OOF (Outstanding Organisation Friend) - Developer Guide

By: `Team W17-4` Since: `Aug 2019` Licence: `MIT`

1. Setting up

1.1. Prerequisites

1. **JDK** `11` or above

The `oof.jar` file is compiled using the Java version mentioned above.

2. **IntelliJ** IDE

IntelliJ by default has Gradle and JavaFx plugins installed. Do not disable them.
If you have disabled them, go to `File` > `Settings` > `Plugins` to re-enable them.

1.2. Setting up the project in your computer

1. Fork this repo, and clone the fork to your computer
2. Open IntelliJ (if you are not in the welcome screen, click `File` > `Close Project` to close the existing project dialog first)
3. Set up the correct JDK version for Gradle
   a. Click `Configure` > `Project Defaults` > `Project Structure`
   b. Click `New…` and find the directory of the JDK
4. Click `Import Project`
5. Locate the `build.gradle` file and select it. Click `OK`
6. Click `Open as Project`
7. Click `OK` to accept the default settings
8. Open a console and run the command `gradlew processResources` (Mac/Linux: `./gradlew processResources`). It should finish with the `BUILD SUCCESSFUL` message.
   This will generate all resources required by the application and tests.

1.3. Verifying the setup

1. Run `Oof` and try a few commands
2. Run tests to ensure they all pass. *(coming soon)*

2. Design

2.1. Architecture

2.2. UI component

2.3. Logic component

2.4. Model component

2.5. Storage component

2.6. Common classes

3. Implementation

3.1. Recurring task feature

3.1.1. Implementation

The `RecurringCommand` class extends `Command` by providing methods to set a current `Task` in the persistent `TaskList` of the main program `OOF` as a recurring task. It also generates future instances of `Task` as indicated by the user.

`TaskList` is stored internally as an `ArrayList` in the Oof Program as well as externally in persistent storage in `output.txt`.

Additionally, it consists of the following features:

- User can select a `Task` in the `TaskList` to be a recurring task.
- User can choose an integer between `1 - 10` for the number of times the task should recur.
- User can also choose the `Frequency` of recurrence with the choices being `DAILY, WEEKLY, MONTHLY, YEARLY`

These features are implemented in the `parse` method of the `CommandParser` class that parses user input commands.

Given below is an example usage scenario and how the `RecurringCommand` class behaves at each step.

Step 1.
The user types in `recurring 1 2`. The `parse` method in `CommandParser` class is called to parse the command to obtain integers `1` as the `Index` of the `Task` in `TaskList` and `2` as the `number of recurrences`.

`OofException` will be thrown if the user enters invalid commands.

Step 2.
The `parse` method then prompts the user to input the `frequency` of recurrence.

The choices are as follows:
1. DAILY
2. WEEKLY
3. MONTHLY
4. YEARLY

The user chooses `1`. The `parse` method then parses the command to obtain an integer `1` which sets the `Frequency` of recurrence as `DAILY`.

`OofException` will be thrown if the user enters invalid commands.

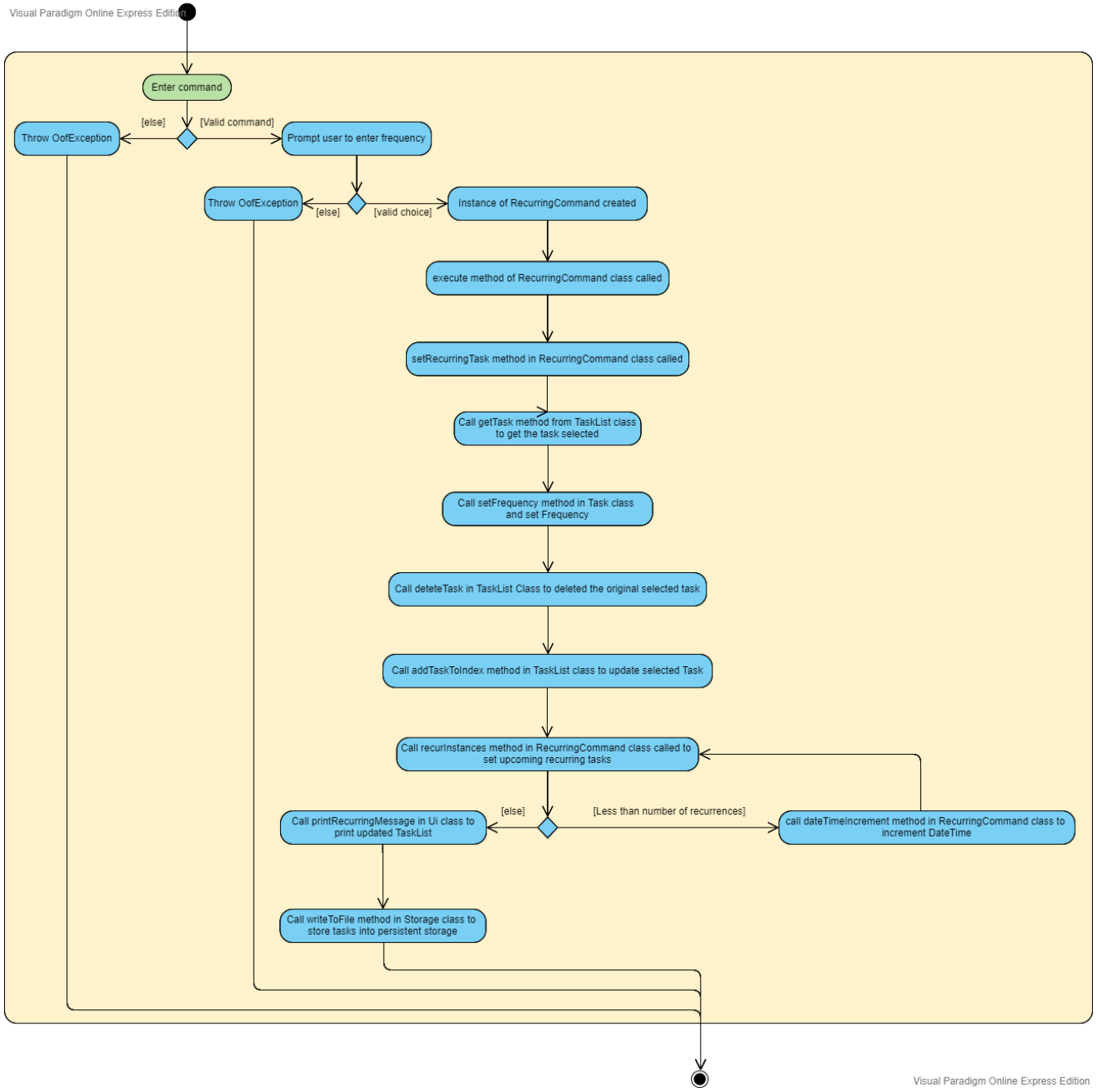
Step 3.
A new instance of `RecurringCommand` class is returned to the main `Oof` program with the parameters `1, 2, 1` as described above. The `execute` method of `RecurringCommand` class is then called.

