Trevor Mayo

CS 320

12/18/2022

**Summary**

Unit testing all of the features throughout the project were all very similar. The software requirements that were given for all of the features all had very similar attributes like character length and nullability. For example, in all of the classes that weren’t service’s, the id’s all had to be under a character length of 10 and couldn’t be null. The id’s also couldn’t be changed once added initially. For the rest of the classes, some attributes couldn’t have lengths over 50 characters, while other classes have attributes that can’t have lengths over 20 characters. To create my unit tests, I approached it the same way I read and interpreted the requirements. If an attribute couldn’t have a length over 20 characters and it couldn’t be null, then I tested the attribute by creating the class with null as that attribute. I also tested the attribute by creating the class with a length over 20 characters to ensure that an exception was thrown. This method of testing was used throughout the whole process of creating the JUNIT test cases.

By using the coverage percentage, I could assure that my test cases were covering at least 80% of the total things to be tested within the application. By ensuring that the application has such a high coverage percentage, I can assume that the application is doing well in terms of requirements.

When writing the JUNIT test cases, I ended up copying a lot of code that I wrote for a test case and replicated it by changing variable names and arguments. When writing the test cases, I would have multiple properties that could only have a certain length of characters, so I copied the first test case that I knew worked. After, I would paste the test case and change the number of characters that would throw an error for that property and then changed the property name. By copying my own test cases, I could ensure that they all followed the same logic and worked similarly to each other. In a sense, the code was more technically sound by the consistency. I also ensured that the original test cases were done correctly by emulating the tutorial of JUNIT test cases that were similar to mine.

A good example of a valid Contact class would be the following:

Contact contact = new Contact("lasdgo3", "Billy", "Smith", "8882087348", "24 Main St New York City");

Then after, changing certain properties to null created exceptions that I knew were technically sound based on the original Contact.

Ex:

New Contact("lasdgo3", null, "Smith", "8882087348", "24 Main St New York City");

**Reflection**

Most of the tests that I incorporated were specification-based tests. This means that I used the specifications or requirements to base my tests on. I tested a valid input, then tested invalid inputs to see if they threw the exceptions that I hoped for. In a sense, I was testing for failure in all of my tests after finding a valid class creation.

I didn’t use any black box testing within the project. Black box testing refers to a strategy “where the design of the software system being tested is unknown to the tester.” All of my tests referred to the requirements listed in the rubric of the project, which means that I knew exactly what I was testing and how to test it. Black box testing is very helpful for QA teams that have an unbiased opinion of the codebase, which can lead to interesting test cases that one might not think about as a programmer who wrote the code base. I also didn’t use any automated testing techniques. These automated testing techniques could include frameworks such as Selenium or Cucumber. Using automated testing techniques help when checking general package vulnerabilities within the codebase, along with helping the time constraints that some testers might incur. Having created this project from scratch, I also didn’t use any regression testing either. There was no need to regression test a version one of a code base. Regression testing helps when you’re trying to ensure that the code base hasn’t come up with new errors after updating certain aspects of the system. This can be a quick way to ensure a bug free application after large or even small applications.

I employed quite a bit of caution when creating and testing the code. I wanted the tests to be efficient along with having the correct errors being thrown. It’s very important to double check your code base when emulating requirements through code. I probably quadruple checked my error checks when creating the initial classes to ensure that they followed the application requirements effectively. I also used caution when creating the unit test cases as well, meaning that I was very wary of using the correct variable names along with using the correct test cases for those variables. For example, the following two lines of code look similar, but they pertain to two different properties of the class.

new Contact("lasdgo3", null, "Smith", "8882087348", "24 Main St New York City");

new Contact("lasdgo3", "Billy", null, "8882087348", "24 Main St New York City");

Both of these lines of code pertain to two different “non-nullable” properties of the class, but they look very similar. Having caution ensures that I’m making sure that I’m testing the correct property and naming the test cases accordingly.

I’m sure that bias could be a problem in a general sense when writing code. However, I feel like I didn’t have too much bias when writing my test cases. I went through the test cases very methodically and I feel confident in the outcome. Bias and confidence can become construed, however, they’re entirely different. Bias is more of a problem when you’re confident without caution. If you’re confident with caution, it shouldn’t be a problem. Limiting bias all comes down to being extremely cautious and taking the correct steps to ensure quality code.

In the end, quality code is the most important quality you can have as a programmer. Quality code creation creates the pathway and base for a scalable and cooperative project. Not cutting corners ensures that we’re not missing test cases that can be the make or break for an application of scale. It can become difficult if programmers are missing test cases, which leads to coworkers having to go back and find issues that could have been prevented in a tenth of the time. To avoid technical debt within the computer science field, I’m going to follow general programming practices that adhere to code quality standards.

Citations

The QA Lead, & Boog, J. (2022, December 10). *9 types of software testing in software engineering*. The QA Lead. Retrieved December 18, 2022, from https://theqalead.com/test-management/types-of-software-testing/