Project 2:

Summary and Reflection Report

By: Austin Bryan

Overview

In this project, I developed and tested several services for a mobile application. Each service had JUnit tests to ensure correctness and that it met the requirements. This report reflects on used techniques, lessons learned, and how the testing process ensured a reliable product.

Testing Approach and Alignment with Requirements

My approach focused on verifying that each class met its specified constraints and handled invalid input properly. The unit tests covered both valid and invalid cases to ensure robustness.

- **Contact Service:** The unit tests (ContactServiceTest.java, Line 34) confirmed that IDs were unique, names were within limits, and phone numbers were exactly 10 digits long. The tests also ensured that contacts could be added, updated, and deleted as expected. The `testAddDuplicateContact()` method (Line 52) ensures that adding a contact with an existing ID throws an exception. According to **Orellana et al. (2017)**, unit testing helps identify defects within a component before integration occurs, making it easier to identify the root cause of an issue when testing.
- **Task Service:** The task service had similar constraints, ensuring valid names and descriptions. The TaskServiceTest.java, Line 45, `testUpdateTaskPartialFields()`, validated that updating a task didn't affect unchanged fields. The `testUpdateNon-ExistentTask()` method (Line 67) verifies that updates on non-existent tasks correctly throws an error. A survey on unit testing practices by **Fucci et al. (2009)** found that well-structured unit tests significantly improve defect detection rates in early stages.
- **Appointment Service:** One of the most important aspects of appointment validation was ensuring that appointments could not be scheduled in the past. The AppointmentService-

Test.java, Line 52, included tests such as `testAppointmentCreationInvalidDate()` to confirm this behavior. Also, `testDeleteNonExistentAppointment()` (Line 73) ensures that trying to delete an appointment that doesn't exist results in an exception.

These unit tests aligned with the software requirements by preventing invalid data from entering the system and ensuring that all operations performed correctly under different conditions.

Test Coverage and Quality

Achieving high test coverage was a priority. The individual application files all had **100% coverage** (when excluding test files), meaning that every line of code was executed during testing. The unit tests covered all CRUD operations (Create, Read, Update Delete), valid and invalid input cases, edge cases, and exception handling for missing or duplicate entries.

According to **Garousi et al. (2017)**, while achieving 100% test coverage is often a goal in safety-critical industries, it is not always a reasonable requirement due to diminishing returns and complexity in larger projects. Instead, focusing on high-priority areas with well-structured tests provides better software reliability.

By achieving complete coverage of application logic, I am confident that the test cases effectively validate the correctness of the implementation.

Reflection on Testing Techniques

Throughout this project, I primarily employed **unit testing**, a method that isolates individual components to validate their correctness. This approach was effective for verifying input validation, exception handling, and CRUD operations. However, there are other testing techniques that were not used:

1. **Integration Testing:** Ensures multiple components work together correctly. Integration tests confirm smooth data retrieval and updates. Research suggests that integration tests take longer to resolve defects because the search space is larger (**Orellana et al., 2017**).
2. **System Testing:** Evaluates the entire application under real-world conditions. Since this project lacked a UI or API interactions, system testing was unnecessary.
3. **Regression Testing:** Ensures new changes do not break existing functionality. Automated regression testing would be useful for projects that undergo frequent updates.

Each of these techniques has its own role in software development, and in a real-world application, combining them would provide a more comprehensive testing approach.

Mindset and Discipline in Testing

1. **Using Caution:** As a software tester, I took a defensive approach by introducing invalid input to ensure proper error handling. For example, `testSettersInvalidInput()` in ContactTest.java (Line 92) tests null and empty values.
2. **Avoiding Bias in Testing:** Since I wrote both the code and the tests, I made sure to take a step back and analyze potential blind spots. I ensured the tests included scenarios that I might not anticipate during development, such as attempting to delete an already removed contact (ContactServiceTest.java, Line 78).
3. **Commitment to Quality:** Imperfect testing leads to technical debt (TD), making future maintenance costly. To avoid this I used reusable validation methods like `validateString()` (Appointment.java, Line 40) that ensure consistent enforcement of

constraints. According to **Krishna (2013)**, neglecting TD means increased complexity and maintenance costs, reinforcing the importance of following best practices.

Conclusion

This project emphasized the value of strong unit tests for software reliability. By adhering to requirements and implementing thorough validation, I built a robust system that prevents invalid operations. The experience will aid future projects, enhancing maintainability and accuracy through test-driven development (TDD).

Works Cited

1. Crispin, L., & Gregory, J. (2009). *Agile testing: A practical guide for testers and agile teams.* Addison-Wesley.
2. Fucci, D., Turhan, B., & Oivo, M. (2009). *A survey of unit testing practices.* Retrieved from ResearchGate
3. Garousi, V., Felderer, M., & Mäntylä, M. V. (2017). *Is 100% test coverage a reasonable requirement? Lessons learned from a space software project.* Retrieved from ResearchGate
4. Krishna, V. (2013). *Software engineering practices for minimizing technical debt.* Retrieved from ResearchGate
5. Orellana, G., Laghari, G., Murgia, A., & Demeyer, S. (2017). *On the differences between unit and integration testing in the TravisTorrent dataset.* Retrieved from ResearchGate