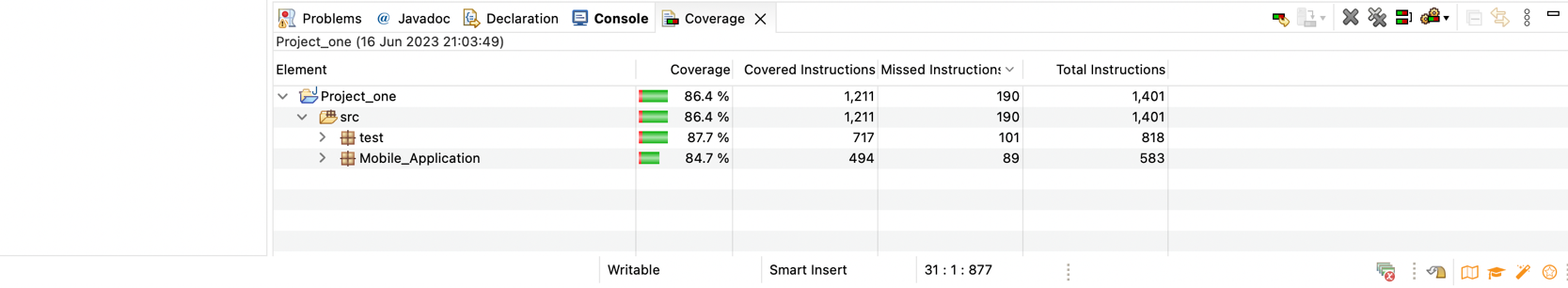
**Reflection on Unit Testing Approach**

**By Mariam Haji**

I followed a systematic unit testing approach for each of the three features. Firstly, I identified the specific functionalities and scenarios that needed to be tested. Then, I designed test cases to cover different inputs, boundary conditions, and potential edge cases. Finally, I implemented JUnit tests with appropriate assertions to verify the expected behavior of the code.

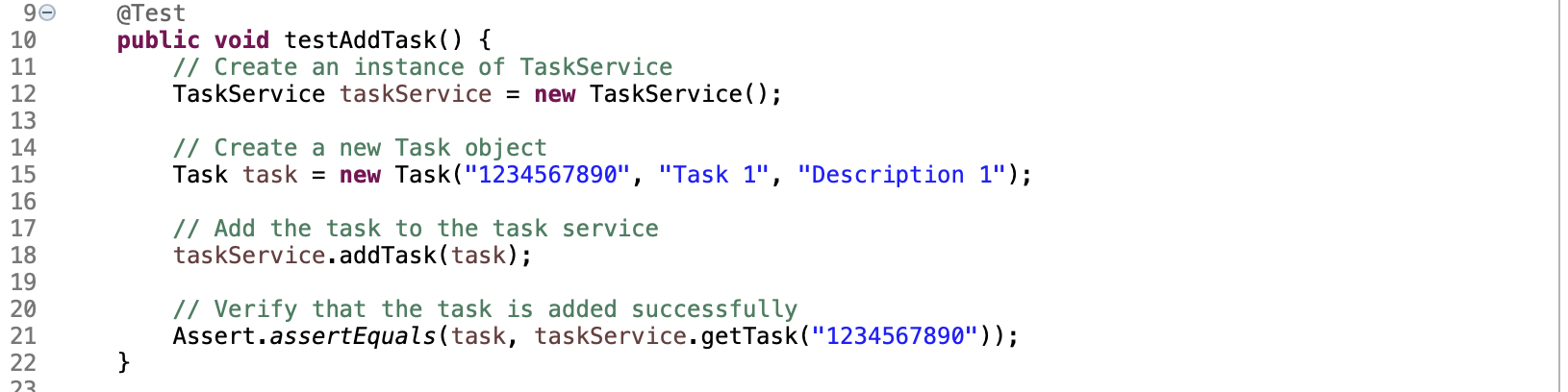
To align with the software requirements, I ensured that the tests generated expected outputs from the software. For example, in the Contact Class, I tested the unique contact ID, firstName, lastName, phone, and address fields to meet the requirements. In the Contact Service, I validated the ability to add contacts with unique IDs, delete contacts by contactId, and update specific fields per contactId, such as firstName, lastName, PhoneNumber, and Address. Similarly, for the Task Class, I tested the unique task ID, name, and description fields, and in the Task Service, I verified the ability to add tasks with unique IDs, delete tasks by taskId, and update specific fields like name and description per taskId. Additionally, in the Appointment Class and Appointment Service, I tested the unique appointment ID, appointmentDate, and description fields, and ensured the ability to add and delete appointments by appointmentId. By conducting these unit tests, I can confidently state that the functionality of each feature aligns with the specified requirements.

The quality of the JUnit tests was assessed based on the coverage percentage. I used code coverage tools such as EclEmma to measure the extent to which the tests covered the codebase. By analyzing the coverage report, I could identify any areas that were not adequately covered by the tests and address them accordingly. The coverage percentage which came to a total of 86.4% (fig 1.) served as evidence of the effectiveness of the tests in ensuring code coverage.

