# Project Portfolio Page

Author: Denis Tjia <denis@u.nus.edu>

## Preface

The purpose of this portfolio is to document the contributions that I have made to a software engineering project, during my second year of study at the National University of Singapore.

## About the Project

The project was structured to resemble an intermediate stage of a real life software project. We were tasked to implement enhancements to the software, and also ensure that the finished product can be easily taken over by other developers.

My team of five decided to remix the application to keep track of events and tasks, very much like a calendar application. Our application, named **Horo**, aims to allow efficient management and organization of daily events and tasks, for any busy student.

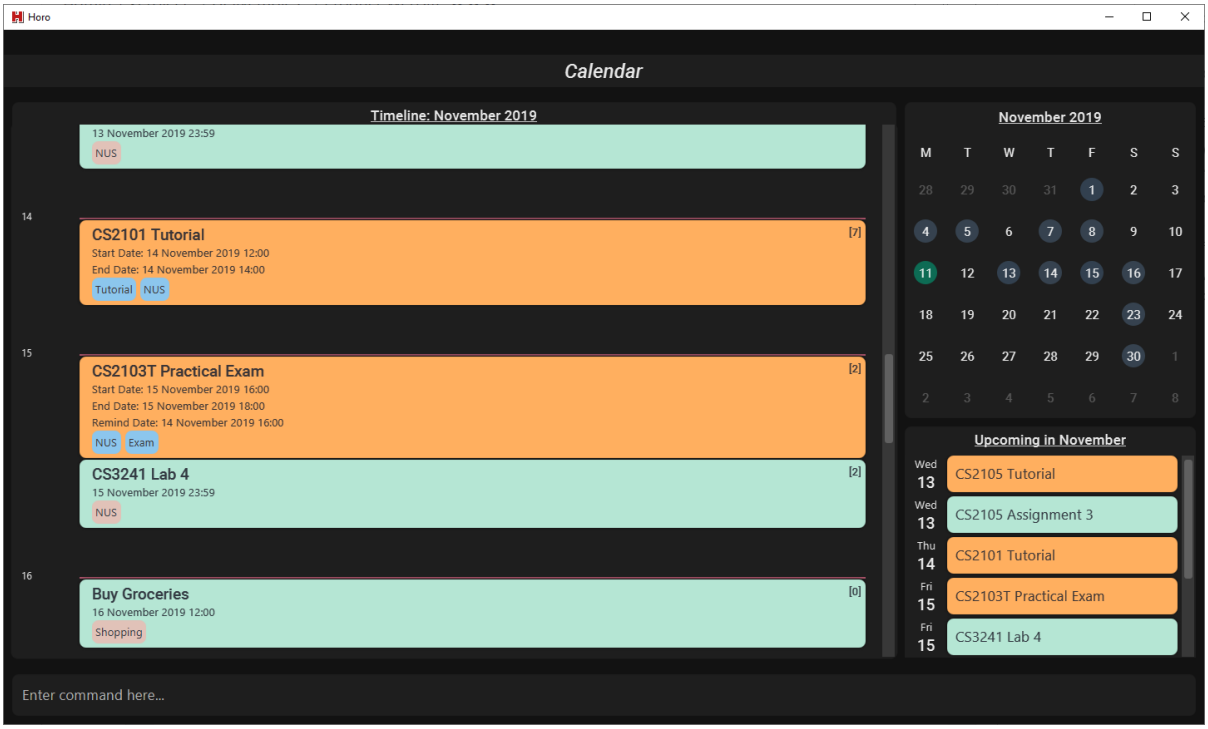


*Figure 1. Graphical user interface (GUI) of Horo*

For this project, I was given a few critical roles:

