

OOF (Outstanding Organisation Friend) - Developer Guide

1. Introduction	3
1.1. What is <i>OOF</i> ?	3
1.2. What is the purpose of this guide?	4
1.3. Acknowledgements	4
2. Setting up	4
2.1. Prerequisites	4
2.2. Setting up the project in your computer	4
2.3. Verifying the setup	5
3. Design	5
3.1. Architecture	5
3.2. UI component	6
3.3. Logic component	6
3.4. Model component	12
3.5. Storage component	12
3.6. Common classes	13
4. Implementation	13
4.1. Recurring task feature	13
4.1.1. Implementation	13
4.1.2. Design Considerations	15
4.2. Help feature	16
4.2.1. Implementation	18
4.2.2. Design Considerations	20
4.3. View tasks for the week feature	21
4.3.1. Implementation	21
4.3.2. Design Considerations	23
4.4. View calendar for a month feature	24
4.4.1. Implementation	24
4.4.2. Design Considerations	26
4.5. Find free time slots feature	27
4.5.1. Implementation	28
4.5.2. Design Considerations	31
4.6. Task Tracker feature	32
4.6.1. Implementation	32
4.6.2. Design Considerations	38
5. Documentation	39
5.1. Introduction	39

5.2. Editing Documentation	39
5.3. Editing diagrams	39
5.4. Publishing Documentation	39
5.4.1. Setting up Travis CI	39
5.4.2. Enabling auto-publishing of documentation	41
5.4.3. Converting Documentation to PDF format	43
6. Testing	43
6.1. Troubleshooting Testing	43
7. Dev Ops	44
7.1. Build Automation	44
7.2. Continuous Integration	44
7.3. Coverage Reporting	44
7.4. Documentation Previews	44
7.5. Making a Release	44
Appendix A: Product Scope	44
Appendix B: Requirements	45
B.1. User Stories	45
B.2. Use Cases	47
B.3. Non Functional Requirements	55
Appendix C: Glossary	56
Appendix D: Instructions for Manual Testing	56
D.1. Managing Semesters	56
D.1.1. Adding a semester	56
D.1.2. Deleting a semester	56
D.1.3. Selecting a semester	57
D.2. Managing Modules	57
D.2.1. Adding a module	57
D.2.2. Deleting a module	57
D.2.3. Selecting a module	58
D.3. Managing Lessons	58
D.3.1. Adding a lesson	58
D.3.2. Deleting a lesson	58
D.4. Managing Tasks	58
D.4.1. Adding a todo task	59
D.4.2. Adding a deadline task	59
D.4.3. Adding an assignment task	59
D.4.4. Adding an event task	59
D.4.5. Adding an assessment task	60
D.4.6. Deleting a task	60
D.5. Starting Task Tracker	60
D.5.1. Understanding the parameters	60

D.5.2. Testing the command	60
D.6. Stopping Task Tracker	60
D.6.1. Understanding the parameters.....	61
D.6.2. Testing the command	61
D.7. Pausing Task Tracker	61
D.7.1. Understanding the parameters.....	61
D.7.2. Testing the command	61
D.8. Viewing Task Tracker Diagram	61
D.8.1. Understanding the parameters.....	61
D.8.2. Testing the command	61
D.9. Viewing a list of Task Trackers.....	61
D.9.1. Understanding the parameters.....	61
D.9.2. Testing the command	62
D.10. Deleting a Task Tracker	62
D.10.1. Understanding the parameters.....	62
D.10.2. Testing the command	62
D.11. Setting recurring tasks	62
D.11.1. Selecting the task to be recurred	62
D.11.2. Choosing the number of recurrences and frequency	62
D.11.3. Entering the command	62
D.12. Finding free time slots	63
D.12.1. Understanding the parameters.....	63
D.12.2. Testing the command	63
D.13. Viewing your tasks in a calendar format.....	63
D.13.1. Understanding the parameters.....	63
D.14. Viewing your tasks for a week in a tabular format.....	63
D.14.1. Understanding the parameters.....	63
D.14.2. Testing the command	64

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

1. Introduction

1.1. What is *OOF*?

OOF, short for Outstanding Organisation Friend, is a command-line interface desktop application built to improve University Students' productivity and efficiency by reducing the need to manually organise tasks.

1.2. What is the purpose of this guide?

This guide aims to provide information for you, future contributors of **OOF** so that you can have an easy reference for understanding the features implemented in **OOF**.

1.3. Acknowledgements

Original source: [PersonalAssistant-Duke](#) created by [SE-EDU initiative](#)

2. Setting up

This section will show you the requirements that you need to fulfill in order to quickly start contributing to this project in no time!

2.1. Prerequisites

1. **JDK 11** or above



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

2. **IntelliJ** IDE



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

2.2. Setting up the project in your computer

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

2.3. Verifying the setup

1. You can run `Oof` and try a few commands
2. You can also run tests using our instructions for manual testing to explore our features.

3. Design

3.1. Architecture

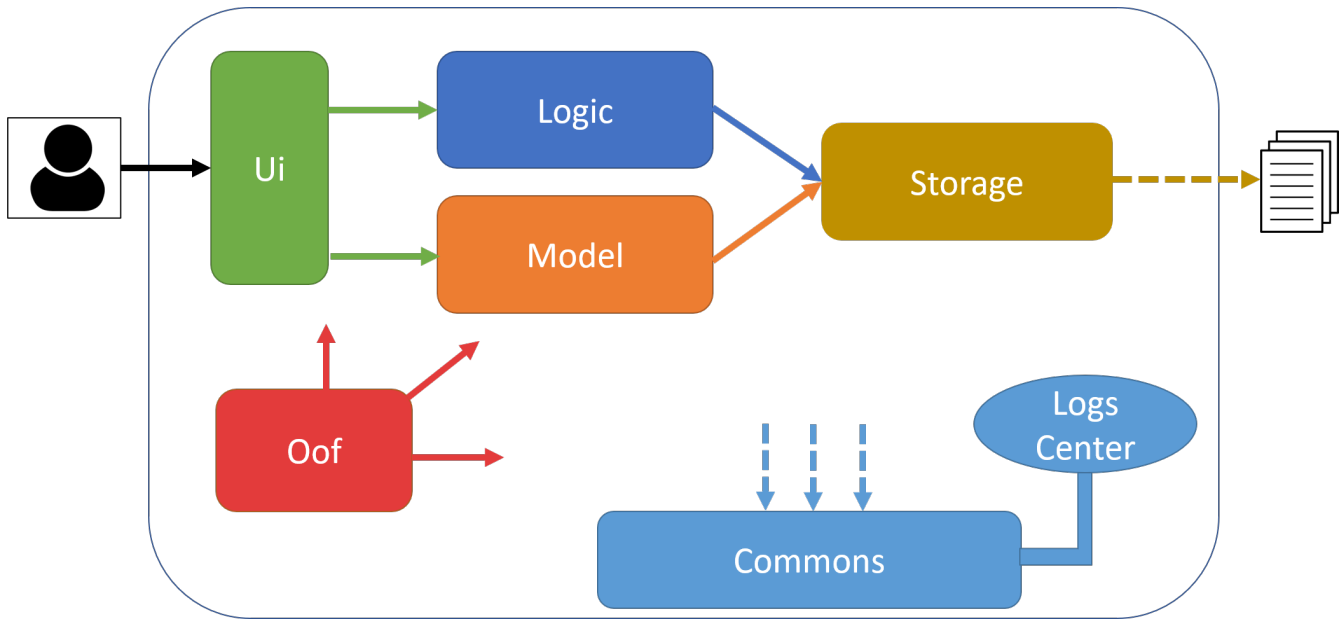


Figure 1. Architecture Diagram

The **Architecture Diagram** shown above depicts the high-level construct of **OOF**. Given below is a quick overview of each component.

- **Oof** has only one class called **Oof** that is responsible for:
 - Bootstrapping process for initialising instances of classes in the **Ui**, **Storage** and **Command** packages.
 - Handling your input during runtime and terminating the program when you wish to exit from **OOF**.
- The **Ui** package is responsible for visual feedback and taking in your input.
- The **Logic** package contains all of **OOF's** commands in the subpackage **command**, the **CommandParser** and **Reminder** classes.
- The **Model** package contains all the object containers that are used by our **commands**.
- The **Storage** package contains classes to help store all your data to the hard disk.
- The **Commons** package contains the subpackage **command** which holds all the customised **exception** classes for all our commands, followed by miscellaneous **exception** classes for non-command exceptions.



Logging is implemented in our project to facilitate the checking of bugs and error messages. Thus, the **Commons** package that is being utilised by all our classes is linked to the logs center to show that the handled exceptions we have caught are properly logged.

3.2. UI component

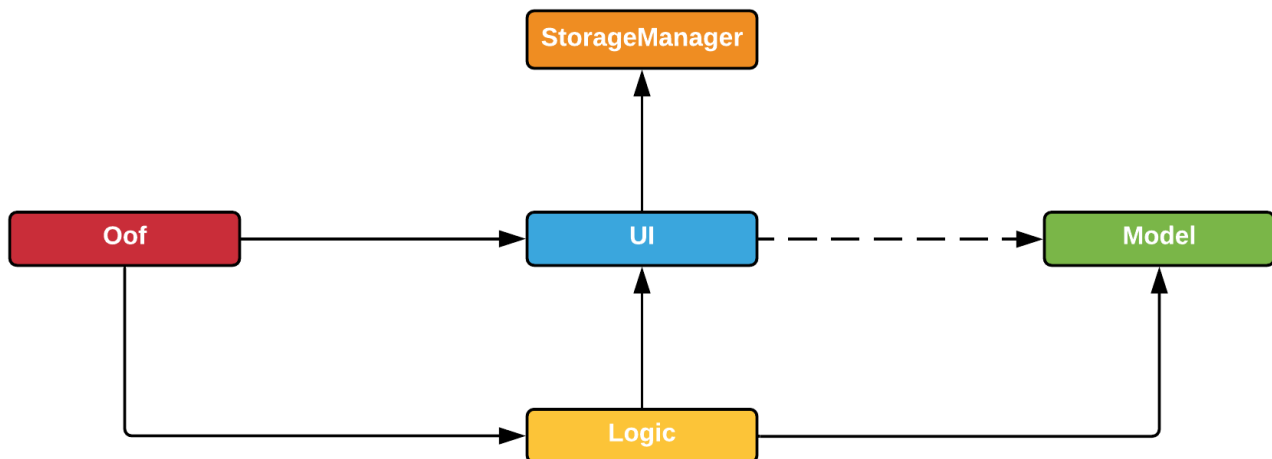


Figure 2. Class Diagram for Ui Component

The **Class Diagram** above shows the different interactions of the **Ui** component when printing output.

The **Ui** component is responsible for:

- Taking in and executing user commands via the **Logic** component.
- Listening for changes to **Model** data so that the **UI** component can be updated with the latest data.
- Displaying output to the user.

3.3. Logic component

The **Class Diagram** illustrates the relationship between the individual components of the **Logic** component.

The **Logic** component consists of the **command** subpackage alongside the **CommandParser** and **Reminder** classes.

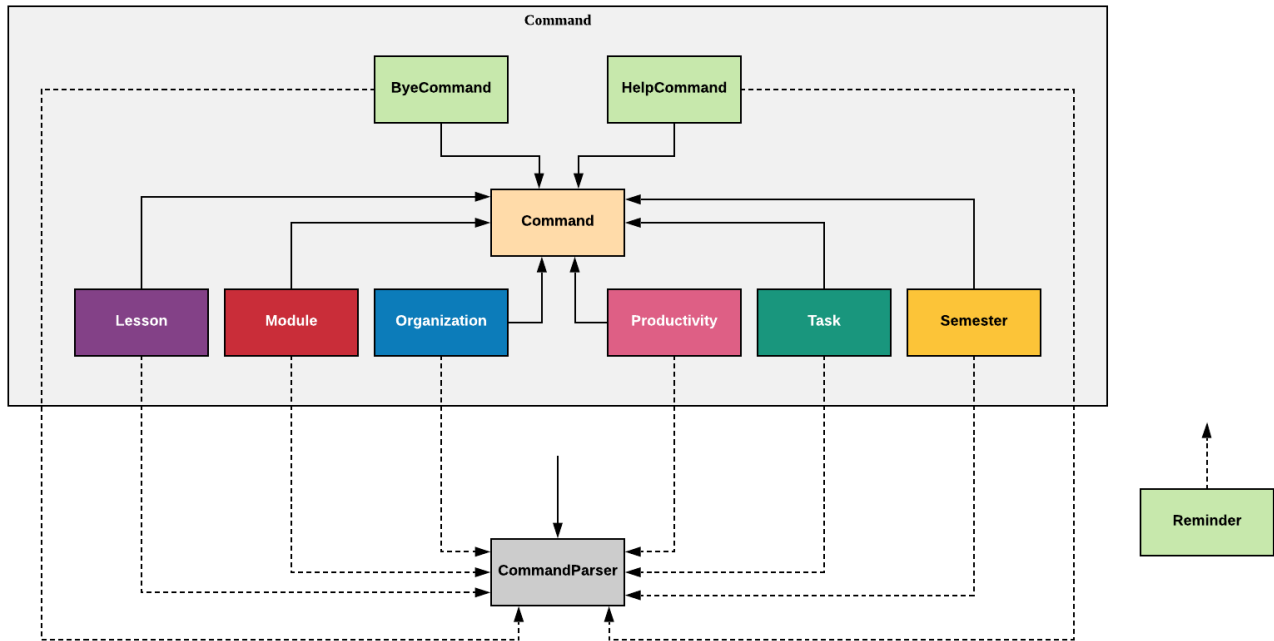


Figure 3. Structure of the Logic Component

The `command` subpackage consists of the following classes and subpackages.

