Mod Manager - Developer Guide

By: `Team AY1920S2-CS2103T-F10-4` Since: `Jan 2020` Licence: `MIT`

1. Introduction

1.1. Purpose

This document describes the architecture and system design of Mod Manager.

1.2. Audience

The developer guide is for software developers, designers and testers who wants to understand the architecture and system design of Mod Manager.

2. Setting up

Refer to the guide here.

3. Design

3.1. Architecture

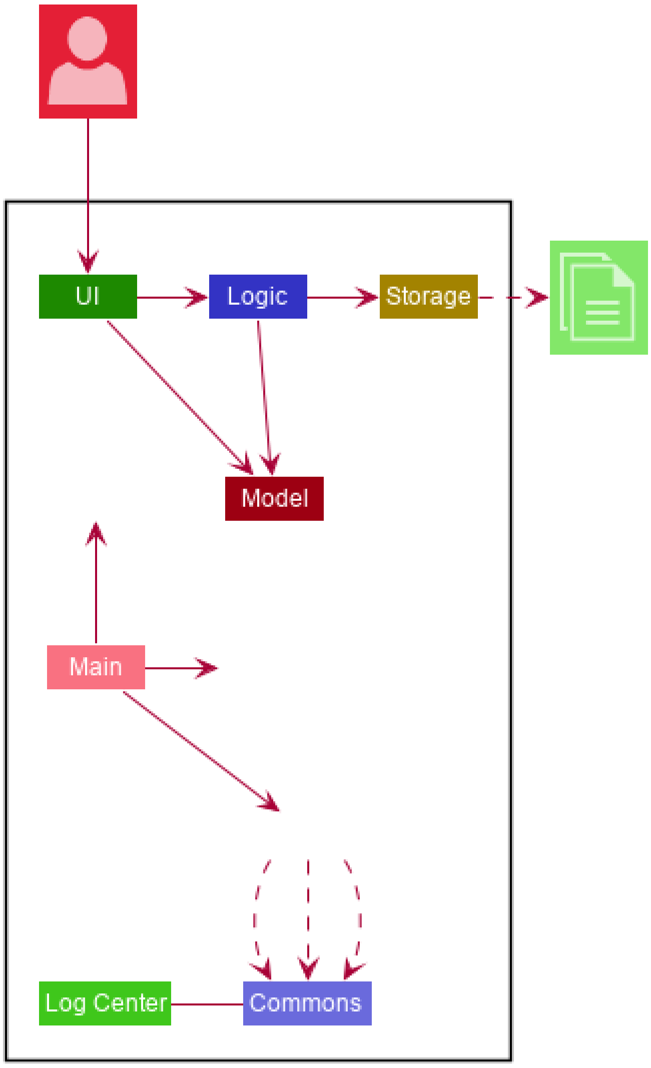


Figure 1. Architecture Diagram

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

`Main` has two classes called `Main` and `MainApp`. It is responsible for,

- At app launch: Initializes the components in the correct sequence, and connects them up with each other.
- At shut down: Shuts down the components and invokes cleanup method where necessary.

`Commons` represents a collection of classes used by multiple other components. The following class plays an important role at the architecture level:

- `LogsCenter` : Used by many classes to write log messages to the App's log file.

The rest of the App consists of four components.

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

Each of the four components

- Defines its *API* in an `interface` with the same name as the Component.
- Exposes its functionality using a `{Component Name}Manager` class.

