Project 2: Summary and Reflections Report

Bradley Wells

Southern New Hampshire University

CS-320

Professor Wilson

June 17, 2024

For this mobile application, I wrote the appointment, contact, and task services as well as test cases for those classes. The approach I took in writing those classes aligns with the specific requirements laid out in the assignment. The test cases also align with these requirements as I learn to become better with writing tests and increase the overall coverage percentage. Each test case was designed to validate the functionalities specified by the requirements. Evidence of this can be seen in the ContactServiceTest where there is a test for adding contacts with unique IDs (line 19), updating contact fields (line 78), and deleting contacts (line 59). Similarly, in the TaskServiceTest, there are tests to ensure tasks can be added (line 20), task fields can be updated (line 52), and tasks can be deleted (line 34).

In testing, I can stand behind the JUnit tests for all the services based on the coverage percentage. Testing is running with an overall coverage percentage of 85.6% as can be seen below:

A screenshot of a graph

Description automatically generated

The coverage percentage over 80% was achieved by ensuring there was adequate coverage of major functionalities like adding, updating, and deleting tasks and contacts. Additionally, cases were adding tasks with duplicate ID’s or updating non-existent tasks were covered. Through testing I can ensure that IDs are unique and validated as well as proper exception handling for invalid operations.

Code can be ensured to be technically sound by validating outcomes that are expected as well as checking exceptions. In the TaskServiceTest, exception handling is evident in the test at line 26. In this test, the first line adds a task with a unique ID ("123"), which should succeed. Next, the test attempts to add another task with the same ID ("123"). According to the requirements, this should throw an IllegalArgumentException because task ID’s must be unique. The assertThrows method is then used to verify that the expected exception (IllegalArgumentException) is thrown when trying to add a duplicate task ID. This ensures that the exception handling mechanism in the addTask method is correctly implemented. Data is validated in the same test case at line 59. A task is added with a specific ID, name, and description. This sets up the initial state needed for the update operation. Then the test updates the description of the task with ID "123". The new description is "Updated description". The assertEquals method is used to validate that the description of the task with ID "123" has been correctly updated. The stream API is utilized to find the task by its ID and retrieve its description. The test ensures that the task description is properly updated.

Efficiency in code is important so that it is easy to follow, update, and understand by all who will be utilizing it. Efficiency in code was achieved here by optimizing the test cases to avoid redundant operations and ensuring clear and concise validation checks. In the ContactServiceTest, the Stream API is used for more efficient searching (line 38). By providing a high-level abstraction for processing sequences of elements. This allows for cleaner, more concise code. Bulk operations are performed and evident in the ContactServiceTest (line 19). In this test, bulk operations are performed using the Stream API to filter and match a contact within the contactList. The anyMatch method checks if any contact in the list matches all the provided criteria (ID, first name, last name, phone number, and address), ensuring the added contact's correctness. These tests demonstrate efficiency in coding in this application in progress.

The primary testing techniques used for each of the services were unit testing, boundary testing, exception testing, and the use of the Stream API for bulk operations. Unit testing focused on verifying individual methods within each class, ensuring that each unit of code performed as expected. Boundary testing was applied to validate the maximum length constraints for IDs, names, and descriptions, ensuring that the application correctly handled edge cases. Exception testing was used to confirm that the application threw appropriate exceptions for erroneous conditions, such as adding duplicate IDs. The Stream API was utilized for efficient and readable bulk operations, such as checking if a contact or task existed in the list.

For these services there were several software testing techniques that were not employed, including integration testing, system testing, regression testing, and user acceptance testing. Integration testing focuses on verifying interactions between integrated units or modules, ensuring they work together as expected. System testing evaluates the entire integrated system against specified requirements, providing an end-to-end assessment of the application. Regression testing ensures that recent code changes do not negatively impact existing functionality, maintaining stability and functionality after updates are performed. User acceptance testing is conducted to confirm that the system meets business requirements and is ready for deployment, often involving real-world scenarios and user interactions although not practically applicable in these milestones.

Each of the testing techniques discussed have their own practical uses and implications based on the software development project. Unit testing is crucial for early bug detection and simplifying debugging, ideal for projects with many small, independent modules. Boundary testing enhances code robustness by handling edge cases, preventing unexpected behaviors. Exception testing improves durability and user-friendliness by ensuring proper error handling. Integration testing is vital for applications with multiple interdependent components, detecting issues in module interactions early. System testing provides a more comprehensive assessment of the application's readiness for deployment. Regression testing maintains long-term code quality and functionality, essential for projects with frequent updates. User acceptance testing ensures the application meets user needs and business requirements, reducing post-deployment issues and increasing end-user satisfaction. The used testing techniques in this application will ensure validation of individual components but incorporating the unused techniques offers a more complete testing environment that could enhance overall software quality and reliability.

Throughout this project, I adopted a mindset that was both cautious and thorough in software testing. As a software tester, it is crucial to approach each test case with a critical eye, considering various scenarios and potential edge cases. For example, when testing the ContactService class, I ensured that the methods for setting contact information validated input data. This caution extended to handling exceptions gracefully, anticipating potential errors, and validating outputs against expected results. Appreciating the complexity and interrelationships of the code being tested was of course, important. For example, in the TaskService class, I considered how updates to task fields could impact other parts of the application. By understanding the code's dependencies and interactions, I crafted test cases that covered not only individual method behavior but also how changes in one part of the codebase might affect other components.

In reviewing the code, I made conscious efforts to limit bias and approach the review objectively. One strategy was to focus on the functionality and adherence to requirements rather than personal preferences or assumptions. In the AppointmentService class, I reviewed the validation logic for appointment dates objectively, ensuring that dates in the past were rejected as per specifications, without bias towards specific date formats or conventions. As a software developer responsible for testing my own code, bias could indeed be a concern. To mitigate this, it is essential to have a mindset of self-critique and to challenge my own assumptions. For instance, in the Task class, I would not assume that a particular algorithm implementation is flawless just because I wrote it. Instead, I would subject it to testing, considering various inputs and edge cases to uncover potential flaws or biases in my approach.

Being disciplined in the commitment to quality is fundamental in software engineering. Cutting corners in writing or testing code can lead to technical debt, where unresolved issues accumulate, hinder future development, and potentially become unexpected costs. In the ContactTest class, I ensured that all test cases covered both positive and negative scenarios, leaving no room for shortcuts or incomplete testing. I thoroughly tested boundary conditions for input fields, preventing potential bugs from slipping through. Avoiding technical debt requires being proactive and thorough with testing. As a practitioner, I plan to regularly refactor and review code, prioritize automated testing to catch regressions early, and advocate for thorough testing practices within the development team. By investing time and effort upfront in writing code and comprehensive tests, we can avoid costly rework and maintenance challenges down the line, ensuring long-term quality and sustainability in software projects.

References

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing: An ISTQB-BCS certified tester foundation guide* - *4th edition*. BCS Learning & Development Limited.

*JUnit - Test Framework.* Tutorials Point. <https://www.tutorialspoint.com/junit/junit_test_framework.html>

SNHU Media. (2020, Sep 11). *JUnit Tutorial*. YouTube. <https://www.youtube.com/watch?v=KbXhK9HUng4>

*Tutorial Using JUnit.* Southern New Hampshire University. <https://learn.snhu.edu/content/enforced/1615486-CS-320-11262.202451-1/course_documents/CS%20320%20Module%20Four%20Tutorial%20Using%20JUnit.pdf?ou=1615486>