COMPal - Developer Guide

By: `AY1920S1-CS2113T-W17-1` Last Updated: `26/09/2019` License: `MIT`

Table of Contents

1. Introduction

COMPal is a desktop application specifically designed for the **hectic schedule** for the **modern student** in mind. By **simply inputting** their busy and compact schedule, the application is able to automatically **generate a prioritized daily schedule** for the user! This ensures that the student can **focus** on the more **important upcoming task**! Additionally with **features such as** reminders of task and also finding of free time slot, COMPal allows the **ease of planning** for future task.

It is catered to student-users who prefer to use and are adept at using a **Command-Line Interface (CLI)**, while still having a clean **Graphical User Interface (GUI)** to properly **visualize schedules** and **organize tasks** better.

2. About This Developer Guide

This **Developer Guide** provides a detailed documentation on the implementation of all the time-management tools of **COMPal**. To navigate between the different sections, you could use the **Table of Contents** above.

For ease of communication, the following **terminology** will be used:

Term	**Definition**
Task	The general term that is used to describe an action that might need to be done by the user.

Additionally, throughout this **Developer Guide**, there will be various **icons** used as described below.

Icon	**Description**
ℹ	Additional important information about a term/concept
💡	A tip that can improve your understanding about a term/concept
⚠	A warning that you should take note of

3. Setting Up

3.1. Prerequisites

1. **JDK 11** or later
2. **IntelliJ** IDE

3.2. 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 your existing project dialogue first)
3. Set up the correct **JDK** version for Gradle
    1. Click Configure > Project Defaults > Project Structure
    2. 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. Use Gradle to run the project.
    1. Click on the small Gradle icon at the top right of your screen. It should open up the Gradle sidebar.
    2. Click on the small Gradle icon at the centre of the Gradle sidebar.
    3. Type gradle run to run the project.
    4. The **GUI** should show up in a few seconds. Try running a few commands.
9. Observe for any code errors displayed in the **console** of the **IntelliJ** IDE.

3.3. Verifying the Setup

1. Run the project using gradle run. Try a few commands in the **GUI**.

2. Run the tests to ensure that they all pass.

3.4. Configurations to do before Writing Code

4. Design

4.1. Architecture

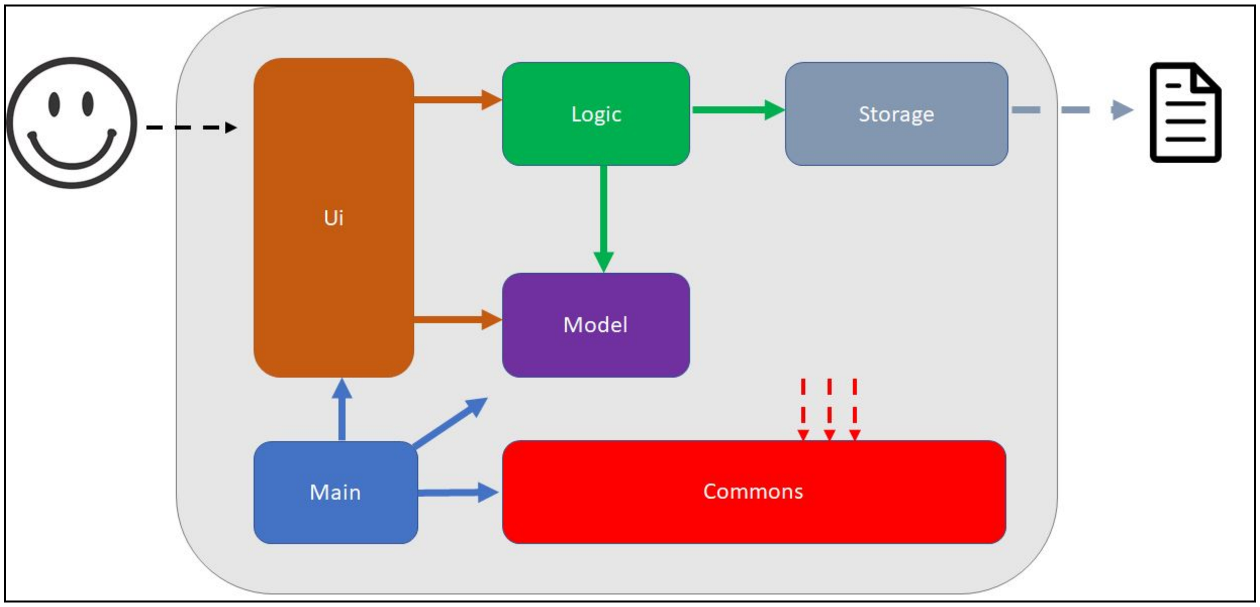

Figure 1. Architecture Diagram

💡	The .pptx files used to create diagrams in this document can be found in the diagrams folder. To update a diagram, modify the diagram in the pptx file, select the objects of the diagram, and choose Save as picture.

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

Commons represents a collection of classes used by multiple other components. Two of those classes play important roles at the architecture level.

- Messages : This class contain all types of messages that are to be called from depending on the type of execution. E.g. Invalid syntax commands messages

- COMPal : Used by many classes to call the needed functionality such as user interface, storage or parser.

The rest of the App consists of four components.

- UI: The UI of the App.
- Logic: The command executor.
- Model: Holds the data of the App in-memory.
- Storage: Reads data from and writes data to the hard disk.

For example, the Parser component (see the class diagram given below) defines it's API in the CommandParser.java interface and exposes its functionality using the ParserManager.java class.

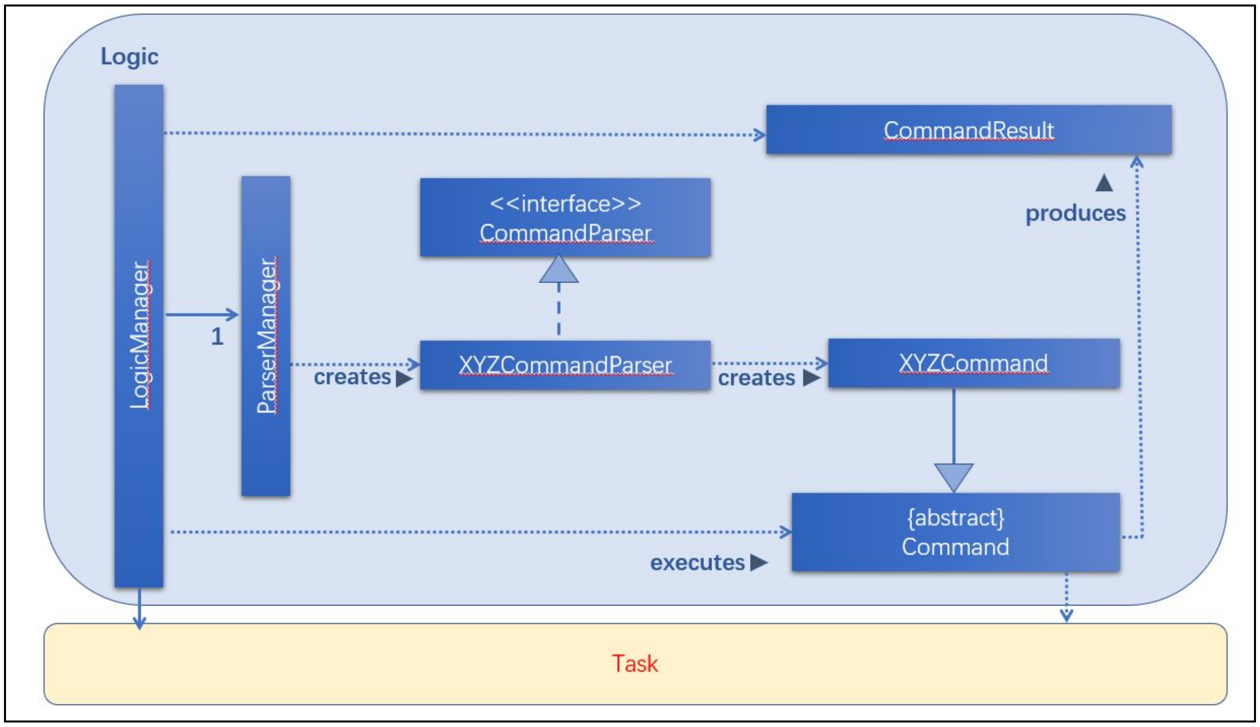

Figure 2. Class Diagram of Logic Parser Component

Events-Driven nature of the design

The *Sequence Diagram* below shows how the components interact for the scenario where the user issues the command delete 1.