Step 4.
The `setRecurringTask` method in `RecurringCommand` class is then called by `execute` method. This method does three main things:

- Calls `getTask` method from `TaskList` class to get the user selected `Task`.
- Updates the `Task` to a `recurring Task` by:
  - Calling `setFrequency` method in `Task` class to set `Frequency` to `DAILY`
  - Calling `deleteTask` and `addTaskToIndex` methods in `TaskList` class to update the selected `Task`.
- Calls `recurInstances` method in `RecurringCommand` class to set upcoming recurring `Tasks` based on user selected `Number of recurrences` and `Frequency` by:
  - `recurInstances` method calls `dateTimeIncrement` method in `RecurringCommand` class to increment the `DateTime` based on the user input `Frequency`.

Step 5.
After `setRecurring` method finishes its execution, the `execute` method continues on to print the updated `TaskList` by calling `printRecurringMessage` method in `Ui` class and saves the new `Tasks` into persistent storage by calling `writeToFile` method in `Storage` class.

The following activity diagram summarises what happens when a user executes a new command:



3.1.2. Design Considerations

- **Selecting currently available `Task` to be set as a `recurring Task`**
  - Rationale:
    It allows the `RecurringCommand` class to capitalise on the existing features of adding `Deadlines` and `Events.
  - Alternatives considered:
    Allow users to add new `recurring Task` instead of selecting from existing `Tasks`. Allowing users to add new recurring tasks strongly overlaps with existing features and this increase coupling in the `OOF` program.
- **Fixing lower bound and upper bound of the `Number of recurrences` to be `1` and `10` respectively**
  - Rationale:
    It ensures a controlled number of recurrences are added to the `TaskList` instead of being a

variable amount as a user may unintentionally break the `TaskList`.

- Alternatives considered:
  Insert an upcoming recurring task when the `recurring Task` is nearing. There may be too many `Tasks` to keep track and add when `OOF` starts up especially in the case when the number of `Tasks` in the `TaskList` gets potentially large. This decreases the scalability of the project in the long run.

- **`Frequency` fixed to four different default frequencies**
  - Rationale:
    It requires significantly less effort to choose from a default list of four options than to manually type in customised time ranges.
  - Alternatives considered:
    Users can enter a customised `Frequency` for the `recurring Task`. It may be a viable option to allow users to set such parameters. However, since the `OOF` program is solely a Command Line Interface program, it may not be user friendly for users to enter so many details just to set a customised `Frequency` for the `recurring Task`.

3.2. Help feature

3.2.1. Implementation

The `HelpCommand` class extends `Command` class by providing functions to display a manual with the list of `Command` available and how they may be used in the main program `OOF`.

The list of `Command` and their instructions are stored externally in persistent storage in `manual.txt`.

In addition, it contains the following feature: * User may request for `Help` with a specific command.

All `Help` features are implemented in the `parse` method of `CommandParser` class that parses user input.

Provided below is an example scenario of use and how `HelpCommand` class behaves and interact with other relevant classes.

Step 1:
The user enters `help Deadline`. The `parse` method in `CommandParser` class is called to parse the user input to obtain the String `Deadline` as the `keyword` that the user requires `Help` for.

`OofException` will be thrown is the user enters an invalid command.

Step 2:
The `execute` method of `HelpCommand` class will read the list of `Command` and their instructions from persistent storage in `manual.txt` and store them into a `commands` ArrayList by calling the `readManual` method from `Storage` class.

- **Step 2a:**
  The `readManual` method of `Storage` class will retrieve and read `manual.txt` from persistent storage by using `FileReader` abstraction on `File` abstraction.

- **Step 2b:**
  The `BufferedReader` abstraction will then be performed upon `FileReader` abstraction to allow `manual.txt` to be read line-by-line, adding each line as an element of the `commands` ArrayList. The `commands` ArrayList is then returned to the `execute` method of `HelpCommand` class.

`OofException` will be thrown if `manual.txt` is unavailable, resulting in `IOException` getting caught.

Step 3:
If the `keyword` is empty, the `printHelpCommands` method of `Ui` class will be called. The elements of `commands` ArrayList will then be printed in ascending order through the use of a for loop.

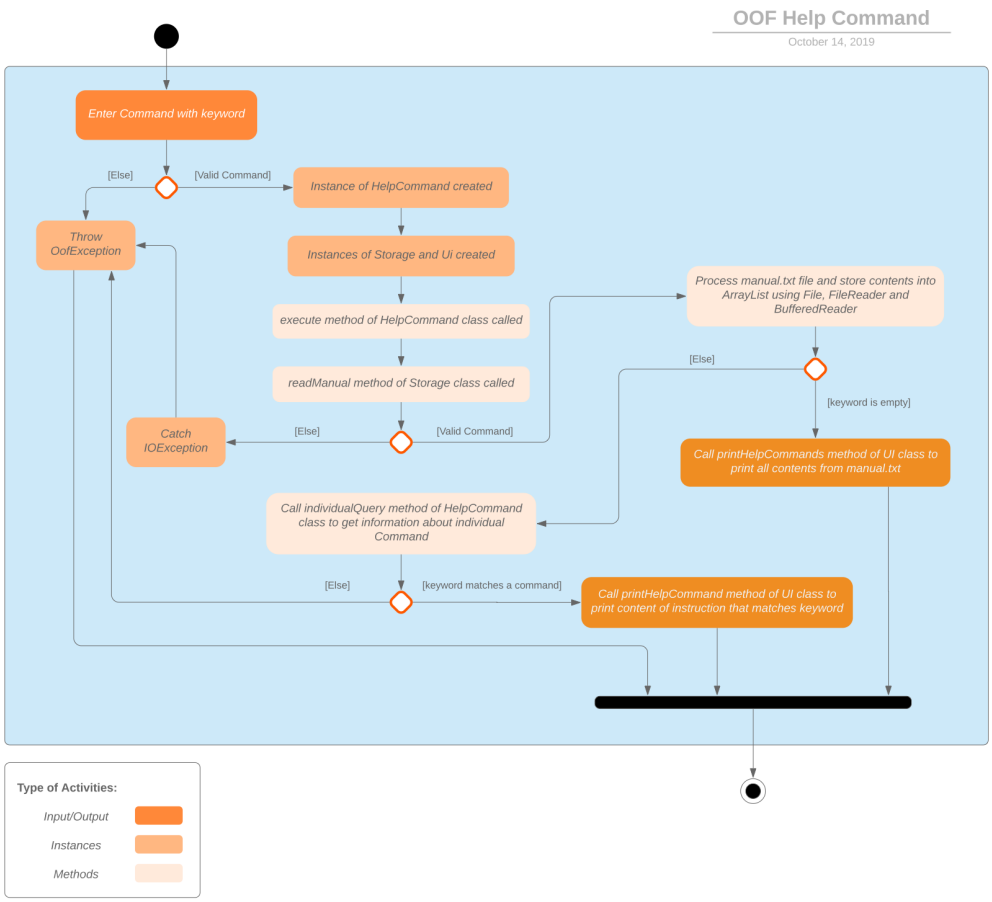
If the `keyword` is specified, the `individualQuery` method of `HelpCommand` class will be called with the `keyword` and `commands` ArrayList as parameters.

