

Le Duc - Developer Guide

1. Target User	1
2. User Stories	1
3. Use Cases	2
4. Non-functional requirements	3
5. Glossary	4
6. Design	4
6.1. Class Diagram	4
7. Implementation	5
7.1. Customization	5
7.2. Modify a Task	11
7.3. Sort the task list	17
7.4. Remind	18

1. Target User

- **Target:** Super busy student who has "multiple life".
- **Problem Scope:** Organize a students school tasks and non-academic tasks.
- **Value Proposition:** A student will be able to better manage their time, and be more productive.

2. User Stories

Target	User Stories	Priority
As a student,	I want to sort my task by type of task/by date/by description/by tag	* * *
As a student,	I want to prioritize different task	* * *
As a student,	I want to have recurrent task	* * *
As a student,	I want to display the task by day/week/month/year	* * *
As a student,	I want to edit a task	* * *
As a student,	I want to a better find feature (fuzzy matching)	* * *
As a student,	I want to postpone a task	* * *
As a student,	I want to have shortcut	* * *
As a student,	I want to display two or more different task list (for example professional and personal)	* * *

Target	User Stories	Priority
As a student,	I want the app to send me an email about my task	* *
As a student,	I want to display only the task that hasn't been done yet	* *
As a student,	I want the app to alarm me when a deadline is near	* *
As a student,	I want to import from other task list	* *
As a student,	I want to have subtask	* *
As a student,	I want to see which task was done late, which task hadn't been done and which task was done on time	* *
As a student,	I want to set a reminder	* *
As a student,	I want to see my progression	* *
As a student,	I want to add an excepted time	* *
As a student,	I want to see the time left until the allocated time for that task is over	* *
As a student,	I want to have different languages	* *
As a student,	I want to know the location, the address	* *
As a student,	I want to combine two tasks	* *
As a student,	I want to visualize the task (GUI)	*
As a student,	I want to have a login and a password or have a profile	*
As a student,	I want to customize the welcome message	*
As a student,	I want to have secret task	*
As a student,	I want to have shared task	*
As a student,	I want to delete a profile (like an admin)	*
As a student,	I want to have some statistics	*
As a student,	I want to see others students task	*
As a student,	I want to have a message feature	*

3. Use Cases

- **Edit Command:** (only in multi-step commands yet)
 - **System:** Le Duc
 - **Actor:** High school student

- **Use Case:** Modify task
 - User will type “edit”
 - Le Duc will list the entire task list
 - User type the index of the task
 - Le Duc asks which part will be modified if it is not a Todo task
 - User answer and modify
 - **Shortcut:**
 - **System:** Le Duc
 - **Actor:** High school student
 - **Use Case:** Create shortcut
 - User type “shortcut”
 - Le Duc will show the first command to be modified
 - User type the shortcut for that command
 - Le Duc will show the second command to be modified
 - User type the shortcut for that command
 - ...
 - Le Duc will show the k command to be modified
 - User type the shortcut for that command
 - Le Duc prompt an error, because there is a conflict between two shortcuts, and will ask to enter a new shortcut
 - User type another shortcut for that command
 - ...
 - Le Duc shows all the shortcuts
-
-

4. Non-functional requirements

- **Task list size requirement:** The user is a super busy students, so he will have a lot of task. Size of task list possibly infinite (use of ArrayList, depends on the computer and the storage doesn't use much as it is a written file).
 - **Quality requirement:** The system is easy to understand and to be handled by a new user.
 - **Mastery requirement:** The system is easy to be mastered, the typing of new task should be easy and fast.
 - **Disaster recovery requirement:** If the system crash, the user shall find all his tasks in the storage file.
-
-

5. Glossary

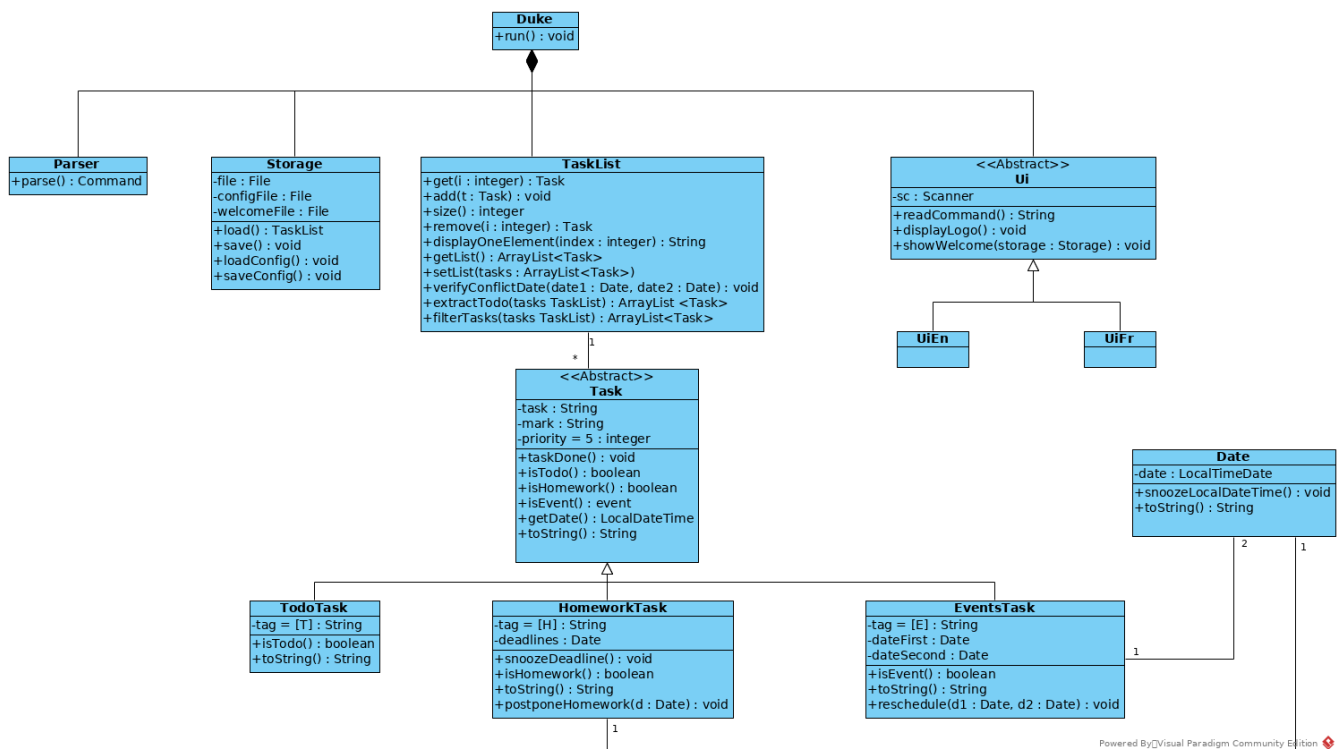
- **Fuzzy matching:** When searching for task descriptions via keyword, the "Sorensen-Dice" Fuzzy Matching algorithm is used to return top matches. This ensures that typos in the user query does not affect search performance
- **Recurrent task:** A task that repeat every day/week/month...

