Ty Wheeler

CS 320

10/16/24

Project Two

*Summary*

My unit testing approach focused on creating detailed JUnit tests that covered all key operations which were adding, updating, and deleting objects. Each test verified that input validation rules were applied correctly, such as checking for the maximum string length and ensuring that date and phone number formats adhered to project requirements. This approach ensured that both positive and negative cases were covered and validated not only expected functionality but also error handling.

My approach aligned closely with the software requirements, which called for the validation of input data and specific constraints for each field. For example, in the Contact service, first and last names had a maximum length of 10 characters, and phone numbers had to be exactly 10 digits long. These constraints were enforced in my unit tests to ensure that the code complied with the project’s specifications. I also ensured the uniqueness of IDs and proper error handling for duplicate entries.

The overall quality of my JUnit tests was demonstrated by achieving over 80% test coverage as required in the guidelines. This high coverage indicated that the tests adequately exercised the code’s logic and identified edge cases. The effectiveness of these tests was supported by their ability to detect issues early in development which reduces the likelihood of defects. Specific cases such as adding a contact with an invalid ID or trying to update a task with invalid data, were successfully caught by the tests.

Writing the JUnit tests was challenging but rewarding once done correctly. To ensure the code was technically sound I had implemented thorough assertions in each test such as assertEquals to validate expected outcomes. For example, in the testUpdatePhoneNumber method, I used assertions like assertEquals("0987654321", contactService.getContact("12345").getPhone()); to ensure that the phone number was updated correctly. Efficiency in the code was achieved by avoiding redundant operations. For example I reused instances of objects in multiple tests rather than creating new instances. In the testAddContact method I ensured the test followed best practices by initializing the Contact object once and reusing it across tests.

*Reflection*

The primary software testing technique I employed was unit testing using JUnit, “JUnit is the most important testing framework for the JVM and one of the most influential

in software engineering in general” (Boni Garcia). Unit testing focuses on testing individual components of code in isolation to ensure that each function or method works correctly. By breaking the application down into manageable pieces I was able to thoroughly test the functionalities of the Contact, Task, and Appointment services. Each test case targeted specific input validations such as ensuring string lengths, format validation, and uniqueness of IDs. For example in the testAddContactWithDuplicateId I simulated adding a contact with a non-unique ID to confirm that the program prevented this action in order to align with the project’s requirements. Another testing technique that could have been employed but was not used in this project is integration testing. While unit testing focuses on individual components, integration testing examines how different parts of the system work together. Its purpose is to detect issues that arise when components are combined like how the Contact service interacts with the database or other services. System testing could have also been used to test the system as a whole, ensuring that the entire application met the customer's requirements and performed as expected in real-world scenarios. This level of testing could have been beneficial for identifying issues that arise from interactions across the entire system. The practical use of unit testing is essential in catching issues early in development by isolating bugs in specific parts of the code. For smaller or individual features, such as the ones in this project, unit testing is highly effective. However, for larger, more complex projects, integration testing or system testing would be more beneficial to ensure that the interactions between components don’t introduce new bugs.

Throughout the project, I had a detail oriented mindset as I had to recognize the complexity of the code in order to best avoid errors. It was essential to employ caution when testing each feature, particularly when dealing with input validation. A failure to thoroughly test all inputs could have resulted in errors that were difficult to trace later on. For instance, when testing the update functionality in the ContactService, I ensured that multiple versions of invalid data were tested recognizing that different inputs might expose different issues in the code. My mindset also included anticipating potential edge cases such as what might happen if a contact were added with a null ID.

Limiting bias in reviewing the code was another priority. When acting as both the developer and the tester it can be easy to overlook issues in the code you’ve written. To minimize this bias I tried to follow a test driven development mindset treating the tests as independent checks on the system. I tried to design my JUnit tests to be impartial by ensuring they were written to fail in cases of bad input not simply pass with expected input. This helped avoid the natural inclination to assume that my code works just because it passes one test case. Bias can be a concern when a developer tests their own code. Maintaining quality and consistency in my code was significant as cutting corners could lead to major issues down the line and technical debt. To avoid technical debt I ensured that all tests were comprehensive and ran correctly even if it meant rewriting parts of the code or rethinking my approach. For example after detecting an issue in the testAddContactWithDuplicateId() method, I revisited how unique IDs were being generated and enforced this validation throughout the system’s code. I ensured that the project met the requirements and maintained code quality.

References

Boni Garcia. (2017). Mastering Software Testing with JUnit 5 : A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language. Packt Publishing.