Eric L. Foster.

CS-320

Summary and Reflections Report.

Summary

It was realized rather quickly that the development of the contact, task, and appointment back end services would require a comprehensive unit testing approach that would ensure the stability, functionality, and maintainability of these services. The unit testing approach was rooted in the principles of isolation, precision, and thorough validation (CodeFresh, 2024). For each service, a focus was placed on creating test cases that would validate individual methods, enforce constraints on object properties, and ensure proper handling of exceptions.

For the Contact Service, the testing approach focused on verifying CRUD (Create, Read, Update, Delete) operations (Escott, 2020). The addition, deletion, and updating of contacts was tested to ensure that they met system constraints, and to verify the immutability of unique identifiers. Each contact field, including the first name, last name, phone number, and address, was tested for both valid and invalid inputs. For instance, the test named testInvalidContactId() worked to ensure that contact IDs exceeding ten characters or null values raised an IllegalArgumentException. The same methodology was then applied to the other fields, where tests like testInvalidPhone() worked to validate that phone numbers had to be exactly ten numeric characters.

The Task Service required unit tests that supported similar CRUD functionality with an added emphasis on constraint validation (MozDevNet, 2024) for the task name and description. Test cases were created for task creation, updates to the task name and description, and task deletions. To ensure system integrity, strict enforcement of the constraints was placed on the name and description, ensuring the name was within the character limit and that descriptions did not exceed fifty characters. Tests like testTaskCreationInvalidDescription() demonstrated this approach, raising exceptions when descriptions surpassed the allowed limit. Furthermore, tests like testAddTaskDuplicateId() guaranteed that no two tasks could have the same task ID, preventing duplication errors.

For the Appointment Service, the testing focused on ensuring accurate appointment scheduling, date validation, and proper exception handling for past or null dates. It was then verified that appointments could only be created for future dates, as evidenced in TestInvalidAppointmentDate(), which raises an IllegalArgumentException for past dates or null inputs. Additionally, the immutability of the appointment ID was tested using testImmutableContactId(), ensuring that once an ID was assigned, it could not be altered.

The unit testing approach adhered strictly to the requirements. Compliance was ensured by creating dedicated tests for each requirement outlined in the specifications. For instance, if the requirement stated that contact IDs should not exceed ten characters, this constraint was directly enforced using the testInvalidContactId() method. Additionally, boundary testing, such as using edge-case data like "1234567890" (valid) and "12345678901" (invalid), ensured compliance with length constraints.

The overall quality of the JUnit tests was demonstrated by the high coverage percentage achieved for each service. The coverage was maximized by testing not only positive paths but also negative scenarios, such as null inputs, out-of-bound characters, and deletion of non-existent records. For example, in the Task Service, testDeleteTaskNotFound() confirmed that the system properly raised exceptions when attempting to delete a task that did not exist. These tests ensure that the system will not silently fail, making the code more robust and maintainable. By validating both the "happy path" and the "error path," comprehensive code coverage was achieved (Pitt, 2023).

The experience of writing JUnit tests ( Kanai, 2022) was insightful and rewarding. To ensure technical soundness, A focus was placed on asserting specific expectations for every method call.

For example, in testDeleteContact(), the test asserts that service.getContact("12345") returns null, validating that the contact was successfully deleted. To ensure the code was efficient, assertions were kept to a bare minimum while maintaining comprehensive coverage. For instance, instead of writing multiple assertions for each field individually, all expected field values in a single method call would be asserted, as evidemced in testTaskCreationSuccess(), which checks that the task ID, name, and description are all correct in a single test. This approach seemed more efficient at reducing redundancy while maintaining clarity.

Reflection

The key testing techniques employed in this project were unit testing, negative testing, boundary testing, exception testing, and immutability testing. Each of these techniques served a specific purpose.

Unit testing was central to the approach (CodeFresh, 2024), ensuring that each class and method was tested in isolation. By using unit testing, issues could be pinpointed at the component level rather than debugging the entire system. Negative testing was applied to validate how the system would respond to invalid data, such as null inputs, excessively long IDs, and non-numeric phone numbers. Boundary testing verified the limits of constraints, ensuring fields like names and descriptions adhered to size requirements(GeeksforGeeks, 2024).

This technique is evident in testInvalidContactId(), where the system rejects IDs with eleven characters. Exception testing (Gupta, 2024) confirmed that errors were raised correctly for invalid operations, such as deleting non-existent tasks or scheduling appointments with past dates. Lastly, immutability testing was applied to ensure that unique identifiers, like contact IDs, task IDs, and appointment IDs, could not be altered after object creation, as seen in testImmutableContactId().

