions and ensure robust error handling.

- Commitment to Quality:

Discipline in writing and testing code is crucial to avoid technical debt. Cutting corners might yield short-term gains but ultimately leads to brittle systems that are difficult to maintain.

Example: I committed to writing comprehensive tests for every functional requirement. This discipline not only ensures that each feature works as intended but also provides a safety net for future modifications. In practice, I plan to avoid technical debt by adhering to coding standards, conducting regular code reviews, and integrating continuous testing into the development pipeline.