Project 2

Grand Strand Systems

Jessica Megaro

SNHU CS-320

When asked how the testing approach aligns with the requirements, the answer to me is clear. That answer is “entirely”. My testing approach aligned entirely with the requirements because it is the goal to create a product according to the requirements given. A specific example can be seen in TaskService with the function that starts at line 48. I used the predefined requirements specifically to form the testing to ensure that the product is doing what it is intended to do. This can also be seen in Task.java at lines 38 and 49, where the length is set, and an illegal argument is thrown if null is returned.

It was not difficult to ensure that the tests were effective. This code is easy enough to test for the specific conditions which exist. For both contact and task service, each area had a specific set of rules regarding length and whether the item was null. As seen in the code for both services, each item was tested against being null and tested for the length, and all passed. Conversely, all items were set to fail against parameters, and failed when expected to fail. It was not until project 1 that I implemented and paid attention to coverage testing and had an overall coverage of 93%, which well exceeded the requirement.

As seen in the code for task and contact service, the classes and methods were named appropriately according to the rubric and performed the tasks they needed to perform. Proper syntax and annotation were used throughout both services. Code was also structured using hashmap, making it more efficient than reviewing a list repeatedly. Line 49 in task service displays this, which was also fun for me to learn versus the use of a list.

I kept my code efficient by using standards that I have learned so far in my courses. These practices include making sure my code is clean and readable. By creating and calling functions, not having undeclared variables, and eliminating any unnecessary operations, my code is kept clean and efficient. Lines 59-66 in task service display efficient and functioning code. I also used only one assert\* per test as advocated in feedback by the professor, allowing easier debugging. I also implemented industry standard by using camelCase styling conventions and being descriptive.

I employed unit testing on all milestones. Specifically, I wanted to ensure that the testing being used passed appropriately but also failed appropriately. In AppointmentTest I ensured that a 10-character id was valid, and that an 11-character id would throw an illegal argument. In AppointmentTest I also made sure that the description number of characters of 50 was followed and that it was not null. Testing to see that the date was valid became a bit more difficult and had to be reworked. The first time that I employed testing for the date, it failed because the date was captured several lines of code prior to the test being done. When the date was being validated, the date was technically in the past. After receiving feedback from the last milestone, I decided to employ “test driven development” and write my tests before I wrote most of my code. In the previous milestones, I had written most of my code prior to writing my tests for both the Contact and Task milestones. I enjoyed the challenge of looking at it in a different way and changing the process. The benefits of using unit testing and specifically JUnit in these projects are finding bugs and errors almost instantaneously and having a simplified system to examine small chunks of code at a time. I also reviewed my code as the IDE alerted me to any syntactical errors or warnings from the IDE’s linter. This almost works as a peer review as it goes over the general usage of the language and identifies common mistakes (*Software testing techniques 2021*).

Some techniques that I did not use are experience based techniques. I have yet to dive into error-guessing or exploratory testing or using testing techniques that would go over larger areas of code at a time or test a multitude of areas at once (*Software testing techniques 2021*). Performance testing and integration are techniques that I look forward to as well as implementing the AAA approach as an improvement to my practice.

JUnit testing has been fun and challenging to learn. It is practical and covers the principles of testing by ensuring that the tests meet the customer requirements and starting with small parts that can then extend to larger parts when needed. Unit testing can test cases based on the internal structure, which is exactly what our milestones need at this point. Determining the appropriate output for each method and class helped to ensure that the code is efficient and functional throughout each of the milestones (*JUnit tutorial with examples: Setting up, writing, and running Java Unit tests* 2022).

My personal mind set while working is based in caution. Previously, while working in my career as a registered nurse in various trauma centers and emergency departments, critical thinking was one of the major keys of success. Appreciating complexity and not taking words at face value had massive positive effects on my practice as well as outcomes while treating patients in what sometimes was a life-or-death situation. While coding does not seem to be that serious, failures in software can and have cost human lives and major disasters, as well as billions of dollars collectively. In the mobile app that we designed, it is seemingly harmless and not very complex. However, if someone was able to wrongfully access information as the admin, they could get ahold of personal information that can be used to steal identities or commit fraud. As I work through my code, I try to employ my most catastrophic thinking to fully understand everything that can go wrong, as well as if I am doing something in the most effective way or if I can improve upon what I am trying to do. An example of this would be knowing that both an ArrayList and HashMap were appropriate to use for this project, but I chose the HashMap instead since unique keys were used. Understanding the details of each made that choice easier to make.

