7-2 Project Two

After developing JUnit tests for the different features, the approach that aligned with the software requirements was to test the key components of the features. Instead of testing less relevant methods, such as getters and setters, which are common parts of the code that can fail, I started testing the behavior of the key components by using distinct inputs, including wrong and null arguments. This way, I ensured that the tested code was working as expected and aligned with the requirements. Regarding the quality of the JUnit tests, my testing code was intended to cover the most critical components of the features by aiming for a coverage of 50% to 80% to consider it effective. According to what I learned, it is not necessary to reach 100%, especially in big company systems such as Salesforce, because it wastes time and effort, as there is always code that commonly does not need to be tested, such as getters and setters, as I mentioned earlier. Also, testers should verify through the JUnit coverage that the key code pieces they are testing are included in the test cases to ensure that the code is tested as intended. All in all, avoiding testing unnecessary code and focusing on critical components was the approach I used to test each feature.

There are sections where my JUnit tests demonstrate soundness and efficiency. First, an example of sound testing code is the following excerpt from TaskTest.java:

@Test

**public** **void** testTaskConstructorIdParameter() {

*assertAll*(

() -> *assertThrows*(IllegalArgumentException.**class**, () -> **new** Task(**null**, "Assignments", "I am lazy.")),

() -> *assertThrows*(IllegalArgumentException.**class**, () -> **new** Task("12345678990", "Assignments", "I am lazy."))

);

}

In this code, I test the Id parameter of the Task class by creating objects with invalid parameters. The first object has a null Id, and the second has an Id longer than 10 characters, which should not be allowed according to the requirements that were followed. This way, I tested potential errors that can occur in this key feature. Next, another example demonstrating efficient code is the following excerpt from ContactTest.java:

@Test

**public** **void** testGetAndSetFirstName() {

Contact contact = **new** Contact("1234567899", "Jean", "Farfan", "4843334444", "6665 S Hello World Rd");

contact.setFirstName("Jane");

*assertEquals*("Jane", contact.getFirstName());

}

In this code, I test the getter and setter for the FirstName parameter in the same test block because it makes the entire code file more readable and tests both non-key components. By adding these types of tests, I achieved more than 80% coverage in JUnit to meet the requirements of each feature for Project One. Both sound and efficient examples are reflected in the test cases created to comply with the requirements and demonstrate that the features are fully tested.

The testing techniques I used were Unit testing and Experience-based testing. First, Unit testing technique focuses on testing individual parts, or units, of the software to check if they are doing what they are supposed to. For example, in my code, I wrote tests to check if the Task constructor worked correctly. I made sure it threw an error if the Id was null or too long. Next, Experience-based testing relies on testers' knowledge and intuition to identify potential issues in the software. An example of this is seen in the approach I took when testing the Contact class. I explored various scenarios, such as invalid inputs for parameters like ID, first name, and phone number, drawing on my experience to anticipate potential issues and avoid testing unnecessary code parts. In other words, I reviewed the software while testing to find any problems that might not be obvious. Hence, Unit testing and Experience-based testing help me to test all the features to ensure that the code meets the requirements.

The techniques I did not use were Boundary value testing and Dynamic testing. Boundary value analysis focuses on testing the boundaries of input values to identify any defects. This technique includes testing the software at the boundaries between valid and invalid input values. For example, if a system accepts values from -777 to 666, boundary value analysis would test inputs like -778, -777, 666 and 667 to ensure that the software handles boundary conditions correctly. It can be valuable for identifying issues related to boundary conditions that might not be caught through other testing methods. An example can be financial applications where precision and accuracy in numbers are critical; boundary value analysis helps ensure that calculations and transactions are processed correctly with correct input ranges. On the other hand, Dynamic testing techniques, such as exploratory testing, are executing tests based on the dynamic behavior of the application under test. Exploratory testing works simultaneous test design and execution, where testers explore the software to uncover defects without following a predefined test plan. This technique can be useful when detailed specifications are unavailable or incomplete, making it suitable for Agile development environments or for testing user interfaces and user experiences. While I did not use exploratory testing in the features, it can be useful for identifying unexpected behaviors or issues that may not be shown through normal test cases. For example, in a system where user interaction plays an important role, exploratory testing can help uncover issues that might affect user satisfaction.

During this project, I adopted a cautious mindset, especially when testing core components. I understood the importance of thoroughly testing the key components of each feature, rather than focusing on less critical methods like getters and setters. For example, when testing the Contact class, I made sure to cover various scenarios, such as invalid inputs for parameters like ID, first name, and phone number. Understanding the complexity of the code was crucial in ensuring that all possible interactions and dependencies were accounted for. Hence, in the TaskServiceTest, I tested adding, deleting, and updating tasks, considering how these operations could affect the overall functionality of the system.

To limit bias in my review of the code, I approached testing with an objective mindset, focusing on verifying that the code met the specified requirements rather than imposing personal preferences. If I were to test my own code as a developer, bias could become a concern, as I might unintentionally overlook potential flaws or assume that the code behaves as intended without thorough testing. One example was in the TaskTest class, where I ensured that the Task constructor handled invalid inputs correctly, regardless of any biases I might have had regarding the expected behavior of the code.

Being disciplined in my commitment to quality as a developer and tester is crucial. Cutting corners when writing or testing code can generate bugs, errors, and unsatisfied users. For example, ignoring thoroughly test edge cases in the ContactService class could result in unexpected results when interacting with contacts in the code. To avoid consequences in my future work, I plan to follow coding standards, regularly review and refactor code, and prioritize thorough testing at every stage of development. This disciplined approach will ensure that the systems I will develop remain stable and maintainable over time.