El Duque - Developer Guide

By: `CS2113-T16-2` Since: `Aug 2019` Licence: `MIT`

1. Setting up

Refer to the guide here.

2. Design

2.1. Architecture

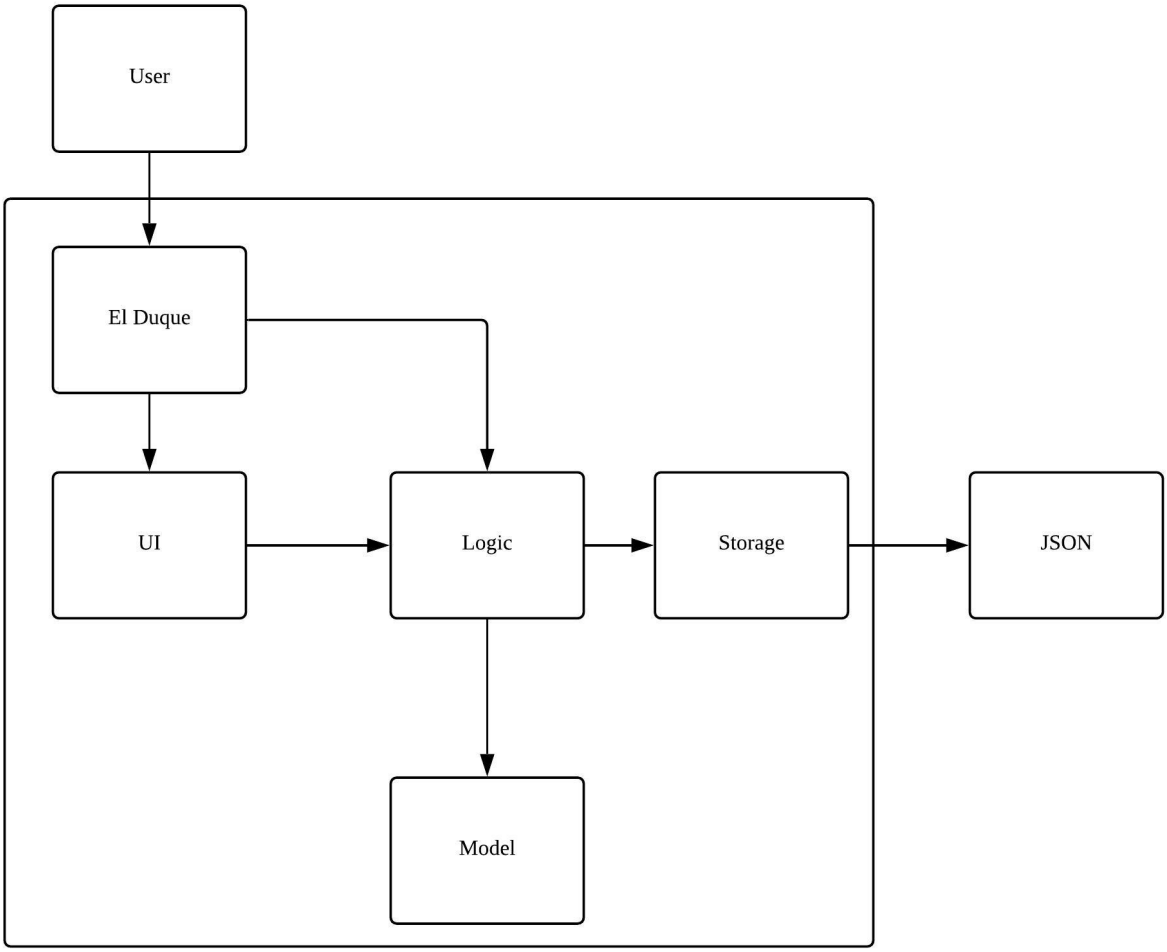

Figure 1. Architecture Diagram

The ***Architecture Diagram*** given above explains the high-level design of the El Duque.

The 5 main components of the app are:

- `El Duque`: Responsible for initialising all the other components correctly
- `UI`: UI of the app
- `Logic`: The command executor of the app
- `Model`: Stores the data of the app in-memory
- `Storage`: Reads and writes data to the JSON file

Interactions between components
The sequence diagram below shows how the different components interact with each other for the scenario where the user issues the “task sleep well” command.