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

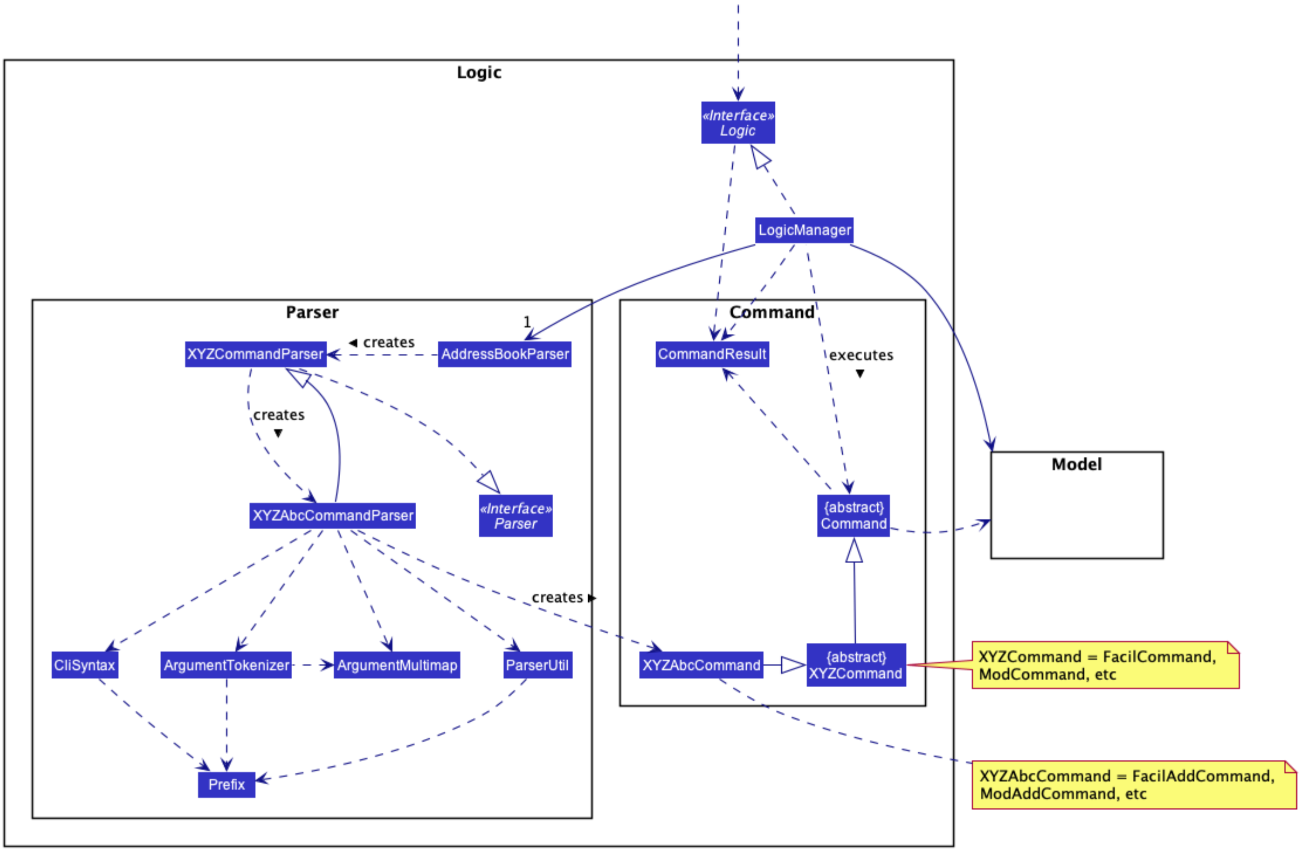


Figure 2. Class Diagram of the Logic Component

How the architecture components interact with each other

The *Sequence Diagram* below shows how the components generically interact with each other for the scenario where the user issues the command `some command`.

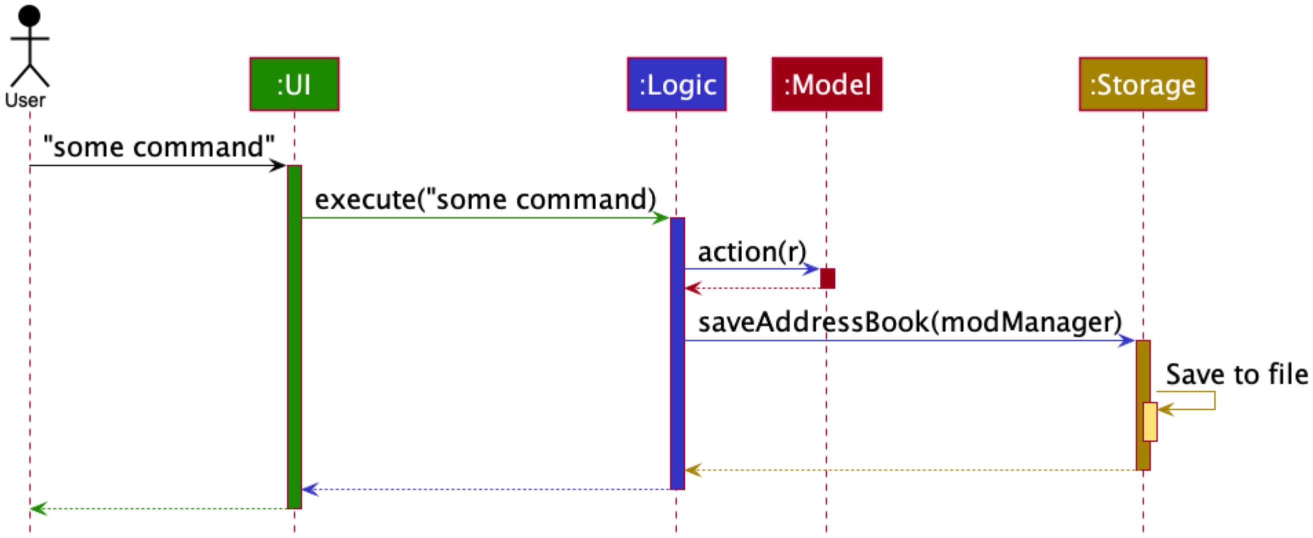


Figure 3. Component interactions for `some command` *command*

The sections below give more details of each component.

