**Summary and Reflections Report**

**Summary**

My testing approach tried to cover as much as possible the stablished requirements. There were several specifications for each class as to how input should be admitted in order to create a Contact, Task or Appointment object (as well as the services that manages each class). For instance, I coded conditional statements that would manage that a string is always passed to the fields of each class. If the input was left blank, then a “NULL” string would be set as a placeholder (which I can assume later on could be used to prompt the user for new input). Additionally, I also trimmed any input that would surpass the desired number of characters, to ensure it satisfied the requirements limit of characters per field. In order to test all of that, I included specific JUnit tests for each possible scenario. For instance, for the TaskServiceTest class I included a test for each situation: adding a new task, deleting a task, updating name of a task, and updating description of a task, which covered all the requirements for the TaskService class. Similarly, I included a test per requirement for all the other classes.

I believe my JUnit tests covered a good percentage of the code, in all classes covering almost 100%. This is due to the detailed testing on possible thought scenarios, verifying that every input that does not meet the criteria is properly modified before being passed to update the proper fields. All testing classes evaluated cases where input was null or too long. Tests were made to both visualize and confirm that blank input was passed with a string “NULL”, or that long strings were reduced to a shorter substring as the maximum number of accepted characters.

To ensure my code was technically sound I made sure that I used comments on my code on each section that would require extra clarification, so anyone else could easily navigate and understand how my code has been created. I also organized the layout of the classes by first stating the fields of each class, then the constructor, and finally all the setters and getters, just like I did with my Contact, Task, and Appointment classes. On my JUnit tests, I used annotations to display the name of each test for a better readability, and also made sure that output verification was included before the assertions, so if any error was found I could easily see it on the console.

To ensure that my code was efficient I attempted to avoid repeating code as much as possible, minimized the use of unnecessary lines of code, and considered how errors would be handled. For instance, because requirements stated that no input should be left null, I included a private helper function “verifyNull”, which would return the same value if the input was not empty, or a string “NULL” if it was indeed non-existent (in both cases some type of non-empty string would be returned). This helper function allowed me to reduce repetitive code for each setter method that needed to evaluate that requirement before passing a value to a field, so I simply called the function in each setter method to simply evaluate the number of the input characters.

**Reflection**

The main software testing technique used on this project was dynamic testing, as it was needed to run already written code to evaluate its behaviors and how it satisfied the requirements. The technique used was unit tests, which is a type of testing where the programmer would choose specific inputs for different cases, to then make a comparison to what would be the anticipated output (Chacon, 2018). To make these tests I used the JUnit framework, which helped easily create short tests to evaluate the behavior of my classes whenever I selected an input that would not follow the easy path and would require error handling. For instance, I made a unit test where the input for the name of a new task is empty, expecting to obtain a task with an actual name, “NULL”, rather than leave it empty, as per the requirements, no task name is supposed to be blank, no matter what.

Other techniques that are typically used are static testing, which consists in doing an evaluation without running the code, sometimes even before code has been written. These techniques focus on what is that may cause an issue, not the issue itself (García, 2017). For these milestones I did not use these techniques, except when reviewing highlighted code before trying to compile it. Normally, static testing would implicate reviews of the specification documents to clarify ambiguities on the requirements or performing other tests such as dependency checks.

The practical use of dynamic testing, is verifying that the code is behaving as it was expected, ensuring that the requirements are being met. For instance, using unit tests, in the class “Appointment”, it is required that the appointment description is no longer than 50 characters; unfavorable input would be a long String over 50, so the class had to present a way to handle that case, which was taking only the first 50 characters and excluding the rest (at least being the prototype solution), which complied with the specifications in the requirements. The practical use of static testing is evaluating what could possibly cause any issues before they appear. Documentation is analyzed to spot any possible conflictive requirement, or a design that may be too ambiguous to work with. The implication of these techniques is to identify sources of errors before the development process, which would then in consequence be less expensive to rectify rather than doing so in later stages of the cycle (García, 2017),

Working on this project I employed caution in regard to how I saw my own code, accepting that it might inherently present errors that I did not notice during the development process. It was very important to consider as much as possible the many situations that would cause some type of error to evaluate how such situations were being handled (if that was the case). For instance, what would happen if the user forgot to put anything when creating a contact, task or appointment? such objects are not supposed to present empty fields, so somehow such situations should have been handled internally in their own classes. With that in mind, testing would then focus on forcing such situations to happen to see the response of the program. This in consequence helped me find little details that I did not considered, which allowed me to make the necessary fixes until I passed all my tests.

Definitely it could be tricky to limit bias when working on my own code and testing it myself. In fact, I was aware of the requirements while developing the code and how undesirable input should be handled to satisfy the testing process. However, I was mainly focused on the requirements rather than how I should be able to pass a certain test. When I started the second part of the project which consisted in testing it, I basically introduced situations with all those undesirable scenarios, even the ones that may be overlooked, such as the input parameters for the phone number of a contact: I could have only tested the need for 10 characters in the input for it to be valid, but another aspect to consider was that such characters could only be digits, so a testing case was required whenever the input was indeed 10 characters, but not all of them were digits. As a tester I should then set aside my own expectations of a perfect code and be inclined to find errors, even if they were not explicitly mentioned on the requirements. Furthermore, it seems to be necessary that a second pair of eyes tests the code as well to avoid bias that may be introduced by a developer testing their own code.

It is important to have an adequate discipline developing software, as it definitely can impact the quality of the product that is being requested by the client. Faulty code cannot only affect the overall functionality and reliability of the product, it can also cause a considerable economic loss, or even cause harm to people (UTOR, 2019). It is crucial to consider that software will not only be released to the client, but to all the possible customers that may use it, which is why quality impacts directly how much it will affect the final users. Cutting corners is never a good idea, after all, errors are way more expensive to be corrected later than earlier. Although it may be a little difficult not to fall in technical debt due to the nature of agile and the focus on speed, one way I can think to prevent it is by implementing automated testing (Kanjilal, 2021), just like I used in this project. Besides having to write the test cases, I could simply run the tests every time I made a change to the main code without having to manually test it every time (although some tests may need to be updated if testing parameters are changed, for instance, if the class now accepts null input without having to create a String).

**References**

Chacon, D. (2018). JUnit Tutorial: Setting Up, Writing, and Running Java Unit Tests. Parasoft. Retrieved from <https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/>

García, B. (2017). Mastering software testing with JUnit 5. Packt Publishing.

Kanjilal, J. (2021). How to manage and reduce technical debt in Agile. TechTarget. Retrieved from https://www.techtarget.com/searchsoftwarequality/tip/How-to-manage-and-reduce-technical-debt-in-Agile

Utor (2019). The Most Expensive Software Bugs In History. Utor. Retrieved from https://u-tor.com/topic/the-most-expensive-software-bugs-in-history