Andrew Eeckman
914834317

Problem Description

The goal of this project is to develop a time tracking application that can be used to keep track of the time spent on one or more tasks. The only requirement the application has of its users is that they give their tasks a name, and optionally, provide it with a description. From there, they can stop, restart, or view a summary of this task or of all of the currently known tasks. This summary provides the user with some key information, namely a given task's name, description, and current runtime. The results of this application must be persistent and exist long after the application has finished its execution. This ensures that no matter when it is run again, the program knows exactly what tasks it was busy handling before.

Given the generality of this time tracking application, there are many cases where it can be effectively used. For example, a student may want to know how long they spend on YouTube versus how long they spend doing homework. Such information could be useful in helping them develop better studying strategies. Or, this application could be used by athletes to record how long they spend doing each individual exercise of their routine and if, in fact, they are rushing through specific parts of it. Essentially, anybody who wants to manage their time better can use this application to help them do so. Yet, its use is not just limited to individuals. Corporations or other organizations could use this application to keep track of their employees and their work habits. For example, in the scenario provided in the project instructions, a developer is using this type of application in order to report the times spent on each individual task they are working on to their manager. This could help the corporation make salary, hiring, and termination decisions. If two employees are consistently given the same tasks and one always does them faster and yet to the same degree of quality, it might be time for that employee to get a raise. Regardless of the use case, this application provides its users with a simple interface to help them better manage their time.

In order to begin using this application, the user must first decide on a name for the task they want to begin tracking. After doing so the user can use the "start" command to initialize the given task. The application will then store its "start time" and save such information in a file known as "task_log.txt". From there, the user can decide to do a multitude of different things. They could either end the task with the "stop" command, add a description to the task with the "describe" command, view a summary of it with the "summary" command, or even create a whole new task. There is no limit of how many tasks can be created by using "start". The application will handle them all separately as different objects. When a task is finally finished, however, the program will save its end time and its duration alongside its name, start time, and optional description in the file mentioned above. Although no longer active, this task will very much still be accessible by the program and capable of being restarted, given another or new description, or having its information viewed with the "summary" command. The above covers most of what was outlined in the given instructions, however, some key requirements are missing

or left intentionally vague in these given instructions. We must therefore consider a few of them before actually writing any code.

The commands outlined in the instructions are given one or two uses and in specific cases only. It is not discussed how these commands should operate outside of these uses and in varying cases. For example, what should “start” do if a task with that same name is still active? In order to come up with answers to these types of questions, let’s try to rationalize these cases in the eyes of the user starting with the command previously mentioned.

A user may type “start” in a variety of cases. If they are starting a new task, this command should simply create a new one. If they are using it with a previously completed task, then this command should restart it, giving it a new start time and adding the new time tracked to the old duration recorded. It is likely in this case, that the user just ended up taking a break and is now ready to return to the given task. If they are attempting to start a command currently in execution, then an error message should be printed and nothing more. The task is already executing and starting a new one with the same name would be a mistake as then the two tasks would be indistinguishable from one another. We can rationalize this as the user simply forgetting that they had already entered in and started such a task.

Moving on to the “stop” command, we see some similar situations occurring. If a task is currently active, then “stop” should simply do as described in the instructions and set both an end time and record a final duration for the given task. If a task is not currently active and “stop” is entered, we can assume similarly to the example given above with “start”, that the user forgot they had already entered such a command in, and print a message to notify them of this error. If a task has been restarted then “stop” once again ends the task normally, this time just adding the new duration to the old one.

As for the “describe” command, there is just one special instance we want to consider, namely when a task that does not exist is first described. We can assume that this is a task that the user is actually working on at the moment. Perhaps they either forgot to start it or they simply just wanted a shortcut. Regardless of what the situation is, if “describe” is called on a task that has not yet been recorded by the application, the program will create a new one with the given name, description, and start time matching the time when the command was called.

The command “summary” has no real special cases. If it is called on a specific task, whether or not that task is active and whether or not it has a description, the output should always be the same. The task’s name and duration should be printed as well as its given description if it has one. If “summary” is called on all tasks in general or, in other words, no task name is given to the command, then all tasks regardless of the attributes mentioned above in the single’s case should be printed.

Noun/Verb Analysis

Starting with the problem statement given in the project’s instructions, we will work to construct a list of potential classes and methods from a set of deduced nouns and verbs. To be

clear, it is likely that the nouns will come to represent potential classes and the verbs will come to represent potential methods within those classes.

