When working on the development of my models & the service classes, I started confirming my coverage using Junit testing. The mindset behind this is to focus on reliability, coverage, & robustness of the code. Trying to place myself in the mind of a software programmer by focusing on implementation of specific code to accurately perform the task in mind. The side of the brain was used to scheme ways to defeat the code via manipulation of flaws & general oversight in the programmer’s work. This dual approach allowed me to thoroughly evaluate the code’s resilience and adherence to requirements.

In order to test the boundaries of the specifications, I used unit tests designed to push the code to its limit. The Task class specification needed to display the fullName & not be null & not exceed 20 characters. I did this using 4 branches,

* *if (uniqueId == null || uniqueId.length() > 10)*
* *if (name == null || name.length() > 20)*
* *if (description == null || description.length() > 50)*

These tests ensure that the validation logic is working. I ran an assertion to verify that exceptions were thrown when invalid inputs were provided. used assertions to verify that exceptions were thrown when invalid inputs were provided, such as an empty string or an excessively long name. Two examples of these tests included checking for an empty name and a name that far exceeded the character limit. By doing so, I confirmed that the code handled edge cases appropriately and adhered to the specified constraints. This did not always run as thought of due to errors in my code.

A screen shot of a graph

AI-generated content may be incorrect.

Though I attempted to have comprehensive testing, I was unable to achieve 100% in any of the classes. My java classes seemed to be good, but small errors in the testing kept tripping me up. Some constructors required three or more parameters that I deemed unnecessary and additional testing. Instead, I focused on achieving 100% coverage for the getters and setters, testing only one valid constructor and one invalid constructor to ensure basic functionality. This approach balanced thoroughness with practicality, allowing me to validate the core functionality with mixed results in my work.

I made efforts to ensure the software was sound. My ContactService class had a method to delete a contact by ID. I was able to test this by adding three objects to the collection, deleting one, & confirming it was deleted. I made a mistake & ended up storing a list of ContactServices instead of contacts. Which was technically working, just not to the expectation. Which could be checked by verifying it was no longer present after deletion. Though I received some misleading results, with a lot of errors coming from null. For instance, checking only the collection size might allow the test to pass even if the wrong item was deleted. Conversely, searching for the deleted item alone might not detect if all items were accidentally removed. To address this, I implemented both checks in my tests, ensuring that the collection size was correct and that the deleted item was truly absent.

Alongside deletion testing, I verified that objects were being instantiated only when the request was made. Confirming that the ContactService collection was initially empty before adding any contacts, checking that failed updates don’t alter the original data, verifying if an update to a task description failed due to the uniqueID, the original remains unchanged. These steps keep the consistency of data validation & function as a reliable update mechanism.

Throughout the modules, I used a combination of unit testing & static testing. Reviewing the code against the software & specs to find bugs was crucial once my JUnit tests failed. While I coded, I aligned logic with the specs but during the testing, I challenge my implementation to uncover the errors, like using wrong attributes or misaligned naming nomenclature. I had multiple instances where exceptions weren’t thrown when my code was in a po0sition to do so. In one case, a constructor had failed, but the exception wasn’t what I expected, highlighting the need to verify exception types to avoid false positives.

While the service classes underwent some system testing, they lacked integration testing. The specifications outlined a model class and a service class that interacted with it. In theory, the model class could be tested through the service class, treating this interaction as a system. Integration testing, which would involve testing the entire application by bringing all systems together, was not performed. Such testing would validate that the TaskService, AppointmentService, and other classes worked seamlessly together. Additionally, I did not implement automated testing; all JUnit tests were run manually. In a larger application, automated tests could be integrated into a continuous integration pipeline, running after each build event to ensure ongoing reliability.

Security testing was also absent from my project. Although it doesn’t involve databases or user input, making it less vulnerable to threats like SQL injection or buffer overflows, security scanning is crucial in the more complex systems. Libraries & components should be regularly scanned for vulnerabilities to protect against emerging threats.

Unit testing was invaluable in identifying small issues before they can evolve into larger problems. By hardening the code against often overlooked logic branches, such as handling the null values, missed specifications, & enforcement of maximum string lengths. As the codebase grows & begins connecting more components, system integration testing grows in importance & value.

*@Test*

*public void testTaskDescriptionConstraints() {*

*// Test null description*

*assertThrows(IllegalArgumentException.class, () -> {*

*new Task("2225678900", "Task Name", null);*

*}, "Task description cannot be null.");*

*// Test description exceeding 50 characters*

*assertThrows(IllegalArgumentException.class, () -> {*

*new Task("2225678900", "Task Name", "This description is way too long and should throw an exception because it exceeds 50 characters.");*

*}, "Task description cannot exceed 50 characters.");*

*// Test description exactly 50 characters (valid case)*

*assertDoesNotThrow(() -> {*

*new Task("2225678900", "Task Name", "This description is exactly 50 characters long!!!!!");*

*}, "Task description should be valid if it is exactly 50 characters.");*

*}*

During coding, it is important to focus on the intent of requirements, not just their wording. For example, the Appointment class required a description under 50 characters that couldn’t be null or empty. I worked to overcome the bias of assuming my code was correct. Usage of Junit tests helps verify the correct exceptions were thrown, ensuring meaningful feedback in testing run.

Another step that could be performed is the technical debt, or the ongoing process that requires regular attention. Security vulnerabilities can pop up suddenly & missed use cases can lead to unforeseen problems. There are tools available such as SonarQube for static & dynamic testing, while UserVoice is a feedback tool that can help developers identify & resolve issues. If an engineering team can dedicate 20-30% of an engineer’s time to testing, they can maintain high-quality software.

Beyond their immediate benefits, JUnit tests also serve as an indicator of code quality. Mature software with high test coverage tends to be more sustainable and reliable. Low test coverage, on the other hand, should be treated as technical debt. In this application, the JUnit tests acted as a “second set of eyes,” catching instances of improper copy-pasting and ensuring that the code behaved as intended. To derive the greatest benefit from unit tests, it is crucial for developers to set aside pride and rigorously challenge their assumptions, striving to prove that the code functions correctly without unintended side effects.