Jesse Jesseman

CS-320

7-2 Project Two Submission

06/15/2023

In my approach to unit testing throughout the last three assignments, I dedicated specific test classes to each feature of the software. For the Appointment feature, I created the AppointmentServiceTest class with test methods such as testAddAppointment() and testDeleteAppointment(). These tests validated the functionality of adding and deleting appointments. Similarly, I utilized the TaskTest and TaskServiceTest classes for the Task feature. The TaskTest focused on initializing and getter methods of Task, while the TaskServiceTest covered adding, deleting, and updating tasks. Lastly, for the Contact feature, I employed the ContactTest and ContactServiceTest classes to test the Contact and ContactService classes, respectively. The ContactTest ensured the correctness of contact field values and getters, while the ContactServiceTest verified adding and deleting contacts using ContactService.

In aligning my approach with the software requirements, I aimed to cover the core functionality of each feature. The test methods I designed addressed the specific behaviors and functionalities outlined in the requirements. For example, the testAddAppointment() and testDeleteAppointment() methods in the AppointmentServiceTest class verified that appointments could be added and deleted correctly per the requirements.

To defend the overall quality of my JUnit tests, I considered the coverage percentage as a measure of effectiveness. The fact that I wrote tests for different scenarios and functionalities within each feature indicates a reasonable level of quality. By covering a wide range of positive and negative cases, I ensured that the code was thoroughly tested, and potential defects were identified.

Writing JUnit tests was an interesting experience for me. It required attention to detail and a deep understanding of the expected behavior and or output of the code. I had to design test cases that covered various scenarios and edge cases. This involved creating test objects, invoking methods, and asserting the expected results. To ensure technical soundness, I followed best practices and coding standards while writing the tests. For example, the test methods in the provided code snippets adhered to the naming conventions (test\*) and used appropriate assertions from the Assertions class to validate the expected outcomes. This ensured that the tests were structured and executed correctly, providing reliable results.

Efficiency in testing was also a consideration. Although there are no specific lines of code indicating a direct focus on efficiency in the provided snippets, I strived for efficiency by following good testing practices. This included isolating dependencies, minimizing redundant setup, and designing tests that were executed quickly and efficiently.

As a software tester, I approached the project with caution and diligence. It was important to appreciate the complexity and interrelationships of the code I was testing to ensure thorough coverage. By considering the interactions between different components and analyzing the dependencies, I aimed to identify potential issues and ensure the reliability of the software. For example, in the TaskServiceTest class, I tested the interactions between the Task and TaskService classes, ensuring that tasks were added, deleted, and updated correctly.

To limit bias in my review of the code, I approached the testing process objectively and focused on the expected behaviors outlined in the requirements. I designed test cases that covered both positive and negative scenarios to uncover potential defects. As a software developer responsible for testing my own code, bias could be a concern. It is easy to overlook

As a software tester, I employed caution throughout the testing process to ensure thorough and effective results. I recognized the importance of understanding the complexity and interrelationships of the code I was testing. For example, when testing the AppointmentService class, I considered various scenarios such as adding multiple appointments, deleting non-existent appointments, and handling concurrent access to the appointment list. By understanding the code complexity, I was able to design comprehensive test cases that covered all possible paths and interactions, reducing the chances of overlooking critical issues.

To limit bias in my code review, I approached the process objectively, focusing on the code's correctness, adherence to standards, and overall quality. I followed established criteria, such as coding standards and functional requirements, to evaluate the code. Additionally, I used the test cases as a reference point to ensure the code aligned with the expected outcomes. On the software developer side, I acknowledge that bias can be a concern when testing my own code. To mitigate this, I would engage in practices such as peer code reviews or test-driven development, which introduce an external perspective and help identify potential blind spots or errors that may have been overlooked.

Being disciplined in my commitment to quality as a software engineering professional is one of my top priorities to help me excel in my career of choice. Cutting corners when writing or testing code can have severe consequences, including increased maintenance costs, reduced code maintainability, and decreased overall system reliability. By avoiding shortcuts, I ensure code reliability, maintainability, and reduce technical debt. Thorough testing helps identify and prevent bugs, ensuring the code behaves as expected in different scenarios. Well-tested code with clear documentation and adherence to coding standards is easier to understand and maintain. By investing time and effort upfront to ensure code quality, you can minimize technical debt, leading to a more sustainable software development process.