COMPal - Developer Guide

By: `AY1920S1-CS2113T-W17-1` Last Updated: `10/11/2019` 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***! **COMPal** also allows ***easy planning*** of future **tasks**, with features such as reminders for **tasks** and finding ***free time slots***.

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 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	This general term 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

This section contains the architectural design of the different components in **COMPal**.

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

    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. Run the `Main` class (right-click the `Main` class and click `Run Main.main()`), and try executing a few commands.

9. Run all the tests (right-click the `test` folder, and click `Run ‘All Tests’`) and ensure that they pass.

10. Open the `tasks.txt` file and check for any code errors.

11. Open a console and run the command `gradlew processResources` (Mac/Linux: `./gradlew processResources`). It should finish with `BUILD SUCCESSFUL` message. This will generate all the resources required by the application and tests.

12. Open `Main.java` and check for any code errors.

    a. Due to an ongoing issue with some newer versions of **IntelliJ**, code errors may be detected even if the project can be built and run successfully.

b. To resolve this, place your cursor over any of the code section highlighted in red. Press `ALT` and `ENTER` keys, and select `Add ‘--add-modules=...’` to `module compile options` for each error.

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.

4. Design

This section contains the architectural design of the different components in **COMPal**.

4.1. Architecture

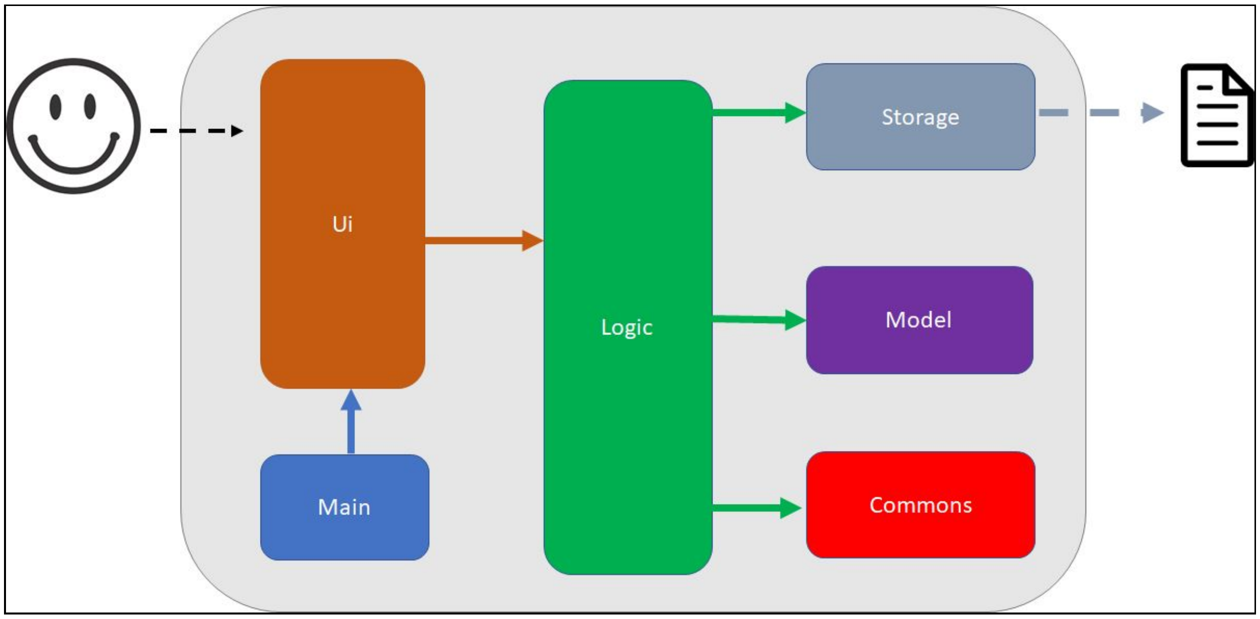

Figure 1. Architecture Diagram

The **Architecture Diagram** given above explains the high-level design of **COMPal**. The six components shown are:

1. **Main**: Our main classes comprises of 2 classes, `MainLauncher.java` and `Main.java`. At app launch: `MainLauncher.java` will instantiate `Main.java`. `Main.java` will initialize the UI to start listening to user input.

2. **Commons**: A collection of miscellaneous classes used by other components in **COMPal**.

3. **UI**: The User Interface of the **COMPal**.

4. **Logic**: Parses user input and executes valid commands.

5. **Model**: Holds the data of the **COMPal** in-memory.

6. **Storage**: Handles reading and writing of **COMPal** data to and from the hard disk.

Please read the following sections for more information on each component.

4.2. Commons Component

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

- **CompalUtils**: contains miscellaneous methods used to abstract out similar simple operations that can be carried out by multiple unrelated classes in **COMPal**, e.g. the `stringToDate()` method used to convert a date in the form of a String object into a Date object.

- **LogUtils**: contains methods that perform logging for various classes in **COMPal**.

4.3. UI component

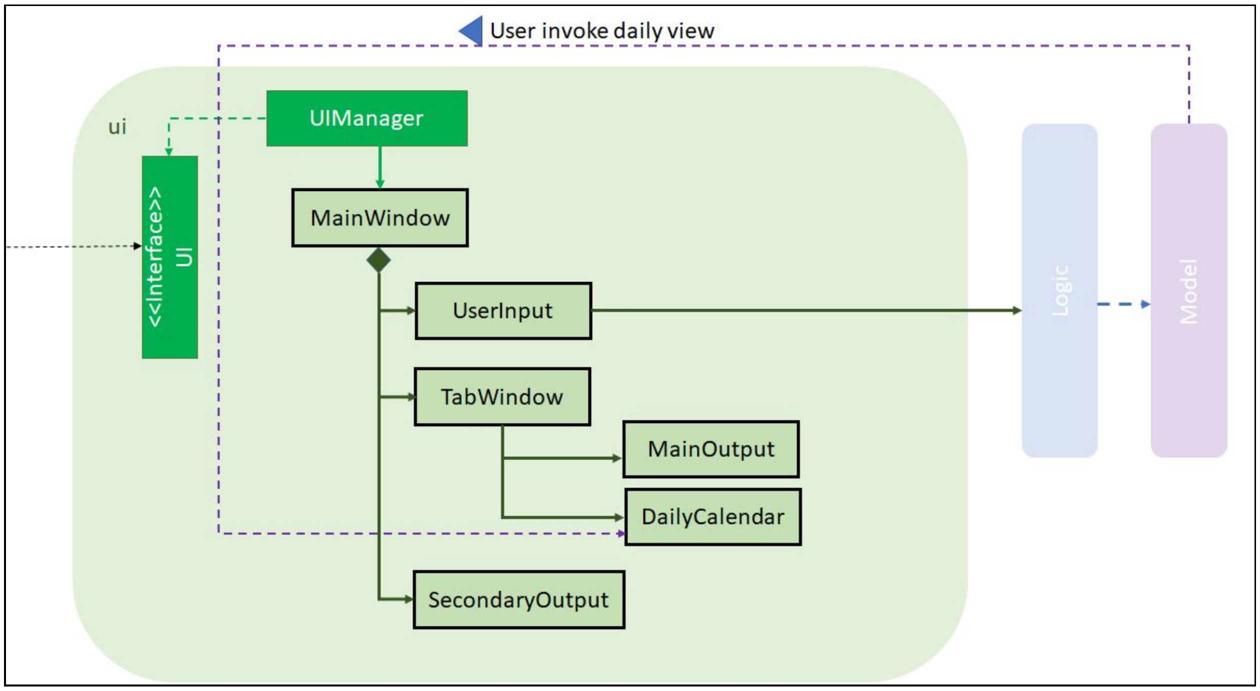

Figure 4. Structure of the **UI** Component

API: `Ui.java`

The **UI** consists of a `MainWindow` that is made up of parts e.g. `UserInput`, `SecondaryOutput`, `tabWindow` whose tabs consist of `MainOutput`, `DailyCalendar`. Although the application relies on text-based input, 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 `DailyCalendar` uses information from the **Model** 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 to the user via `MainOutput` or `SecondaryOutput`.
- Display daily calendar of the user via `DailyCalendar`.

