Developer Guide

Design & implementation

Architecture

The ***Architecture Diagram*** given above explains the high-level design of the ATHENA. Given below is a quick overview of each component.

`Main` has one class called `Athena`. It is responsible for,

- At app launch: Initializes the components and connects them up with each other.
- At app shut down: Shuts down the components.

The rest of the App consists of these components.

- `Ui`: The UI of ATHENA.
- `Parser`: Parses user input and command executor.
- `TaskList`: The list that stores the user's tasks.
- `Storage`: Reads data from, and writes data to, the hard disk.

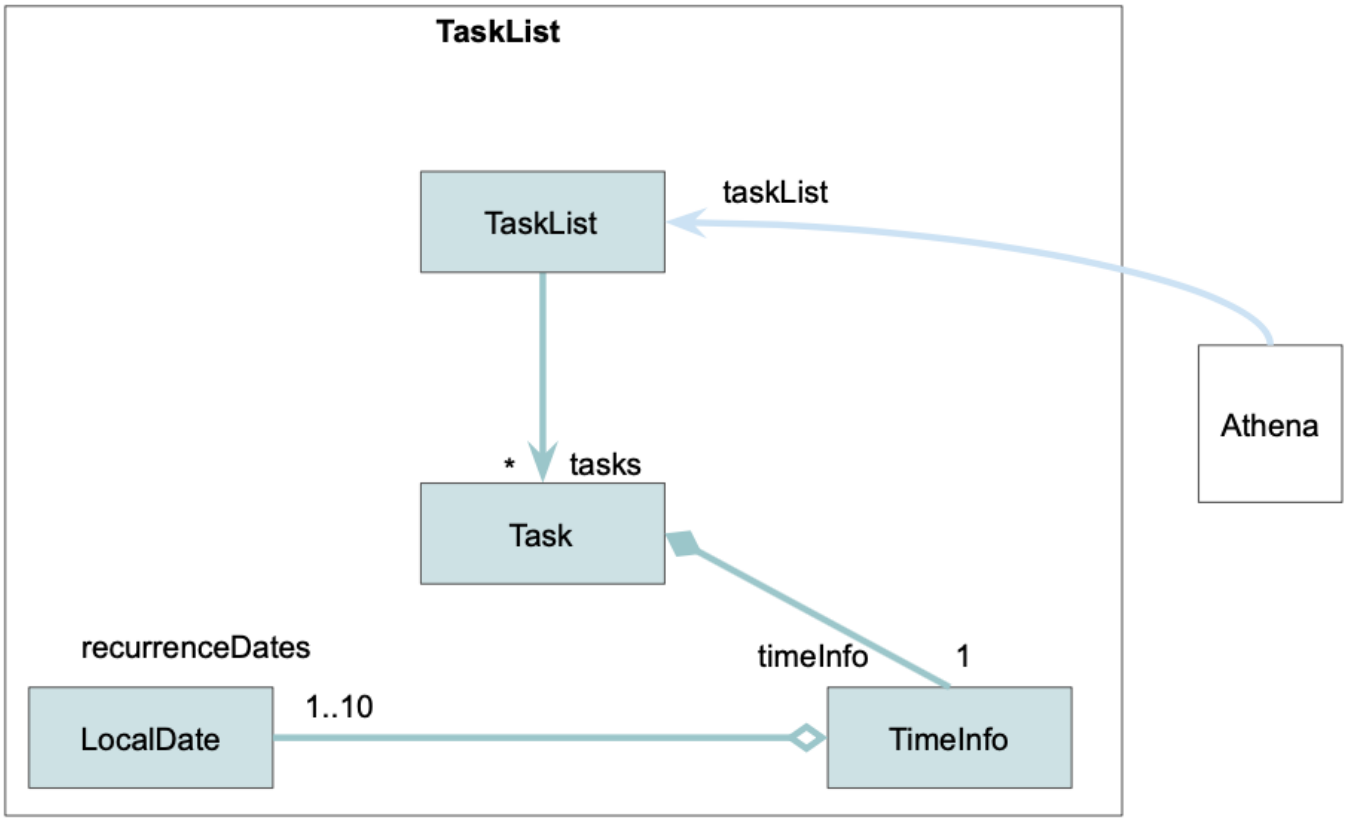
UI component

API : `Ui.java`

Parser component

API : `Parser.java`

TaskList component



API : `TaskList.java`

1. The `TaskList` stores task data in `Task` type objects.
2. The `TaskList` is updated in `Athena`.
3. A new `Task` object is created everytime the user uses the command `add`.

