Frederick Brehm

6/20/2024

Southern New Hampshire University

CS 320 Project 2

Summary

In my task of creating a mobile web application for customers that allows that contains a contact, task, and appointment services. The main purpose of these services is the ability to add, delete, and update tasks and appointment objects within the mobile application. The contact service class shall have a unique ID that is not updatable nor null, firstName string, and lastName string that cannot be null, and neither of these can be more than 10 characters in length. The class will also have a phone number that must be exactly 10 numbers long and not null, and an address class that cannot be longer than 30 characters and cannot be null. The taskService class must have a unique task ID that is not updatable, is not null and cannot be longer than 10 characters. Task shall contain a name string field that cannot be longer than 20 characters and not null, and a required description field that is no longer than 50 characters in length. The appointmentService class will have a unique ID not longer than 10 characters and cannot be null or updatable. The class will contain a date field which cannot be in the past and cannot be null. There must also be a required description field that can be no longer than 50 characters in length and shall not be null. All the service classes will be able to add, delete or update its parameters dependent on the service. I was able to ensure that these requirements were met by using the following code,

if(id == null || id.length()>10) {

throw new IllegalArgumentException("Invalid id");

which I utilized for each service to adhere to the requirements for each contact, appointment, and task. For example, during the Junit test period I utilized.

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Contact("123456", "Bob", "Lee", "123 Avenue Lane", "11122233333");

to test whether the input was too long and whether to accept or deny the input. These assertions were used for each contact, task, and appointment service to meet the requirements set.

The quality of my Junit tests was off to a rocky start. I had issues when performing the Junit tests and effectively seeing coverage throughout the specific tests. I had to do some research and search into other resources where I realized that at first, I was testing the classes wrong. I was unsure of how to thoroughly test, and once figured out I was able to see that the Junit tests were effective due to the coverage percentage. My contact class had a higher coverage percentage which showed that the class was more effective but there was still a smaller coverage in the others. Even seeing the smaller coverage, I was still happy that I was able to correctly run the Junit tests as intended.

To ensure that my code was technically sound there were a few key areas I wanted to focus on. This included making sure that the code for each service was organized, readable, and full of comments so that any developer could go through it one day and understand what the purpose of the code was. On a more technical side of the code, I focused on utilizing lists like in the appointmentService class, for example.

// create new list for ApptService

public ApptService() {

appointments = new ArrayList<>();

which would make it easy to store and add, delete, or update any appointments for that service. I also focused on utilizing common algorithms such as add, matches, and length. In contact class you can see this by using “id.length()>10;” or by “phoneNumber.matches(\\d{10}).” These expectations plus using assertions like assertTrue, assertEquals, and asserThrow would help keep the code universal and clean when writing each of the different services classes.

Coding is still relatively new to me and overly complex so for my code to be efficient I had to use several resources. I researched several tutorials and outside articles to understand problems and how to fix those problems through the proper channels. Many times, this involved using trial and error until I was able to figure out how to use the correct code such as methods and variables in the services. I was also new to Junit tests so utilizing these resources also helped me to learn and understand the proper way to utilize a Junit test. In certain classes I used correct test input and incorrect test input to tests that the code was working for both. The code

Assertions.*assertThrows*(IllegalArgumentException.class, () -> task.setDescription(tooLongDesc));

tested to see if the description was too long based off a set of variables that were implemented at the beginning of each class. This would test to ensure the service parameter either met or did not meet the goal. This was utilized for several of the service classes throughout the coding and testing phase.

Reflection

The type of software testing techniques that were utilized for this project and the previous milestones would consist of black box and white box techniques. Black box testing is a type of software testing in which the tester is not concerned with the software’s internal knowledge or implementation details but rather focuses on validating the functionality based on the provided specifications or requirements. (GeeksforGeeks, 2024)White box testing analyzes the internal structure, data structures, internal design, code structure, and the working of the software rather than just the functionality as in black box testing. (GeeksforGeeks, 2024) In believe to have been utilizing white box testing over black box as we were specifically testing to meet a provided specifications or requirement but were responsible for the inner workings and data structures that would require the application to function and test properly.