1. Designing the overall code architecture.
2. Implementation of the `Logic` and `Model` in Horo.

## Major Contribution

| | |
|---|---|
| **What** | Design of `CommandManager`, under `Logic`. The `CommandManager` class manages the addition and invoking of `Commands` in **Horo**. |
| **Why** | This feature serves to make it very simple for future developers to add new commands. It was also designed to be very extensible for developers to build upon it. |
| **Highlights** | • `CommandBuilder` is an application programming interface (API) for adding new commands.<br>• `CommandParser` parses user input using a finite state machine (FSM). |
| **Functional Code** | CommandManager, CommandBuilder, CommandParser |
| **Test Code** | CommandManagerTest, CommandParserTest |
| **Credits** | Design Patterns: Elements of Reusable Object-Oriented Software (1994) by The "Gang of Four". |

## Minor Contribution

| | |
|---|---|
| **What** | Implementation of `AddEventCommand`, `DeleteEventCommand` and `EditEventCommand`. |
| **Why** | These commands allow the user to create, modify and delete events in Horo! |
| **Functional Code** | AddEventCommand, DeleteEventCommand, EditEventCommand |
| **Test Code** | AddEventCommandTest, DeleteEventCommandTest, EditEventCommandTest |

## Other Contributions

| | |
|---|---|
| **Project Management** | • Managed all releases (v1.1 - v1.4)<br>• Integration of my team's code: #49 #86 #132 |
| **Community** | • Give suggestions to a team member: #52<br>• Report bugs to a team member: #72 |
| **Tools** | Integrated a third party library JSONAssert, to help with JSON unit testing. #223 |
| **Code Contribution** | RepoSense |

### User Guide Contributions

The section below contains excerpts taken from the User Guide, written by me.

# 3. Features

#### Command Format for Horo

- You can enter a command by typing the command name first followed by arguments of the command.
  - E.g. `command_name <argument1> <argument2>⋯`
- If you want to type an argument containing blank spaces, surround your sentence within quotation marks.
  - E.g. `"Buy Groceries"`
  - E.g. `add_task "Buy Groceries"`
- Command parameters are prefixed by `--`, and they are optional.
  - E.g. `--date 09/2019`
  - E.g. `calendar --date 09/2019`
- Please take note that commands with extra arguments given are truncated.
  - E.g. The `calendar` command does not accept any arguments, therefore if you enter `calendar 09/2019`, `09/2019` will be silently discarded.

#### Command Format for this User Guide

- Arguments are fully capitalized which are to be supplied by you.
  - E.g. `add_task TASK`
- Arguments suffixed with ellipsis (⋯) indicate that the argument can be entered zero or more times.
  - E.g. `delete_task INDEX⋯`
- Parameters separated by a vertical bar `|` can be used interchangeably.
  - e.g. given the command format `-d|--description DESCRIPTION`, the following inputs accomplish the same thing:
  - `--description "Buy Groceries"`
  - `-d "Buy Groceries"`