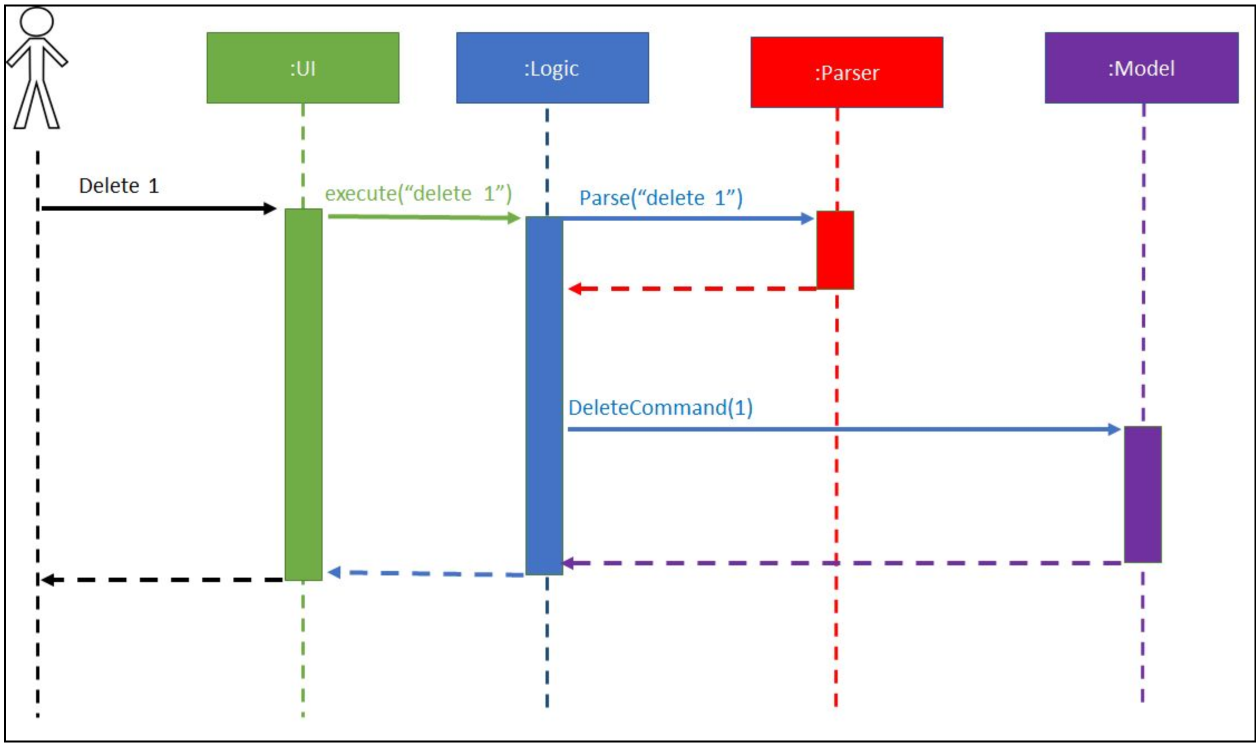

Figure 3. Component interactions for `delete 1` command.

The sections below give more details of each component.

4.2. UI component

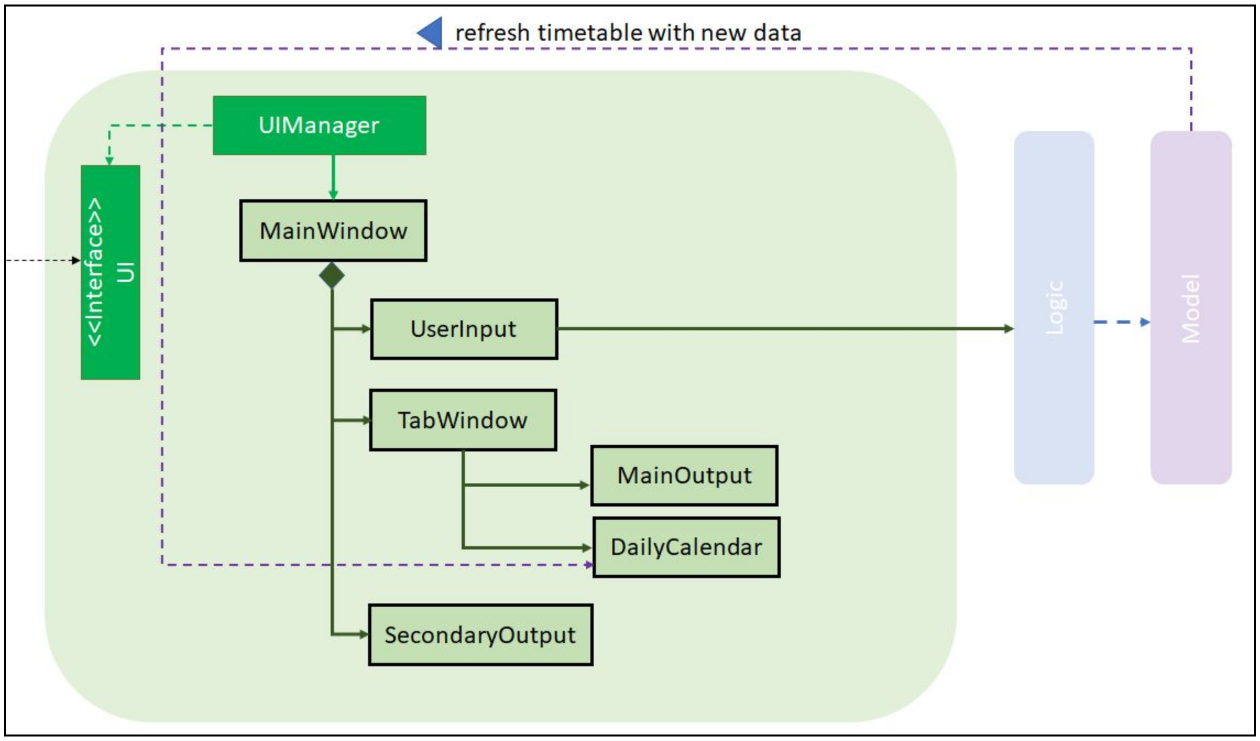

Figure 4. Structure of the UI Component

API : Ui.java

The UI consists of a MainWindow that is made up of parts e.g.UserInput,SecondaryOutput, tabWindowwhich tabs consist of MainOutput, DailyCalender. Although the application is only input text-based application, our outputs are both GUI and text-based.

The UI component uses JavaFx UI framework. The layout of these UI parts are defined in matching .fxml files that are in the src/main/resources/view folder. For example, the layout of the MainWindow is specified in MainWindow.fxml

The DailyCalender use information from the Model and COMPal component to generate or refresh the stage to reflect changes made to the data.

The UI component,

- Executes user commands using the Logic component.
- Displays text-based command results in to the user via MainOutput or SecondaryOutput.
- Display daily calendar of the user via DailyCalender.

4.3. Logic component

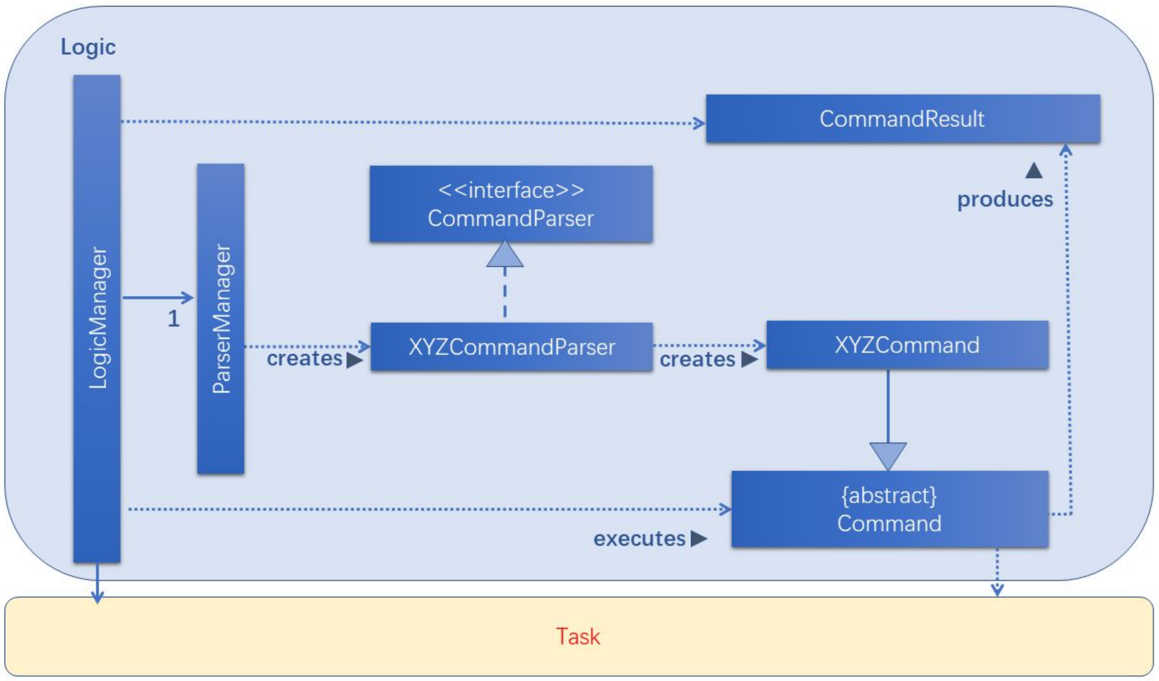

Figure 5. Structure of the Logic component

The Logic component handles the parsing of user input and interacts with the Task objects.

1. Uses the CommandParser class to parse user input.

    a. This results in a Command object which is executed.
2. The execution of Command can affect a Task object (e.g. adding a Task to the TaskList)
3. The result of the Command execution is encapsulated as a CommandResult object which is passed to the UI to be rendered as output for the user.

