Task Tracker Design Document

Maksym Turkot
04/26/2021

Introduction

The program implements an interactive task tracker program using MVC architecture. This document outlines the structure of the project program, goals achieved, examples of how files are formatted, how tags and tasks are stored in the data structure, classes, methods, data used, and a description of new challenges encountered in the process.

Project Goals

The overall goal of the project is to develop a task tracking program.

1. **Program Interface.** The program has a terminal-based interface where the user types different commands to interact with the program. The goal was to develop such interface functionality that prints information from the Model to the terminal, waits for user to type a command, identifies the command, and calls Controller to perform corresponding action to manipulate the data.

2. **Tag manipulation.** Functionality for adding, removing, and displaying tags. When tag is removed, or when upon task creation no tag is specified, a default “untagged” tag will be added to the task.

3. **Task manipulation.** Functionality for adding a task, adding tags to a task, editing task description and due date, deleting a task, saving task list to a specified file.

4. **Task searching.** Implement functionality to search for tasks with different combinations of attributes. Ability to search tasks by: a single tag, a single date, a single date range, multiple tags, dates, and date ranges with OR, multiple tags, dates, and date ranges with AND, combinations of each, and a search by text in the task description.

5. **Saving tasks.** This goal involves creating functionality to write information to files, both by overriding existing file, and by adding information without destroying contents.

6. **Reading from files.** An ability to read a file of the same format as a returned task file, and store all relevant information. An ability to read a help file and provide all necessary information to the Interface (The help file contains the initial message and explanation for each command that can be retrieved).

7. **Error Handling.** Verifying if date and date ranges are correctly formatted and make sense, if duplicate task is being created, if non-existent tag is being used, if empty task (or one containing only spaces) is being created, and if entered command does not exist.

8. **Using MVC Design.** Model will be implemented by a class that will reference Tasks, and use Lists to store data. View will be implemented in a form of Interface class, that will

handle information output to the terminal and interaction with the user. Controller class will store the functionality for various operations and will be used by the Interface class to access data from Model.

File Formatting

File formats were simplified, which made parsing much easier.

The Task file will be formatted in the following way:

tags

tag1,tag2,tag3,tag4,

tasks

task1,04/11/2021,tag1,tag2,

task2,04/15/2021,tag1,tag2,tag3,

task3,04/19/2021,tag2,

task4,04/11/2021,tag1,tag2,

task5,04/11/2021,untagged,

Space after tags line and after the final task is required.

The Help file will be formatted in the following way:

msg

Type any of the following commands to get more information:

e / exit

h / help

atg / add-tag

det / delete-tag

…

e

Asks if tasks should be saved and exits the program

h

Displays help menu

atg

Adds a new tag with an entered name

…

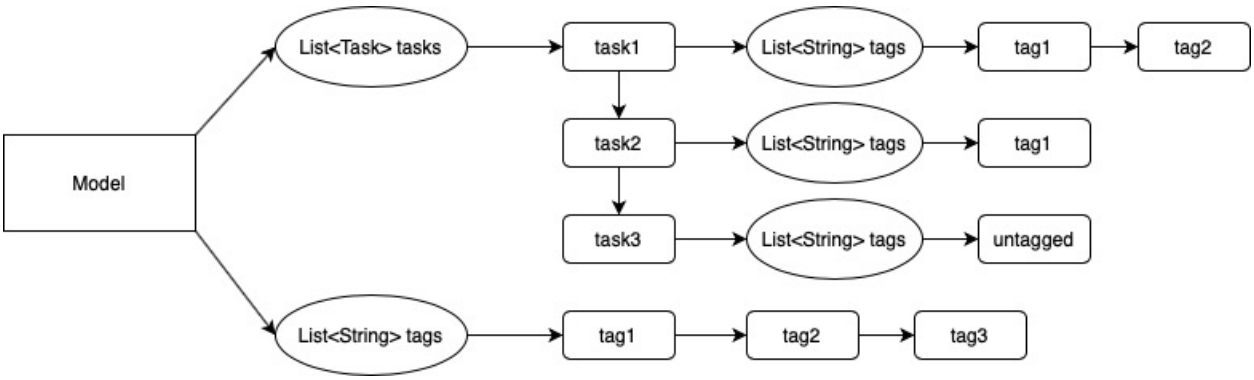
Space after each block of info is required.

