**CS 320 – Project Two**

Arguably the most important part of testing software is making sure that the tests align with the requirements. There is no way to ensure that the test code tests all of the requirements besides making a direct comparison. Since direct comparison is the main way of determining whether every requirement is implemented, the tester should make sure the layout of the test code matches the layout of the specification document requirements. For example, In the Contact service application, the first requirement listed is that the ID variable can’t be null, it can’t be longer than 10 characters, and can’t be updateable. So the first test in my test code was to make sure the ID can’t be null, my second was to make sure it can’t be longer than 10 characters, and the third was to make sure it can’t be updateable. After I completed the tests for the first requirement, I would move on to the next. Repeating this order until I had reached the end of all of the requirements. Then, because I did the tests in the same order as the document, I could go back to the top and walk through it step by step, ensuring no requirements were missed.

Another approach to ensure the quality of testing is by checking the coverage percentage. I am confident there are no bugs in this mobile application because I made sure it had a hundred percent coverage for all of the program code. If all of the code is tested and all of the tests are valid tests, you can guarantee there will be no bugs. This is important, in addition to testing for each requirement, because there will almost always be additional features or code the programmer added that aren’t listed in the requirements. Having a hundred percent coverage makes sure that no matter what the programmer added to make the program meet the specifications, we know that there are no bugs in it. It is different from testing for requirements because it isn’t checking the code for what it is supposed to do, but rather, testing for whatever code is there.

In my experience writing the JUnit tests for this project, the best way to ensure that the code was technically sound, was by using one effective technique. This technique consisted of making sure the Unit test was working properly, before testing it on the actual program code. For example, in the contact class, one of the requirements is that the ID member variable can’t be null. So instead of writing the test, and then writing the entire constructor, hoping that the test was coded correctly. I would instead write the test, and then make a simple temporary constructor that would confirm whether the test was working correctly. I did this by putting a simple if statement inside the constructor that threw an invalid argument error if the ID argument was equal to null. This strategy works well because the program code is so simple I know that it is correct, so I can see if the test is failing when it should, effectively testing the test.

I ensured my code was efficient by finding the difference in the time between when a unit test began and when it ended. I also used the “finished after” Eclipse feature, which tells you how long it took a JUnit test to complete, which is built into Eclipse. This gave me a pretty good idea if a class was working efficiently. If you know the whole test script is completed in 0.06 seconds, you know it likely works efficiently enough, especially if there are a lot of tests in it. This was the case with most of my test scripts, they were mostly in the 0.1 seconds or less range. But there was one that took a longer 0.46 seconds. So I had to use the previously mentioned technique to find which part was taking longer and determine if the longer time was necessary or if it needed to be fixed.

Looking at the testing techniques I used, I think they could be categorized as three fairly broad techniques: 1. Ensuring the rejection of invalid arguments 2. Ensuring the accurate retrieval of retrieval functions and 3. Ensuring the proper functionality of a function given an argument. The most common technique I used had to have been Ensuring the rejection of invalid arguments. This test technique works by using an assertThrow function, which fails the test if it doesn’t receive an error of the specified type. For example, in the Appointment class test script, I made a test to make sure it wasn’t possible to make a new Appointment with a null date. I made an assertThrows function, specified to look for an invalid argument error, and then tried to make a new Appointment with a null date. If the test failed that means that an invalid argument was never received.

A more obscure technique that wasn’t mentioned previously, that I had a harder time with initially, is what I call a negative test technique. This test technique involves making sure that a certain functionality doesn’t exist. For example, the Contact Class, the Appointment Class, and the Task Class all had the same requirement, their ID wasn’t allowed to be updated once created. But, how do you test for the nonexistence of a function? I decided that the best thing to do was to make sure that the function exists, just in a nonfunctional way. I made the test so that it would fail if an error message like “ID can’t be updated” wasn’t received from that function. Then I would make another test to make sure that the ID indeed couldn’t be changed by that function. This makes the coder have to write a broken function, which should ensure they are on the same page.

All three parts of this project had a similar form, In that they each had a data class and a data management class, so the variety of testing techniques was somewhat limited. However, the application of these techniques apply to many different types of programs. Valid argument tests could apply to any program that has functions, which is most if not all programs. The correct retrieval testing techniques would be useful with any programs that utilize objects with setter and getter functions. Testing a method's functionality is similar to testing its arguments in that it could apply to any program with functions. The implications of these testing techniques are; functions exist and receive arguments, functions do a specific thing, and getter functions should return a specific variable.

If I’m being completely honest, I didn’t approach this project with as high of caution as I could have. It was hard for me to shift my mindset from, this is an assignment I need to get done that will never be used, to this will be used by people who could be hurt by it, or worse. But, I did intentionally make sure every unit test checked for one thing, and I did my best to make sure it was done accurately. This is particularly important because the complexity and interrelationships of the program can make it confusing to know what has been tested for and what hasn’t. It is easy to believe that things like test layout, making sure a test checks for one thing, and clear test names are less important than the test's implementation, but this is not true. They are equally important to the implementation because if the test programmer can’t understand what has been tested for and what hasn’t, there could be significant mistakes in

their tests.

I can see how bias can be a potential problem, especially when the developer is also the tester. In some cases in this project, I wrote the code first and then wrote the test after, which I suspect would be an approach that is at risk for a lot of biases. If I am ever part of a small development team where I am both the developer and tester, I might make all of the tests first and then only after they are all complete, work on the program code. The more time between the better, because this gives me another chance to catch bias’ when I have to reacquaint myself with the requirements. It also would keep me from tailoring the tests to the program code. Another useful approach might be talking to the product owner about any vague requirements. The more precise the language on the requirement document the better, when it comes to reducing bias.

Testers need to be disciplined because they are, in most cases, the last step of proofing software goes through before it is released to the public. If they don’t catch a bug, that bug could render the software obsolete, hurt people, or damage the company's image. It might be easy to assume that developing software is a less serious field compared to something like Engineering. Or at least that was what I thought when I first started my Computer Science career. But, these assumptions are wrong, hackers exploiting an advantage of a bug in software might be less tangible than a collapsing bridge, but it’s just as real. People's money, personal information, and even sometimes their health or life, have been affected by defective software. I would argue, assuming people will depend more and more on technology, that the potential severity of these offenses will only increase.

SOURCES

* Parthiban, P. (2021). *7 Reasons Why Software Testing is Important.* Indium Software. <https://www.indiumsoftware.com/blog/why-software-testing/>
* Neeru360. (2023). *Software Testing Techniques*. Geeks for Geeks. <https://www.geeksforgeeks.org/software-testing-techniques/>
* Garfinkel, S. (2005). *History’s Worst Software Bugs.* Wired. <https://www.wired.com/2005/11/historys-worst-software-bugs/>