Frangerwy Brito Feliz

CS 320

Mr. Kraya

23Jun2024

**Summary and Reflections Report**

**Testing Strategy Overview**

Throughout the development of our application's model and service classes—Contact Service, Task Service, and Appointment Service—I adopted two distinct perspectives. One was that of a meticulous software engineer committed to fulfilling precise requirements. The other was that of a skeptical analyst, intent on identifying potential vulnerabilities as if trying to exploit the system maliciously. This dual approach allowed me to design comprehensive tests, ensuring the robustness and reliability of our services.

**1. Contact Service**

The primary focus was vigorous data handling for adding, deleting, and editing contacts. I concentrated on testing against null values and incorrect data formats to verify boundary conditions and error handling.

The ContactService was required to manage contact data within specific constraints. For example, ensuring the proper handling of the first name field was crucial:

test.addContact(firstName, lastName, phoneNumber, address);

assertEquals(firstName, ContactService.contactList.get(0).getFirstName());

This test verifies that the first name is correctly captured and stored, adhering to the non-null constraint specified in the requirements.

**2. Task Service**

The goal was to validate the functionality for adding, updating, and removing tasks, ensuring compliance with the defined specifications, such as uniqueness and length restrictions of task attributes.

Tasks are required to have a unique identifier, name, and description, each with specific constraints:

tempTask.addUniqueTask(fullName, description);

assertTrue(TaskService.tasks.containsKey(id));

This test ensures each task is properly identified and stored, conforming to the requirements of task uniqueness and data integrity.

**3. Appointment Service**

Tests for the AppointmentService focused on adding appointments, validating against incorrect dates, and ensuring descriptions were neither null nor improperly formatted. Given the constraints around appointment scheduling, it was necessary to ensure the system handled date and description fields correctly:

assertThrows(IllegalArgumentException.class, () -> {

tempAppt.addUniqueAppointment(goodDate, null);

});

This scenario tests the system's compliance with the requirement to reject null descriptions, crucial for maintaining data consistency and integrity.

**Success of JUnit Tests**  
Rather than aiming for 100% coverage, my approach focused on covering significant functionalities, particularly getters, setters, and critical constructor behavior. Complete coverage in these areas is fundamental to the stability and functionality of our application. For instance:

assertEquals(2, ContactService.contactList.size());

assertFalse(ContactService.contactList.stream().anyMatch(c -> c.getContactID().equals("1")));

These tests ensure both the functionality and side effects of operations are as expected, demonstrating thoroughness and attention to detail.

**Experience Writing JUnit Tests**  
I emphasized not just functional correctness but also the robustness of the tests:

@AfterEach

void tearDown() {

ContactService.contactList.clear();

}

This cleanup ensures each test is independent and the test environment is consistently controlled.  
I streamlined testing by grouping related assertions, minimizing redundant operations, and maximizing test efficiency:

test.addContact(firstName, lastName, phoneNumber, address);

assertEquals(3, ContactService.contactList.size());

This test not only validates the addition of a contact but also checks the list's size, effectively verifying multiple aspects of functionality.

**Testing Techniques Employed**  
I utilized both unit testing and static testing techniques. Static testing was particularly useful in reviewing the code against specifications to catch bugs early. My unit testing was rigorously designed to challenge the implementation by testing boundary conditions and error handling.  
While I focused on white-box testing, integration and system testing were not within the project's scope. These would be essential as the application grows and components need to interact more intensively.  
Adopting a mindset of trying to 'break' the code, I approached testing from the perspective of proving the code wrong, not just confirming it works. This was crucial in uncovering hidden issues and ensuring the code could handle unexpected or incorrect inputs gracefully.  
By acting both as the developer and an external critic, I mitigated the natural bias of assuming my code was faultless. This dual-role approach facilitated a more objective evaluation of code quality.

**Commitment to Quality**  
A disciplined approach to testing, combined with a commitment to not cutting corners, underpins the overall reliability of our software. Comprehensive test coverage and high standards in testing practices help mitigate technical debt and promote sustainability in software projects.  
The ongoing process of testing, coupled with regular code reviews and adherence to best practices, is vital in maintaining high-quality standards. This is crucial not only for the current project's success but also for future scalability and maintainability.

**Conclusion**

The structured approach to JUnit testing undertaken in this project not only ensured compliance with requirements but also established a robust framework for ongoing software development and maintenance. Each test crafted was a step towards validating the application's functionality comprehensively, highlighting the critical role of meticulous testing in software development.