Data Organization

Tag organization: Tags that can be used for task reference are stored as strings in a List object inside of the Model class. Tags are also stored inside of lists associated with each object.

Task organization: Tasks are stored in a List object of tasks inside of the Model class. Once user searches for tasks, tasks with corresponding attributes are copied to an output list, also stored inside of a Model class. Each task will have a tag list associated with it.

Following diagram shows how tags and tasks are structured:



Program Organization

Class Interface

This class runs the program, reading user's commands and triggering actions. It is the sole class that is allowed to interact with the user, that is, only Interface can interpret commands, or print data for a user to see. To make interface focus only on formatting and reading data, all error-handling is passed on to other parts of a program, including Controller, Model, and other helper classes.

Fields:

- None

Methods:

- static void main(String[] args) - runs a while loop which continuously reads user's commands, interprets them, and calls other parts of the program to perform various actions accordingly.

Class Controller

This class contains methods that perform various actions based on what Interface requested. It calls Model to return, change, or remove data, and calls HelpFile to retrieve information from a file.

Fields:

- static List<String> taskTags - temporary storage of task tags during task creation
- static DateTimeFormatter formatter - formatter of LocalDate data
- static Scanner input - scanner for command reading

Methods:

- static String runExitCommand(String command) - based on passed command, provides logic of checking whether user wants to save current tasks, and directs next actions accordingly, either telling interface to exit the program, call runSaveTasksCommand(), or telling Interface that command was not recognized.
- static String runHelpCommand(String command) - based on passed command, calls HelpFile class to read a specific portion of the help.txt file and returns passed info, or tells Interface that command was not recognized. *TESTED by checking if command info is printed in the console.*
- static String runAddTagCommand(String tag) - calls Model to add a passed new tag to the system. Checks if tag already exists in the system and returns an error

message if so. *TESTED by automatically checking if tags are correctly added to the system.*

- static String runDeleteTagCommand(String tag) - calls Model to remove the passed tag from the system. Checks if tag exists in the system before the removal. If not, returns an error message. *TESTED by automatically checking if tags are correctly removed from the system.*
- static String runDisplayTagsCommand() - calls Model to return a list of tags in the system. *TESTED by automatically checking if a string of tags is returned correctly.*
- static String runAddTaskCommand(String task, String dueStr, String tag) - calls model to add a new task to the system. Checks if task already exists in the model, if date was provided in a correct format, and if tags being added exist in the system, or if duplicate tags are being added, in which cases returns an error message. *TESTED by automatically checking if a new tag is correctly added to the system.*
- static void runSearchTagsCommand() - calls Model to search for tasks containing given set of tags.
- static void runSearchDatesCommand() - calls Model to search for tasks containing given dates.
- static void runSearchTagsDatesCommand() - calls Model to search for tasks containing given set of tags and dates.
- static void runSearchTextCommand() - calls Model to search for tasks containing given text.
- static String runSaveTasksCommand(String filename) - calls Model to trigger process of saving tasks currently in the system to a file with a passed name. If error occurs, returns an error message. *TESTED by checking if controller correctly triggers saving of tasks to a file with specified name.*
- static String runLoadTasksCommand(String filename) - calls Model to trigger process of loading tasks from a specified file. If error occurs, returns an error message. *TESTED by automatically checking if controller correctly triggers loading of tasks from a specified file.*
- *static LocalDate parseDate(String date) - helper method that creates a LocalDate from a string.*
- *static void printItemizedList(List<Task> resultTasks) - prints and handles commands in an itemized list of searched tasks. TESTED during program test runs.*

Class Model

This class stores tags and tasks present in the system. It has a sole access to this information, and any part of a program that wishes to access this data has to use Model's getter method. Model also performs various operations to manipulate data, calling relevant helper methods. To perform many of such operations, program has to use Model.

