Adam Sissoko

12/11/2022

CS 320

**Project Two**

In order to avoid missing any of the requirements for each of the three features, I started my testing process by creating tests for every class in the project. As a result, I eventually wound up with ten test classes, eight corresponding to the model classes, and two corresponding to the service classes. Each of the program’s features used all of the tests, with the exclusion of the Employee test class. Furthermore, I made a series of test suites to bundle all of the tests into three groups. One with the service tests, another with model tests into another group, and a third with both service and model tests.

Testing is crucial to building a successful project, so I made sure to be very thorough during my testing process. I’d rather spend too much time on the testing process than too little. In order to stay organized, I made an effort to use similar naming schemes for all of my tests. This helped me make sure that each test that I ran corresponded to the correct class. It also made it very easy to make sure that I didn’t miss any features, or include them more than once.

With my testing approach, I focused on one feature requirement at a time. I started with the Doctor Tests, since they only relied on a few tests. The Medical Record feature was somewhat more complex since it included the allergy class, so I focused on the bigger test. When I first made the tests, I forgot to check the Doctors for Duplicate IDs. After checking the list of requirements, I remembered to include this feature, and added the necessary code.

At first, I tried to set up my program so that the tests would automatically pass without encountering any errors, but eventually I realized that the more bugs I found, the sooner I would be able to eliminate any issues with my program. For a while, I was having major issues with the Medical Record test. I just couldn’t figure out how to get the patient history. Eventually, I was able to edit my code for the test, and get the right result.

I had the same logical process for all of my tests. I started with the @before function for code that had to run before each test. Then, I used @test for each test within the classes. In my Doctor Service test, for example, the @before section of the code added two doctors before tests ran in order to get doctors and their IDs. All of my tests follow this logical flow in order to help me make sure that I have input to use for the tests instead of empty classes that haven’t received any data yet. In the patient tests, I added patient data @Before the tests ran in order to verify the data. This made the testing process easier and more effective.

Using JUnit tests is a form of dynamic testing, since it’s used to detect bugs and errors within code that’s already been created. This testing takes place while the code is being executed in order to check that the code is functional. Dynamic testing usually takes place after static testing, but it can happen at any stage of the Software Development Lifecycle. It’s used to fix bugs that static testing may have missed, so it takes place after compilation.

Along with Dynamic Testing, I used White Box Testing, since I used the JUNIT tests to complete logical and structural testing internally. These tests took a long time, so they’d probably be better suited for projects that are not under any major time constraints. White box testing could be effective for program updates that involve adding new features to a system, like new options on a phone. Since there’s no real time limit for releasing software updates on a phone, developers can take their time testing, and release a working product.

Static Testing is when you examine code to detect errors and issues. Additionally, it involves design and requirement documentation. Static testing takes place in the earlier stages of development, in order to find errors before execution.

Black Box tests are functionality tests designed to see how the system performs. For these tests, you don’t need to know anything about the internal structure and design, so the tests are usually completed by dedicated software testers. Black Box Testing can be automated very easily, and it is usually quite fast. Therefore, this type of testing works well for projects that come with tight deadlines. This could include software patches or bug fixes designed to address security risks.

I employed caution by making an effort to avoid spending too much time on any single part of the code, and making sure to look at the whole project as a whole. I split the model classes and service classes into two separate groups to make it easier for me to check each group for bugs. I ran my code very frequently, and followed industry standards for code design and readability.

While I was working on this project as a tester, I had to make an effort to stay focused on finding issues with the code. It’s common for developers to just focus on building a working program, without making any effort to identify bugs. Due to the code’s complex nature, and the service classes using the model classes, I had to make sure I didn’t just worry about the code functioning. I had to change my mindset and choose a more granular approach to make sure that each part of the program worked properly.

Building these tests was often quite complex and overwhelming, especially when I was struggling with the Medical Record Service Requirement. Without staying disciplined and making an effort to overcome the challenges I encountered, the program would have been released with unidentified bugs and defects, potentially rendering it unusable. Staying committed and making sure to stay focused makes it easy to avoid issues while programming, and it will be very helpful throughout my future career.