Chiara Aziz

Project 2

Project one gave me good experience with unit testing. Unit testing was the main approach when testing all three sections of code. In testing the code I aimed to hit all the requirements. I set out the requirements and went through them one by one and made sure I had tackled that test. For example, in testing the contact class one of the requirements was that the first name field could not be longer than 10 characters and could not be null. To ensure this requirement was covered I wrote a test on line 64 of the contact test class that would throw an error message if the name was null, blank, or was over 10 characters. The test case passed in a null value, a blank value and a value that was greater than 10 digits. I proceeded to do the same thing for the rest of the requirements and the test classes.

I felt fairly confident in the quality of my test coverage. Most of my code was covered with overall coverage being at least the required 80%. I also made sure to check the line coverage and branching coverage to ensure that most of the code was touched and that branching logic was covered. Making sure all these areas are covered leads to tests that are better quality and ensure your code is properly tested.

To make sure my code was technically sound I tried to follow good practices. I made sure that my unit tests were only testing a small unit of code and not multiple parts. I worked on naming the tests appropriately with what was being tested, for example of line 81 of the contract test class I have a test named settingLastName and on line 92 of the same file I have a test name settingAddress. I find that it is much more useful when you test to write tests with clear names instead of test 1 or test 2. When you get a large number of tests, naming tests in numerical order becomes confusing. When running a whole test suite, when you look at the out put there is no fast way to see what the failed tests were meant to test. Ensuring that the code was efficient was accomplished by keeping the scope narrow and to the point in the tests. I didn’t write a lot of extra lines to test things, I tried to only write tests that tested small parts of the code. I used parameterized testing so I was able to run a test several times with different parameters. On line 57 of the task test class there is a parameterized test called settingTaskName. This test passes in a null task name value, a blank task name value and a task name that is greater than 10 characters and an appropriate error message that should be thrown if any of these cases are true. The test is able to test all three outcomes with few lines of code.

Since this code base had an in memory database only and not an external one the only testing technique that was used was unit testing. Unit testing worked well for this code base to ensure that setting the names of tasks or contacts followed the requirements. Unit testing also showed that the code was actually creating a contact, task or appointment. Ensuring that your code does what you think it does or what you expect it to do is an essential part of the testing process.

Integration testing was a test technique that wasn’t used in this project. The text notes that integration testing is once all the units have been written they are put together to create a larger system. Integration testing exposes problems in the interactions between components (Hambling, 2019, p.62). Systems testing wasn’t used on this project but systems testing deals with testing a product end to end and focuses on the behavior of the whole system (Hambling, 2019, p.64). End user testing also was not used in this project, but end user testing allows the user to test and interact with the system and software to ensure if functions as expected. Integration testing is useful in the testing cycle as it helps make sure that everything is talking to everything properly, for example, that the code can talk to a database. Systems testing can tell developers if the whole system functions as a whole and works. Acceptance or end user testing ensures that the product does what users want and need. So when developing it can provide a great feedback loop on what features users find useful or irritating. Changes can then be made to adjust functionality to bester suit users.

While writing tests for this project I used more caution than if I were writing code. I used caution because I was more aware of what I needed to test. For example the requirements for a task name were specific. So I took extra time making sure that the tests that I wrote actually covered the requirements specific to the task class. I tired to limit bias while writing the tests by making sure that I wrote tests that targeted those requirements. I tried to write tests that failed first and then I would work them to pass. I found this helped me from writing tests that would pass regardless of the inputs. It helped me catch errors in my testing and write more targeted tests.

Testing software can take time, but it is a very important step in the software development process. It is important to start small with unit tests and not just jump into large overall tests right away. Starting small with unit tests allows you to test small sections of code so you know things are working properly or code returns what you think it should. Moving down the line to integration, systems and acceptance testing build upon unit testing to make sure that the code base is fully tested. Small errors in code can cause large problems in a production environment and millions of dollars in lost revenue for companies. Not properly testing or cutting corners might get a product to market faster but the problems or customer complaints cause more damage in the end.

Avoiding technical debt all together is hard but making sure you are following good coding or testing practices can help to alleviate tech debt. Writing solid tests and enough tests can help catch bugs earlier in the deployment process, which helps in reducing the amount of tech debt later. When I notice tech debt, making note of that and letting the appropriate parties know is important so that the tech debt can be prioritized and fixed. While fixing tech debt doesn’t produce a new feature, it is an important issue to tackle in code bases. Tackling tech debt makes for a better product and application in the long run.

References

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing : An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.