- **Step 3a:**
  The first segment of each element in the `commands` ArrayList will be retrieved by adding a String `command` delimited by two whitespaces.
- **Step 3b:**
  Once a check is completed to ensure that `command` is not empty, both `keyword` and `command` String will be formatted through the use of `toUpperCase` function and String comparison will be performed through the use of `equals`. If they match, that particular element of `commands` ArrayList will be stored into a String called `description` and the for loop will break before returning `description` to the `execute` method of `HelpCommand`.

`OofException` will be thrown is no successful match between `keyword` and `command` String is found.

Step 4:
The `execute` method of `HelpCommand` calls `printHelpCommand` in `Ui` class with `description` String as the parameter. This is where the individual `Command` and its instruction will be printed.

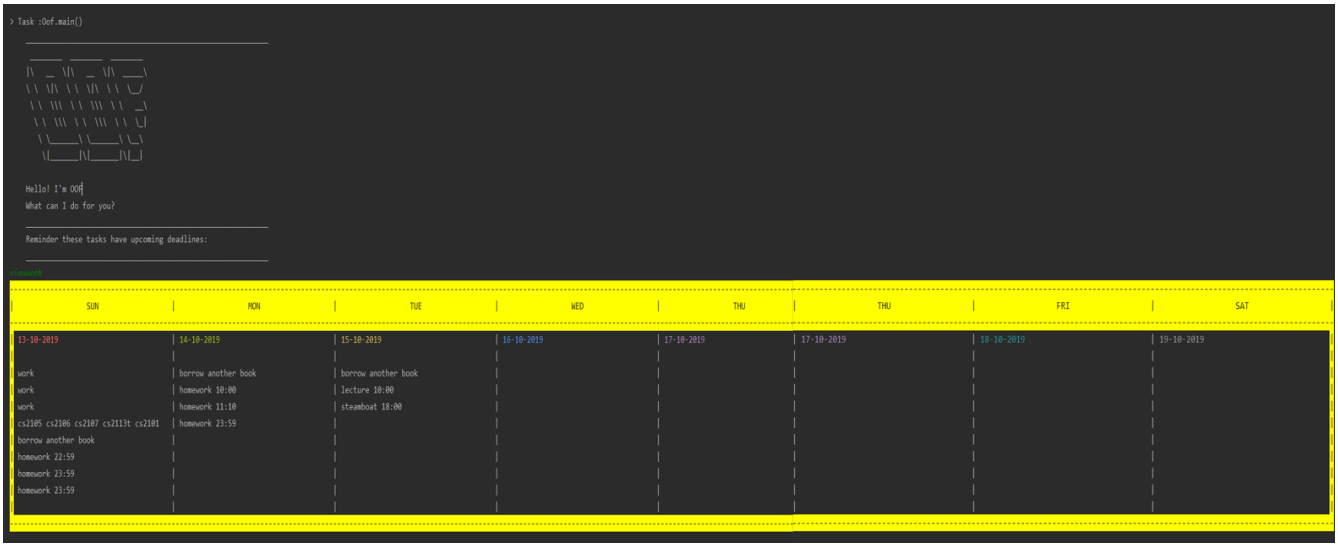
The following activity diagram summarises what will happen when a user executes a `Help` command:



3.2.2. Design Considerations

- Created `manual.txt` to store available commands and their instructions
  - Rationale:
    With scalability in mind, the use of persistent storage will grant developers a common location to update the list of `Command` and their instructions.
  - Alternatives Considered:
    Numerous String variables can be added to an ArrayList through the `HelpCommand` class. This would not require the use of `File`, `FileReader` or `BufferedReader` abstractions. However, this would bring developers inconvenience during project extension as more functions will be added and this may eventually lead to unorganised code, especially in the `HelpCommand` class.
- Implement ArrayList to display `Help` for an individual command and its instructions
  - Rationale:
    The use of ArrayList offers flexibility due to its unconfined size. This allows increased convenience and scalability due to the large list of `Command` and their instructions available to our users.
  - Alternatives Considered:
    The use of an Array will allow increased efficiency given the smaller number of `Command` we had in our earlier versions, such as `v1.1`. However, this is not a beneficial solution in the long run as we create extensions and expand upon `OOF`.

3.3. View tasks for the week feature



3.3.1. Implementation

The `ViewWeekCommand` class extends `Command` by providing methods to display tasks for a particular week.

The command can be run in the `OOF` program without a specific `date` e.g. `viewweek` instead of `viewweek 01 01 2019`. In this case, the `ViewWeek` command prints tasks for the current week. The same applies if the date entered by the user is invalid.

Features elaborated:

- The output of the `ViewWeekCommand` is ANSI colour enabled.This distinguishes the different days of the week in the output.

As the output is ANSI colour enabled, there is a need to enable ANSI colour support on Windows machines.

Type of fix	Description
Permanent fix	Enter the command `CMD / POWERSHELL: reg add HKCU\Console /v VirtualTerminalLevel /t REG_DWORD /d 1` in either `CMD` / `POWERSHELL`. Launch a new console window to activate the changes. *Disable line wrapping in terminal for optimal view.*
Adhoc fix for `POWERSHELL`	run `OOF` with the command `java -jar .\v1.X.jar | Out-Host`

- The output of `ViewWeekCommand` resizes automatically based on the length of the `description` of tasks.

Given below is an example usage scenario and how the 'ViewWeekCommand` class behaves at each step.

Due to heavy abstraction in the Ui and the limitation of the software used to draw UML diagrams, trivial helper functions in the Ui to print the output will be omitted.

Step 1.
The user types in `viewweek`. The `parse` method in the `CommandParser` class returns a new `ViewWeekCommand` object.

Step 2.
Since no date is passed by the user, the constructor for `ViewWeekCommand` class retrieves the current date using the `calendar.get()` methods. The `execute` method in `ViewWeekCommand` class is then called by the `Oof.run()` method in the main class `Oof`.

Step 3.
In the `execute` method, the first day of the week is retrieved using the `getStartDate()` method in the current class for indexing purposes. Tasks are to be sorted into the data structure of `ArrayList<ArrayList<String[]>>` called `calendarTasks`. The size of `calendarTasks` is `7` which represents each day in the current week. Each index in `calendarTasks` is an `arrayList`of `string[]` which represents the tasks in that respective day of the week in the form of `{TIME, DESCRIPTION}`.

Step 4.
The `execute` method iterates through the current list of tasks and parses the `date`, `time` and `description` of each task. The `dateMatches()` method is then called to verify if the task falls in the same week as the current week. If the current task falls in the current week, the `date` of the task is compared with the first day of the week to obtain an `index` to slot the task into calendarTasks.