A Sequence Diagram for interactions within the Logic component will be uploaded soon.

4.4. Commons Component

Classes used by multiple components are in the commons package. It contains 2 important classes: Compal and Messages.

Compal.java creates an instance of the Ui, Storage, TaskList and ParserManager. Other classes will then use Compal to call on the aforementioned classes for different method invocations.

In addition, Compal contains the viewReminder method, which will be called when the GUI is initialised. This provides the user with the reminders set or due within 7 days.

Messages.java contains all the error messages that will be printed on the GUI when the user has made an error in their input. This will notify the user to check what he/she has keyed in the command box, and make necessary adjustments.

4.5. Storage Component

API: StorageManager.java

We use very simple and user-editable text files to store user data. Data is stored as data strings separated by underscores. The separation token however, can be easily changed if desired. Data is thereafter parsed as a string and then processed by our storage API into application-useful datatypes such as Task Objects.

While it might be viewed as primitive, the advantage of this approach is that it is an almost no-frills implementation and is easily comprehended by the average developer. The average user can also understand and easily directly edit the data file if so desired.

4.6. Model Component

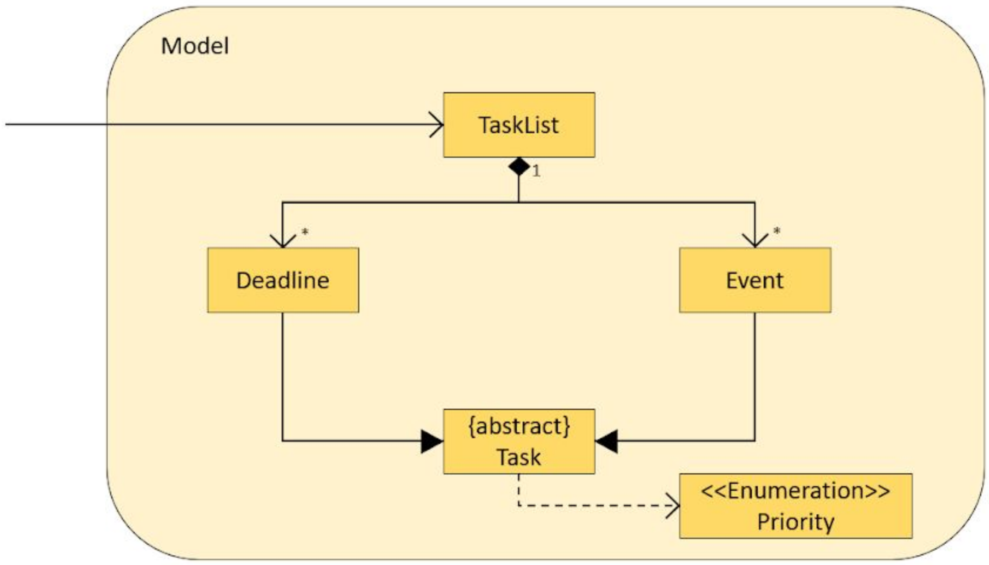

Figure 6. Overall structure of the Model Component

API: Model

The Model component

- stores a TaskList object that represents the list of user's tasks
- stores the Schedule data.
- does not depend on any of the other four components.

5. Implementation

5.1. View feature

This feature presents the timetable in text and daily calendar formats to the user.

The available formats are the day view, week view, and the month view. This section will detail how this feature is implemented.

5.1.1. Current Implementation

5.1.1.1 Command: `view`

Upon invoking the `view` command with valid parameters (refer to UserGuide.md for `view` usage), a sequence of events is then executed.

For clarity, the sequence of events will be in reference to the execution of a `view day` command. A graphical representation is included in the Sequence Diagram below for your reference when following through the sequence of events. The sequence of events are as follows:

1. The `view day` command is passed into the `logicExecute` function of `LogicManager` to be parsed.
2. `LogicManager` then invokes the `processCmd` function of `ParserManager`.
3. `ParserManager` in turn invokes the `parseCommand` function of the appropriate parser for the view command which in this case, is `viewCommandParser`.
4. Once the parsing is done,`ViewCommandParser` would instantiate the `ViewCommand` object which would be returned to the `LogicManager`.
5. `LogicManager` is then able to invoke the `commandExecute` function of the returned `ViewCommand` object.
6. In the `execute` function of the `ViewCommand` object, task data will be retrieved from the `TaskList` component.
7. Now that the `ViewCommand` object has the data of the current task of the user, it is able to invoke the `displayDailyView` method.
8. With the output returned from the `displayDailyView`, the `CommandResult` object will be constructed.

⚠	Only the `view day` command will create a `daily task:<DATE>` tab which consists of the daily schedule of the user.

9. Only if the user input `view day,`will the `CalenderUtil` object will be constructed and invoke a `dateViewRefresh` to create the `daily Calendar tab` represented in GUI format.

10. The `CommandResult` object would then be returned to the `LogicManager`, which then returns the same `CommandResult` object back to the `UI` component.

11. Finally, the `UI` component would display the contents of the `CommandResult` object to the user. For this `view day` command, the displayed result would be the daily task view of the current day in both text and GUI format..

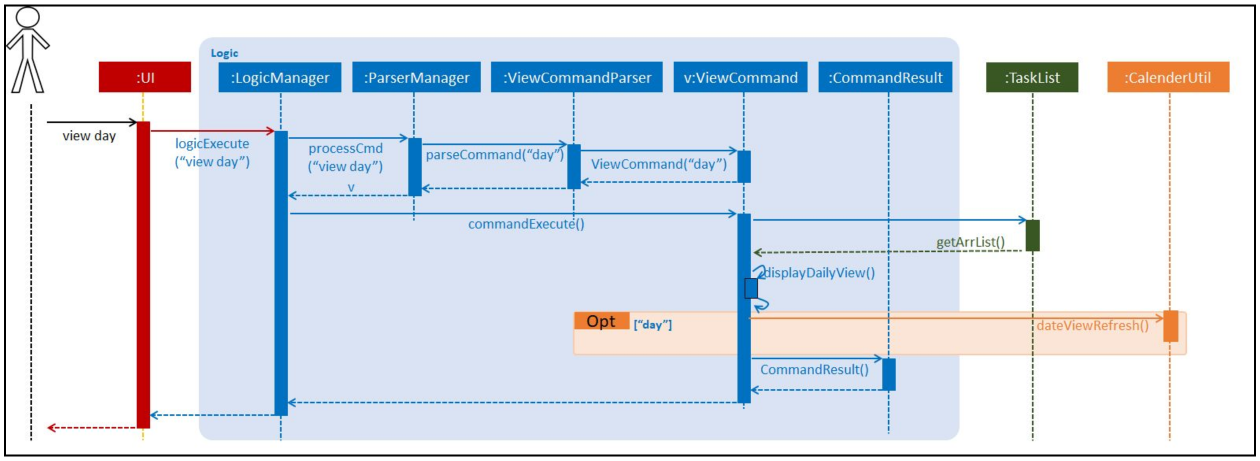

Figure 7. Sequence Diagram executing the `view day` command.

The 3 general types of view are generated by the methods `displayDayView`, `displayWeekView`, `displayMonthView` from the `ViewCommand` class and the implementation of these methods is explained below.

ℹ	If a given date is not input by the user, COMPal will deem that specified date is the current date of the user system.

`ViewCommand#displayDayView()` method displays the details of all the *task* of a specified date. The implementation of this method can be broken down into 3 parts:

1. Retrieve all *tasks* for the specified day.
2. Prints the daily header of *day,date* (e.g Tue,22/10/2019).
3. Print all the details of each *task* found in chronological time and priority order.

`ViewCommand#displayWeekView()` method displays the weekly calendar format of a specified week. The implementation of this method can be broken down into the following steps:

1. From the current date or input date, determine the next 6 days of the calendar days.
2. Prints the week header (e.g. 21/10/2019 - 27/10/2019).
3. Invoke displayDayView method for the week given.

`ViewCommand#displayMonthView()` method displays the current month in a monthly calendar format. The implementation of this method can be broken down into 2 parts:

1. From the current date or input date, determine the month and amount of days in a month.
2. Print the month header (e.g. January 2020)
3. Invoke displayDayView method for each day in a given month.

5.1.1.2 `Daily view` of calendar schedule for `view day`

Upon invoking the `view day` command the daily task, the daily calendar schedule for the user will be created. The daily calender GUI logic could be found in DailyCalUi.Java .This section uses JavaFX components along with the user stored task to create the daily calendar.

For clarity, the sequence of events will be in reference to the execution of a `view day /date 22/10/2019` command. Additionally, a graphical representation is included in the activity diagram below for your reference when following through the sequence of events in the creation of the daily calendar. The sequence of events are as follows:

1. Using the `TaskStorageManager` class,loads the stored task of the user.
2. Calls `CreateDailyArrayList` method to create a daily task list of user of *tasks* that matches the date 22/10/2019 and tasks that are not marked as done. Additionally, the

list will be sorted by descending priority of high to medium to low.