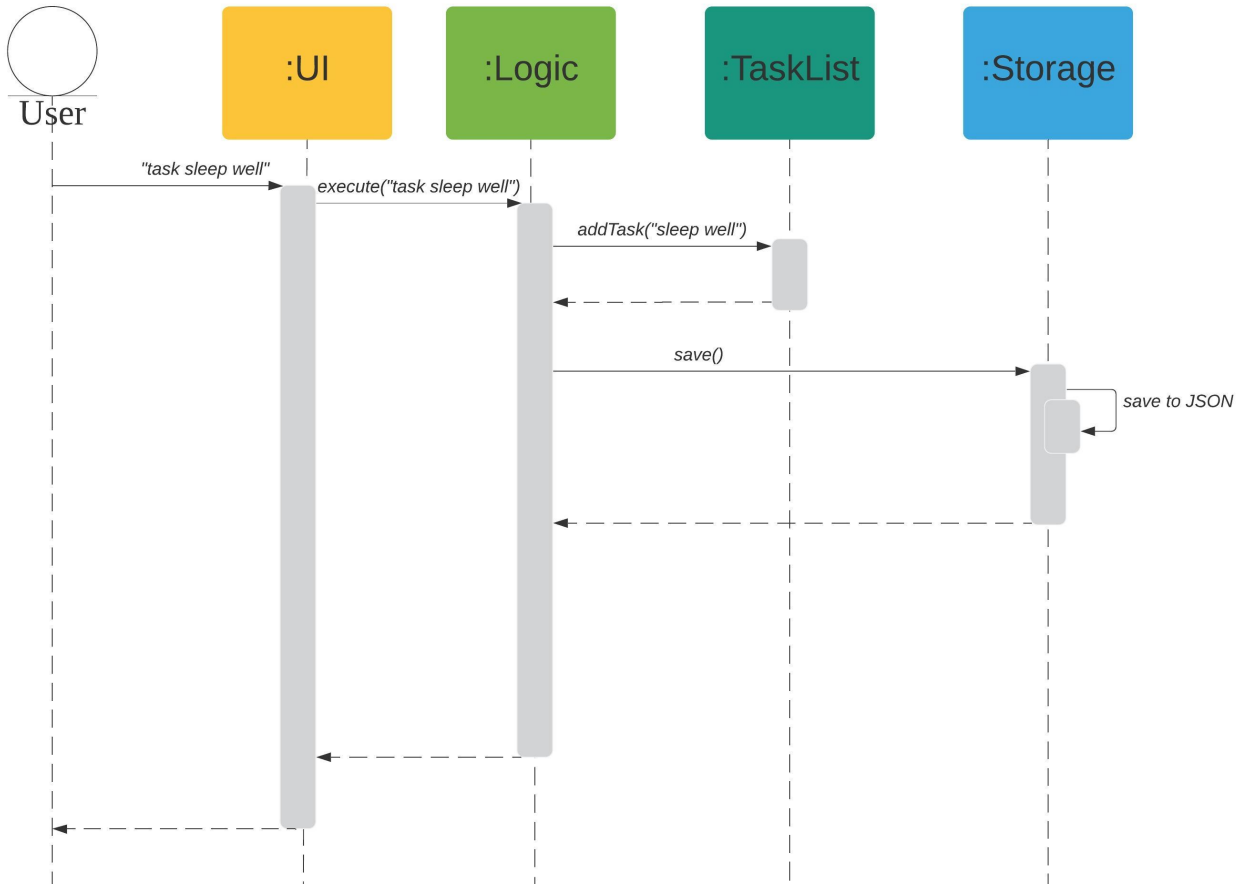

Figure 2. Component interactions for "task sleep well" command

The sections below give more details of each component.

2.2. UI component

[UiClassDiagram] | *UiClassDiagram.png*

Figure 3. Structure of the UI Component

The ui package is relatively simple, since we have no GUI. The ui package consists of 2 classes - `TaskListPrinter` and `Ui`. `TaskListPrinter` is a class that handles the printing of the `TaskList` for the user to view. `Ui` is a class that deals with user interactions. `TaskListPrinter` depends on `Task` and `TaskList` while the logic package relies heavily on `UI` (e.g. all commands have execute method which uses `Ui`).

