Summary and Reflections Report

Yakuba Conde

Southern New Hampshire University

CS-320-18647-M01 Software Test, Automation QA 2024 C-6 (Oct - Dec)

Professor Jacks

December 15, 2024

# Summary

# Unit Testing Approach for Each Feature

# For the Contact, Task, and Appointment services, I developed unit tests that aligned with the requirements by validating constraints and core functionalities for each feature:

# 1. Contact Service:

# - Verified that `contactId` is unique, non-null, and no longer than 10 characters.

# - Tested the `phone` field to ensure it is exactly 10 digits and rejects invalid input.

# - Confirmed adding, deleting, and updating contacts by `contactId` works as expected.

# 2. Task Service:

# - Ensured `taskId` is unique and immutable.

# - Validated `name` and `description` fields for null values and length constraints.

# - Checked the add, delete, and update functionality for tasks.

# 3. Appointment Service:

# - Confirmed `appointmentId` is unique and immutable.

# - Validated `appointmentDate` to ensure it is not null and not in the past.

# - Tested addition and deletion of appointments for correct handling of invalid operations.

# Alignment to Software Requirements

# The tests were directly aligned with requirements. For example, the `phone` field in the `Contact` class was tested using both valid and invalid inputs, ensuring only valid 10-digit phone numbers were accepted. Each requirement had corresponding tests to cover edge cases.

# Defending the Quality of JUnit Tests

# The JUnit tests achieved over 80% coverage, validating their thoroughness. Each test addressed both positive and negative scenarios, ensuring the program handled expected and unexpected inputs effectively. Regular test execution confirmed functionality and caught regressions.

# Experience Writing JUnit Tests

# Writing tests involved careful analysis of requirements and edge cases. Examples include:

# -Technical Soundness: In TaskServiceTest.java, invalid inputs for the `description` field were explicitly tested:

# @Test

# void testUpdateDescriptionWithInvalidInput() {

# Exception exception = assertThrows(IllegalArgumentException.class, () -> {

# taskService.updateTaskDescription(taskId, null);

# });

# assertEquals("Description cannot be null.", exception.getMessage());

# }

# - Efficiency: Repeated tests were streamlined using parameterized methods in ContactServiceTest.java:

# @ParameterizedTest

# @ValueSource(strings = {"", "ThisNameIsWayTooLong"})

# void testInvalidFirstName(String invalidName) {

# Exception exception = assertThrows(IllegalArgumentException.class, () -> {

# new Contact("12345", invalidName, "LastName", "1234567890", "Address");

# });

# assertEquals("First name must not be null and must be 10 characters or fewer.", exception.getMessage());

# }

# 

# Reflection

# Testing Techniques

# Techniques Used:

# - Boundary Testing: Validated fields at and beyond their constraints.

# - Negative Testing: Ensured invalid inputs, such as null fields or past dates, were handled gracefully.

# - Unit Testing with Mock Data: Used in-memory data structures for test cases.

# Techniques Not Used:

# Integration Testing: Not applicable due to no external systems being involved.

# Performance Testing: Not relevant for these lightweight services.

# Practical Uses of Techniques:

# - Boundary Testing ensures reliable input validation.

# - Integration Testing is ideal for systems interacting with external APIs or databases.

# - Performance Testing is critical for systems with heavy traffic or resource-intensive operations.

# Mindset

# Caution and Complexity Appreciation

# I approached testing with attention to detail, recognizing code dependencies. For example, the immutability of appointmentId maintained data integrity across operations.

# Limiting Bias

# I reviewed the code as a tester, focusing on potential weaknesses. Writing tests for null inputs and invalid lengths ensured robustness.

# For example:

# if (description == null || description.length() > 50) {

# throw new IllegalArgumentException("Description must not be null and must be 50 characters or fewer.");

# }

# Commitment to Quality

# I prioritized completeness over shortcuts, addressing all edge cases to prevent technical debt. Clear, maintainable tests support long-term quality.

# Avoiding Technical Debt

# By sticking to requirements and ensuring comprehensive test coverage, I minimized potential defects. This approach enables future developers to confidently extend and maintain the application.

# 

# References

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley.

Beck, K. (2003). Test-driven development: By example. Addison-Wesley.

Freeman, S., & Pryce, N. (2009). Growing object-oriented software, guided by tests. Addison-Wesley.