3. Calls `buildTimeTable` method, which in order of invocation :
    3.1. Set the start time and end time of the daily calendar using the `setTime` method.
        3.1.1. By default the start time is set to 8 AM. However, if the user has any tasks that starts before 8 AM, the start time of the daily calendar is set to the earliest hour of the daily tasks.
        3.1.2. By default the end time is set to 5 PM. However, if the user has any tasks that starts after 5 PM, the end time of the daily calendar is set to the latest hour of the daily tasks.

    3.2. Prints date 22/10,2019 on the top left corner of the daily task tab using the `genDateSlot` method using .

    3.3. Create and print deadlines of the user's which is formatted in JavaFX `rectangle` using the `buildDeadline` method.
        3.3.1. The rectangle consists of the deadline information and the due date of the deadline.

⚠	Only the `5` deadline or task per time slot can be printed out to ensure that user focus on the higher priority tasks at hand.

    3.4. Using the `genTimeSlot,makeHorizontalLines and makeVerticalLines` method, to create the necessary daily schedule of the user.
        3.4.1. For each time slot (e.g 8 AM), check existence of task and mark that a slot has been added for that hour until the end of task. There can only be at most 5 clashes for each hourly slot using `drawScheduleSquare` method.
        3.4.2. This step is repeated for each time slot until the set end time of the day.

    3.5. Using the data concluded from the previous method invocation , drawScheduleSquare will then draw the necessary schedule of the user. Which is output in JavaFX rectangle which contains the priority,description and status of the task.

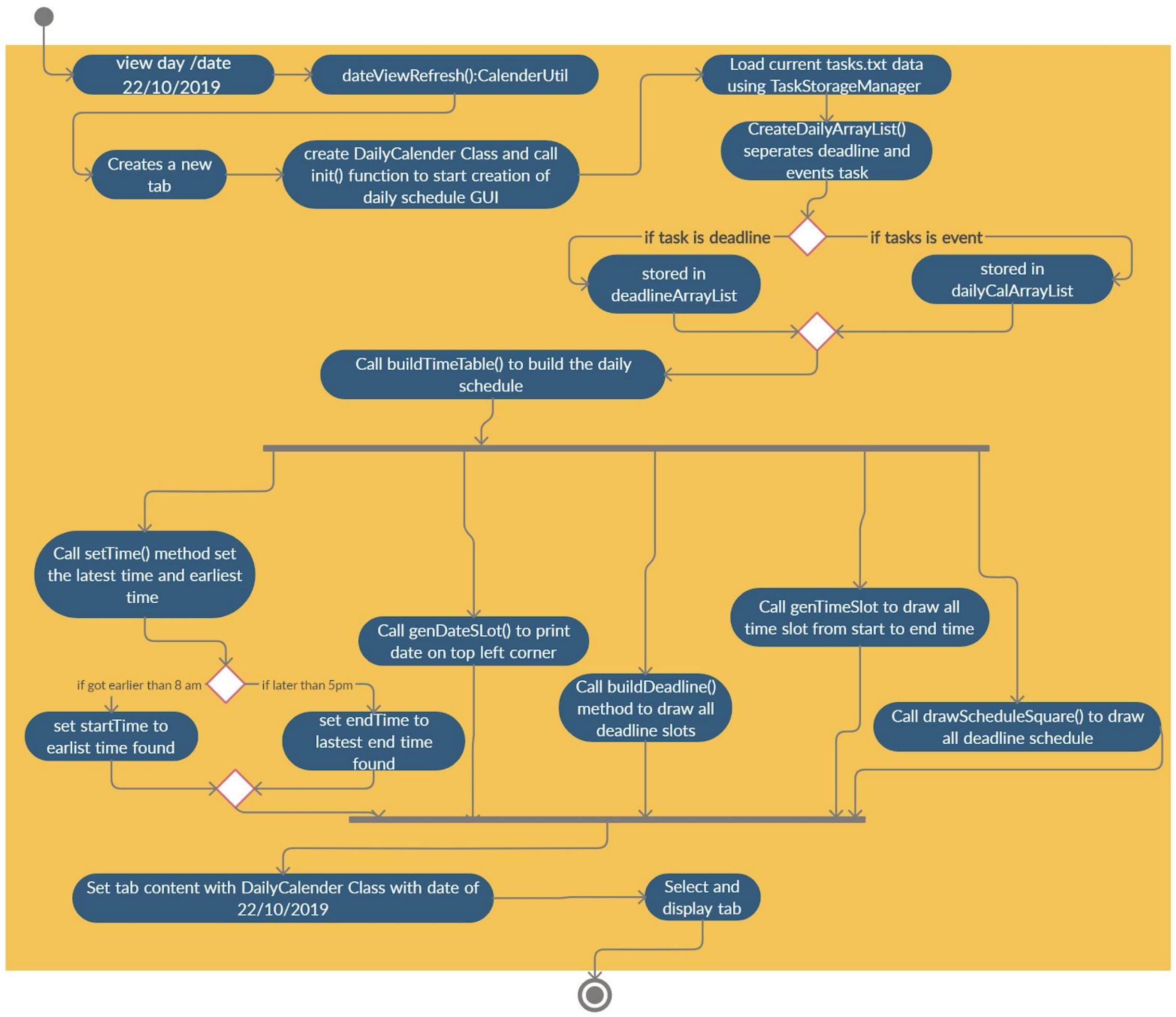
view day /date 22/10/2019
dateViewRefresh():CalenderUtil
Load current tasks.txt data using TaskStorageManager
Creates a new tab
create DailyCalender Class and call init() function to start creation of daily schedule GUI
CreateDailyArrayList() seperates deadline and events task
if task is deadline
if tasks is event
stored in deadlineArrayList
stored in dailyCalArrayList
Call buildTimeTable() to build the daily schedule
Call setTime() method set the latest time and earliest time
Call genDateSLot() to print date on top left corner
Call buildDeadline() method to draw all deadline slots
Call genTimeSlot to draw all time slot from start to end time
Call drawScheduleSquare() to draw all deadline schedule
if got earlier than 8 am
if later than 5pm
set startTime to earlist time found
set endTime to lastest end time found
Set tab content with DailyCalender Class with date of 22/10/2019
Select and display tab

Figure 8. Activity Diagram for the `view day` command,which creates `view Task:<Date>` tab.

5.1.2. Design Considerations

This section details our considerations for the implementation of the `view` feature.

Aspect: Functionality of `view day` command

- **Alternative 1 (current choice):** Implement **singleton design pattern** software design consideration for DailyCalUi.java .
    - Pros: Since COMPal design pattern is following the **Command design pattern,** it would be more feasible to create the Calendar class as a singleton class as it will only be instantiate when the user input **`view day`**.
    - Cons: Requires user to key in command for the interface to be updated rather than automatic update when an event is updated on a particular date.
- **Alternative 2 :** Implement **Observer Pattern** software design consideration to observer for adding of events or deadline on particular dates.
    - Pros: Automatic update of the interface since the system will update the users Daily Scheduler when a new event or deadline is added based on observation.
    - Cons: Observer Pattern may cause COMPal to demand high usages of resources it can easily add complexity to the code which lead to performance issues. Additionally, notifications can be unreliable and may result in race conditions or inconsistency as there might be many events to observe.

Alternative 1 was chosen as the benefits of a singleton design pattern which instantiate the DailyCalUi.Java **only** when invoked during the **`view day`** command outweigh the disadvantages involving adding complexity to the code which may lead to COMPal have a high system resource usage and affecting its performance.

5.1.3 Future Implementation

Though the current implementation has much prevent cluttering the application which allow the users to focus and prioritize on the upcoming tasks at hand. There are still possible enhancement for the view command.

1. GUI for `weekly view` tab of *task* in a weekly schedule GUI format
2. GUI for `monthly view` tab of *task* in a monthly schedule GUI format

5.2. Task Management

This feature involves mainly the interaction between users and their Tasks. This section will detail how this feature is implemented.

5.2.1. Current Implementation

COMPal accepts two types of **Tasks**:

1. **Deadlines** refer to **Tasks** that users have to do by a **specified time** by a **specified date**.
2. **Events** refer to **Tasks** that users have to do in a **specific fixed duration** on a **specified date**.

Users can interact with their **Tasks** using the following commands, together with a system of parameter keywords:

1. deadline
    a. Add a single **deadline** without using the optional /final-date parameter keyword
    b. Add multiple **deadlines** using the optional /final-date parameter keyword
2. event
    a. Add a single **event** without using the optional /final-date parameter keyword
    b. Add multiple **events** using the optional /final-date parameter keyword
3. edit
    a. Edit the attributes of the **Task**, using parameter keywords relevant to the **Task**

Since **deadlines** and **events** have some differences, they share some common parameter keywords and differ in others, as illustrated in the table below.

Table X.X: Parameter keywords and their descriptions.

Parameter Keyword	Description
/date	**Deadline**: refers to the date that the **Task** has to be completed by
	Event: refers to the date that the **Task** is happening on
/start	**Deadline**: The deadline command does not accept this keyword
	Event: refers to the start time of the **Task**, because it has a fixed duration.
/end	**Deadline**: refers to the time that the **Task** has to be completed by
	Event: refers to the end time of the **Task**, because it has a fixed duration.

