Kyler Wiles

10/19/2025

CS 320

Project 2

1. Summary

When I was doing my unit testing, I made sure to touch on every single software requirement. For example, for most of the requirements I am supposed to make sure that certain data does not exceed a character count, dates are correct and cannot be null. Here is the code I used to test that these errors were being checked. @Test

//id too long

**void** testFailIdLong() {

Appointment two = **new** Appointment("13213213132132132121",**new** Date(),"124");

}

@Test

//id null

**void** testFailIdNull(){

Appointment three = **new** Appointment(**null**,**new** Date(),"123");

@Test

//date null

**void** testFailDateNull(){

Appointment four = **new** Appointment("123",**null**,"123");

}

@Test

//date in past

**void** testFailDatePast() {

Appointment five = **new** Appointment("123",**new** Date(-600),"123");

In both examples you can see I am testing for null data and the other requirements that were stated. This testing applies specifically to the testing requirements and adheres to them clearly. If I were to only test one of these scenarios then how would I know that the other requirements were working correctly as well. This was a lot of coding and checking but my screen shots clearly show that every requirement is being considered in the testing process. I know that these tests were effective because my feedback showed failures for every intended failure and success for every intended success. This is a very good strategy because independently looking for every failure should have a 100% success rate. I know my code is technically sound and efficient based off the screenshots I included above. I made sure to use simple tests, making sure that the only errors that could occur are the errors I am looking for. Simple code leads to less issues which allows me to find my intended issues easier without external issues giving me false negatives. I wanted to include another example here from one of my service files because I have not talked about them yet. @Test

**void** testAdd() {

AppointmentService add = **new** AppointmentService();

Appointment adding = **new** Appointment("123",**new** Date(),"123");

add.addAppt(adding);

}

//testing that remove works

@Test

**void** testRemove() {

AppointmentService remove = **new** AppointmentService();

Appointment removing = **new** Appointment("123",**new** Date(),"123");

remove.addAppt(removing);

remove.removeAppt(removing);

Like my other tests I made sure to test only for what could fail. These tests are simple because I am simply testing if the add and remove functions work. I know that they are working correctly because my testing feedback gave successes for both tests. For my remove function I made sure to add an appointment before removing it because you cannot remove what does not exist. I also made sure to keep this code efficient and technically sound by using simple commands that leave little possibilities for an unrelated error. I think it is important to only include what you want to test in your tests.

1. Reflection

I used Junit testing which allowed be to see errors in the code based on the “input” that was given. For example, for my null testing I purposely added null in the input to allow me to see if the system will flag an error based on null. My code does this by throwing an error whenever null is detected. This helps me test the code quickly and makes sure that the logic being employed is working as expected. Testing one value at a time was where I found the most success with Junit testing. By isolating failures, I know for a fact what failed and where it happened. Junit testing is a form of white box testing because I am testing based on the actual code. I had to make sure I knew how everything inside the code worked so that my tests could be accurate.

Other testing techniques that were not used are black box, manual and performance testing. Manual testing entails running the code and looking directly at the output to see if everything is working correctly. For Junit testing you are technically only looking at things from a pass or fail perspective. For most of my classes so far, I have been doing manual testing which works but is slower than Junit. Performance testing works by seeing how the program works from a performance perspective. Since this did not apply to the project, I did not do this because that would be a waste of resources and time. Black box testing is where you test the software based purely on input and output. An example of this would be loading up the software and giving it some input to see if the output is expected or not. This would include looking only at the output instead of the code itself.

Manual testing is great if you are trying to get the code to work and your project is on the smaller side. It is not a big deal to do manual testing if there is not much code to look at because it will not be a time waste. Black box testing would be good if your project is close to completion and you are expecting things to work correctly. Performance testing is great when you are expecting many users for your software. If it cannot run well with a lot of users, then your testing will show this.

I employed caution by considering every possible failure inside the code. It is important to do this because making sure every aspect of the code is working will lead to a fully working project. In my code example shown on page 1 you can see that I tested for every failure mentioned because I was asked to. This will make sure I adhere to the requirements. Your clients’ needs are important, and you want them to be happy with your work.

I limited biases inside my code by testing things that I knew would work anyways. For example, in my code shown on the first page I test null for id and appointment even though I know it will work for both. To clarify, if it worked for one of them it should work for all of them because they were made with the same format. However, what if I made a mistake? People make mistakes all the time and this is another reason to not have any biases as a programmer.

It is important to not cut corners while writing code because that will lead to confusion and errors. For example, I talked about how my code on page 1 has tests that I knew would work but I did them anyways. But imagine if someone else looked at my code and did not know why I only tested one aspect. They would be confused with my testing process. Also, what if I made a mistake and did not test for it because I did not have discipline. I plan to avoid technical dept by taking my time and testing every aspect of my code. This will account for human error and will allow others to understand my code on a deeper level.