Fields:

- static List<Task> tasks - private list storing tasks present in the system.
- static List<String> tags - private list storing tags present in the system.

Methods:

- static List<String> getTags() - returns a list of tags. *TESTED by automatically checking if model correctly returns a list of tags.*
- static void addTag(String tag) - adds a specified tag to the tags list. *TESTED by automatically checking if model correctly adds a new tag to the tags list.*
- static boolean removeTag(String tag) - removes a specified tag from the system and from all lists that are associated with it. If a task had no other tags other than the tag removed, it is tagged with "untagged" tag. *TESTED by automatically checking if model correctly removes a tag, and all its occurrences in tasks, replacing it with "untagged" tag if needed.*
- static void clearTags() - removes all tags from the tags list and from all tasks, marking them as "untagged". *TESTED by automatically checking if model correctly removes all tags from the system, and if all tasks are marked as "untagged".*
- static List<Task> getTasks() - returns a list of tasks. *TESTED by automatically checking if model correctly returns a list of tasks.*
- static boolean containsTask(String task) - checks if a task with a given text exists in the system. *TESTED by automatically checking if presence of a task is correctly determined.*
- static void addTask(String task, LocalDate dueDate, List<String> tempTags) - adds a specified task to the system. *TESTED by automatically checking if model correctly adds a new task.*
- static boolean removeTask(String task) - removes a task with a given text from the system. *TESTED by automatically checking if model correctly removes a task with a given text from the system.*
- *static void clearTasks()* - removes all tasks from the tasks list. *TESTED by automatically checking if model correctly removes all tasks from the system.*

- static List<Task> searchTags(String tags) - searches the system for tasks with a given set of tags. *TESTED by automatically checking if model correctly finds tasks with given combinations of tags.*
- static List<Task> searchDates(String dates) - searches the system for tasks with a given due date. *TESTED by automatically checking if model correctly finds tasks with given due date.*
- static String saveModel(String filename) - triggers SaveModelFile class to save tags and tasks currently in the system to a specified file. If error occurs, message is returned. *TESTED by checking if model correctly triggers process of storing model data in a file.*
- static String loadModel(String filename) - triggers LoadModelFile to load data from a specified file. If error occurs, message is returned. Otherwise, current model is cleared and new one loaded. *TESTED by automatically checking if model correctly triggers process of loading model from a file.*

Class List<E>

This generic class constructs lists that are used throughout the program. It also implements major functionality that allows for an effective manipulation of data.

Fields:

- Node<E> head - first element of a queue

Methods:

- int size() - returns the number of elements in this list.
- boolean isEmpty() - returns true if this list contains no elements. *TESTED by automatically checking if correct length of a list is returned.*
- boolean contains(E n) - returns true if this list contains the specified element. *TESTED by automatically checking if presence of an element is correctly identified.*
- String toString() - returns current list in a form of a CSV string. *TESTED by automatically checking if correct string is returned.*
- boolean add(E e) - appends the specified element to the end of this list. *TESTED by automatically checking element is correctly added to the list.*
- boolean remove(E o) - removes the first occurrence of the specified element from this list, if it is present. If this list does not contain the element, it is unchanged. *TESTED by automatically checking element is removed from the list.*
- void clear() - removes all of the elements from this list. *TESTED by automatically checking if all elements are removed from the list.*

