jTasks: A task application developed in Java with third-party support for Notion

Radoslaw Dulny

Facultad de Ciencias - Universidad de Salamanca

Programación III

ET6524099

December 8, 2024

Introduction

This document outlines the design and functionality of a program called **jTasks**, which manages task creation, storage, and retrieval. It emphasizes key design choices such as using a ***HashMap*** for efficient task storage, adopting a standardized date format, and enabling integration with the ***Notion API***. The application uses the ***Model View Controller (MVC)*** architecture for modularity and supports exporting/importing tasks in **CSV** and **JSON** formats. The interface allows users to perform CRUD operations, sort tasks by priority or date, and manage ***Task*** repositories, with robust error handling throughout. The conclusion highlights areas for improvement, including enhanced user interfaces, flexible save/load directories, and integration with additional third-party platforms.

Design Choices

Before going into the details of how the program works I wanted to highlight some of the design choices that I made throughout the program:

Storing tasks as a *HashMap* in *BinaryRepository*: Storing tasks as a *HashMap* allows for O(1) search time when searching by the *UUID* identifier.

modifytask() uses addTask() in Binary Repository: *modifyTask()* simply uses *addTask()* as it replaces the corresponding identifier entry in the HashMap of tasks.

Using static final path in Exporters: This will ensure that the path will remain the same whether the app is exporting or importing.

Imports and Exports in BaseView: Import and export views are implemented in *BaseView* as they will be the same no matter what the view is.

Task Sorting: Tasks are sorted by priority by default but *DateComparator* is available for actions such as exporting and task history.

YYYY-MM-DD Date Format: Using the same date format in the entire program minimizes the chance of errors during conversion.

Binary Repository is used if API Key/Database ID is incorrect: The user is informed that the connection has been unsuccessful and that the Binary Repository is used.

User Interface (Interactive View)

The ***InteractiveView*** UI has been implemented in a way that gives the user the ability to navigate through the program with ease and access all of the available options that it provides. The ***Controller*** has been set up in a way that allows for easy implementation of new views. At the start, the program displays a message about the connected repository (with messages also available for unsuccessful connections) as well as a list of options available to the user:

```
Attempting Notion connection...
Successfully connected to the Notion Repository.
```

or

```
Loading tasks from the Binary Repository...
Previous tasks have been loaded successfully.
```

```
Welcome to the Interactive View of jTasks!

Select one of the following options:
1. Add new task
2. List due tasks
3. List task history
4. Export to JSON
5. Export to CSV
6. Import from JSON
7. Import from CSV
8. Exit

Selection:  (1 <= numero <= 8)
```

CRUD Operations

All CRUD operations are handled using a **try-catch** statement that throws a ***RepositoryException*** with a description in the case of an error. The user is also notified if their changes have been executed successfully.

🟢 Add New Task

Users are asked for a title, content, priority and an estimated duration which is the minimum amount of information required to create a new ***Task***. The ID of a ***Task*** is assigned to it during its creation in the constructor with a random ***UUID***. Existence of a ***Task*** with the same ID is checked in the model before adding to the corresponding repository.

```
Enter the name of the task: Sample Task Name
Describe the task: Sample Task Description
What's the priority of the task (1 (Low) - 5 (High))? (1 <= numero
<= 5) 5
How long do you think this task will take to complete (minutes)?
25
Task uploaded successfully.
```

🟣 List Due Tasks

A list of tasks that have not been completed is fetched from the repository through the ***Controller*** and ***Model***. These are ranked by priority using the comparable property of the ***Task*** class. The user is informed whether the fetch was successful and the list empty.

```
Due tasks (Ranked by Priority):
1. Sample Task A (5)
3. Sample Task C (4)
4. Sample Task D (2)

Select task by number to edit/view task details or press 0 to go
back:
```

🟣 List Task History

Similar to above but this time all of the tasks are shown and are sorted by date instead of priority using the ***DateComparator*** class.

```
All tasks (Sorted by Date):
1. Sample Task C (2024-12-03)
2. Sample Task A (2024-12-05)
3. Sample Task B (2024-12-06)
4. Sample Task D (2024-12-10)

Select task by number to edit/view task details or press 0 to go
back:
```

🟣 Select Task

After either of the lists are presented, the user can choose the ***Task*** that they would like to view/edit by using the position in the list or exiting back to the main menu by pressing '0'. In the case that a ***Task*** is chosen, its details are displayed and editing options are given. The options below are continually shown until the user either returns to the main menu or deletes the ***Task***.

```
Task Details:
Task ID: 73b09e5c-ad81-45c0-b516-80991214f21c
Title: Sample Task
Date Added: 2024-12-05
Description: Sample Description
Priority: 5
Estimated Duration: 5 minutes
Completed: No

Select one of the following options:
1. Mark as completed
2. Edit title
3. Edit date
4. Edit description
5. Edit priority
6. Edit estimated duration
7. Delete task
8. Return to main menu
```

```
Selection:  (1 <= numero <= 8)
```