3.2. UI component

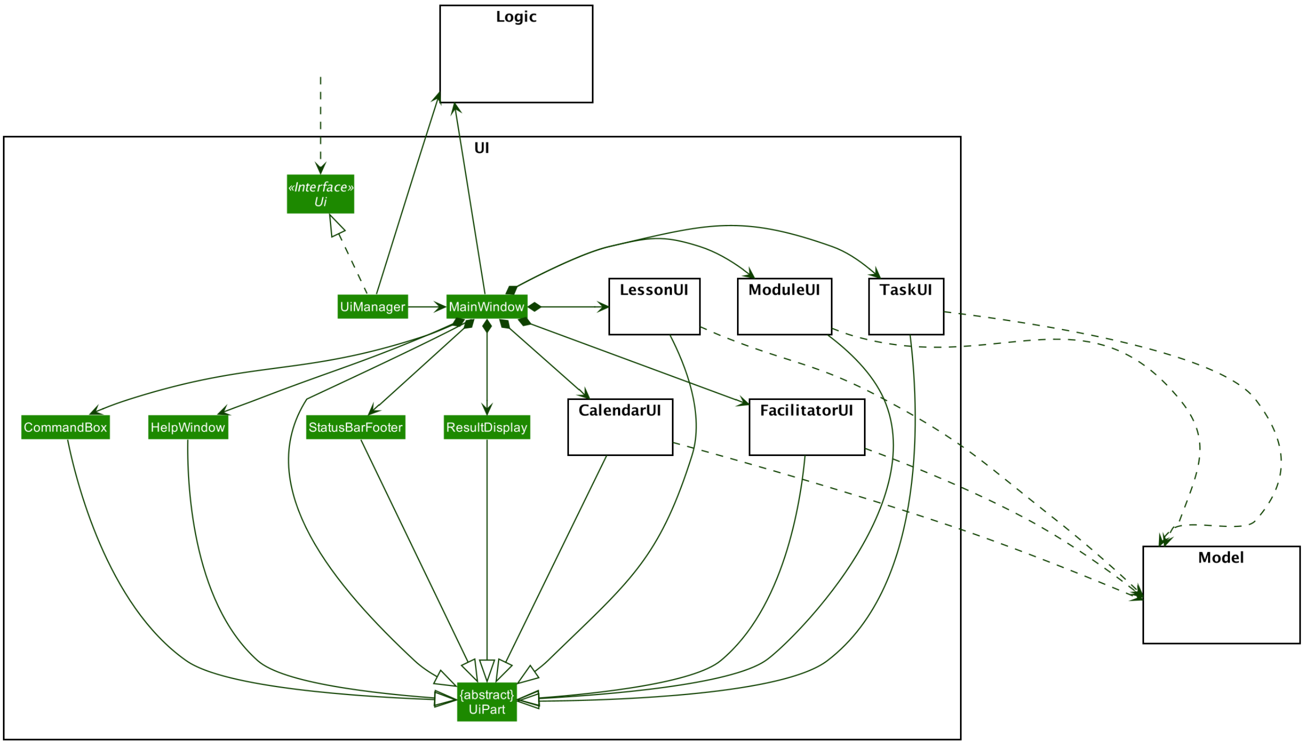


Figure 4. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `HelpWindow`, `StatusBarFooter` etc. All these, including the `MainWindow`, inherit from the abstract `UiPart` class.

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

The `UI` component,

- Executes user commands using the `Logic` component.
- Listens for changes to `Model` data so that the UI can be updated with the modified data.