6. Design

6.1. Class Diagram

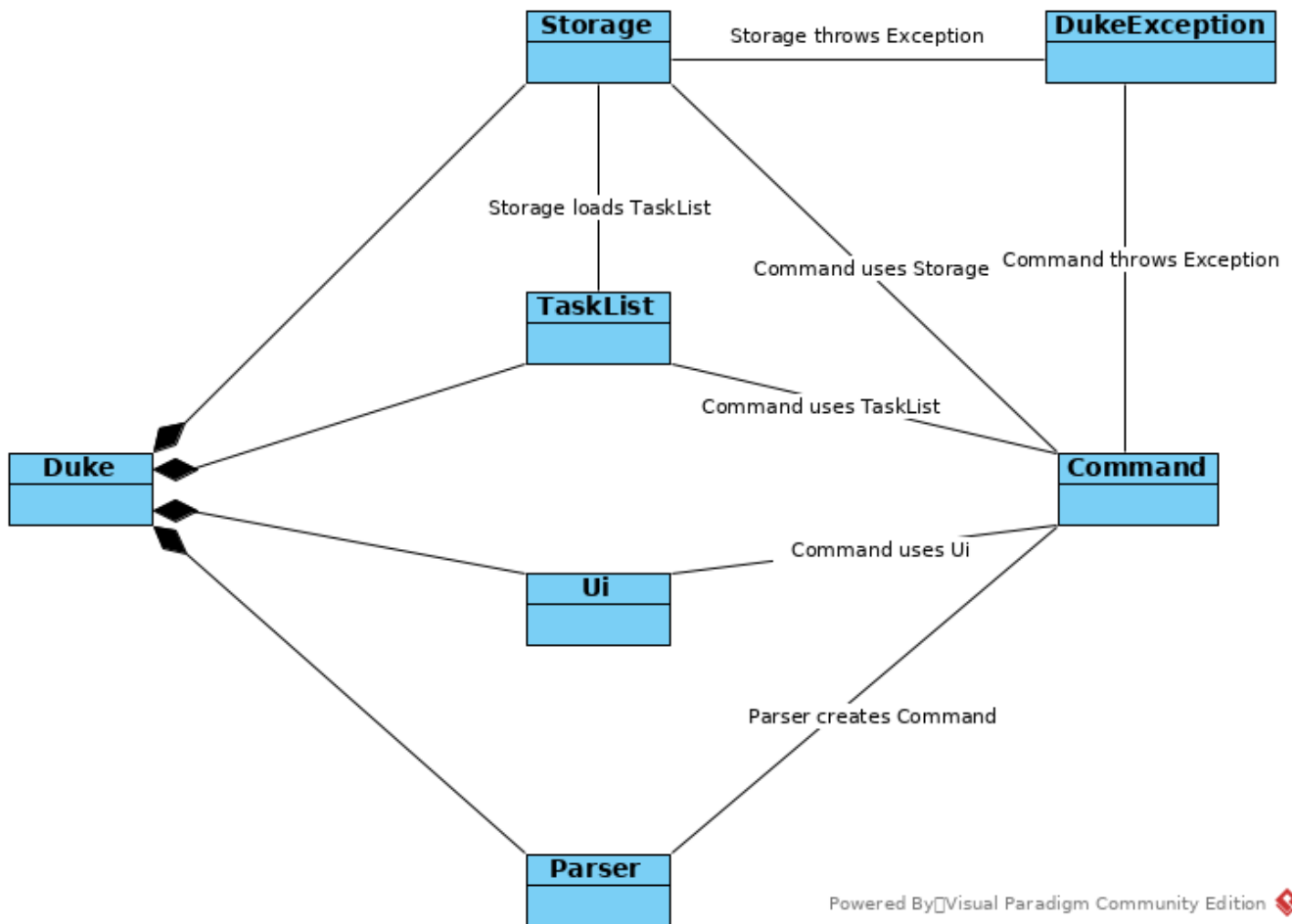
Le Duc main class, called **Duke**, is composed of 4 classes : **Storage**, **Ui**, **Parser**, **TaskList**.

- **Storage** deals with saving and loading files such as the file containing the config or the file containing all the tasks.
- **Ui** deals with the interaction between the user and the program.
- **Parser** given an user's input (through Ui), the Parser will return the corresponding command
- **TaskList** represents the list containing all the tasks.



In this class diagram, constructors, getters and setters were not all represented.

The following class diagram will represent the interaction between all classes without being too specific. The class diagram for the abstract class **Command** and his concrete class is not represented because it is only one abstract class connected to a multitude of concrete class. The same goes for the **DukeException** class.



7. Implementation

7.1. Customization

The user can customize Le Duc in the following ways :

- **shortcut**: The user can implement and use shortcut for every command.
- **language**: The user can change the language for Le Duc.

7.1.1. Shortcut

The shortcut mechanism is done by the **ShortcutCommand**. As every other command, it extends Command with a HashSet containing all the command's shortcut name and another HashSet containing all the default command's shortcut name. Others commands include now a static attribute named shortcut that correspond to the command's shortcut. It implements these following methods:

- **ShortcutCommand#setOneShortcut** — set the shortcut of one command
- **ShortcutCommand#initializedSetShortcut** — initialized the HashSet contains all the default command's shortcut name

There are three cases:

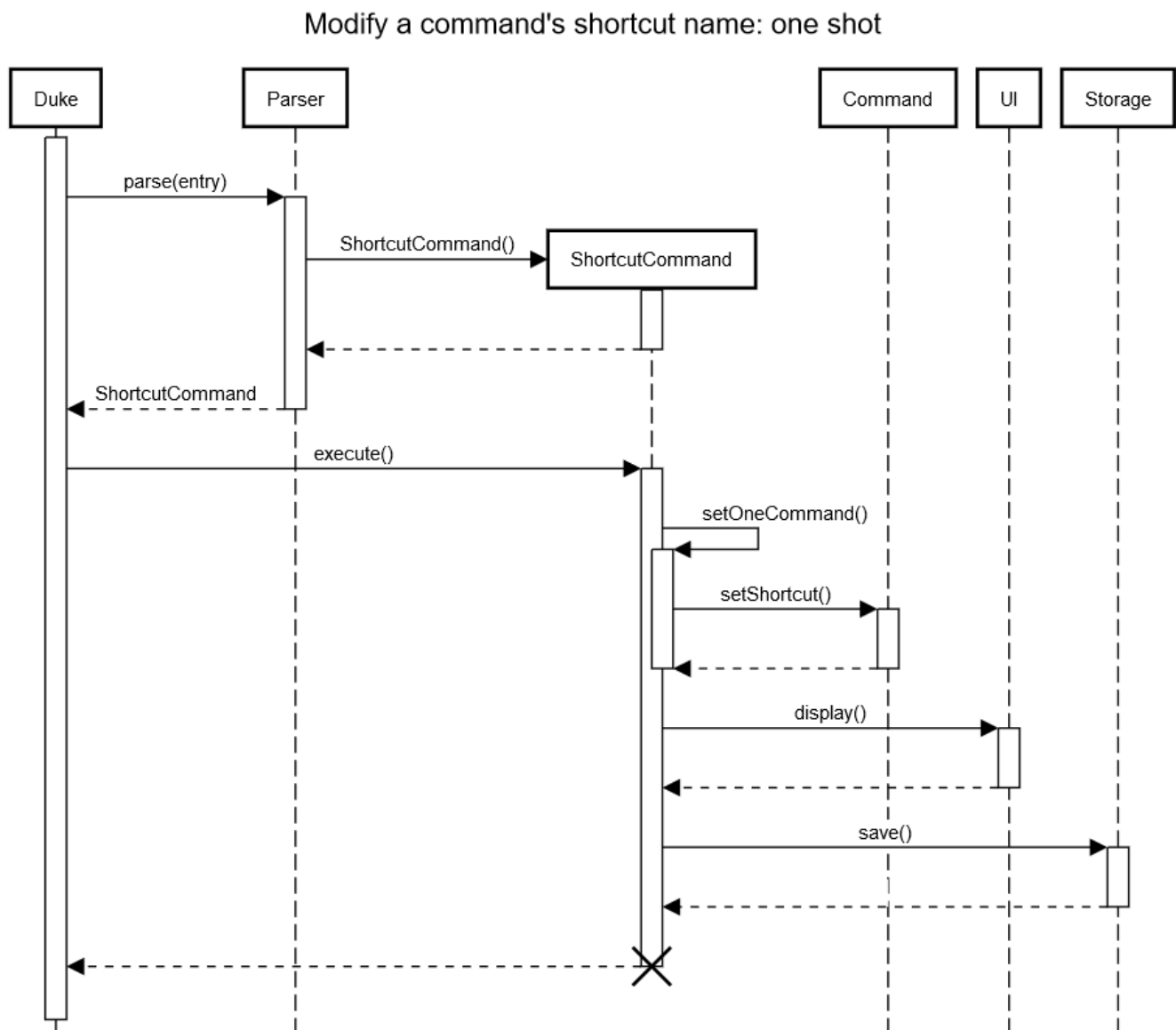
- one shot one command: The user write the command for the shortcut in one line
- multi step one command: The user write which command he wants to add a shortcut to, then the console ask what is the shortcut, and the user write the name of the shortcut
- multi step every command: The user asks the console that he wants to modify all the command, and the console will show one by one every command, and the user will modify one by one each command.

