**Summary and Reflections Report**

**Summary:**

1. Testing Approach:
   1. My approach was almost completely aligned with the testing requirements. I didn’t do anything extra past the requirements that were listed in each assignment. I would look at each requirement, make a function for it, and then move on to the next requirement. For example, the TaskService class needed a function to update a task. So, when I got to this requirement, I added an add function to my project.
   2. My JUnit tests were of very good quality. I achieved 85% coverage for my tests, which means I tested for 85% of all possible use cases. Most of my files had 100% coverage, however, some were a little lower which brought my average down. I’m not sure why some had lower coverage when it seemed like I was testing everything. I would need someone more experienced in testing to find out for sure.
2. Experience writing JUnit tests:
   1. To ensure that my code was technically sound, I made sure that variable names were all easy to read and understandable. It can be difficult to read code when names don’t have any correlation to what they are doing. To combat this, all my functions/variables have names that tell you exactly what they are. For example:

public String getTaskId() {

return taskId;

}

public String getTaskName() {

return taskName;

* 1. I did my best to make sure my code stayed efficient by making functions that could be scaled if needed. I also tried to use as few variables as possible to keep memory usage down. Instead of making a new variable I made use of “this” to avoid making new variables. For example:

public void setTaskName(String taskName) {

this.taskName = taskName;

}

public void setTaskDesc(String taskDesc) {

this.taskDesc = taskDesc;

**Reflection:**

1. Testing Techniques:
   1. I believe that the only technique that I used for this project is Unit Testing. Unit testing is when you test individual parts of the software to make sure each part works on its own. I used JUnit tests provided by Eclipse to use unit testing on this project. Finding errors on small segments of code is typically easier than testing large segments of code, however it does take more time to test small segments.
   2. Unit testing is one of many different testing techniques. One technique that comes to mind is security testing. Testers can use security testing to find areas where there is a potential security risk in the code. They try to find areas where attacks could come from and fix them before it could happen. Another technique would be performance testing. Performance testing is used to find out how the program performs under high stress. I didn’t find this approach necessary since this project is so small, so it shouldn’t be under any stress.
   3. Unit testing can be done while you are developing to help limit errors later in the project. It is much easier to find errors when testing small segments rather than the whole project. So, unit testing is a great way to find errors before they become a problem. Performance testing can allow you to accurately gauge how users will use the software and see how it performs under stress. If a product has bad security, it can be a big privacy risk to users, so it’s important that security is thoroughly tested.
2. Mindset:
   1. Since this project was relatively simple, I don’t think I employed much caution during it. However, I only employed enough caution to make sure that there are no bugs or errors in the code. It is important to appreciate the testing process since it can help smooth the whole process, even if it slows you down in the short term. I think time is gained in the long-run since less testing is required after the coding is completed. Since all three services were relatively similar the testing process was the same for each one, where all I really had to do was change a few variable/function names. All three required “add” functions, so all I had to do was copy and paste the function and the test. By developing tests early, you can easily test similar functions later.
   2. It is important to limit bias during your coding. An egotistical programmer may believe that there are no errors since they are great at coding. It is important to look objectively towards your own code to make sure it works as best as it can. For example, a tester could be responsible for testing their own code. They might have an unconscious bias and not test as deeply into their code as they should, or see how the user may interact with the software.
   3. Being disciplined in your commitment to quality is very important as a developer. Customers will not want to use a faulty product, that is full of bugs and errors. Cutting corners in the development/testing process can cause all sorts of problems later. Releasing faulty products can cause a loss of trust between the company and customers, potentially leading to the downfall of a company if they have a bad reputation. To prevent technical debt, it is important to put a high priority on customer requirements and the testing process. I need to make sure that I follow all customer requirements to the letter and make sure that the code works as it should with minimal errors. To make sure that I follow all customer requirements I should ask clarifying questions to make sure I really understand what they are asking for.

**Citations:**

*Why developers shouldn’t do their own QA testing*. number8. (2022, January 13). https://number8.com/104417-2why-developers-should-not-test-their-own-code/