**

Writing JUnit tests was a challenging yet valuable experience as it helped me thoroughly understand the functionality and behavior of the code. By following best practices of unit testing. I was able to ensure that each test method focuses on testing a specific aspect or behavior of the code and that the tests are independent of each other. I used assertions to verify the expected results and designed the tests to be easily maintainable and readable by using descriptive test method names and appropriate annotations.

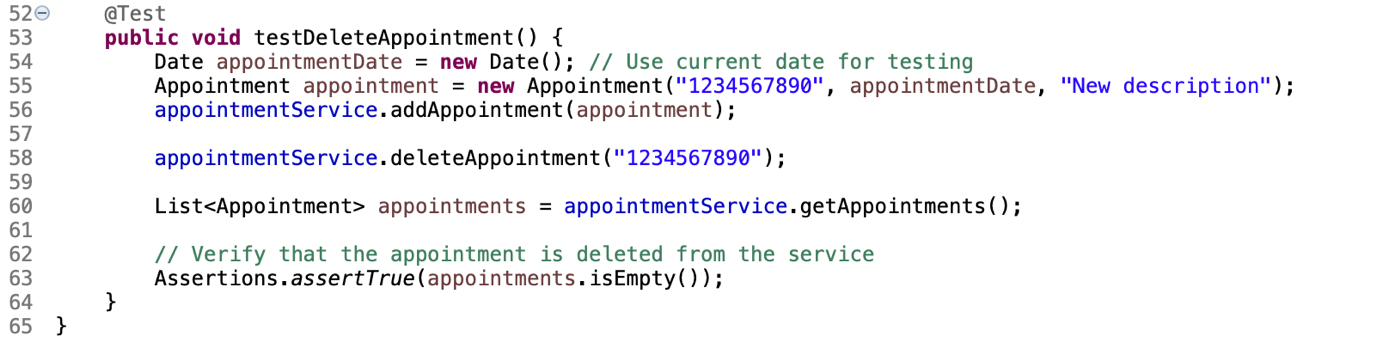
To ensure technical soundness, I reviewed the code and the corresponding unit tests. In the tests, I checked if the methods under test were being invoked correctly and if the expected results matched the actual results. For example, in the TaskServiceTest class, the correctness of the addTaskt() method can be ensured by verifying that the task is added successfully.



To ensure technical soundness, I reviewed the code and the corresponding unit tests. In the tests, I checked if the methods under test were being invoked correctly and if the expected results matched the actual results. For example, in the AppointmentServiceTest class, the correctness of the addAppointment() method can be ensured by verifying that the appointment is added to the list and that the size of the list increases by one.



I ensured my code was efficient by evaluating the performance of the code and the tests. For instance, in the AppointmentServiceTest class, the efficiency of the deleteAppointment() method can be evaluated by checking the time complexity of the operation. If the code uses efficient data structures or algorithms to perform the deletion, it indicates a focus on efficiency.



Some of the testing techniques I used are functional testing which verifies that the functions and features of the software meet the specified requirements. As well as boundary testing which evaluates the behavior of the software at the boundaries of input values.

I did not use performance testing that assesses the performance and responsiveness of the software under different load conditions, neither did I use security testing to identifying vulnerabilities and weaknesses in the software's security measures.

Functional testing is crucial to ensure that the software functions correctly and meets the specified requirements. In addition to that it helps identify bugs and ensures the software behaves as expected. Boundary testing is useful for testing edge cases and ensuring that the software handles extreme or critical inputs correctly. It helps prevent issues that may arise due to invalid or unexpected inputs.

In working on this project, a cautious mindset was essential as a software tester. It was crucial to appreciate the complexity and interrelationships of the code being tested to identify potential areas of failure or unexpected behavior. For example, when testing the Appointment class, I needed to consider scenarios where the appointment date was in the past and ensure that the code handled such cases correctly.

To limit bias in code review, I separated my role as a tester from being the developer, I ensured that the code was evaluated impartially and this helped me to focus on the expected behavior defined by the requirements and verified the code met those expectations.

When testing your own code, bias can be a concern as you may overlook potential issues or make assumptions about correctness. It's important to have independent testing to mitigate bias and ensure a comprehensive evaluation.

Being disciplined in the commitment to quality is crucial for software engineering professionals. Cutting corners in writing or testing code can lead to technical debt, causing future issues and maintenance difficulties.

To avoid technical debt, I prioritized code quality, invested in proper testing and code reviews, addressed issues promptly, and practiced regular refactoring and continuous integration. This approach helped to minimize technical debt, improves efficiency, and produce a more robust software product.

**References:**

Software Testing Help. (2023). What Is Technical Debt And Why QA Testers Should Be Concerned About It? Retrieved from: <https://www.softwaretestinghelp.com/technical-debt-and-qa/>

Whatis. (2023). technical debt. Retrieved from: <https://www.techtarget.com/whatis/definition/technical-debt#:~:text=Technical%20debt%20is%20a%20common,is%20sometimes%20called%20code%20debt>.

Javapoint. Unit Testing. Retrieved from: <https://www.javatpoint.com/unit-testing>

BrowserStack. Software Testing Techniques: Explained with Examples. (2023). Retrieved from: https://www.browserstack.com/guide/software-testing-techniques