3.3. Logic component

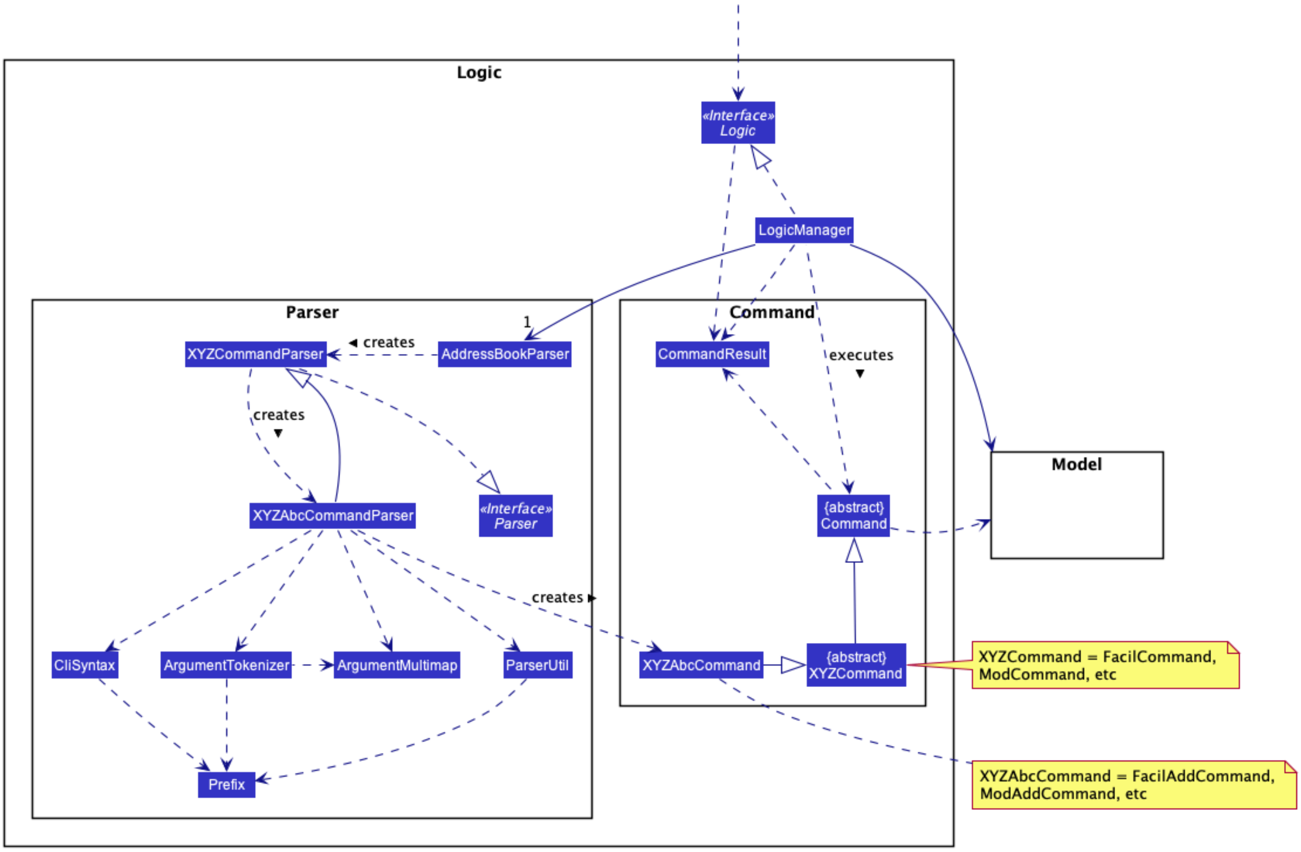


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `ModManagerParser` class to parse the user command.
2. This results in a `Command` object which is executed by the `LogicManager`.
3. The command execution can affect the `Model` (e.g. adding a facilitator).
4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.
5. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.

3.4. Model component

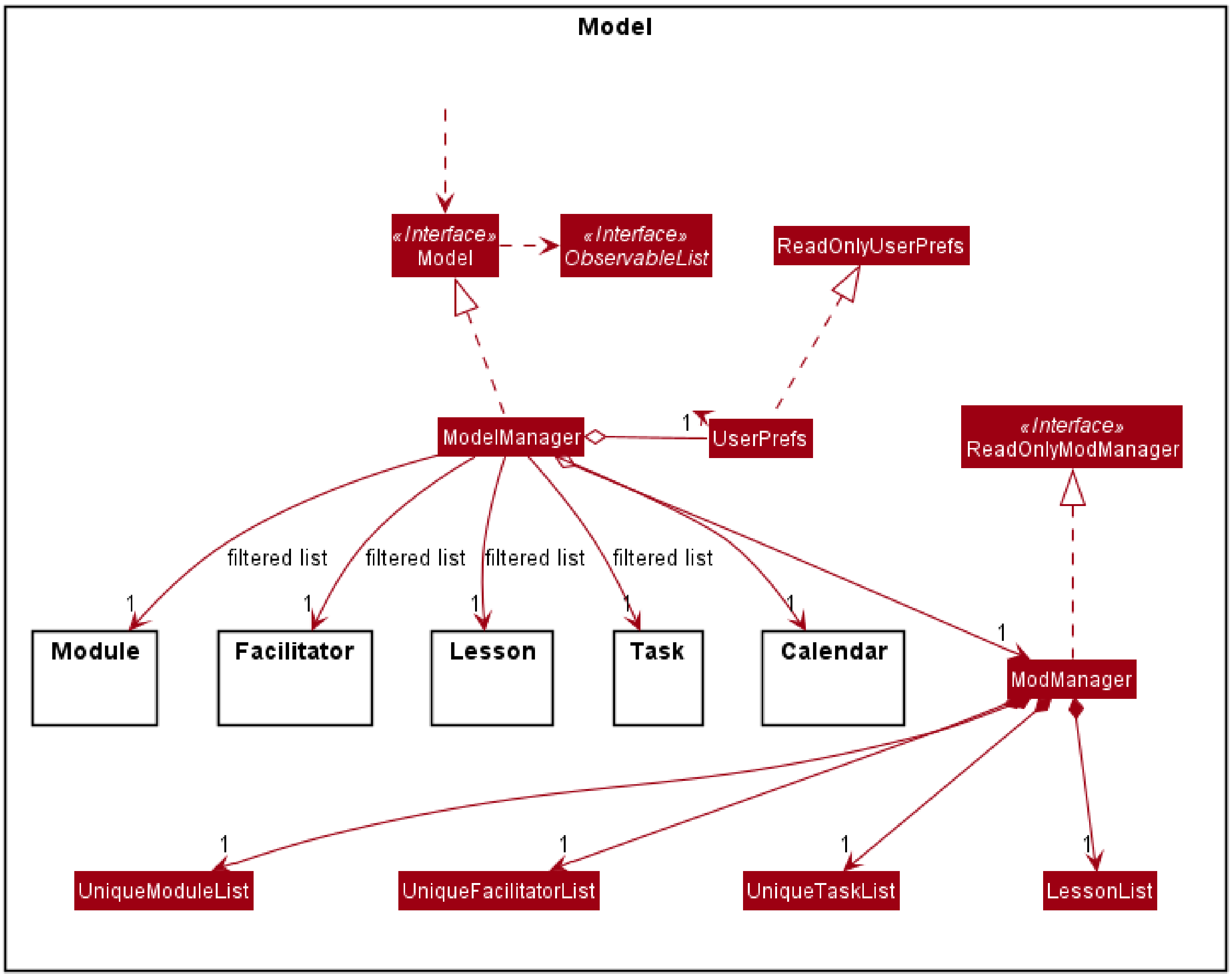


Figure 6. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user’s preferences.
- stores the Mod Manager data.
- exposes an unmodifiable `ObservableList<Facilitator>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- does not depend on any of the other three components.