/priority	Optional parameter keyword. Refers to the priority of the **Task** (either **Deadline** or **Event**). This feature will be further elaborated on in **5.3 Priority**.
/final-date	Optional parameter keyword, used to support the addition of multiple **deadlines** / **events**.

Command: `deadline`

Upon invoking the `deadline` command with valid parameters (refer to [UserGuide.md]() for `deadline` usage), a sequence of events is then executed.

For clarity, the sequence of events will be in reference to the execution of a `deadline a_deadline /end 1000 /date 09/08/2019` command. A graphical representation is included in the Sequence Diagram below for your reference when following through the sequence of events. The sequence of events are as follows:

1. The `deadline a_deadline /end 1000 /date 09/08/2019` command is passed into the `logicExecute` function of `LogicManager` to be parsed.
2. `LogicManager` then invokes the `processCmd` function of `ParserManager`.
3. `ParserManager` in turn invokes the `parseCommand` function of the appropriate parser for the `deadline` command which in this case, is `DeadlineCommandParser`.
4. Once the parsing is done, `DeadlineCommandParser` would instantiate the `DeadlineCommand` object with arguments obtained from parsed user input. The `DeadlineCommand` will be returned to the `LogicManager`.
5. `LogicManager` is then able to invoke the `commandExecute` function of the returned `DeadlineCommand` object.
6. In the `commandExecute` function of the `DeadlineCommand` object, a `Deadline` object will be instantiated using the existing arguments in the `DeadlineCommand` object.
7. Now that the `Deadline` object has the data of the current task of the user, it will be added to the `TaskList` component.
8. With the output returned from the `Deadline` object, the `CommandResult` object will be instantiated.
9. The `CommandResult` object would then be returned to the `LogicManager` which then returns the same `CommandResult` object back to the `UI` component.
10. Finally, the `UI` component would display the contents of the `CommandResult` object to the user.

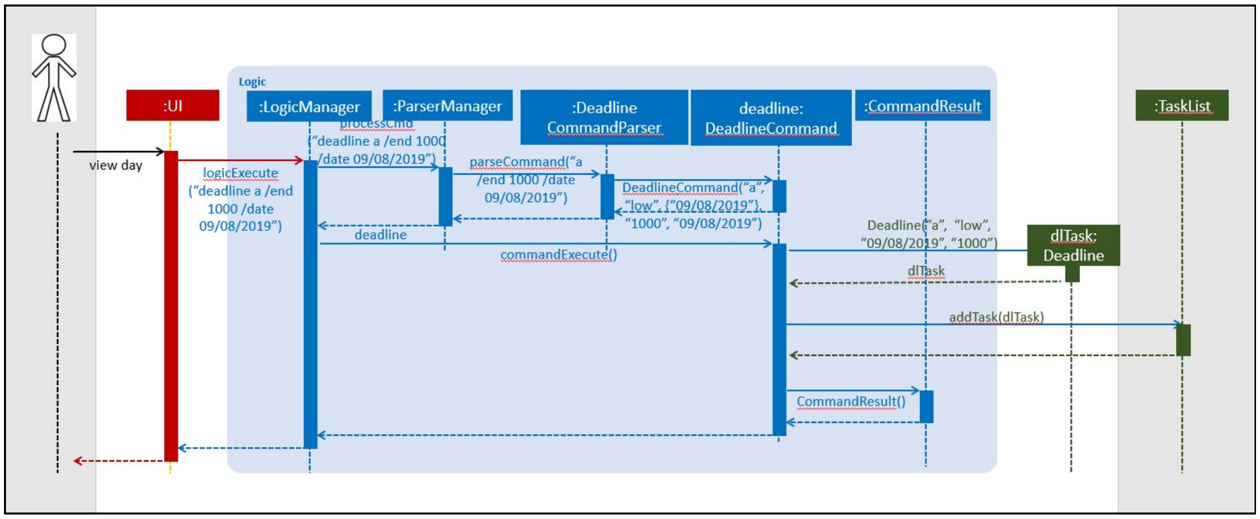
Logic
:UI
:LogicManager
:ParserManager
:Deadline CommandParser
deadline: DeadlineCommand
:CommandResult
:TaskList
view day
logicExecute ("deadline a /end 1000 /date 09/08/2019")
processCmd ("deadline a /end 1000 /date 09/08/2019")
parseCommand("a /end 1000 /date 09/08/2019")
DeadlineCommand("a", "low", ["09/08/2019"], "1000", "09/08/2019")
deadline
commandExecute()
Deadline("a", "low", "09/08/2019", "1000")
dlTask: Deadline
dlTask
addTask(dlTask)
CommandResult()

Figure 9: Sequence Diagram executing `deadline` command

5.2.2. Design Considerations

This section describes what we considered during the implementation of the Task Management feature. Choices were made mainly due to the need for better user experience and ease of use.

Aspect: Handling order of parameters

- **Alternative 1 (current choice):** Parse parameters without regard to order
    - **Pros:** greater user experience as user does not need to remember the exact order of parameters
    - **Cons:** More computationally expensive and tougher development process due to accounting for more test cases
- **Alternative 2:** Accept only a specific ordering of parameters
    - **Pros:** less computationally expensive
    - **Cons:** hinders user experience as user needs to remember the exact order of parameters

Aspect: Handling missing parameters

- **Alternative 1 (current choice):** allow for optional parameters such as /final-date and /priority, and use defaults when needed
    - **Pros:** greater user experience, user may not need to enter parameters that they do not need
    - **Cons:** Tougher development process due to accounting for more test cases
- **Alternative 2:** require all parameters to be compulsory
    - **Pros:** easier development of current commands
    - **Cons:** hinders user experience as user needs to enter every parameter, even if they do not require them

        - **Cons:** requires a different command to support the addition of multiple recurring Tasks

5.2.3. Future Implementation

1. Implementing a **Task** that can extend over multiple days

5.3. Priority Feature

This feature allows user to set priority for their tasks. Priorities will influence the order shown on the timetable when **view** functions are called. This section will detail how this feature is implemented.

5.3.1. Current Implementation

All **Tasks** must have priorities. Each task much have one of the three priority levels (**low, medium, high).**

The priority of a task can be set by two ways:

1. Set the priority of a task during addition of tasks (Refers to User Guide for more information about how to add a task (either **Deadline** or **Event**) with **/priority** tag). If the user does not set the priority during addition, its priority level will by default set to **low**.
2. Set the priority of a task using edit function (Refers to User Guide for more information about how to edit a task (either **Deadline** or **Event**) with **/priority** tag).

The priority will influence their order on the timetable and task list when **view** methods are called (Refers to view for more information) A task with higher priority will be at the left of the timetable in the schedule GUI format and at the top of a task list shown on main window tab.

5.3.2. Design Considerations

This section describes what we considered during the implementation of the Priority feature.

Aspect: Functionality of showing tasks with higher priority first when showing timetable

- **Alternative 1 (current choice):** show tasks with higher priority at the left part of the timetable in the schedule GUI format and at the top of the task list on main window tab in text format
  - Pros: User can see these tasks first when they open the timetable in the schedule GUI format
  - Cons: Tasks with relatively lower priority may not appear on the timetable
- **Alternative 2:** show tasks with higher priority at the top of the task list on main window tab in text format
  - Pros: All tasks will be shown in the list
  - Cons: User cannot see the timetable and get an intuitive image of task priority levels

Alternative 1 was chosen as it is more intuitive for the user to understand the different priority levels between tasks while it keeps the task list on the main window so the user can still see the overview list of all tasks.

5.3.3. Future Implementation

1. Implement a timetable which can show more tasks in the schedule GUI format.

5.4. Reminder feature

This feature allows users to keep track of undone tasks that are urgent or important.

Undone tasks that are due within the week and overdue tasks are preset to be included. Additionally, users can manually turn on reminders for important tasks they want to keep track of.

- To manually turn on/off reminders, the format is `set-reminder /id <TASK ID> /status <Y/N>`. This edits the reminder settings of the task with the specified task ID to the specified status. (on/off)
- To view urgent and important tasks, the format is `view-reminder`. This displays the list of undone tasks that are either overdue, due within the week, or have the reminder settings turned on.

This section will detail how this feature is implemented.

5.4.1. Current Implementation

Command: `set-reminder`

Upon invoking the `set-reminder` command with valid parameters (refer to UserGuide.adoc for `set-reminder` usage), a sequence of events is then executed.

For clarity, the sequence of events will be in reference to the execution of a `set-reminder /id 1 /status Y` command. A graphical representation is included in the Sequence Diagram below for your reference when following through the sequence of events. The sequence of events are as follows:

1. The `set-reminder /id 1 /status Y` command is passed into the `logicExecute` function of `LogicManager` to be parsed.
2. `LogicManager` then invokes the `processCmd` function of `ParserManager`.
3. `ParserManager` in turn invokes the `parseCommand` function of the appropriate parser for the `set-reminder` command which in this case, is `SetReminderCommandParser`.
4. Once the parsing is done, `SetReminderCommandParser` would instantiate the `SetReminderCommand` object which would be returned to the `LogicManager`.
5. `LogicManager` is then able to invoke the `commandExecute` function of the returned `SetReminderCommand` object.
6. In the `commandExecute` function of the `SetReminderCommand` object, task data will be retrieved from the `TaskList` component.
7. Now that the `SetReminderCommand` object has the data of the current task of the user, it is able to invoke the `setHasReminder` method.