2.3. Logic component

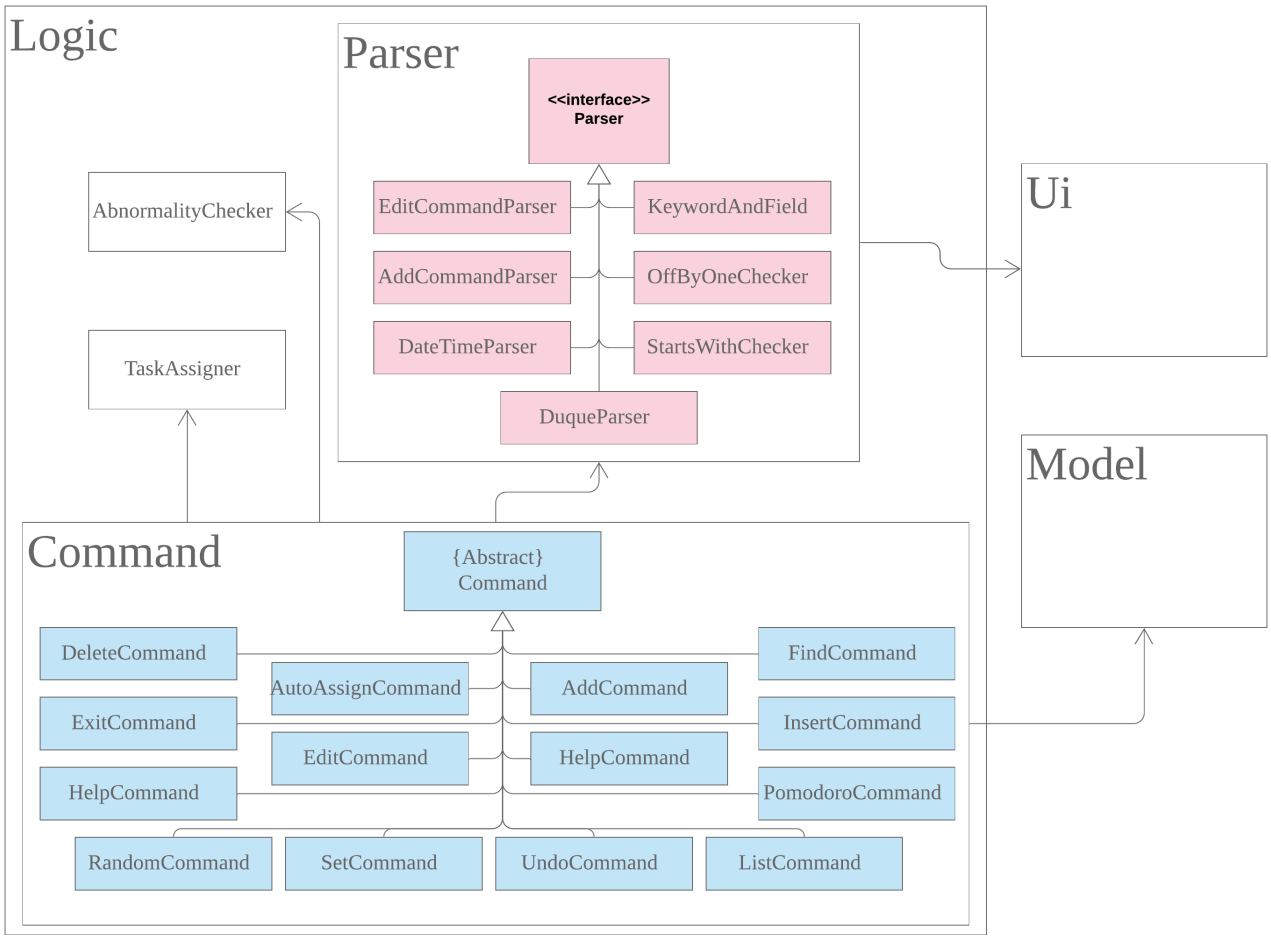

Figure 4. Structure of the Logic Component

1. `Logic` uses the `DuqueParser` to parse the user command.
2. Depending on the user command, a different parser will be created to handle the command.
3. This results in a `Command` object which is executed by `El Duque`.
4. The command execution can affect the `Model` (e.g. adding a `Task`).
5. The result of the command execution, which may affect the `Model`, will be passed back to the `Ui` to display confirmation of the successful execution to the user. === Model component

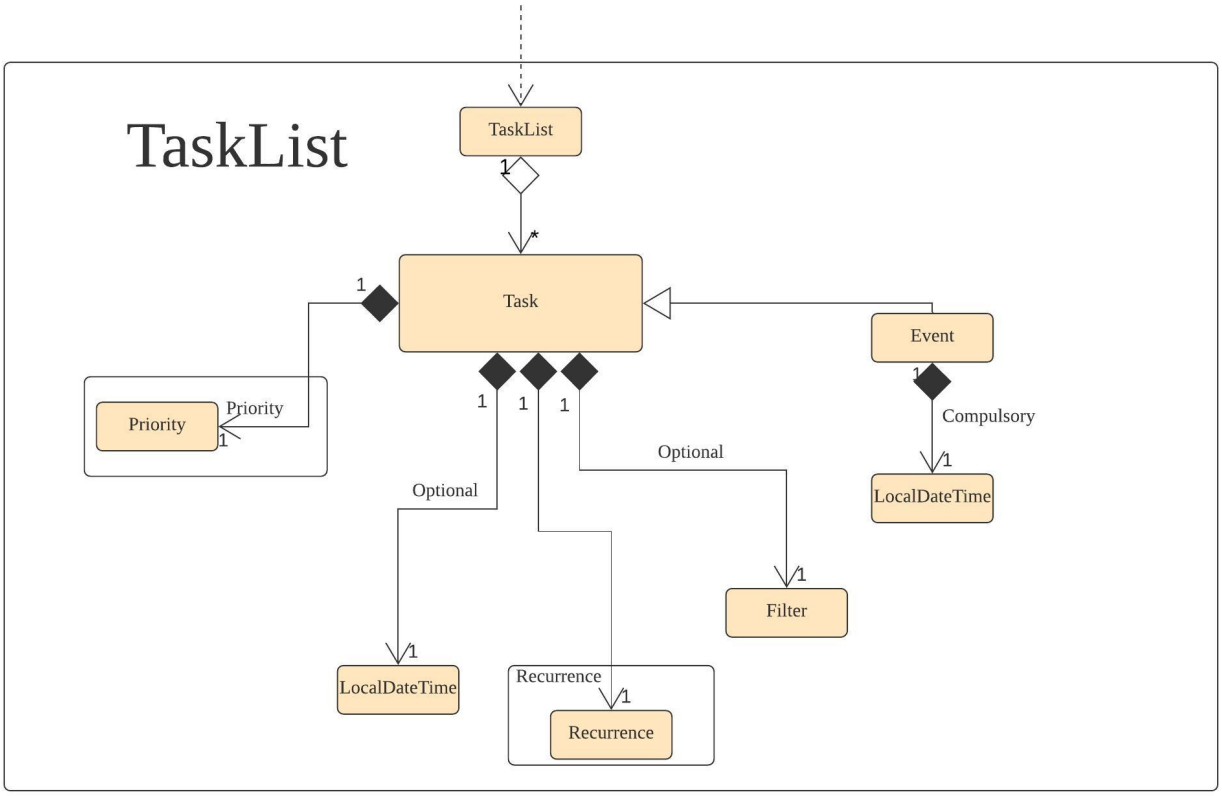

Figure 5. Structure of the TaskList Component

API : `Model.java`

The `TaskList`:

- Stores an array list of tasks that represents tasks that the user wishes to store.
- Event object extends tasks and enforces a compulsory `LocalDateTime` object denoting the time of the event
- `TaskList` does not depend on other components

2.4. Storage component

[StorageClassDiagram] | *StorageClassDiagram.png*

Figure 6. Structure of the Storage Component

The storage package contains 3 classes - `RuntimeTypeAdaptorFactory`, `Storage` and `UndoStack`. `RuntimeTypeAdaptorFactory` is a class that facilitates serializing and deserializing since we store our persistent data in a json file. Storage is a class that deals with the saving and loading of all task data the user has saved. `UndoStack` contains all the mirror commands of commands which can be undone.

