Unit Testing Approach

For the Contact, Task, and Appointment services, I followed a structured test-driven development (TDD) approach; creating unit tests prior to writing the methods. This made sure that the implementation met the requirements and worked under both normal and edge cases. The ContactService was tested to confirm that no duplicate IDs could be created, which aligns with the requirement that each contact must have a unique ID.

My approach was aligned with software requirements as I wrote JUnit tests that validated logic constraints like string length, valid future dates, and ID uniqueness. In the Appointment class, I included a test to ensure that appointments cannot be created with past dates, which enforces the requirement that appointments must be scheduled in a future date:

void testConstructor\_withPastDate\_shouldThrowException() {

Date pastDate = new Date(System.currentTimeMillis() - 10000);

assertThrows(IllegalArgumentException.class, () -> {

new Appointment("A124", pastDate, "Too late");

});

}

This test makes sure input validation aligns with the requirement that the appointment date must be scheduled in the future and not a past date.

JUnit Test Effectiveness

The effectiveness of my JUnit tests shows in the way it validates both typical and edge-case behaviors on all three services. Each method in the ContactService, AppointmentService, and TaskService had corresponding test cases that handle both successful execution and failures.

For example, in the ContactServiceTest, the testUpdateContactFields() method confirms that updates to all contact fields; first name, last name, phone number, and address, are processed correctly. The test updates each field and verifies the changes using assertions, ensuring that the update methods function as expected. In the AppointmentTest, the testConstructor\_withPastDate\_shouldThrowException() method shows that the system enforces the requirement that appointments cannot be scheduled in the past, protecting integrity.

Experience Writing JUnit Tests

To ensure my code was technically sound, I followed the Arrange-Act-Assert (AAA) pattern, which helped maintain clarity and consistency in my tests.

For example, in the testUpdateContactFields() method:

service.updateFirstName("001", "Alex");

service.updateLastName("001", "Johnson");

service.updatePhone("001", "0987654321");

service.updateAddress("001", "789 Sunset Blvd");

Contact updated = service.getContact("001");

assertEquals("Alex", updated.getFirstName());

assertEquals("Johnson", updated.getLastName());

assertEquals("0987654321", updated.getPhone());

assertEquals("789 Sunset Blvd", updated.getAddress());

Testing Techniques

I used black box testing and boundary value analysis. Black box testing let me to validate output without needing knowledge of the methods and boundaries; testing helped catch edge cases like input lengths.

Techniques I didn't use include white box testing and equivalence partitioning. White box testing has a focus’ on structures and paths; could have been useful for branch level coverage. Equivalence partitioning might simplify test cases by grouping inputs that are similar, which can be beneficial in reducing any potential redundancy.

In real world applications, black box testing suits validation needs, while white box testing is important in low level libraries. Equivalence partitioning is ideal in UI testing, where user input is predictable.

Mindset

Working on this project, I tried to have a cautious and analytical mindset. I reviewed each method if defects could happen under rare conditions. I tested the updateAppointment() method for invalid dates and ensured it handled null input, which is often overlooked.

Appreciating code complexity helped prevent issues. Much like the Ariane 5 disaster where a data conversion error from a 64-bit float to a 16-bit signed integer caused a catastrophic failure (Arnold, n.d.), I ensured type conversions and input validation was in my service classes.

Limiting Bias

To limit bias, I tried to act as the developer and end user tester. I wrote my JUnit tests a day after implementing the logic, giving me time to think about the method behavior without assuming it was correct. Bias can be a concern when developers test their own code, as seen in the Ariane 5 case, overconfidence in reused software led to insufficient real world simulation testing (Coder Academy, 2020).

Discipline and Quality

Discipline in software testing is a non-negotiable. Cutting corners in testing or assuming correctness leads to potential technical issues down the road.

To avoid this, I will:

* Always write tests before implementation (TDD)
* Integrate code coverage tools like JaCoCo.
* Use peer reviews or pair testing to uncover blind spots

Much like the Ariane 5 failure that cost millions and delayed space programs due to error, even small bugs in enterprise systems can be extremely costly.

**References**

Arnold, J. (n.d.). Ariane 5 Flight 501 Failure. University of Minnesota. https://www-users.cse.umn.edu/~arnold/disasters/ariane.html

Coder Academy. (2020). When Coding Goes Wrong. Medium. https://medium.com/@coderacademy/when-coding-goes-wrong-e46d84c6565f