"As a **developer**, I want to be able to **track the time that I spend** on a **named task** so that I can **report accurate times to my manager**."

The above statement consists of two verbs and two nouns. One thing not included in the above statement, that is mentioned in the project description however, is the ability to describe a given task. We also know that multiple tasks can be recorded, and so, we will have to keep this in mind as well while conducting our analysis. Going back to the above example, however, we see that the first noun in this phrase is "developer." Now as discussed in the previous section, this application can be used by anyone who wants to be able to manage their time. So, "developer" in this case realistically refers to the user or clients of this program. With that being said and having considered that this is a command-line application, no class is needed for this noun. A "main" function, however, will be necessary and used to interact with this aforementioned client. The second noun in this statement is a "named task." This is something that the program must manage and so we will likely need a class for it. Specifically, we should define a class called "Task" that keeps track of a member variable for its name. In regards to the verb phrases in this statement, we see that the first one is "track the time that I spend." This suggests that some part of the program, whether that be the "Task" class previously mentioned or not, needs to be able to track the duration of the time that a particular task is in use. Finally, "report accurate times to my manager" suggests that some part of the program needs to be able to display the given times already mentioned. From these statements, we can deduce the following noun/verb phrases:

1. The **program** must **track** **a set of tasks**.
2. Each **task** **must have** a **name** and a **start time**.
3. A **task** **can have** an optional **description**.
4. A **task** **that has ended has** an **end time** and a **duration**.
5. A **task** must be **started by the user to exist**.
6. A **user** can **start**, **stop**, **describe**, and **summarize** **tasks**.
7. The **program** must **persistently store any changes** to **individual tasks** and/or to the **set of tasks**.
8. The **program** must be able to **handle invalid input**.

Number 1 above illustrates the need for a separate class that can keep track of the various tasks the user wishes to record. Numbers 2 through 4 describe the potential attributes of a given task and make it clear that a task should be stored as a separate object. Combining the above, we can visualize two classes. One named "Task" that stores the above attributes and another named "TaskHandler" that stores an ArrayList or collection of these given objects. Moving on, number 5 states that the user should be responsible for the creation of the tasks and that a function is needed for this process. Number 6 details a number of methods that the user should be able to

perform on the collection of tasks stored in “TaskHandler.” Number 7 brings about the need for another class capable of storing this collection in a file and making updates to this file when necessary. Finally, with Number 8, we see that it is necessary to vet incoming commands from the user with potentially yet another class.

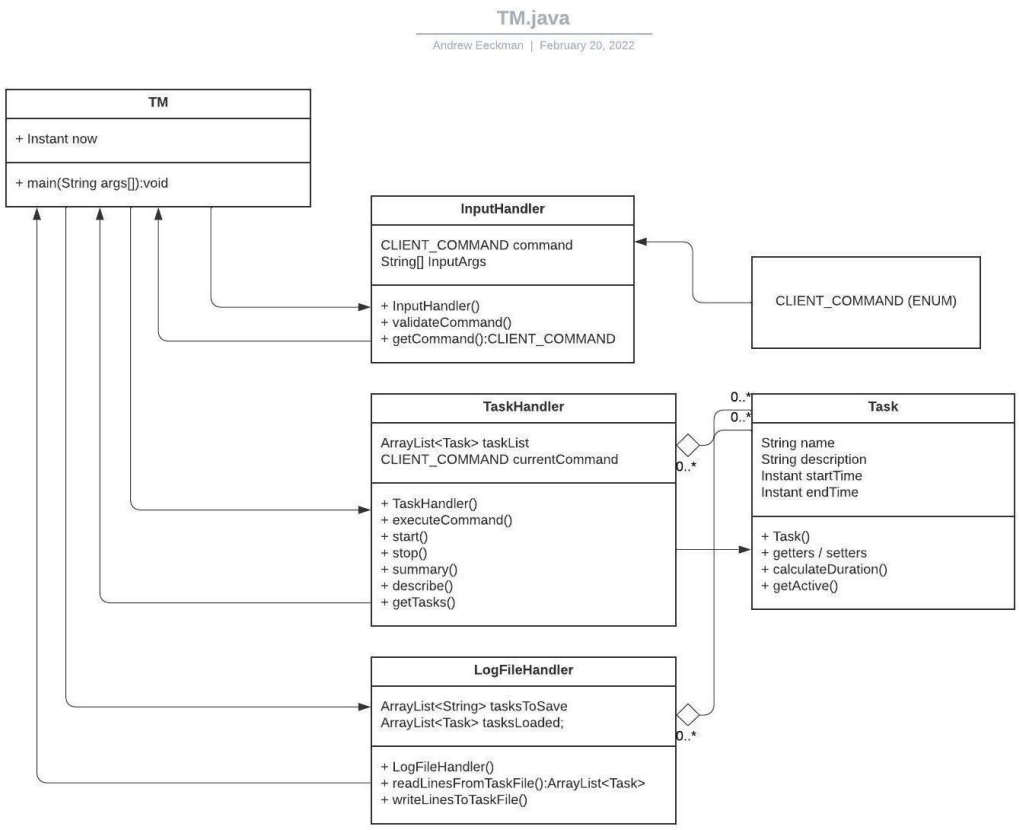
From the above, we get the following suggested class list: Task, TankHandler, InputHandler, and LogFileHandler. The Task class should be responsible for keeping track of the attributes associated with a given task and should be able to report on and change these attributes with getter and setter methods. The TaskHandler class should keep track of all Task objects in the program and make them accessible to the verified user commands (i.e. start, stop, describe, and summary). The InputHandler class should be able to verify the user’s command and the arguments provided alongside it with methods like validateCommand and validateCommandArgs. And finally, the LogFileHandler class should be the one responsible for reading and storing the current status of all initialized tasks currently loaded into the program.