Testing techniques not explicitly used in this project include integration testing, system testing, performance testing, and user acceptance testing (UAT). While unit testing focused on individual methods and classes, integration testing would have tested how these services interacted with each other or external systems. System testing would validate the entire application from an end-user perspective, ensuring a complete end-to-end workflow. Performance testing would evaluate the speed and responsiveness of services, especially useful in large-scale production environments. Lastly, user acceptance testing (UAT) would allow end users to interact with the system to confirm it meets their expectations.

The practical uses of these techniques are significant. For example, unit testing is ideal for internal code validation, while system testing ensures the product works as expected in a production environment. UAT is crucial in user-facing applications, as it identifies usability issues before launch. Performance testing is critical for high-traffic applications where speed is essential. Each technique has its unique role, and it is critical to understand when to employ each to be a successful software developer.

The mindset that was adopted during this project was one of caution and precision. Acting as both developer and tester there was an appreciation placed the complexity of code interactions and potential errors were anticipated before they could occur. For example, it was assumed that users would enter invalid phone numbers, like "1234" or "123abc7890," and created tests like testInvalidPhone() to handle these inputs. This proactive mindset prevented issues from escaping into production.

To limit bias, the code was approached from the perspective of a third-party tester. Rather than assuming the code was correct, ways were actively sought to "break" it. A question then arose, "What would a careless user do?" and designed tests accordingly. For example, edge cases were tested like zero-length strings and null values for names, which might be overlooked during development. By anticipating user error, it was ensured that the tests were thorough and unbiased.

The discipline of maintaining high-quality code was essential. Cutting corners during testing introduces technical debt, which can be costly to resolve later. By being disciplined, the trap of assuming "this function is simple, it doesn't need a test" was avoided. Instead, all paths were tested, even for "simple" functions like deleteContact(). This approach reduced potential rework and allowed for faster development down the line. For instance, care was taken to ensure that every exception thrown had a descriptive error message and tested for the presence of those messages.

To avoid technical debt in future projects, there will be adherence to a few principles. First, always striving for test-driven development (TDD), where tests are written before the actual implementation. Second, an enforcement of the "leave it better than you found it” rule (Curtis, 2020), refactoring code and tests when opportunities arise. Lastly, a prioritization will be placed automation and CI/CD + CM pipelines (Interrupted, 2023), ensuring that tests are run automatically before any code is merged into production. This will catch issues early, prevent regressions, and reduce long-term maintenance costs.

The development and testing of the contact, task, and appointment services required a disciplined, comprehensive testing strategy. By focusing on unit testing, boundary testing, negative testing, and exception testing, it was ensured the system was robust and maintainable. The experience has reinforced the importance of caution, attention to detail, and objectivity in software testing. As a software engineer, there exists a commitment to uphold these principles in every future project to ensure quality, reduce technical debt, and deliver software that meets user needs.

References.

CodeFresh. (2024, August 8). *Unit testing: Principles, benefits & 6 quick best practices*. Codefresh. <https://codefresh.io/learn/unit-testing/>

Curtis, S. (2020, June 26). *The boy scout rule in coding*. Medium. <https://medium.com/@stevenpcurtis/the-boy-scout-rule-in-coding-a0b4d8130798>

Escott, E. (2020, October 8). *How do you test your crud operations?*. Codebots. <https://codebots.com/crud/how-to-test-CRUD>

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

Gupta, V. (2024, June 27). *A complete guide to exception tests in testng [with examples]*. LambdaTest. <https://www.lambdatest.com/blog/exception-tests-in-testng/>

Interrupted, D. (2023, January 26). *Why your team needs to add CM to its CI/CD pipeline*. HackerNoon. <https://hackernoon.com/why-your-team-needs-to-add-cm-to-its-cicd-pipeline>

Kanai, S. (2022, May 25). *Junit testing framework - A comprehensive guide: Headspin*. HeadSpin. <https://www.headspin.io/blog/junit-a-complete-guide>

MozDevNet. (2024). *Constraint validation - HTML: Hypertext markup language: MDN*. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/HTML/Constraint_validation>

Pitt, M. (2023, April 10). *What is the “Happy path”?* Zeplin Gazette. <https://blog.zeplin.io/what-is-the-happy-path/>