Step 5.
The task is then added to calendarTask using the `addEntry()` method. After iterating through the

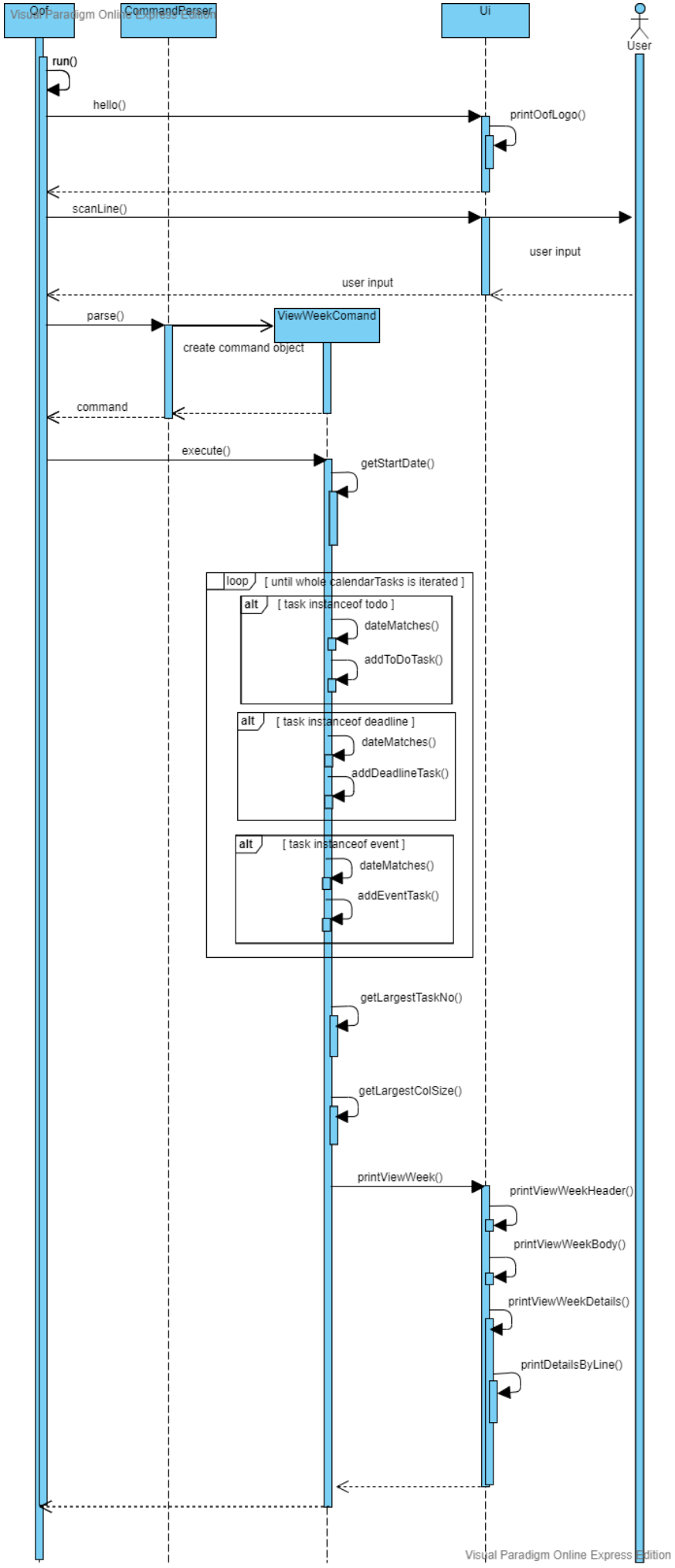
current list of tasks, the `printViewWeek() method` in `Ui` class is then called to print the tasks for the current week.

Step 6.
In the `printViewWeek()` method, 3 main methods are being called to print the final output. Firstly, `printViewWeekHeader()` method is called to print the header of the output which consists of the top border and the days of the current week.

Step 7.
Secondly, `printViewWeekBody()` method is called to print the dates of the current week in the next line of output.

Step 8.
Lastly, `printViewWeekDetails()` method is called to print relevant empty lines, tasks and the bottom border of the final output.

The following sequence diagram summarises what happens when a user executes a new command:

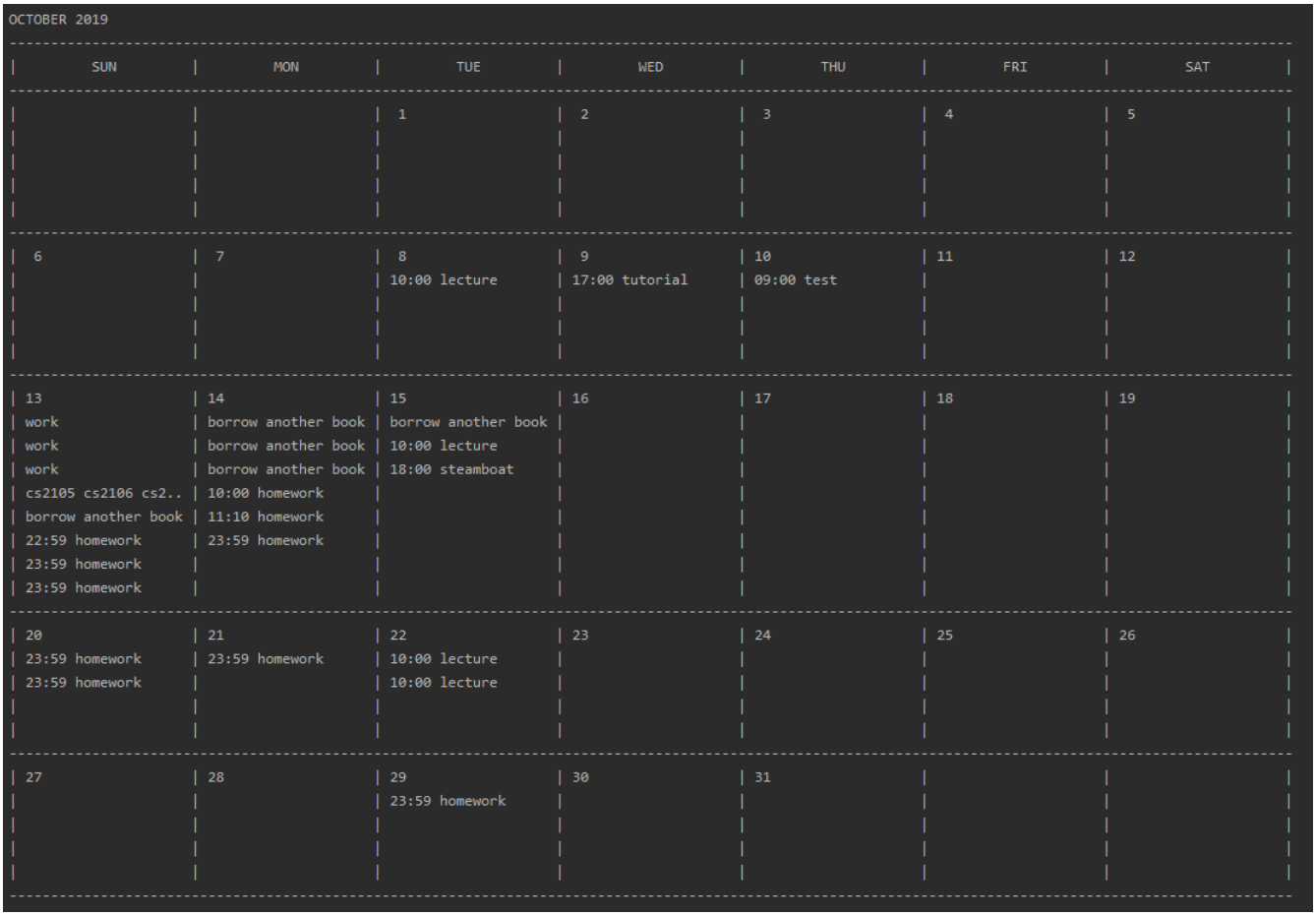

Visual Paradigm Online Express Edition
Oof
CommandParser
Ui
User
run()
hello()
printOofLogo()
scanLine()
user input
user input
parse()
ViewWeekComand
create command object
command
execute()
getStartDate()
loop
[until whole calendarTasks is iterated]
alt
[task instanceof todo]
dateMatches()
addToDoTask()
alt
[task instanceof deadline]
dateMatches()
addDeadlineTask()
alt
[task instanceof event]
dateMatches()
addEventTask()
getLargestTaskNo()
getLargestColSize()
printViewWeek()
printViewWeekHeader()
printViewWeekBody()
printViewWeekDetails()
printDetailsByLine()
Visual Paradigm Online Express Edition

The lifeline of the User is a bar due to limitations in the software used to draw the diagram. Full details of the entire program are omitted to prevent over-cluttering of the diagram.

