Devin Wheeler

06/23/2024

Project 2

7-2 Project Two Submission

The unit testing approach for the Contact, Task, and Appointment Services focused on validating the correctness of object attributes and handling invalid inputs. Tests were designed to ensure that objects were created correctly, and exceptions were thrown for invalid data, such as null values, long strings, and duplicate IDs. This approach ensured the robustness of each service by covering various edge cases and ensuring that constraints were enforced.

@Test

public void testSetDescriptionWithInvalidValue() {

Task task = new Task("12345", "Test Task", "This is a test task description.");

assertThrows(IllegalArgumentException.class, () -> {

task.setDescription("This description is way too long and exceeds the fifty characters limit.");

});

}

Alignment to Software Requirements: The unit testing approach was closely aligned with the software requirements, as each test was designed to validate the specific constraints and conditions outlined in the requirements. For example, the requirement that contact IDs must be unique was tested by attempting to add duplicate contacts and verifying that an exception was thrown. And this was the case for all tests.

Overall Quality of JUnit Tests: The effectiveness of the JUnit tests was supported by a high coverage percentage, in this case 100%, indicating that the tests covered a wide range of scenarios and edge cases. This thorough testing ensured that the core functionality of the services was validated, and potential issues were identified early. Writing the JUnit tests was a tedious process that involved thinking of potential issues and edge cases. Ensuring that the code was technically sound required careful planning and thorough review of the requirements. The use of assertThrows to validate exception handling was very helpful.

@Test

public void testUpdateContactAddressInvalid() {

assertThrows(IllegalArgumentException.class, () -> {

service.updateContactAddress(contact.getContactId(), "456 Avenue with a very long address that exceeds the limit");

});

assertThrows(IllegalArgumentException.class, () -> {

service.updateContactAddress(contact.getContactId(), null);

});

}

Efficiency of Code: Efficiency was achieved by writing concise and focused tests that targeted specific functionality. Reusing setup methods to initialize common test data helped reduce redundancy and improve test maintainability.

@BeforeEach

public void setUp() {

service = new ContactService();

contact = new Contact("12345", "John", "Doe", "1234567890", "123 Street");

service.addContact(contact.getContactId(), contact.getFirstName(), contact.getLastName(), contact.getPhone(), contact.getAddress());

}

Testing Techniques: The primary testing techniques used were unit testing and boundary testing. Unit testing involved testing individual methods and classes to ensure they functioned correctly. Boundary testing focused on testing the limits of input values to verify that the system handled edge cases appropriately. Integration testing, which tests the interactions between different modules, was not used in this project. Integration testing could be used for verifying that the contact, task, and appointment services work together seamlessly. Additionally, stress testing, which assesses the system's performance under heavy load, was not conducted but could provide data into the system's scalability. Unit testing is essential for verifying the effectiveness of individual components and is useful for identifying issues early in the development process. Integration testing is critical for ensuring that different parts of the system work together as expected, which is particularly important in complex applications. Stress testing helps ensure that the system can handle high volumes of data and user interactions, which is vital for applications expected to scale.

Mindset: Adopting a cautious and detail-oriented mindset was crucial for this project. It was important to appreciate the complexity and interrelationships of the code to ensure comprehensive testing. For example, ensuring that adding a duplicate contact ID throws an exception required understanding how the contact service manages unique IDs. To limit bias, tests were designed to be objective and based on clearly defined requirements. Reviewing the code from a tester's perspective, separate from the developer's viewpoint, helped identify potential issues without preconceived notions. This approach is critical to avoid overlooking potential bugs or missing easy to test steps. Maintaining a disciplined commitment to quality is essential in software engineering. Cutting corners in testing can lead to undetected bugs and technical debt, which can be costly to fix later. To avoid technical debt, it is crucial to write comprehensive tests, perform regular code reviews, and refactor code when necessary. By following these practices, software engineers can ensure that their code remains maintainable and reliable.

Reference:

Fowler, M. (2006). Continuous Integration. Retrieved from https://martinfowler.com/articles/continuousIntegration.html