4.4. Logic component

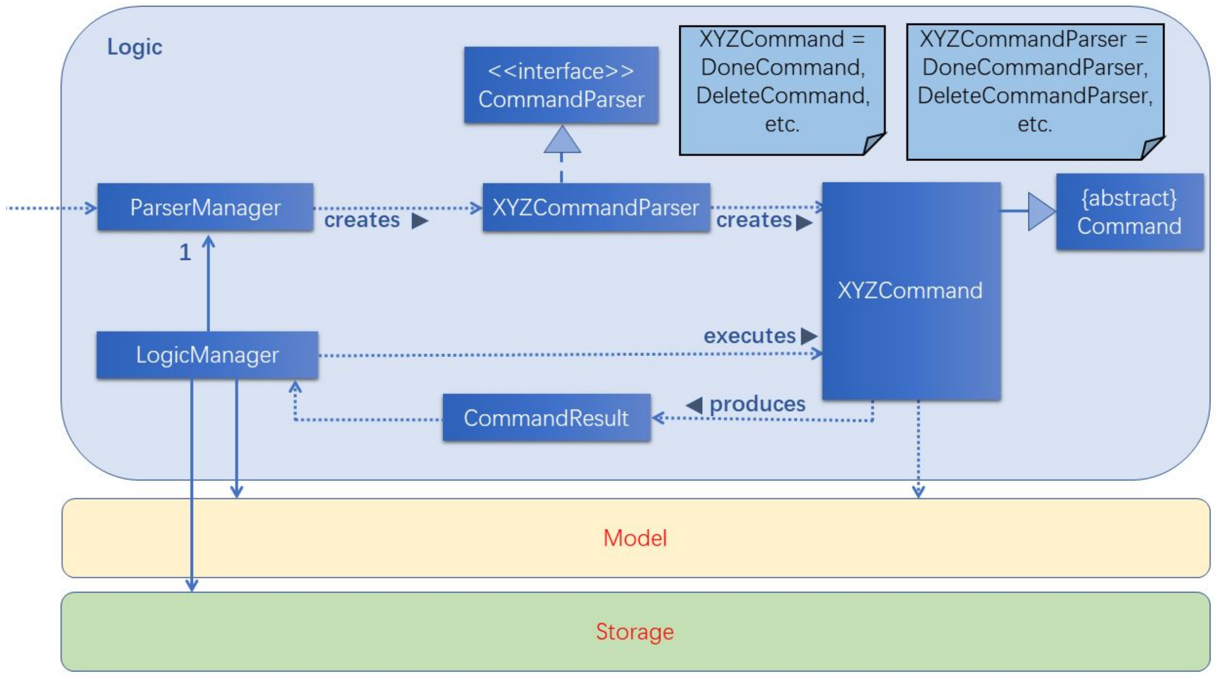
Logic
<<interface>> CommandParser
XYZCommand = DoneCommand, DeleteCommand, etc.
XYZCommandParser = DoneCommandParser, DeleteCommandParser, etc.
ParserManager
creates
XYZCommandParser
creates
XYZCommand
{abstract} Command
1
LogicManager
executes
CommandResult
produces
Model
Storage

Figure 5. Structure of the **Logic** component

API for `LogicManager`: [`LogicManager.java`]

API for `ParserManager`: [`ParserManager.java`]

API for `CommandParser`: [`CommandParser.java`]

API for `Command`: [`Command.java`]

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.

Given below is the Sequence Diagram for interactions within the **Logic** Component for the `logicExecute` `(delete /id 1)` API call.

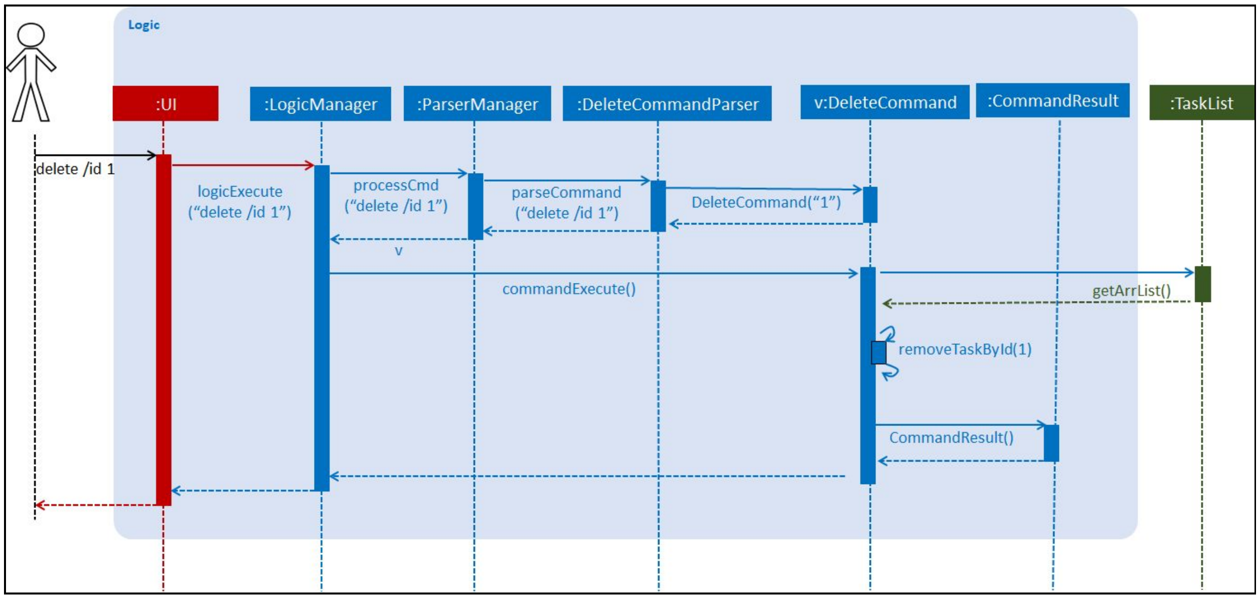

Figure 6: Interactions inside **Logic** Component for the `delete /id 1` command

4.5. Model Component

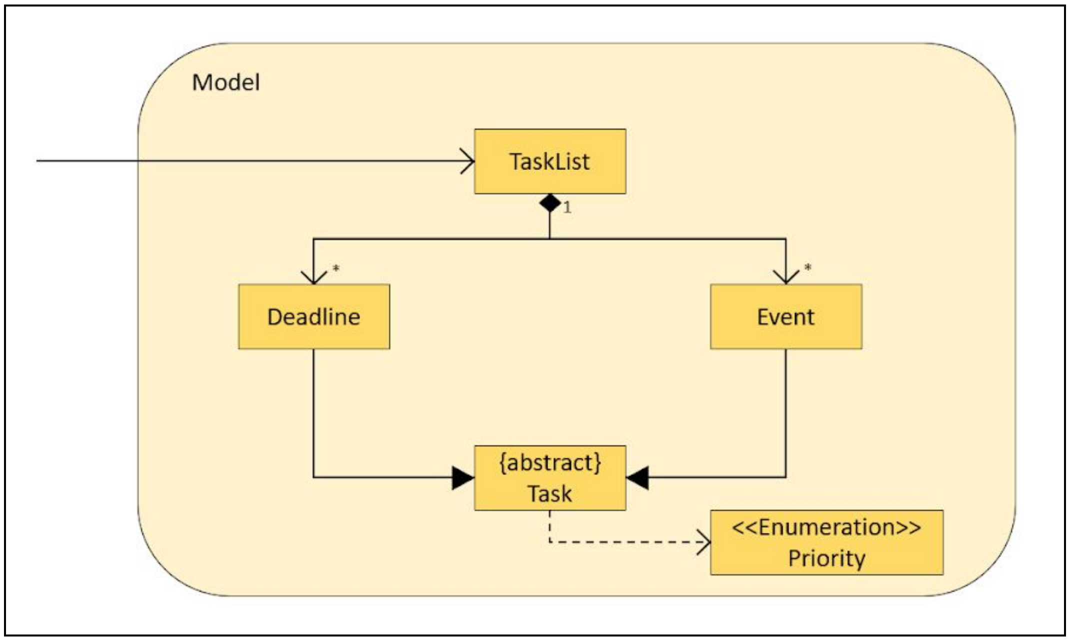

Figure 6. Overall structure of the **Model** Component

API: `Model`

The **Model** component is an abstract representation of tasks which

- 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.