3.3.2. Design Considerations

- **Resizing column size instead of wrapping description of tasks**
  - Rationale:
    Each task has a different description length and timing. Thus, it may be difficult to come up with a logic to wrap at indexes that make the output sensible. Furthermore, it is more difficult to find a one size fits all logic than to resize the columns to fit the task `description` and `time`.
  - Alternatives considered:
    Truncating the description of tasks so that no resizing nor wrapping is needed. A lot of information may be lost in this process and the `ViewWeekCommand` may not be very useful to the user in this case.
- **Coloured output instead of plain output**
  - Rationale:
    It clearly demarcates the header and borders of the output and highlights the dates shown in the `ViewWeekCommand` output. Without the coloured scheme, users still need to scan through the headers to realise the useful task information is located below it.
  - Alternatives considered:
    The tasks in each day can be classified into visual blocks to aid the users into visualising the timeline in each day. In addition to that, the tasks in each day has already been chronologically sorted in the `ViewWeekCommand` class. This alternative can be an extension to be used in conjunction with `Find free time slots` in future milestones.

3.4. View calendar for a month feature



3.4.1. Implementation

The `CalendarCommand` class extends `Command` by providing methods to display tasks for a particular month.

The command can be executed without the `month` and `year` argument e.g. `calendar` instead of `calendar 10 2019`. In this case, the `calendar` command prints the calendar and task for the current month and year. The same applies if the month and year entered by the user is invalid.

The following is an example execution scenario and demonstrate how the `CalendarCommand` class behaves and interact with other relevant classes.

Step 1
The user enters the command `calendar 10 2019`. The `parse` method in the `CommandParser` class is called to parse the command to obtain an array containing `10` and `2019` as it elements as arguments for the `CalendarCommand` class returned by the `CommandParser` class.

Step 2
The constructor for the `CalendarCommand` class will parse and validate the arguments, `10` and `2019`, in the argument array.

An `IndexOutOfBoundsException` will be thrown if less than 2 arguments are provided, a `NumberFormatException` will be thrown if argument provided is not an integer while an `OofException` will be thrown if `month` argument is not within `1` and `12`. In these cases, the program will retrieve the current `month` and `year` from the system.

Step 3
The `execute` method in the `CalendarCommand` class is then called by the `executeCommand()` method in the `Oof` class. This method does the following:

- Iterates through the `ArrayList` of `Task` from the `TaskList` class and checks if the `Task` belongs to the queried `month` and `year` using the `verifyTask` method.
- `Task` belonging to the queried `month` and `year` are added to the `ArrayList` corresponding to its `day`.
- Each `ArrayList` is then sorted in ascending order of `time` using the `SortByDate` comparator.

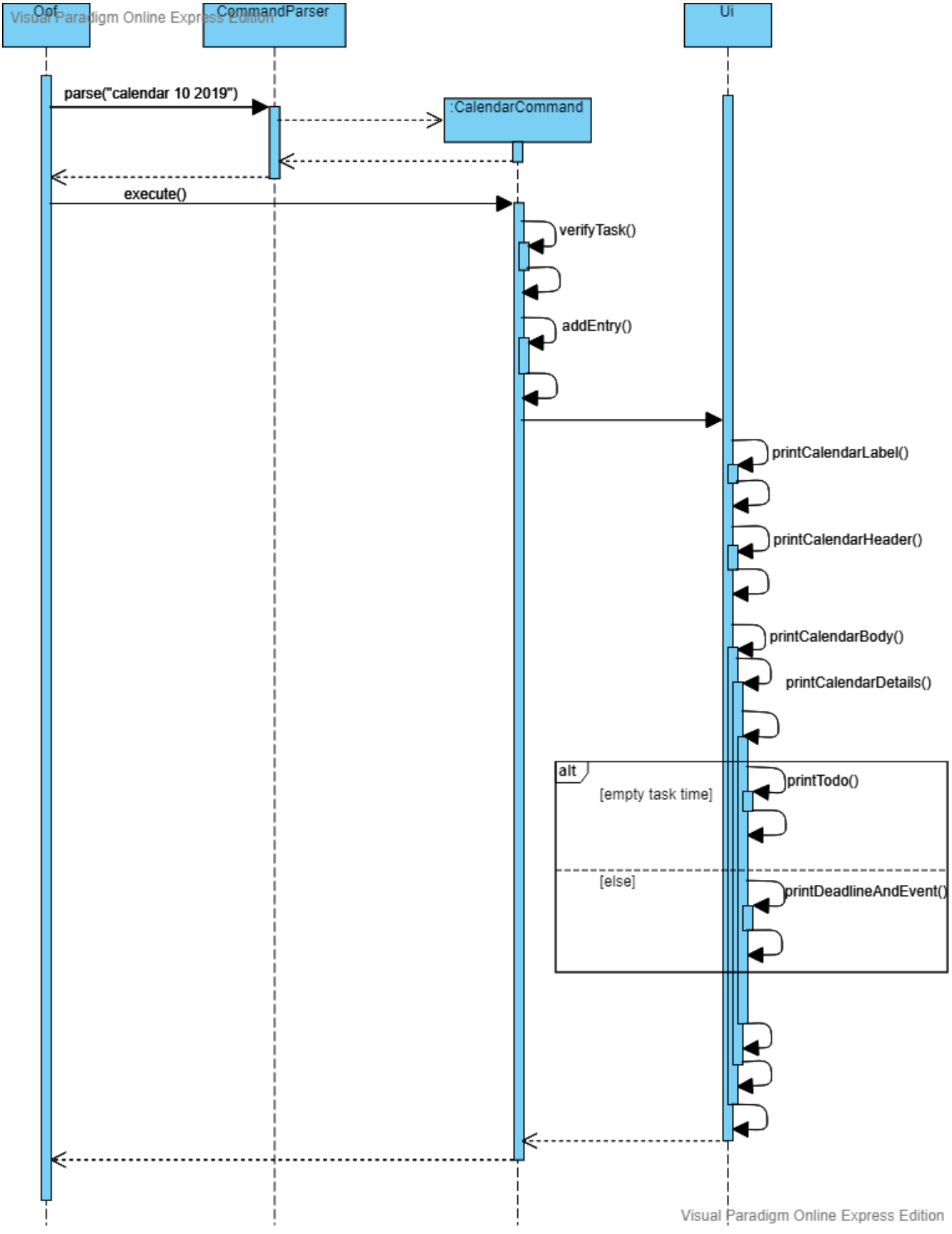
Since `Todo` objects do not have a `time` attribute, they are always sorted to the front of the `ArrayList`.