## 3.2.2. Deleting Events

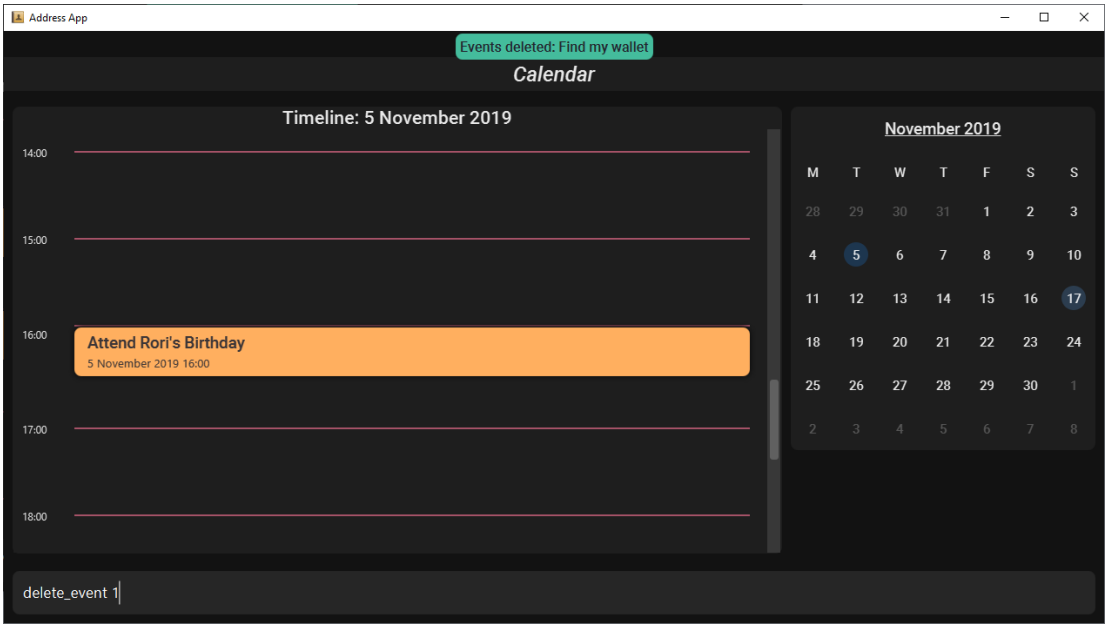


*Figure 2. Delete Event Command*

You may use the `delete_event` command to delete one or more events from the calendar.

Command Format:

> **delete_event** INDEX...

Command Parameters:

- `--tag TAG…`

Examples:

- `delete_event 0`
- `delete_event 0 1 2` : Deletes events 0, 1 and 2.
- `delete_event --tag Birthday` : Deletes events tagged as `Birthday`.
- `delete_event --tag Birthday Rori` : Deletes events tagged as `Birthday` and `Rori`.
- `delete_event 0 1 2 --tag Birthday Rori` : Deletes events 0, 1 and 2 only if they have been tagged as `Birthday` and `Rori`.

**NOTE**

- Events which match both `INDEX` and `TAG` will be deleted.
- `INDEX` can be found on the event's top right corner.


*Figure 3. Index found on the top right of the event.*

### Developer Guide Contributions

> The section below contains excerpts of the CommandManager taken from the Developer Guide, written by me.

## 3.3 CommandManager Component

The `CommandManager` class manages the addition and invoking of `Commands` in Horo.

