Mitchell Fontaine

CS-320

10/18/2024

7-2 Project Two: Summary and Reflections Report

I have been tasked with providing a follow-up summary and reflection report explaining how I analyzed various approaches to software testing based on requirements and applied appropriate testing strategies to meet requirements while developing the customer's mobile application. Three features were required in this project and the first I will explain is the Appointment feature. For the Appointment and AppointmentService classes, a series of unit tests were crafted to ensure both the creation of objects and the validity of data inputs. The focus was verifying that appointments could be created successfully with valid inputs and that any invalid data, such as incorrect dates or IDs, would be rejected. The AppointmentService tests covered CRUD operations extensively, ensuring that appointments could be added, updated, retrieved, and deleted without errors. A special emphasis was placed on edge cases. For instance, tests were written to verify how the system handles appointments with invalid or too-long IDs, invalid dates (such as past dates or incorrectly formatted dates), and descriptions that might exceed length constraints. The service’s methods for adding and deleting appointments were scrutinized to ensure they functioned correctly, even when faced with non-standard inputs or operational edge cases.

The Contact and ContactService classes were similarly tested to ensure robust functionality. The unit tests were designed to confirm that the system correctly handles the creation of Contact objects, checks the validity of inputs (such as phone numbers and email addresses), and appropriately manages CRUD operations. A critical aspect of these tests was the verification of unique identifier generation. For each new contact added, the system had to guarantee that no duplicate IDs could be generated, ensuring data consistency and uniqueness. The tests also explored scenarios where invalid data was introduced, ensuring that the system rejects invalid phone numbers, names, or email addresses gracefully. Edge case handling was a key component of these tests, ensuring that even in the face of highly unlikely or extreme inputs, the system remains reliable and predictable.

For the Task and TaskService classes, the focus of the unit tests was on the correct creation of tasks, updates to tasks, and input validation. Unit tests examined whether tasks could be added and updated correctly, ensuring that tasks with invalid names, descriptions, or deadlines were properly rejected. The service-level tests evaluated the system’s ability to handle typical CRUD operations, such as adding, deleting, and updating tasks. Special attention was paid to error conditions, such as attempts to update or delete non-existent tasks. The system’s error-handling capabilities were tested through the use of exception-based unit tests, ensuring that meaningful errors were thrown in invalid scenarios. Overall, the tests aimed to guarantee that the task management features functioned as expected and that all operations on tasks, including updates and deletions, adhered to the required business logic and input constraints.

The unit testing approach aligned closely with the project's software requirements, ensuring that each feature was tested against the specified functionality. Every feature underwent rigorous input validation testing, where tests ensured that invalid data was appropriately rejected. AppointmentTest.testInvalidAppointmentId\_TooLong() tested for ID length verification, ContactServiceTest.testAddDuplicateContactID() ensured unique contact IDs, and TaskServiceTest.testUpdateNonExistentTask() validated input constraints for tasks. CRUD operations were thoroughly tested for all three features. The tests verified that the system’s core functionality behaved as expected when adding, updating, retrieving, and deleting data. The system’s ability to handle errors was a critical focus. Tests were implemented to verify that exceptions were thrown when invalid operations were attempted. The generation and validation of unique identifiers were tested in the ContactService, ensuring that no two contacts could share the same ID.

The JUnit tests demonstrated a high level of quality due to several factors. The tests covered various scenarios, from regular user actions to edge cases. This thorough coverage gave confidence in the system’s reliability and covered 87.4% of all code. Each test focused on a specific aspect of functionality, which ensured that any failures could be traced to a particular feature or input. The use of JUnit’s @BeforeEach annotation helped maintain test isolation by setting up a clean test environment before each test was run. Test method names were clear and descriptive, making it easy to understand what functionality each test was verifying.

From a technical standpoint, the unit tests were sound and well-structured. Assertions such as assertEquals, assertTrue, and assertThrows were used to verify expected outcomes. For example: assertThrows(IllegalArgumentException.class, () -> {

*new* Appointment(null, futureDate, "Valid description");

   });

was used to verify exception handling. By verifying that the appropriate exceptions were thrown for invalid operations, the tests ensured that the system handled errors correctly. The @BeforeEach annotation was used effectively to reset the test environment between test cases like this: @BeforeEach

   void setUp() {

       contactService = *new* ContactService();

   } This ensured that the results of one test did not affect another, contributing to the reliability and reusability of the tests.

The efficiency of the tests was evident in a few ways. Each test was designed to create only the objects necessary for the specific test case. This approach ensured that the tests remained fast and focused. The use of @BeforeEach allowed common setup code to be reused across multiple test cases, reducing code duplication and improving the overall maintainability of the test suite. Assertions were made directly on the results of method calls, ensuring that the system’s outputs were verified without unnecessary complexity.

Several key testing techniques were employed throughout the unit testing process. Unit testing was used to ensure each feature was tested in isolation, ensuring that individual components worked correctly. The tests frequently checked edge cases, such as maximum and minimum input values, to ensure that the system handled boundaries effectively. Values from both valid and invalid ranges were tested, ensuring that the system responded correctly across a wide variety of inputs. Potential error conditions were anticipated, and tests were created to verify that the system handled these errors appropriately (GeeksforGeeks, 2024c).

While the focus was on unit testing, several other testing techniques could have been utilized if the project called for them. Integration testing would be important for verifying that different components worked together seamlessly. System testing would ensure that the entire application, once integrated, met its functional and non-functional requirements. Performance testing would evaluate how the system performs under different loads and conditions would be important for large-scale or performance-critical applications. Security testing would verify the system’s defenses against common vulnerabilities, such as injection attacks or data breaches, which would be critical for applications handling sensitive information (Bose, 2023).

In acting as a software tester, I adopted a cautious and detail-oriented mindset, approaching each feature with skepticism and a focus on potential failure points. This caution was essential for identifying edge cases and ensuring functionality, as seen in testing appointment ID validation and task updates. Understanding the complexity and relationships of the code was crucial, as changes in one feature could impact others (Verbat, 2024). For example, consistent input validation across AppointmentService and ContactService was necessary to maintain system integrity.

To limit bias in reviewing my code, I consciously approached tests from a fresh perspective, imagining how users might interact with the system in unexpected ways. By writing tests designed to break the code, I was able to uncover issues that may have been missed. Bias can be a significant concern for developers testing their code, as familiarity often leads to overlooking potential flaws. Committing to quality and avoiding technical debt is essential for long-term success. Practices like test-driven development (TDD), regular refactoring, and comprehensive testing ensure that code remains reliable, maintainable, and scalable over time (Geethanjali, 2018).

References

Bose, S. (2023, May 25). *Software Testing Techniques: Explained with Examples | BrowserStack*. BrowserStack. <https://www.browserstack.com/guide/software-testing-techniques#:~:text=What%20are%20Software%20Testing%20Techniques?%20As%20the%20name%20suggests,%20testing>

Geethanjali S. (2018, October 25). *Cognitive Bias in Software Testing: Why Do Testers Miss Bugs?* Software Testing Help. <https://www.softwaretestinghelp.com/cognitive-bias-in-software-testing/>

GeeksforGeeks. (2024c, September 26). *Software testing techniques*. GeeksforGeeks. <https://www.geeksforgeeks.org/software-testing-techniques/>

Verbat. (2024, April 25). *The Importance of Software Testing - Verbat*. Software Development Company Dubai UAE - Verbat Technologies. <https://www.verbat.com/blog/the-importance-of-software-testing-all-you-need-to-know/>