Project Two: Summary and Reflection Report

Harshilkumar Jayswal

CS-320 @SNHU

**Summary**

During the development of the mobile application, I used a testing approach called JUnit testing which involved testing individual methods and components of each feature to ensure that they met the specified requirements. I performed this testing as a developer and an end user. Once any functional module was developed, I took on the role of the end user and started thinking critically about the short comings and possibilities. Once I identified the possible breaking points in my code, I was able to take on the developer role and fix them. I repeated this cycle throughout the project development to ensure smooth execution of the software.

Contact feature had certain sets of functional requirements, for example one requirement was to not allow null values for the name fields. I developed two JUnit test cases where null values were being passed to ensure that an exception is thrown. Below JUnit test case checks whether the setLastName() method of the Contact class correctly throws an exception when an invalid null value is passed as an argument. This test threw an exception at runtime which was expected and passed the test case.

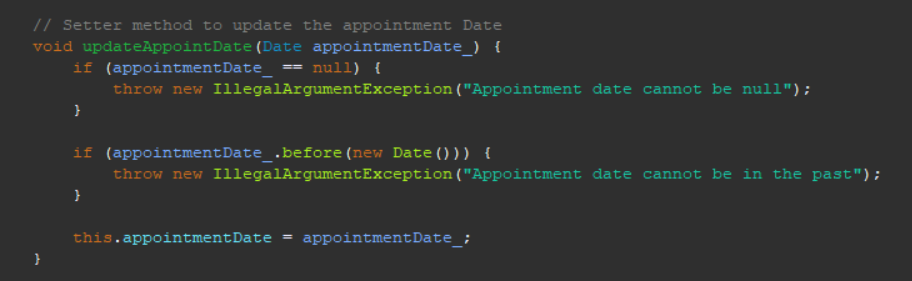
Text

Description automatically generated

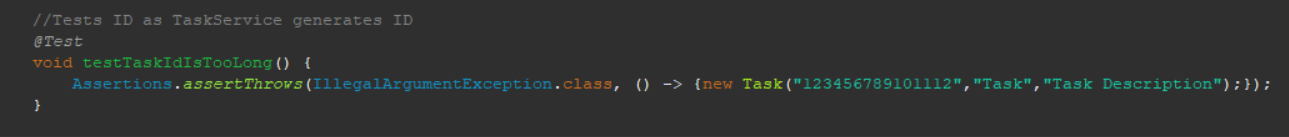
Similar test cases were implemented in Appointment features of the app. For example, one of the requirements was to not allow a date that was set in the past. I developed a JUnit test case that compares the date entered to the current date and throws an exception if it is in the past. Below is the JUnit test case to catch the exception where an old date is passed. Once passed, Appointment setter method throws an exception and validates out test case.

Text

Description automatically generated



And similarly, Task class had a requirement that taskID cannot be longer than 10 characters, I implemented JUnit test that catches an exception when a task ID longer than 10 characters.



Text

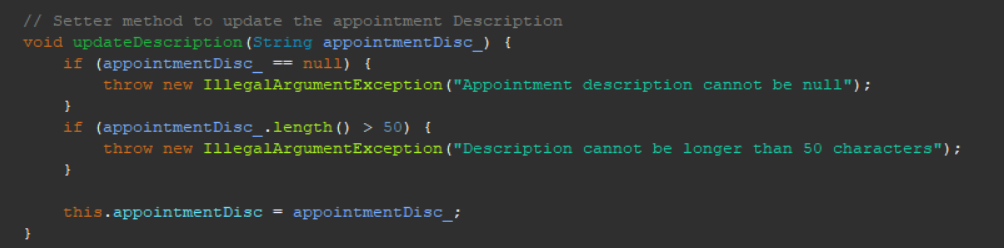
Description automatically generated

The JUnit tests coverage was 90.2%. While a high coverage percentage indicates that many code paths have been evaluated, it does not guarantee that all edge cases, error conditions, or unexpected scenarios have. So, I covered every scenario and implemented tests that an end user can encounter. I made sure that all functional requirements were tested extensively. I included assertions that covered all outcomes of the method being tested, such as checking for null inputs, valid outputs, and error handling. To ensure that my code was efficient, I made use of parameterized tests and test cases to reduce redundancy and increase test coverage.

Text

Description automatically generated with medium confidence

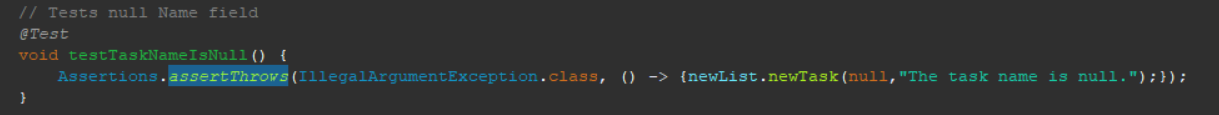
To ensure that my code was technically sound, I followed good coding and design practices while keeping the code well-organized and easy to understand. For example, in the Appointment class below, the updateDescription setter method, I validated before updating the description to check if the input is null or is longer than 50 characters as per the requirements. If so, I threw exceptions and if not, exception was thrown the description was updated. This validation ensures that the data is in the expected format and helps prevent potential errors or data inconsistencies. The code is simple without any redundancy to be as efficient as possible.