3.5. Storage component

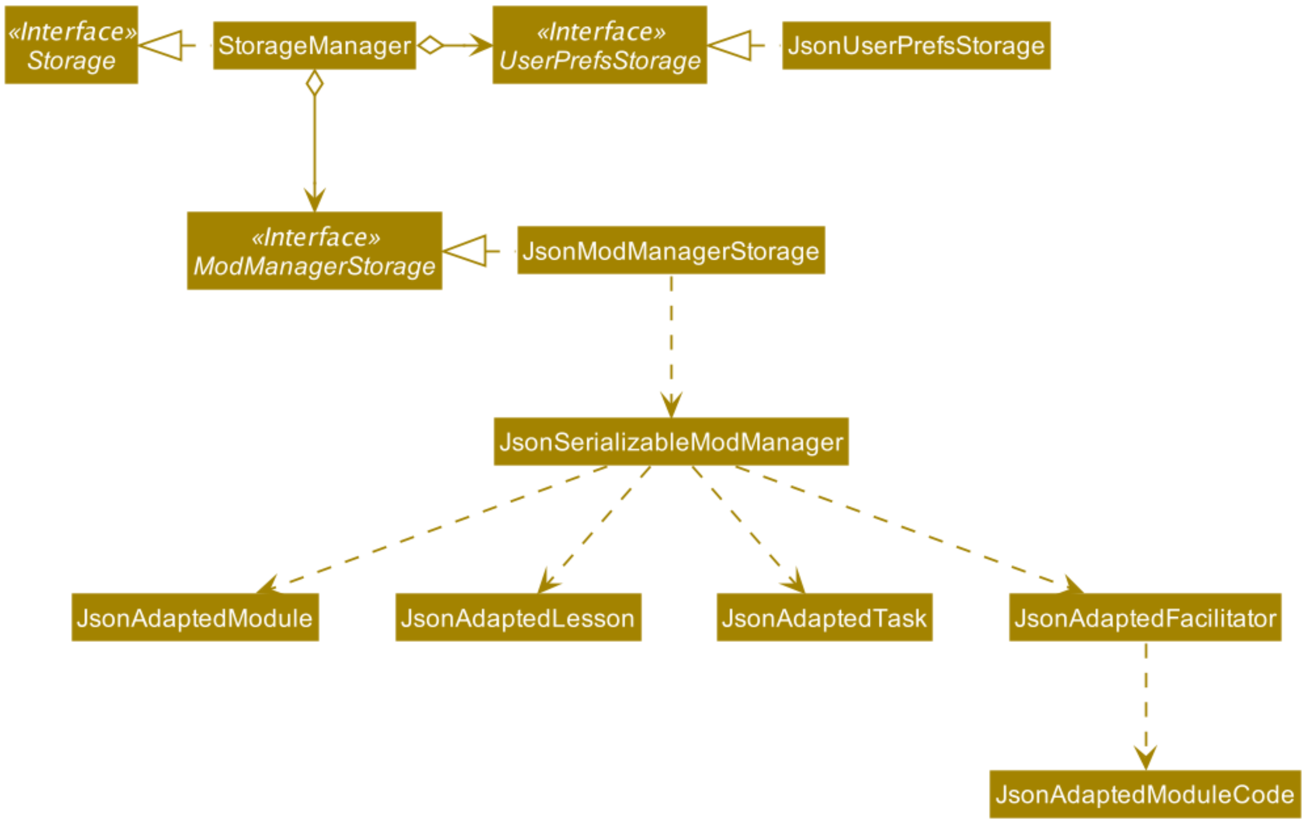


Figure 7. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

- can save `UserPref` objects in json format and read it back.
- can save the Mod Manager data in json format and read it back.

3.6. Common classes

Classes used by multiple components are in the `seedu.addressbook.commons` package.

4. Implementation

This section describes some noteworthy details on how certain features are implemented.

4.1. Modules Management Feature

The module feature manages the modules in Mod Manager and is represented by the `Module` class. A module has a `ModuleCode` and an optional `Description`.

It supports the following operations:

- add - Adds a module to Mod Manager.
- list - Lists all modules in Mod Manager.

- view - View information of a module in Mod Manager.
- edit - Edits a module in Mod Manager.
- delete - Deletes a module in Mod Manager.

4.1.1. Implementation Details

Adding a module

The add module feature allows users to add a module to Mod Manager. This feature is facilitated by `ModuleCommandParser`, `ModuleAddCommandParser` and `ModuleAddCommand`. The operation is exposed in the `Model` interface as `Model#addModule()`.

Given below is an example usage scenario and how the module add mechanism behaves at each step:

1. The user executes the module add command and provides the module code and description of the module to be added.
2. `ModuleAddCommandParser` creates a new `Module` based on the module code and description.
3. `ModuleAddCommandParser` creates a new `ModuleAddCommand` based on the module.
4. `LogicManager` executes the `ModuleAddCommand`.
5. `ModManager` adds the module to the `UniqueModuleList`.
6. `ModelManager` updates the `filteredModules` in `ModelManager`.

