Cristian Chavez

12/9/22

CS 320

Fulfilling the software requirements that were laid out was a priority so once the initial code meant to meet those requirements was completed, it was tested as thoroughly as possible to ensure that the requirements were met. The requirements laid out explicitly what objects would be created and what attributes they would need to have. Since the requirements were explicit, then tests would be relatively straightforward. One of the requirements for the contactClass was that the contactId could not be null, or be longer than 10 characters. Two tests would be created for the contactId, where they would each fail if the input was null, or a string longer than 10 characters. Since the expectation was that they would fail, and then they did, it meant that these inputs would not be processed. The JUnit tests that were written for the ContactService, TaskService, and Appointment Service were able to effectively test the code written to fulfill the software requirements. The coverage on the JUnit tests for each class always above 80%, which gives me confidence that the code is being thoroughly tested.

I ensured that the code written was technically sound as well as efficient by grouping together appropriate statements, ensuring that the correct input was accepted, and ensuring that a detailed error message would display the test was failed. In my ContactService, the contact class has this code:

if (contactId == null || contactId.length()>10) {

throw new IllegalArgumentException("Invalid Contact ID");

}

Instead of writing two statements, where one checks for null, and the other for length, one statement was written with “or” to reduce the number of lines. When this IF statement is taken, the message will appear, and the user will have an idea of what produced the error.

There’s a variety of testing techniques that can be used to ensure the code written is secure and properly fulfils the requirements. White box testing was used throughout to ensure that each component is working correctly. Tests were written to make sure there was great statement coverage, where we verified that each statement worked as intended, and that it was being used. I refer to the previous example where a statement that says it doesn’t allow a variable to be null or larger than 10, we tested this by seeing if the program would react as expected if it was assigned null or more than 10 characters.

A testing technique that wasn’t utilized was Black Box testing. There wasn’t much Black box/behavior testing since there wasn’t a menu that the users would navigate through. This form of testing doesn’t concern itself with the internals of the program, it just makes sure that the system is responding as expected when given the appropriate inputs. If there was a menu, I would most likely do this testing manually but verifying that the system flowed properly and there weren’t any issues.

If a developer was creating a program that required user input, or the user to navigate a menu, there would be a mix of black and white box testing, since the developer would want to make sure the user is able to properly move through the program, and that the program works as intended when given the appropriate inputs.

When acting as a software tester, I will confess that there wasn’t too much focus on caution. If I stretch caution to mean that I ensured that the code written, and the tests made were clear on what they were testing and what the variables belonged to, then caution was given. By having a consistent naming convention, there was less room for the tests written to accidentally test the wrong variable. And by giving the tests clear names that stated what they were testing for, it would be easy for others to look at the tests written and know what they’re checking for.

Thoroughly testing my own code if it was more complex wouldn’t be easy. There would definitely be a bias where I would not test something because I assume it works, or I would not write a test because I know the program would not pass it. I limited this bias by writing tests for everything the program did, despite sometimes not being sure if the program actually would pass. Writing a test for everything proved to be the best thing to do in this situation, since I realized that some tests were failing that should’ve passed, and they were failing because of simple mistakes.

There is lots of trust given to systems that we use in our everyday lives, systems that we just expect to work perfectly and keep us safe. If these systems weren’t perfect, there’s a risk of failed business relationships, lost income, or loss of life. Stoplights for example, we expect them to work perfectly and feel secure when we drive on a green light. If the stoplights began to give green lights improperly, a scenario where a collision would occur becomes possible. Creating bug free code should be a developers mission, however testers also shoulder that burden, because a faulty program could lead to disastrous consequences. For this reason, testing shouldn’t be an afterthought, and should be actively done through the development process. This will reduce technical debt and lead to a more securely built program.

Citations:

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