- `HelpCommand` class
- `ByeCommand` class
- `productivity` subpackage
- `task` subpackage
- `semester` subpackage
- `organization` subpackage
- `module` subpackage
- `lesson` subpackage

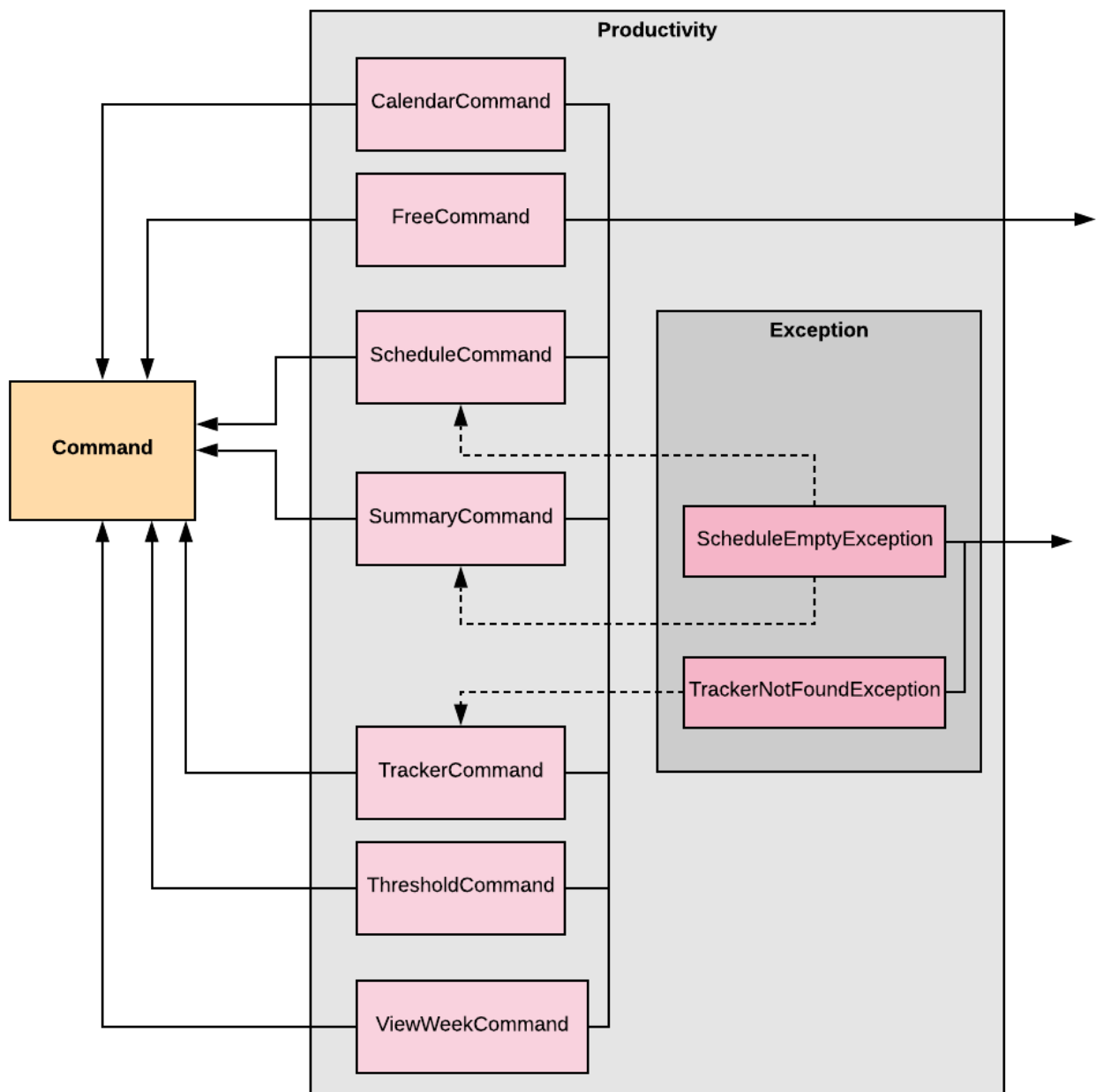


Figure 4. Structure of the productivity subpackage in the Logic Component

The **Class Diagram** above illustrates the relationship between the individual components of the **productivity** subpackage in the **Logic** component.

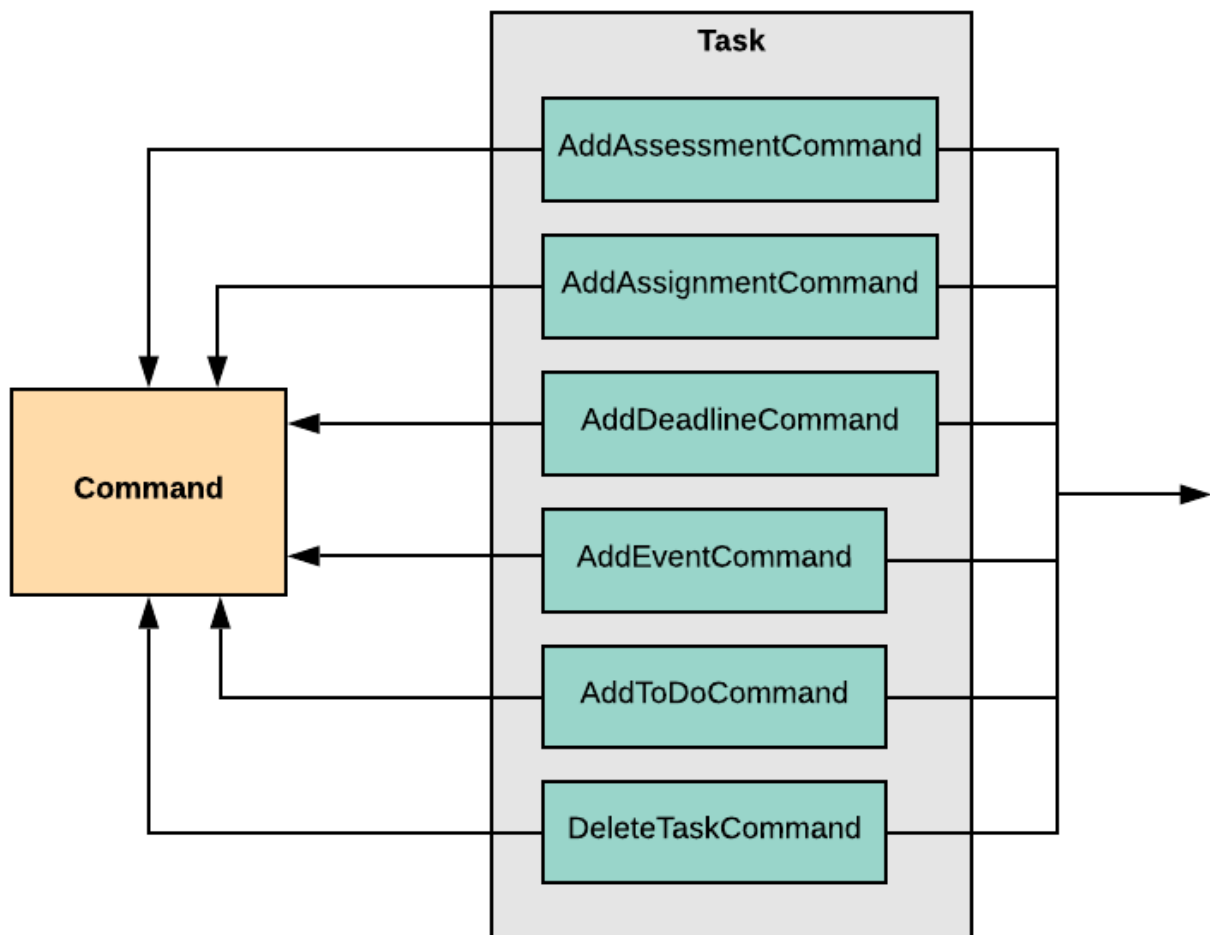


Figure 5. Structure of the task subpackage in the Logic Component

The **Class Diagram** above illustrates the relationship between the individual components of the **task** subpackage in the **Logic** component.

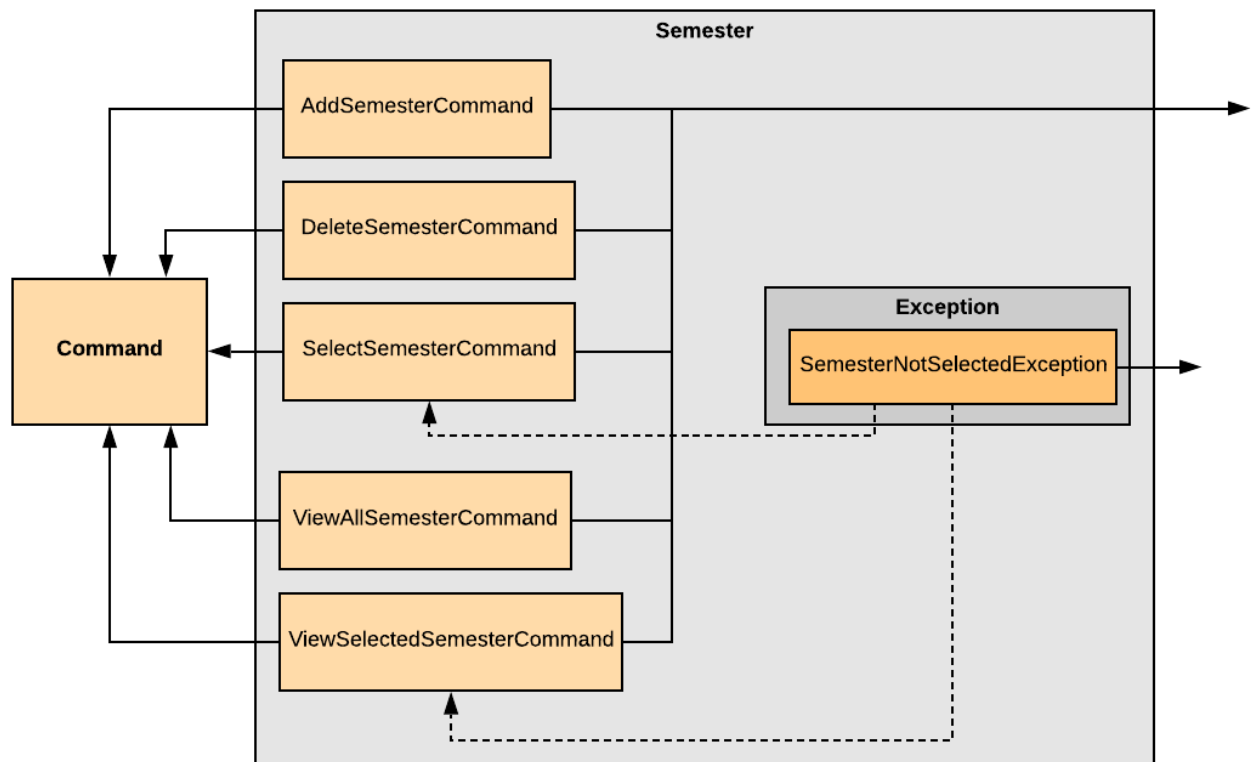


Figure 6. Structure of the semester subpackage in the Logic Component

The **Class Diagram** above illustrates the relationship between the individual components of the **semester** subpackage in the **Logic** component.

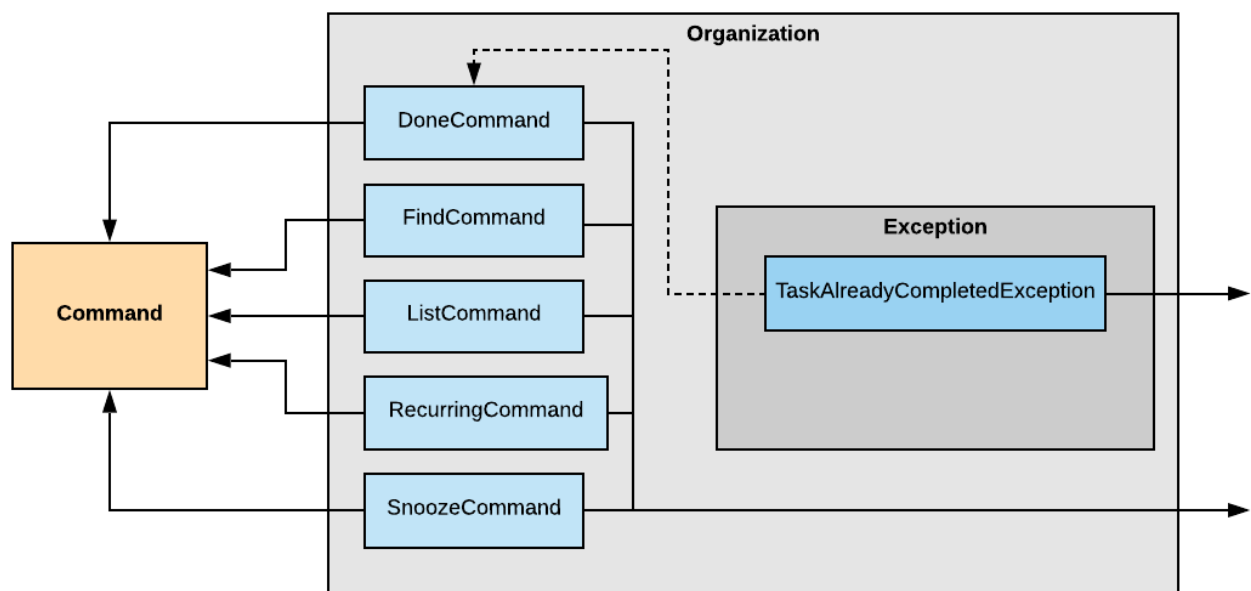


Figure 7. Structure of the organization subpackage in the Logic Component

The **Class Diagram** above illustrates the relationship between the individual components of the **organization** subpackage in the **Logic** component.

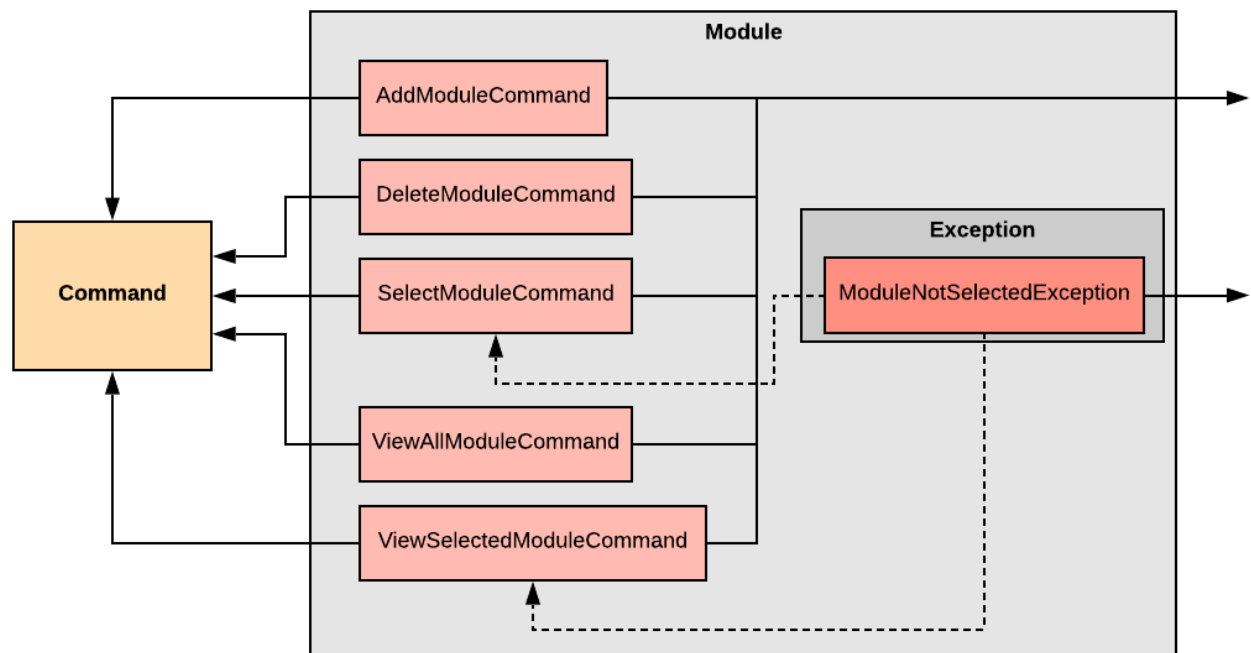


Figure 8. Structure of the module subpackage in the Logic Component

The **Class Diagram** above illustrates the relationship between the individual components of the **module** subpackage in the **Logic** component.