Approaches to Class Modeling

In consideration of the design patterns we have thus far learned about in class, one potential approach to this problem is to utilize the State pattern. Tasks can occupy one of three potential states either “Completed” or “In Progress” or “Nonexistent”. Based on which state a given task is in, the results of some of the user commands will be different. For example, the start command should return an error message for a task that is currently “In progress” but it should restart a task that is “Completed.” As another example, the “summary” command when used on a “Nonexistent” task should not return anything but for either “Completed” or “In Progress” tasks, it should return their given details. While this is certainly one way of creating the desired application, such states could be potentially over-complicating what is essentially a simple problem.

In the absence of creating separate states, one could simply use a boolean value to indicate whether or not a task is currently active or in use (i.e. an isActive() function). This function or flag, while some might consider it to be an example of the anti-pattern, Primitive Obsession, is actually perfectly reasonable when representing either a “Completed” or “In Progress” Task. The boolean value returned by this function maps to the required domain space (i.e. a task is either active or inactive) and is only accessible through this function. We know that all finished tasks are assigned end times so the function “isActive()” would merely have to check if such a value exists. To tell if a task is “Nonexistent”, one merely has to look for it in the taskList held by the TaskHandler instance. Creating three separate classes to handle these flags creates a lot of unnecessary code and convolutes the project at hand. Also, by avoiding states, we can take the functions responsible for executing user commands out of these Task classes and put them in TaskHandler, making the execution of commands like “summary” much more succinct.

Ultimately, for these reasons, I am choosing to implement the above implementation and not the former. This implementation can be seen in the UML Class Diagram as seen below:



Reflection and Summary

Looking back at the decisions made above, I realize that the simple approach negatively harmed both the readability and organization of my code. While it allowed me to program the assignment much more quickly, it blurred the responsibilities held by each individual class. In particular, some of the methods meant for LogFileHandler ended up being a part of TaskHandler and some of the methods meant for TaskHandler ended up being part of Task. LogFileHandler which was meant to read and write lines to a text file, converting the lines into tasks on read and tasks into lines on write, ended up not having either of these latter responsibilities. The class ended up simply handling the strings meant to go to and from the application's persistent store, leaving any necessary conversions up to TaskHandler to deal with instead. And yet, this too became quite convoluted, as TaskHandler lacked all the necessary access to all of each Task's individual raw data. Therefore, the actual line that was saved by LogFileHandler was generated in Task through a call made in TaskHandler. It would have been much easier to have

LogFileHandler handle these conversions instead, giving the class access to the same strings made in Task and creating its own task list to be copied into TaskHandler. As for other issues I encountered along the way, I found it difficult to avoid using switch statements to manage program execution. And, in many parts of my code, I fear as though I may have become reliant on the Switch Statement Antipattern. Later in development, I tried to reverse course and implement the State design pattern discussed above to alleviate some of these problems; however, I had run out of time and such a drastic change was only likely to introduce more bugs and development bottlenecks. I feel as though I am still stuck in the past, reliant on old programming practices I learned years prior. In the future and for later projects in this class, I would like to stick more true to the design patterns and principles we have thus far learned.