Christopher Hardy

CS320 Software Test and Automation

Project 2

My approach for each of the software requirements was good, but not perfect. I tried to take care in making sure each test, tested for exactly what the software requirements asked for. For Milestone 2 I failed on checking for the description length. I copied my length check code from the name or ID check and failed to update the length requirement. After feedback I updated this to reflect the requirements accurately.

With my JUnit tests I made sure they had more than 80% coverage. I did strive for 100% coverage but some of them were only at the threshold. The better I got with these JUnit tests, the better I became about writing code that would play nice with tests. At first I was adding extra code to my classes to ensure they would pass the tests that I wrote. I knew this was bad practice, but I was learning from it. On my last JUnit test, I had the AppontmentServiceTest call the AppontmentTest class. This meant that I only had to run “one” test to test all of my classes and they passed with 100% coverage.

As mentioned previously, I made my tests efficient by having them call another test so that I could have multiple tests completed at once. I have started to get better about reducing my amount of code through classes as well as JUnit tests. My last milestone was significantly cleaner and more efficient than previous Milestones. With my JUnit tests I made sure to initialize the class, initialize a single object if possible. And then use that object to complete all methods and class required for coverage and software requirements tests.

For the first Milestone I had to reteach myself a lot of Java again. Over the years and different courses I have been focused on finishing assignments for grade sake rather than to learn. I also needed to learn how to use Eclipse finally and that was a good push. For the first Milestone I created switch statements and tested most of my code through user input. After seeing that my code was sound, I tore it apart to make it more accessible for JUnit tests. This was a challenge at first having to learn what felt like a whole other language and having so many issues with classes not being able to interact with others. Going forward from that point I was able to understand how all these techniques worked together. In my Appointment MIlestone I even had the AppointmentService JUnit test call the Appointment JUnit class. This meant that I could just run one JUnit test and it tested both of my Appointment and AppointmentService classes.

As I am completing this milestone and testing on my own there are a few testing techniques that I did not use. Although, I did do static testing while writing the code and doing dynamic testing through JUnit tests. However, I could have solved other issues with techniques such as peer reviews, decision tables, or any type of formal review.

Static testing is incredibly important early on in each part of the project. This helps identify errors before they can cause issues that would only appear after running the program. Dynamic testing is important because we can use it to catch errors that are predictable or unpredictable. With dynamic testing we can see how software reacts with other software or hardware. This should only be done after thorough static testing and reviews such as peer reviews, technical reviews, or inspections.

As I progress I am keeping testing in mind as I write my software. This means I do not need to create lengthy menus or GUI’s to do any manual testing. If I can create software that has all the requirements and nothing more, then I can use JUnit tests to call all the things I would have done with manual testing. This reduces the time typing and thinking of items to try. Automation certainly has provided an easier way to test.

I did try to limit my own bias in later tests. In my earlier tests I was adjusting my own software and making it untidy as I was just trying to get it to work with the JUnit tests. This is certainly a concern as the author of your own software and as the tester of your own software. We can overlook many issues and errors and think it is technically sound just because it meets the base requirements. Software should meet more than the base requirements but not too much as to bloat it.

Keeping all this in mind it is important to stay professional and remember that this isn’t just your project. It will likely be for a consumer and needs to be as sound as possible. We shouldn’t write some janky software and claim it as good just because it does what the customer wants. Issues like this could introduce software vulnerabilities or defects that could be very costly to fix. More upfront testing and peer reviews are great assets to mitigating the cost of defects later down the line. Testing procedures have been tested and laid out for us to use and we should continue to employ and update these procedures as needed to ensure quality work is done.