When the user launches the application, the program will read the config file, then set all shortcuts to previous shortcuts that the user has decided. If the user has not decided to customized shortcuts, it will be the default shortcut.

These following diagram show how the 3 cases were implemented:

One shot one command

The user type the "entry" (not shown in the sequence diagram) as `shortcut CommandName ShortcutName`.

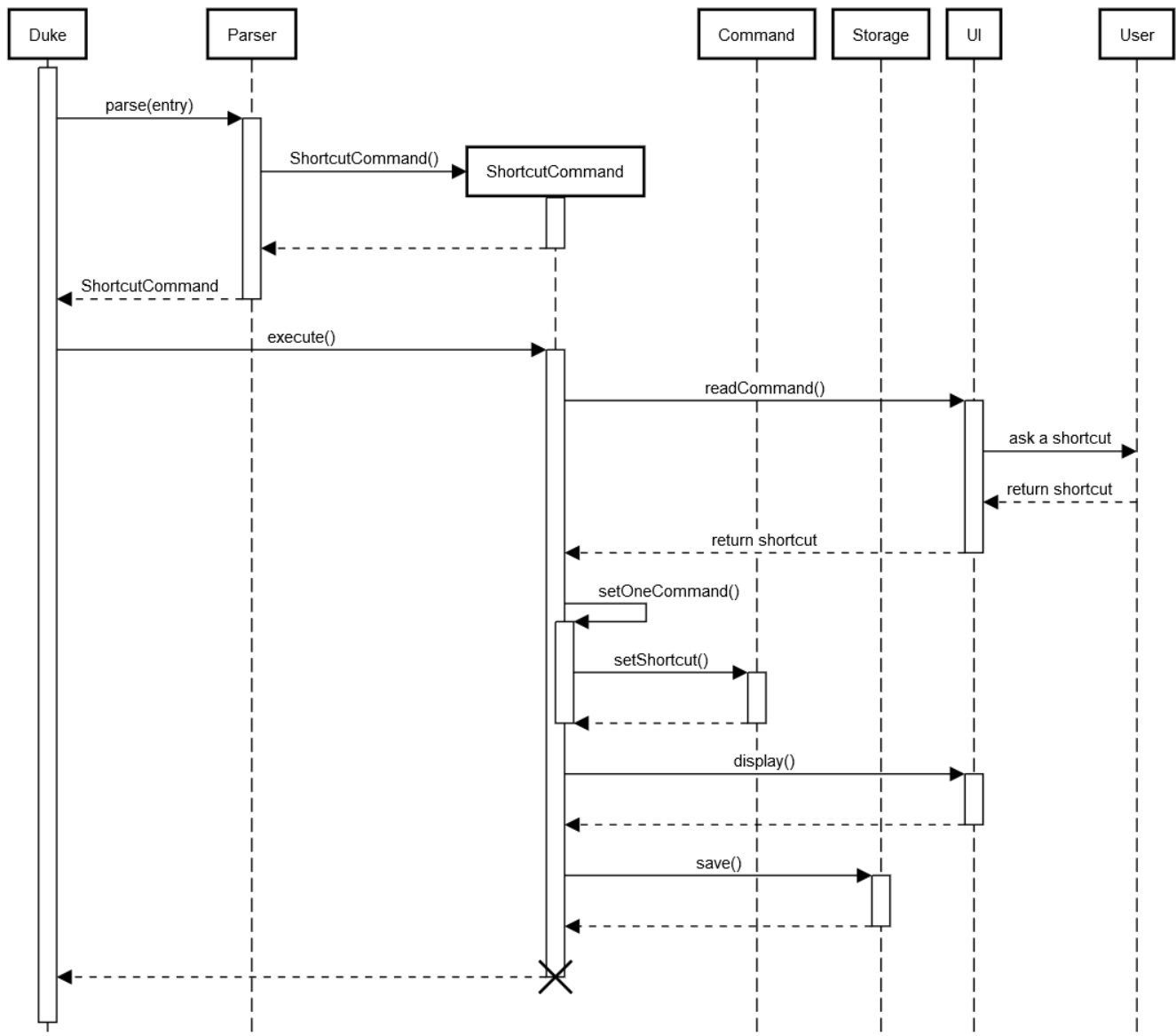


The method `setShortcut` is static, thus an object `Command` won't be created

Multi-step one command

The user type the "entry" (not shown in the sequence diagram) as `shortcut CommandName`. Then the console will ask what will be the new name for the shortcut.

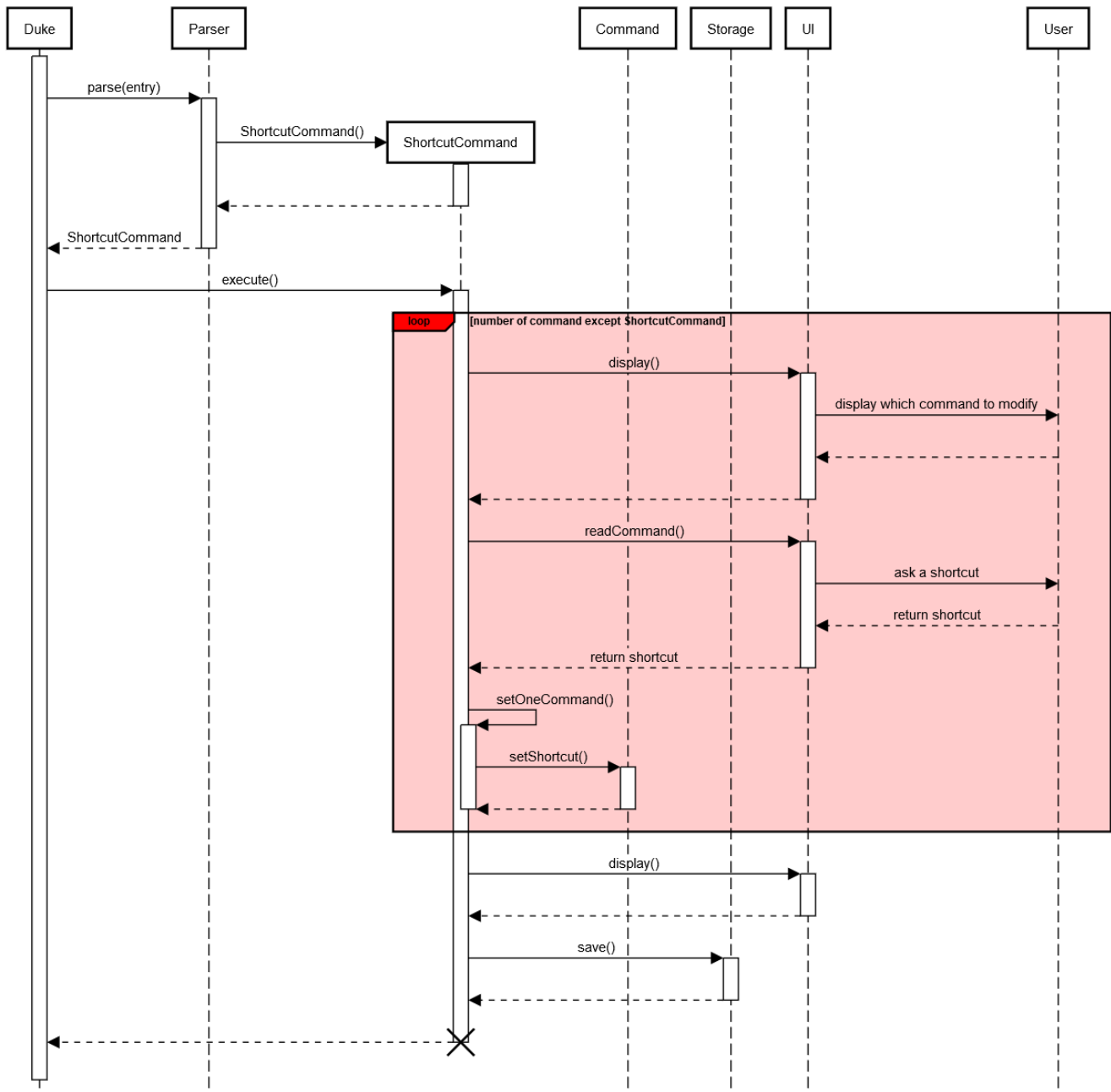
Modify a command's shortcut name: one command



