Task Tracker User Report

Maksym Turkot
04/26/2021

Introduction

This document outlines overview of project goals, challenges, and solutions to those challenges. It also provides a user guide, a diagram of major project components, program verification discussion, and performance analysis.

Project Goals

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

Program Interface:

Goal:

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.

Challenges:

As mentioned in final design document, splitting functionality among Controller and Model was not the simplest task.

Solutions:

In some cases I had controller method return status strings that allowed Interface to take appropriate next steps, in others I allowed controller to have brief interactions with the user.

Tag manipulation:

Goal:

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.

Challenges:

When a tag is removed, it must also be removed from all tasks containing it.

Solutions:

I had Model contain the functionality to check if tasks in it contained given tag. Then, it would remove this tag, and add an “untagged” tag to every task without tags.

Task manipulation:

Goal:

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.

Challenges:

When adding a new task, it is difficult to keep track if all information entered is correct. Having tags stored in a temporary list required copying them at a later point.

Solutions:

A careful and thoughtful implementation of the method.

Task searching:

Goal:

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.

Challenges:

While tags searching was fairly straightforward, date-range search was more challenging.

Solutions:

I focused on fully implementing the tag search and single date search commands, along with AND and OR functionality.

Saving tasks:

Goal:

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

Challenges:

Making sure that the output file is of the same readable format as the input was an interesting challenge.

Solutions:

Having a relatively simple file structure helped with implementation and provided a readable format.

Reading from files:

Goal:

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).

Challenges:

Only relevant information was supposed to be read from the help file.

Solutions:

Reading from a help file required a very specific reading method, that went line by line through the file reading only relevant info.

Error Handling:

Goal:

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.

Challenges:

As mentioned previously, keeping track of a few layers of error identification to provide timely warnings to the user was a challenging task.

Solutions:

Having Controller notify the Interface of its status was an effective solution.

Using MVC Design:

Goal:

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.

Challenges:

Splitting functionality between Interface and Controller was sometimes a tricky task due to close interaction between the two.

Solutions:

A concrete structure of the program made the task a little easier, as methods worked very similarly in terms of communication between Interface and Controller.

User Guide

PROGRAM STRUCTURE:

In the project folder user may find the “data” folder. In it, there is a “help.txt” file, containing help information, and “model1.txt” file, which contains tags and tasks that can be loaded into the system.

CONFIGURING AND RUNNING THE PROGRAM:

In order to configure the program:

User may change configuration by changing or adding a new model file. File must follow the format described in the Final Design document for the program to run correctly.

In order to run the program:

1. Run the “package.bluej” file.
2. Right-click the “Interface” class.
3. Click on the “void main(String[] args)” method.
4. Click “OK” in the “Method Call” window that pops up.
5. The program will run and open a terminal window with a field where to type commands.

6. To get information about what commands are available, type “help” or “h” and hit return key.
7. Examples of features from the program:

• Exiting program and saving tasks:

```
> exit
Would you like to save your tasks? (Y/N): Y
File name: model2.txt
Exiting program
```

• Running a help command and getting info fo a specific command:

```
> help
Type any of the following commands to get more information:
e    / exit
h    / help
atg  / add-tag
det  / delete-tag
dit  / display-tags
atk  / add-task
st   / search-tags
sd   / search-dates
std  / search-tags-dates
stxt / search-text
stk  / save-tasks
ltk  / load-tasks

Command help: atk
Adds a task, its due date, and all associated tags. If no tags are provided, task will be
marked as "untagged".

Command help:
>
```

• Adding tags:

```
> add-tag
Tag: tag1
Tag: tag2
Tag: tag3
Tag: tag2
Tag tag2 already exists
Tag: tag4
Tag:
>
```

- Displaying and deleting tags

```
> display-tags
tag1,tag2,tag3,tag4,
> delete-tag
Delete tag: tag2
Delete tag: tag4
Delete tag: tag5
Tag tag5 doesn't exist
Delete tag:
> dit
tag1,tag3,
>
```

- Adding a task

```
> add-task
Task: task1
Due date: 04/25/2021
Tag: tag1
Tag: tag2
Tag: tag4
Tag tag4 does not exist.
Tag: tag2
Tag tag2 already assigned.
Tag:
>
```

- Saving and loading tasks

```
> save-tasks
File name: model3.txt
> load-tasks
File name: model1.txt
>
```

- Searching for tasks with tags:

```
> st
Tags to search: tagA
1: task0 (04/25/2021)
2: task1 (04/26/2021)

a. add-tag b. edit c. delete d. save
Action:

> st
Tags to search: AND tagA tagB
1: task0 (04/25/2021)

a. add-tag b. edit c. delete d. save
Action:

> st
Tags to search: OR tagA tagB
1: task0 (04/25/2021)
2: task1 (04/26/2021)

a. add-tag b. edit c. delete d. save
Action:
```

