Gage Sovey

CS-320: Project 2

Prof. Cross

10/15/2024

1. **Summary**

**Unit Testing Approach:**

For ContactService, TaskService, and AppointmentService, I used a structured unit testing approach with JUnit. Each service was thoroughly tested, focusing on positive cases and boundary conditions like maximum field length and null inputs. For example, testSetTaskName() in TaskService ensured task names longer than 20 characters were truncated correctly, while testAppointmentDateInPastThrowsException() confirmed appointments couldn’t be scheduled in the past. Modular testing streamlined testing across all services.

**Alignment with Requirements:**

The tests closely aligned with project requirements, ensuring all constraints were validated, such as limiting task descriptions to 50 characters and generating unique IDs for tasks and contacts. Each test verified both valid and invalid inputs. For instance, testAddTaskWithUniqueID() ensured proper functionality, while testDeleteNonExistentContact() checked for proper handling of invalid inputs without throwing exceptions. The alignment is reflected in the high coverage percentages, such as 84% for AppointmentService and 90.5% for TaskService.

**JUnit Test Quality:**

The high code coverage indicates thorough testing of core logic and edge cases. For example, testUpdateFirstNameByID() verified the successful updating of first names, while also testing the system’s ability to handle null inputs. By incorporating both setter and constructor tests, I ensured the system was well-prepared to handle unexpected behavior, making the tests highly reliable.

**Experience Writing JUnit Tests:**

Writing JUnit tests required planning to ensure positive and negative scenarios were tested. Tests like testDeleteNonExistentContact() confirmed the system’s ability to handle invalid inputs without crashing. Efficiency was achieved by using @BeforeEach methods for service initialization, reducing redundant code across tests. Assertions such as assertThrows() for exception handling allowed for efficient verification of the correct handling of invalid data inputs.

**Technical Soundness:**

I ensured technical soundness by testing validation rules, for example, assertThrows(IllegalArgumentException.class, () -> appointment.setAppointmentDate(pastDate)). This validated the system’s ability to handle exceptions. Another example is assertTrue(contact.getAddress().contains("NULL")), which confirmed that default values were properly set when invalid inputs were encountered.

**Efficiency:**

Efficiency was maintained through reusable logic in constructors and setters. Tests like testSetTaskDesc() efficiently checked boundary conditions, such as ensuring descriptions longer than 50 characters were truncated without redundantly testing every possible input.

1. **Reflection**

**Testing Techniques Used:**

I employed black-box testing to verify service outputs against requirements without looking at internal logic, while white-box testing was used to ensure internal logic like addTask() and addAppointment() worked as expected. Boundary value analysis ensured that fields like task name lengths were properly validated at their limits, such as testing a name with exactly 20 characters.

**Testing Techniques Not Used:**

I did not use integration testing, which would have been useful for ensuring the interaction between services worked as expected. Regression testing, another technique I didn’t use, would have been valuable for ensuring that future changes didn’t introduce new bugs. Additionally, exploratory testing was not utilized, which involves unscripted testing to find unexpected issues.

**Practical Implications:**

Integration testing is crucial in complex systems with interconnected components to ensure that individual units communicate correctly. Exploratory testing is beneficial in agile environments with rapidly changing requirements. Regression testing is essential in long-term projects to ensure that new code does not break existing functionality.

**Mindset:**

As both a developer and tester, I adopted a cautious mindset, especially when testing the interrelationships between services. For example, testing update methods like testUpdateTaskDescByID() ensured that small bugs didn’t escalate into bigger issues. Testing for edge cases, like null values, was essential for identifying unexpected behavior that could lead to system instability.

**Limiting Bias:**

To reduce bias, I rotated between testing modules I hadn’t recently worked on (Since the initial module milestones). For example, after working on the ContactService, I focused on testing TaskService with fresh eyes, allowing for more objectivity. Bias can arise when a developer assumes their own code is correct, which can result in missing critical tests. This is why testing invalid cases like testDeleteNonExistentTask() was so important.

**Commitment to Quality:**

A disciplined approach to testing is critical for avoiding technical debt. Skipping tests can lead to long-term maintenance challenges, as bugs are harder to track down and fix. For example, failing to validate the uniqueness of contact IDs could lead to data corruption. Moving forward, I plan to continue writing comprehensive unit tests and regularly refactor code to avoid complexity. Using automated test suites and adopting continuous integration will help ensure each change is verified, reducing risk and maintaining quality over time.