`Storage` depends on `TaskList` and `Task` because it needs to have the `Task` and `TaskList` variables before saving them. `UndoStack` has a stack of `Commands`, so it depends on `Command`. `UndoCommand` needs the `UndoStack` to perform “un-doing” while `AddCommand`, `EditCommand`, `DeleteCommand` and `DoneCommand` need to add the “negation” of performed tasks to UndoStack.

3. Implementation

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

3.1. Filtering

3.1.1. Implementation

The filter mechanism involves attaching filters to tasks or events which is facilitated by `Parser` and `AddCommand`.

`Parser` reads user input and if the first word in the user input starts with a `-` character, `Parser` will identify the first word as a filter and carry out the remaining command using the filter. This enables the following user commands:

- `[<filter>] task/event [<description>] [<labels>]` — Creates a new task or event with relevant description and modifications as well as a filter for the task.
- `[<filter>] list [<labels>]` — List all tasks and events with the relevant filter and labels
- `[<filter>] edit [<index>/<description>] [<labels>]` — Edit a task or event seen in the relevant filtered list by its index in that list or by its description.

Given below is an example usage scenario and how the filter mechanism behaves at each step:

```
-CS2113 task DG submission -d 2 -t 251019 2359
-CS2113 list
-CS2113 edit 1
```

1. The user launches the application for the first time. The `Duke` main class will be initialized, and the `Ui` class will prompt the user to key in input.
2. The user executes `-CS2113 task DG submission -d 2 -t 251019 2359` command to add a new task called `DG submission` which will have a `filter` `CS2113`, a `duration` of `2 hours` to complete, and a deadline at `25/10/2019 23:59`.

> **NOTE** If the user's system somehow crashes after executing the above command, the new task entry will still be saved into the JSON storage file and can be recovered on the next launch of the application.

3. The user executes `-CS2113 list` to view all tasks and events associated with `CS2113`.
4. The user now decides that setting the duration of `DG submission` to be only 2 hours was a mistake, and decides to increase the duration needed by executing `-CS2113 edit 1 -d 4`. The `edit` command will call `EditCommand`, which will search for the corresponding task in the `TaskList`, updating whatever values the user has input, in this case updating the `duration` to `4 hours`.

3.1.2. Design Considerations

Aspect: How filter works

- **Alternative 1 (current choice):** Use an `Optional<String>` attribute within `Task` to keep track of what filter each `Task` has.
  - Pros: Easy to implement.
  - Cons: May have performance issues in terms of speed when calling list because a new list must be filtered from all existing tasks within the current `TaskList`.
- **Alternative 2:** When a new `filter` is created, create a new `TaskList` specific to that `filter` to store those tasks.
  - Pros: Will be faster to show filtered list to the user
  - Cons: Must change implementation of `TaskList`, `TaskListPrinter`, `AddCommand`, `EditCommand` to facilitate this. We must ensure that the implementation of each individual command is correct.

3.2. Undo

3.2.1. Implementation

The undo mechanism is facilitated by `Duke`, `UndoStack`, and `UndoCommand`. `UndoStack` stores the current undo history internally as a `java.util.Stack` object. `UndoStack` only stores undo information for when the user executes a command that we consider "undo-able". Undo-able commands include the following: `AddCommand`, `DeleteCommand`, `EditCommand`, and `DoneCommand`. Undo information is actually mirror `Command` classes that will do the opposite of what the current command has done. For example, the mirror of `AddCommand` is `DeleteCommand`. This storing of undo information is facilitated by `Duke`. `Duke` calls the `savePrevState()` method of these commands to create respective mirror classes.

When `UndoCommand` is executed, it will check whether `UndoStack` contains any commands using the `UndoStack` method `isEmpty()`. If `UndoStack` is not empty, `UndoCommand` will call the `UndoStack` method `retrieveRecent()` to obtain the most recent undo-able command that the user called. `UndoCommand` will then execute that command, undoing the user's most recent undo-able command.

Given below is an example usage scenario and how the undo mechanism behaves at each stage:

```
task mistake
undo
```

1. The user launches the application for the first time. The `Duke` main class will be initialized, and the `Ui` class will prompt the user to key in input.
2. The user executes `task mistake` command to add a new task called `mistake`. `AddCommand` will be executed to create the task and add it to the current `TaskList`. `Duke` will call the method `savePrevState()` on the `AddCommand` created, which will store a mirror `DeleteCommand` that corresponds to the newly added task into the `UndoStack`.

3. The user realises that adding that task was a mistake and wants to undo his action of adding the task. The user now executes `undo` command to undo his previous action of adding the mistake task.
4. `UndoCommand` will be created and executed. `UndoCommand` calls the `isEmpty()` method of `UndoStack` and realises that there is at least one command that can be undone. `UndoCommand` then calls `retrieveRecent()` method of `UndoStack` to obtain the most recent undo-able command's mirror command. `UndoCommand` will execute this mirror command and undo the recent adding of the task `mistake`.