4.6. Storage Component

API: `TaskStorageManager.java`

We use very ***simple and user-editable text files*** to store user data. Data is stored as data strings separated by **underscores**. This separation token, however, can be easily changed if desired. Data is then parsed as a string and then processed by our **Storage API** into application-useful data types such as **task** objects.

While it might be viewed as primitive, the advantage of this approach is that it is a 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.

5. Implementation

This section contains the implementation details of the different features in **COMPal**.

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 User Guide 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 is 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. The `CalendarUtil` object will be constructed only if the user enters `view day`. It will invoke `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**.

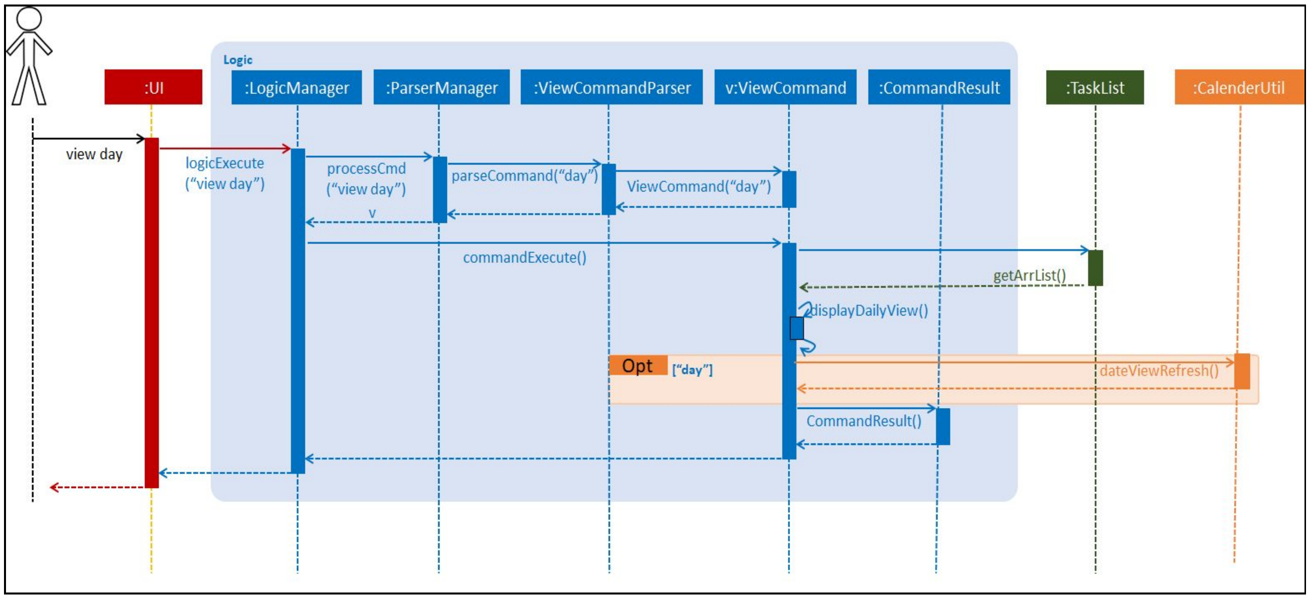

Figure 7. Sequence Diagram executing the `view day` command.

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 date is not inputted by the user, COMPal will deem that the 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 calendar 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 is as follows:

1. Using the `TaskStorageManager` class, load 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 start 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 start 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.

3.3. Create and print **deadlines** of the user 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 deadlines or task per time slot can be printed out to ensure that the user focuses 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 the existence of **task** and mark that a slot has been added for that hour until the end of the **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.

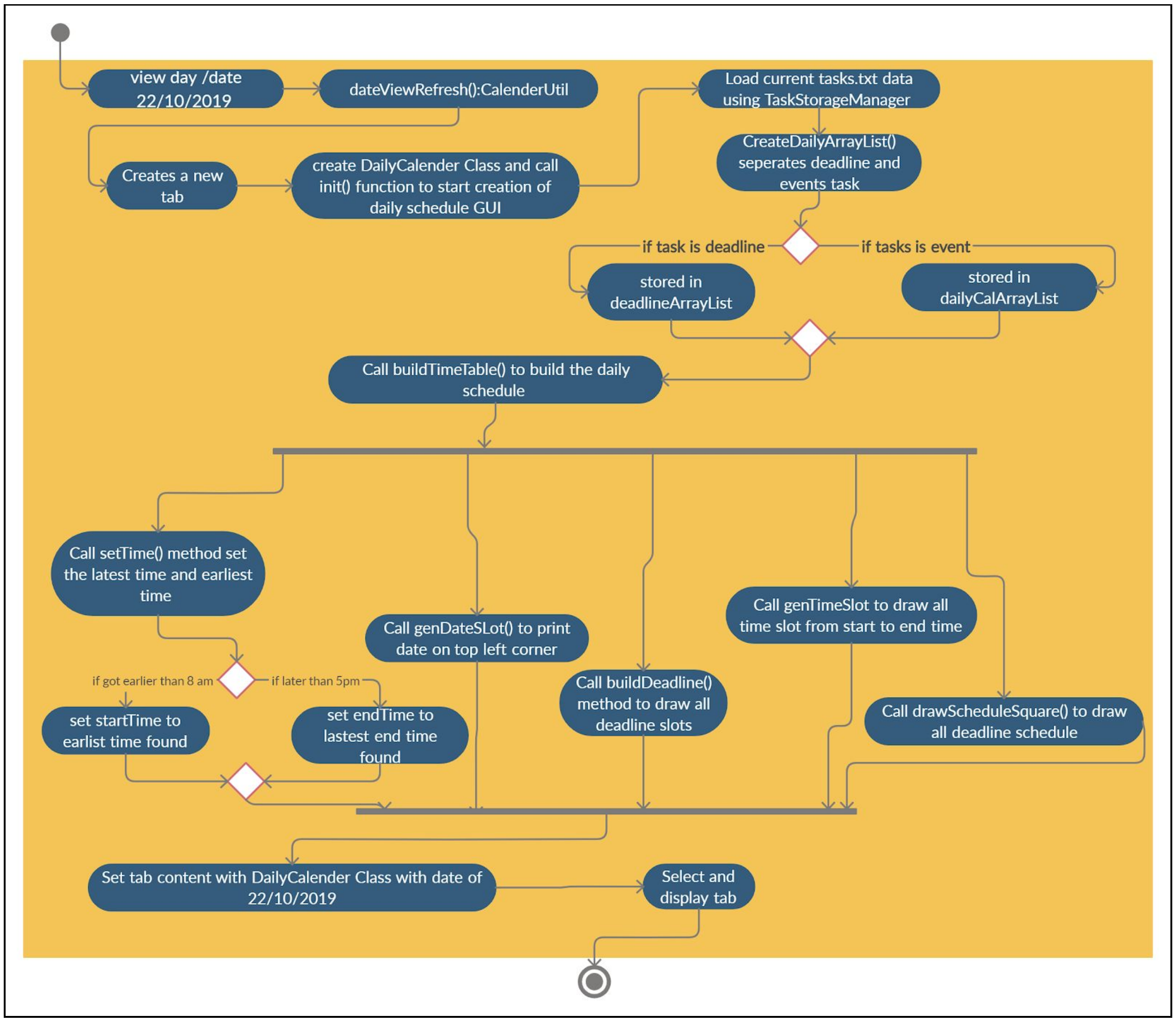
view day /date 22/10/2019
dateViewRefresh():CalenderUtil
Load current tasks.txt data using TaskStorageManager
Creates a new tab
create DailyCalender Class and call init() function to start creation of daily schedule GUI
CreateDailyArrayList() seperates deadline and events task
if task is deadline
if 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 instantiated 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 usage of resources. This 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 is 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

The current implementation prevents cluttering the application which allows 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 1: 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 User Guide 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 is 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.

