CS 320 Project Two: Summary and Reflections Report

Summary

Unit Testing Approach

In this project, I implemented unit testing for the contact, task, and appointment services of the mobile application using JUnit. Each feature required thorough validation to ensure compliance with software requirements and prevent defects.

* Contact Service: Test cases focused on validating input fields, ensuring that user data such as name, phone number, and contact ID met required constraints. The test cases checked for invalid phone numbers, duplicate contact entries, and empty name fields.
* Task Service: Unit tests were designed to confirm correct task creation, deletion, and updates. The tests also verified that duplicate tasks were not allowed and that deadlines were properly enforced.
* Appointment Service: Tests ensured appointments were scheduled without conflicts, valid time slots were selected, and that appointments could be modified or canceled as expected.

JUnit’s testing framework allowed for automated execution, making it easier to detect regressions and ensure functional correctness across these core application components.

Alignment to Software Requirements

To ensure alignment with software requirements, I referenced the functional specification document while designing test cases. Each test case was mapped to specific user requirements to confirm that expected behavior was met.

For example, one requirement stated that contact names cannot be empty or exceed 50 characters. My test cases included:

1. An attempt to create a contact with an empty name (expecting failure).
2. An attempt to create a contact with a name exceeding 50 characters (expecting failure).
3. A valid contact entry with a properly formatted name (expecting success).

Similar approaches were taken for tasks and appointments, ensuring every functional requirement had corresponding test cases. By structuring tests this way, I could defend the completeness of the unit testing approach and ensure each feature was adequately validated.

Effectiveness of JUnit Tests

The effectiveness of my JUnit tests was measured using test coverage analysis. The key indicators of effectiveness included:

* Code Coverage: Over 85% of the project’s code was covered by unit tests, ensuring that major functionalities were tested.
* Boundary Testing: Edge cases were tested, such as scheduling an appointment at the last available time slot and handling maximum input lengths.
* Exception Handling: The application was tested to ensure it correctly handled invalid inputs, such as incorrect task due dates or overlapping appointments.

The assert statements in JUnit were critical in confirming expected outputs. Using assertions like assertEquals() , assertThrows(), and assertTrue(), I was able to verify correct behavior, making the tests reliable and effective in catching potential bugs.

Technical Soundness and Efficiency

Ensuring that test cases were technically sound was a priority. The following strategies were used to maintain code integrity:

* Mocking dependencies: For database-related operations, mock objects were used to isolate the logic being tested.
* Test reusability: Common setup operations were placed in @BeforeEach methods to reduce redundant code.
* Parameterization: Repeated test cases were optimized using parameterized tests to improve efficiency.

Efficiency was maintained by reducing execution time while keeping test reliability high. By writing efficient tests, the test suite remained maintainable without slowing down development workflows.

Reflection

Testing Techniques Used

The primary software testing techniques implemented in this project included:

1. Unit Testing – Focused on verifying individual components in isolation using JUnit.
2. Boundary Testing – Ensured that input constraints and limits were handled correctly.
3. Exception Handling Tests – Validated system behavior when incorrect data was entered.

These techniques were effective in catching errors early in the development cycle. For instance, boundary tests helped prevent issues where tasks or appointments could be scheduled incorrectly, and exception handling ensured the program did not crash under invalid inputs.

Other Testing Techniques Considered

Although this project primarily involved unit testing, additional techniques could have been beneficial:

* Integration Testing: Would ensure modules interact correctly, such as verifying that contact data flows properly into the appointment service.
* System Testing: Could validate the overall application’s performance under real-world conditions.
* User Acceptance Testing (UAT): Involves testing with actual users to verify usability and business rule adherence.

Had more time been available, implementing integration testing would have been valuable to ensure seamless interactions between different modules.

Uses and Practical Implications

Unit testing is an essential part of software development best practices, especially in production environments where reliability is critical. The practical implications of this project’s testing strategies include:

* Reduced Debugging Time: Catching bugs early means fewer issues in later development stages.
* Improved Software Quality: Tests ensure that updates or new features do not introduce regressions.
* Scalability and Maintainability: A well-tested codebase is easier to modify and expand.

In professional software engineering, automated unit testing supports continuous integration (CI/CD) pipelines, helping to ensure stable and frequent releases.

Caution in Testing

Throughout this project, I adopted a cautious mindset when designing test cases. Software testing is not just about checking expected outputs; it requires anticipating failure points. I carefully considered:

* Edge cases that could break the system.
* Potential user errors and invalid inputs.
* How different components interact under various conditions.

By testing for **unexpected scenarios**, I was able to **improve the system’s robustness** and prevent defects from slipping into production.

Minimizing Bias in Testing:

Since I developed the code, there was a risk of bias in testing because I already knew how the system was “supposed” to behave. To limit bias:

* I wrote test cases that intentionally tried to break the system instead of just confirming expected behavior.
* I reviewed my code from a tester’s perspective, rather than assuming it worked correctly.
* Peer reviews could have further improved objectivity by introducing fresh perspectives.

Emphasis on Discipline:

Maintaining discipline in testing was crucial for ensuring long-term code quality and maintainability. Cutting corners in testing often leads to:

* Increased technical debt
* Higher defect rates in later stages
* Difficult-to-maintain code

By staying disciplined in writing comprehensive tests, I was able to build a strong foundation for the project, ensuring future updates and modifications would not introduce unforeseen issues.

Conclusion:

This project emphasized the importance of unit testing in software development. Through structured JUnit tests, boundary testing, and exception handling, I ensured that the core functionalities of the contact, task, and appointment services met software requirements.

Reflecting on the process, I recognized the importance of caution, bias reduction, and discipline in testing. Expanding testing techniques to include integration and system testing in the future would further improve software reliability. Ultimately, this project reinforced best practices for writing maintainable, efficient, and reliable test cases, which are critical for professional software engineering.