Jessie Smith

SNHU

CS 320

22 February 2024

Project Two

**Summary**

My approach aligned with the software requirements because I tested to ensure that each part of the program could be updated, or not updated, according to the given requirements. For example, in the Appointment.java file there is a description variable, which is a string that cannot be longer than 50 characters, cannot be null, and cannot be empty. In my program, when an Appointment object is created, the objects variables are created by one of the constructors, depending on the information given, and if there is not a description initially given, the description is filled with “EMPTY” instead of being empty, or null. I also ran a setDescriptionTest() test, which ensures that if the setDescription() function is called, the object description matches the description given by the user. It also ensures that an IllegalArgumentException is thrown if the description is set to null, or if the description is longer than 50 characters. I did this test for all the setter functions in the project. The only variable that wasn’t tested this way was the ID variables in each part of the project, but that is because they are not supposed to be updatable, and I did not create any functions or methods to update the ID variables. The ID variables are created using UUID.randomUUID() when an object is created and cannot be updated after it is created. I also tested all of the update functions withing AppointmentService.java and the other parts of the project to ensure that when updated, the variable is equal to the user given string, and that the variables cannot be empty, null, or longer than the characters limits.

To ensure that my code was technically sound, I used the Assert() functions from JUnit, as a way to assert that the variables either equal user specified strings or throw IllegalArgumentExceptions. For example, in the AppointmentTest.java part of the project for the setDateTest() test, which is where I tested the setDate() function, I first used assertEquals(appt.getAppointmentDate(), dateTest), and then assertThrows(IllegalArgumentException,class) () - > appt.setDate(null) to ensure that all requirements were met and being tested in the JUnit tests, I also used assertThrows() to ensure that a date could not be set to a date before present. To ensure that my code was efficient, I made sure to test that each constructor was creating objects with the given information. For example, in newAppointmentTest(), which adds each appointment to a list of appointments, I made sure that each variable was created by the constructor. I also created tests for the other constructors, and ensured that when information was passed into the constructor, the variables were equal to the given information. Here is an example of where this was done in AppointmentServiceTest.java:

@Test

**void** newAppointmentTest3()

{

AppointmentService appt = **new** AppointmentService();

appt.newAppointment(dateTest, descTest);

*assertAll*("newAppointment test3",

() -> *assertNotNull*(appt.getAppointments().get(0).getAppointmentId()),

() -> *assertEquals*(appt.getAppointments().get(0).getAppointmentDate(), dateTest),

() -> *assertEquals*(appt.getAppointments().get(0).getDescription(), descTest)

);

}

**Reflection**

The testing technique that I mainly used was black box testing. I tested that the program ran as specified in the requirements, that certain input would provide certain output, and that variables were updated, or that exceptions were thrown based on input. Since there were specific requirements, like object variables not being null, not being empty, not being updatable, or having a specific character limit, I tested that each of these requirements were followed in the program. A testing technique that I did not use is static testing, where testing is performed without running code, which can be done using requirement documentation, user stories, etc. I feel that static testing is more appropriate for larger programs with more requirements, and this project was not a very large program, and I could keep up with each variable and how the program should run without having documents in front of me to remind me the requirements. I felt that black box testing would be better for my project because of this, I knew how the program should react, and the requirements were specific and to the point.

While working on this project I adopted a mindset that allowed me to learn more through research, as this was the first course where I had to use JUnit to test my code. Because of this I employed a lot of caution, I made sure to test each part of the code, since it was my first time. It was important to appreciate the complexity and interrelationships of the code because there were multiple files working together, and each variable had specific requirements that was not consistent with each package. For example, Appointment.java has a date variable and the others do not, while Contact.java had firstName and lastName variables that the others do not. I couldn’t simply copy and paste the code, and then edit the names, I had to ensure that each variable was following the given requirements.

I definitely think that bias would be a concern when testing my own code, I would feel confident in my work and possibly skip over errors or malfunctions. To eliminate bias in my code, I made sure to test everything in the code, even if I thought I did not need to. Each variable was tested to ensure that it met requirements and each function was tested as well. For example, each variable in each package was tested to ensure that they could not be null, could not be empty, and could not be over the character limit, even if I wrote the code the exact same way in each file.

It is important to be disciplined in your commitment to quality as a software engineer because cutting corners can result in errors that could be dangerous to users. Security was not a part of this project, but an example could be cutting corners when testing the security of a program, a data leak could cause many problems for a user, the client, and even myself. To avoid technical debt as a software engineer, I plan to follow requirements, request information when unsure about requirements, provide suggestions that could help the client whether it be security or anything, follow agile techniques, follow OOP, and test my software without cutting corners. Using best practices will help me be successful in my field, protect user data, etc.