Ezra Huffman

Prof. Wilson

CS-320

February 26th, 2023

**Summary and Reflections Report**

**Summary:**

The unit tests used in each milestone were written to reflect the requirements given. For example testing the getId() method of a the Appointment class involved creating an Appointment with a known id and then ensuring that that getId() returned the same id. In the ApplicationService milestone errors were also checked to ensure that attempting to create appointments that did not meet the requirements given for each parameter failed with a useful error message. Unit tests were written for both under and overflow for each parameter that referenced a minimum and maximum length. Every parameter that had restrictions was also tested to ensure that a ‘happy path’ and ‘sad path’ (failure and success) worked as intended.

I ensured that my code was technically sound by using unit tests to test the functionality of specific methods and classes. I wrote unit tests to ensure that the requirements given were met. Specifically to test the ApplicationService I wrote tests to ensure that the correct error messages were thrown when an appointment was initialized with an invalid set of parameters.

I did not do any stress testing to test how efficient my code was, as I am of the opinion that preemptive optimization is not a good idea. I tried not to do anything too bad in regards to performance from an algorithmic perspective, but performance often fluctuates based on the input size, hardware, and many other factors that are not worth speculating about without knowing the specific context in which the application will be used. I could probably optimize how items are found by ID if look-up times are a problem and memory is not an issue. Currently, the program iterates through a list until it finds a matching ID where it could also use a hash-table, associative array, or dictionary to have constant lookup times.

**Reflection:**

* **Techniques:**

For each milestone in this class, I used unit tests to test the submitted software. This meant righting tests that either fail or succeed; testing the requirements set in the assignment’s rubric. I used JUnit as the assignments were written in Java. I attempted to have ample coverage of the submitted code, meaning testing methods and classes that were explicitly mentioned in the assignment’s requirements. Although I don’t think of it as a formal method of testing I also visually reviewed the code myself to attempt to catch any bugs that I might have unintentionally introduced while trying to meet the requirements.

I did not do any integration or end-to-end testing during the milestones. This is mainly because the assignments were not interconnected pieces of software, and there was not anything to test that they integrated with. I also did not do any end-to-end testing because there was not a cohesive product at the end, but rather functional components without much use in themselves.

Using unit tests to test software is one of the most prevalent and useful ways to ensure that code does what it was intended to do. Unit tests are great for testing individual functions or components that have a known output for a given input. Often the most difficult part of writing unit tests is writing edge cases that cover a large portion of the software's possible behaviors. Unit tests can be used in most situations and the benefits of having them usually outweigh the time spent writing the tests. Integration testing is really useful if the project you are working on is part of a larger whole. Integration testing means ensuring that what you submit for your part doesn’t have unintended consequences for the rest of the whole. For smaller projects, that are not part of a larger whole, integration testing may not be worth the time spent implementing them or in some cases, it may not even be possible. End-to-end testing is when a product is tested from the “beginning” to the “end” going throw the entire flow if possible. This is just another way of ensuring that changing one piece of code doesn’t break another part of the project. Sometimes software doesn’t really have a clear linear flow and therefore, end-to-end testing might not be useful. End-to-end testing can also become very time consuming so it is important to weigh the cost against the benefits.

* Mindset

Working as a software tester I employed caution in trying not to make assumptions. I have fallen into the trap of thinking I knew what code did and not adequately testing it for that reason. During these milestones, I tried my best to write the unit tests without making assumptions about how the rest of the code worked. It was important to consider the complexity of the code I was testing because there are things that could be easily missed or overlooked without taking time to consider how the parts worked as a whole.

I tried to limit my bias by not jumping to conclusions about how things would work beforehand, taking writing the tests as a separate exercise from writing the applications. It is very easy to think you know exactly what your code will do before testing it and just go through the motions of writing unit tests, missing important edge cases. I have fallen into this trap in the past, and I tried to be conscious of it while writing the tests for the milestones.

It is important to commit to quality as a software engineer for a few reasons. First, you don’t want to cause damage to a company or people by accidentally crashing a service, or creating any number of bugs that might have a range of undesirable consequences. Depending on the software being written, errors may not be that big of a deal, only resulting in a project never being delivered and time wasted; however, others can have dramatic consequences. There is also the concern that companies generally don’t want to hire programmers that are going to submit sub-par work that is full of bugs, costing them time and money as they incur more and more technical debt. Lastly, if one thinks what they are doing is important then they should put effort into doing it right.