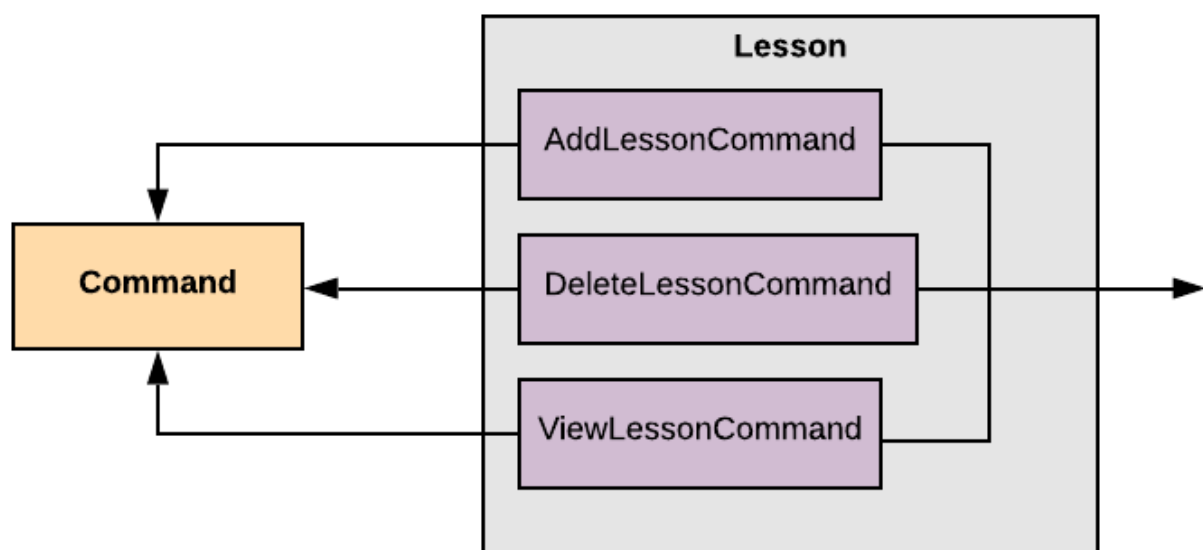


Figure 9. Structure of the lesson subpackage in the Logic Component

The **Class Diagram** above illustrates the relationship between the individual components of the **lesson** subpackage in the **Logic** component.

The **Logic** component is responsible for:

- Executing user commands.
- Listening for changes to **Model** data so that the **Logic** component can be updated as expected.

- Displaying output to the user via the **Ui** component

3.4. Model component

The **Model** component consists of the **task** and **university** packages and shows how they are associated with each other.

The class diagram below illustrates the relationship between the individual components of the **Model** component.

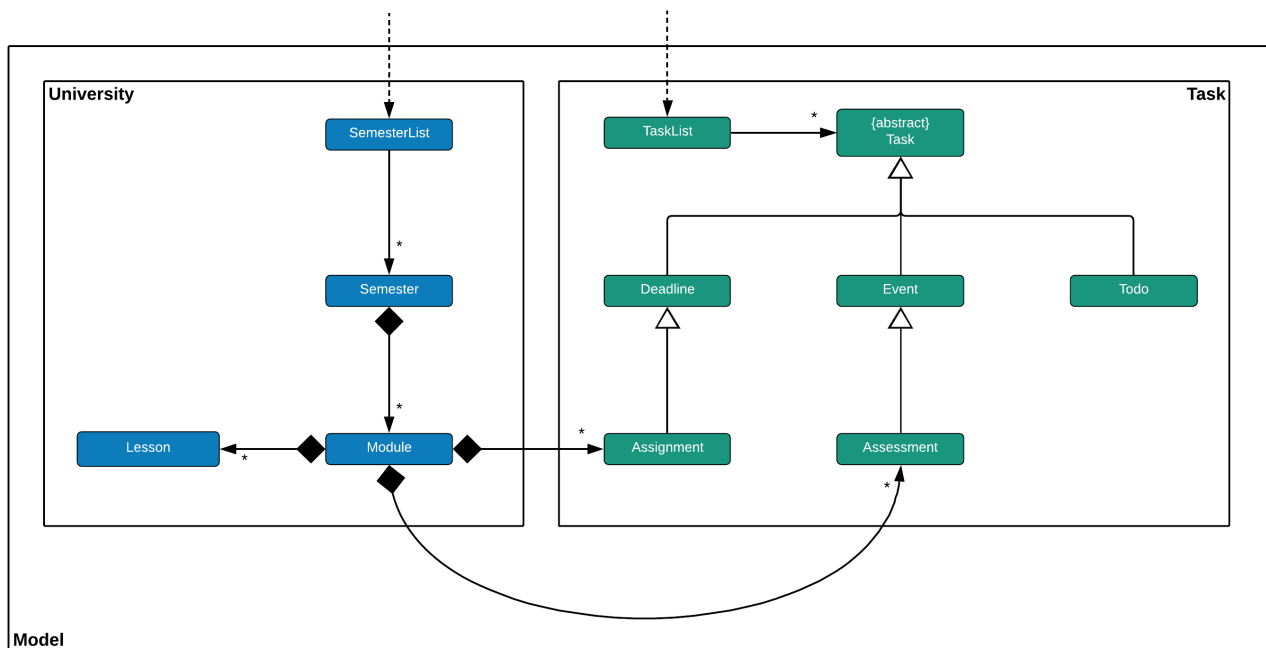


Figure 10. Structure of the Model Component

The figure shows the individual components of the **Model** component. The **University** component is modelled after real-world university curriculum structure.

The **Model** stores:

- a **SemesterList** object that contains individual **Semester** objects. Each **Semester** object consists of **Module** objects that represents a module that a University student takes and each **Module** object can contain any number of **Lesson**, **Assignment** and **Assessment** objects.
- a **TaskList** object that contains **Task** objects. A **Task** object can be any of **Deadline**, **Event** and **Todo** as they represent different categories of tasks. **Assignment** and **Assessment** inherits from **Deadline** and **Event** respectively and represent the tasks that University student will have.

When either **SemesterList** or **TaskList** is changed, the system will update the persistent storage via the **Storage** component, which will be explained in the next section.

3.5. Storage component

The **Storage** component consists of the **Storage**, **StorageManager** and **StorageParser** classes and shows how they are associated with one another.

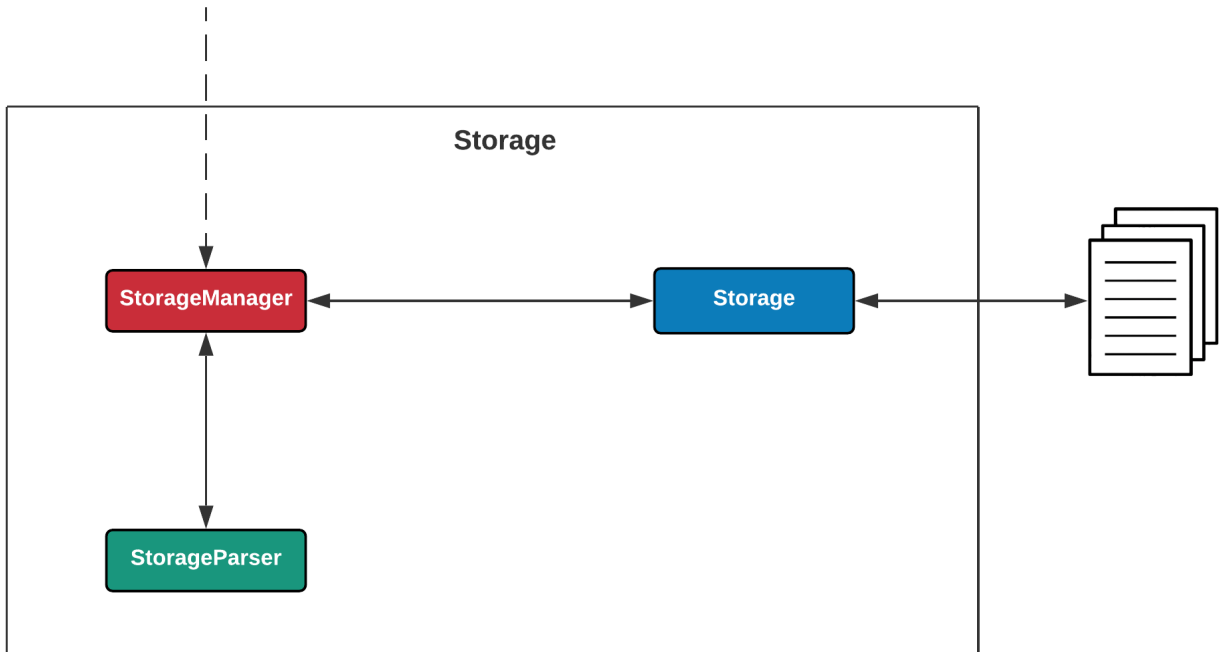


Figure 11. Class Diagram for Storage Component

The **Class Diagram** above illustrates the relationship between the individual classes in the **Storage** component.

The **Storage** component is responsible for:

- Parsing data from/to persistent storage via the **StorageParser** component.
- Loading and writing data from/to persistent storage which is managed by the **StorageManager** component.

3.6. Common classes

Classes used by multiple components are in the **oof.common** package.

4. Implementation

4.1. Recurring task feature

4.1.1. Implementation

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



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

Additionally, it consists of the following features:

- You can select a **Task** in the **TaskList** to be a recurring task.
- You can choose an integer between **1 - 10** inclusive for the number of times the task should recur.
- You can choose an integer between **1 - 4** inclusive for the **Frequency** of recurrence.

The choices are as follows:

1. DAILY
2. WEEKLY
3. MONTHLY
4. YEARLY

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

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

Step 1.

The user types in **recurring 1 1 1**. The **parse** method in **CommandParser** class is called to parse the command to obtain integers **1** as the **Index** of the **Task** in **TaskList**, **1** as the **number of recurrences** and **1** as the **frequency** of recurrence.



Customised **MissingArgumentException** and **InvalidArgumentException** will be thrown if the user enters invalid commands.

Step 2.

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

Step 3.

The **setRecurringTask** method in **RecurringCommand** class is then called by the **execute** method. This method does three main things:

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

Step 4.

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

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

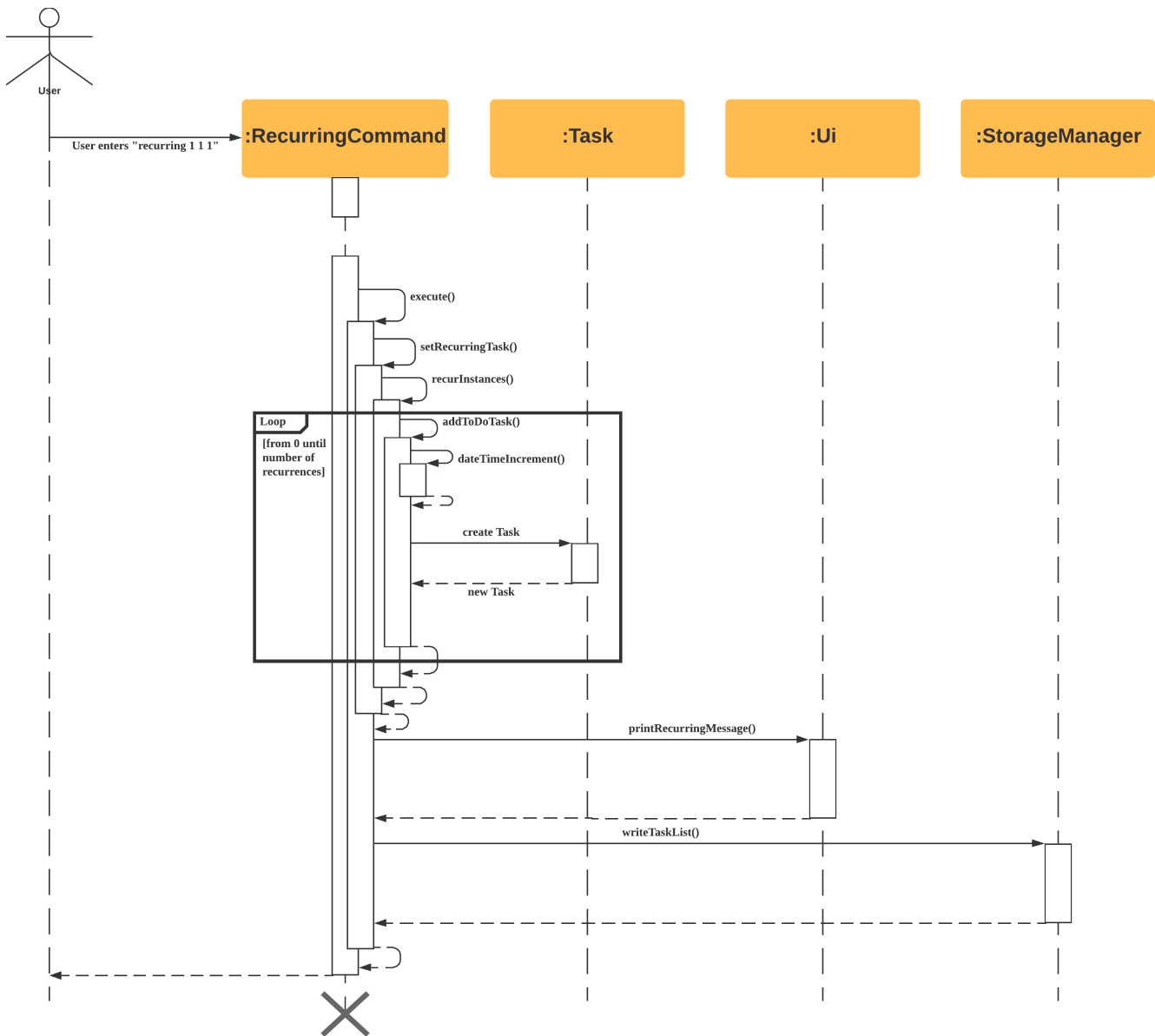


Figure 12. Sequence diagram for Recurring Command

4.1.2. Design Considerations

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

- Alternatives considered:

Insert an upcoming recurring task when the **recurring Task** is nearing. There may be too many **Tasks** to keep track and add when **00F** starts up especially in the case when the number of **Tasks** in the **TaskList** gets potentially large. This decreases the scalability of the project in the long run.

- **Frequency fixed to four different default frequencies**

- Rationale:

It requires significantly less effort to choose from a default list of four options than to manually type in customised time ranges.

- Alternatives considered:

Users can enter a customised **Frequency** for the **recurring Task**. It may be a viable option to allow users to set such parameters. However, since the **00F** program is solely a Command Line Interface program, it may not be user-friendly for users to enter so many details just to set a customised **Frequency** for the **recurring Task**.

4.2. Help feature


```

===== OOF MANUAL =====

NAME
    OOF -- Outstanding Organisation Friend

DESCRIPTION
    The following options are available:

Help                help

Deadline            deadline DESCRIPTION /by DD-MM-YYYY HH:MM

Event              event DESCRIPTION /from DD-MM-YYYY HH:MM /to
DD-MM-YYYY HH:MM

Todo               todo DESCRIPTION /on DD-MM-YYYY

Recurring          recurring INDEX NUMBER_OF_OCCURRENCES FREQUE
NCY

List              list

Done              done INDEX

Delete            delete INDEX

Find              find DESCRIPTION

Threshold          threshold HH

Schedule           schedule DD-MM-YYYY

Summary           summary

Free              free DD-MM-YYYY

ViewWeek          viewweek DD MM YYYY

Calendar          calendar MM YYYY

Add Semester      semester /add YEAR /name SEMESTER /from STAR
T_DATE /to END_DATE

View Semester     semester /view

Delete Semester   semester /delete INDEX

Select Semester   semester /select INDEX

```

Figure 13. Output of Help Command

```
Enter a command:
help deadline

Deadline          deadline DESCRIPTION /by DD-MM-YYYY HH:MM

Enter a command:
```

Figure 14. Output of Individual Help Command

4.2.1. Implementation

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



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

Additionally, it contains the following feature:

- Users may request for `Help` with a specific command.

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

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

Step 1:

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



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

Step 2:

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

- **Step 2a:**

The `readManual` method of `Storage` class will retrieve and read `manual.txt` from persistent storage by using `FileReader` abstraction on `File` abstraction.

- **Step 2b:**

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



`OofManualNotFoundException` will be thrown if `manual.txt` is unavailable.

Step 3:

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

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

- **Step 3a:**

The first segment of each element in the `commands` `ArrayList` will be retrieved by adding a String `command` delimited by two whitespaces.