3.2.2. Design Considerations

Aspect: How undo works

- **Alternative 1 (current choice):** Use a `java.util.Stack` to store mirror commands that facilitate the undoing of undo-able commands.
  - Pros: Easy to implement, fast, saves space.
  - Cons: Difficult to concurrently implement a redo feature.
- **Alternative 2:** Use a java.util.List to store each state of the TaskList whenever it changes regardless of whether calling Undo/Redo.
  - Pros: Able to efficiently execute undo/redo.
  - Cons: Takes up more space which might slow down the program overall or even cause an eventual `Memory Limit Exceeded` exception.

3.3. Editing

3.3.1. Implementation

The edit feature extends `Command` class. This commands takes in the filter and a string containing the attributes and it's update field.

Given below is an example usage scenario of an edit that includes a filter and how it behaves at each step. The command given is:

```
-cs edit 5 -priority 2
```

1. `Parser` separates the full command into a filter, `-cs` and the string, `edit 5 -priority 2`. Parse`r subsequently returns an `EditCommand` with these two strings as parameters.
2. `Duke` calls `Command.execute()`, which runs the overridden function in `EditCommand` and takes in `TaskList`, `UI` and `Storage`.
3. `Edit` command will call 3 functions to get the relevant fields. `getParameters()` which splits the command by regex ` -` and returns a string array, in this case `priority`. `getIndexFromCommand()` returns an integer of the index in the command, `5`. `getIndexFromTaskList()` takes in the filter, index from command and the list of tasks. It will loop through `TaskList` and finds tasks with the priority filter and matches the index specified by the user to the actual index in the task list.

4. `EditCommand` will access the specified task using the index and calls `setPriority` to update the priority of the task.

The following sequence diagram summarises how the edit operation works:

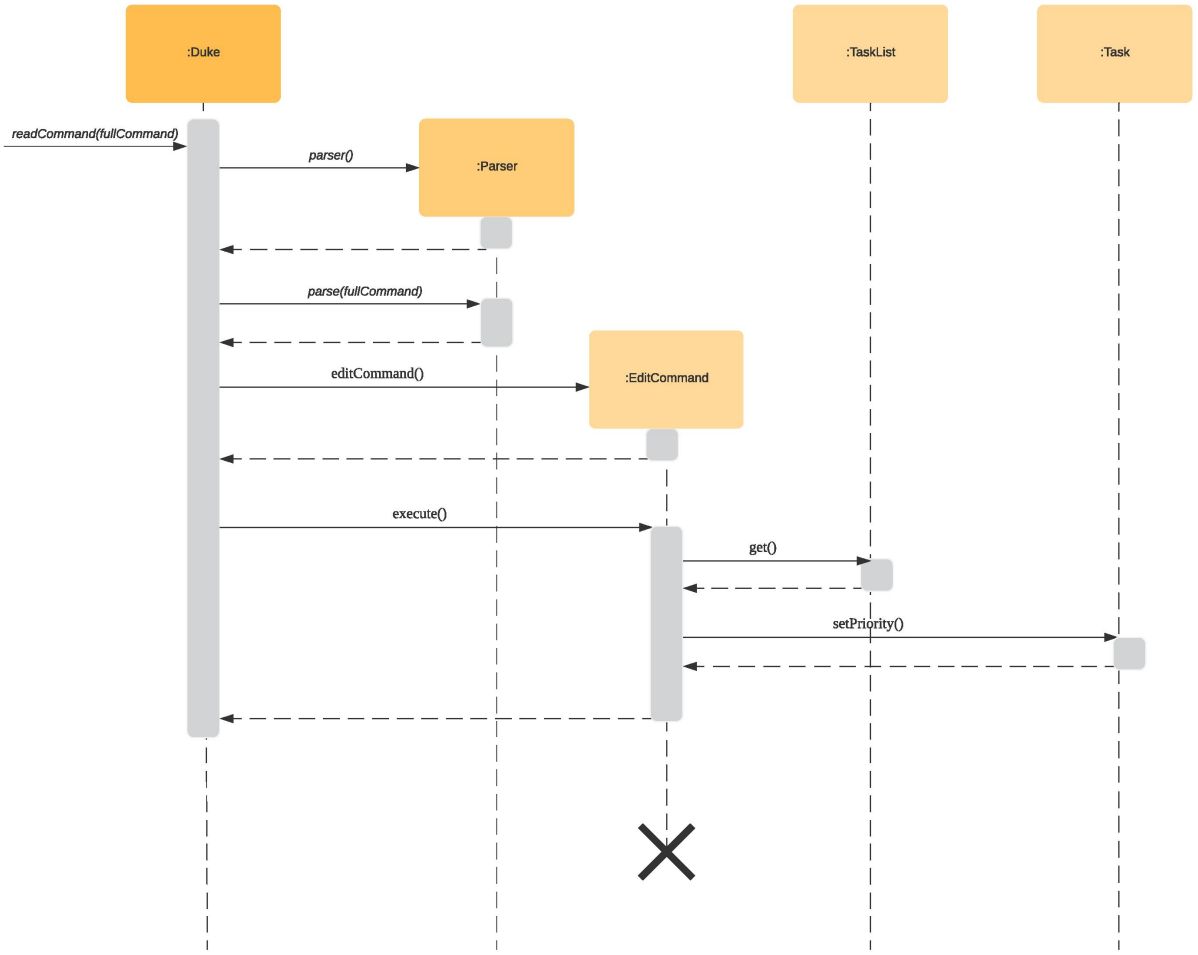

Figure 7. Sequence Diagram for Edit Command

3.3.2. Design Considerations

Aspect: How actual index of task is found when given an index based on the filtered list

- **Alternative 1 (current choice):** Using 2 separate counters to keep track of actual index versus filtered index
  - Pros: Easy to implement.
  - Cons: Harder to understand.
- **Alternative 2:** Saves a copy of the filtered list
  - Pros: Clearer to understand.
  - Cons: Have to deal with multiple lists.

