Jayd Cheshire

Project 2

4/22/22

**Summary**

My testing approach was very aligned to the software requirements. Specific lines of code include lines 92-183 in the TaskServiceTest.java file. In the requirements to the TaskService.java file it is outlined that user should be able to: add a task with a unique ID, delete tasks with task ID, and update the name as well as description fields. With lines 91 – 110 I cover adding a task with a unique ID. With lines 131-137 I cover deleting a task with task ID. Finally, with Lines 117 -130 cover updating name and description fields. With the TaskServiceTest.java I have a covered 95% of the TaskService.java file. And with the ContactServiceTest.java I have covered 93% of the ContactService.java file. Based on Coverage percentage this is considered very good because the cutoff for acceptability is about 80% coverage. One important thing I did, among other things, to ensure my code was technically sound happened between lines 171 – 183 of the TaskService.java file. In these lines I test to ensure that my code will not introduce another Task with the same Task ID. Efficiency is defined as speed and reliability of code. With respect to this definition the reliability of code aspect will relate to technically sound code. Lines 171-183 accomplish show this reliability. Although I did not directly test for speed in my test cases, the speed is indirectly tested through the Junit tests. Each one of my unit tests in all four of the respective test classes take less than 1 second to execute.

**Reflection**

I used static testing which is the testing that occurs without running the code. In other words, static testing is basically desk checking. This desk checking that I did took place in the form of reviews where the goal of the review is to find errors such as dead code, variables that are never used, inconsistent interfaces, etc. I wrote out the code on a piece of paper and went through it by hand looking for errors. Static testing is key to an efficient and successful project because of the cost escalation model. The earlier an error is found the cheaper it is to fix that error, though in my case the “cheaper” aspect for me is time. The further along an error went unnoticed in the project the longer it would take me to fix. Dynamic testing was another approach that I took. Dynamic testing includes methods like black box testing, white box testing, use case testing, decision table testing, etc. White box testing was one of the ways I tested my code. Ways I employed white box testing was to try to have each statement in the code executed once. I tested the different structures that were within my program such as the iterative structure, the sequential structure, and the selective structure. I accomplished this through various Junit tests such as testing for duplicate tasks. I also incorporated black box testing by orientating a chunk of the tests around the specifications. This included tests such as testing the length on a piece of data or ensuring that it was not null. Types of testing that I did not do for this project include but are not limited to decision table testing and state transition testing. Dynamic testing, while more expensive to undertake because of its place in the development cycle, can help us to find errors we never would have known were there had we only done static testing. Everything I discussed above has practical uses. This project was an emulation of a real-world project that was just simplified. All the same strategies apply and scale. Dynamic testing, while more expensive to undertake because of its place in the development cycle, can help us to find errors we never would have known were there had we only done static testing.

**Mindset**

In acting as a software tester, I employed caution by testing my code with above 80% coverage. I also wrote test cases using multiple testing techniques to ensure that my tests were thorough. While I did not necessarily have a lot of tests, the tests that I did do tested key points in the code that should be representative of a larger whole. It is important to appreciate the complexity and interrelationships of the code I was testing. Without this appreciation white box testing would have been so much less effective. For instance, had I not had an appreciation for the complexity with which I store the data I would not have been able to effectively test things like duplicate tasks. The ways I tried to limit the bias in the review of my code was in the way I tested. Using black box techniques to test against the specifications I was able to abstract away that bias that I had as the developer. The specifications included things like making sure certain bits of data were not null. I tested for these sorts of things under the umbrella of black box testing. Being disciplined in your commitment to quality as a software engineering professional is key when it comes to writing or testing code. Cutting corners when it comes to writing or testing code is only creating the illusion that you are saving time. In reality you will probably have to go back later and spend more time than you originally would have to fix the problem. This then leads to the idea of technical debt. Technical debt is the idea of cutting corners in the interest of saving time knowing that you will have to go back and embellish or rework certain things. I plan to avoid technical debt by being disciplined when it comes to writing or testing my code. This statement then answers the question of the importance of being disciplined more explicitly. The importance of being disciplined is to help avoid technical debt.

References

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) - 3.3.1 Review Process.* BCS The Chartered Institute for IT. Retrieved from   
https://app.knovel.com/hotlink/pdf/id:kt00UC2IF2/software-testing-an-istqb/review-process