**Project Two Summary and Reflections Report**

Carter Wooton

SNHU CS320

Dec. 11, 2022

**Summary**

For each of the three features, the main purpose of the JUnit tests is to verify that the requirements for each class are met. After deciding what methods would be required in each class of the feature, I began writing the code, considering the requirements; for each requirement, I would write the method and the JUnit test side-by-side to make it easier for them to work together properly. As examples, the tests for the object classes were focused on ensuring proper input validation, while the tests for the service classes were focused on ensuring that those methods properly added, deleted, or updated an object.

To be sure that the JUnit tests are effective, not only do they have to pass, but they also must pass for the right reasons. To ensure that your tests are representative of your source code, the coverage should be relatively complete, such as above 80%. Although that number is arbitrary, and the coverage percentage is not a representation of the quality of your tests, it is nevertheless important that you allow the program to execute most of the code that is written. This will ensure that your code works as expected with good impot and will work as expected when handling bad input.

To ensure that the code was technically sound, the tests had to prove that the add/update/delete methods in the Service classes responded correctly to illegal arguments or otherwise unexpected input. For the add method, this was ensuring that the method that tested if the provided id is already in the list threw an exception. For the update method, I checked that the object’s attributes are not updated if illegal arguments are given for its attributes or if the id is not found. For the delete method, I checked that it deleted nothing from the list if the ids did not match. This ensures that the code isn’t crashing when unexpected input is provided. Text

Description automatically generated

JUnit tests need to be completed in the shortest time possible. Each test shouldn’t take longer than a second to execute. In this coding example, the tests are very simple and take very little time to execute. In fact, on my computer, the entire test class takes less than 1ms to execute and complete, which I can view in the JUnit window after running the test class.

Text

Description automatically generated

**Reflection**

Perhaps the most relevant unit testing technique for these assignments is the boundary testing. Boundary testing is useful with input validation, where you test a method using values that are just within the legal range and values just outside the legal range to see if the program responds properly. This kind of test is used instead of checking every single legal value, which would be cumbersome and would most likely be a waste of time. I use this testing technique to test that the class constructor and its setter methods are properly throwing exceptions when illegal data is used, such as a null value for id or a description that is too long. The next testing method I used was the coverage method. I used this exclusively for the class that contained the methods for adding, deleting, or updating objects in the list. It was easy to verify that the methods for adding or removing objects worked, as I just tested to see that the list increased or decreased in size; accordingly, but these methods also had code for when they did not execute as expected, such as when the searched id does not exist in the list. For this, I designed tests that would cause the additional code to execute to be sure that it ran properly. Code coverage is not absolute, and it doesn’t usually make sense to have 100% coverage, but it is also important that all the important code is executed in the tests to ensure that it is also working properly.

There are other testing methods that I did not use in these assignments. Regression testing is a form of testing that takes place after a finished product is changed. The tests are designed to check that the system is running as it was before the change and make sure that bugs are not introduced into the system with the new updates or that existing bugs are found and solved. Another testing technique that is not expressed in these assignments is benchmark testing. Benchmark testing checks the speed and efficiency of the program. There are third-party tools to assist with these tests, but they essentially involve measuring the amount of time it takes for certain code to execute under certain conditions. This is helpful in programs that require high performance, or to catch when a program takes abnormally long to execute code, which would be a hard-to-find bug unless caught in testing. There is also extensive security testing that can be implemented. Security testing can involve static testing or dynamic testing that checks random inputs into the system with the goal of triggering faulty program states and is useful in simulating attacks from the outside in.

Each of these testing methods can and should be used when deploying software, especially web-software that must remain up-to-date and secure. Many non-enterprise level companies may be able to focus on third-party libraries for many testing features, and automated testing exists that can check that the code is at least minimally acceptable; however, it is always important that extra steps are taken to ensure that the software is as clean as possible and as secure as possible before it reaches consumers.

When developing these features, I took extra caution to create a loose coupling between the methods and classes. These features aren’t complex, so this wasn’t a difficult job, but is still relevant. When testing a method using the JUnit tests, it is important to only test one feature at a time. For example, the delete method in each Service class requires at least one object to be created and added to a list; instead of including the add method within the delete method test, I made sure to just manually add an object to the list. Preparing the test this way avoids the possibility that the delete method test will fail due to a bad add method. If this happens, I could be led down the wrong path when trying to solve the error.

As the developer, I like to believe that my code is perfect. Unfortunately, it never is. When you review your own code, you have probably already read it plenty of times. It is always harder to spot errors in your own code than it is for someone else: same goes for proofreading a paper. While it’s okay to create your own unit tests while you code to check that your methods are working properly, it is also important that someone else reviews your code to check that it meets standards. It’s difficult to account for your own bias when writing code, but I made sure to follow best practices with naming conventions and comments, such as documentation comments that would help someone understand the methods if they were doing a review.

Software development is a very sensitive task. Any error in the design or the code can lead to unexpected behavior at best or total failure at worst. As with bias, when you test your own code, you are at risk of cutting corners and being lazy, because you may feel that your code is flawless. When developing software, you will likely spend more time debugging and testing your code than you will writing it; for young developers, this may seem frustrating and unnecessary, and they may rush the process, leaving lackluster tests. You may be saving time in the short term, but when your product ships with multiple bugs and errors, you will eventually spend more time fixing the software after release, or worse it may never be released. This technical debt will add up and will threaten your budget and your SDLC. One way that I was able to reduce technical debt was by going for the highest possible test coverage to ensure most of the code was executed. I also was sure to give two hours of testing and debugging for every hour of coding.