Multi-step every command

The user type the "entry" (not shown in the sequence diagram) as `shortcut`. The console will display one command's name, then the console will ask what will be the new name for the shortcut. The console will repeat until every command have a shortcut.

Modify a command's shortcut name: every command



Consideration

- The config file that contains all the name for the shortcut can be edit by hand, because it is faster to edit the config file than doing it via the application.
- When a command's shortcut is set, the default shortcut can still be used

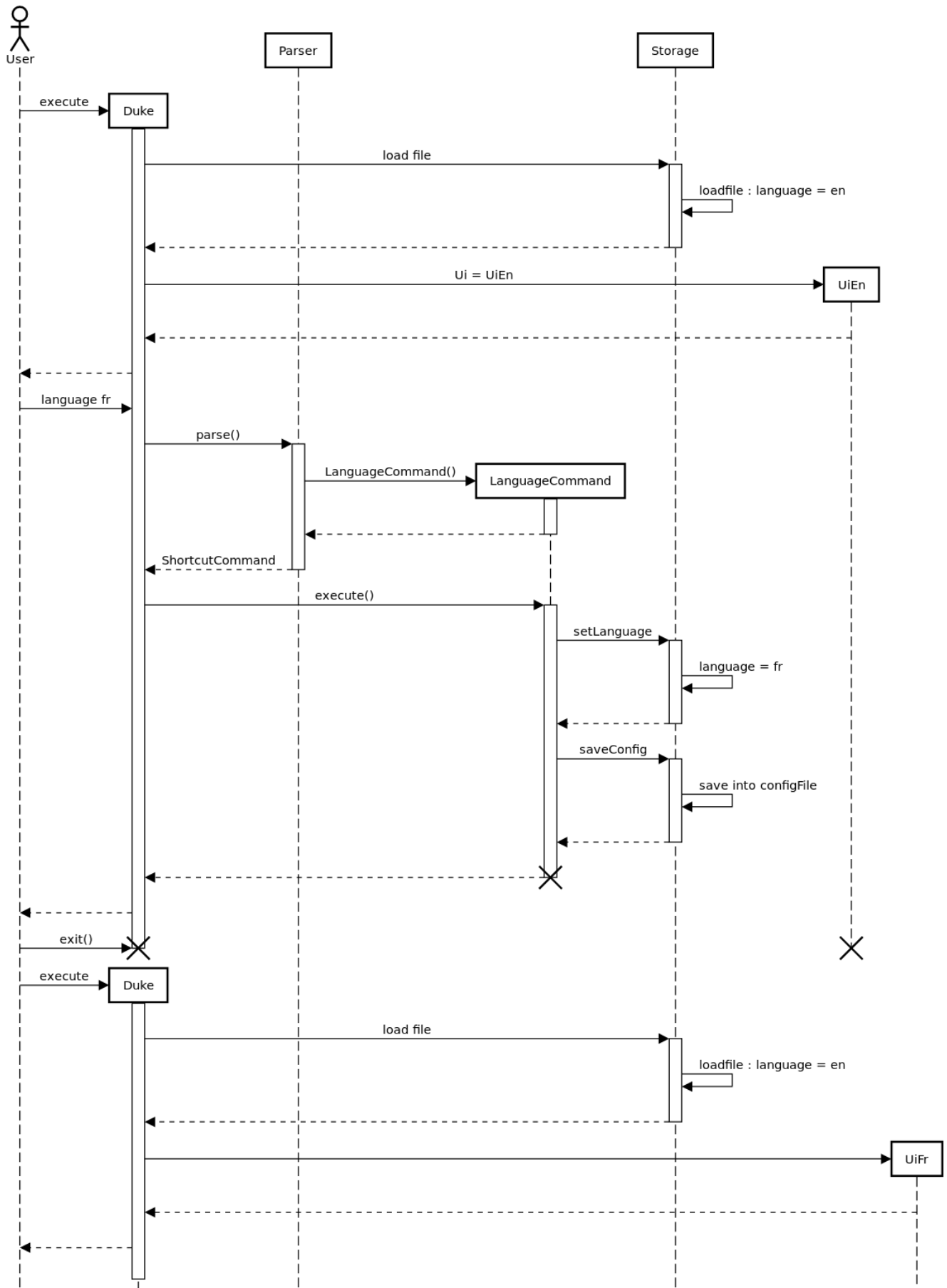
7.1.2. Language

Changing the language mechanism is done by the `LanguageCommand`. For the moment two languages are available : french and english. Only the return message after a command and the error message are changed. After typing the command to change the language, the language is changed at the next execution of the program.

The following are the steps to change a language :

- The user open Le Duc (the program).
- Le Duc create the object `ui` as an instance of `UiEn`.
- The user type `language fr` (the program is previously in english)
- The program will change the config file.
- The user exit the program.
- The user reopen Le Duc.
- Le Duc load the config file with the new language.
- Le Duc create the object `ui` as an instance of `UiFr`.
- The language of Le Duc is french.

Modify a language for Le Duc



In the sequence diagram, **Parser** and **Storage** should be created and destroyed when Duke is created or destroyed, but for more clarity, it was not represented.

Consideration

- (Current implementation) Each message displayed to the user (error or a message returned by a command) correspond to an abstract method in `Ui` and an override method in `UiFr` and `UiEn`. It was done so because it is easier to add a new language because it is sufficient to create a new class and override the method.
- (Alternative) Make an if statement for each new language and a static attribute in `Ui`. There are less methods and less classes but if a new language is added in the future, every single command and every single exception have to be edited.

