Andrei Kourouchin

CS-320-H7099

Project Two

During project one, I implemented three features; an appointment service; a task service; and contact service. These three features were created as back-end services for a mobile application tasked by my supervisor. I applied unit testing to all three features after building them, however, the unit testing was slightly different for each of the features. As an example, for the appointment service, I created tests to test whether we can add new appointments, or delete them, and then as the appointment object should have been unique, I also added testing to ensure that the appointments were unable to be created with the same appointment ID. This test coverage gave me coverage of 90.1%. As the goal for the project was 80%, and the code that was not tested was for error catching if you attempt to delete a nonexistent appointment, this meets client specifications. Similarly, with the task service, I had coverage of 91.4%, with the main missing coverage being the for loop that returns if the appointment already exists when adding one. This could be fixed with just a few lines, however as is, meets client specifications. The final service I constructed was for the contact service, and this is the service that had the highest coverage percentage at 92.3%. Similarly, to the previous service, the only code that was not tested was the for loop that performed correctly in another test. However, as this was not part of the client's specifications, I did not implement a test for it.

To ensure my code was technically sound, I first laid out exactly what I wanted to test on paper, figured out how I would do that with my code, and then proceeded to implement it with JUnit testing. When I went about creating the tests, I attempted to make my code deliberate and do the minimum it needed to do to ensure the tests were done correctly. I removed all superfluous code and ensured I was using the methods in the java classes I had previously created for the services, rather than creating new ones in the JUnit tests.

There were a few software testing techniques I employed in this project, one of which was static analysis as I was writing the code to ensure I was meeting requirements. Walking through the code line by line helped a lot to ensure my code was well written as well as did not have any extraneous steps. If I could not explain why the line of code was there, it was removed or altered to ensure it made sense. After which, I implemented JUnit testing and ensured that I had good coverage and tested all sorts of different things to ensure the needs were met.

There were a few other techniques available that I did not use in this situation, such as peer review. Peer review will have one of two ways of testing code, one where you provide the code base to another person, or you sit with another person and go through step by step and figure out if there are any issues or missing requirements in your code. There is also Boundary Value Analysis (BVA) which is a testing methodology that tries to find the outer boundaries and test them, as outliers are typically not handled well unless the software has been built very well. When working on a very large set of data, manual code review becomes very labor-intensive, and as such is worth building automated tests. When building smaller modules for existing software, there is nothing wrong with doing peer code review, but if you’re going through thousands of lines, sitting with a peer, and reviewing the code is just not practical. When doing BVA, depending on the type of software you are building, you might not be able to find boundaries in a way that allows you to test them effectively.

When working on this project, I employed caution by ensuring my test coverage was high. I did not want to be underinclusive of code and potentially have bugs in it, so I went for having as high a test coverage as possible to cover all test cases. This ensured that my requirements were fulfilled and that the code was executed without any issues. If you do not appreciate the complexity and interrelationships in the code you are testing, you might end up writing tests that do not execute correctly or give the wrong results. As an example, if you use a function that adds an appointment object to your appointment list, but the appointment class is improperly set up, even though your test case is running correctly, you might get a bad result.

When reviewing my code, I attempted to limit bias by going about it with what I felt were fresh eyes. Reviewing code and testing were done on a separate day from coding so I don’t have the mindset of programming the code, and I attempted to do it as matter-of-factly as possible. On the software developer side, there is for sure concern of bias if I am testing my code as I understand why I did it the way I did it. However, this might not be the best way of doing it. Someone with fresh eyes might look at my code and say this was a horrible way to implement this and suggest something completely different but is much more secure and faster.

Being disciplined in my commitment to quality as a software engineer is incredibly important. When you cut corners and make mistakes, you will affect the rest of the software development with that code base. You must build a solid foundation, and continually write good code to ensure that when you’re implementing things a year from now, when you’ve forgotten all the code, you know exactly what you need to do, and exactly how to do it. Continually learning and reading and writing code will help avoid technical debt. However, you will always be missing something, and that is the reason why peer review and other testing methodologies will help immensely. When you figure out, you’re missing something, you can go back and learn and then fix what you’ve done.