### Southern New Hampshire University

Journal: Software Testing Techniques

Nicholas Markel

### Dr. Albanie T. Bolton

CS-320-T3325 Software Test Automation& QA 23EW3

2/14/2023

During my development of project 1, I used the requirements as an outline to complete my project. When a customer pays a developer to make a program, the customer should receive a product that does what was agreed upon. Thus, using the requirements for an outline ensures that the developer completes all aspects of the program that was requested. This is shown in my project with the if statements that I use to throw the illegal arguments in the main java files. The requirements state that the string must be less than 10 characters and cannot be null. This is easily added by making an if statement that will throw an exception if the string is less than 10 characters or null.

When running my Junit testing, I used the run as coverage feature often. This allowed me to make changes and ensure adequate coverage as I went instead of making it an afterthought where I had to respond to the lack of coverage. Initially, I did not know how to use the Junit testing. However, through much practice and some research, I became very confident in using it and trusting the results it gave me. Eventually, I utilized Junit’s suggestions that would recommend more cases to test for better coverage. This took a lot of the guesswork out of the process, and it even wrote a lot of the code I needed for me. I worked until my coverage was 80%, the amount recommended by our textbook(Boni, 284). This was a good suggestion for coverage as it ensured that much of the input was covered.

Ensuring that my code was secured was a breeze with Junit. The first roadblock I came across was trying to ensure that the tests acted how they were supposed to. After I found out the proper way for a test to work, I was able to reuse the same exceptions and tests on multiple other lines. For instance, when project 1 asked me to make sure that the input was only able to accept 10 characters or less, I wrote a test that would be displayed if someone were to try and enter more than 10 characters. Then, I had to handle that expression in the test file. Here I wrote a function that would detect the input error. From there, it would flag in Junit showing that it worked correctly.

To ensure that my code ran efficiently, I kept the amount of code to a minimum. I went through a couple of different iterations on how to write the code that would test for inputs outside the recommended limits. The easiest way was to write a different statement for each requirement. However, this made my codebase very large and affected my performance. I then slimmed my code down by using the and operator and putting the different requirements all into one if statement. This checked for a wrong condition one time inside each if statement instead of trying all the different scenarios multiple times for multiple if statements.

The software testing techniques we used in this project were exclusively Junit. It’s a testing package that is included in Eclipse and native to Java. There were many other techniques we learned. However, Junit was already installed in Eclipse and did a very good job of ensuring that our test covered a good portion of our code. We learned about other techniques such as TestNG that would have been more work to install. All other testing tools likely would have done what we needed them to. Junit was just used thanks to the ease of use.

Testing our code this semester really got me in the mindset of what it would look like to ship a full product working as a software developer. It was nice opening my eyes to the fact that testing should be done at all stations in the program’s life cycle instead of as an afterthought. From now on, I will be using Junit to test any Java that I write to ensure that my inputs are handled and any exceptions are covered. This will make sure that my code is more robust and ensure that I create secure programs.

When developing this program, I employed caution in writing all my test cases. While testing the program is needed. I feel it shouldn’t take away or introduce any new bugs into requirements laid out by the customer. Therefore, I made sure to write enough test cases to fulfill the requirements but not too much to burden the program or cause the code to break. One should not take away from the core of a program to ensure that all test cases are covered. However, one also should not disregard testing altogether. Finding a happy middle where someone employs enough tests and fulfills all the requirements is what is needed of a well-rounded developer.

Bias was a hard thing to steer away from during the development of these tests. My main bias during this was that the end-user should know not to do this. However, as a software developer, I greedily assume that people just know their way around something. Assumptions should not be made, and tests should be installed to ensure that there is no guesswork and no added possibility of errors. To overcome the biases that I had when developing this, I used self-discipline. I put in the extra work that I did not see the need for to ensure the extra layers of security. Specifically, when asked to ensure that a string was not over 30 characters. I took the time to add 30 words to my string which is an easy task when hardcoding. However, this ensured that the aspect of that requirement was covered and, in the end, made my code more secure.

References:

Boni Garcia. (2017). Mastering Software Testing with JUnit 5 : A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language. Packt Publishing.