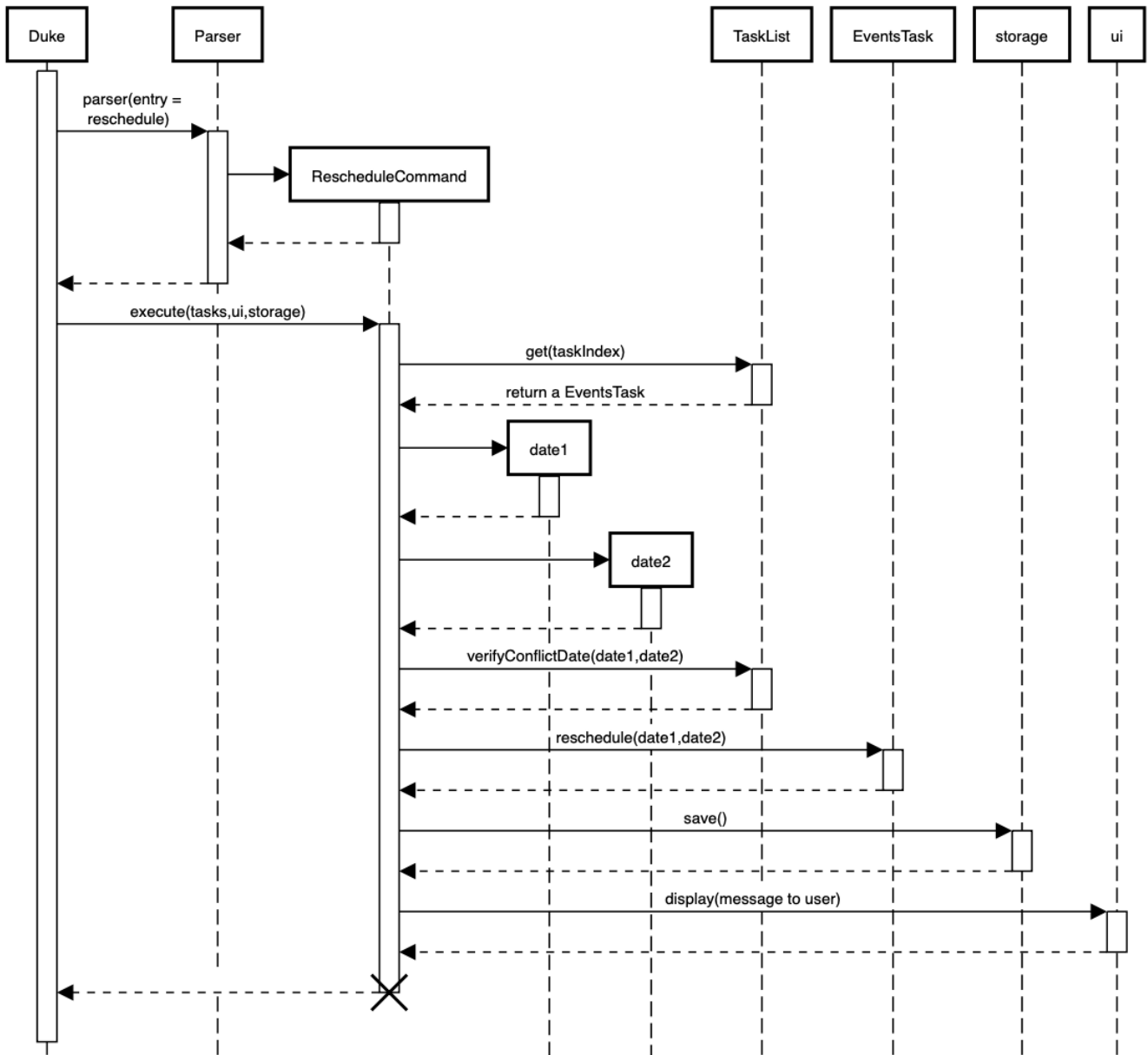
7.2. Modify a Task

Several commands allow the user to modify a task: `reschedule`, `postpone`, `snooze`, `edit` and `prioritize`. As every other command, these commands extend `Command`. As these commands relate to the modification of tasks, each command needs to write into the data file after its execution.

7.2.1. Reschedule an event task

When rescheduling an event, two dates can't clash. This verification is done with the `verifyConflictDate` method which is in the `TaskList` class. Indeed, all task dates are needed to verify if there is a conflict. So, this allows to improve the cohesion.

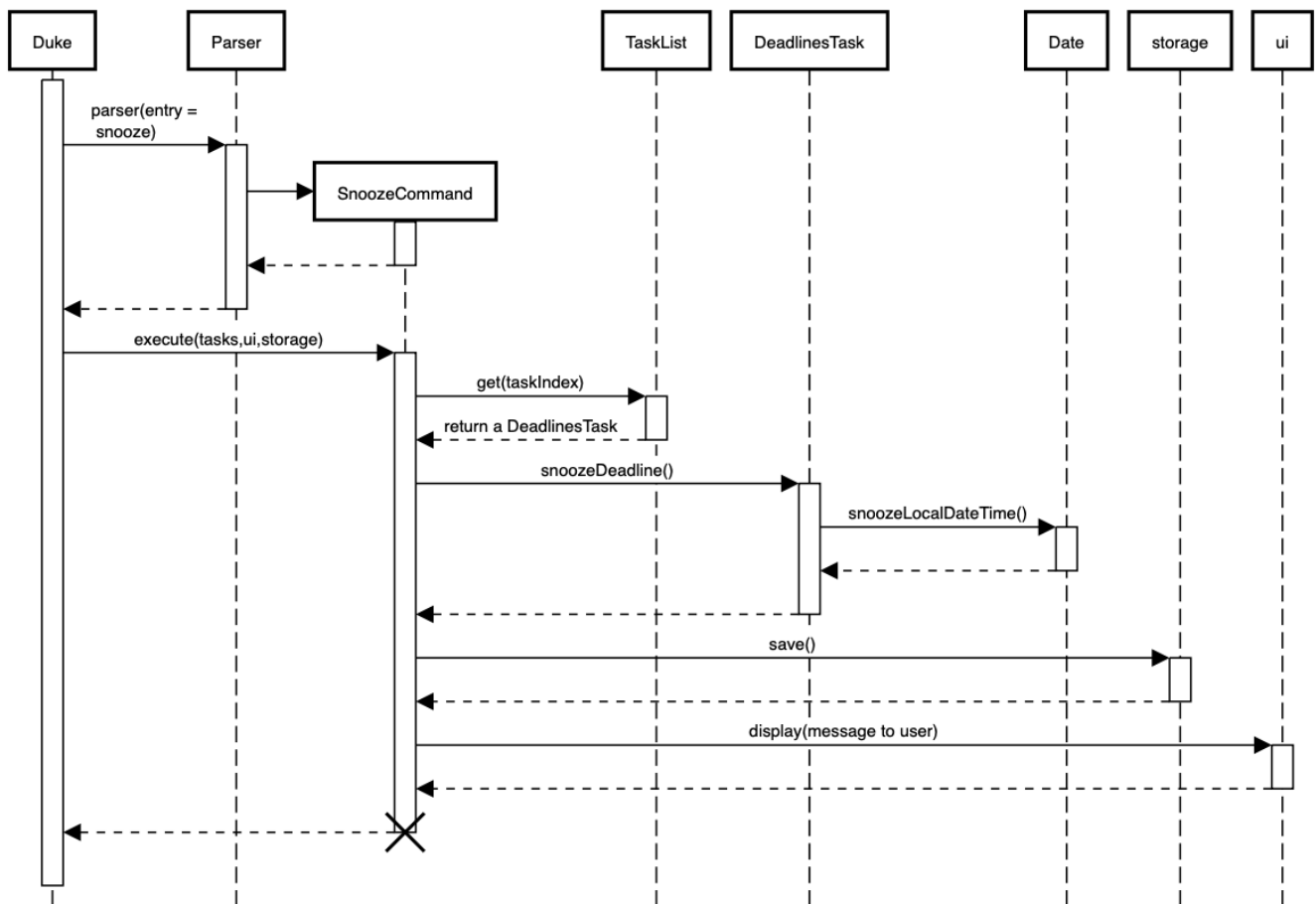
Sequence Diagram : reschedule



7.2.2. Snooze an homework task

Snooze is applicable to a homework task. The snooze time is fixed at 30 minutes(it could be easily changed in the snoozeLocalDateTime() method of Date.

