**Project Two**

# **Summary**

## **Alignment to Requirements**

After developing classes and unit tests for all three unique features in the previous weeks, reviewing the requirements, and creating the test cases, I would say that I aligned the software requirements with the unit testing that was done. To support my claims I will include the requirements that were given to me and explore test cases that were assigned to those requirements. Here is how the specifications for the Contact, Task, and Appointment Classes should look. All of them required a unique ID string that couldn’t be longer than 10 characters, couldn’t be null, or couldn’t be updatable. This was added to the Classes and the Junit tests were created to make sure the requirement is supported. The JUnit test cases for the Contact class included first name, last name, address, and phone number as these were its specific requirements. In the Contact Service class, the Junit test cases were created for creating, deleting, or updating contact information and the tests were successful. The JUnit test cases for the Task class included name and description as these were its specific requirements. In the Task Service class, the Junit test cases were created for creating, deleting, or updating the task’s information and the tests were successful. The JUnit test cases for the Appointment class included date and description as these were its specific requirements. In the Appointment Service class, the Junit test cases were created for adding and deleting the appointments and the tests were successful.

## **Effective Tests**

After creating the test cases and running the Junit and the coverage tests, we get full coverage in all of the classes and their specifications and requirements. This indicates that we successfully developed all the classes and they have corresponding Junit Test classes. Within the test classes, I created test units which are specific tests for each requirement of the mobile application. These test cases fully cover all the possibilities where an exception can be thrown such as when creating an ID it must fulfill multiple criteria points. Those test cases ensure that all of them are met and if a user inputs something wrong an exception is thrown for the ID being too long, being required, cannot be null, or cannot be updatable. These cover all the possibilities giving the test unit full coverage over any possibility. These claims are also supported by the screenshots provided in the first project from last week.

## **Technically Sound Code**

To ensure the code is technically sound, the test cases were divided into smaller parts where I tested a specific function. As one of the requirements in the code is that the information cannot be null, I tested if adding a null value instead of each of the user inputs would result in an exception being thrown. This was successful and made the code technically sound and clear.

## **Efficient Code**

To ensure that the code and test cases are efficient, there were getter and setter functions created and tested for their functionality. The main part of testing the efficiency of the code was the adding, deleting, or updating functions. Test cases were created to see if adding, deleting, and updating either the contact, task, or appointment with valid or invalid input worked as expected. This was all tested successfully in a covered test and it also ensures that the code is efficient.

# **Reflection**

## **Techniques Employed**

Software testing techniques that were employed during the entirety of the project were static testing, white box testing, and unit testing. Unit testing was used as a main source of the testing bundle because using JUnit testing was a main part of both of the milestones. It is used to run automated and repeatable tests. Its main purpose is to test how the classes and methods we created behave. To evaluate the quality of the code I used static testing. This type of test is used before running the code. I as a developer and tester went over the classes and methods I created and tried to look for any easily recognizable errors in the code. These errors can be typos in the code language, syntax errors (forgotten closing bracket or semi-colon, etc.), and others. The last type of testing that was employed is White Box testing. As mentioned in the previous type of testing, I was the developer and tester. This allows me access to the source code and therefore the white box testing is used. White box testing can be explained as running the code while knowing how the classes and methods behave. Doing this is allowing us to have expectations of how the code works and we can create test cases covering all of the possible outcomes.

## **Other Techniques**

Here are some techniques that were not used during the testing. The first one is called black box testing. This type of testing was not used because I am both developer and a tester, therefore I know the source code and how the code works. Black box testing is only used if the tester and developer are different people, so the tester does not know how the code behaves and how classes and methods are functioning. Another type of testing is Smoke Testing. This was not used in our modules so far. This type of test is used to check the basic functionality of the software/ program. This is useful to evaluate if the software developed so far is functioning and if further complex tests can be started.

## **Uses and Implications of Techniques**

Software testing techniques that were discussed are Unit testing, static testing, white box and black box testing, and Smoke testing. Unit testing is very useful when trying to test every possible requirement and every possibility of the branches of the code. This can be explained by our milestones where there are multiple requirements for the variables. For example, when ID needs to be unique, cannot be null, cannot have more than 10 characters, and cannot be changed once assigned. After knowing all of these characteristics we can start developing unit tests using JUnit. Static testing is used in most projects because it is as simple as reviewing the code by the developers and finding typos or syntax errors. White box testing is a type of testing that is used when a developer and tester are the same person or team. They develop tests while knowing the classes and methods and knowing the source code as well. Black box testing is the opposite of White box testing because the developer and tester are different people or teams. The last testing, I talked about is Smoke testing. It is practical because it is fast and simple to create. It helps the testers evaluate if the code is stable and if the basic functionality is met. After this more complex or more expensive tests can be started. This makes sure there is no simple error that would slow down the testing and development of the application.

## **Caution**

During the entirety of this project, I learned a lot. Previously, I did not have any experience with testing applications or software. I would say that my mindset improved knowing how important it is to create clear, sound, and effective code and how a simple mistake or lack of caution can lead to a catastrophe. During this class, I was taking a software security class and these combined helped me understand how important it is to make sure the code is strong and tested. I have to admit before this project I was relentless when writing my code, just trying to make it work. Now I know that was a very bad path to walk on.

## **Bias**

I believe bias can play a big role in developing and testing your code. Knowing from my own experience there were many times I got frustrated or angry when hours of developing the code resulted in errors or the tests that I developed did not pass. You feel bad for yourself because you spent all this time trying to make it work and you failed. Then you feel that you do not want to start all over again and you try to make the code work no matter what. This is the bias that got over you. We can also call it emotions. I believe that the best outcome would be having two different people develop and test the software or application. This eliminates the emotional attachment of the developer and the tester can objectively look at the requirements and the code without making any irrational or biased decisions. In my case, I like to think that the code is not mine when I am testing it. I fully emerge myself into the testing part while forgetting that I am also the developer of that particular code.

## **Discipline**

As we learned from this week’s discussion, there are many instances when the corners were cut or things were overlooked and disasters happened. It is important to make sure you develop the code with the industry's best practices. It is even more important to test the code with caution, discipline, and the industry's best practices. Being disciplined is a very important skill that both testers and developers should possess. Making sure that every aspect of the code is checked helps prevent any errors or potential threats to the software or the users.

# **References**

1. Novosad, D. (2023). Week 4 Journal (David Novosad 4-2 Journal Unit Testing Approach and Writing JUnit Tests)
2. Novosad, D. (2023). Week 5 Journal (David Novosad 5-2 Journal Reflection)