**Project Two: Software Testing Strategies**

Tyler Walker

Southern New Hampshire University

CS320: Software Testing Automation & QA

Instructor Omar Toledo Lopez

April 16, 2023

My approach to developing the JUnit tests throughout the course was directly aligned with the software requirements. First, I used the software requirements as a guide to develop the classes. This involved applying object-oriented programming (OOP) principles to the class structure and implementing each requirement as its own method/attribute. For example, each package (Contact, Task, Appointment) had its own service class that managed each respective object. Each service class was required to fulfill certain functionalities such as add/remove task, add/remove contact, and add/remove appointment. Therefore, I made a method for each of those in the service class and used the appropriate coding to allow the service method to add an object to the object lists. In doing this, I was also able to look back at the requirements as a reference for unit tests that could “assert” whether the methods acted correctly.

An important aspect of ensuring technically sound code was to utilize object-oriented programming (OOP) principles, however that was not everything. Object-oriented programming helped me to organize the code and increased the likelihood of full test coverage because it made it very easy to separate every function into its own module and understand the purpose of every line. Also, when it became necessary to edit/remove/add tests, OOP made it very simple because I did not have to worry about if I was affecting my testing coverage; Each test was separate from all the others. For example, if I made a change to my unit test that was intended for ensuring the Id parameter was not null, this would have absolutely no effect on any other test, and I could not harm test coverage by making changes to it. Another important aspect of keeping the code technically sound was that I used redundancy in my constructors and setters to ensure the classes could not instantiate an object with invalid data (Too much redundancy can hurt efficiency, but for this application the effects would be negligible). Finally, I ensured the coding was technically sound by creating a completely new object for each test that was meant to cater specifically to that test. For example, when developing a test that checks whether the service class will instantiate an object with a null description, I created an object in which all arguments were valid except the description, which was null. I did this for every test. To ensure efficiency in the code, I used minimal variables and simplified the code as much as possible. This helped to make the code run efficiently. In terms of ensuring efficiency in writing the code, I did this by first writing each requirement into the class as a comment and focusing on one at a time. Then, I did the same for the tests. By writing all the requirements each as their own comment, it made it easier to stay focused without having to switch between tabs often to ensure I didn’t forget something.

For this project, I implemented white-box, functional unit testing. The inner workings of the code were utilized as a roadmap to making the code, and therefore can be considered white-box testing. Furthermore, the tests were intended to ensure the software fulfilled the requirements by functioning properly, and therefore can be considered functional testing. Finally, the tests are considered unit tests because each test checks the functionality of a very small “unit”. In this case, a unit is considered the smallest bit of code that can perform an action. Some software testing techniques I did not use for this project are black-box testing, non-functional testing, integration testing, acceptance testing, performance testing, and security testing among others. My tests do not qualify for these for many reasons. Firstly, security testing is great for limiting unauthorized access to system resources, but this testing does not do that because it does not check whether these methods are secure. Also, performance testing is important because it helps to ensure the software functions properly under various loads and system conditions. This testing doesn’t include performance testing because it is still early in development, and it is necessary to first provide functioning code that could potentially be integrated. Once that is finished, it can be determined how well the various bits of code perform when integrated as a system. Finally, this testing is not considered acceptance testing, because that is generally performed near the end of development to verify all the other tests have provided full “coverage”.

During this project I learned a lot about the need for caution when developing tests that were specifically designed to test the functionality of small units of code. I learned to have the mindset that tests should always be independent and should not rely on the results of any previous tests. This is why I decided to create specifically catered objects for each test. It is also a good idea to “clean up” after the test by getting rid of the information, to ensure it cannot affect another unit of code; essentially acting as if the test didn’t occur. I also learned the importance of considering every scenario that can occur in the development of these tests, and that testing should occur early and often. For example, I realized quickly that the method for testing one function can be used to test others, but not always. It is okay to use the same logic for testing multiple units, but copying and pasting isn’t the greatest idea because every test is almost always different. Copying and pasting tests is a great way to forget to change something in the code, and integrate a defect into the system. Furthermore, these kinds of defects are hard to find because the syntax may not cause any errors; it is simply a defect in the logic that could have been corrected by not cutting corners.

Bias is another important consideration, because when developing code, it can be easy to assume I know every possible outcome, and that a very small unit I have written is so simple there is nothing that could go wrong. Therefore, bias can be limited by taking the extra time to write tests even why I think they are unnecessary. After all, we are writing these tests to get the highest coverage percentage possible. While writing more tests cannot increase this percentage, the more avenues of error that we consider, the more defects we can keep from occurring. This idea of due diligence and not becoming complacent also comes into play with commitment to quality. I think it is very important to have a growth mindset, and to understand that because I will never stop learning, there can never be a “pointless” or “unnecessary” test. To avoid technical debt, first and foremost I think it is important to communicate effectively with the customer to ensure we aren’t developing more than is necessary, but that we are considering how they will grow and expand in the future. Following proper coding standards is a great way to ensure the code is reliable and can be maintained in the future. More generally, I will mitigate technical debt by acting honestly and ethically, while collaborating effectively with team members with quality as the core of the conversation.