- **Step 3b:**

Once a check is completed to ensure that `command` is not empty, both `keyword` and `command` String will be formatted through the use of `toUpperCase` function and String comparison will be performed through the use of `equals`. If they match, that particular element of `commands` `ArrayList` will be stored into a String called `description` and the for loop will break before returning `description` to the `execute` method of `HelpCommand`.



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

Step 4:

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

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

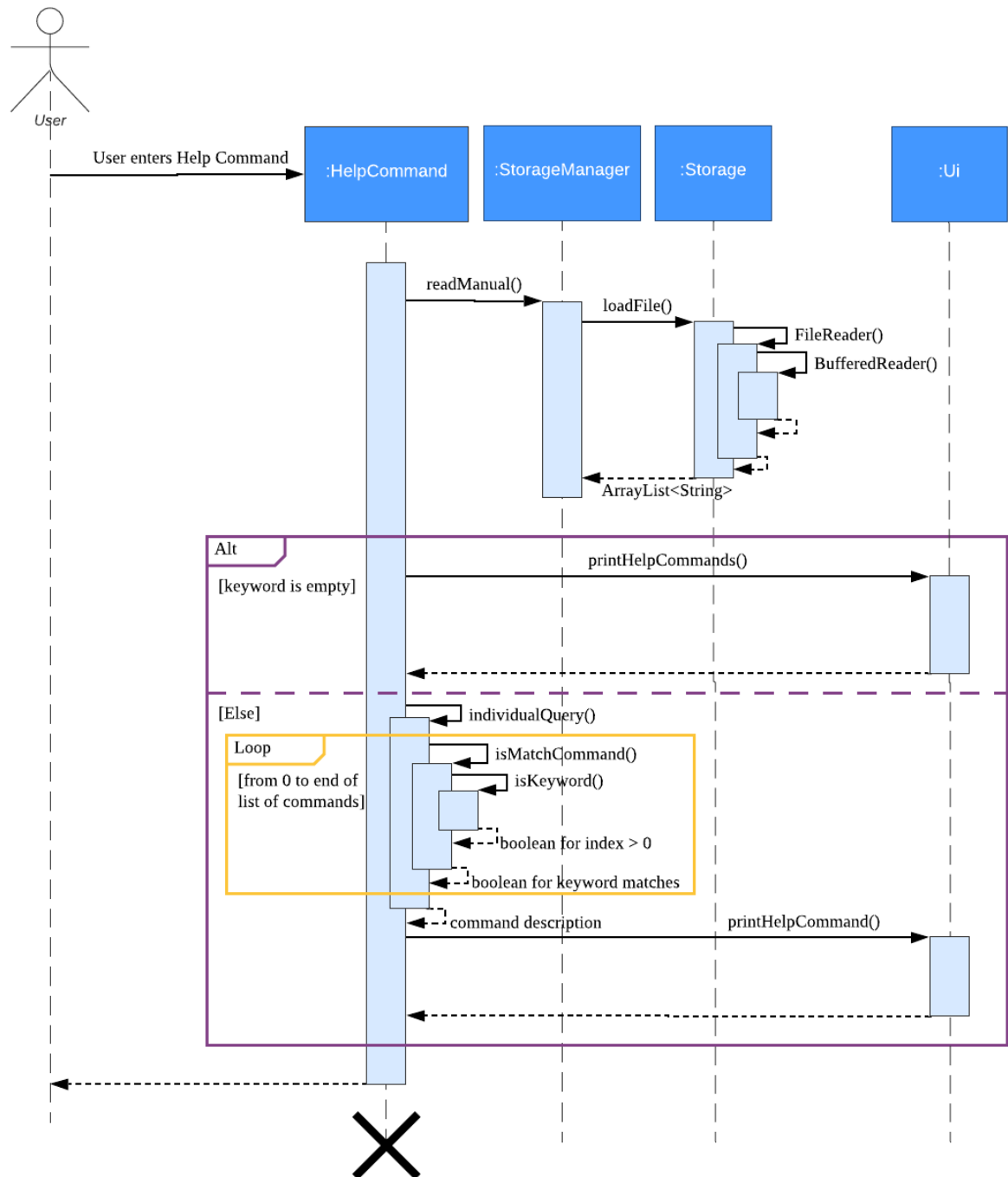


Figure 15. Sequence diagram for Help Command

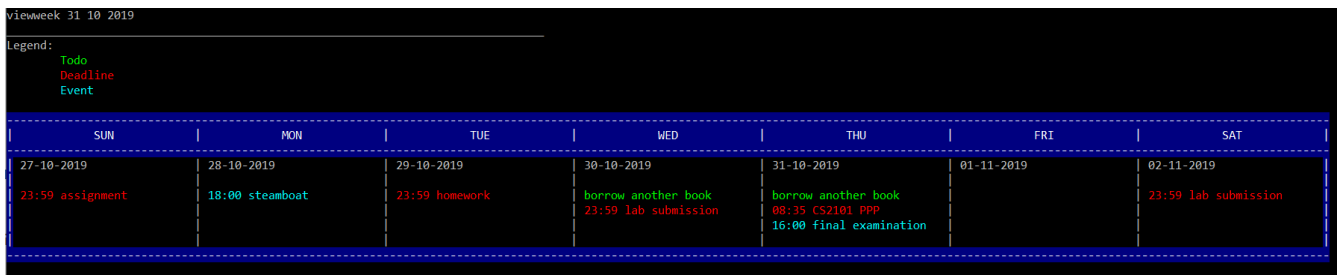
4.2.2. Design Considerations

- Created `manual.txt` to store available commands and their instructions
 - Rationale:
With scalability in mind, the use of persistent storage will grant developers a common location to update the list of `Command` and their instructions.
 - Alternatives Considered:
Numerous String variables can be added to an ArrayList through the `HelpCommand` class. This would not require the use of `File`, `FileReader` or `BufferedReader` abstractions. However, this would bring developers inconvenience during project extension as more functions will be

added and this may eventually lead to unorganised code, especially in the `HelpCommand` class.

- Implement `ArrayList` to display `Help` for an individual command and its instructions
 - Rationale:
The use of `ArrayList` offers flexibility due to its unconfined size. This allows increased convenience and scalability due to the large list of `Command` and their instructions available to our users.
 - Alternatives Considered:
The use of an `Array` will allow increased efficiency given the smaller number of `Command` we had in our earlier versions, such as `v1.1`. However, this is not a beneficial solution in the long run as we create extensions and expand upon `OOF`.

4.3. View tasks for the week feature



```
viewweek 31 10 2019
Legend:
  Todo
  Deadline
  Event
```

SUN	MON	TUE	WED	THU	FRI	SAT
27-10-2019	28-10-2019	29-10-2019	30-10-2019	31-10-2019	01-11-2019	02-11-2019
23:59 assignment	18:00 steamboat	23:59 homework	borrow another book 23:59 lab submission	borrow another book 08:35 CS2101 PPP 16:00 final examination		23:59 lab submission

Figure 16. Output of `ViewWeek` Command

4.3.1. Implementation

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



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

Features elaborated:

- The output of the `ViewWeekCommand` is ANSI colour enabled. This distinguishes the different days of the week in the output.
- The output of `ViewWeekCommand` resizes automatically based on the length of the `description` of tasks.

Legend:
Todo
Deadline
Event

SUN	MON	TUE	WED	THU	FRI	SAT
27-10-2019	28-10-2019	29-10-2019	30-10-2019	31-10-2019	01-11-2019	02-11-2019
23:59 assignment	18:00 steamboat	23:59 homework	borrow another book 23:59 lab submission	borrow another book 08:35 CS2101 PPP 16:00 final examination		23:59 lab submission

Enter a command:
viewweek 13 10 2019

Legend:
Todo
Deadline
Event

SUN	MON	TUE	WED	THU
13-10-2019	14-10-2019	15-10-2019	16-10-2019	17-10-2019
cs2105 cs2106 cs2107 cs2113t cs2101	borrow another book			

Figure 17. Automatic resize feature in ViewWeek Command

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



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

Step 1.

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

Step 2.

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

Step 3.

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

Step 4.

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

Step 5.

The `task` is then added to `calendarTasks` using the `addEntry()` method. After iterating through the current list of tasks, the same logic is applied to the `semesterList` to retrieve appropriate lesson timings via the `parseLessons()` method which calls `queryModules()` and `addLesson()` methods. The `printViewWeek()` method in the `Ui` class is then called to print the tasks for the current week.

Step 6.

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

Step 7.

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

Step 8.

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

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

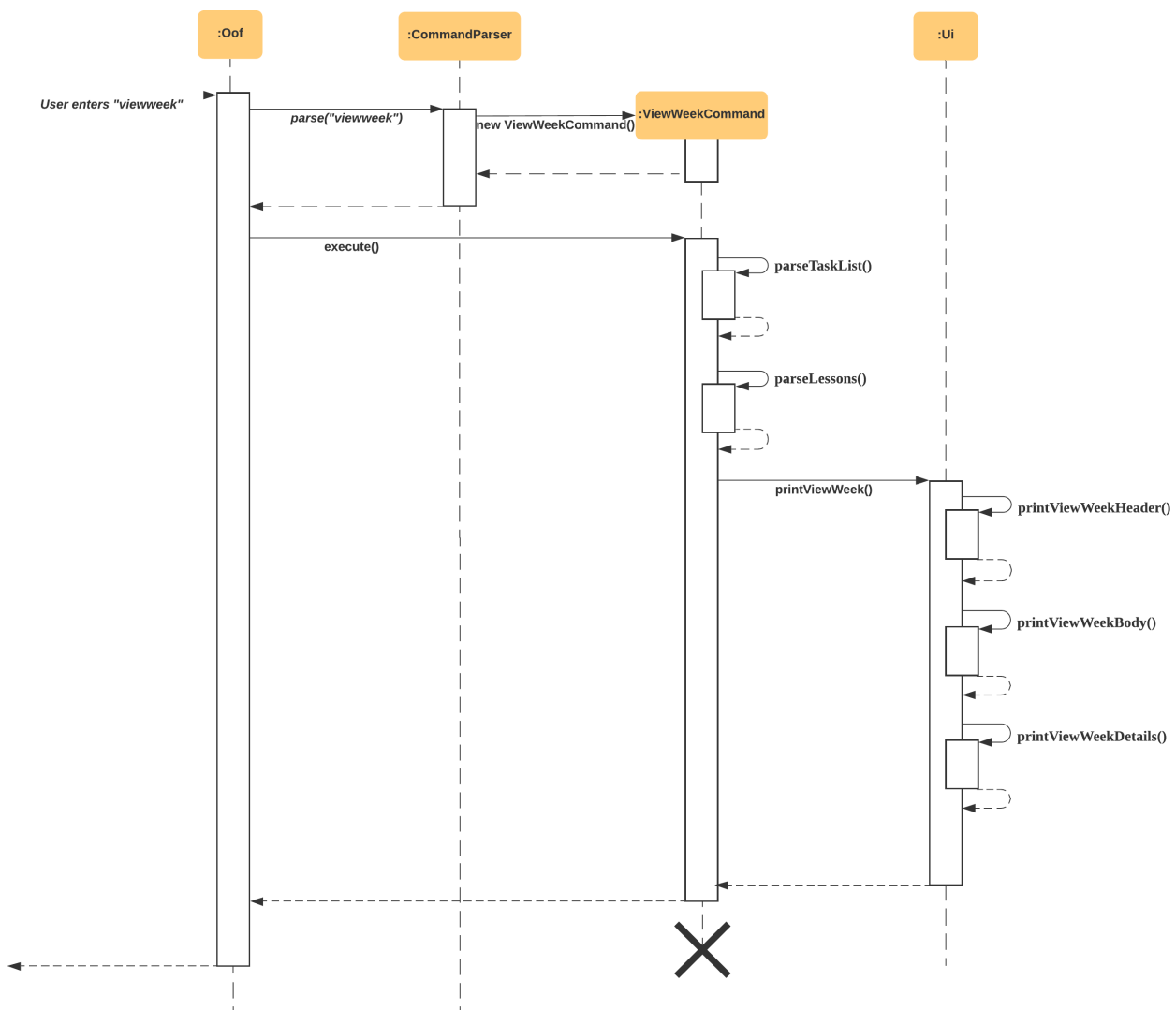


Figure 18. Sequence diagram for ViewWeek Command



Trivial details that are not important in describing the implementation of the feature are left out.

4.3.2. Design Considerations

- Resizing column size instead of wrapping description of tasks

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

4.4. View calendar for a month feature

NOVEMBER 2019						
SUN	MON	TUE	WED	THU	FRI	SAT
					1 10:00 CS2107 Lecture 16:00 CS2113T Lecture	2
3	4 08:00 CS2101 Tutorial 12:00 CS2107 Tutorial 13:00 CS2106 Lab 13:00 CS2106 Lab 14:00 CS2105 Lecture	5 14:00 CS2105 Tutorial	6 14:00 CS2106 Lecture 17:00 CS2113T Tutorial	7 08:00 CS2101 Tutorial	8 10:00 CS2107 Lecture 16:00 CS2113T Lecture	9
10	11 08:00 CS2101 Tutorial 12:00 CS2107 Tutorial 13:00 CS2106 Lab 13:00 CS2106 Lab 14:00 CS2105 Lecture	12 14:00 CS2105 Tutorial	13 14:00 CS2106 Lecture 17:00 CS2113T Tutorial	14 08:00 CS2101 Tutorial	15 10:00 CS2107 Lecture 16:00 CS2113T Lecture	16
17	18 08:00 CS2101 Tutorial 12:00 CS2107 Tutorial 13:00 CS2106 Lab 13:00 CS2106 Lab 14:00 CS2105 Lecture	19 14:00 CS2105 Tutorial	20 13:00 homework 13:00 project meeting 14:00 CS2106 Lecture 17:00 CS2113T Tutorial	21 08:00 CS2101 Tutorial	22 10:00 CS2107 Lecture 16:00 CS2113T Lecture	23 23:59 CS2106 Lab
24	25 08:00 CS2101 Tutorial 12:00 CS2107 Tutorial 13:00 CS2106 Lab 13:00 CS2106 Lab 14:00 CS2105 Lecture	26 14:00 CS2105 Tutorial	27 14:00 CS2106 Lecture 17:00 CS2113T Tutorial	28 08:00 CS2101 Tutorial	29 10:00 CS2107 Lecture 16:00 CS2113T Lecture	30

Figure 19. Sample output of Calendar Command

4.4.1. Implementation

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



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

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

Step 1

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

Step 2

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



A `MissingArgumentException` will be thrown if less than 2 arguments are provided while an `InvalidArgumentException` will be thrown if the `month` argument provided is not an integer or not between 1 and 12. The program handles the exceptions by retrieving the current month and year from the system as arguments for `CalendarCommand`.

Step 3

The `execute` method in the `CalendarCommand` class is then called by the `executeCommand()` method in the `Oof` class.