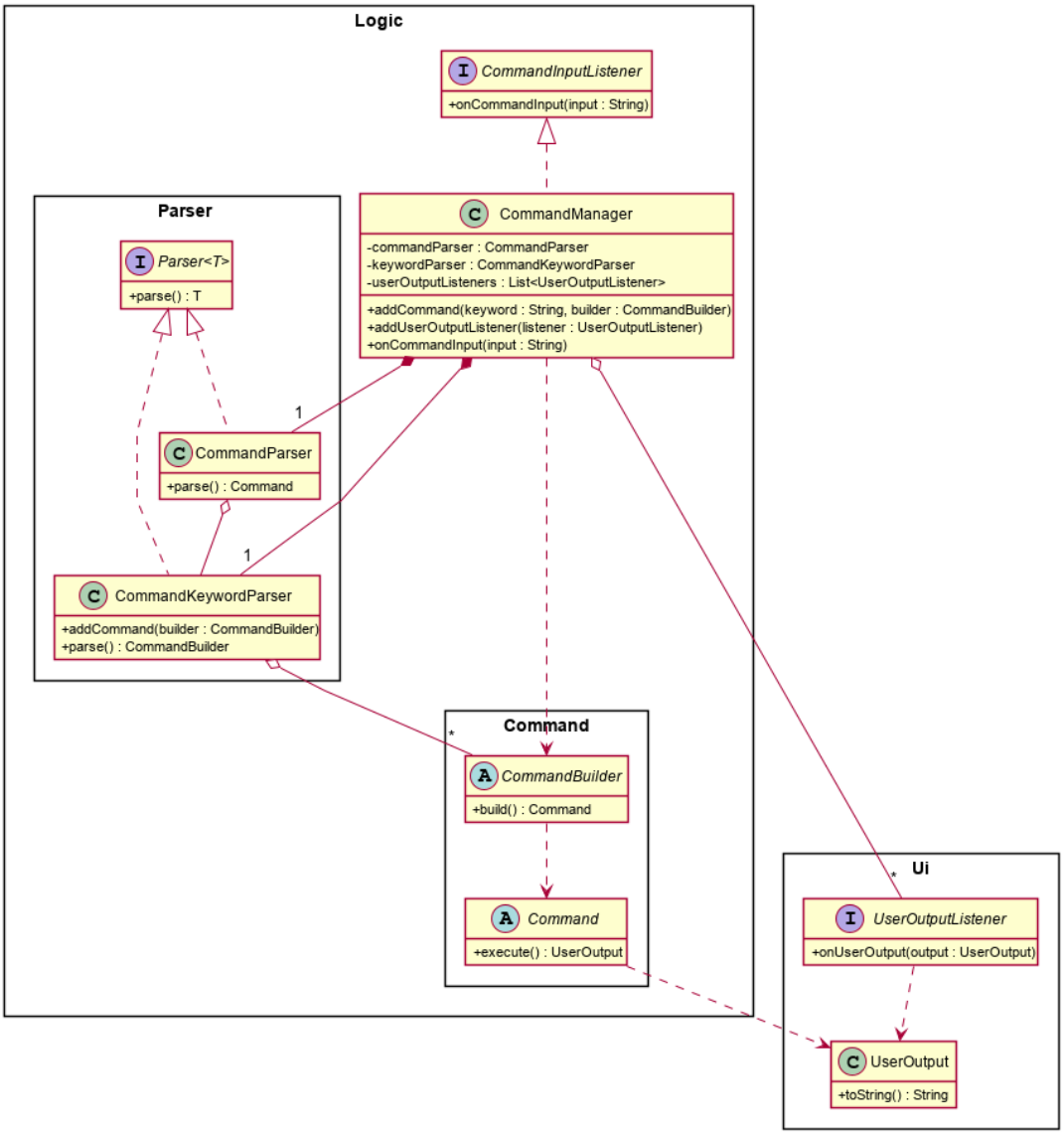


*Figure 4. Class diagram of CommandManager*

Referring to the diagram above, it performs the following operations:

1. Listen for user input in `onCommandInput()`.
2. Pass the user input to `commandParser`, to obtain a `Command`.
3. Execute the `Command` and obtain a `UserOutput`.
4. Notify all `userOutputListeners` about the `UserOutput`.

To give a more concrete example of how `CommandManager` functions, refer to the sequence diagram below of **adding a task** to Horo:

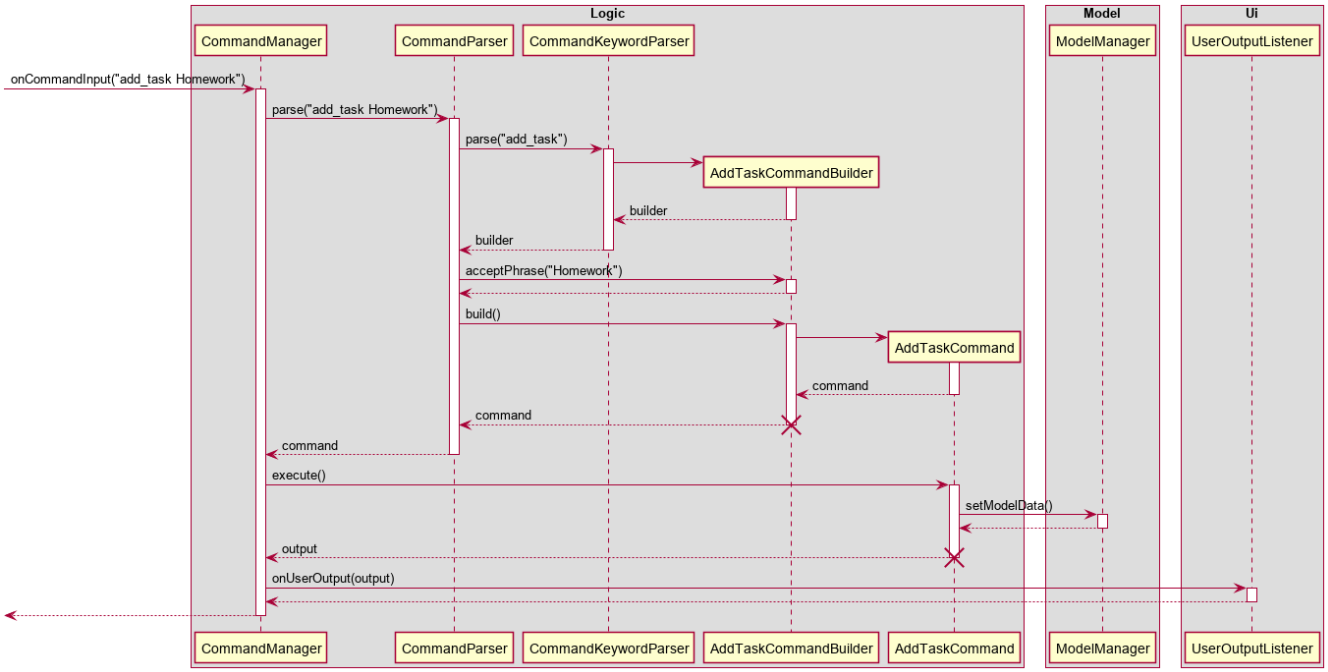


*Figure 5. Sequence diagram of adding a task*

## 3.3.2 CommandParser

`CommandParser` is trying to tokenize any command input into one command keyword, and zero or more command phrases. (i.e. `[keyword] [phrase] [phrase] [phrase] ⋯`). To understand how the FSM works, study the activity diagram below:

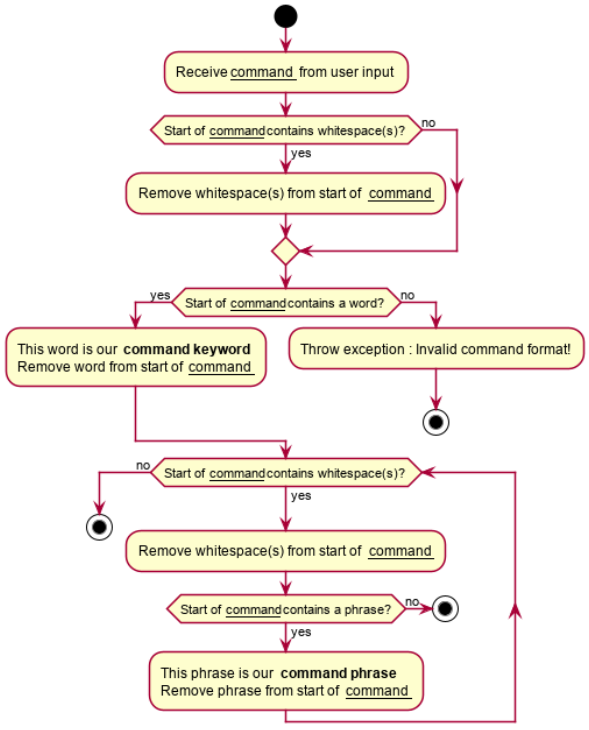


*Figure 6. Activity diagram of CommandParser*

**Design Considerations**

| | Option 1 | Option 2 |
|---|---|---|
| **What** | Use `String.split()` to break up command input into tokens. | Implement a FSM to break up command input into tokens. |
| **Difficulty** | Easy | Moderate |
| **How** | Split the command input by whitespaces, into words. The first word will be the **command keyword**. All subsequent words will have to be joined into **command phrases**. | Create a `State` class, and design a state diagram to tokenize the command input into a **command keyword** and **command phrases**. |
| **Evaluation** | I did not choose this option because:<br><br>Joining words into command phrases can become quite complex, especially when introducing quotation marks.<br><br>Additionally, it is difficult for future developers to maintain and extend logic like this. | **I chose this option because:**<br><br>A state machine is easy to understand and configure.<br><br>A state machine can tokenize complex command inputs. This grants future developers the ability to parse more advanced command inputs. |

## 3.3.4 CommandBuilder

A `CommandBuilder` is defined to be able to accept an arbitrary amount of command phrases, and eventually create a `Command` using those phrases.