- `execute` then calls the `printCalendar` method in the `Ui` class.

Step 4
`printCalendar` calls `printCalendarLabel`, `printCalendarHeader` and `printCalendarBody` to print the calendar:

- `printCalendarLabel` prints the `month` and `year` being queried.
- `printCalendarHeader` prints the header of the calendar which consists of the top border and the days of a week.
- `printCalendarBody` prints the each day of the week and corresponding tasks belonging to each day.

The following sequence diagram summarises what happens when a user executes a `CalendarCommand`:



3.4.2. Design Considerations

- Extending row size instead of limiting number of tasks displayed
    - Rationale:
      Limiting number of tasks displayed might misrepresent the number of `Task` a person have

for that day.

- Alternatives considered:
  Implementing a GUI which includes a scroll pane for each day such that calendar size can be fixed.

- Truncation of task name instead of extending column size
  - Rationale:
    Since row size is extendable, extending column size would severely affect readability when column and row sizes increase independently of each other. Also, `ScheduleCommand` class can be used in conjunction with `CalendarCommand` to allows the user to view the list of tasks for any date.
  - Alternatives considered:
    Wrapping of task name which will allow the display of the full task name. Not feasible as it will increase the number of rows further.

3.5. Find free time slots feature

3.5.1. Implementation

The `FreeCommand` class extends `Command` by providing methods to search for free time slots by determining if `Event` times stored in the persistent `TaskList` of the main program `OOF` clashes with a default time slot of 07:00 to 00:00 in the user specified date.

`TaskList` is stored internally as an `ArrayList` in the Oof Program as well as externally in persistent storage in `output.txt`.

All features are implemented in the parse method of the CommandParser class that parses user input commands.

Given below is an example usage scenario and how the `FreeCommand` class behaves at each step.

Step 1.
The user enters `free 30-10-2019`. The `parse` method in the `CommandParser` class is called to parse the input to obtain `30-10-2019` as the date to search for free time slots in.

`OofException` will be thrown if the user enters an invalid command.

Step 2.
The `execute` method in `FreeCommand` class is then called by the `Oof.run()` method in the main class `Oof`.

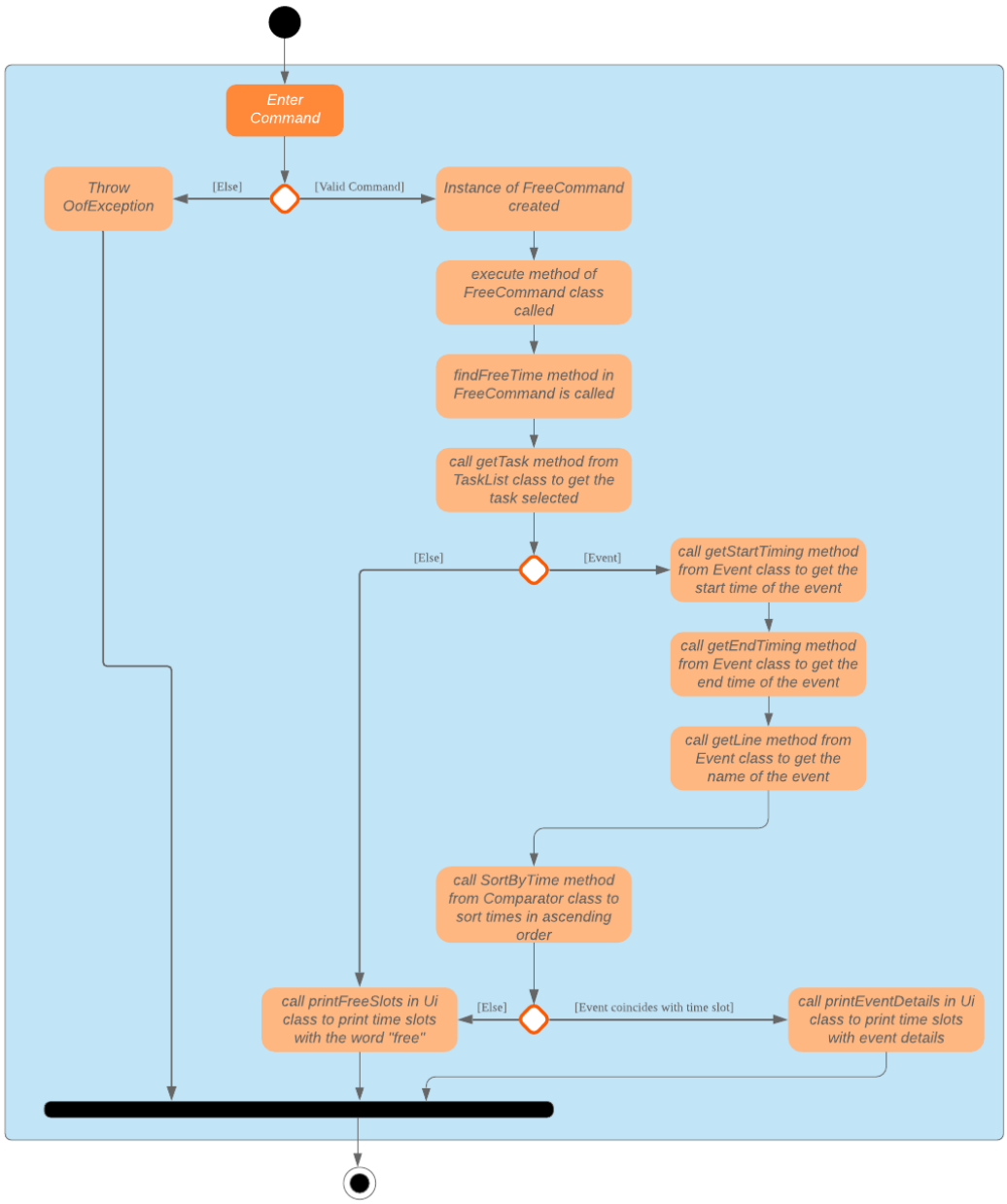
Step 3.
The `findFreeTime` method in `FreeCommand` class is then called by the `execute` method. This method does four main things:

- Finds all `Event` previously stored by the user and checks if it lies within the date given.
- Calls `sortByTime` method in `FreeCommand` class to sort all `Event` start and end times in ascending

order.

- Checks if an `Event` coincides with a time slot.
- Prints the time slots with the relevant details by:
  - Calling `printEventDescription` method in `Ui` class if `Event` coincides with the time slot
  - Calling `printFreeSlot` method in `Ui` class if `Event` does not coincide with a time slot.