This method does the following:

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



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

Step 4

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

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

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

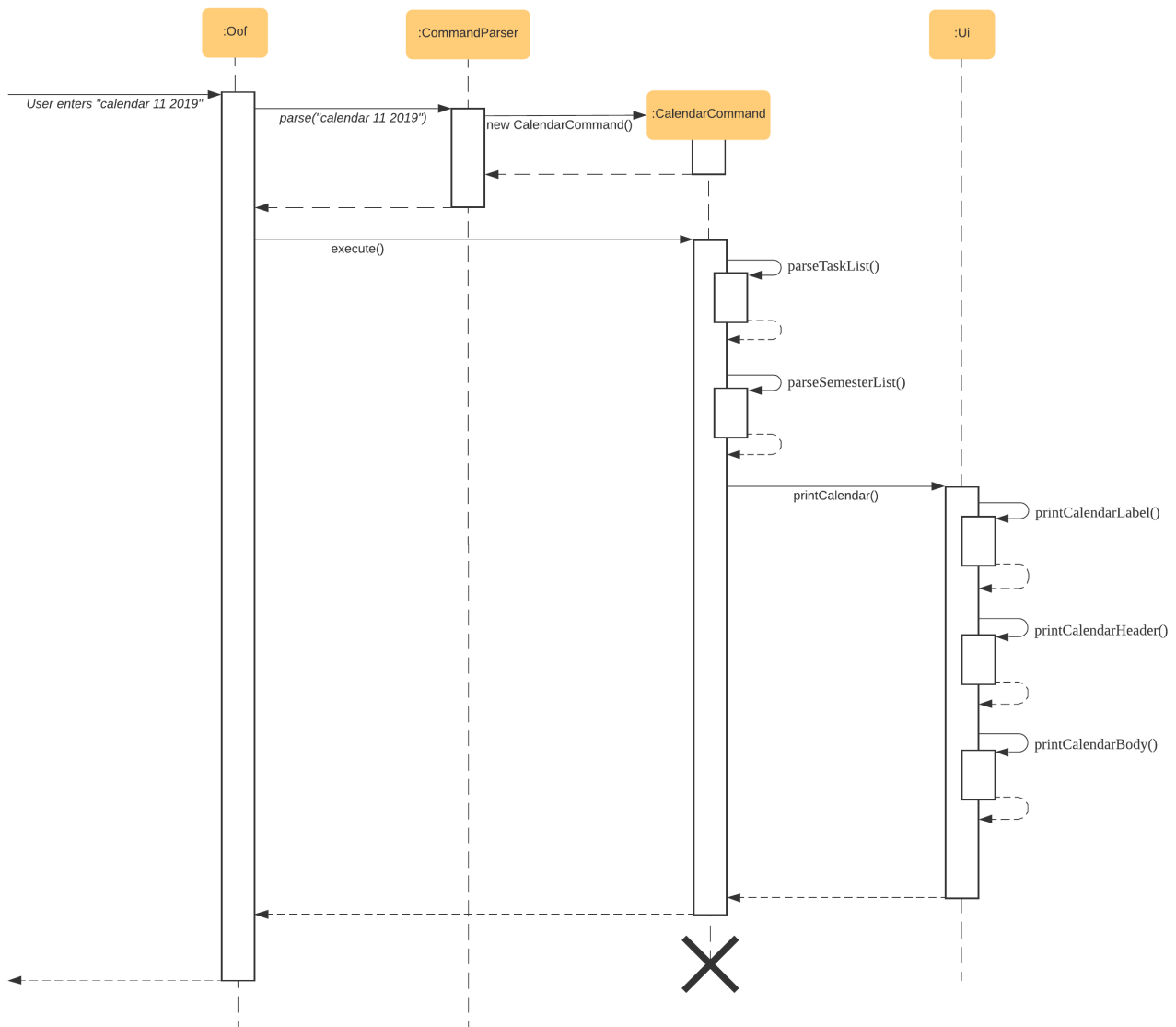


Figure 20. Sequence diagram for Calendar Command

4.4.2. Design Considerations

- Extending row size instead of limiting the number of tasks displayed
 - Rationale:
Limiting the number of tasks displayed might misrepresent the number of **Tasks** a person has for that day.
 - Alternatives considered:
Implementing a GUI which includes a scroll pane for each day such that calendar size can be fixed.
- Truncation of task name instead of extending column size
 - Rationale:
Since row size is extendable, extending column size would severely affect readability when column and row sizes increase independently of each other. Also, **ScheduleCommand** class can

be used in conjunction with `CalendarCommand` to allows the user to view the list of tasks for any date.

- Alternatives considered:

Wrapping of task name which will allow the display of the full task name. Not feasible as it will increase the number of rows further.

4.5. Find free time slots feature

```

free 08-11-2019
-----
|           Friday 08-11-2019           |
-----
| 07:00 - 08:00 |           free           |
-----
| 08:00 - 09:00 |           free           |
-----
| 09:00 - 10:00 |           free           |
-----
| 10:00 - 11:00 |          BUSY           |
-----
| 11:00 - 12:00 |          BUSY           |
-----
| 12:00 - 13:00 |           free           |
-----
| 13:00 - 14:00 |           free           |
-----
| 14:00 - 15:00 |           free           |
-----
| 15:00 - 16:00 |           free           |
-----
| 16:00 - 17:00 |           free           |
-----
| 17:00 - 18:00 |           free           |
-----
| 18:00 - 19:00 |           free           |
-----
| 19:00 - 20:00 |           free           |
-----
| 20:00 - 21:00 |           free           |
-----
| 21:00 - 22:00 |           free           |
-----
| 22:00 - 23:00 |           free           |
-----
| 23:00 - 23:59 |           free           |
-----
You may plan to complete the following deadlines in your free time:
1. [D][N] assignment 3 (by: 14-11-2019 23:59)

```

Figure 21. Output of Free Command

4.5.1. Implementation

The `FreeCommand` class extends `Command` by providing methods to search for free time slots and the suggestion of deadlines to complete during their free time.

Features elaborated:

- The output of `FreeCommand` is ANSI colour enabled to easily differentiate free time slots and busy time slots.

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

Step 1.

The user enters `free 08-11-2019`. The `parse` method in the `CommandParser` class returns a new `FreeCommand` with `08-11-2019` as the input date to search for free time on.



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

Step 2.

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

Step 3.

In the `execute` method, the `isDateAfterCurrentDate()` and `isDateSame()` methods are called to check if the input date entered is either the current date or a date in the future. If the input date is valid, the `findFreeTime` method is then called.



`InvalidArgumentException` will be thrown if the user enters a date that has passed.

Step 4.

The `findFreeTime()` method iterates through the current list of `Tasks` from the `TaskList` class and checks for both `Event` and `Deadline` tasks. If an `Event` or `Deadline` is found, the `populateEventTimes` or `populateDeadlines` method is then called respectively.

Step 5.

The `populateEventTimes` method calls the `isEventDateWithin()` and `isDuplicateEvent()` methods to check if the `Event` date lies within the input date and if they are a duplicate `Event` respectively. If the `Event` date lies within the input date and is not a duplicate entry, its start and end time will be added to an `ArrayList` corresponding to `startTimes` and `endTimes` respectively.

Step 6.

The `populateDeadlines` method calls the `isDeadlineDueNextWeek()`, `isDuplicateDeadline()` and `isCompleted()` methods to check if the `Deadline` due date lies within one week from the input date, whether they are a duplicate `Deadline` and if they have already been completed respectively. If the `Deadline` due date lies within one week from the input date given that is not a duplicate entry and has not been completed yet, its due date will be added to an `ArrayList` corresponding to `deadlinesDue` while its name will be added to both `deadlineNames` and `sortedDeadlineNames`.

Step 7.

The `findFreeTime()` method then calls the `parseSemesterList` method, which uses the same logic in Step 4 to obtain the lesson start and end times if the lesson day coincides with the input day. The lesson start and end times are then added into an existing `ArrayList` called `startTimes` and `endTimes` respectively after checking that it is not a duplicate.

Step 8.

All `startTimes`, `endTimes` and `deadlinesDue` are sorted in ascending order by calling the `sort` method

in the `SortByTime` class. The `sortDeadlineNames()` method is then called to sort the deadline names according to their due dates.

Step 9.

The `printFreeTimeHeader` method in the `Ui` class is then called to display to the user the header of the input date.

Step 10.

The `parseSlotStates` method is then called to determine if the time slot is `free` if the `Event` does not coincide with the time slot or `BUSY` if the `Event` coincides with the time slot.

Step 11.

The `parseOutput` method is then called to print the time slots with the relevant details by:

- Calling `printFreeSlots` method in `Ui` class if the slot state is `free`.
- Calling `printBusySlots` method in `Ui` class if the slot state is `BUSY`.
- Calling `printSuggestionDetails` method in `Ui` class if 4 consecutive `free` slots are present.

```
Here are your tasks for 08-11-2019:
    1. [E][N] lecture (from: 08-11-2019 10:00 to: 08-11-2019 12:00)

Enter a command:
schedule 14-11-2019

Here are your tasks for 14-11-2019:
    1. [D][N] assignment 3 (by: 14-11-2019 23:59)
```

Figure 22. Example of tasks on the input date and deadlines due for Free Command.

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

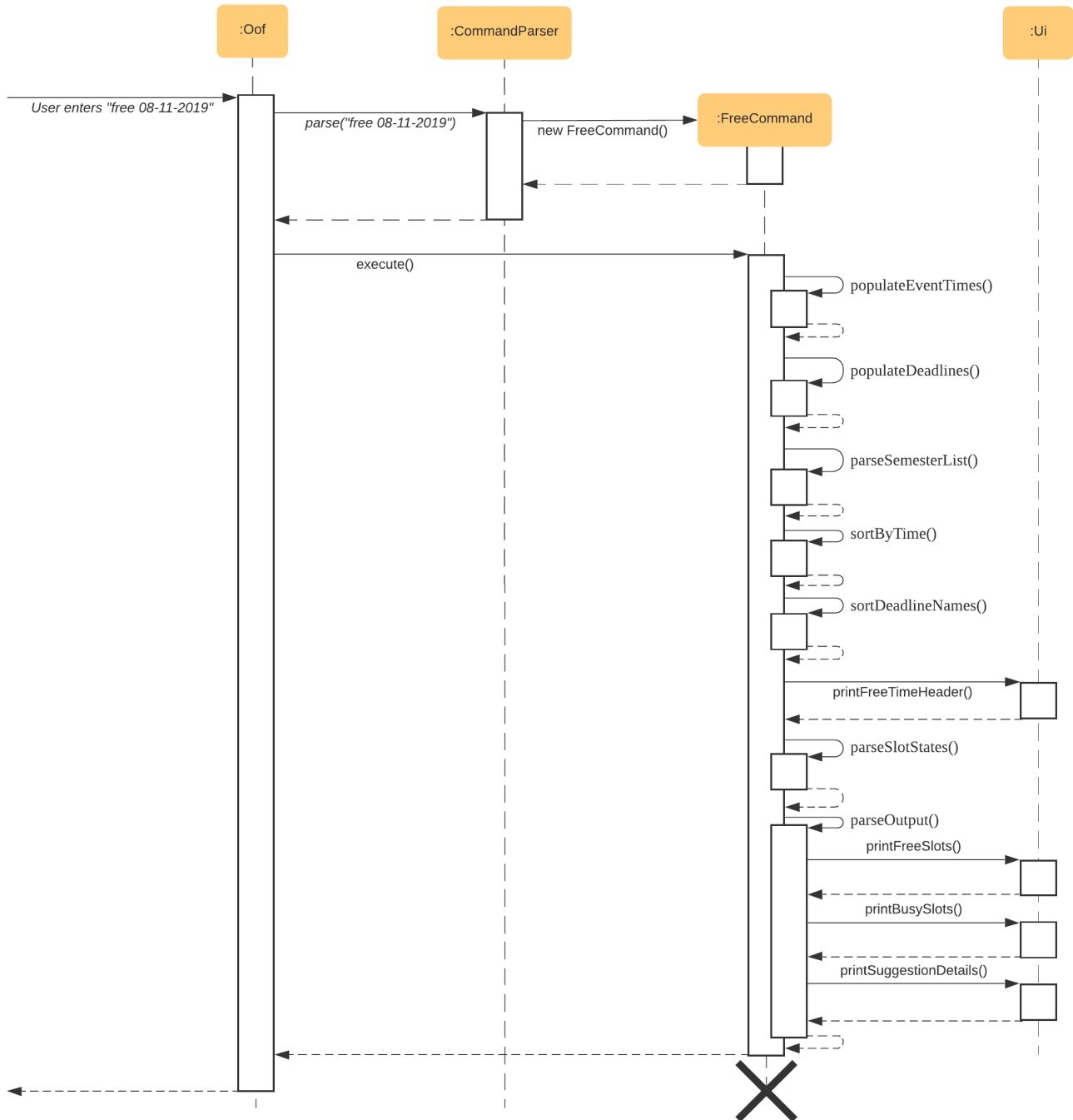


Figure 23. Sequence diagram for Free Command

4.5.2. Design Considerations

- **Selecting a single date to search free time slots in.**

- **Rationale:**

It allows the user to view which time slots they have free time in for a specific day so that they can quickly schedule team meetings.

- **Alternatives considered:**

Allow users to specify an end date in which they want to search for free time slots up to instead of just a single date. Allowing users to do so will result in displaying unwanted time slots such as during hours where users are resting which would lead to a redundant display of free time slots.

- **Displaying free time slots in hourly blocks.**

- Rationale:

This would give users a clean and easy view of the free time slots for that specific day.

- Alternatives considered:

Show free time slots in user-specified time blocks. This alternative can be an extension of the current implementation of the `FreeCommand` class.

- **Displaying suggestions for deadlines at the end of the free time slots display.**

- Rationale:

This would allow the users to view the suggestions easily without having to scroll up since the display for free time slots is very long.

- Alternatives considered:

Show suggestions directly in the 4 consecutive free time slots instead. This alternative would inhibit users in optimally viewing their free time since the free time slot will be replaced with the suggested deadline to complete. Thus, showing suggestions in the current implementation gives the user the freedom to plan what to do with their free time.

4.6. Task Tracker feature

4.6.1. Implementation

The `TrackerCommand` class extends the `Command` class by providing functions to start, stop and pause trackers as well as display a histogram visualising the amount of time spent on each `Module`. `TrackerCommand` class also provides functions for viewing and deleting tracker entries.

Also, it contains the following features:

```
Enter a command:
tracker /start 13 cs2101

Begin Task: homework
Module Code: CS2101
It is currently Sun Nov 10 13:40:59 SGT 2019
Current total time spent on homework: 0 minutes
```

Figure 24. Output of Tracker Start Command

- You may `tracker /start` by `taskListIndex` with a specific command.

```
Enter a command:
tracker /stop 13 cs2101

Ending Task: homework
Module Code: CS2101
It is currently Sun Nov 10 13:43:50 SGT 2019
Total time spent on homework: 3 minutes
```

Figure 25. Output of Tracker Stop Command

- You may `tracker /stop` by `taskListIndex` with a specific command.

```
Enter a command:
tracker /pause 13 cs2101

Pausing Task: homework
Module Code: CS2101
It is currently Sun Nov 10 13:43:01 SGT 2019
Total time spent on homework: 3 minutes
```

Figure 26. Output of Tracker Pause Command

- You may `tracker /pause` by `taskListIndex` with a specific command.

```
Enter a command:
tracker /view week

|
|      st2334 -- 2 minutes
|
| #      cs2106 -- 10 minutes
|
| #      cs2101 -- 10 minutes
|
| #      cs2101 -- 10 minutes
|
| #      cs2101 -- 10 minutes
|
| ##### cs2105 -- 40 minutes
|

Total Time: 82 minutes
```

Figure 27. Output of Tracker View Command

- You may `tracker /view` by `day` with a specific command.
- You may `tracker /view` by `week` with a specific command.
- You may `tracker /view` by `all` with a specific command.

```
Enter a command:
tracker /delete 6

Deleting tracker: homework -- 3 minutes
Now you have 5 trackers in your list.
```

Figure 28. Output of Tracker Delete Command

- You may `tracker /delete` by `taskListIndex` with a specific command.

Enter a command:

```
tracker /list
```

```
1. CS2106 lab -- 10 minutes
2. CS2101 PPP -- 4 minutes
3. lecture -- 40 minutes
4. homework -- 2 minutes
5. homework -- 3 minutes
6. homework -- 1 minutes
```

Figure 29. Output of Tracker List Command

- You may `tracker /list` with a specific command.

Provided below is an example usage scenario and how `TrackerCommand` class behaves and interacts with other relevant classes.