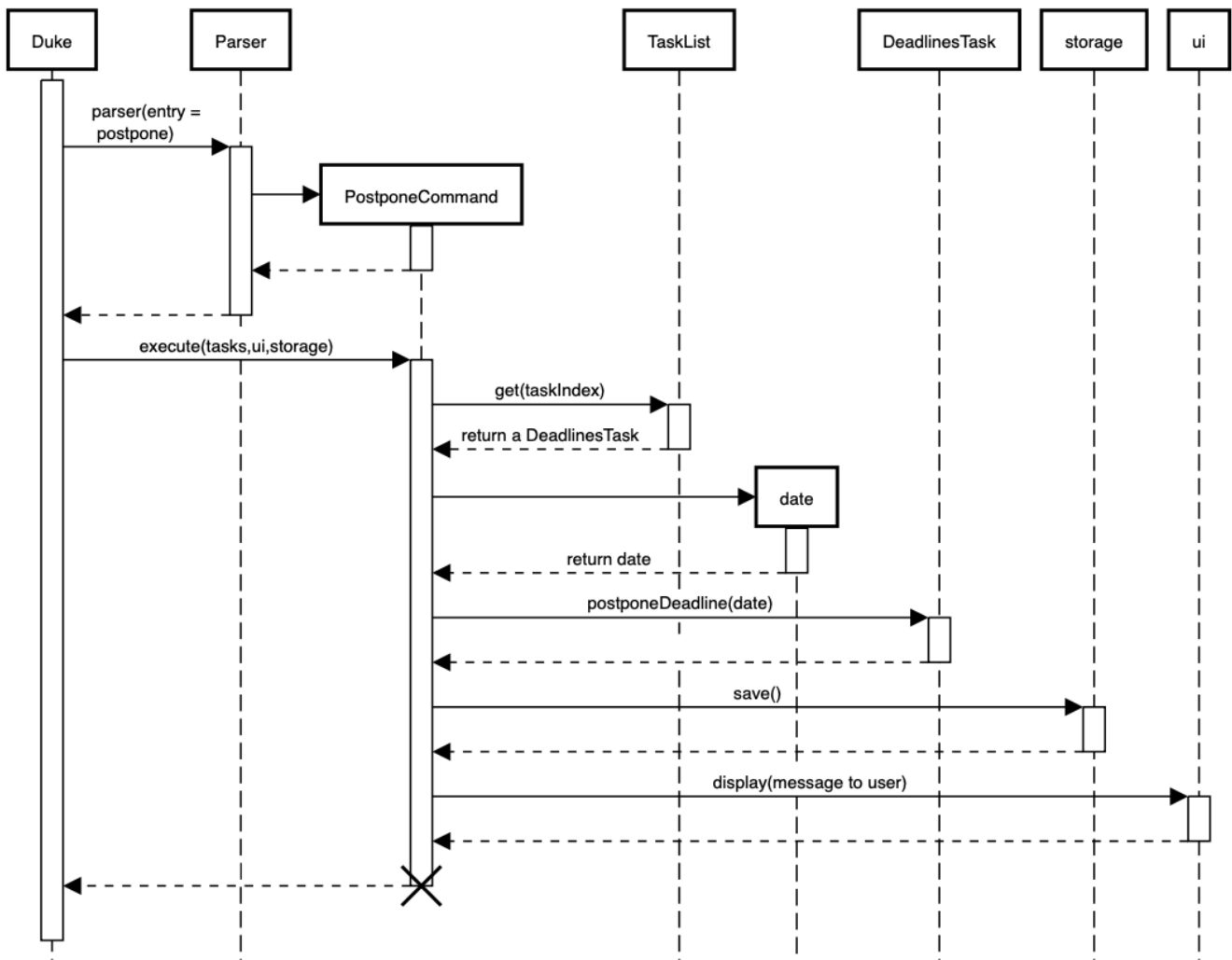
Sequence Diagram : Snooze



7.2.3. Postpone an homework task

Postpone is also only applicable to a homework task. The new date should be after the old one. This is verified inside the execution of the postponeCommand.

Sequence Diagram : Postpone

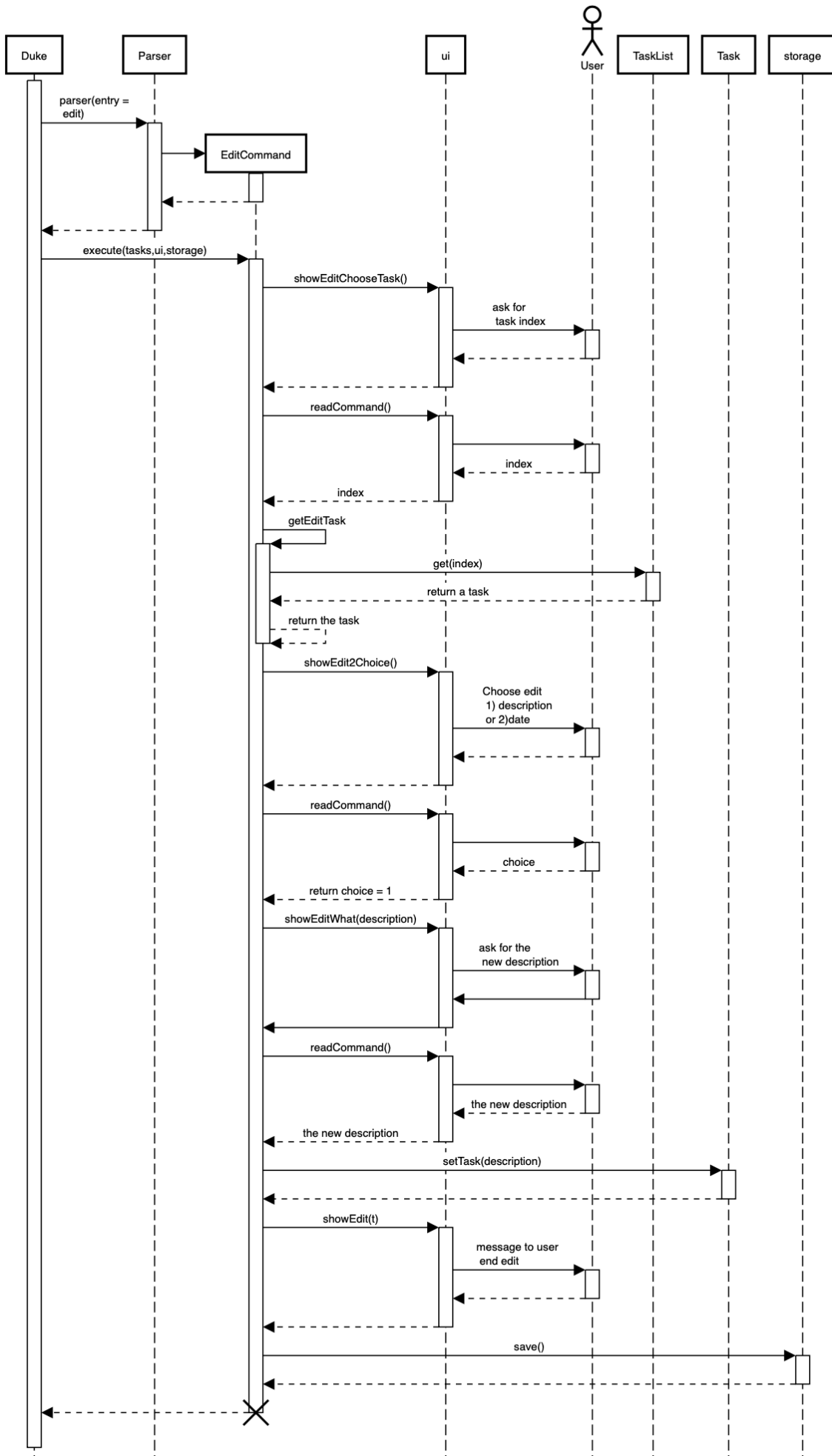


7.2.4. Edit a task

- Multi-steps command: to edit a task, the user has to follow these instructions:
 1. **edit**
 2. All of the tasks will be displayed, you have to choose a task INDEX
 3. Depending on the type of task:
 - If it is a todo task, you have to enter the new DESCRIPTION
 - If it is not a todo task, you have to choose 1) if you want to edit the description or 2) if you want to edit the date
 - Then, enter the new DESCRIPTION or the new DATE of the task

The sequence diagram shows the interactions between different classes when the user want to edit the description an homework or event task with a multi-steps edit command.