The following sequence diagram shows how the module add command works:

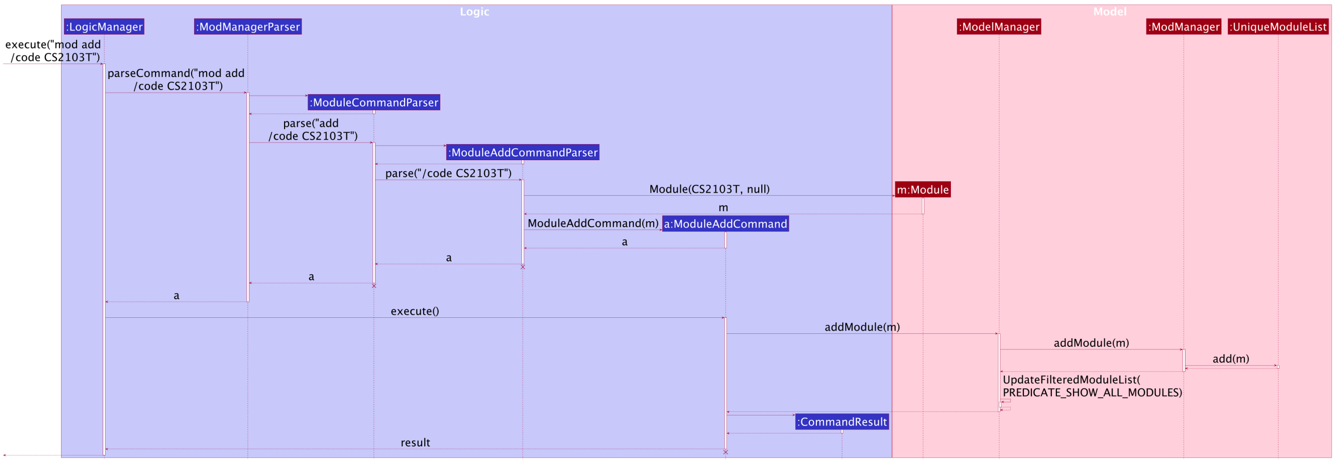


Figure 8. Sequence diagram for `mod add` command

NOTE The lifeline for `ModuleCommandParser`, `ModuleAddCommandParser` and `ModuleAddCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

The following activity diagram summarizes what happens when a user executes a module add command:

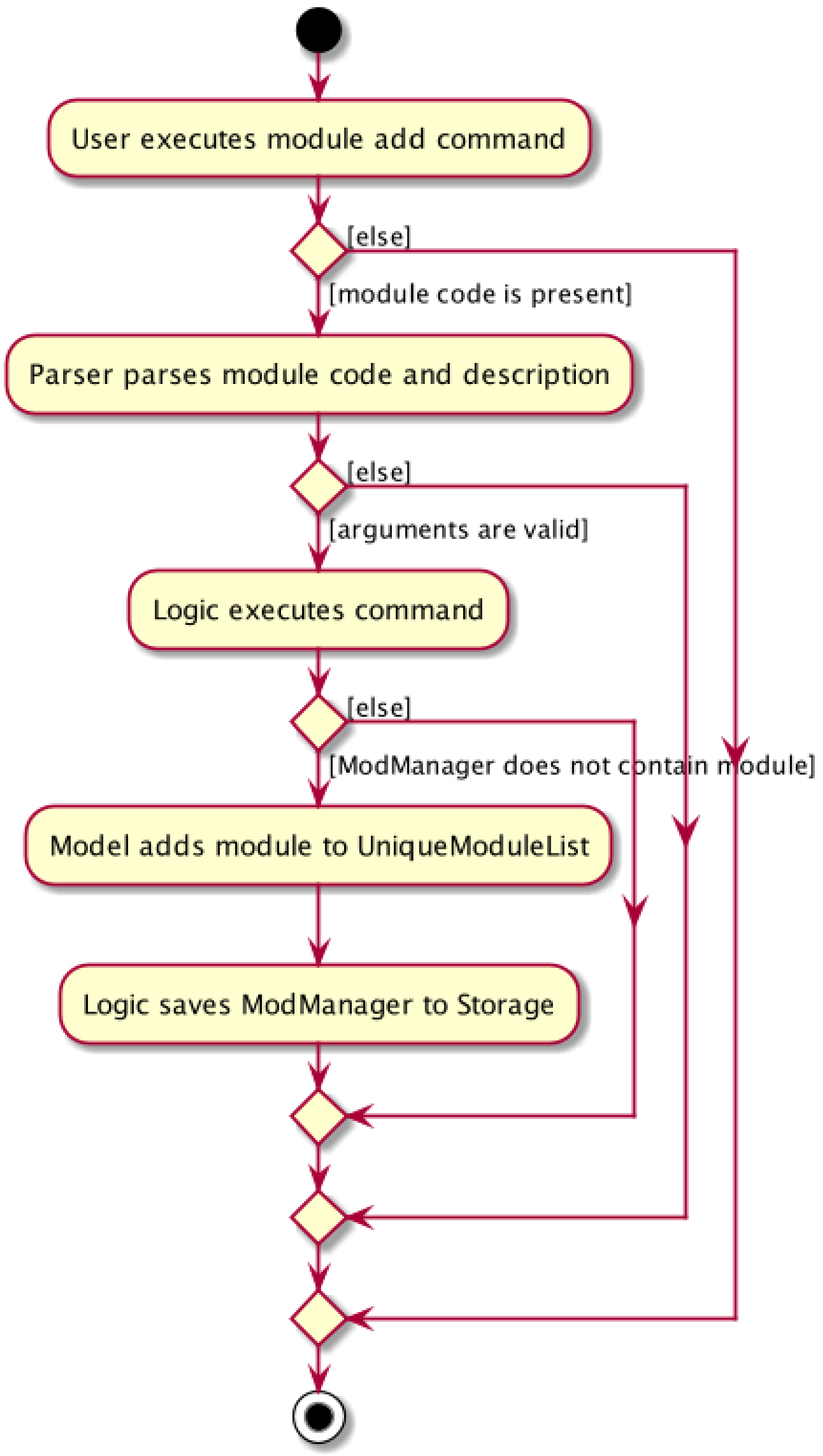

User executes module add command
[else]
[module code is present]
Parser parses module code and description
[else]
[arguments are valid]
Logic executes command
[else]
[ModManager does not contain module]
Model adds module to UniqueModuleList
Logic saves ModManager to Storage

Figure 9. Activity diagram for `mod add` command

Listing all modules

The list module feature allows users to list all modules in Mod Manager. This feature is facilitated by `ModuleCommandParser` and `ModuleListCommand`. The operation is exposed in the `Model` interface as `Model#updateFilteredModuleList()`.

Given below is an example usage scenario and how the module list mechanism behaves at each step:

1. The user executes the module list command.
2. ModuleCommandParser creates a new ModuleListCommand.
3. LogicManager executes the ModuleListCommand.
4. ModelManager updates the filteredModules in ModelManager.

The following sequence diagram shows how the module list command works:

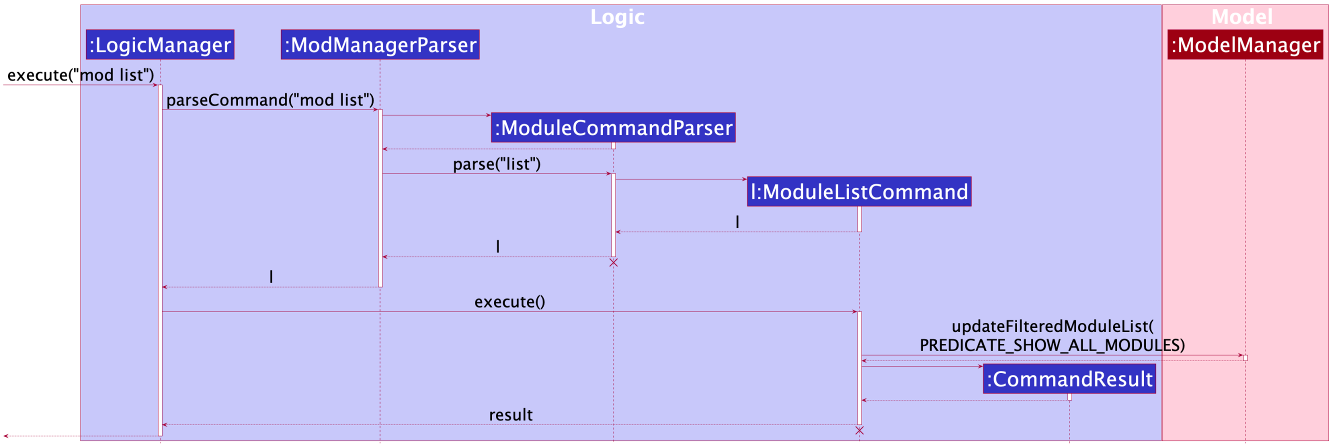
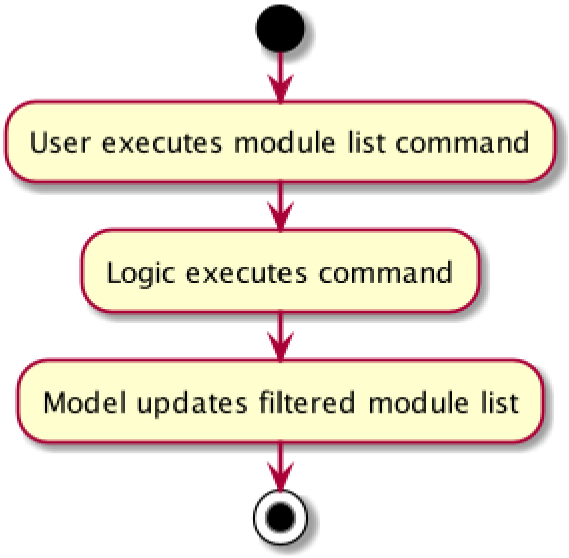


Figure 10. Sequence diagram for `mod list` command

NOTE The lifeline for ModuleCommandParser and ModuleListCommand should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

The following activity diagram summarizes what happens when a user executes a module list command:



User executes module list command
Logic executes command
Model updates filtered module list

Figure 11. Activity diagram for `mod list` command

4.1.2. Design Considerations

Aspect: Whether to support editing of module code

- **Alternative 1 (current choice):** Allow users to only edit the description of a module.
  - Pros: Easier to implement.
  - Cons: More rigid for users.
- **Alternative 2:** Allow users to edit the module code of a module.
  - Pros: Provides more flexibility for users.
  - Cons: Implementation is more complex as the classes, tasks and facilitators all store module codes and have to be edited too.

