Student: Dylan Clivio Instructor: Robert Tuft CS 320 6/20/2024

Summary and Reflections Report

Summary:

Unit Testing Approach:

1. Appointment Service:

For the Appointment_Service folder, I've implemented JUnit tests to validate adding, deleting, and retrieving appointments. In doing so I've ensured proper handling of exceptions and boundary conditions. The code also meets requirements like not allowing past dates or duplicate IDs.

2. Contact Service:

For the Contact_Service folder, I've developed tests to validate CRUD operations for contacts. The code also checks for the uniqueness of contact IDs, and other fields like the phone number format and length constraints. Updates to any contact information are properly incorporated into the system,

3. Task Service:

For the Task_Service folder, I created tests to validate adding, updating, and deleting tasks. I have also verified the proper handling of task ID and its constraints. Again, any updates are reflected accurately.

Alignment to Software Requirements:

The unit testing approach was aligned with software requirements as specified in the project documentation. For example, in the Appointment Service, tests were designed to ensure that only future dates are accepted. This aligns with the requirement to prevent scheduling appointments in the past. Each test case was created to validate specific functionalities, ensuring the implemented services met functional and non-functional requirements.

Defend the Overall Quality of the JUnit Tests:

The quality of JUnit tests was robust due to comprehensive coverage across all critical functionalities. Coverage was verified using code coverage tools, ensuring that all code paths were exercised. For instance, in AppointmentServiceTest, the coverage percentage was above 90%, indicating thorough testing of edge cases and normal flows.

Experience Writing JUnit Tests:

Technically Sound:

Writing JUnit tests was a valuable experience in ensuring code correctness and reliability. Tests were structured to be independent, with clear setup and teardown methods to maintain test isolation.

Example code:

```
@Test
public void testAddAppointment() {
    // Test setup
    Appointment appointment = new Appointment("12345", futureDate, "Description");


    // Method under test
    appointmentService.addAppointment(appointment);
    // Assertion
    assertEquals(appointment, appointmentService.getAppointment("12345"));
}
```

This test verifies that adding an appointment with a valid future date works as expected.

Code Efficiency:

Efficiency was maintained by focusing tests on critical functionalities and boundary conditions. Tests were designed to execute quickly and reliably without unnecessary dependencies.

Example of efficient code:

```
@Test
public void testDeleteNonExistentAppointment() {
    assertThrows(IllegalArgumentException.class, () -> {
        appointmentService.deleteAppointment("12345");
    });
}
```

This test efficiently verifies that deleting a non-existent appointment throws the expected exception.

Reflection:

Testing Techniques:

Black Box Testing: Validated functionalities based on external specifications without knowledge of internal implementation.

Boundary Value Analysis: Checked behavior at boundaries, such as minimum and maximum input values, ensuring robust handling.

Not Used Techniques:

White Box Testing: Did not directly examine internal logic or paths within the services.

Mutation Testing: Did not simulate faults to check the effectiveness of tests in detecting changes.

Practical Uses of Used Techniques:

Black Box Testing: Ideal for customer-driven projects where requirements are clear but internal workings are abstracted. This type of testing will not detect performance/efficiency issues.

Boundary Value Analysis: Crucial for ensuring edge cases are handled correctly, preventing errors in critical scenarios. This type of testing is not always easily scalable.

Mindset:

During this project, I practiced caution in testing to uncover potential vulnerabilities and edge cases. It is also important to appreciate the complexity of interconnected systems to anticipate unintended interactions. For example, I ensured contact IDs were unique across the system to prevent data integrity issues.

Limiting Bias in Code Review:

During this project, I was able to avoid bias by focusing on expected behaviors rather than personal assumptions or preferences. I Maintained neutrality by executing tests against documented specifications rather than personal expectations. For example, I tested updates to ensure they correctly reflected changes without assumptions about implementation details.

Importance of Discipline:

Cutting corners in code or testing compromises long-term reliability and maintainability. Commitment to thorough testing reduces technical debt, ensuring future modifications are safe and efficient. For example, I made sure to avoid shortcuts in validation to prevent bugs and ensure the robustness of each service. There are a couple of ways you can avoid technical debt. First, you should plan to regularly refactor and review code to maintain high-quality standards. You can also use automated testing to catch regressions early and prevent the accumulation of defects. For example, I Implemented automated regression tests to validate system integrity after each code change.

Conclusion:

In conclusion, I believe the unit testing approach for the Contact, Task, and Appointment services was methodical and aligned closely with project requirements. The experience gained from writing JUnit tests highlighted the importance of thoroughness and efficiency in ensuring code reliability. Adopting a disciplined mindset in testing and code review proved essential in maintaining high-quality standards and avoiding technical debt. As a software engineer, continuous improvement in testing techniques and a commitment to quality will remain pivotal in delivering robust back-end services for future projects at Grand Strand Systems.