🔵 Mark as Completed/Uncompleted

This option is presented giving the user the ability to toggle between a ***Task*** being completed or uncompleted based on the current state of the completed boolean.

```
Completion status changed successfully.
```

🔵 Edit Title

Allows the user to edit the title of the ***Task***.

```
Enter new title: Sample Title
Title changed successfully.
```

🔵 Edit Date

This asks the user for a year, month and date and converts it to a ***Date*** object that can be assigned to the ***Task***. The maximum day that a user can enter depends on the year and month inputted.

```
Enter year of new date (e.g. 2024): 2024
Enter month of new date:  (1 <= numero <= 12)12
Enter day of new date:  (1 <= numero <= 31)10
Date changed successfully.
```

🔵 Edit Description

Allows the user to edit the description/content of the ***Task***.

```
Enter new description: Sample Description
Description changed successfully.
```

🔵 Edit Priority

Allows the user to edit the priority of the ***Task***.

```
Enter new priority (1 (Low) - 5 (High)):  (1 <= numero <= 5)5
Priority changed successfully.
```

🔵 Edit Estimated Duration

Allows the user to edit the estimated duration of the ***Task***.

```
Enter new estimated duration (minutes): 20
Estimated duration changed successfully.
```

🔴 Delete Task

Allows the user to remove the ***Task*** that is currently selected.

```
Task Sample Task with ID db2735e5-e903-49cc-8306-e8044e6893d9 has
been removed successfully.
```

Export/Import (CSV/JSON)

The view functionality of these options is placed in the ***BaseView*** class as it will be equal among all views. All operations in this section throw ***ExporterException*** in the case of any errors and let the user know what the problem is.

Export CSV/JSON

The exporter of the program is set to the ***CSVExporter/JSONExporter*** using the ***ExporterFactory***. Tasks are exported to the home directory of the user with the name ***output.csv/output.json.***

```
Tasks have been exported successfully. You can find them in your
home directory.
```

Import CSV/JSON

The exporter of the program is set to the ***CSVExporter/JSONExporter*** using the ***ExporterFactory***. Tasks are imported from the file ***output.csv/output.json*** in the home directory of the user. The user is informed if this file does not exist.

```
Tasks have been imported successfully.
```

Task Object

The ***Task*** object has all the variables required as well as a ***Serial Version UID*** in order to be ***Serializable***. Tasks are identified using a ***UUID*** identifier that is generated at creation. ***Task*** is ***Comparable*** and sorted by priority by default.

Repositories/Persistence

The repository can be set by the user using command line arguments when launching the application, which are specified below (***BinaryRepository*** by default). Persistence varies based on the repository chosen by the user.

BinaryRepository

In the case of ***BinaryRepository***, it can be serialised in a **.bin** file. Thanks to the repository being ***Serializable***, the ***Model*** can save data that can then be loaded into the program at startup the next time a user decides to use ***BinaryRepository***.

```
java -jar app.jar --repository bin
```

NotionRepository

In the case of ***NotionRepository***, the data is uploaded to **Notion** with every operation. This data is available to use whenever the user next uses the application and decides to use ***NotionRepository***. In order to connect, the **API Key** and **Database ID** have to be specified by the user in the command line arguments.

```
java -jar app.jar --repository notion "API_KEY" "DATABASE_ID"
```

Robustness

All of the interaction between the user and the program is robust and resistant to incorrect input. In the case of incorrect input the user is guided to give correct input and in the case of exceptions outside of the control of the user, the user is informed about the problem and given a detailed description of what went wrong.

MVC Architecture

The program is designed using a ***Model View Controller*** architecture model that allows for the interchangeability of **Views**, **Models**, **Repositories** and **Exporters**. This allows for any developers to easily add new features and segments to the application.

One More Thing (Notion Integration)

The application has a real time connection to **Notion** thanks to the **Notion API**. Any changes made by the user in **jTasks** are reflected in **Notion** and vice versa.

Conclusion

I didn't come across any major problems when developing the project but here are some things that could be implemented/improved in order to make a better application:

Adding more views: A view with a more user friendly interface rather than a text based version would enhance the quality of the application and increase the probability of users using it day to day.

Ability to change save/load directory: Not every user wants to use the home directory as the default so it would be beneficial to have the ability to change this.

Support for more third-party applications: Support for third-party applications other than Notion would again attract more users.

Notification System: A notification system that reminds users to complete their tasks could be implemented.