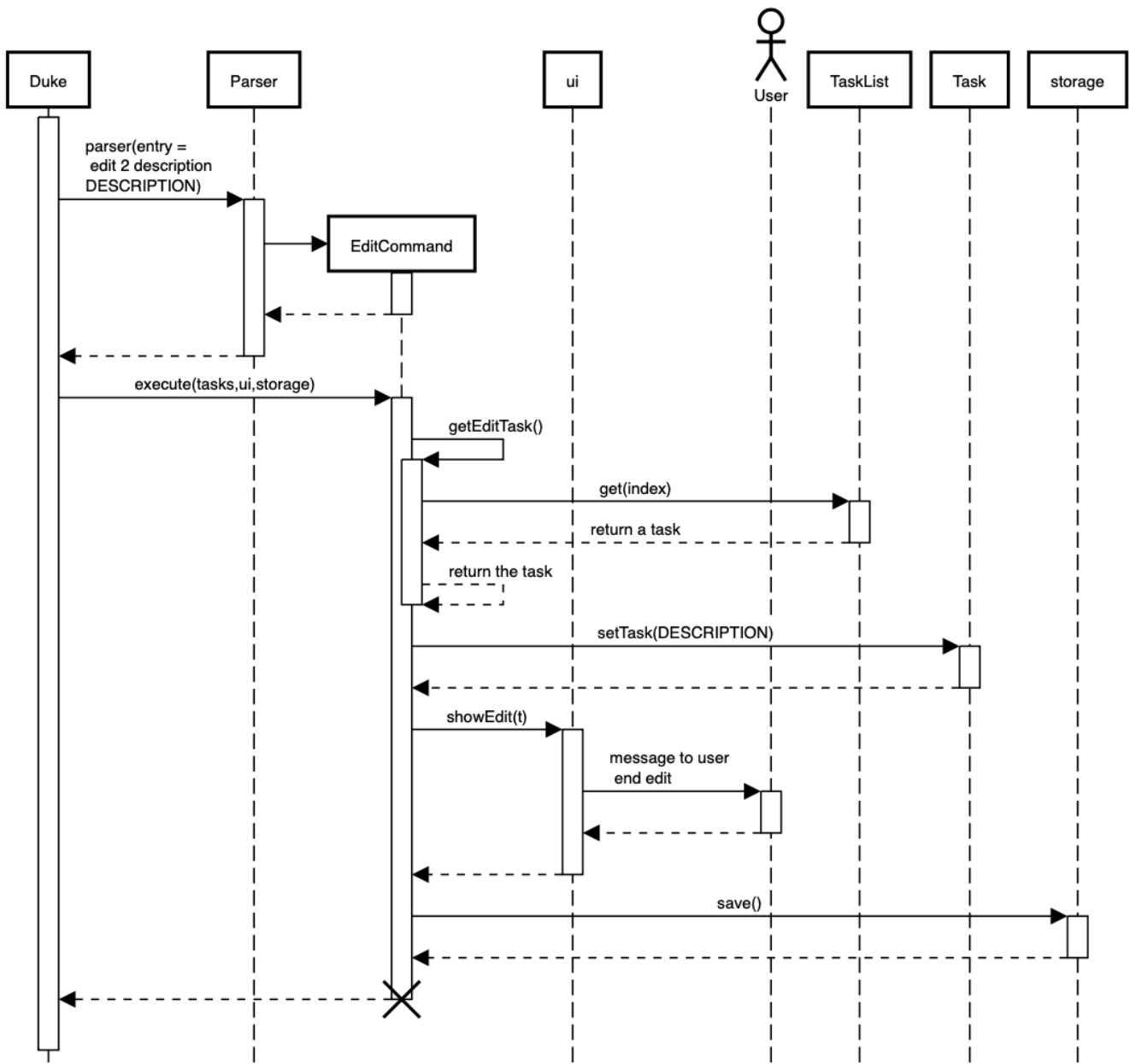
Sequence Diagram : Edit multi-steps



- For one shot command:
 - edit the description: `edit INDEX description DESCRIPTION`
 - edit the date of an homework task: `edit INDEX /by DATE`
 - edit the period of an event task: `edit INDEX /at DATE - DATE`

The sequence diagram shows the interactions between different classes when the user input `edit 2 description DESCRIPTION`.

Sequence Diagram : Edit one shot

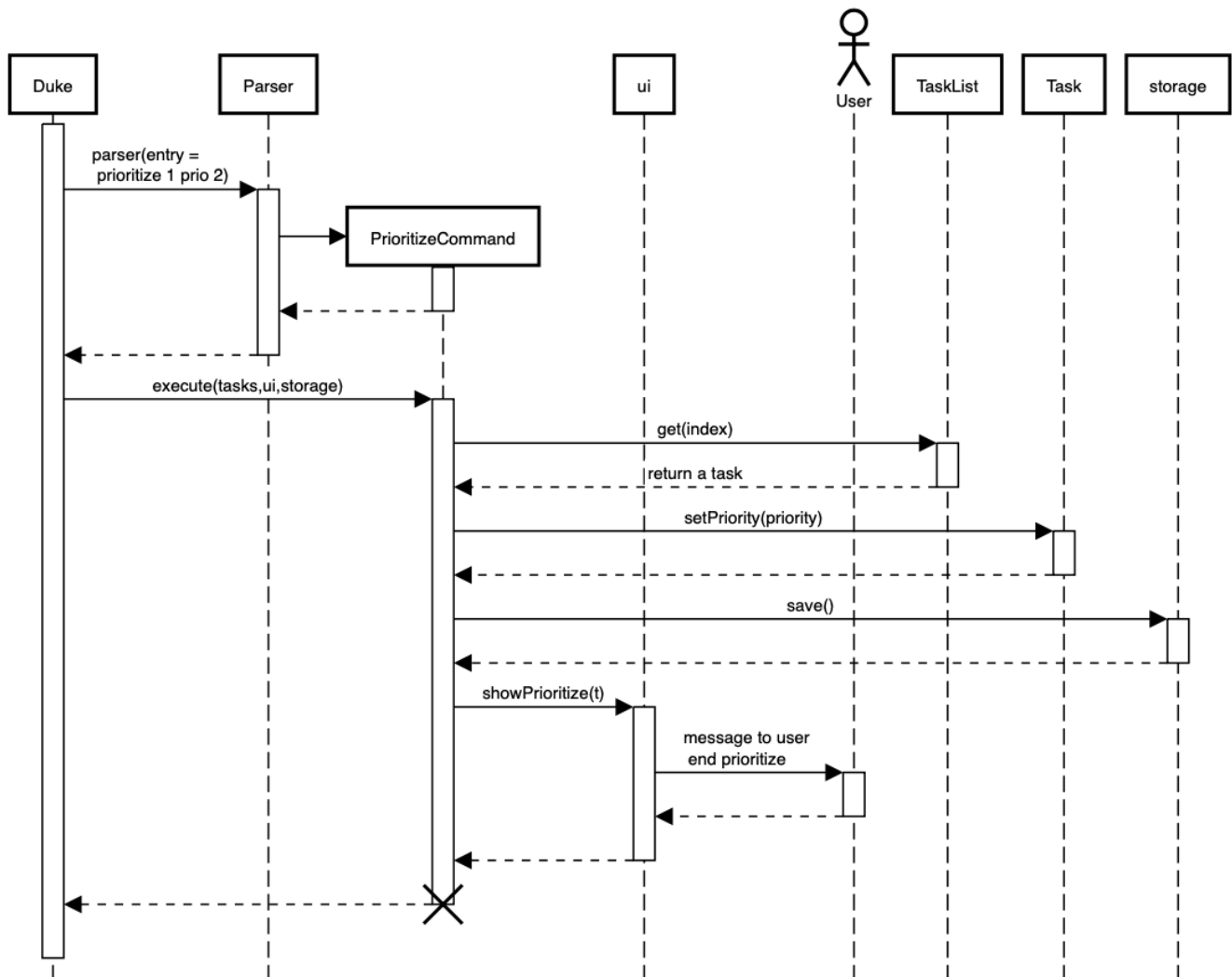


7.2.5. Prioritize a task

A task has initially a priority of 5. The priority of a task goes from 0 to 9. This command allows the user to change the priority of a task.