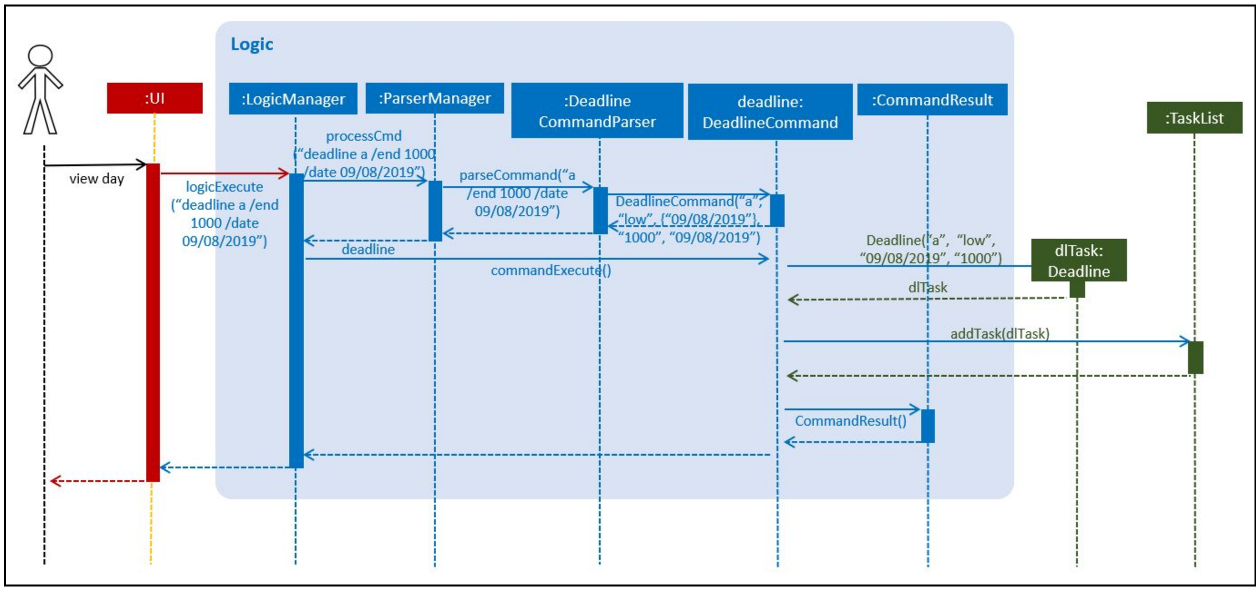
Logic
:UI
:LogicManager
:ParserManager
:Deadline CommandParser
deadline: DeadlineCommand
:CommandResult
:TaskList
view day
logicExecute ("deadline a /end 1000 /date 09/08/2019")
processCmd ("deadline a /end 1000 /date 09/08/2019")
parseCommand("a /end 1000 /date 09/08/2019")
DeadlineCommand("a", "low",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

Aspect: Implementing the Task object

- **Alternative 1 (current choice):** A **Task** object is instantiated for every `deadline` and `event` command, and added to an `ArrayList` of **tasks** contained in the `TaskList` component.
    - **Pros:** Simple and easy to understand for new developers
    - **Cons:** Can be unnecessarily memory-intensive if user uses the recurring functions of `deadline` and `event` commands to create multiple **tasks** with the same attribute. A possible consistency problem if the user has made a mistake in the input of the recurring **task - COMPal** will have to go through every copy of the **task** to make any correction/deletion.

- **Alternative 2:** Use the **abstraction occurrence pattern** in the creation of recurring **tasks**, where a **task** is represented by two objects: common information into one class and the number of occurrences into another class.
    - **Pros:** Will not waste as much storage space if the user uses the recurring function often.
    - **Cons:**
        - Can still be memory-intensive, if the user does not use the recurring feature often, but simply adds many **tasks**.
        - Can be unnecessarily complicated in performing certain operations on **tasks**, such as sorting according to date, time and priority. May not fulfil

the **open-closed principle**, as such an implementation is not easy to add new operations (not open to extension)

Alternative 1 was chosen because it is simpler and easier to understand for new developers. It allows for easier extension of current features and adding of newer potential features. In addition, the recurring feature is not expected to be used excessively.

Aspect: Implementing the sorting and indexing of Tasks in TaskList

- **Alternative 1 (current choice):** The `ArrayList` in `TaskList` component is stable sorted according to the attributes of each **task** (the dates and times that each **task** is starting and ending at, and their priorities) on two occasions: when the data is to be written to disk (`tasks.txt`), and when `findfreeslot` command is used to find a duration of time that contains no **tasks**.
    - **Pros:** An `ArrayList` retains its order once it is sorted. Therefore it is easier to access **tasks** by attributes like date, time and priority for most operations, such as the `findfreeslot` and `view` features (for finding slots in a specific time and date range).
    - **Cons:** Can be computationally expensive to look through the entire `ArrayList` for the **task** that matches the Task ID entered by the user.

- **Alternative 2: Tasks** are stored in a `HashMap` object contained in the `TaskList` component.
    - **Pros:** O(1) access time for less computationally expensive access, delete and add, using Task ID attribute of each **task**.
    - **Cons:** Task ID is the only attribute that is easy to access.

Alternative 1 was chosen because most of the operations in **COMPal** require accessing the other attributes of each **task** object (such as stable sorting the `ArrayList` of **tasks**, and finding a free slot for a specific range of dates / times.

5.2.3. Future Implementation

1. Implementing a **Task** that can extend over multiple days (more than one day).

5.3. Priority Feature

This feature allows the 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 priority level from the enumeration (**low, medium, high).**

The priority of a **task** can be set in 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:** Use integer to store the three different levels of priority(1, 2, 3).
    - **Pros:** It is easy to store and update priority.
    - **Cons:** The user could purposely type invalid priority such as 6, which require extra detecting algorithm to check and prevent.

- **Alternative 2 (current choice):** Use enumeration to store three different levels of priority(high, medium, low).
    - **Pros:** It is guaranteed that COMPal can only accept high, medium and low levels of priority and there is no need for any extra check to detect invalid user input.
    - **Cons:** Priority levels lost the flexibility of being integers. No more integer performance can apply to priority levels. e.g. priority ++. Need to import the enumeration for all classes.

Alternative 2 was chosen as it restricts the user input and does not require extra checking methods. Alternative 2 also produce more intuitive format(high, medium and low) compared with abstract priority level using alternative 1(1, 2, 3)

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 User Guide 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 is 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.