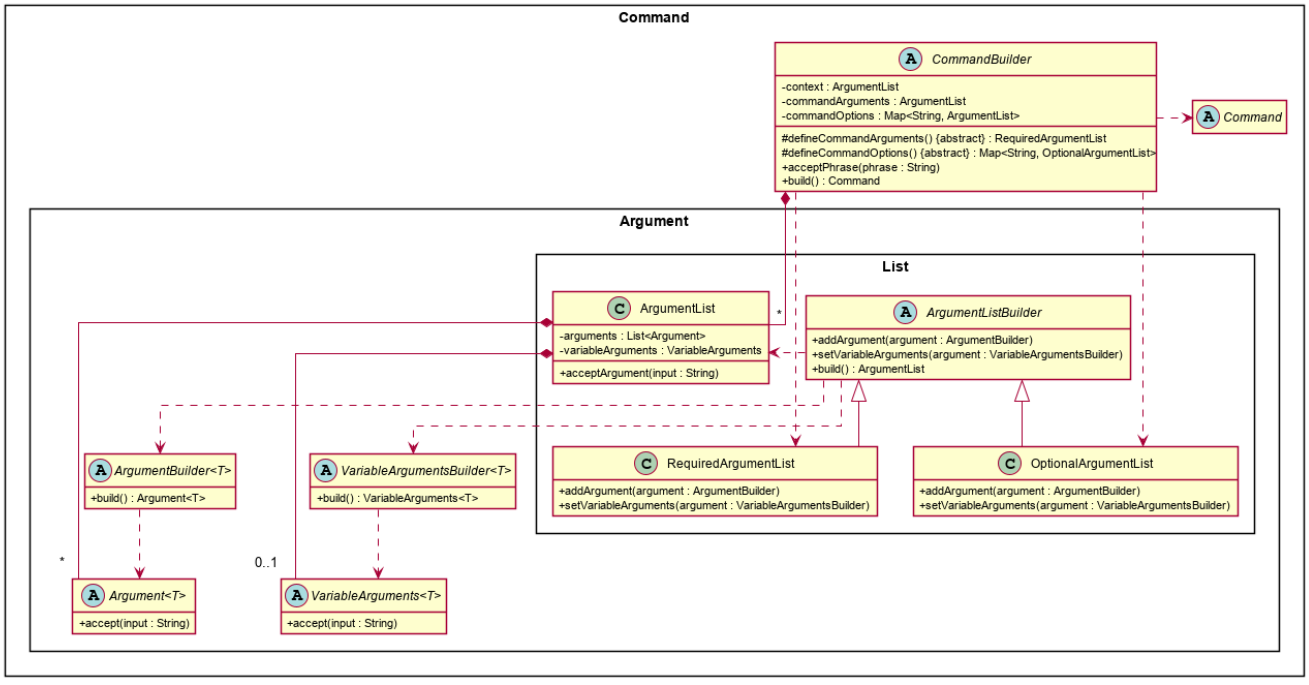


*Figure 7. Class diagram of CommandBuilder*

Referring to the diagram above, the definition the command is implemented in two methods:

1. `defineCommandArguments()`
2. `defineCommandOptions()`

...

To understand how `CommandBuilder` works, study the activity diagram below:

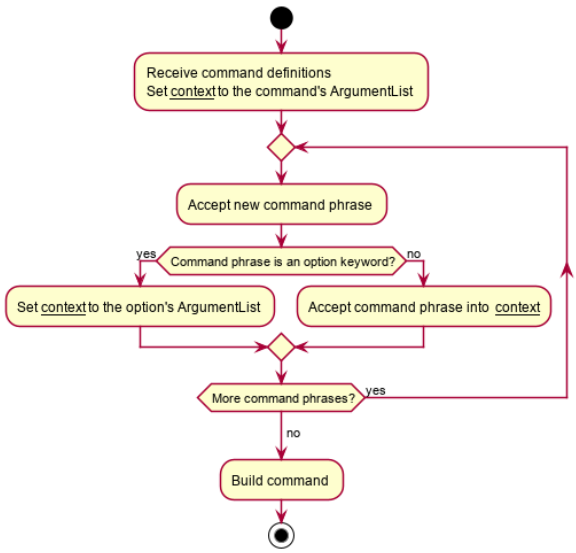


*Figure 8. Activity diagram of CommandBuilder*

**Design Considerations**

| | **Option 1** | **Option 2** |
|---|---|---|
| **What** | Each `Command` is created by parsing user input using it's own `Parser`. | Each `Command` is defined by a `CommandBuilder`, and created by a `CommandParser`. |
| **Difficulty** | Easy | Moderate |
| **How** | Implement a utility class which can parse user input into arguments. Use this class in each command parser. | Implement `CommandBuilder` which can build a `Command` with any number of arguments. Commands provide what arguments they require. |
| **Evaluation** | I did not choose this option because:<br><br>Each command parser will need to implement logic to use the utility class, handle argument checking and parsing errors.<br><br>It is difficult for future developers to create, extend and test `Commands`. | **I chose this option because:**<br><br>Each `Command` does not require any logic, only arguments are required to be defined.<br><br>Since all logic is in `CommandBuilder`, it is simple for developers to test. |

The section below contains excerpts of the Architecture and Model from the Developer Guide, written by me.

# 3.1. Architecture

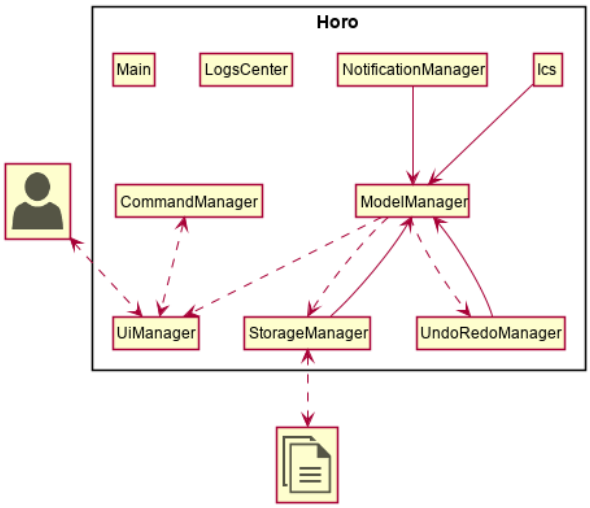


*Figure 9. Architecture Diagram*

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

...

The rest of Horo is managed by seven components:

1. `CommandManager` : Responsible for executing commands from the user.
2. `Ics` : Responsible for handling the importing and exporting of ICS files.
3. `ModelManager` : Responsible for reading and writing to the in-memory data of Horo.
4. `NotificationManager` : Responsible for sending notifications to the user.
5. `StorageManager` : Responsible for reading and writing the Model to an external disk.
6. `UiManager` : Responsible for managing the user interface (UI) of Horo.
7. `UndoRedoManager` : Responsible for tracking changes in ModelManager, and reverting its history when needed.

Most components follow the observer design pattern, to reduce tight coupling and increase cohesion. They implement these listeners:

1. `CommandInputListener` : The component will be notified about command input from the user.
2. `ModelDataListener` : The component will be notified whenever the Model changes.
3. `UserOutputListener` : The component will be notified whenever a message needs to be displayed to the user.

# 3.4. ModelManager Component

The `ModelManager` is responsible for the reading and writing of events and tasks in Horo.