**Reflection**

**Testing Techniques**

For this project I employed unit testing practices such as JUnit testing. In summary, JUnit is an open-source testing framework for specific code. It reduces the debugging time at the end of the project. JUnit testing also provided coverage analysis that involved measuring the percentage of code covered by the tests to ensure that all code paths were evaluated.



Here, the annotation "@Test" instructs JUnit to use this as a test method. Following the convention for JUnit test method names, "void testTaskNameIsNull" is used. The constructor throws an "IllegalArgumentException" when a null value is given, from the assertion made by the test using the "Assertions.assertThrows()" function. The expected exception class and a "newList" object with a non-null value for the Description field and a null value for the Name field are the two inputs that the "assertThrows()" method accepts. I made sure that an exception is thrown when anticipated by testing this behavior.

Black Box testing is one software testing technique that I did not employ for this project. It tests the functionality of a system without requiring knowledge of its internal workings. In other words, the tester is not concerned with how the system is implemented, but instead focuses on the inputs and outputs of the system. Tester will perform specified tasks which can mimic end user experience. This provides an unbiased review of the software which would be valuable to developers to tweak and tune it. Black box testing is frequently used to assess a system's user interface as well as its performance and functionality. It is a good technique to find glitches or flaws in the interconnected systems and make sure it is operating as it should.

Early bug detection with JUnit testing can help reduce time and costs at the end of the SDLC. This assists in identifying unaccounted issues that could emerge following frequent changes in code. JUnit testing can be automated, and the green passing response message makes it simple to verify success. This method is essential to the continuous integration and delivery cycles of modern software. Thus, development cycles are shortened, and scalability is enhanced.

Meanwhile developers can gain meaningful insight into user experience from Black Box testing. Since the testers are unfamiliar with the inner workings of the software, training them can be completed quickly and at a lower cost. The fine-tuning and optimization made possible by this testing can enhance user experience. Additionally, it makes sure that all systems are operating as intended.

**Mindset**

Acting as a software tester, I employed caution when identifying potential bugs and errors in the developed code. I frequently switched my mindset between a developer and tester to think of all scenarios that an end user might encounter. I made sure to test every result of each method because I was aware of the complexity and interrelationships of the code being tested. It is helpful to always think of what is being tested in terms of how it will impact other code or systems. As tester, you should know that the code you are testing is often interdependent, meaning that changes in one part of the code can affect the behavior of other parts of the application. You can also identify potential areas of weakness in the application, which may need more thorough testing. Understanding the code also makes it easier for you to communicate with developers to refine it. For example, below the AppointmentService is using an object from Appointment class to search and remove the appointment with the specified ID. While testing the deleteAppointment method, the tester should keep in mind the inheritance properties between the two classes and how it can impact objects other than the one being tested.

A screenshot of a computer

Description automatically generated with medium confidence

To limit bias in my code, I made sure to approach each test case objectively and follow a systematic process that covered all outcomes. I carefully examined the test cases and ensured that each assertion was testing the intended behavior. Bias can cause gaps and make it harder to identify potential issues, so I made sure to lessen the effect by adhering to a strict procedure and carefully evaluating the code. For example, I employed similar validation for null, too long inputs, unique values, and invalid inputs throughout all three classes. Appointment, Task and Contact classes have the same field validation in the methods and similar JUnit test cases for each.

To accomplish this project there was a certain discipline that was required of me as a software developer. Software that is not well-designed, written, or tested can lead to significant problems, including bugs, crashes, security vulnerabilities, and system downtime. These issues can have a negative impact on end-users and can damage a client's reputation, resulting in loss of revenue and customer trust. Hence, it is my obligation to adhere to the best coding principles and structure my code as much as I can so that anyone in the future can easily understand it and expand upon it. Cutting corners when it comes to writing or testing code can lead to technical debt. This is also referred to as “We will fix it in post-production.” When developers cut corners or miss potential bugs, technical debt can quickly accumulate. Technical debt increases in cost in terms of time, effort, and resources to rectify the longer it is ignored. To avoid this, the quality of the code must be prioritized. By following industry standard continuous integration and continuous delivery (CI/CD) practices, we can avoid technical debts. This involves continuous testing in cycles, refactoring, and code review.

**References:**

Hambling, B., Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). Software testing an Istqb-Bcs certified tester foundation guide - 4th edition. BCS, The Chartered Institute for IT.

Junit - Test Framework. Tutorials Point. (n.d.). Retrieved March 31, 2023, from https://www.tutorialspoint.com/junit/junit\_test\_framework.htm

The QA Lead, & Boog, J. (2023, February 6). 9 types of software testing in software engineering. The QA Lead. Retrieved March 31, 2023, from <https://theqalead.com/test-management/types-of-software-testing/>

Atlassian. (n.d.). What is continuous integration. Atlassian. Retrieved April 15, 2023, from https://www.atlassian.com/continuous-delivery/continuous-integration

GeeksforGeeks. (2022, June 28). Coding standards and guidelines. GeeksforGeeks. Retrieved April 15, 2023, from https://www.geeksforgeeks.org/coding-standards-and-guidelines/