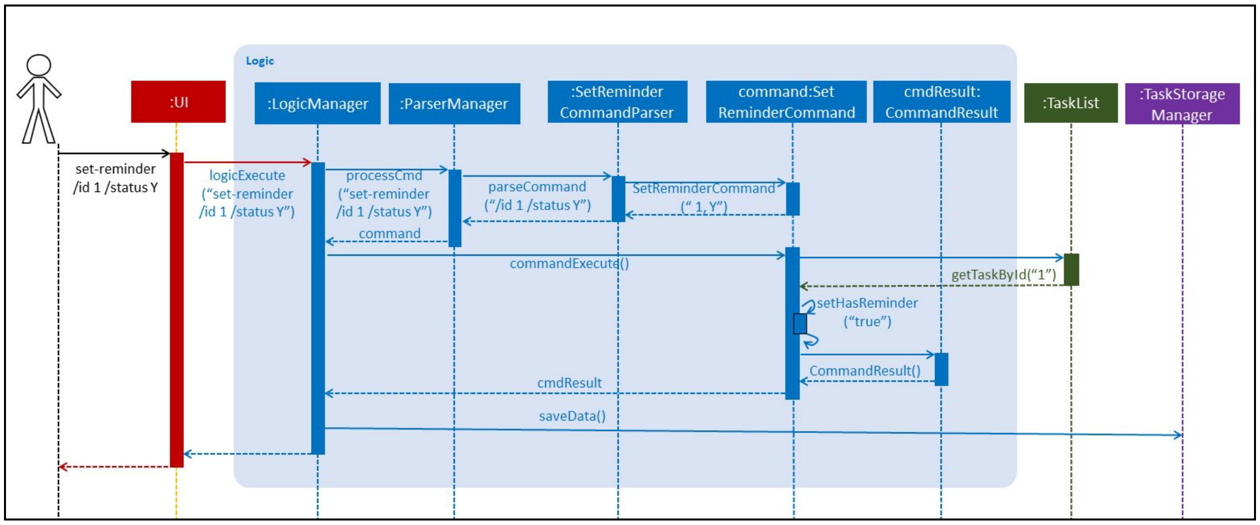
Logic
:UI
:LogicManager
:ParserManager
:SetReminder CommandParser
command:Set ReminderCommand
cmdResult: CommandResult
:TaskList
:TaskStorage Manager
set-reminder /id 1 /status Y
logicExecute ("set-reminder /id 1 /status Y")
processCmd ("set-reminder /id 1 /status Y")
parseCommand ("/id 1 /status Y")
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 User Guide 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 is 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.

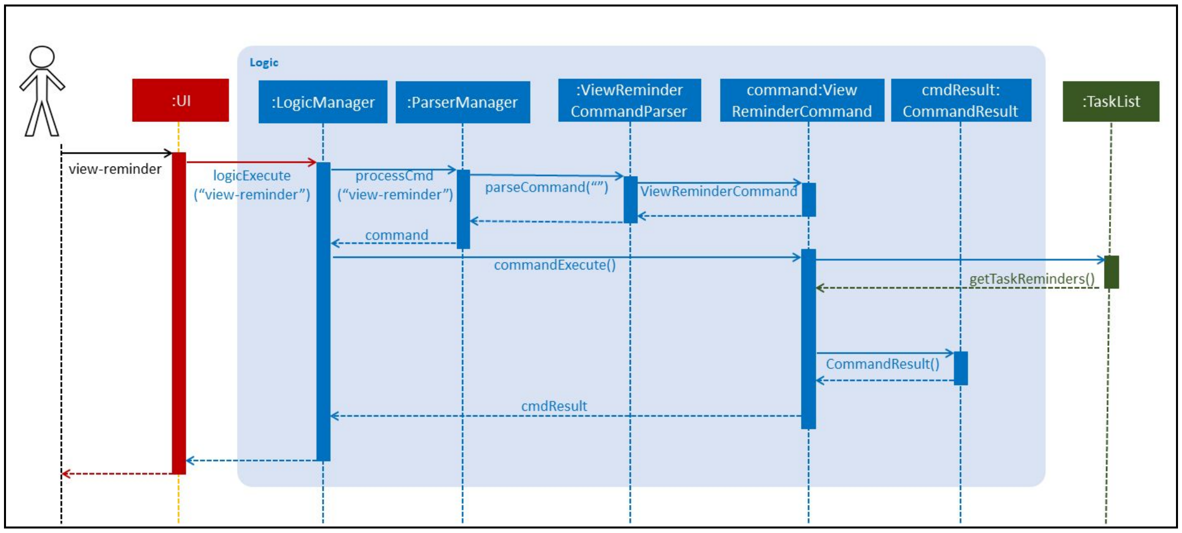

Figure 11. Sequence Diagram executing the **`view-reminder`** command.

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):** Implements Command Pattern under the Software Design Pattern.
  - **Pros:** More efficient as reminders are only shown when the user invokes the function.
  - **Cons:** The user is not notified when a change is made to the reminder list. The user is only aware if a change has been made when he calls on the function
- **Alternative 2:** Implements Observer Pattern under the Software Design Pattern.

- **Pros:** The user is notified that every time the reminder list changes. Hence, the user is constantly updated on the reminder list.
- **Cons:** It is inefficient to invoke this function every time a change is made.

Alternative 1 was chosen as it is more efficient. Since most commands can potentially change the reminder list contents, by showing reminders only when the user calls on the function, it increases the efficiency of the application as the function is only executed whenever necessary.

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:

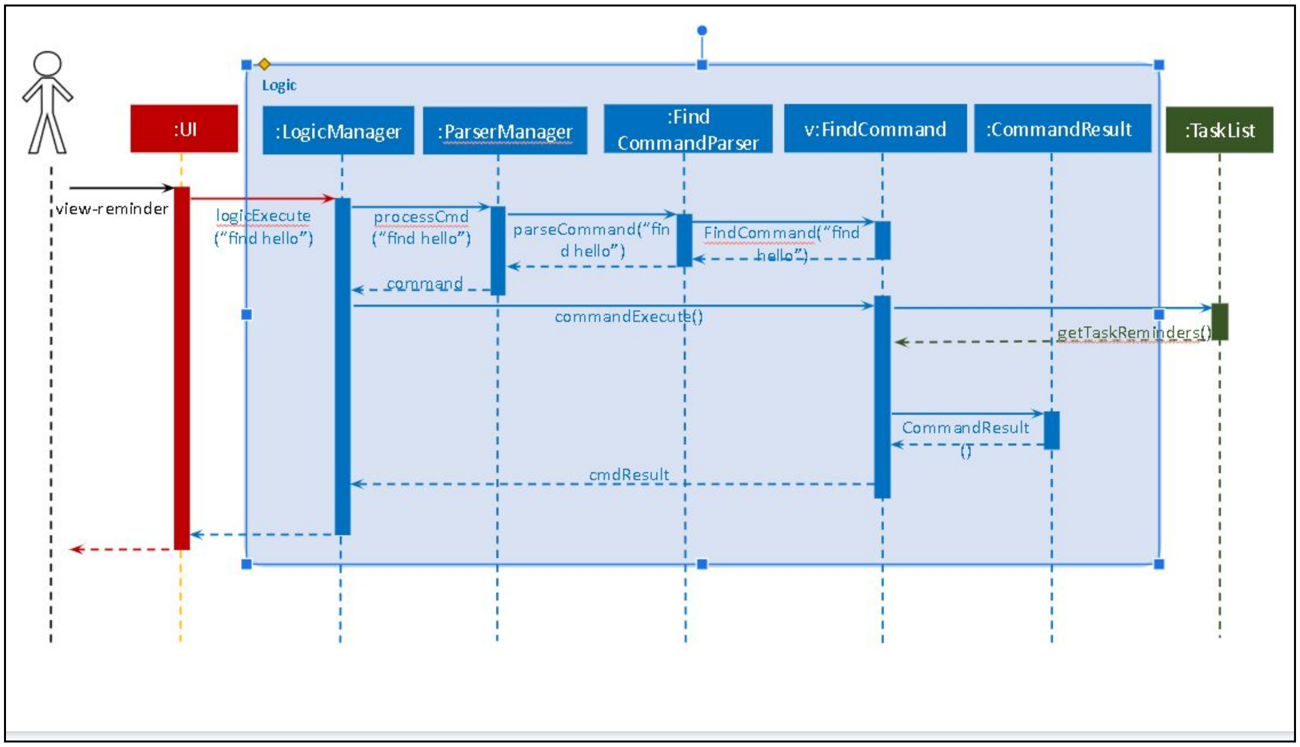

Figure 12: Sequence Diagram executing `find` command.

5.5.2. Design Considerations

This portion explains alternative implementations as well as the rationale behind my chosen method.