Step 1:

The user enters `tracker` command. The `execute` method of `TrackerCommand` class reads and saves all `Tracker` objects stored in persistent storage, `tracker.csv` through the `readTrackerList` method in the `StorageManager` class.

- **Step 1a:**

The `readTrackerList` method in `StorageManager` class retrieves and processes `tracker.csv` from persistent storage through the `loadFile` method in `Storage` class.

- **Step 1b:**

The `loadFile` method calls the `dataToTrackerList` method in `StorageParser` class, which in turn calls the `processLine` method.

- **Step 1c:**

The `processLine` method of `StorageParser` class splits each line into its respective fields through the use of `,` delimiters before parsing and assigning them to the correct fields. A new `Tracker` object will be created with the processed data and returned to the `dataToTrackerList` method.

- **Step 1d:**

The `Tracker` object returned to `readTrackerList` will be added into the `ArrayList` `Tracker` objects and upon completing the entire `tracker.csv` file, the `ArrayList` will be returned to the `execute` method of `TrackerCommand`. The `execute` method of `TrackerCommand` class will then detect what instructions the user has indicated.



`StorageFileCorruptedException` will be thrown if `tracker.csv` cannot be processed and a new `ArrayList` of `Tracker` objects will be created.

Step 2:

If the user given instruction is `/view`, the `execute` method of `TrackerCommand` will get the `period` indicated by the user. The `execute` method of `TrackerCommand` calls the `processModuleTrackerList` method.



`InvalidArgumentException` will be thrown if the instruction given by the user is invalid. `InvalidArgumentException` will be thrown if the `tracker` command is incomplete.

- **Step 2a:**

The `processModuleTrackerList` method creates a new `ArrayList` of `Tracker` objects and processes the user input to determine if it is to be filtered by `day`, `week` or `all` `Tracker` entries.

- **Step 2b:**

If the user indicated to filter `/view` by `day`, a new `Date` instance is created and parsed into the format of `dd-MM-yyyy` before the `timeSpentByModule` method is called. If the user indicated to filter `/view` by `week`, a `Date` instance containing the exact date seven days ago is created and parsed into the format of `dd-MM-yyyy` before the `timeSpentByModule` method is called. If the user indicated to filter `/view` by `all`, the `timeSpentByModule` method is called upon immediately.



`InvalidArgumentException` will be thrown if the `period` cannot be processed. `TrackerNotFoundException` will be thrown if the `ArrayList` of `Tracker` objects is empty.

- **Step 2c:**

The `processModuleTrackerList` method calls `sortAscending` method. This is where the new `ArrayList` of `Tracker` objects are sorted by their `timeTaken` property with the use of a `Comparator`.

- **Step 2d:**

The `execute` method of `TrackerCommand` calls `printTrackerDiagram` from the `Ui` class.

Step 3:

If the user given instruction is `/list`, the `execute` method of `TrackerCommand` calls the `printTrackerList` method in the `Ui` class. Else, the next input field will be retrieved.



`InvalidArgumentException` will be thrown if the instruction given by the user is invalid.

Step 4:

If the user given instruction is `/delete`, the user input will be used as `taskIndex` to identify the tracker from the `ArrayList` of `Tracker` objects. It will then be removed from the `ArrayList` before the `execute` method of `TrackerCommand` calls the `printTrackerDelete` method in the `Ui` class and updates `tracker.csv` by calling `writeTrackerList` method from `StorageManager` class.



`InvalidArgumentException` will be thrown if the instruction given by the user is invalid. `InvalidArgumentException` will be thrown if the `taskIndex` is invalid.

Step 5:

If the instruction is not `/view`, `/list` or `/delete`, the `execute` method of `TrackerCommand` will obtain the `TASK_INDEX` and `MODULE_CODE` given by the user and check if the relevant `Task` has been completed.



`InvalidArgumentException` will be thrown if the instruction given by the user is invalid. `TaskAlreadyCompletedException` will be thrown if the `Task` has already been completed.

- **Step 5a:**

The `execute` method of `TrackerCommand` calls `isValidDescription` method to check if the `description` of the `Task` matches the `description` of the `Tracker` of the same `TaskList` index.

Step 6:

If the user given instruction is `/start`, a new `Tracker` will be added into the `ArrayList` of `Tracker` objects if the `Task` has never been tracked in the past. If the `Task` has been previously tracked, the `updatedTrackerList` method is called.



`InvalidArgumentException` will be thrown if the instruction given by the user is invalid. `InvalidArgumentException` will be thrown if the saved `Tracker` object `description` does not match the `Task` object `description` where they are of the same `TaskList` index.

- **Step 6a:**

The `updatedTrackerList` searches for the matching `Tracker` object in the `ArrayList` of `Tracker` objects before updating the `lastUpdated` and `startDate` with the current `Date`.

- **Step 6b:**

The `execute` method of `TrackerCommand` calls the `writeTrackerList` method of `StorageManager` class to update `tracker.csv`.

- **Step 6c:**

The `execute` method of `TrackerCommand` calls `printStartAtCurrent` method of `Ui` class.

Step 7:

If the user given instruction is `/stop`, the `execute` method of `TrackerCommand` class calls `updateTimeTaken` method.



`InvalidArgumentException` will be thrown if the instruction given by the user is invalid. `TrackerNotFoundException` will be thrown if the `Tracker` has no `startDate` or cannot be found. `InvalidArgumentException` will be thrown if the saved `Tracker` object `description` does not match the `Task` object `description` where they are of the same `TaskList` index.

- **Step 7a:**

This is where the `timeTaken` of the matching `Tracker` object will be updated, calculating the time difference between `startDate` and current `Date`.

- **Step 7b:**

The `execute` method of `TrackerCommand` calls the `setStatus` method in the `Task` class to mark the `Task` as completed.

- **Step 7c:**

The `execute` method of `TrackerCommand` calls `writeTrackerList` and `writeTaskList` methods in `StorageManager` class to update the `ArrayList` of `Tracker` objects and `TaskList`.

- **Step 7d:**

The `execute` method of `TrackerCommand` calls `printEndAtCurrent` method of `Ui` class.

Step 8:

If the user given instruction is `/stop`, the `execute` method of `TrackerCommand` class calls `updateTimeTaken` method.



`InvalidArgumentException` will be thrown if the instruction given by the user is invalid. `TrackerNotFoundException` will be thrown if the `Tracker` has no `startDate` or cannot be found. `InvalidArgumentException` will be thrown if the saved `Tracker` object `description` does not match the `Task` object `description` where they are of the same `TaskList` index.

- **Step 8a:**

This is where the `timeTaken` of the matching `Tracker` object will be updated, calculating the time difference between `startDate` and current `Date`.

- **Step 8b:**

The `execute` method of `TrackerCommand` calls `writeTrackerList` and `writeTaskList` methods in `StorageManager` class to update the `ArrayList` of `Tracker` objects and `TaskList`.

- **Step 8c:**

The `execute` method of `TrackerCommand` calls `printEndAtCurrent` method of `Ui` class.

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

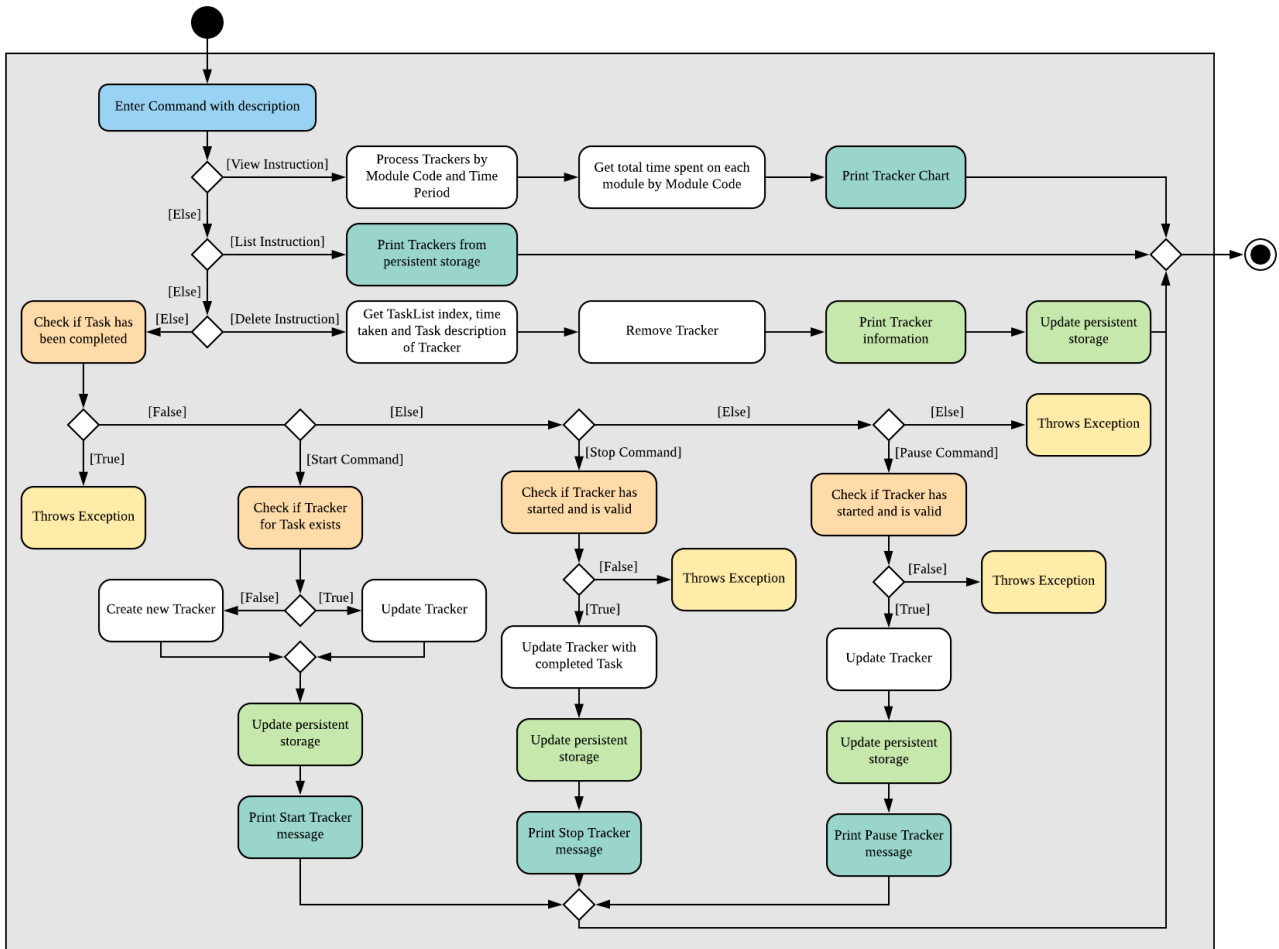


Figure 30. Activity Diagram for TrackerCommand

4.6.2. Design Considerations

- **Creating `tracker.csv` to store past entries and their associated information**

- **Rationale:**

With scalability in mind, the use of persistent storage will grant our users access to previous `Tracker` entries that they have made and allow our tracker diagram to be generated over a more extensive range of entries made before the current run of `OOF`. The use of `.csv` format for persistent storage and delimiting each respective field by `,`. As some data fields can contain multiple whitespaces and tabs, the use of a whitespace delimiter may affect the processing algorithm negatively. The use of a comma is also less likely in module codes, task descriptions, and dates.

- **Alternatives Considered:**

The use of `.txt` and delimited by `\t` has been considered. However, the use of a tab may interfere with the processing algorithm should the user input contains four consecutive whitespaces — which is processed as an equivalent to `\t`.

- **Splitting the `timeTaken` property in `ArrayList` of `Tracker` objects sorted by `moduleCode` into blocks of ten minutes in the histogram**

- **Rationale:**

As more tasks get completed over time, the `timeTaken` property in `Tracker` objects will increase exponentially. With the estimated ten work hours weekly on each module, this may

result in hundreds of minutes spent on `Tasks` for each `moduleCode`. By splitting the `timeTaken` property in the `ArrayList` of `Tracker` object sorted by `moduleCode` into blocks of ten minutes, the number of `#` printed will reduce drastically and allow a more compact diagram to be printed without compromising its accuracy beyond tolerance.

- Alternatives Considered:

Without the splitting of the `timeTaken` property of `Tracker` objects in the `ArrayList`, an additional variable `segmentedTimeTaken` will not be required and the user will be able to see a more accurate histogram as it will be printing one `#` to represent one minute instead.

5. Documentation

5.1. Introduction

We use AsciiDoc for writing documentation.



We chose AsciiDoc over Markdown because AsciiDoc, although a bit more complex than Markdown, provides more flexibility in formatting.

5.2. Editing Documentation

- `AsciiDoc`

Converts AsciiDoc files in `docs` to HTML format. Generated HTML files can be found in `build/docs`.

- `deployOfflineDocs`

Updates the offline user guide, and its associated files, used by the Help window in the application. Deployed HTML files and images can be found in `src/main/resources/docs`.



You can also choose to download IntelliJ's `.adoc` plugin to edit and render `.adoc` files locally.

5.3. Editing diagrams

We use `LucidChart` to create and edit our UML diagrams in the developer guide.

5.4. Publishing Documentation

5.4.1. Setting up Travis CI

1. Fork the repo to your own organization.
2. Go to <https://travis-ci.org/> and click `Sign in with GitHub`, then enter your GitHub account details if needed.

Sign in with GitHub 

onfidence

Figure 31. Button for signing into Github

- Head to the [Accounts](#) page, and find the switch for the forked repository.
 - If the organization is not shown, click [Review and add](#) as shown below:

Is an organization missing?
[Review and add your authorized organizations.](#)

Figure 32. Reviewing and adding an organization

This should bring you to a GitHub page that manages the access of third-party applications. Depending on whether you are the owner of the repository, you can either grant access

4j 

Grant access

Figure 33. Granting access

or request access

AY1617S1 

Request access

Figure 34. Requesting access

to Travis CI so that it can access your commits and build your code.

- If repository cannot be found, click [Sync account](#)
- Activate the switch.



Figure 35. Syncing account

- This repo comes with a link that tells Travis what to do. So there is no need for you to create one yourself.
- To see the CI in action, push a commit to the master branch!
 - Go to the repository and see the pushed commit. There should be an icon that will link you

to the Travis build.