Aspect: How the task are updated

- **Alternative 1 (current choice):** Updating each attribute of a task separately.
  - Pros: Reduces the dependency needed, `EditCommand` only needs to call a setter function instead of a getter for existing attributes followed by `AddCommand`.
  - Cons: Have to create a setter for each attribute of task
- **Alternative 2:** Create a new task with updated fields.
  - Pros: Won't need to call individual setters for each attribute.
  - Cons: Have to deal with AddCommand class to create a new task.

3.4. Pomodoro feature

3.4.1. Implementation

The Pomodoro feature is an inbuilt timer based on the pomodoro workflow. It allows the user to run a timer based on the 25-5 work-rest work ratio and reminds users to manage their work-rest ratio properly.

This feature is implemented using the Singleton design pattern with only one instance of the Pomodoro object. Accessing this instance is through the getInstance() method.

The Pomodoro object instantiates a java.util timer and the pomodoroTimerTask class and manages the starting and stopping of the timer. The class is configured as follows:

- The timer is configured to run the pomodoroTimerTask.run() every minute.
- pomodoroTimerTask takes in an integer parameter minutesRemaining and decrements it every minute. This determines how many minutes to run this timerTask for. After it reaches 0, the timer task will cancel itself.

Pomodoro States The states are configured based on the pomodoro workflow and are as follows:

- Work : Set to 25 minutes intervals, once completed will switch to a break.
- Short Break: Set to 5 minutes interval, runs after ever 1st, 2nd and 3rd work cycle.
- Long Break: Set to 15 minutes interval, runs after every 4th work cycle.

Commands This feature uses the keyword pomo followed by a few commands to control it's actions as listed below.

```
pomo [MODIFICATION]
```

`start`: Begins the timer.

`status`: Outputs the current status of the pomodoro, minutes remaining and current cycle.

`reset`: Resets the timer back to the previous state.

`restart`: Restarts the pomodoro to the 1st work cycle.

`stop`: Stops the timer.

`[Coming in v1.4] set`: Assigns a task to the current pomodoro and prints out during status updates.

Given below is the sequence diagram of how the pomodoro feature work.