8. With the output returned from the `setHasReminder`, the `CommandResult` object will be instantiated.
9. The `CommandResult` object would then be returned to the `LogicManager` which then returns the same `CommandResult` object back to the `UI` component.
10. Finally, the `UI` component would display the contents of the `CommandResult` object to the user.

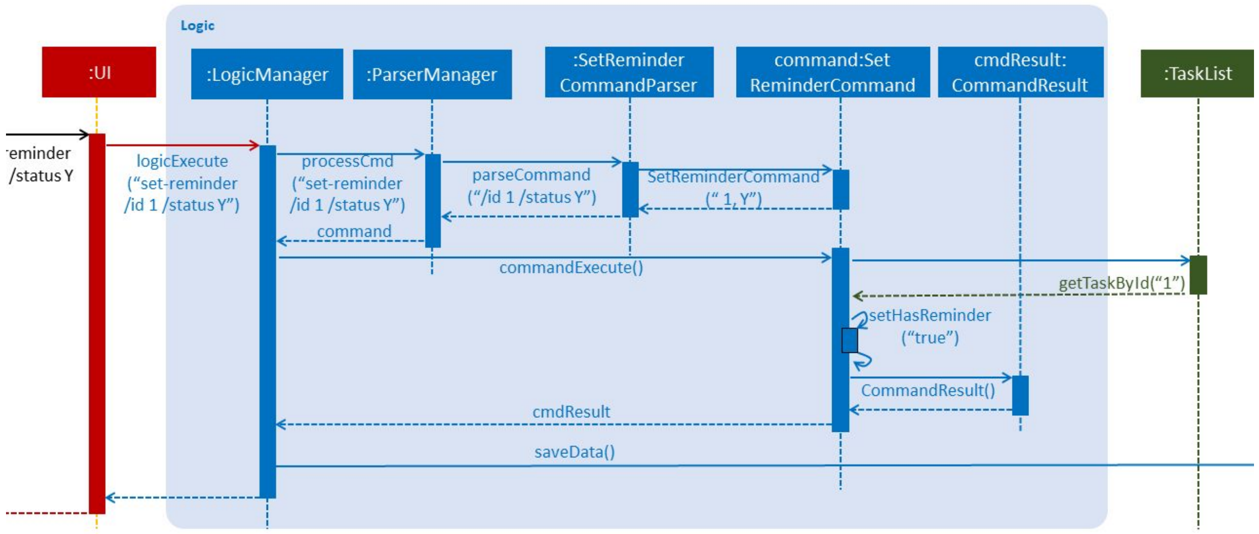
Logic
:UI
:LogicManager
:ParserManager
:SetReminder CommandParser
command:Set ReminderCommand
cmdResult: CommandResult
:TaskList
eminder /status Y
logicExecute ("set-reminder /id 1 /status Y")
processCmd ("set-reminder /id 1 /status Y")
parseCommand ("/id 1 /status Y")
SetReminderCommand (" 1, Y")
command
commandExecute()
getTaskById("1")
setHasReminder ("true")
CommandResult()
cmdResult
saveData()

Figure 10. Sequence Diagram executing the **`set-reminder`** command.

Command: `view-reminder`

Upon invoking the `view-reminder` (refer to UserGuide.adoc for `view-reminder` usage), a sequence of events is then executed.

A graphical representation is included in the Sequence Diagram below for your reference when following through the sequence of events. The sequence of events are as follows:

1. The `view-reminder` command is passed into the `logicExecute` function of `LogicManager` to be parsed.
2. `LogicManager` then invokes the `processCmd` function of `ParserManager`.
3. `ParserManager` in turn invokes the `parseCommand` function of the appropriate parser for the `view-reminder` command which in this case, is `ViewReminderCommandParser`.
4. Once the parsing is done, `ViewReminderCommandParser` would instantiate the `ViewReminderCommand` object which would be returned to the `LogicManager`.

5. `LogicManager` is then able to invoke the `commandExecute` function of the returned `ViewReminderCommand` object.
6. In the `commandExecute` function of the `ViewReminderCommand` object, task data will be retrieved from the `TaskList` component.
7. Now that the `ViewReminderCommand` object has the data of the current tasks of the user, it is able to invoke the `getTaskReminders` method.
8. With the output returned from the `getTaskReminders`, the `CommandResult` object will be instantiated.
9. The `CommandResult` object would then be returned to the `LogicManager` which then returns the same `CommandResult` object back to the **`UI`** component.
10. Finally, the **`UI`** component would display the contents of the `CommandResult` object to the user.

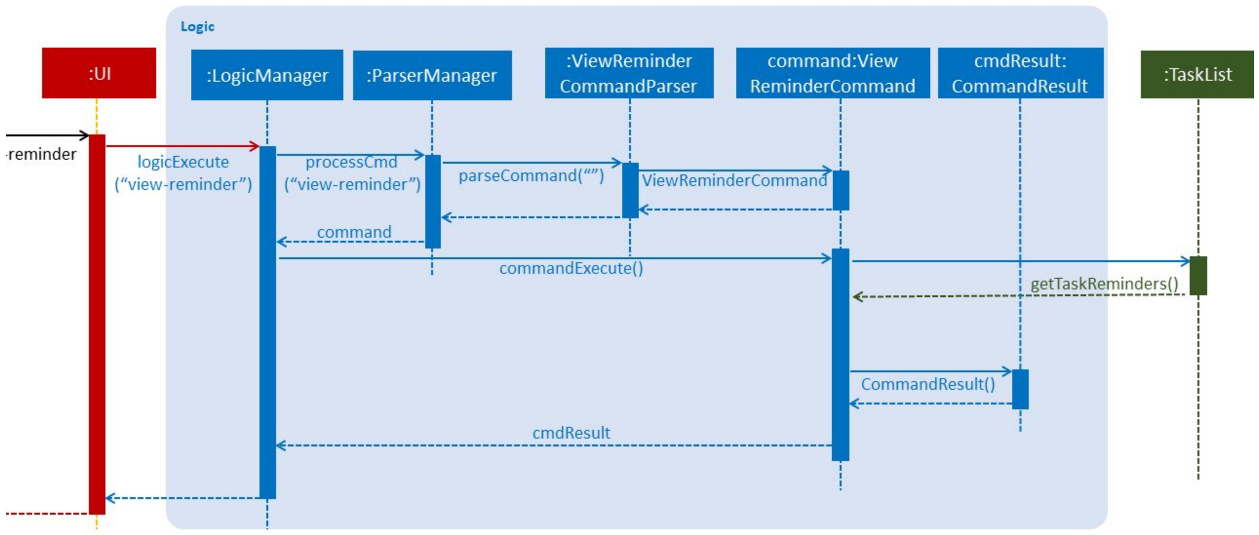

Figure 11. Sequence Diagram executing the **`view-reminder`** command.

5.4.2. Design Considerations

This section details our considerations for the implementation of the `set-reminder` and **`view-reminder`** feature.

Aspect: Functionality of `set-reminder` command

- **`set-reminder`** has a **`/status`** field so that the user can choose to turn on/off the reminder for the specified task.

Aspect: Functionality of `view-reminder` command

- Alternative 1 (current choice): Shows the user undone tasks that are either overdue, due within the week, or have reminder settings turned on.
    - Pros: Overdue tasks and tasks that are due within the week are automatically included in the **`view-reminder`** command. The user does not have to manually turn on the reminder settings for each upcoming task.
    - Cons: There are some restrictions on the user as the user cannot exclude overdue tasks or tasks due within a week from the **`view-reminder`** command.
- Alternative 2 : Shows the user only tasks that have reminder settings turned on.
    - Pros: The user can entirely dictate whether the tasks are shown when **`view-reminder`** command is executed by turning the reminder settings on/off.
    - Cons: It is troublesome for the user to change each task's reminder settings one by one every time the tasks are near their deadlines. The user might also forget to turn on reminders at times, missing important deadlines.

Alternative 1 was chosen as it is more user-friendly. By automatically including upcoming and overdue tasks, the user can easily keep track of the more urgent tasks that need to be completed. The user can also include important tasks in the reminders by turning the reminder settings on for the respective tasks.

5.4.3 Future Implementation

1. Allow the automatic addition of tasks due within a certain time period to reminders to be user-defined. Example: If the user inputs 14 days, tasks due within 14 days will be automatically included in the **`view-reminder`** command.

5.5. Find Feature

This feature allows the user to search for a keyword or phrase in the description field belonging to all of the tasks.

5.5.1. Current Implementation

The current implementation matches the keyword or phrase exactly to the description. As long as the keyword or phrase is a sub-string in the description field, the task is returned as a match. Likeness of the words are not considered at the moment e.g 'frst' will not match 'first'.

1. Upon the user entering the find command with a valid keyword, the `LogicManager` is called and sends the user input to `ParserManager`.