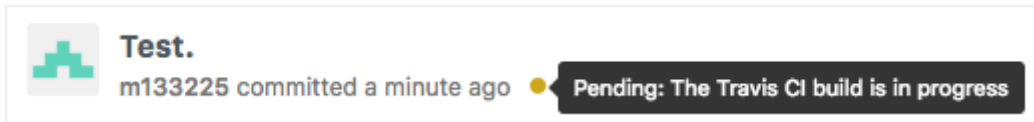


Figure 36. Travis build progress

- As the build is run on a provided remote machine, we can only examine the logs it produces:

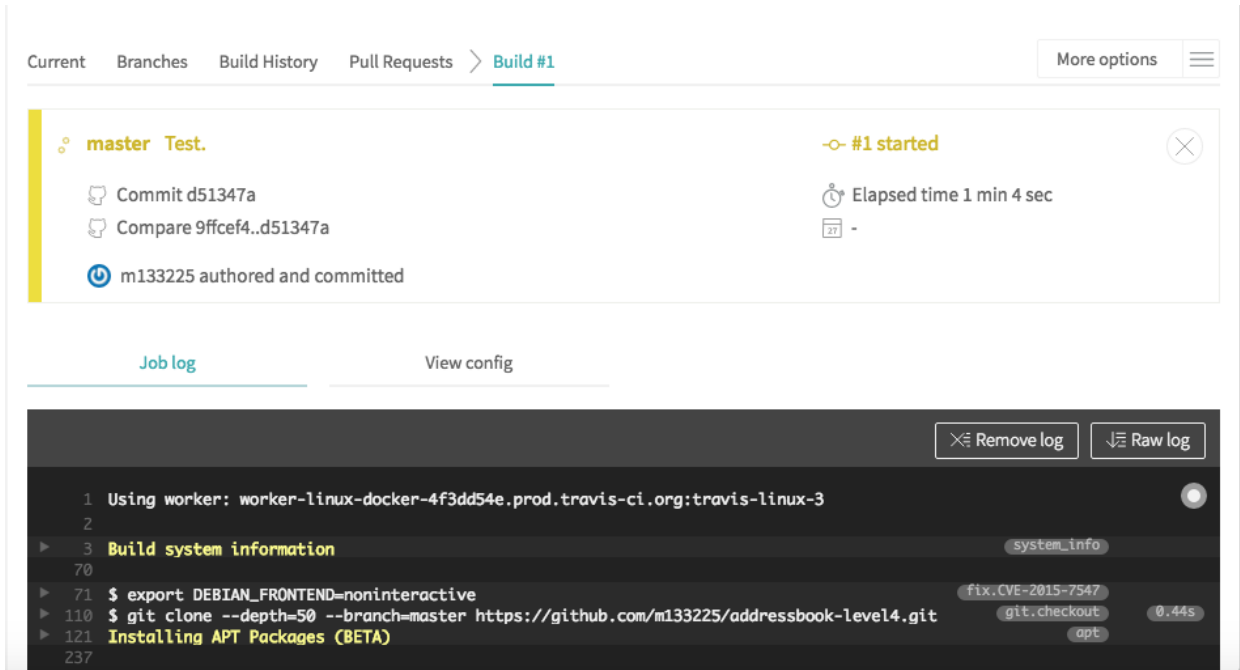


Figure 37. Checking Travis logs

7. If the build is successful, you should be able to check the coverage details of the tests at [Coveralls](#)
8. Update the link to the 'build status' badge at the top of the `README.adoc` to point to the build status of your own repo.

5.4.2. Enabling auto-publishing of documentation

1. Ensure that you have followed the steps above to set up Travis CI.
2. On GitHub, create a new user account and give this account collaborator and admin access to the repo.
Using this account, generate a personal access token [here](#).



Personal access tokens are like passwords so make sure you keep them secret! If the personal access token is leaked, please delete it and generate a new one.



If you are the only one with write access to the repo, you can use your own account to generate the token.

- Add a description for the token. (e.g. `Travis CI - deploy docs to gh-pages`)
- Check the `public_repo` checkbox.

- Click **Generate Token** and copy your new personal access token.

You will use this token to grant Travis access to the repo.

Token description

Travis CI - deploy docs to gh-pages

What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input type="checkbox"/>	repo	Full control of private repositories
<input type="checkbox"/>	repo:status	Access commit status
<input type="checkbox"/>	repo_deployment	Access deployment status
<input checked="" type="checkbox"/>	public_repo	Access public repositories
<input type="checkbox"/>	admin:org	Full control of orgs and teams

Figure 38. Generating a token

- Head to the [Accounts](#) page, and find the switch for the forked repository.



Figure 39. Syncing the repository

- Click on the settings button next to the switch. In the Environment Variables section, add a new environment variable with
 - name: **GITHUB_TOKEN**
 - value: personal access token copied in step 1
 - Display value in build log: **OFF**

Environment Variables

Notice that the values are not escaped when your builds are executed. Special characters (for bash) should be escaped accordingly.

Name	Value	Display value in build log	
GITHUB_TOKEN	OFF	<div> <div></div> <div>Add</div> </div>

Figure 40. Adding a token



Make sure you set `Display value in build log` to `OFF`.

Otherwise, other people will be able to see the personal access token and thus have access to this repo.

Similarly, make sure you **do not print `$GITHUB_TOKEN` to the logs** in Travis scripts as the logs are viewable by the public.

5. Now, whenever there's a new commit to the master branch, Travis will push the latest documentation to gh-pages branch.

To verify that it works,

1. Trigger Travis to regenerate documentation. To do so, you need to push a new commit to the master branch of the fork.
Suggested change: Remove the Codacy badge from `README`.
2. Wait for Travis CI to finish running the build on your new commit.
3. You should see your `README` file displayed on your team repository.

5.4.3. Converting Documentation to PDF format

Follow the instructions for AsciiDoc conversion on this [page](#) to set up `asciidoctor-pdf` for converting `.adoc` files to PDF.

6. Testing

Testing is vital to ensure that the code you will be contributing in the future does not cause existing features to fail. There are 2 ways to run tests.

Method 1: Using IntelliJ JUnit test runner

- To run all tests, right-click on the `src/test/java` folder and choose `Run 'All Tests'`
- To run a subset of tests, you can right-click on a test package, test class, or a test and choose `Run 'ABC'`

Method 2: Using Gradle

- Open a console and run the command `gradlew clean allTests` (Mac/Linux: `./gradlew clean allTests`)

6.1. Troubleshooting Testing

Problem: `HelpWindowTest` fails with a `NullPointerException`.

- Reason: One of its dependencies, `HelpWindow.html` in `src/main/resources/docs` is missing.
- Solution: Execute Gradle task `processResources`.

Problem: Keyboard and mouse movements are not simulated on macOS Mojave, resulting in GUI Tests failure.

- Reason: From macOS Mojave onwards, applications without **Accessibility** permission cannot simulate certain keyboard and mouse movements.
- Solution: Open **System Preferences**, click **Security and Privacy** → **Privacy** → **Accessibility** and check the box beside **IntelliJ IDEA**.

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](#) to perform *Continuous Integration* on our projects. See [UsingTravis.adoc](#) for more details.

7.3. Coverage Reporting

We use [Coveralls](#) to track the code coverage of our projects. See [UsingCoveralls.adoc](#) for more details.

7.4. Documentation Previews

If a pull request contains changes to AsciiDoc files, you can use [Netlify](#) to see a preview of how the HTML version of those AsciiDoc files will look like when the pull request is merged. See [UsingNetlify.adoc](#) for more details.

7.5. Making a Release

Here are the steps to create a new release.

1. Update the version number in **build.gradle**.
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: Product Scope

Target User Profile:

- University students
- Prefer desktop Command-Line-Interface (CLI) over other types
- Able to type on the keyboard fast

- Prefers typing over mouse input
- Proficient in using CLI applications

Value proposition:

- Helps you plan your tasks, modules and lessons more effectively
- Helps you coordinate common free time slots with other people
- Automatically reminds you of upcoming deadlines
- Automatically organizing your tasks for viewing in calendar, tabular and list format
- Allows you to plan your semester in advance
- Works offline

Appendix B: Requirements

B.1. User Stories

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

Table 1. Table consolidating the user stories

S/N	Use Case No	Priority Level	As a ...	I can ...	So that I ...
01	01	* * *	University Student	Add a task	Won't forget the tasks I have to complete
02	02	* * *	University Student	Mark a task as complete	Can keep track of what is left to be completed
03	03	* * *	University Student	View my tasks in a calendar	Can manage my time properly
04	04	* *	University Student	View a summary of tomorrow's task	Will know what to expect for the next day
05	05	* * *	University Student	Add an event with the relevant dates, start and end times	Can keep track of my upcoming appointments and examinations
06	06	* * *	University Student	Get reminders of deadlines due within 24 hours	Can prioritize those tasks to be completed first
07	07	* * *	University Student	Sort my tasks	Can see my tasks in chronological order
08	08	*	University Student	Find my tasks	Do not need to scroll through the entire calendar to find certain tasks

09	09	**	Double degree University student	Color code the tasks	Can quickly distinguish different type of tasks
10	10	**	University Student	View my tasks for the week	Can plan my time for the week
11	11	***	Busy University Student	Find free time slots	Will know which dates and times I am free to conduct project meetings
12	12	***	University Student	Cancel events	Keep my schedule updated
13	13	***	University Student	Postpone the deadline of tasks	Can properly manage my priorities
14	14	**	University Student who procrastinate s	View undone tasks carried forward to the next day in a bright color	Will know what assignments are lagging behind
15	15	***	University Student	Add a recurring task	Do not have to do it multiple times
16		***	Impatient University Student	Quickly type in one-liner commands	Can see the tasks being updated in the program quickly
17		*	University Student	View trends for my tasks	Can see if I am lagging behind
18		**	Paranoid University Student	Set the threshold for an alert to complete my tasks	Can stay ahead of my schedule
19		*	Organized University Student	View all the tasks in a strict format	Will know what to type to enter my tasks
20		*	University Student in NUSSU	Export my calendar to a shareable format	Can quickly share my schedule with other people
21		**	University Student	Have a do-after task	Know what tasks need to be done after completing a specific task
22		***	University Student	Have a task that needs to be done within a period	Can better plan my schedule
23		*	University Student	Add my estimated time taken to complete a task	Know how much free time I would have

24		**	Undergraduate Tutor	Have two instances of calendar	Can separate my tutor tasks and personal tasks
25		**	University Student	Filter my calendar by different categories	Can view my tasks for that category easier
26		***	University Student	Add a tentative task	Can confirm it at a later date
27		***	University Student	View all commands	Do not need to memorise all the commands
28		***	University Student	Get warnings if an event I add clashes with an existing event	Will not have multiple events at the same time
29		*	University Student	Sync my tasks to my phone via bluetooth	Can view my tasks on the go and not just on my laptop
30		**	University Student	Print out my tasks stored	Can view my tasks even if my laptop runs out of battery

B.2. Use Cases

(MSS refers to Main Success Scenario.)

System: Outstanding Organization Friend (OOF)

Use case: UC01 - Add a task

Actor: User

MSS:

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

Use case ends.

Extensions:

- 3a. OOF detects an empty date and time in the description of the task.
 - 3a1. OOF requests for date and time of task.
 - 3a2. User enters the required data.
 - Steps 3a1-3a2 are repeated until the correct data is entered.
 - Use case resumes from step 4.
- 3b. OOF detects a clash in date and time with another task.
 - 3b1. OOF warns the User of such a clash by displaying the task(s) that clash(es) and prompts

for continuation or cancellation.

- 3b2. User decides for continuation or cancellation.
- 3b3. OOF requests to confirm the decision.
- 3b4. User confirms the decision.
- Use case ends if the User decides to cancel the action. Use case resumes from step 4 otherwise.
- *a. At any time, User chooses to re-enter the task description.
 - *a1. OOF requests confirmation to re-enter the task description.
 - *a2. User confirms to re-enter the task description.
 - Use case resumes from step 3.

System: Outstanding Organization Friend (OOF)

Use case: UC02 - Mark a task as complete

Actor: User

MSS:

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

Use case ends.

Extensions:

- 3a. OOF detects a non-existent task index.
 - 3a1. OOF requests for existent index and displays a range of indexes to choose from.
 - 3a2. User enters the required data.
 - Use case resumes from step 4.

System: Outstanding Organization Friend (OOF)

Use case: UC03 - View tasks in calendar

Actor: User

MSS:

1. User wants to view tasks in calendar format.
2. OOF requests for the month and year the user wishes to view in calendar format.
3. User enters a month and year.
4. OOF displays the tasks requested in calendar format.

Use case ends.

Extensions:

- 3a. OOF detects an invalid date.
 - 3a1. OOF requests for a valid month and year.
 - 3a2. User enters the required data.
 - Use case resumes from step 4.

System: Outstanding Organization Friend (OOF)

Use case: UC04 - View a summary of the next day's tasks

Actor: User

MSS:

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

Use case ends.

Extension:

- 3a. OOF detects there are no tasks for the next day.
 - 3a1. OOF prints to the console to warn the User that there are no tasks for the next day.
 - Use case ends.

System: Outstanding Organization Friend (OOF)

Use case: UC05 - Adding tasks with date and time

Actor: User

MSS:

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

Use case ends.

Extension:

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

- *a1. OOF requests to confirm the cancellation.
- *a2. User confirms the cancellation.
- Use case ends.

System: Outstanding Organization Friend (OOF)

Use case: UC06 - Reminder for expiring tasks (within 24hrs)

Actor: User

MSS:

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

Use case ends.

Extensions:

- *a. At any time, User chooses to cancel the activation.
 - *a1. OOF requests to confirm the cancellation.
 - *a2. User confirms the cancellation.
 - Use case ends.

System: Outstanding Organization Friend (OOF)

Use case: UC07 - Sort tasks in chronological order

Actor: User

MSS:

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

Use case ends.

Extensions:

- 4a. OOF detects that there are no tasks to be sorted.
 - 4a1. OOF warns User that there are no tasks to be sorted
 - Use case ends.
- *a. At any time, User chooses to cancel the request.
 - *a1. OOF requests to confirm the cancellation.
 - *a2. User confirms the cancellation.
 - Use case ends.

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

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

Use case ends.

Extensions:

- 3a. OOF detects that there are no tasks that match the description given.
 - 3a1. OOF requests for the User to enter a new description.
 - 3a2. User enters a new description.
 - Steps 3a1-3a2 are repeated until at least one task matches the description.
 - Use case resumes from step 4.
- *a. At any time, User chooses the stop finding tasks.
 - *a1. OOF requests to confirm the request.
 - *a2. User confirms the requests.
 - Use case ends.

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

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

Use case ends.

Extensions:

- 3a. OOF detects an error in the information entered.
 - 3a1. OOF prompts for User to enter the correct information.
 - 3a2. User enters the correct information.
 - Steps 3a1-3a2 are repeated until the User enters in the correct information.

- Use case resumes from step 4.
- 4a. OOF detects that there are no tasks to be colour coded.
 - 4a1. OOF displays the warning that no tasks are available to be colour coded.
 - Use case ends.
- *a. At any time, User requests to cancel this action.
 - *a1. OOF requests to confirm the cancellation.
 - *a2. User confirms the cancellation.
 - Use case ends.

System: Outstanding Organization Friend (OOF)

Use case: UC10 - View tasks for the week

Actor: User

MSS:

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

Use case ends.

Extensions:

- 4a. OOF detects that there are no tasks for the week.
 - 4a1. OOF warns the User that there are no tasks for the week.
 - Use case ends.
- *a. At any time, User chooses to cancel this action.
 - *a1. OOF requests for confirmation.
 - *a2. User confirms the requests.
 - Use case ends.

System: Outstanding Organization Friend (OOF)

Use case: UC11 - Find free time slots

Actor: User

MSS:

1. User requests to find free time slots.
2. OOF requests for a date from the User.
3. User enters in the date of interest.
4. OOF displays the free time slots for that particular day.

Use case ends.

Extensions:

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

System: Outstanding Organization Friend (OOF)**Use case: UC12 - Delete tasks****Actor: User****MSS:**

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

Use case ends.

Extensions:

- 2a. OOF detects that there are no tasks saved in the program.
 - 2a1. OOF warns the User that there are no tasks to be deleted.
 - Use case ends.
- 3a. OOF detects an error in the task that was selected by the User.
 - 3a1. OOF prompts the user to enter a valid input.
 - 3a2. User enters a valid input.
 - Steps 3a1-3a2 are repeated until the User enters a valid input.
 - Use case resumes from step 4.
- *a. At any time, User chooses to cancel the action.
 - *a1. OOF requests for confirmation from the User.
 - *a2. User confirms the cancellation.
 - Use case ends.

System: Outstanding Organization Friend (OOF)

Use case: UC13 - Postpone tasks

Actor: User

MSS:

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

Use case ends.

