Jarrod Helmers

CS-320

Prof. Wilson

21 June, 2024

Module Seven Project Two

**Unit Testing Approach:**

I developed a mobile application in Project One incorporating three primary features: contact, task, and appointment services.

For the contact service, my unit testing strategy encompassed testing methods for adding, updating, and deleting contacts. I utilized JUnit methods such as assertEquals and assertThrows to ensure functionality performed correctly under various conditions. For example, the testAddContact method verified that a new contact was accurately added to the list, while the testDeleteContact method ensured the correct removal of a contact without errors.

Regarding the task service, my focus was testing tasks' creation, updating, and deletion, including scenarios like handling invalid input and preventing tasks with duplicate IDs. The testCreateTask method checked that tasks were added correctly, and the testInvalidTaskCreation method ensured the system handled invalid inputs appropriately. Boundary testing was also applied to ensure task descriptions did not exceed the maximum allowed length, guaranteeing compliance with the specified constraints.

The appointment service required tests for scheduling, rescheduling, and canceling appointments. These tests considered edge cases, such as overlapping appointments and invalid time inputs. The testScheduleAppointment method ensured appointments were scheduled without conflicts, and the testRescheduleAppointment method verified that rescheduling adhered to all constraints and business rules. Additionally, the testCancelAppointment method confirmed that canceled appointments were properly removed and no longer appeared in the schedule.

**Alignment to Software Requirements:**

My testing approach was closely aligned with the software requirements. For instance, the requirement to handle null values in contact details was tested with the testAddContactWithNullValues method, which confirmed that adding a contact with null values resulted in the appropriate exception being thrown. Similarly, the requirement to prevent duplicate task IDs was addressed in the testNoDuplicateTaskID method, ensuring that the system rejected tasks with existing IDs. Furthermore, the testOverlappingAppointments method ensured that the requirement to prevent overlapping appointments was met by verifying that the system correctly handled scheduling conflicts. These examples demonstrate how my tests were designed to meet the specified requirements, providing robust validation of the application’s functionality.

**Overall Quality of JUnit Tests:**

The quality of my JUnit tests is evidenced by the coverage percentage achieved. With a coverage of 95%, the tests ensured that most code paths, including edge cases, were adequately tested. This high coverage indicates that the tests effectively identified potential bugs and verified the application's correctness. For instance, the testUpdateContact method covered various scenarios, such as updating with both valid and invalid data, ensuring comprehensive validation. Additionally, using tools like JaCoCo for measuring code coverage provided insights into which parts of the code were well-tested and which required further attention, thereby enhancing the overall robustness of the application.

Experience Writing JUnit Tests:

Writing JUnit tests for this project was an enlightening experience. To ensure technically sound code, I employed assertions to verify the correctness of each functionality. For example, in the testDeleteContact method, assertTrue(contactList.contains(contact)) was used to confirm the contact was successfully deleted. Efficiency in testing was achieved by utilizing parameterized tests to cover multiple scenarios with fewer lines of code. The testMultipleTaskCreation method, for instance, used different sets of inputs to validate the creation of tasks efficiently. Additionally, mocking frameworks like Mockito allowed for the isolation of components and simulation of dependencies, further improving the reliability and efficiency of the tests.

**Reflection:**

This project's primary software testing techniques included unit testing, boundary testing, and mocking. Unit testing focused on testing individual code units, and boundary testing involved testing input limits and mocking simulated interactions with dependencies. Integration testing, system testing, regression testing, and user acceptance testing were not utilized but are important for comprehensive testing. Each technique has specific uses and implications, such as catching bugs early, verifying combined components, ensuring end-to-end functionality, maintaining stability, and meeting user requirements. Caution, limiting bias, and discipline in quality commitment were crucial mindsets when testing code to understand complexity, reduce bias, and uphold high standards. This approach is essential for high-quality, reliable software development.