Aspect: Usage of Java API.

- **Alternative 1:** Using `Matcher` and `Pattern` **API** from the `util.regex` **Java library** to do regular expression matching.
    - **Pros:** Can search for any `regex` pattern
    - **Cons:** User has to understand/know `regex`, or we will need to convert their specified search criteria into a regular expression

- **Alternative 2 (current choice):** Use the simple `contains()` function from `String`
    - **Pros:** Simple to use and sufficient for the purposes of `find`
    - **Cons:** Cannot expand further if we desire to implement more advanced searches

Alternative 2 was chosen because of its simplicity and because we felt that the find feature need not be too overly complicated since our application was designed to be very simple to operate.

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 substring 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 the user to get the basic information about all commands with any invalid input or specific instructions of one command with **`help`**:

Command: Any possible invalid input or `help`

Upon invoking an invalid command (refer to User Guide 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 is 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.

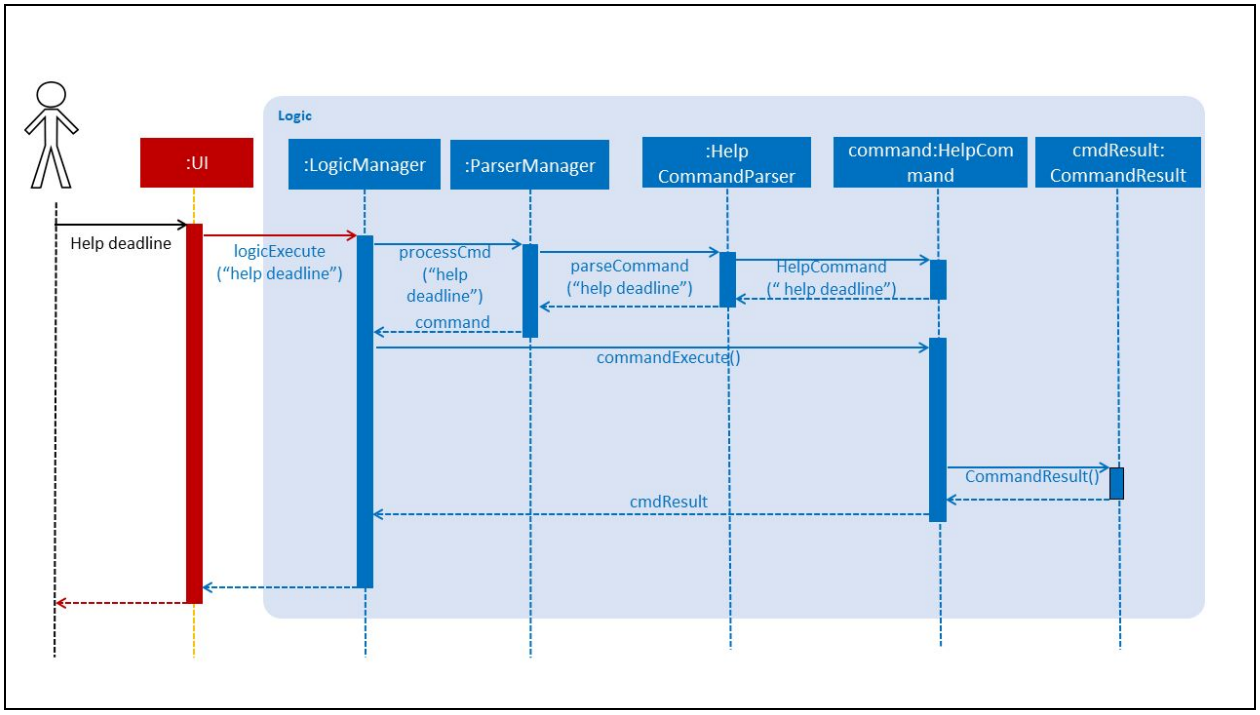

Figure 13. Sequence Diagram executing the `Help` command.

5.6.2. Design Considerations

Aspect: Functionality of `help` command

- **Alternative 1 (current choice):** Store a brief help in a string for all commands for invalid command and store detailed instructions for a specific command in strings in respective command classes for `help <command name>`.
    - **Pros:** It is easier to add help for new commands and can modify the help for specific command easily.
    - **Cons:** Need to store things in multiple classes. When adding a new command, need to change strings in both help and the new command class.

- **Alternative 2:** Store the help for all commands in a string in help class when `help` is called.
    - **Pros:** Only need to store everything in help class. When any command changes, only need to modify help command string.
    - **Cons:** The string in help command could be very long and need to be careful to deal with updating.

Alternative 1 was chosen as it is more user-friendly and easier to update for developers. The user can get an overview of all commands first and then search for usage of specific commands. The developer can add new command's help easily. If more commands are added, alternative 2 requires the developer 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.

5.7. Export Feature

This feature allows COMPal to exports its stored **tasks** into an iCalendar file. This section will detail how this feature is implemented.

5.7.1. Current Implementation

Upon invoking the export command with valid parameters (refer to User Guide for `export` usage), a sequence of events is then executed.

For clarity, the sequence of events will be in reference to the execution of an `export /file-name myCal` command. A graphical representation is also 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 `export /file-name myCal` 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 `ExportCommandParser`.

4. Once the parsing is done,`ExportCommandParser` would instantiate the `ExportCommand` object which would be returned to the `LogicManager`.

5. `LogicManager` is then able to invoke the `commandExecute` function of the returned `ExportCommand` object.

6. In the `execute` function of the `ExportCommand` object, **task** data will be retrieved from the `TaskList` component.

7. Now that the `ExportCommand` object has the data of the current **task** of the user, it is able to invoke the `creatIcsCal` function which converts all stored **tasks** of user into `ical4j Calendar model` .

8. Once all **tasks** are converted, using `ical4j calenderOutputer` api, which writes an iCalendar model to an output stream. The **task** converted will be saved into `myCal.ics`.

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. For this `export /file-name myCal` command, the displayed result would be that the program has successfully exported to mycal.ics file.

Given below is the Sequence Diagram upon executing the `export` command:

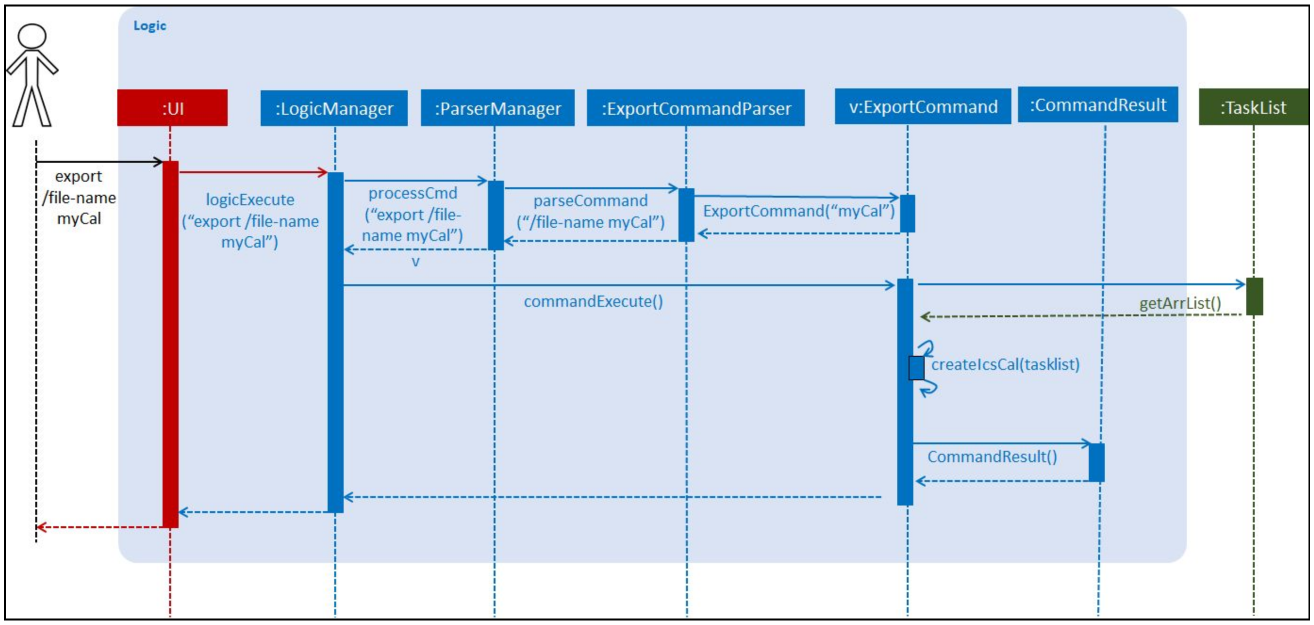

Figure x. Sequence Diagram executing the `export /file-name myCal` command.