- Adding tags to a task in an itemized list:

```
> st
Tags to search: tagA
1: task0 (04/25/2021)
2: task1 (04/26/2021)

a. add-tag b. edit c. delete d. save
Action:
a
Task number: 1
Tag: tagC
This task already has this tag.
Tag: tagD
Tag:
a. add-tag b. edit c. delete d. save
Action:
```

- Editing a task:

```
> st
Tags to search: tagA
1: task0 (04/25/2021)
2: task1 (04/26/2021)

a. add-tag b. edit c. delete d. save
Action:
b
Task number: 1
a. task b. date
Action: a
Task: newTask
a. task b. date
Action:
a. add-tag b. edit c. delete d. save
Action:

> st
Tags to search: tagA
1: newTask (04/25/2021)
2: task1 (04/26/2021)

a. add-tag b. edit c. delete d. save
Action:
```

```
> st
Tags to search: tagA
1: task0 (04/25/2021)
2: task1 (04/26/2021)

a. add-tag b. edit c. delete d. save
Action:
b
Task number: 2
a. task b. date
Action: b
Due date: 01/01/2021
a. task b. date
Action:
a. add-tag b. edit c. delete d. save
Action:

> st
Tags to search: tagA
1: task0 (04/25/2021)
2: task1 (01/01/2021)

a. add-tag b. edit c. delete d. save
Action:
```

Project Components and Diagram

COMPONENTS:

1. 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.

2. 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.

3. 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.

3. 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.

4. Class Node<E>

This class constructs a node of a List object.

4. Class Task

This class constructs task objects.

5. Class HelpFile

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

6. Class SaveModelFile

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

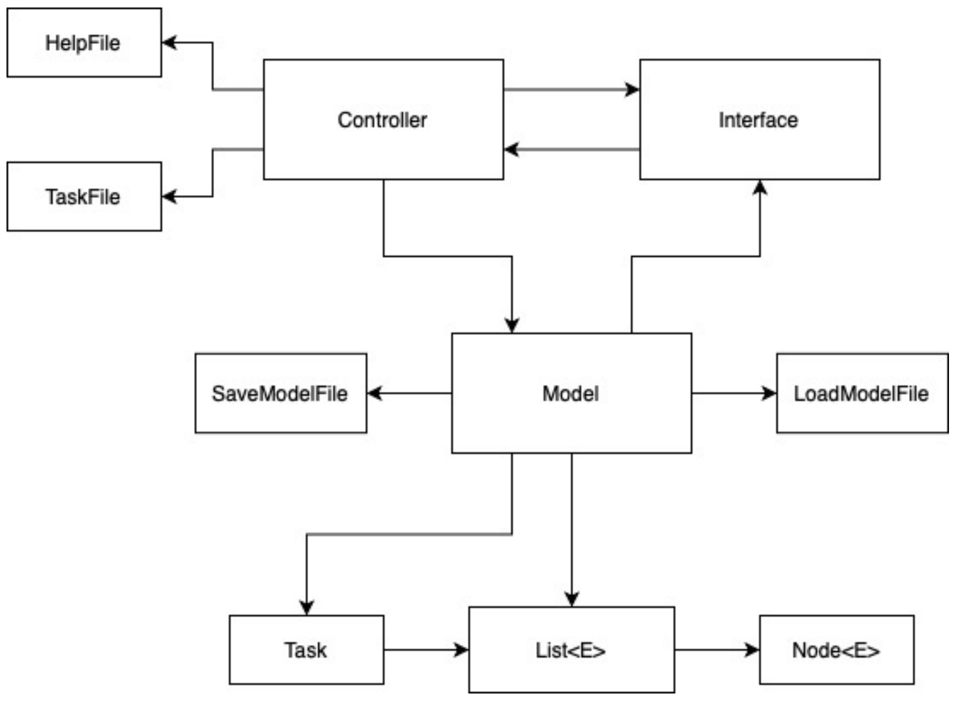
7. Class LoadModelFile

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

8. Class TaskFile

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

DIAGRAM:



Verification of Program's Functionality

Program's functionality was verified with multiple runs of the program with different model files, performing various combinations of operations.
To test the interface, each command was run a couple of times, providing different inputs and checking how program reacts to intentionally invalid input. To test the overall performance of a program, the following commands were executed in the corresponding order:

1. Run the program.
2. Load the saved file.
3. Display tags.
4. Add tags.
5. Display tags again to verify the change.
6. Delete some tags.
7. Display tags again to verify the change.
8. Add tasks.
9. Search for tasks with given tags.
10. Add new tags to the first task in the list.
11. Edit information of the second task.
12. Delete the the third task.
13. Save tasks.
14. Save model.

All key functionality was automatically unit-tested. Some unit tests printed output into the terminal to manually verify validity of output, for example, readHelpFileTest.

Data Analysis

This project produced no data to be analyzed.