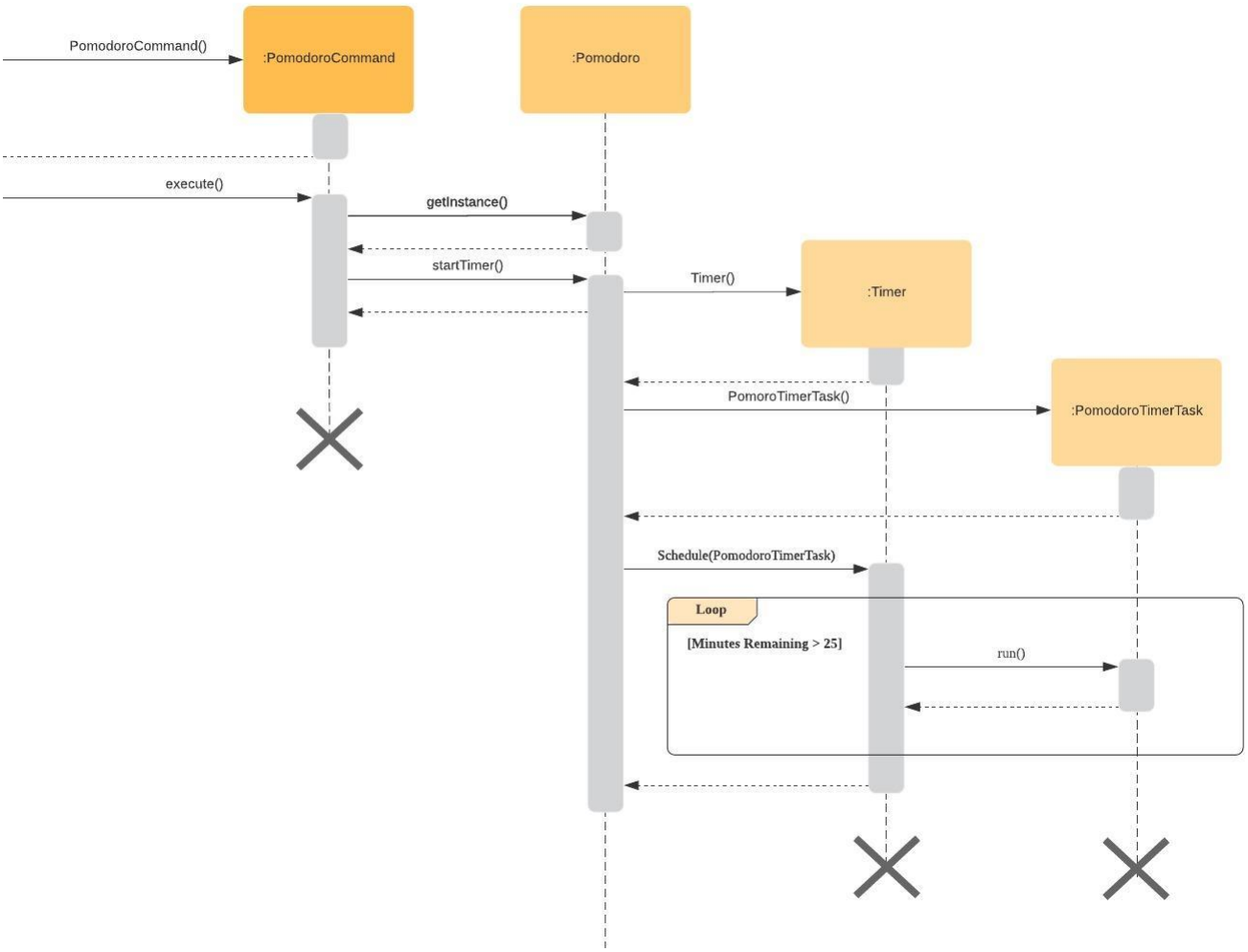

Figure 8. Sequence Diagram for Pomodoro Feature

3.4.2. Design Considerations

Aspect: How actual index of task is found when given an index based on the filtered list

- **Alternative 1 (current choice):** Using 2 separate counters to keep track of actual index versus filtered index
  - Pros: Easy to implement.
  - Cons: Harder to understand.
- **Alternative 2:** Saves a copy of the filtered list
  - Pros: Clearer to understand.
  - Cons: Have to deal with multiple lists.

Aspect: How the task are updated

- **Alternative 1 (current choice):** Updating each attribute of a task separately.
  - Pros: Reduces the dependency needed, `EditCommand` only needs to call a setter function instead of a getter for existing attributes followed by `AddCommand`.
  - Cons: Have to create a setter for each attribute of task
- **Alternative 2:** Create a new task with updated fields.
  - Pros: Won't need to call individual setters for each attribute.
  - Cons: Have to deal with AddCommand class to create a new task.

3.5. Finding

3.5.1. Implementation

The find feature is an extension of the `ListCommand`. As the name suggests, it allows the user to list tasks whose description matches the text.

Given below is an example usage scenario of FindCommand:

```
-cs find tutorial
```

1. The command `-cs find tutorial` is parsed into `Parser`. `Parser` then splits the command into two parts delimited by the first . Filter contains "cs" and command contains "find tutorial".
2. `Parser` will return `FindCommand` with the two parameters to duke. The two parameters are keyword which is "tutorial" and filter which is "cs". `FindCommand` will then create a `ListCommand` with the filter.
3. The actual execution begins. First initialize new `TaskList` foundTasks to store tasks whose description matches the keyword. Next, pass foundTasks to `ListCommand` to show the relevant tasks.

3.5.2. Design Considerations

FindCommand uses the ListCommand object as an attribute. This can be seen as a HAS-A or composition relationship. We chose not to use inheritance as FindCommand is not strictly a ListCommand. Also, this follows the composition over inheritance principle in object-oriented programming. Care was taken not to mutate the "main" TaskList in place, but rather, create a new TaskList to store tasks that match the keyword.

3.5.3. Future Scope (Next Milestone)

Use similar idea of substring matching with other commands. E.g. done tutorial ⇒ show user all the tasks that contain tutorial and ask him to choose which one he has completed.

3.6. Feature to handle typos and shortforms

3.6.1. Implementation

It is possible that the user typed the command wrongly. E.g. “tasl” instead of “task”. The user will be prompted `(Y/N)` for the correct command name. Helps save time - type a letter instead of full command. Off by one means off by 1 substitution error. This functionality is implemented in `OffByOneChecker`. Also, sometimes the user can be lazy to type the whole command, so we allow shortforms - ‘t’ instead of ‘task’, ‘del’ instead of ‘delete’ etc. This functionality is implemented in `StartsWithChecker`.

Given below is an example usage scenario of `Off-By-One`:

```
-cs tasl tutorial
> Did you mean task? (Y/N)
Y
```

1. The command “tasl tutorial” is parsed into `Parser`. `Parser` then splits command into two parts delimited by the first “ “. Filter contains cs and command contains “tasl tutorial”.
2. The `Parser` always uses `OffByOneChecker` to check if the keyword is off by one. How this works is simple. For each possible command, check the number of identical characters. E.g. for “task” and “tasi”, the number of characters is 3. If number of identical characters == length of command, just return the keyword. If number of identical characters == length of command - 1, prompt user (Y/N). If Y, return command, else if N, return keyword. If number of identical characters != (length of command or length of command - 1) for all commands, just return keyword.
3. Execute the command as per normal. In this case. “tutorial” task is created under “cs”.

3.6.2. Design Considerations

The `OffByOneChecker` and `StartsWithChecker` is designed such that regardless of a valid command, off by one command or wrong command, the expected results are consistent. Hence, we always use `OffByOneChecker` and `StartsWithChecker` on each input/keyword. We can think of `OffByOneChecker` and `StartsWithChecker` as “filtering” the keyword before matching the keyword to some command. The benefit of doing this is that this results in minimal changes in existing code. The Parser class only needs to add 2 lines of code.

3.6.3. Future Scope (Next Milestone)

Use a more efficient data structure. Extend this to other keywords like “priority”.

3.7. Automatically assign filter to some task feature

3.7.1. Implementation

Some tasks which the user has created are yet to be assigned to a filter. If the user wants to clean up

his unassigned tasks by assigning them to some existing filter but he is either not sure what the filter should be or cannot recall the name of the filter, he could use the auto assign feature.

The autoassign command requires taking in the index of an unassigned task. If the index given refers to a task with an existing filter already, the user will be informed that the task has already assigned to a filter.