One of the less popular testing techniques that were not used for this project is known as Experience Based Testing. This type of testing is used for executing testing activities with the help of experience gained over the years or on similar projects done in the past. Domain skill and background are major contributors to this type of testing. There are two types of experienced based testing techniques, error guessing and exploratory testing. Error guessing is used by a tester who has exceptionally good experience in testing or with the application that is under test, hence they may know where a system will have a weakness. This technique is not effective, however when used as a stand-alone technique, it must be used along with structured techniques. Exploratory testing is hands-on-testing where the aim is to have maximum execution coverage with minimal planning. The key aspect of this type of testing is the tester learns about the strengths and weaknesses of an application under the test. Like error guessing, it is not effective when used as a stand-alone technique but effective when paired with a structured technique. (GeeksforGeeks, 2024)

Black box testing, white box testing, and experience-based testing all have many practical uses and implementations. More specially, black box testing is going to be utilized primarily by a software tester. Black box testing revolves around the exterior functionality of the software itself and is preferred when testing a product where the inner workings of the software do not need to be known. White box testing is primarily used by the software developer, as the developer knows the inner workings of the code and data structures as they are the ones who designed it. This type of testing is better suited for a software application as it focuses on mistakes and defects that are made in the inner workings of the code. Experience based testing is practical when used in conjunction with one or more of the structured testing techniques mentioned. It is a good secondary testing technique in the fact that someone who has experience in coding or the specific software being developed can notice mistakes otherwise blind to the less experienced eyes. Each of these are implemented in different ways with different projects dependent on what is needed.

While working on this project I had to adapt to a mindset that extended to logical, optimistic, and growth. As I am not a professional or experienced developer, I had to ensure that I was paying diligence, thinking critically when uncertain about my code, but open to the possibility of mistakes and the growth that would come from it. As a developer I had to employ caution towards all my code. This required me to utilize several resources such as tutorials, articles, and experienced developers and their knowledge. Taking caution each step of the way helped create organized and readable code but created an atmosphere in where I knew I could not rush through what I was writing. It required me to double, and triple check my code for mistakes and defects as well as test the code several times in a trial-and-error fashion to develop better code by working though it little by little. It is important to appreciate the complexity and the interrelationships of the code being tested as each service would have to work interconnected to the other. Even though Junit tests specific bits of code, each piece is there to form a small piece of the puzzle. When dealing with complexity and interconnection it is important to understand that one must be crucial to the details and workings of the code. I saw growth from these methods as when I started the code early on my coverage was less than ideal. However, after practicing these methods and understanding how complex and interconnected the code was, it was easier to ensure that I understood what I was developing. This helped secure higher coverage later.

Unconscious bias can be in direct contrast with the beliefs and values you think you hold. You may not even be aware that you hold these biases, or that they are affecting your attitudes and behaviors. (Murphy, 2023) Bias is constantly surrounding us, no matter if you know it or not or what kind it may be. Many will not recognize they are biased until it is too late and in software development, this can be detrimental. If one doesn’t control their bias they may let mistakes and defects slip by which could cause programs not to run all the way up to a sever catastrophe. To ensure that I was not biased while developing this project I constantly checked over my code as well as performed intermittent testing. For example, while testing the code through Junit I was able to ensure that each variable was being tested for the correct parameters. Without doing this I would not have caught the fact that when checking phone numbers, I had written code which allowed for a phone number up to 10 characters and not exactly 10 characters. Staying unbiased to yourself will help develop a mindset of growth and truth where your code can thrive in conditions where it is correct and efficient.

Discipline when coding refers to the act of writing code that is well-organized, well-documented, and free of errors. It also means following best practices for coding standards and style guides. (Capuzzo, 2023) Having discipline and learning discipline will help develop and conform a person to be the best version of themselves possible. This will help create a routine and environment where they can lead and help others in the same skill set. Discipline as stated above also means following best practices. Cutting corners is a form of laziness or complacency and these two things lead to disaster. Our world has evolved immensely through technology in the last 20 years to where applications are used every day for everything. If software developers are not disciplined this could lead to faulty code, bad testing practices, and overall, disasters both economically and physically. In the “Software Engineering Code of Ethics” it states that “Software engineers shall act consistently with the public interest” and “Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.” (Gotterbarn, Miller, Rogerson, 1997) These are only two of the codes of ethics, whereas there are several more to ensure the safest and best approach to ensuring everyone and everything’s safety. As a field practitioner I plan to avoid technical debt by utilizing safe and consistent practices like agile, meeting high quality standards, and open to growth, to create the highest quality products possible in accordance with the software engineering code of ethics.

References

GeeksforGeeks. (2024, June 13). *Differences between black box testing and white box testing*. https://www.geeksforgeeks.org/differences-between-black-box-testing-vs-white-box-testing/

Murphy, N. (2023, April 20). *Types of bias*. CPD Online College. https://cpdonline.co.uk/knowledge-base/safeguarding/types-of-bias/

The art of discipline in coding, and in learning to code. (n.d.). <https://blog.qwasar.io/blog/the-art-of-discipline-in-coding-and-in-learning-to-code>

Don Gotterbarn, Keith Miller, and Simon Rogerson. 1997. Software engineering code of ethics. Commun. ACM 40, 11 (November 1997), 110-118. DOI: [10.1145/265684.265699](http://dx.doi.org/10.1145/265684.265699)