Storage component

API : `Storage.java`

Other Guides: Documentation, logging, testing, configuration, dev-ops

This section contains links to other relevant guides.

- Documentation guide
- Testing guide
- DevOps guide

Implementation

This section describes some noteworthy details on how certain features are implemented.

Time allocation to task in timetable

The time allocation mechanism is facilitated by `TimeAllocator`. It allocates time slots to `Task`s in a `TaskList` that are not assigned a fixed time slot by the user. It implements the following operations:

- Work in progress hehe

Given below is an example usage scenario and how the allocation mechanism behaves at each step.

Step 1. The user launches the application. The *data.csv* file located next to the application jar file contains 5 tasks. These tasks are loaded into the `TaskList`. 3 of them have a fixed time slot, while the other 2 are not assigned any time slot.

Step 2. The user executes `list` to get an overview of the week. The user sees the 3 tasks with a fixed time slot in the printed timetable.

Step 3. The user executes `allocate` to let the application allocate time slots to `Task`s without a fixed time slot.

`TimeAllocator` iterates through the `TaskList`, and checks whether a `Task` has a fixed time slot or should be allocated by the application to an empty slot. Tasks that have a fixed slot are added to a `fixedTaskList`, while the rest are added to a `flexibleTaskList`. The `Task`s in the `flexibleTaskList` are sorted based on their `importance` and `deadline`. Finally, the `TimeAllocator` iterates through the `fixedTaskList` to find empty time slots in a day, which is then allocated to the `Task`s in the sorted `flexibleTaskList`.

The following sequence diagram illustrates how the allocate operation works:

Work in progress

Step 4. The user executes `list` to get an overview of the week. The user sees all 5 tasks in the printed timetable.

Data storage

The storage mechanism is facilitated by `Storage`. It reads and writes the `Task`s in a `TaskList` to `data.csv`, a csv file located next to the application jar file. It implements the following operations:

- `Storage#saveTaskListData` - Writes the current tasks into the save file.
- `Storage#loadTaskListData` - Loads the tasks in the save file into the application.

These operations are called by the `LogicManager`.

Given below is an example usage scenario and how the storage mechanism behaves at each step.

Step 1. The user launches the application for the first time. The `TaskList` is initialized to be empty. At this time, there is no *data.csv* file present. So, when `Storage` calls `Storage#loadTaskListData`, this is detected and an empty *data.csv* file is created next to the jar file. Since there was no save file, the `TaskList` remains empty.

Step 2. The user adds a task to the application, by executing `add n/Assignment1 t/1100 D/16-09-2020 d/2 r/Today i/high a/Refer to lecture notes`. The `TaskList` now contains 1 task (Assignment 1). After the command is executed, `LogicManager` calls `Storage#saveTaskListData` to automatically save the tasks in the `TaskList` into the save file.

Step 3. The user closes the application. Nothing happens since the data in the `TaskList` is already saved.

Step 4. The user launches the application again. The `TaskList` is initialized to be empty. `Storage#loadTaskListData` will read from `data.csv` and add the tasks inside the file into the empty `TaskList`. The `TaskList` now contains the task added earlier (Assignment 1) in **step 3**.

Step 5. The user executes `list` to get an overview of the week. The user sees *Assignment 1* in the printed timetable.

Appendix: Requirements

Product scope

Target user profile

- is a university student
- has a need to manage a significant number of tasks
- can type fast and prefers typing to mouse interactions
- is comfortable using the command line interface

Value proposition

- ATHENA helps students to automate the process of organising their schedule. After the user inputs pre-allocated time slots for work and relaxation, ATHENA figures out the best theoretical timetable based on the user's needs.
- It can be updated anytime during the week.
- ATHENA helps to reduce the amount of time and effort that users need to spend planning their time by finding free spaces to slot tasks in, with the goal of reducing dead space in the user's timetable.
- The planner will also make sure the user has enough time to eat, exercise and sleep. The user can set up ATHENA to follow a fixed weekly routine, and only needs to update a task list. ATHENA will then plan the timetable based on the importance and deadlines of the tasks in the list, making sure that the user is able to finish everything on time.