The following activity diagram summarises what happens when a user executes a new command:


Enter Command
Throw OofException
[Else]
[Valid Command]
Instance of FreeCommand created
execute method of FreeCommand class called
findFreeTime method in FreeCommand is called
call getTask method from TaskList class to get the task selected
[Else]
[Event]
call getStartTiming method from Event class to get the start time of the event
call getEndTiming method from Event class to get the end time of the event
call getLine method from Event class to get the name of the event
call SortByTime method from Comparator class to sort times in ascending order
call printFreeSlots in Ui class to print time slots with the word "free"
[Else]
[Event coincides with time slot]
call printEventDetails in Ui class to print time slots with event details

3.5.2. Design Considerations

- **Selecting a single date to search free time slots in.**
    - Rationale:
      Allows the user to view which time slots they have free time in for a specific day so that they can quickly schedule team meetings.
    - Alternatives considered:
      Allow users to specify an end date in which they want search for free time slots up to instead of just a single date. Allowing users to do so will result in displaying unwanted time slots such as during hours where users are resting which would lead to redundant display of free time slots.
- **Displaying free time slots in hourly blocks.**
    - Rationale:
      This would give users a clean and easy view of the free time slots for that specific day.
    - Alternatives considered:
      Show free time slots in user-specified time blocks. This alternative can be an extension of the current implementation of the `FreeCommand` class.

3.6. Logging

3.7. Configuration

4. Documentation

5. Testing

6. Dev Ops

Appendix A: Product Scope

Target User Profile:

- Has a need to manage multiple tasks at once
- Prefer desktop Command-Line-Interface (CLI) over other types
- Able to type on the keyboard really fast
- Prefers typing over mouse input
- Proficient in using CLI applications

Value proposition: manage contacts faster than a typical mouse/GUI driven app

Appendix B: Requirements

B.1. User Stories

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

S/N	Use Case No	Priority Level	As a …	I can …	So that I can …
01	01	* * *	University Student	Add a task	Won't forget the tasks I have to complete
02	02	* * *	University Student	Mark a task as complete	Can keep track of what is left to be completed
03	03	* * *	University Student	View my tasks in a calendar	Can manage my time properly
04	04	* *	University Student	View a summary of tomorrow's task	Will know what to expect for the next day
05	05	* * *	University Student	Add an event with the relevant dates, start and end times	Can keep track of my upcoming appointments and examinations
06	06	* * *	University Student	Get reminders of deadlines due within 24 hours	Can prioritize those tasks to be completed first
07	07	* * *	University Student	Sort my tasks	Can see my tasks in chronological order
08	08	*	University Student	Find my tasks	Do not need to scroll through the entire calendar to find certain tasks
09	09	* *	Double degree University student	Color code the tasks	Can quickly distinguish different type of tasks
10	10	* *	University Student	View my tasks for the week	Can plan my time for the week
11	11	* * *	Busy University Student	Find free time slots	Will know which dates and times I am free to conduct project meetings
12	12	* * *	University Student	Cancel events	Keep my schedule updated
13	13	* * *	University Student	Postpone the deadline of tasks	Can properly manage my priorities

14	14	* *	University Student who procrastinates	View undone tasks carried forward to the next day in a bright color	Will know what assignments are lagging behind
15	15	* * *	University Student	Add a recurring task	Do not have to do it multiple times
16		*	Impatient University Student	Quickly type in one-liner commands	Can see the tasks being updated in the program quickly
17		*	University Student	View trends for my tasks	Can see if I am lagging behind
18		* *	Paranoid University Student	Choose the threshold before the programs sends an alert for me to complete my tasks	Can stay ahead of my schedule
19		*	Organized University Student	View all the tasks in a strict format	Will know what to type to enter my tasks
20		*	University Student in NUSSU	Export my calendar to a shareable format	Can quickly share my schedule with other people
21		* *	University Student	Have a do-after task	Know what tasks need to be done after completing a specific task
22		* * *	University Student	Have a task that needs to be done within a time period	Can better plan my schedule
23		*	University Student	Add my estimated time taken to complete a task	Know how much free time I would have
24		* *	Undergraduate Tutor	Have two instances of calendar	Can separate my tutor tasks and personal tasks
25		* *	University Student	Filter my calendar by different categories	Can view my tasks for that category easier
26		* * *	University Student	Add a tentative task	Can confirm it at a later date
27		* * *	University Student	View all commands	Do not need to memorise all the commands
28		* * *	University Student	Get warnings if an event I add clashes with an existing event	Will not have multiple events at the same time
29		*	University Student	Sync my tasks to my phone via bluetooth	Can view my tasks on the go and not just on my laptop

30		**	University Student	Print out my tasks stored	Can view my tasks even if my laptop runs out of battery

B.2. Use Cases

(MSS refers to Main Success Scenario.)

System: Outstanding Organization Friend (OOF)
Use case: UC01 - Add a task
Actor: User
MSS:

1. User wants to add a task.
2. OOF requests for description of the task.
3. User enters the description of the task.
4. OOF records the task and displays the description.

Use case ends.

Extensions:

- OOF detects empty date and time in description of task.
    - OOF requests for date and time of task.
    - User enters required data.
    - Use case resumes from step 4.
- OOF detects a clash in date and time with another task.
    - OOF warns the User of such a clah by displaying the task(s) that clash(es) and prompts for continuation or cancellation.
    - User decides for continuation or cancellation.
    - OOF requests to confirm decision.
    - User confirms decision.
    - Use case ends if the User decides to cancel the action. Use case resumes from step 4 otherwise.
- At any time, User chooses to re-enter task description.
    - OOF requests confirmation to re-enter task description.
    - User confirms to re-enter task description.
    - Use case resumes from step 3.

System: Outstanding Organization Friend (OOF)
Use case: UC02 - Mark a task as complete
Actor: User

MSS:

1. User wants to mark a task as complete.
2. OOF requests for index of task to mark as complete.
3. User enters the index of the task to mark as complete.
4. OOF records the task completion status and displays the description.

Use case ends.

Extensions:

- OOF detects non-existent index of task.
    - OOF requests for existent index and displays a range of indexes to choose from.
    - User enters required data.
    - Use case resumes from step 4.

System: Outstanding Organization Friend (OOF)
Use case: UC03 - View tasks in calendar
Actor: User
MSS:

1. User wants to view tasks in calendar format.
2. OOF requests for range of index of the tasks the user wishes to view in calendar format.
3. User enters the range of index of the task to view in calendar format.
4. OOF displays the tasks requested in calendar format.

Use case ends.

Extensions:

- OOF detects non-existent index of task in the range.
    - OOF requests for existent index and displays a range of indexes to choose from.
    - User enters required data.
    - Use case resumes from step 4.

System: Outstanding Organization Friend (OOF)
Use case: UC04 - View a summary of the next day's tasks
Actor: User
MSS:

1. User wants to view a summary of the next day's tasks.
2. OOF requests for user input.
3. User enters the summary command.
4. OOF displays the summary of the next day's tasks.

Use case ends.

Extension:

- OOF detects there are no tasks for the next day.
    - OOF prints to the console to warn User that there are no tasks for the next day.
    - Use case resumes from step 4.

System: Outstanding Organization Friend (OOF)
Use case: UC05 - Adding tasks with date and time
Actor: User
MSS:

1. User wants to add a task with date, start and end time.
2. OOF requests for description, date, start and end time of the task.
3. User enters the requested details.
4. OOF records the task and displays the task recorded.

Use case ends.

Extension:

- OOF detects an error with the entered data.
    - OOF requests for the correct data.
    - User enters new data.
    - Steps 3a1-3a2 are repeated until the data entered are correct.
    - Use case resumes from step 4.
- At any time, User choose to stop adding a task.
    - OOF requests to confirm the cancellation.
    - User confirms the cancellation.
    - Use case ends.