The sequence diagram show the interactions between different classes when the user wants to change to priority of the first task to 2.

Sequence Diagram : Prioritize



7.2.6. Consideration

There are two different commands for modifying the priority (**prioritize**) and the description/date (**edit**) of a task. Indeed, the edit command is considered to be used when a user has initially created an incorrect task, whereas the prioritize command is supposed to be used regularly as the priority of a task generally increases with time. However, these two commands are obviously easy to combine into one command.

7.3. Sort the task list

Sort all task by date/description/priority/type of task/ done or not: **sort SORTTYPE** SORTTYPE is either date, description, priority, type, done

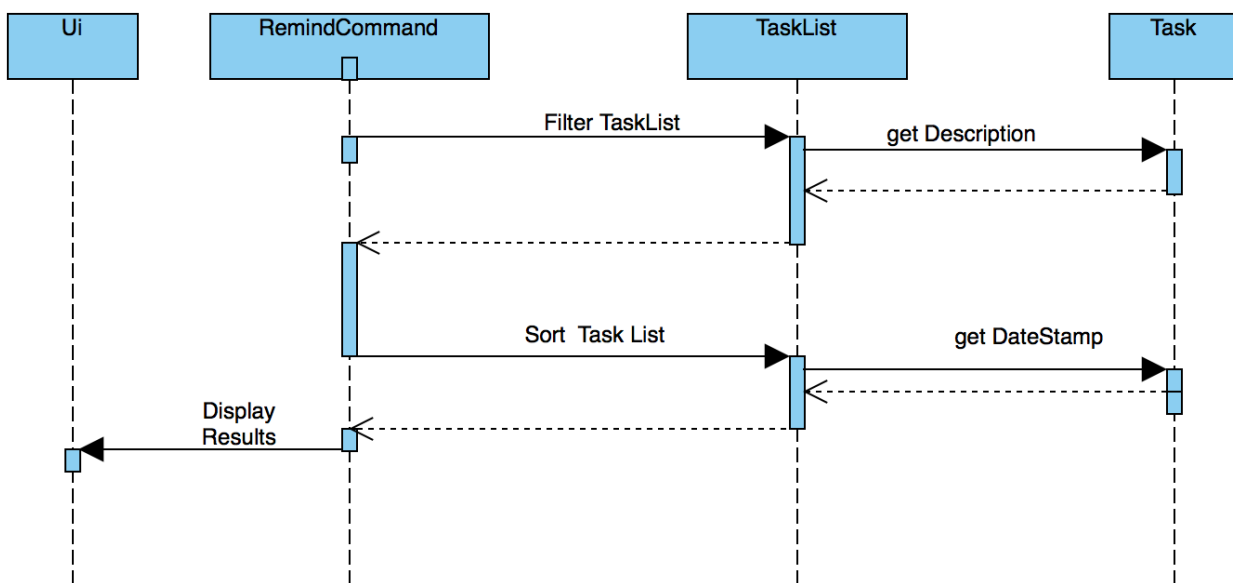
- Sorting by date will sort tasks in chronological order
- Sorting by description will sort the descriptions in alphabetical order
- Sorting by priority will sort tasks in ascending urgency
- Sorting by type will sort tasks depending on its task type (event, homework, todo)
- Sorting by done will sort tasks depending on if the task is done or not.

To implement the sort command, the comparing static method of Comparator interface introduced in Java 8 is used. So, here the sort key are the description or the priority of the task.

7.4. Remind

The Remind feature is done by the RemindCommand. Along with all of the other implemented commands, it extends Command. The feature will process each tasks date/timestamps to order them, and then remind the user of the top 3 upcoming tasks. The following methods were implemented in this feature:

- **filterTasks** - Extracts the Homework and Event tasks into a separate ArrayList
- **sort** - Orders the filtered TaskList in chronological order.
- **Sequence Diagram of the Remind Feature:**



There are 4 cases:

- TaskList contains only Homework and Event objects
- TaskList contains only Todo objects
- TaskList contains a mix of all objects
- TaskList contains no objects

The task of sorting the tasklist in chronological order becomes challenging as not all tasks have associated timestamps. This problem is divided into 4 subproblems, each of which are handled separately.

7.4.1. TaskList contains only Homework/Event tasks

- The original TaskList is passed through a filter to isolate the Homework and event tasks.
- The TaskList.extractTodo() method will attempt to isolate all todo tasks. It will return an empty array in this case

- After filtering by TaskList.filterTasks(), the filtered TaskList will be equal to the original TaskList, as there are no Todo objects to filter out in this particular case.
- The filtered TaskList will then be sorted by TaskList.sort().
- The sort() method will call each tasks task.getDate() method in preparation for sorting. For "Event" tasks, the first timestamp will be used for the purposes of sorting.
- After performing insertion sort, the array of Todo's will be appended to the end of the sortedlist. In this case, there are no "todo" tasks, so nothing will be appended.
- The first 3 most upcoming tasks will be displayed to the user.
- **Output:**

```
remind
1. [H][X] d1 by: 14/09/2019 22:33 [Priority: 5]
2. [E][X] e1 at: 21/09/2019 00:00 - 28/10/2019 22:22 [Priority: 5]
3. [H][X] d2 by: 22/09/2019 22:33 [Priority: 5]
```

7.4.2. TaskList only contains Todo tasks

- The original TaskList is passed through a filter to isolate the Homework and Event tasks. Because there are only Todo tasks, it will return an empty array.
- The TaskList.extractTodo() method will attempt to isolate all todo tasks. It will return the original tasklist in this case, because every task is a "todo"
- The filtered TaskList will then be sorted by TaskList.sort(). The method will return an empty array, because the input array containing Homework and Event tasks is empty.
- The sort method will return an empty array, as the input array of Homework/Event tasks is empty in this case.
- The array of Todo's is appended to the empty, returned array from TaskList.sort(), resulting in the original tasklist of only Todo tasks.
- **The original tasklist is considered to be the sorted tasklist. The order in which the Todo's were created are treated as an "implicit order".**
- The first 3 most upcoming tasks will be displayed to the user.
- **Output:**

```
remind
1. [T][X] todo1 [Priority: 5]
2. [T][X] todo2 [Priority: 5]
3. [T][X] todo3 [Priority: 5]
```

7.4.3. TaskList contains a mix of all task types

- The original TaskList is passed through a filter to isolate the Homework and Event tasks.
- The TaskList.extractTodo() method will attempt to isolate all todo tasks. It will return an array of

all todo's in this case

- After filtering by `TaskList.filterTasks()`, the filtered `TaskList` will contain only Homework and Event tasks.
- The filtered `TaskList` will then be sorted by `TaskList.sort()`.
- The `sort()` method will call each task's `task.getDate()` method in preparation for sorting. For "Event" tasks, the first timestamp will be used for the purposes of sorting.
- After performing insertion sort, the array of `Todo`'s will be appended to the end of the sortedlist. If a task is a "todo" it is assumed that it has a "soft deadline" unlike a Homework task. Therefore, it is pushed to the bottom of the sorted tasklist.
- The first 3 most upcoming tasks will be displayed to the user.
- **Output:**

```
remind
1. [H][X] d1 by: 14/09/2019 22:33 [Priority: 5]
2. [E][X] e1 at: 21/09/2019 00:00 - 28/10/2019 22:22 [Priority: 5]
3. [T][X] t1 [Priority: 5]
```

7.4.4. TaskList Contains No Objects

There are no upcoming tasks in your list

7.4.5. Consideration

- Sorting the `TaskList` in place was considered, but it reduced cohesion of the design, as the same function is reused by other features.
- It was considered to only remind the user of tasks that are coming up in the next week, but that would limit its potential utility