Extensions:

- 2a. OOF detects that there are no tasks saved in the program.
 - 2a1. OOF warns the User that there are no tasks to be postponed.
 - Use case ends.
- 3a. OOF detects an error in the task that was selected by the User.
 - 3a1. OOF prompts the user to enter a valid input.
 - 3a2. User enters a valid input.
 - Steps 3a1-3a2 are repeated until the User enters a valid input.
 - Use case resumes from step 4.
- *a. At any time, User chooses to cancel the action.
 - *a1. OOF requests for confirmation from the User.
 - *a2. User confirms the cancellation.
 - Use case ends.

System: Outstanding Organization Friend (OOF)

Use case: UC14 - Overdue tasks

Actor: User

MSS:

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

Use case ends.

Extensions:

- 3a. OOF detects that there are no overdue tasks.

- 3a1. OOF warns the User that there are no overdue tasks.
- Use case ends.
- *a. At any time, User chooses to cancel the activation.
 - *a1. OOF requests to confirm the cancellation.
 - *a2. User confirms the cancellation.
 - Use case ends.

System: Outstanding Organization Friend (OOF)

Use case: UC15 - Recurring tasks

Actor: User

MSS:

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

Use case ends.

Extensions:

- 2a. OOF detects that there are no tasks saved in the program.
 - 2a1. OOF warns the User that there are no tasks to be marked as recurring.
 - Use case ends.
- 3a. OOF detects an error in the task that was selected by the User.
 - 3a1. OOF prompts the user to enter a valid input.
 - 3a2. User enters a valid input.
 - Steps 3a1-3a2 are repeated until the User enters a valid input.
 - Use case resumes from step 4.
- *a. At any time, User chooses to cancel the action.
 - *a1. OOF requests for confirmation from the User.
 - *a2. User confirms the cancellation.
 - Use case ends.

B.3. Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed
2. Should be able to hold up to 200 tasks/events without performance deterioration

3. A user with above-average typing speed for regular English Text should be able to store their tasks faster using commands than using the mouse

Appendix C: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Appendix D: Instructions for Manual Testing



The instructions and sample test cases only act as a guide for you to start testing on some of our application features. You are free to test our features with more test cases of your own. Refer to [Section 2.1, “Prerequisites”](#) for the instructions to set up our program on your computer.

D.1. Managing Semesters

D.1.1. Adding a semester

1. Prerequisites: List all semesters using the `semester /view` command.
2. Test case: `semester /add 19/20 /name Semester 2 /from 05-01-2020 /to 05-05-2020`
Expected: Using the `semester /view` command will display Academic Year 19/20, Semester 2 (05-01-2020 to 05-12-2020) at the latest index.
3. Test case: `semester /add 19/20 /name Semester 2 /from 05-01-2020 /to 05-05-2020` after running the same command above
Expected: An error would be displayed regarding the adding of a semester that clashes with an existing semester. `semester /view` will not show a second copy of the semester added above.
4. Test case: `semester /add 19/20 /name Semester 2 /from a /to 05-05-2020`
Expected: An error would be displayed regarding an invalid command argument.
5. Other incorrect commands to try: `semester /add /name Semester 2 /from 05-01-2020 /to 05-05-2020`, `semester /add 19/20 /name Semester 2 /from 32-01-2020 /to 05-05-2020`
Expected: An error would be displayed regarding a missing and an invalid command argument respectively.

D.1.2. Deleting a semester

1. Prerequisites: List all semesters using the `semester /view` command. At least 1 semester is on the list.



In the provided `.jar` file, the semester at index 1 has been rigorously populated with test data for testing purposes. You are recommended to add your own semester before testing the `semester /delete` command to avoid the deletion of the populated semester.

1. Test case: `semester /delete 2`
Expected: The 2nd semester is deleted from the list. Details of the deleted semester will be echoed as an acknowledgement message.
2. Test case: `semester /delete 0`
Expected: Error displayed regarding an invalid index.
3. Other incorrect delete commands to try: `semester /delete a`, `semester /delete -1`
Expected: Similar to previous.

D.1.3. Selecting a semester

1. Prerequisites: List all semesters using the `semester /view` command. At least 1 semester is on the list.
2. Test case: `semester /select 1`
Expected: An acknowledgement message displayed regarding the first semester being selected.
3. Test case: `semester /select 0`
Expected: Error displayed regarding an invalid index.
4. Other incorrect select commands to try: `semester /select a`, `semester /select -1`
Expected: Similar to previous.

D.2. Managing Modules

D.2.1. Adding a module

1. Prerequisites: Semester has been selected using the `semester /select` command.
2. Test case: `module /add CS1010 /name Programming Methodology`
Expected: Acknowledgement message displayed regarding the successful addition of `CS1010 Programming Methodology`. Using `module /view` will show `CS1010 Programming Methodology` on the list of `modules`.
3. Test case: `module /add /name Programming Methodology`
Expected: Error displayed about requiring a module code argument.
4. Test case: `module /add CS1010 /name`
Expected: Error displayed about requiring a module name argument.

D.2.2. Deleting a module

1. Prerequisites: Semester has been selected using the `semester /select` command. List all modules using the `module /view` command. At least 1 module is on the list.
2. Test case: `module /delete 2`
Expected: The 2nd module is deleted from the list. Details of the deleted module will be echoed as an acknowledgement message.
3. Test case: `module /delete 0`
Expected: Error displayed regarding an invalid index.
4. Other incorrect delete commands to try: `module /delete a`, `module /delete -1`

Expected: Similar to previous.

D.2.3. Selecting a module

1. Prerequisites: Semester has been selected using the `semester /select` command. List all modules using the `module /view` command. At least 1 module is on the list.
2. Test case: `module /select 1`
Expected: An acknowledgement message displayed regarding the first module being selected.
3. Test case: `module /select 0`
Expected: Error displayed regarding an invalid index.
4. Other incorrect select commands to try: `module /select a`, `module /select -1`
Expected: Similar to previous.

D.3. Managing Lessons

D.3.1. Adding a lesson

1. Prerequisites: Module has been selected using the `module /select` command.
2. Test case: `lesson /add Tutorial /day FRIDAY /from 10:00 /to 12:00` Expected: Acknowledgement message displayed regarding the successful addition of `Tutorial` for the selected `module`.
3. Test case: `lesson /add /day FRIDAY /from 10:00 /to 12:00` Expected: Error displayed about missing argument.
4. Test case: `lesson /add Tutorial /day FRIDAY /from a /to 12:00` Expected: Error displayed about invalid argument.
5. Other incorrect commands to try: `lesson /add Tutorial /day APPLE /from 10:00 /to 12:00`, `lesson /add Tutorial /day TUESDAY /from 10:00 /to 10:00` Expected: Similar to previous.

D.3.2. Deleting a lesson

1. Prerequisites: Module has been selected using the `module /select` command. List all lessons using the `lesson` command. At least 1 lesson is on the list.
2. Test case: `lesson /delete 1`
Expected: The 1st lesson is deleted from the list. Details of the deleted lesson will be echoed as an acknowledgement message.
3. Test case: `lesson /delete 0`
Expected: Error displayed regarding an invalid index.
4. Other incorrect delete commands to try: `lesson /delete a`, `lesson /delete -1`
Expected: Similar to previous.

D.4. Managing Tasks

D.4.1. Adding a **todo** task

1. Test case: `todo create a todo list /on 15-11-2019`
Expected: An acknowledgement message displayed regarding the successful addition of a **todo** task.
2. Test case: `todo /on 15-11-2019`
Expected: Error displayed regarding a missing argument.
3. Test case: `todo create a todo list /on`
Expected: Similar to previous.
4. Test case: `todo create a todo list /on a`
Expected: Error displayed regarding an invalid argument.

D.4.2. Adding a **deadline** task

1. Test case: `deadline volunteering sign up /by 15-11-2019 23:59`
Expected: An acknowledgement message displayed regarding the successful addition of a **deadline** task.
2. Test case: `deadline /by 15-11-2019 23:59`
Expected: Error displayed regarding a missing argument.
3. Test case: `deadline volunteering sign up /by`
Expected: Similar to previous.
4. Test case: `deadline volunteering sign up /by a`
Expected: Error displayed regarding an invalid argument.

D.4.3. Adding an **assignment** task

1. Prerequisites: Module has been selected using the `module /select` command.
2. Test case: `assignment Lab /by 15-11-2019 23:59`
Expected: An acknowledgement message displayed regarding the successful addition of a **assignment** task for the selected **module**.
3. Test case: `assignment /by 15-11-2019 23:59`
Expected: Error displayed regarding a missing argument.
4. Test case: `assignment Lab /by`
Expected: Similar to previous.
5. Test case: `assignment Lab /by a`
Expected: Error displayed regarding an invalid argument.

D.4.4. Adding an **event** task

1. Test case: `event date with girlfriend /from 15-11-2019 18:00 /to 15-11-2019 23:00`
Expected: An acknowledgement message displayed regarding the successful addition of an **event** task.
2. Test case: `event /from 15-11-2019 18:00 /to 15-11-2019 23:00`
Expected: Error displayed regarding a missing argument.

3. Test case: `event date with girlfriend /from 15-11-2019 18:00 /to 15-11-2019 15:00`
Expected: Error displayed regarding starting date and time being after ending date and time.
4. Test case: `event date with girlfriend /from a /to 15-11-2019 15:00`
Expected: Error displayed regarding an invalid argument.

D.4.5. Adding an `assessment` task

1. Prerequisites: Module has been selected using the `module /select` command.
2. Test case: `assessment Finals /from 30-11-2019 10:00 /to 30-11-2019 12:00`
Expected: An acknowledgement message displayed regarding the successful addition of an `assessment` task for the selected `module`.
3. Test case: `assessment /from 30-11-2019 10:00 /to 30-11-2019 12:00`
Expected: Error displayed regarding a missing argument.
4. Test case: `assessment Finals /from /to 30-11-2019 12:00`
Expected: Similar to previous
5. Test case: `assessment Finals /from a /to 30-11-2019 12:00`
Expected: Error displayed regarding an invalid argument.

D.4.6. Deleting a task

1. Prerequisites: List all tasks using the `list` command. At list 1 task is on the list.
2. Test case: `delete 1`
Expected: The 1st task is deleted from the list of tasks. Details of the deleted task will be echoed as an acknowledgement message.
3. Test case: `delete 0`
Expected: Error displayed regarding an invalid index.
4. Other incorrect delete commands to try: `delete a`, `delete -1`
Expected: Similar to previous.

D.5. Starting Task Tracker

D.5.1. Understanding the parameters

The `tracker /start` command syntax is as such: `tracker /start TASK_INDEX MODULE_CODE`.

D.5.2. Testing the command

You can enter the command `tracker /start TASK_INDEX MODULE_CODE` to begin tracking your `Task` in `TaskList` given index with association to `moduleCode`.

Feel free to enter a `TASK_INDEX` of your choice.

D.6. Stopping Task Tracker

D.6.1. Understanding the parameters

The `tracker /stop` command syntax is as such: `tracker /stop TASK_INDEX MODULE_CODE`.

D.6.2. Testing the command

You can enter the command `tracker /stop TASK_INDEX MODULE_CODE` to stop tracking your Task in TaskList given index with association to `moduleCode`.

Feel free to enter a `TASK_INDEX` of your choice.

D.7. Pausing Task Tracker

D.7.1. Understanding the parameters

The `tracker /pause` command syntax is as such: `tracker /pause TASK_INDEX MODULE_CODE`.

D.7.2. Testing the command

You can enter the command `tracker /pause TASK_INDEX MODULE_CODE` to stop tracking your Task in TaskList given index with association to `moduleCode`.

Feel free to enter a `TASK_INDEX` of your choice.

D.8. Viewing Task Tracker Diagram

D.8.1. Understanding the parameters

The `tracker /view` command syntax is as such: `tracker /view PERIOD`.

D.8.2. Testing the command

You can enter the command `tracker /view PERIOD` to view a diagram showing the amount of time spent studying by `moduleCode` within a given `PERIOD`.

Feel free to enter one of the following options for `PERIOD`:

- `day` for today's entries
- `week` for entries over the last 7 days
- `all` for all entries

D.9. Viewing a list of Task Trackers

D.9.1. Understanding the parameters

The `tracker /list` command syntax is as such: `tracker /list`.

D.9.2. Testing the command

You can enter the command `tracker /list` to view a list of all `Tracker` objects you have created so far.

D.10. Deleting a Task Tracker

D.10.1. Understanding the parameters

The `tracker /delete` command syntax is as such: `tracker /delete TASK_INDEX`.

D.10.2. Testing the command

You can enter the command `tracker /delete TASK_INDEX` to delete a `Tracker` you have created. Feel free to enter a `TASK_INDEX` of your choice.

D.11. Setting recurring tasks

D.11.1. Selecting the task to be recurred

1. You should use the `list` command to list the `tasks` you have added to `OOF`.
2. Keep in mind the `task` you wish to recur.

D.11.2. Choosing the number of recurrences and frequency

3. A valid `number of recurrence` is an integer between `1 - 10`. You can choose a valid number within this range.
4. A valid `frequency` is an integer from `1 - 4` representing `DAILY`, `WEEKLY`, `MONTHLY` and `YEARLY` respectively. You can choose a valid `frequency` within this range.

D.11.3. Entering the command

5. You can then proceed to enter a command based on the parameters you have chosen.



The command is in the format `recurring INDEX NUMBER_OF_RECURRENCES FREQUENCY`.
`INDEX` refers to the `index` of the `task` you have chosen in step 2.
`NUMBER_OF_RECURRENCES` refers to the number you have chosen in step 3.
`FREQUENCY` refers to the number you have chosen in step 4.
You are free to test out this command with variations of the three parameters.

1. Test case: `recurring 1 1 1`
Expected: `1` task is added to the list of tasks with a date that is `1` day later than the date of the `1st` task in the list of tasks.
2. Test case: `recurring -1 1 1`
Expected: Error displayed to warn you to select a valid task.
3. Test case: `recurring 1 -1 1`
Expected: Error displayed to warn you to enter a valid number of recurrences.

4. Test case: `recurring 1 1 a`

Expected: Error displayed to warn you to enter valid numbers.

D.12. Finding free time slots

D.12.1. Understanding the parameters

The `free` command syntax is as such: `free DD-MM-YYYY`.



The `date` has to strictly be in the format `DD-MM-YYYY`.
You must enter either today's date or a date in the future.

D.12.2. Testing the command

You can enter the command `free DD-MM-YYYY` to view your free time slots on the given date as well as suggestions for upcoming deadlines to complete.

1. Test case: `free 12-12-2020`
Expected: The free time on `12-12-2020` will be displayed along with suggestions for deadlines to complete, if applicable.
2. Test case: `free 01-01-2010`
Expected: Error displayed to warn users to enter either the current date or a date in the future.
3. Test case: `free 32-32-2019`
Expected: Error displayed regarding invalid date.
4. Other incorrect free commands to try: `free a`, `free -1`
Expected: Similar to previous.

D.13. Viewing your tasks in a calendar format

D.13.1. Understanding the parameters

The `calendar` command syntax is as such: `calendar MM YYYY`.



The parameters `MM` and `YYYY` are optional. If either `MM` or `YYYY` is invalid or not provided, the current month (according to system settings) will be displayed.

D.14. Viewing your tasks for a week in a tabular format

D.14.1. Understanding the parameters

The `viewweek` command syntax is as such: `viewweek DD MM YYYY`.



The date parameters `DD`, `MM` and `YYYY` are optional. If the date is not entered, the current week (according to system settings) will be displayed. Similarly, if there is an error in the date supplied, the command will ignore the provided argument and print `tasks` for the current week.

D.14.2. Testing the command

You can enter the command `viewweek` to view `tasks` for the current week.

You are free to enter a `date` of your choice and observe the output of this command.

1. Test case: `viewweek 32 12 2019`

Expected: The `tasks` for the current week are shown.

2. Test case: `viewweek 30 12 2019`

Expected: The `tasks` in the week of `30-12-2019` are shown.