Robby Ormond

2/22/2024

CS320 – Project 2

The approach that I used for unit testing each of the three features included ensuring that my approach fulfilled the requirements in the rubric for the assignments. This was followed in different scenarios such as in the contact class there was a requirement to validate the input to be below 10 characters.

private void validateInput(String value, String field) {  
 if (value == null || value.length() > 10) {  
 throw new IllegalArgumentException(field + " invalid null or length greater than 10");

The Junit test involved with testing this method would then ensure that this is followed independently of other code in the program. The overall quality of the Junit testing was new to me as was the use of Junit testing in general. By running the tests and showing test coverage I was able to know how effective the code was by seeing the percentage of code that was being checked by the tests that I implemented. When checking to see how impactful the Junit tests were in the code I referred to the rubric for what I should strive for, and seeing that an 80% is what was expected in the rubric provided a good standard to chase for the code coverage moving forward in the Junit testing.

When ensuring the code was technically sound I ensured that there was proper sample data being used for future testing and following Junit framework to validate these methods. For example in the Contact class the following code;

public void testContactConstructorAndGetters() {

// Create a Contact object with sample data

Contact contact = new Contact("12345", "John", "Doe", "555-1234", "123 Main St");

// Validate the values returned by getter methods

assertEquals("12345", contact.getContactID());

assertEquals("John", contact.getFName());

assertEquals("Doe", contact.getLName());

assertEquals("555-1234", contact.getPNumber());

assertEquals("123 Main St", contact.getAddress())

In the example from my code that I provided it shows that there was sample data provided and when tested can provide feedback on whether the class behaves as expected, therefore ensuring its technical soundness when following the same strategies in further code. When ensuring that the code was efficient, I found that outside resources like youtube and google were invaluable at learning more concise and efficient methods to ensure that the code wasn’t too convoluted as well as worked well. When making the code, as well as the Junit tests, I made a point to run each test as I wrote it before I got too far into the code. Another strategy that I used to ensure concise code that worked well was declaring variables before they were called.

Software testing techniques that I employed in this project included following the Junit framework to allow easy testing of each function. Beyond following the Junit framework I also ensured that the test method covers positive test cases to get proper feedback on the return of expected values when provided with valid input data. A lot of the software testing techniques that I used through this class were new to me however one of the strategies that I followed that persisted from previous experience was just maintaining clear code and ensuring the variables were declared as well as provided in text comments as I progressed through the code to easily identify uses and keep the code clean.

Other software testing techniques that I did not use for this project included integration testing, boundary testing, negative testing, concurrency testing, and performance testing. Of the testing techniques provided, starting with integration testing the practical uses and implications are extremely vital for large scale projects and involve ensuring the correct functionality of software between different components or modules. The real world implications of integration testing involve ensuring overall reliability in the software system, preferably before deployment. When looking at the practical uses of boundary testing, it helps identify the minimum and maximum values that are acceptable without creating issues such as buffer overflows or other performance related issues.

The impact and implications that boundary testing provides includes ensuring the overall resilience of the software and limiting vulnerabilities as well as data loss. The practical use of negative testing includes covering error handling and exceptions ensuring that these incorrect inputs don’t create further issues or detriment to the system. A lack of negative testing could lead to vulnerabilities in the software and proper testing could allow the system to handle the error exceptions properly. Concurrency testing involves evaluating software behavior under concurrent execution and shows that although multiple processes are running, they don’t interact negatively on the system. Implications of concurrency testing are that neglecting this form of testing can lead to errors and performance issues as well as reliability issues that are only shown when multiple processes are running. With performance testing, the practical uses shown are for identifying performance bottlenecks as well as helping identify where the code could be optimized for a better user experience. The implications of performance testing include ensuring that the system works as expected while mitigating slow response times or excessive resource consumption. Overall, these testing techniques used in tandem are crucial for scalability in larger scale projects and ensure proper functionality in the system.

The mindset that I adopted when working on this project was taking time and ensuring that each test worked before moving on. Learning how Junit testing works was extremely satisfying having not known of it previously. It is an extremely valuable asset to ensure that the code works as intended exclusive to the rest of the code which made debugging a dream in comparison to previous experiences. In acting as a software tester, I employed a lot of caution in ensuring that I tested all the paths as well as checking code coverage. It’s extremely important to appreciate the complexity and interrelationships of the code at hand because they include intricate interactions as well as identifying the different dependencies involved. Specifically, all these things combined are extremely important to take into consideration when designing test cases for the code to ensure proper coverage.

The ways that I tried to limit bias in my review of the code is by trying to take a step back when looking over the code after writing it, I don’t think its possible to fully eliminate bias in reviewing code however there are some things that can be taken into consideration to limit it. Specifically focusing on measurable factors proved beneficial when reviewing the code and focusing on upping the testing coverage of the code seemed to be the most fruitful method of limiting bias. As a software developer testing my own code, I do believe that it is a concern to move forward with bias as one could overlook problems that could arise or hyperfocus on specific parts of the code, which is something I personally have dealt with.

When evaluating the importance of being disciplined in your commitment to quality as a software engineering profession its crucial to not cut corners because its significantly easier to work diligently and ensure things work properly the first time when its still fresh rather than reconvene and fix after the fact. I plan on avoiding technical debt as a practitioner in the field by maintaining clean code practices ensuring easy readability while providing good in text citations. Specifically proper testing in code as well as documentation is crucial to avoid technical debt as it is significantly cheaper to find and debug previous to deployment.