2. `LogicManager` then invokes the `parseCommand` function of `ParserManager.`

3. `ParserManager` in turn invokes the parse function of the appropriate parser for the find command which in this case, is `FindCommandParser.`
4. After parsing is done, `FindCommandParser` would instantiate the `FindCommand` object which would be returned to the `LogicManager`.
5. `LogicManager` is then able to call the execute function of the `FindCommand` object just returned to it.
6. In the execute function of the `ViewCommand` object, task data will be retrieved from the `TaskList` component.
7. Now that the `FindCommand` object has the data of the current task of the user, it is able to execute its logic.
8. `FindCommand` will loop through all tasks to find any description matching (non-case sensitive) the keyword or phrase.
9. The result of the command execution, a list of matches from the keyword/phrase passed in, is encapsulated as a `CommandResult` object which is passed back to Ui for displaying to the user.

Here is a sequence diagram portraying the above sequence of events:

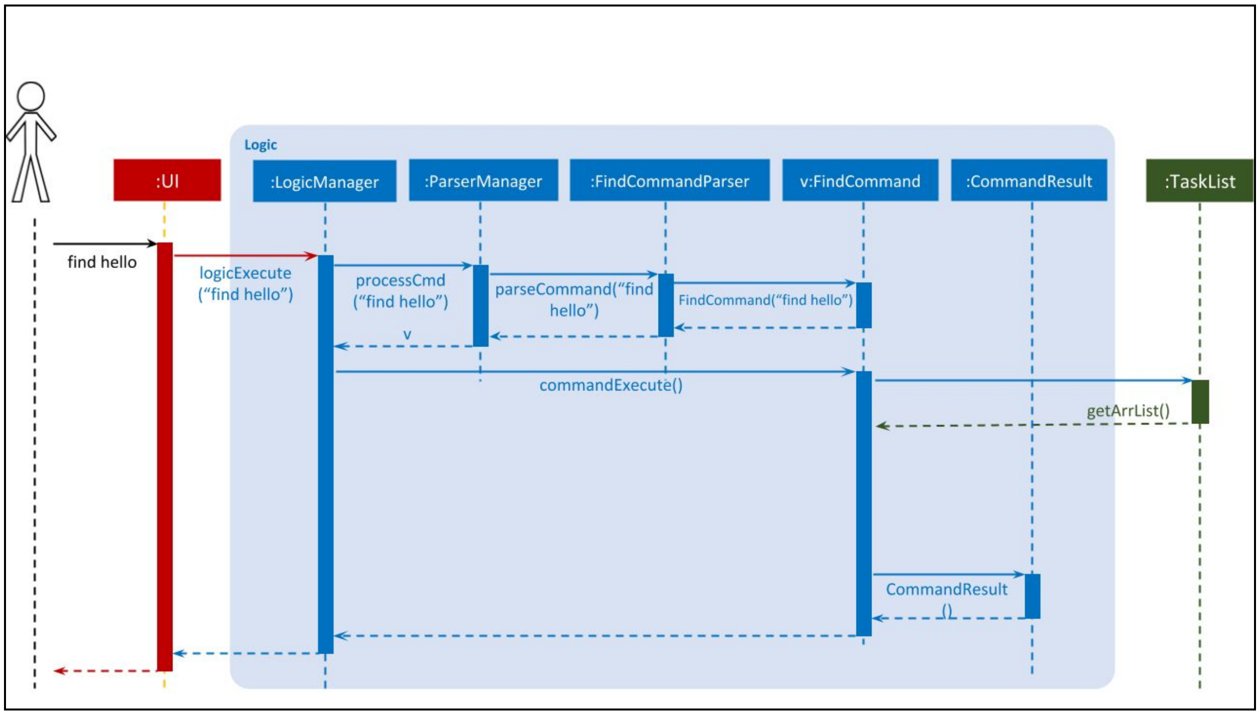

Figure 12: Sequence Diagram executing `find` command.

5.5.2. Design Considerations

The 'find' command should not be too complicated. It should be very intuitive to use and hence there was no need for any much parsing on the part of the FindCommandParser object. The user should be able to search for any task with ease and without referring to the user-guide as much as possible.

5.5.3 Future Implementation

Case-Sensitive Search

Involves just using a different match/regex API

Match Based On Likeness/Regular Expressions

Make use of regex to match words based on likeness/regex rules rather than an exact sub-string match. This will help users with typographical errors but is not considered a must to implement.

5.6. Help Feature

This feature allows the user to search for usage of a command. Whenever the user enters an invalid command, it will be regarded as a help command.

5.6.1. Current Implementation

The current implementation allows user to get the basic information about all commands with any invalid input or specific instructions on one command with **`help`**:

Command: Any possible invalid input or `help`

Upon invoking an invalid command (refer to UserGuide.adoc for `help` usage), a sequence of events is then executed.

A graphical representation is included in the Sequence Diagram below for your reference when following through the sequence of events. The sequence of events are as follows:

1. The `help` command is passed into the `logicExecute` function of `LogicManager` to be parsed.
2. `LogicManager` then invokes the `processCmd` function of `ParserManager`.
3. `ParserManager` in turn invokes the `parseCommand` function of the appropriate parser for the `help` command which in this case, is `HelpCommandParser`.
4. Once the parsing is done, `HelpCommandParser` would instantiate the `HelpCommand` object which would be returned to the `LogicManager`.
5. `LogicManager` is then able to invoke the `commandExecute` function of the returned `HelpCommand` object.
6. In the `commandExecute` function of the `HelpCommand` object, the `HelpCommand` object has the description of the command.
7. With the description of the command, `HelpCommand` will match it with existing commands and the `CommandResult` object will be instantiated with the matched command. If the description is empty `CommandResult` will be instantiated with the default message of basic information of all commands.
8. The `CommandResult` object would then be returned to the `LogicManager` which then returns the same `CommandResult` object back to the **`UI`** component.
9. Finally, the **`UI`** component would display the contents of the `CommandResult` object to the user.

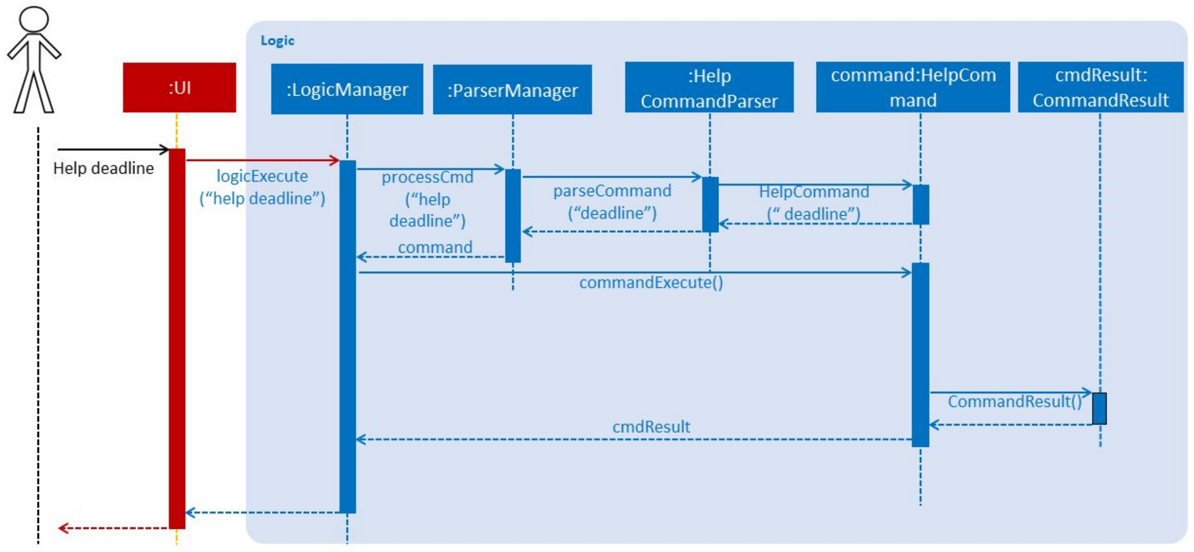

Figure 13. Sequence Diagram executing the `Help` command.

5.6.2. Design Considerations

Aspect: Functionality of `help` command

- Alternative 1 (current choice): Show a brief help for all commands when invalid command is entered and detailed instructions for a specific command when `help <command name>` is used.
    - Pros: The user can get a brief idea of how to use all commands and guide them to use help to do further search.
    - Cons: The user cannot check the specific instructions for multiple commands at the same time. The user must search for several times to understand how to use all commands.

- Alternative 2 : Shows the user all commands format when `help` is called.
    - Pros: The user can learn how to use all commands with entering `help` command once.
    - Cons: Showing all commands at the same time will print huge amount of texts on the main window tab. It may be troublesome for the user to search for specific command needed.

Alternative 1 was chosen as it is more user-friendly. The user can get an overview of all commands first and then search for specific commands. If more commands are added, alternative 2 requires the user to look through a huge page of texts to find the format of one instruction.

5.6.3 Future Implementation

1. More details and examples of each command. With more examples, the user could have a better idea of the full functions of each command.
2. Another method to show the list of full instructions regarding all commands. It is more intuitive for users who use COMPal for the first time to know how to use it in a shorter time.