User Stories

Version	As a ...	I want to ...	So that I ...
`v1.0`	forgetful student	upload my tasks for the week	remember to do them
`v1.0`	student	mark my tasks as done	know that I have done them and can put them aside
`v1.0`	student	get reminded to do the tasks that are due soon	will be on time
`v1.0`	student	edit the tasks I added	update accordingly to small changes
`v1.0`	student	delete the tasks I added	remove tasks that are not needed to do anymore
`v1.0`	student	set my task according to importance	complete the more important tasks first

Version	As a ...	I want to ...	So that I ...
`v1.0`	student	leave some notes for a task	remember about it
`v2.0`	student	have a planner that tells me what time to rest	don't exhaust myself
`v2.0`	student	see an overview of the week ahead	make sure that I am staying on top of my tasks
`v2.0`	busy student	know what tasks to work on next	don't need to spend time planning

Non-Functional Requirements

1. Should work on any *mainstream OS* as long as it has Java `11` installed.
2. A user with above average typing speed for regular English text should be able to use the features of ATHENA faster using commands than using the mouse.

Glossary

- **Mainstream OS**: Windows, Linux, Unix, OS-X

Instructions for manual testing

Given below are instructions to test the app manually.

Launch and shutdown

1. Initial launch
    1. Download the jar file and copy into an empty folder
    2. Double-click the jar file Expected: Shows the command line interface with welcome message.
2. Shutdown ATHENA
    1. Test case: `exit`
       Expected: A farewell message by ATHENA will be shown.

Adding a task

Adding a task to the list.

1. Test case: `add n/Assignment1 t/1100 D/16-09-2020 d/2 r/Today i/high a/Refer to lecture notes`
   Expected: First task is added to the list. Details of the added task is shown.
2. Test case: `add t/1100 D/16-09-2020`
   Expected: No task is added. Error details is shown.

Deleting a task

Deleting a task while all tasks are shown.

1. Prerequisites: List all tasks using the `list` command.
2. Test case: `delete 1`
   Expected: Task with index 1 is deleted from the list. Details of the deleted task is shown.
3. Test case: `delete -1`
   Expected: No task is deleted. Error details is shown.
4. Other incorrect delete commands to try: `delete`, `delete x` (where x is larger than the list size)
   Expected: Similar to previous.

Marking a task as done

Marking a task as done while all tasks are shown.

1. Prerequisites: List all tasks using the `list` command.
2. Test case: `done 1`
   Expected: Task with index 1 is marked as done in the list. Details of the task is shown.
3. Test case: `done -1`
   Expected: No task is marked as done. Error details is shown.
4. Other incorrect delete commands to try: `done`, `done x` (where x is larger than the list size)
   Expected: Similar to previous.

Viewing the full details of a task

Viewing a task details while all tasks are shown.

1. Prerequisites: List all tasks using the `list` command.
2. Test case: `view 1`
   Expected: Details of the task with index 1 in the list is shown.
3. Test case: `view -1`
   Expected: No task details is shown. Error details is shown.
4. Other incorrect delete commands to try: `view`, `view x` (where x is larger than the list size)
   Expected: Similar to previous.

Editing a task

Editing a task details while all tasks are shown.

1. Prerequisites: List all tasks using the `list` command.
2. Test case: `edit 1 n/new name`
   Expected: Name of the task with index 1 in the list will be changed to `new name`.

3. Test case: `edit -1`
   Expected: No task will be edited. Error details is shown.

4. Other incorrect delete commands to try: `edit`, `edit x` (where x is larger than the list size)
   Expected: Similar to previous.

Listing all tasks

Listing all the tasks with or without filters.

1. Test case: `list`
   Expected: All the tasks will be listed.

2. Test case: `list i/HIGH f/TODAY`
   Expected: All the tasks today with high importance will be shown.

Help

Guide on the use of ATHENA.

1. Test case: `help`
   Expected: A guide on how to use ATHENA will be shown.

Data storage

Storage of user data (e.g. tasks).

1. Dealing with corrupted data files

    1. Open data.csv located next to Athena.jar.

    2. Test case (If the file is not empty): Add `,aaaa` at the end of the first line.

       Expected: The task on that line is corrupted. When you launch Athena again, it will fail to start, while providing an error message to the user.

    3. Test case (If the file is not empty): Remove a comma (`,`) from the file.

       Expected: The task on that line is corrupted. When you launch Athena again, it will fail to start, while providing an error message to the user.

    4. Test case: Add `aaaaa` at the end of the file.

       Expected: An invalid task is added. When you launch Athena again, it will fail to start, while providing an error message to the user.