- E get(int index) - returns the element at the specified position in this list. *TESTED by automatically checking correct element is returned from the list.*
- E set(int index, E element) - replaces the element at the specified position in this list with the specified element. *TESTED by automatically checking if element is correctly replaced.*
- void add(int index, E element) - inserts the specified element at the specified position in this list. *TESTED by automatically checking if element is correctly inserted at a specified index.*
- E remove(int index) - removes the element at the specified position in this list. *TESTED by automatically checking if element at the specified index is correctly removed.*
- int indexOf(E n) - returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. *TESTED by automatically checking if index of a first occurrence of a specified element is returned.*
- int lastIndexOf(E n) - returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element. *TESTED by automatically checking if correct index of a last occurrence of a specified element is returned.*
- List<E> subList(int fromIndex, int toIndex) - Returns a copy of the portion of this list between the specified “fromIndex”, inclusive, and “toIndex”, exclusive. *TESTED by automatically checking if a sublist of a list is correctly generated.*

Class Node<E>

This class constructs a node of a List object.

Fields:

- E val - value stored in the node.
- Node<E> next - next node in the list.

Methods:

- E getVal() - returns node’s value.
- Node<E> getNext() - returns next node in the list.
- void setVal(E val) - replaces node’s value.
- void setNext(Node<E> next) - replaces next node in a list.

Class Task

This class constructs task objects.

Fields:

- static int taskCnt - counter of tasks.
- List<String> tags - list of tags.
- String task - text of a task.
- LocalDate dueDate - due date of a task.
- int id - unique id of a task.

Methods:

- List<String> getTags() - returns the list of tags.
- String getTask() - returns task text of a task.
- void setTask(String task) - sets a new task.
- LocalDate getDueDate() - returns the due date of a task.
- void setDueDate(LocalDate dueDate) - sets a new dueDate.

Class HelpFile

This is a helper class which manages retrieval of information from a “help.txt” file.

Fields:

- None

Methods:

- static String readHelpFile(String command) - reads a portion of information stored in the help.txt file, based on a specified command. If error occurs, message is returned. *TESTED by automatically checking if correct info is printed in the terminal.*

Class SaveModelFile

This is a helper class that saves current model data into a file.

Fields:

- static PrintWriter fileWriter - file writer.
- static File outputFile - output file.

Methods:

- static String writeFile(String filename) - creates new file with a specified name, and triggers writeTags and writeTasks to produce strings of data that are printed in the file. *TESTED by manually checking if file with Model data is created.*
- static String writeTags() - produces a string of tags. *TESTED by automatically checking if tags are converted to string.*
- static String writeTasks() - produces a string of tasks. *TESTED by automatically checking if tasks are converted to string.*

Class LoadModelFile

This is a helper class that loads model data from a specified file, overriding current model data.

Fields:

- None

Methods:

- static List<String> readTags(String filename) - searches file for tags and reads this data. *TESTED by automatically checking if tags are correctly loaded from a file.*
- static List<Task> readTasks(String filename) - searches file for tasks and reads this data. *TESTED by automatically checking if tasks are correctly loaded from a file.*

Class TaskFile

This is a helper class that saves current itemized task list into a file.

Fields:

- None

Methods:

- static boolean writeTasks(String list, String filename) - writes given list of tasks to a given file. Appends tasks to an existing file, or creates a new one. *TESTED by automatically checking if tasks are correctly appended to an existing or a new file.*

Challenges

- Distributing functionality between MVC blocks was more challenging that expected. I believe the basic structure is there; however, if I were to code it again it would be cleaner.
- Limiting functionality of Interface was quite challenging. Having controller perform all checking of values, while having Interface wait for user to type new answer if error occurs was sometimes confusing. In some methods I followed this approach, in others I had controller itself interact with a user for a brief portion of the program, such as in the printItemizedList() method. Drawing a line between formatting data (Interface) and controlling actions (Controller) was a little confusing.
- In Controller the temporary list was really tricky. I could't understand why Task and its due date are correctly added to the system, but it's tag list is not. I then realized that "temporary" list wasn't so temporary, since it was still pointing to the list of tags in the model. So when I cleared it for the next task, all tags for previous task would disappear. Using sublist method to create a copy list resolved the situation.
- I ran out of time to fully implement some search mechanisms. However, I feel I have had an idea how to tackle them.