6. Testing

6.1. Running Tests

To run all tests, open a console inside the project directory and run the command `gradlew clean test` (Mac/Linux: `./gradlew clean test`)

6.2. Types of Tests

We have one types of tests:

1. Integration tests that are checking the integration of multiple code units (those code units are assumed to be working).
   e.g. DeleteCommandTest

7. Dev Ops

7.1. Build Automation

See UsingGradle.adoc to learn how to use Gradle for build automation

7.2. Continuous Integration

We use Travis CI and Codacy to perform *Continuous Integration* on our projects. See UsingTravis.adoc and UsingAppVeyor.adoc for more details.

7.3. Making a Release

Here are the steps to create a new release.

1. Update the version number in gradle build file.
2. Generate a JAR file using Gradle.
3. Tag the repo with the version number. e.g. v0.1
4. Create a new release using GitHub and upload the JAR file you created.

Appendix A: User Profile

System: **COMPal**

Target User Profile: Students who

- want to better organize their time not just according to deadlines but by perceived priorities

- prefer interacting with a CLI
- prefers typing over mouse input

Persons that can play this role : Undergraduate student, graduate student, a staff member doing a part-time course, exchange student

Value Proposition: Students wanting to be more organized without going through too much of a hassle can now better manage their schedules and tasks with Compal's clean and intuitive user-interface and user-defined priority-based organization.

Appendix B: User Stories

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

As a ...	I want to...	So that I can ...	Priority
Student	Add the due dates of tasks that I have	Neatly organize my schedule	***
Student	Add my academic timetable	Store my academic schedule	***
Student	Add meeting schedules	Easily remember about scheduled meetings	***
Student	Add examination dates and times	View and track upcoming assessments	***
Student	Add a description to a task that I have	Record necessary information about the task	***
Student	Edit due dates of tasks that I have	Update the description and deadlines of the tasks	***
Student	Edit my academic timetable	Update my academic schedule	***
Student	Edit meeting schedules	Update my appointment timings	***
Student	Edit examination dates and times	Update assessment dates	***
Student	View the application in a graphical user interface	View things in an organised and quick manner	***
Student	View the tasks that are soon to be overdue	Keep track of the things to do	***
Student	View the timetable in a daily view	See the overview of the whole day	***
Student	View my ongoing school-related task	Keep track of my progress	***
Student	Be notified of my classes to attend	Be reminded of my schedule	***
Student	Be notified of the tasks due	Be reminded of my schedule	***
Student	Be notified of upcoming examinations	Be reminded of my schedule	***

Student	Be notified of upcoming meetings	Be reminded of my schedule	***
Student	Sort my tasks according to the deadlines and importance	Know which task needs to be focused on	***
Student	Find specific things in the application using a keyword	Find related things	***
Student	Remove a scheduled slot	Delete cancelled meetings/classes	***
Student	Remove tasks	Delete tasks	***
Student	Priortise more important timetable slots based on personal ranking	rearrange my schedule in the event that there is a timetable clash	***
Student	View the timetable in a monthly view	See the overview of the whole month	**
Student	View the timetable in a weekly view	See the overview of the whole week	**
Student	Mark my ongoing school-related task as completed by task and subtask	Keep track of the progress of individual task and subtasks	**
Student	Track my assignment progress	Know what needs to be done	**
Student	Add the result/grade of module assignment, attendance, midterm results	Store module's component grades	*
Student	Add my received module grades for each semester	Store the semester's grades	*
Student	Edit the result/grade of module assignment, attendance, midterm results	Estimate the grade that I will receive	*
Student	Track my cumulative GPA	Work towards the GPA I aim for	*

Appendix C: Use Cases

Use Case 1: Store task or academic schedule

Main Success Scenario (MSS)

1. . User inputs event command followed by all the mandatory parameters.

2. . System reflects the additions to the planner.

   Use case ends.

Extensions

- 1a. System detects an error in the entered data.

  - 1a1. System outputs error message.

    Use case ends.

- 1b. System detects insufficient parameters in the entered data.

  - 1b1. System outputs error message.

    Use case ends.

Use Case 2: Edit Task

Prerequisite: User is aware of the TaskID

MSS

1. User inputs command to edit a task along with the TaskID, followed by the parameters which is needed to be changed.

2. System changes the specified parameters for the slot.

3. System then reflects the task parameters as well as the parameters changed.

   Use case ends.

Extensions

- 1a. TaskID does not exist in COMPal.

  - 1a1. System outputs error message.

    Use case ends.

- 1b. System detects an error in the entered data.

  - 1b1. System outputs error message.

    Use case ends.

Use Case 3: Mark Task as Done

Prerequisite: User is aware of the TaskID.

MSS

1. User enters command to mark task as done

2. COMPal reflects task status changes

   Use case ends.

Extensions

- 1a. TaskID does not exist in COMPal.

  - 1a1. System outputs error message.

    Use case ends.

- 1b. System detects an error in the entered data.
    - 1b1. System outputs error message.

      Use case ends.

Use Case 4: Change the daily view date

MSS

1. User enters command to change the date of daily calendar view.

2. COMPal displays the selected view date on GUI.

   Use case ends.

Extensions

- 1a. System detects an error in the entered data.

    - 1a1. System outputs error message.

      Use case ends.
- 1b. System detects no task on selected view date.
    - 1b1. System outputs message indicating no task on chosen date.

      Use case ends.

Use Case 5: Search for Tasks

MSS

1. User enter find command along with the parameter to search for.

2. COMPal reflects search results

Extensions

- 1a. System does not find matching keyword

    - 1a1. System indicates that there are no matching keyword.
- Use case ends.

Appendix D: Non-Functional Requirements

1. COMPal can store up to 1,000,000 tasks in a clear text file.

2. COMPal must respond fast, within 2 seconds so that the user does not have to wait too long.

3. COMPal system application should take up relatively little space on the local machine.

4. COMPal's GUI must be intuitive and pleasant to the eyes.

5. COMPal consistently performs specified function without failure.

6. The user's OS date and time must be correctly synchronizes to local date and time.

Appendix E: Glossary

Task: A generic term used to refer to any instance of an object in the user's schedule.

View: The layout in which the schedule is displayed to the user.

GUI : The graphical user interface of the application.

Appendix F: Instruction for Manual Testing

Given below are instructions to test the app manually.

ℹ	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
    i. Download the jar file and copy into an empty folder
    ii. Double-click the jar file
    Expected: Shows the GUI with a set welcome prompt or weekly view of task of the user.

F.2. Adding a task

F.3. Editing a task

F.4. Searching for a task

F.5. Viewing the schedule

1. View all *tasks* for the current day.
   Prerequisites: Add *tasks* or *deadline* using the `deadline` `or` `event` command for any day of the current day.
    i. Test case: `view day`
    Expected: A daily view with all added *tasks* and *deadlines* for the current day in text format and GUI daily schedule output.
    ii. Test case: `view day /date 29/10/2019`
    Expected: A daily view with all added *tasks* and *deadlines* for 29/10/2019 in text and GUI output.
    iii. Other incorrect view commands to try: `view day /date 29/02/2021`
    Expected: Error message returned which shows that date input is not valid as the date does not exist in the calendar.

2. View all tasks for the current week.
   Prerequisites: Add *tasks* or *deadline* using the `deadline` `or` `event` command for any day of the current day.

i. Test case: `view week`
Expected: A weekly view with all added *tasks* and *deadlines* for the current week in text format.
ii. Test case: `view week /date 29/10/2019 /type deadline`
Expected: A weekly view showing only *deadlines* starting from 29/10/2019 - 04/11/2019 in text output.
iii. Other incorrect view commands to try: `view week /type assignment`
Expected: Error message returned which shows that the type does not exist.

F.7. Listing all tasks

1. List all tasks stored in COMPal.
   Prerequisites: Add *tasks* or *deadline* using the `deadline` `or` `event` command for any day.
   I. Test case: `list`
   Expected: List output of all tasks that are stored in COMPal.

   II. Test case: `list /type deadline`
   Expected: List of all deadlines that are stored in COMPal.

   III. Test case: `list /type event`
   Expected: List of all events that are stored in COMPal.

F.8. Changing tasks status

1. Marking a tasks as complete.
   Prerequisites: Add *tasks* or *deadline* using the `deadline` `or` `event` command for any day or using `list` to get `taskID` from list command.
   i. Test case: `done /id 0 /status Y`
   Expected: A confirmation message that states that COMPal have mark the tasks as complete
   ii. Other incorrect view commands to try: `done /id <out-of-range> /status Y`
   Expected: Error message returned which shows that the `taskID` does not exist.

2. Marks a tasks as incomplete
   Prerequisites: Add *tasks* or *deadline* using the `deadline or event` command for any day or using `list` to get `taskID` from list command.
   I. Test case: `done /id 0 /status N`
      Expected: A confirmation message that states that COMPal have mark the tasks as incomplete.
   II. Other incorrect view commands to try: `done /id <out-of-range> /status N`
      Expected: Error message returned which shows that the `taskID` does not exist.

F.9. Setting a reminder

F.10. Viewing reminders

F.11. Deleting a task