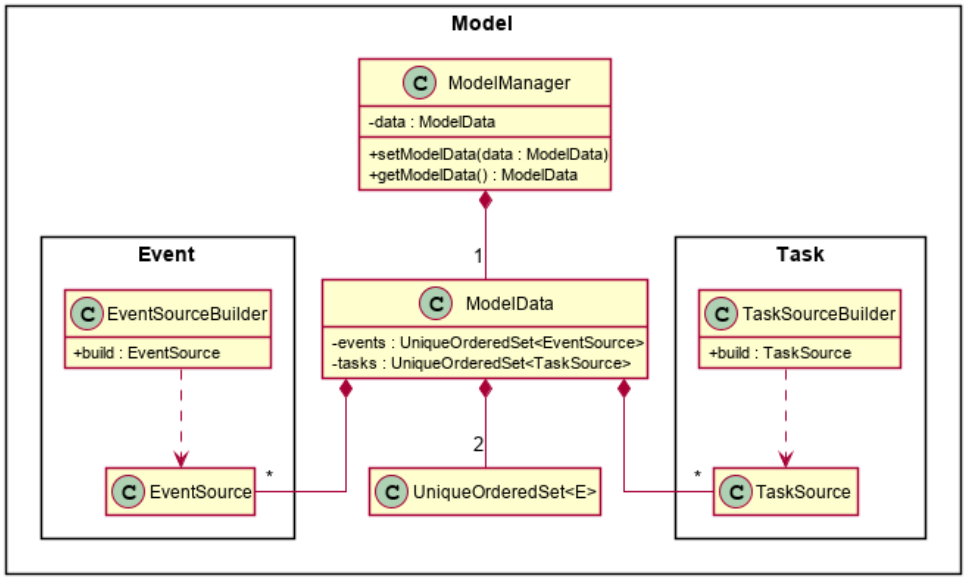


*Figure 10. Class diagram of the ModelManager*

The `ModelManager` has three main functions:

1. Stores all events and tasks in a wrapper class `ModelData`.
2. Notifies all `ModelDataListeners` whenever the `ModelData` changes.
3. Allows any class with a reference to `ModelManager` to update the current `ModelData`.

To give a more concrete example of how `ModelManager` notifies its listeners, refer to the sequence diagram below of **adding a task** to Horo:

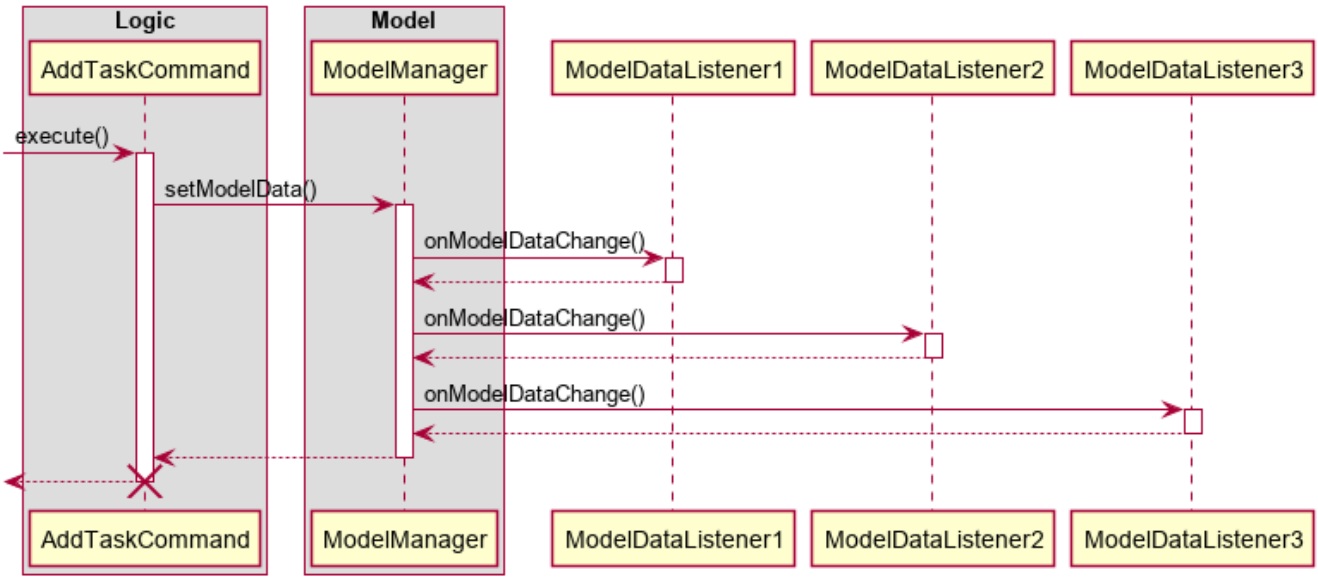


*Figure 11. Sequence diagram of adding a task*