TM Design Documentation

Ryan Stear, SID: 917196751

8 December 2023

1 Brief Introduction to the Design

The Simple Task Manager (TM) had three main hurdles to overcome in the design process: developing a command line interpreter, developing a data store for recording user activity, and designing an abstraction for tasks that adheres to the business rules.

2 Command Line Interpreter

2.1 Command Line Interpreter - Function

The overall function of the command line interpreter is quite simple: the user enters a command, the interpreter parses and validates this command, and then executes the user's input, if valid, or prints an error message if not. If the user input was valid, the command line interpreter records this command to the data store.

2.2 Command Line Interpreter - Design

I designed my command line interpreter to have two main functions: to validate user input and then execute valid commands. To this end, I chose to implement the command line interpreter as a Java Enum class named "Command". I chose to use an Enum class as an abstraction for the command line interpreter to take advantage of the Enum class' associated methods and because it allows for the easy addition of commands in future versions.

The advantages of the Enum class are made apparent when considering the cumbersome process of validating user input. Associated methods such as **valueOf(String)** allow for clean validation of user input by simply checking whether the first argument (the command) entered by the user can be parsed to one of Command's Enum objects; if not, then the command is invalid. Simple and comprehensible methods such as these allowed me to make Command into the sole data-validating mechanism in the design. This enabled a cleaner and more comprehensible structure to the code while remaining consistent with the principles of defensive programming.

The second main advantage of the Enum class is the ease with which it is possible to add additional commands to the interpreter. To add an additional command, all one need do (to the Command class) is to add a new Enum object and then add a supplementary parsing method for the new command (to handle arguments, syntax, etc.). Hence, choosing an Enum class as an abstraction for the interpreter allows for easy and convenient command refactoring and addition of new feature during future development.

Additionally, I designed the Command class to be responsible for instructing my implementation of the data store to write the successful command to the log. This seemed the natural choice; the Command class issues an order and then tells the data store to record it. Entries in the text file log or formatted as follows:

$$< \text{timestamp} >< \text{command} >< \text{arguments} >$$

3 Data Store

3.1 Data Store - Function

There is but one function of the data store: to record a (timestamped) history of successfully executed commands. Errors are not recorded to the data store to avoid its pollution with irrelevant information. Additionally, I designed the data store such that the "back-end" of the data store (that which actually *stores*, the data; i.e. a database, text log, etc.) does not affect the behavior of the *abstraction* of this back-end. This allows for a wide range of possible back-end methods of storing the data.

3.2 Data Store - Design

In order to design a data store with behavior that is invariant under the choice of the back-end storage method, I created a simple Java interface called "DataStore" with only two required methods: **String read()** and **void write(String)**. So long as any other back-end abstraction implements this interface, it is interchangeable with my chosen method: a human-readable text file log.

To create an abstraction of the text file, I created a class called "TextFileLog" which implements DataStore. Since there should only ever be a single log file, I chose to make TextFileLog adhere to the Singleton design pattern to prevent potential corruption of the text file resulting from multiple instances of TextFileLog all targeting the same file. The sole instance of TextFileLog has three fields: a File object corresponding to the real text file, a PrintWriter object for handling the **void write(String)** method, and a BufferedReader object for **String read()** method. All three of these objects are instances of classes from the **java.io** package. Thus, this design is very straightforward: to read from the text file, one need only call **read()**, and to write to the the text file, one need only call **write(String)**. The written line is preceded by a timestamp

specifying the date and time (format day/month/year-hours:minutes:seconds) at which the line was written.

4 Tasks and Business Rules

4.1 The Task Abstraction

A task tracker/manager must have some abstraction for an actual task. Hence, I created a Java class named "Task" to be this abstraction. The Task class has seven fields, three public and four private, which represent the state (active or paused, time elapsed) and meta-data (name, description, T-shirt size). I made all three of the meta-data properties public fields since they do not affect the state of the Task from the perspective of the task manager. This was also done to avoid the need for unnecessary getter and setter methods which would have simply exposed the meta-data fields anyways, had they been private. Additionally, the field representing the size of the task is also represented as an Enum class (named TaskSize) to again take advantage of the aforementioned (section 2.2) benefits of Enum classes.

The four private fields each represent an aspect of the state of a given task. These fields represent the starting time of the current time window, the ending time of the most recent time window, the total time elapsed over all time windows, and whether or not the task is currently active. Each of these fields are named after their purpose (startTime, endTime, totalTimeElapsed, and isTaskActive, respectively). Additionally, the Task class implements getter methods for totalTimeElapsed and isTaskActive and implements setters for startTime and endTime. This latter design choice may seem dubious, but it is worth noting that a Task object is only ever created when parsing command history from the log file. We'll touch more on this later (section 4.3), but for now, suffice it to say that there is no need to ever call these setters outside of parsing.

4.2 Time Management

I chose to measure time in my software relative to the Unix epoch using the **java.time.Instant** class. Therefore, time values are represented as **long** primitive types storing the number of seconds that have passed since 00:00 on January 1st, 1970. To implement this time system and cleanly handle all time related interactions, I created a Java class named "TMTimer". TMTimer is a small class implementing only four methods: one to get the number of seconds since the Unix epoch, one to generate a (string) timestamp using the SimpleDateFormat class in the **java.text** package, one to convert a arbitrary number of seconds into hours, minutes, and seconds, and one to format an arbitrary number of seconds into a human readable format (24-hour clock format with second precision). I chose to make each of these methods static since there is no need to ever generate an instance of TMTimer; it's purpose is just to increase the

cleanliness and comprehensibility of the code.

4.3 The TaskMap and Data Store Parsing

In order to produce a summary of all tasks on which a user is spending time, it is necessary to have a data structure that stores these tasks and a parser that loads them from the log file. To implement this, I designed a Java class named "TaskMap". This class is responsible for reading the command history of the data store and managing (creating, deleting, and modifying) Tasks accordingly. Additionally, I also chose to make this class adhere to the Singleton pattern, since it should only ever load tasks from the data store once (which is done upon constructing a TaskMap object).

TaskMap has only one relevant fields outside of its singleton instance field: a private HashMap object that stores Task objects as values that are keyed by String objects. I chose to use a HashMap because of its compatibility with the **java.util.stream** package and its ability to key Tasks by their name, which is a user provided argument in many commands (summary, describe, size, start, etc.).

When instantiated, TaskMap parses the command history from the text log according to the aforementioned log entry format. This is when commands "execute" their actual behavior; the TaskMap recreates the current state of the entire task manager by parsing line by line and altering Tasks and the HashMap accordingly. If the current command issued by the user is any command but summary, the TaskMap simply remains idle and is deleted when the program terminates. However, if the user requests a summary, the TaskMap uses the Java Streams interface to compute the necessary statistics and then prints the summary to **System.out**. I chose to use Java Streams here because the summary command requires operating on each element of the HashMap field, in which there are arbitrarily many Tasks stored. Thus, using Java Streams is both an appealing and convenient choice. Using lambda expressions and the **java.util.stream.Collectors** package, I designed three methods that computed each of the three required time statistics for tasks of a given size. The results are then printed on three separate lines (one for each statistic) and headed by a title line specifying the given size. However, these values are not printed if the user queries the task manager for information on a single task. In this case, TaskMap simply prints the time spent on that task. Additionally, to update the business rules and summary format, one would only need alter the summary methods in the TaskMap class.