5.7.2. Design Considerations

This portion explains alternative implementations as well as the rationale behind my chosen method.

Aspect: Export type

- **Alternative 1 (current choice):** Using `iCal4j` library to convert **tasks** and write iCalender file.
  - **Pros:** Able to use `iCal4j` library to write `iCalendar` data streams. This **API** provides a quick and easy method to convert a current stored **task** to `iCalendar` data stream such as events.
  - **Cons:** Does not have a special field to store the **priority** object of each **task**.
- **Alternative 2:** Output all stored data into a text file or `.csv` file.
  - **Pros:** Able to store all needed fields stored in **COMPal** to be exported in files which can be used to import to **COMPal** application.
  - **Cons:** Only able to export and import files generated from **COMPal** application.

Alternative 1 was chosen because by using `iCal4j`, in which we are reusing tried-and-tested components, the robustness of **COMPal** can be enhanced while reducing the manpower and time requirement. Additionally, the ability to export into `iCalendar` file format which allows the users to import to any external calendar application enhances the usability of **COMPal**. Furthermore, a workaround to the **alternative 1** cons would be inputting priority field in ICS description field.

5.7.3 Future Implementation

Though the current implementation has much-allowed reusability of reliable open-source code and also allow COMPal to iCalendar files. There are still possible enhancement for COMPal to take advantage of ical4j library and the export command as currently we only create iCalendar `vevents` components.

1. Add `Todo` interface to the current **tasks** Model to allow the creation of `vtodo` components to export to other calendar application.
2. Improved `reminder` function and update the **task** model to create `valarm` component for **tasks** that have reminders.

5.8. Find Free Slot Feature

This feature allows users to find a free time slot of a specified duration on a specified date. This section will detail how this feature is implemented.

5.8.1. Current Implementation

Command: `findfreeslot`

Upon invoking the `findfreeslot` command with valid parameters (refer to User Guide for `findfreeslot` usage), a sequence of events is then executed.

For clarity, the sequence of events will be in reference to the execution of a `findfreeslot /date 12/11/2019 /hour 1 /min 30` 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 `findfreeslot /date 12/11/2019 /hour 1 /min 30` 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 `findfreeslot` command which in this case, is `FindFreeSlotCommandParser`.
4. Once the parsing is done, `FindFreeSlotCommandParser` would instantiate the `FindFreeSlotCommand` object which would be returned to the `LogicManager`.
5. `LogicManager` is then able to invoke the `commandExecute` function of the returned `FindFreeSlotCommand` object.
6. In the `commandExecute` function of the `FindFreeSlotCommand` object, **task** data will be retrieved from the `TaskList` component.
7. Now that the `FindFreeSlotCommand` object has the task data, it invokes the `sortTask` method to sort the **tasks** in chronological order.
8. The `getFreeSlots` function is then invoked to obtain the list of free time slots stored in an array list of strings.
9. With the output returned from the `getFreeSlots` function, the `CommandResult` object will be instantiated.
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.

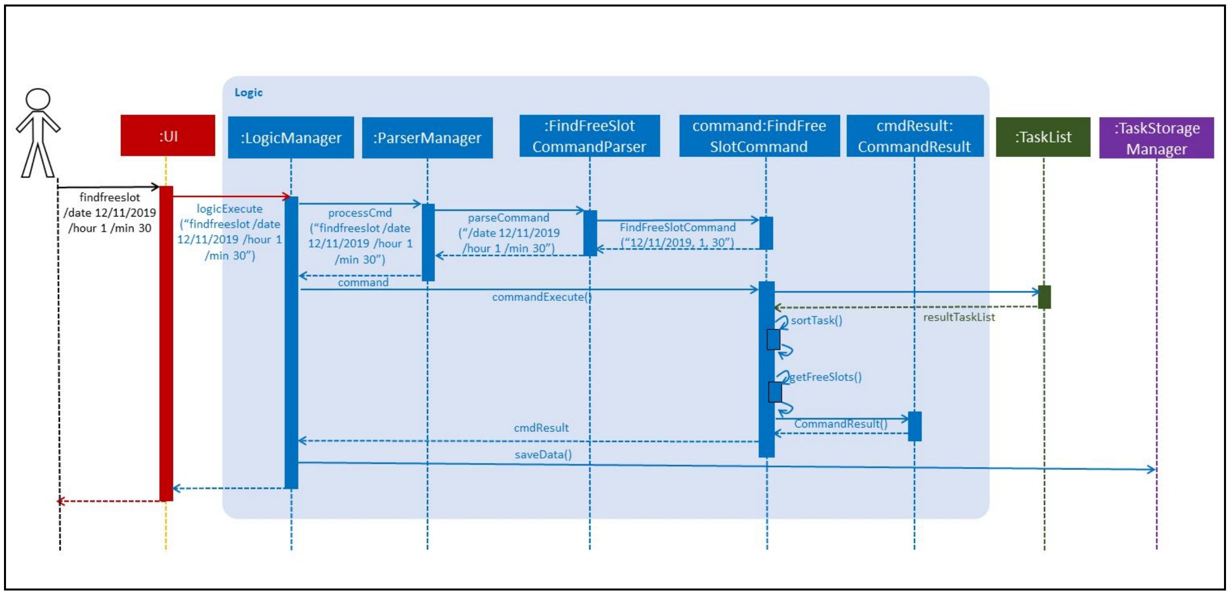
Logic
:UI
:LogicManager
:ParserManager
:FindFreeSlot CommandParser
command:FindFree SlotCommand
cmdResult: CommandResult
:TaskList
:TaskStorage Manager
findfreeslot /date 12/11/2019 /hour 1 /min 30
logicExecute ("findfreeslot /date 12/11/2019 /hour 1 /min 30")
processCmd ("findfreeslot /date 12/11/2019 /hour 1 /min 30")
parseCommand ("/date 12/11/2019 /hour 1 /min 30")
FindFreeSlotCommand ("12/11/2019, 1, 30")
command
commandExecute()
resultTaskList
sortTask()
getFreeSlots()
CommandResult()
cmdResult
saveData()

Figure 10. Sequence Diagram executing the `findfreeslot` command.

5.8.2. Design Considerations

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

Aspect: Data structure to support the functionality of `findfreeslot` command

- **Alternative 1 (current choice):** Use 2 pointers to keep track of the end time of the previous event and the start time of the next event to calculate the duration of the free time slot available.
  - **Pros:** Much more efficient, takes up less memory space.
  - **Cons:** More difficult to implement and prone to bugs.
- **Alternative 2:** Use a 2-Dimensional Boolean array of size 24 by 60 to store the 1440 minutes in a day. Use a for-loop to loop through all events and mark the respective array entries as true for the time slot of the events.
  - **Pros:** Can be implemented easily.
  - **Cons:** Inefficient and takes a much longer time to loop through the events. Since the **task** list can potentially contain a large number of events, looping through the events one by one may slow down the application by a significant amount. More memory space is used up as well.