Biases exist and everyone knows it. There are multiple lenses through which to view any situation or object and perception will always be different. Specifically, strong bias would exist if you were the only one responsible for testing your own code. Naturally, you want items to work, and it might be easy to dismiss smaller problems or issues without effectively resolving them to feel accomplished and complete. However, testing requires much more than a kind eye to be effective. Testing needs to be critical, neutral, and analytical. An example might be knowing that a feature works for its purpose, but not fully testing to failure or seeing if more bugs can be produced. A high-quality QA team would do the full testing, regardless of the deadline or fear of finding new bugs that might delay the project. In addition to fear of feeling incompetent or personally attacked, other types of bias occur frequently. This has prompted large companies such as google to review code anonymously, after uncovering that women at Google faced pushback at a rate of 21% higher than men during code review, Black developers faced pushback at a rate of 54% higher, while Latinos faced 15% higher pushbacks. Also, surprisingly, Asians faced 42% more pushback despite the stereotype of being more capable in engineering fields. These types of excess pushback costs Google more than an extra one thousand hours every day (Tung, 2022).

Dmitri Pavlutin writes “The code is the source of truth for both running and developing the application. The discipline of writing quality, performant, and understandable code is the key to the success of the project.” I could not agree more with his statement (Pavlutin, 2020). Much like verbal communication, the less that is left to interpretation, the better it will be. This translates to code in the way of naming functions and variables appropriately, not leaving anything anonymous, and providing good annotation to industry standard. There is no viable reason to cut corners while coding or testing. A methodical, logical approach with a level head is necessary. Rushing or cutting corners might feel like it saves time, but the errors and fixes that need to occur later will cost more in the long run. At best, all one is doing is “passing the buck” when it comes to hasty or poorly tested code. When one fully understands that their work is valuable and important, and whatever product they produce will be used and affect others, including others on their team that will have to deal with errors or low quality, it is easy to see why quality and strict discipline matters.

At times technical debt might be unavoidable. There are ways to prevent or lessen the debt and the consequences due to them. There are also different types of technical debt. Firstly, knowledge-based debt occurs frequently and highlights the reason collaboration within a team is so important. For example, a team of five engineers is working on a product. Two of those engineers implemented a feature and know everything about it. The other three engineers know nothing about it. This is a knowledge-based debt. This can be resolved easily by actively sharing knowledge during meetings, using good documentation, sharing codebase changes and knowledge about the feature, and being available to answer questions that the others may have. Another type of debt is design debt. This type of debt usually happens in a more competitive environment when teams are trying to be the first one at the finish line to beat out the others. Features are rushed, architecture and structure are neglected, and results in the entire team having multiple fixes later. The only way to lessen this debt is to pay special attention to adhering to design patterns and structuring, while also promptly providing the product. Finally, code debt is writing just plain bad code and not fixing it. Maybe someone is in a rush before going on vacation, they just need to push something through so they can be worry free. Maybe they spent a few days trying to resolve an issue and just can’t. They quickly merge code, forget about testing, and pack their luggage to enjoy their vacation. If everyone did this, the bad code would pile up and snowball quickly. Using automated tools such as linters can prevent many of these errors but reviewing the code with your focused eyes is also a great way to prevent this type of debt. While all three of these debts have different resolutions, there is one important solution that helps to lessen all the debt. That solution is thinking. Setting aside time to think, and critically think, about features and how they are used and how they affect the overall product is one of the most important tasks a team can do. Many organizations even set aside a “think” day to refactor code and address any issues they might have. One thing that is agreed on by all is that technical debt cannot go unchecked and unmanaged. Doing so only leads to ineffective products and unhappy team members (Omeyer, 2022).

**References**

Junit tutorial with examples: *Setting up, writing, and running Java Unit tests.* Parasoft. (2022,

September 29). Retrieved October 2, 2022, from <https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/>

Omeyer, A. (2022, March 16). *Prevent technical debt with these 3 strategies*. Retrieved October

15, 2022, from <https://dev.to/alexomeyer/prevent-technical-debt-following-these-3->

strategies-4nmg

Pavlutin, D. (2020, January 15). *The explicit coding discipline*. Dmitri Pavlutin Blog. Retrieved

October 15, 2022, from https://dmitripavlutin.com/explicit-coding-discipline/

Software testing techniques. GeeksforGeeks. (2021, March 1). Retrieved October 2, 2022, from

https://www.geeksforgeeks.org/software-testing-techniques/

Tung, L. (2022, June 27). *Google looks to reduce pushback bias in developers' Software Code*

*Review*. ZDNET. Retrieved October 15, 2022, from <https://www.zdnet.com/article/google->

looks-to-reduce-pushback-bias-in-developers-software-code-review/