**Summary and Reflections Report**

**Summary (a)**

* To what extent was your testing approach aligned to the software requirements?

My testing approach aligned with the software requirements because I treated each requirement as a separate test case when coding. For example, the requirements for the “taskID” field were that it could not exceed 10 characters and it could not be null. I tested a non-null, less than 10 characters string when assigning taskID in my tests, but I also tested a null taskID and a taskID exceeding 10 characters in length. In this way I was able to ensure that my program was throwing an error as expected when it received bad input.

* Defend the overall quality of your JUnit tests for the contact service and task service.

Because of coverage testing, I am certain that all areas of the code have been reached and I am confident that all branches of the code were tested. This is because, when using coverage testing, JUnit give you visual feedback as to what parts of the code have and have not been tested by the given JUnit tests. I know that my tests were effective because I have built off prompt #1 (above) and ensured that I am creating my tests to adhere to the requirements provided, while also testing invalid input for completeness.

**Summary (b)**

* How did you ensure that your code was technically sound?

To ensure that my code was technically sound I used a @BeforeEach tag and an init() function to reinitialize my variables before every unit test (TaskTest.java:12). In TaskServiceTest.java, I did not use this approach, but in the future I should. In TaskServiceTest.java, I ensured technically sound code by copying and pasting my test variables between unit tests instead of manually typing the values which could introduce an error. I recreated these unit tests in the Appointment and Contact class files as well.

* How did you ensure that your code was efficient?

To ensure that my code was efficient, I tested small chunks of code at a time. In TaskTest.java line 44, I test a single constructor. In other test, I used multiple assert statements, but I tried to make sure that each unit test was only focused on one part of the functionality of the code. I have a unit test for each constructor in Task.java, and a separate test for testing bad arguments to the constructor. No JUnit test should last longer than 1 second, and I have adhered to this rule.

**Reflection – Testing Techniques**

* What were the software testing techniques that you employed in this project?

The main type of testing used was White Box Testing, used to carry out Unit Tests. White Box Testing is a method of software testing in which the tester has intimate knowledge of the code that is being tested, i.e., they wrote it. The most basic form of White Box Testing is the Unit Test, in which many different functions are written solely for the purpose of testing the functionality of the code. The unit test’s goal is to provide complete coverage into all parts of the code (different branches, constructors, etc.) and to ensure that the expected results are achieved (García, 2017).

* What are the other software testing techniques that you did not use for this project?

Other functional testing techniques that I did not use are:

Integration Testing – Individual units are integrated into a single project and are tested as a whole.

System Testing – The software AND hardware are integrated on a machine(s) and the system, as it will be used in production, is tested for functionality, performance, and meeting the specified requirements.

Acceptance Testing – Testing that happens near the end of the development lifecycle. This type of testing basically answers the question: Is the software ready for deployment?

* For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.

Unit Tests – The practical use for unit tests is to make sure each piece or module of software functions as expected. Unit tests are essential to development and also to further stages of testing. Unit tests aid in finding and fixing bugs early in the development cycle, as opposed to finding errors in other stages of testing where it may not be clear exactly where the error is coming from. With unit tests it is always clear where an error is.

Integration Testing – This type of testing is necessary for integrating units into one functional piece of software. Each individual unit is tested in the stage prior (Unit Testing), so in Integration Testing, we are testing the interactions between units. Since we know that the units have passed Unit Tests, we can focus on how the units interact, and we can be sure that any error that arise in this stage are due to the interactions between the units and not the units themselves.

System Testing – System Testing is useful for testing a project in the production environment in which it will be running. For example, if we develop a payroll application for use by a specific business, it would be very beneficial to test this software on the actual hardware (computers) that the business will be using in the real world. System testing ensures that the software developed works as desired while running on the same physical hardware it will run on when it is fully deployed.

Acceptance Testing – This is the final stage of testing, usually done near the end of development. This is practical for developers that wish to ensure that their software meets every single requirement specified by their client. It is done to be sure that the software is “accepted” by the client. This is obviously beneficial for any development team, as client satisfaction is the end goal of all developers.

**Reflection – Mindset**

* Assess the mindset that you adopted working on this project

It is important to adopt a cautious mindset when acting as a software tester. This project may be different than the job of a software tester in the field because I wrote AND tested the software. Usually, testers and programmers are separated. In a sense, developing and testing the software helped me be more cautious when developing because I had the test cases in mind while writing the code. It is important to constantly be thinking about the interrelationships of the code and being aware of how the different pieces fit together. This can help avoid bugs in the future that may be more difficult to detect if not found at this stage. For example, it was helpful for me to think about how the TaskService class would use a Task object when I was writing the Task.java class.

* Assess the ways you tried to limit bias in your review of the code.

Bias in software testing refers to favoring certain test cases over other because you as the programmer can already predict the outcome of the unit test (Shenk, 2022). I do think that bias might be a concern if I was responsible for testing my own code in the field. Ways that I tried to limit my own bias was trying to purposefully “break” my code. Even if I could predict the outcome of a test, I would try many different unit tests in order to get a non-favorable outcome. I think the best way to avoid bias for a developer/tester is to try to look at the software from an outside perspective, as if they were the end user of the software.

* Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional.

It is extremely important to be disciplined when it comes to the quality of the code you write, and the amount of effort you put into it directly reflects on the developer at a personal level. I have personally been in many situations at previous places of employment where corners were constantly cut, and the technical debt incurred was almost insurmountable. Cutting corners and writing “bad” code causes more harm than the time it saves, and it almost cuts to the soul of a person. It is immoral, and I liken it to stealing merchandise from a retail store, or other crimes along those lines. In the end, if a developer can realize that they will be serving not only the customer, but themselves as well, by practicing disciplined software development and testing, they will be a much better developer and person for it.

References

García, B. (2017). Mastering Software Testing with JUnit 5. Packt Publishing.

Shenk, G. (2022, March 9). The Impact of Cognitive Bias on Software Testing. Functionize. Retrieved June 19, 2022, from https://www.functionize.com/blog/the-impact-of-cognitive-bias-on-software-testing