Alternative 1 was chosen as it is more efficient and takes up less memory space. Having a **task** list with a large number of events will not increase the computational time by a huge proportion, and the memory space needed for 2 pointers is small. In contrast, alternative 2's implementation will result in a very slow application, which is not ideal. Storing a 2-Dimensional array of size 24 by 60 is also more space consuming.

5.8.3 Future Implementation

1. Allow the user to input a period of days/weeks/months in which he wants to find a free time slot, instead of the current implementation of one day. Example: If the user inputs a start date and an end date, together with a duration, the output will be all the available time slots with the duration from the start date until the end date.

6. Dev Ops

6.1. Build Automation

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

6.2. Continuous Integration

We use Travis CI and Codacy to perform *Continuous Integration* on our projects.

6.3. 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** and **events** but also by perceived **priorities**
- want to add tasks with ***flexible commands*** that allow ***recurring and overlapping*** **events** or **deadlines** easily.
- view daily tasks to be done in an ***intuitive manner***.
- want to be reminded of upcoming **deadlines** and **events**
- prefer interacting with a **CLI**

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 a command to edit a task along with the TaskID, followed by the parameters which are 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 is 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** can add up to 500 tasks at one go.
3. **COMPal** must respond fast, within 2 seconds so that the user does not have to wait too long.
4. **COMPal** system application should take up relatively little space on the local machine.
5. **COMPal**’s **GUI** must be intuitive and pleasant to the eyes.
6. **COMPal** consistently performs a specified function without failure.
7. The user’s **OS** date and time must be correctly synchronized 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.

iCalendar: The Internet Calendaring and Scheduling Core Object Specification is a **MIME** type which allows users to ***store and exchange calendaring*** and ***scheduling information*** such as events, to-dos, journal entries, and free/busy information. Files formatted according to the specification usually have an **extension** of .ics.

Appendix F: Instruction for Manual Testing

For the test version: **COMPal** will generate test data on load if this is your first startup! If you would like **COMPal** to regenerate test data, simply clear all stored tasks in `tasks.txt` file found in **COMPal** folder.

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***
    1. Download the jar file and copy into an empty folder
    2. 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

1. ***Adding one task with deadline type***
    1. Test case: `deadline cs2113T ppp /date 11/11/2019 /end 2359`
       Expected: Add a deadline task with description cs2113T ppp and end date 11/11/2019 and end time 23:59 with priority by default set to low
    2. Test case: `deadline cs2113T ppp /date 11/11/2019 /end 2359 /priority high`
       Expected: Add a deadline task with description cs2113T ppp and end date 11/11/2019 and end time 23:59 with priority set to high
2. ***Add recursive tasks with deadline type***
    1. Test case: `deadline lecture /date 11/11/2019 /end 2359 /final-date 03/12/2019`
       Expected: Add recursive deadline tasks with description lecture and recurse weekly from 11/11/2019 with end time 23:59 till 03/12/2019 with priority all by default set to low
    2. Test case: `deadline lecture /date 11/11/2019 /end 2359 /final-date 03/12/2019 /interval 2`
       Expected: Add recursive deadline tasks with description lecture and recurse in every 2 days from 11/11/2019 with end time 23:59 till 03/12/2019 with priority all by default set to low

iii. Test case: `deadline lecture /date 11/11/2019 12/11/2019 /end 2359 /final-date 03/12/2019 /interval 2`
Expected: Add recursive deadline tasks with description lecture and recurse in every 2 days from 11/11/2019 with end time 23:59 till 03/12/2019 and Add recursive deadline tasks with description lecture and recurse in every 2 days from 12/11/2019 with end time 23:59 till 03/12/2019 with priority all by default set to low

iv. Test case: `deadline lecture /date 11/11/2019 12/11/2019 /end 2359 /final-date 03/12/2019 /interval 2 /priority high`
Expected: Add recursive deadline tasks with description lecture and recurse in every 2 days from 11/11/2019 with end time 23:59 till 03/12/2019 and Add recursive deadline tasks with description lecture and recurse in every 2 days from 12/11/2019 with end time 23:59 till 03/12/2019 with priority all set to high

F.3. Editing a task

1. ***Editing descriptions of tasks already stored in COMPal***
   Prerequisites: Add *tasks* or *deadline* using the `deadline` or `event` command for any day
   i. Test case: `edit /id 0 /description this is new!`
      Expected: The task with id 0 has a new description of 'this is new!'

2. ***Editing dates and times of tasks already stored in COMPal***
   Prerequisites: Add *tasks* or *deadline* using the `deadline` or `event` command for any day
   i. Test case: `edit /id 0 /date 29/11/2019`
      Expected: The task with id 0 has a new date of 29/11/2019
   ii. Test case: `edit /id 0 /start 1000`
      Expected: The task with id 0 has a new start time of 1000

3. ***Editing priorities of tasks already stored in COMPal***
   Prerequisites: Add *tasks* or *deadline* using the `deadline` or `event` command for any day
   i. Test case: `edit /id 0 /priority high`
      Expected: The task with id 0 has a new priority level of high

4. ***Editing multiple fields simultaneously of tasks already stored in COMPal***

Prerequisites: Add *tasks* or *deadline* using the `deadline` or `event` command for any day

i. Test case: `edit /id 0 /priority high /date 29/10/2019 /description new desc`
Expected: The task with id 0 has a new priority level of high, a new date of 29/10/2019 and a new description of 'new desc'

F.4. Searching for a task

1. ***Finding a stored task in COMPal***
Prerequisites: Add *tasks* or *deadline* using the `deadline` or `event` command for any day
    i. Test case: `find CS2113T`
    Expected: Tasks with the string 'CS2113T' in their description is displayed to the user

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.
    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 11/11/2019`
    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 week.
    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 11/11/2019 /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.6. 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 deadline /status due`
   Expected: List of all deadlines that are stored in COMPal and are incomplete and overdue past end date.

F.7. Changing tasks status

1. ***Marking a task 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 task 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 has 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.8. Setting reminders

1. ***Setting reminder for a task***
   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: `set-reminder /id 0 /status Y`
   Expected: A confirmation message that states that COMPal has changed the reminder status of the task
   II. Other incorrect set-reminder commands to try: `set-reminder /id <out-of-range> /status Y`
   Expected: Error message returned which shows that the `taskID` does not exist.

F.9. Viewing reminders

1. ***Viewing task reminders***
   I. Test case: `view-reminder`
   Expected: A list of undone tasks that are overdue or due within the week, or have reminder status on.

F.10. Finding free time slots

1. ***Finding free time slots in a day***
   I. Test case: `findfreeslot /date 13/11/2019 /hour 3 /min 0`
   Expected: A list of free time slots on 13/11/2019 that have a duration of at least 3 hours
   II. Other incorrect findfreeslot commands to try: `findfreeslot /date 13/11/2019 /hour 25 /min 0`
   Expected: Error message returned indicating that the duration input is out of range.

F.11. Deleting a task

1. ***Deleting tasks***
   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: `delete /id 0`
   Expected: A confirmation message that states that COMPal has deleted the task

II. Other incorrect delete commands to try: `delete /id <out-of-range>`
Expected: Error message returned which shows that the `taskID` does not exist.

F.12. Exporting schedule into iCalendar file

1. ***Exporting a schedule***
   Prerequisites: Add *tasks* or *deadline* using the `deadline or event` command for any day.
   I. Test case: `export /file-name myCal`
   Expected: A confirmation message that states that **COMPal** have export schedule to myCal.ics .

F.13. Importing iCalendar file

1. ***Importing a .ics schedule***
   Prerequisites: Exported a .ics schedule from iCalendar file generated from **COMPal** and ensure that the file is in the same folder as **COMPal** launch application
   I. Test case: `import /file-name myCal`
   Expected: A confirmation message that states that **COMPal** have import myCal.ics schedule to COMPal.

F.14. Viewing Help

1. ***Viewing a list of all commands***
   I. Test case: `help`
   Expected: A list of all commands available
2. ***Viewing the usage for some specific command***
   I. Test case: `help deadline`
   Expected: Usage of deadline command with example