System: Outstanding Organization Friend (OOF)
Use case: UC06 - Reminder for expiring tasks (within 24hrs)
Actor: User
MSS:

1. User chooses to activate the reminder for expiring tasks.
2. OOF requests for confirmation of this action.
3. User confirms the action.
4. OOF displays the expiring tasks everytime OOF is started.

Use case ends.

Extensions:

- At any time, User chooses to cancel the activation.
    - OOF requests to confirm the cancellation.
    - User confirms the cancellation.
    - Use case ends.

System: Outstanding Organization Friend (OOF)
Use case: UC07 - Sort tasks in chronological order
Actor: User
MSS:

1. User requests to sort current tasks in chronological order.
2. OOF requests for confirmation of this action.
3. User confirms this request.
4. OOF sorts and displays the tasks in chronological order.

Use case ends.

Extensions:

- OOF detects that there are no tasks to be sorted.
    - OOF warns User that there are no tasks to be sorted
    - Use case ends.
- At any time, User chooses to cancel the request.
    - OOF requests to confirm the cancellation.
    - User confirms the cancellation.
    - Use case ends.

System: Outstanding Organization Friend (OOF)
Use case: UC08 - Find tasks
Actor: User
MSS:

1. User requests to find certain tasks.
2. OOF requests for the description of the tasks.
3. User enters a description of the tasks.
4. OOF displays the tasks that match the description.

Use case ends.

Extensions:

- OOF detects that there are no tasks that match the description given.

    - OOF requests for the User to enter a new description.
    - User enters a new description.
    - Steps 3a1-3a2 are repeated until at least one task matches the description.
    - Use case resumes from step 4.
- At any time, User chooses the stop finding tasks.
    - OOF requests to confirm the request.
    - User confirms the requests.
    - Use case ends.

System: Outstanding Organization Friend (OOF)
Use case: UC09 - Colour code tasks
Actor: User
MSS:

1. User requests to colour code tasks.
2. OOF displays the current tasks present in the program and prompts for the tasks to be colour coded and their respective colours to be coded.
3. User enters the required information.
4. OOF displays the current tasks present after colour coding the selected tasks.

Use case ends.

Extensions:

- OOF detects that there are no tasks to be colour coded.
    - OOF displays the warning that no tasks are available to be colour coded.
    - Use case ends.
- OOF detects an error in the information entered.
    - OOF prompts for User to enter the correct information.
    - User enters the correct information.
    - Steps 3a1-3a2 are repeated until the User enters in the correct information.
    - Use case resumes from step 4.
- At any time, User requests to cancel this action.
    - OOF requests to confirm the cancellation.
    - User confirms the cancellation.
    - Use case ends.

System: Outstanding Organization Friend (OOF)
Use case: UC10 - View tasks for the week
Actor: User
MSS:

1. User requests to view tasks for the week.
2. OOF requests to confirm the request.
3. User confirms the request.
4. OOF displays the tasks for the week.

Use case ends.

Extensions:

- OOF detects that there are no tasks for the week.
    - OOF warns the User that there are no tasks for the week.
    - Use case ends.
- At any time, User chooses to cancel this action.
    - OOF requests for confirmation.
    - User confirms the requests.
    - Use case ends.

System: Outstanding Organization Friend (OOF)
Use case: UC11 - Find free time slots
Actor: User
MSS:

1. User requests to find free time slots.
2. OOF requests for the time period from the User.
3. User enters in the time period of interest.
4. OOF displays the free time slots within the time period.

Use case ends.

Extensions:

- OOF detects that the time period entered is invalid.
    - OOF requests for the User to input a valid time period.
    - User enters a valid time period.
    - Steps 3a1-3a2 are repeated until a valid time period is entered.
    - Use case resumes from step 4.
- At any time, User chooses to cancel the action.
    - OOF requests for confirmation.
    - User confirms the request.
    - Use case ends.

System: Outstanding Organization Friend (OOF)

Use case: UC12 - Delete tasks
Actor: User
MSS:

1. User requests to delete tasks.
2. OOF lists the current tasks saved in the program and prompts User to select the task to be deleted.
3. User chooses the task to be deleted.
4. OOF deletes and display the task that was deleted and the number of tasks saved in the program.

Use case ends.

Extensions:

- OOF detects that there are no tasks saved in the program.
    - OOF warns the User that there are no tasks to be deleted.
    - Use case ends.
- OOF detects an error in the task that was selected by the User.
    - OOF prompts the user to enter a valid input.
    - User enters a valid input.
    - Steps 3a1-3a2 are repeated until the User enters a valid input.
    - Use case resumes from step 4.
- At any time, User chooses to cancel the action.
    - OOF requests for confirmation from the User.
    - User confirms the cancellation.
    - Use case ends.

System: Outstanding Organization Friend (OOF)
Use case: UC13 - Postpone tasks
Actor: User
MSS:

1. User requests to postpone a task.
2. OOF displays the current tasks saved in the program and prompts the User the indicate the task to be postponed and its postponed date.
3. User enters the task and the postponed date.
4. OOF displays the task that was postponed with its new deadline.

Use case ends.

Extensions:

- OOF detects that there are no tasks saved in the program.
    - OOF warns the User that there are no tasks to be postponed.
    - Use case ends.
- OOF detects an error in the task that was selected by the User.
    - OOF prompts the user to enter a valid input.
    - User enters a valid input.
    - Steps 3a1-3a2 are repeated until the User enters a valid input.
    - Use case resumes from step 4.
- At any time, User chooses to cancel the action.
    - OOF requests for confirmation from the User.
    - User confirms the cancellation.
    - Use case ends.

System: Outstanding Organization Friend (OOF)
Use case: UC14 - Overdue tasks
Actor: User
MSS:

1. User requests to highlight tasks that are overdue.
2. OOF requests to confirm the request.
3. User confirms the request.
4. OOF displays the overdue tasks

Use case ends.

Extensions:

- OOF detects that there are no overdue tasks.
    - OOF warns the User that there are no overdue tasks.
    - Use case ends.
- At any time, User chooses to cancel the activation.
    - OOF requests to confirm the cancellation.
    - User confirms the cancellation.
    - Use case ends.

System: Outstanding Organization Friend (OOF)
Use case: UC15 - Recurring tasks
Actor: User
MSS:

1. User chooses to add recurring tasks.

2. OOF displays the current tasks saved in the program and prompts the User to input the task that is recurring and its respective frequency.
3. User enters the task and recurring frequency.
4. OOF displays the task selected and automatically adds the recurring task at relevant time intervals.

Use case ends.

Extensions:

- OOF detects that there are no tasks saved in the program.
    - OOF warns the User that there are no tasks to be marked as recurring.
    - Use case ends.
- OOF detects an error in the task that was selected by the User.
    - OOF prompts the user to enter a valid input.
    - User enters a valid input.
    - Steps 3a1-3a2 are repeated until the User enters a valid input.
    - Use case resumes from step 4.
- At any time, User chooses to cancel the action.
    - OOF requests for confirmation from the User.
    - User confirms the cancellation.
    - Use case ends.

B.3. 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 200 tasks/events without performance deterioration
3. A user with above average typing speed for regular English Text should be able to store their tasks faster using commands than using the mouse

Appendix C: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Appendix D: Instructions for Manual Testing