The autoassign command suggests a suitable filter for the task based on 2 heuristics.

The first heuristic is filter name matching and is straightforward. If the description of the task contains the name of any filter, `ElDuque` will suggest to the user if he wishes to assign the task to the filter. Note that there can be more than 1 filter that meets this criteria. If user says N, then `ElDuque` will prompt for other filters until the user says Y or there are no filters left.

The second heuristic is using cosine similarity and is slightly more complicated. Here are a list of steps `ElDuque` performs for this part:

1. For all tasks, we get their descriptions and clean them.
2. Cleaning refers to removing beginning and ending whitespace, removing punctuation, changing to lowercase and removing excessing whitespace. E.g. “Buy math textbook.” will be cleaned to “buy math textbook”.
3. All descriptions can now be tokenized into individual words.

   ```
   E.g. "buy math textbook" is tokenized into ["buy", "math", "textbook"].
   ```

4. Next, we get a set of unique tokens. Now, we can express the description of each task as a vector of counts of the unique tokens.

   ```
   For example, suppose the set of unique tokens are:
   ["buy", "sell", "laundry", "textbook", "to", "math", "do", "problems", "pass", "exam"]
   The description will be vectorized like so:
   "buy math textbook to do math problems to pass exam" => [1 0 0 1 2 2 1 1 1 1]
   ```

5. For each filter, we can then find all the corresponding vectors and sum them up.
6. Now, we have a vector for each filter. For the unassigned task, we also have a vector.
7. Find the cosine similarity between the task vector and each of the filter vectors. If the best cosine similarity is 0, inform the user there are no suitable filters to auto assign. Else, suggest the filter with the highest cosine similarity.

3.7.2. Design Considerations

Given the large number of steps for this command, we can isolate many parts of the implementation into methods. For instance, calculating vector counts for a list of tokens can be a method. This improves code reuse and readability.

Appendix A: Product Scope

Target user profile:

- computer science students
- can type fast
- prefers typing over mouse input
- reasonably comfortable using CLI apps

Value proposition: Simple default settings to manage tasks for new users which work well. Advanced commands for power users to further increase efficiency of the app.

Appendix B: User Stories

Priorities: High (must have) - `* * *`, Medium (nice to have) - `* *`, Low (unlikely to have) - `*`

Priority	As a …	I want to …	So that I can…
`* * *`	new user	see usage instructions	refer to instructions when I forget how to use the App
`* * *`	user	add a new task	
`* * *`	user	delete a task	remove entries that I no longer need
`* * *`	user	find a task by name	locate details of tasks without having to go through the entire list

{More to be added}

Appendix C: Use Cases

(For all use cases below, the **System** is the `El-Duque` and the **Actor** is the `user`, unless specified otherwise)

Use case: Create a task

MSS

1. Start system
2. System starts with a welcome message
3. User uses the Create Task command
4. User enters the details of the task such as description, type, timing, tags
5. System will inform the user that the task has been created successfully

   Use case ends.

Extensions

3a. User creates a recurring task.

   3a1. User uses a recurring tag to create a scheduled task e.g. -r week as in **Use Case 2**.

   Use case ends.

4a. System detects invalid input.

   4a1. System informs the user and requests for correct input.

   Use case resumes at step 3.

4b. System detects conflicting event times.

   4a1. System informs the user of conflict and requests for correct input.

   Use case resumes at step 3.

{More to be added}

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be able to hold up to 1000 tasks without a noticeable sluggishness in performance for typical usage.
3. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Appendix F: Instructions for Manual Testing

Given below are instructions to test the app manually.

> **NOTE** These instructions only provide a starting point for testers to work on; testers are expected to do more *exploratory* testing.

F.1. Launch and Shutdown

1. Initial launch
   a. Download the jar file and copy into an empty folder
   b. Double-click the jar file
   Expected: Shows the CLI with a welcome message. The window size may not be optimum.
2. Shutdown
   a. Enter the command `bye`
   Expected: A goodbye message will be shown and the app will exit.

F.2. Adding a task

1. Adding a new simple task
   a. Test case: `task example`
   Expected: New task created called example which can be viewed by calling `list`. This undone task will have no filter, no datetime, no recurrence, no duration, and a low priority.
2. Adding a new complex task
   a. Test case: `-test task example -d 7 -t today -r daily`
   Expected: New task created called example which can be viewed by calling `list`. This undone task will have a filter `test`, the current local datetime, a daily recurrence, a 7 hour duration to complete, and a low priority.

F.3. Deleting a task

1. Deleting a person while all tasks are listed
   a. Prerequisites: List all tasks using the `list` command. At least one task in list.
   b. Test case: `delete 1`
   Expected: First task is deleted from the list. Details of the deleted task shown in the status message.
   c. Test case: `delete 0`
   Expected: No task is deleted. Error details shown in the status message. Status bar remains the same.
   d. Other incorrect delete commands to try: `delete`, `delete x` (where x is larger than the list size)
   _ Expected: Similar to previous.

2. Deleting a person while filtered tasks are listed

    a. Prerequisites: List filtered tasks using the `<filter> list` command. At least one task in filtered list.

    b. Test case: `<filter> delete 1`
    Expected: First task is deleted from the `<filter> list`. Details of the deleted task shown in the status message.

    c. Test case: `<filter> delete 0`
    Expected: No task is deleted. Error details shown in the status message. Status bar remains the same.

    d. Other incorrect delete commands to try: `<filter> delete`, `<filter> delete x` (where x is larger than the filtered list size) _ Expected: Similar to previous.