4.2. Facilitators Management Feature

The facilitator feature manages the facilitators in Mod Manager and is represented by the `Facilitator` class. A facilitator has a `Name`, an optional `Phone`, an optional `Email`, and optional `Office` and one or more `ModuleCode`. A `Module` with the `ModuleCode` of the facilitator should exist in Mod Manager.

It supports the following operations:

- add - Adds a facilitator to Mod Manager.
- list - Lists all facilitators in Mod Manager.
- view - Finds a facilitator in Mod Manager by name.
- edit - Edits a facilitator in Mod Manager.
- delete - Deletes a facilitator in Mod Manager.

4.2.1. Implementation Details

Adding a facilitator

The add facilitator feature allows users to add a facilitator to Mod Manager. This feature is facilitated by `FacilCommandParser`, `FacilAddCommandParser` and `FacilAddCommand`. The operation is exposed in the `Model` interface as `Model#addFacilitator()`.

Given below is an example usage scenario and how the facilitator add mechanism behaves at each step:

1. The user executes the facilitator add command and provides the name, phone, email, office and module code of the facilitator to be added.
2. `FacilitatorAddCommandParser` creates a new `Facilitator` based on the name, phone, email, office and module code.
3. `FacilitatorAddCommandParser` creates a new `FacilitatorAddCommand` based on the facilitator.
4. `LogicManager` executes the `FacilitatorAddCommand`.
5. `ModManager` adds the facilitator to the `UniqueFacilitatorList`.
6. `ModelManager` updates the `filteredFacilitators` in `ModelManager`.

The following sequence diagram shows how the facilitator add command works:

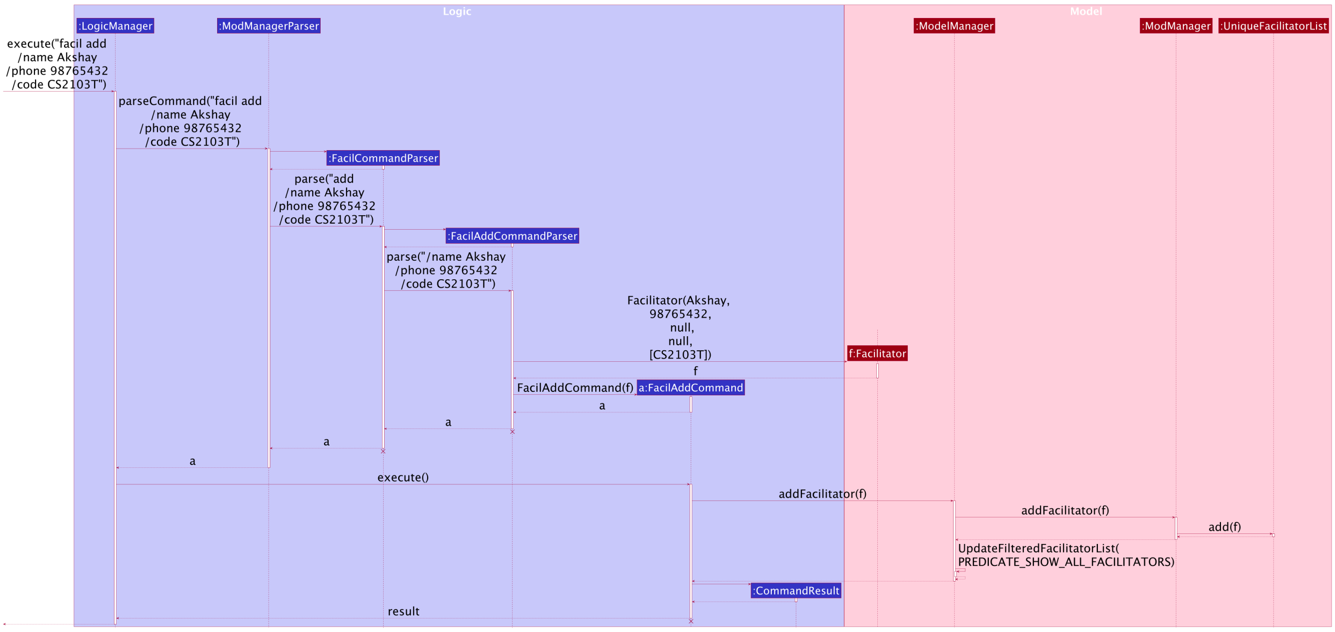

Logic
Model
:LogicManager
:ModManagerParser
:ModelManager
:ModManager
:UniqueFacilitatorList
execute("facil add /name Akshay /phone 98765432 /code CS2103T")
parseCommand("facil add /name Akshay /phone 98765432 /code CS2103T")
:FacilCommandParser
parse("add /name Akshay /phone 98765432 /code CS2103T")
:FacilAddCommandParser
parse("/name Akshay /phone 98765432 /code CS2103T")
Facilitator(Akshay, 98765432, null, null, [CS2103T])
f:Facilitator
f
FacilAddCommand(f)
a:FacilAddCommand
a
a
a
a
execute()
addFacilitator(f)
addFacilitator(f)
add(f)
UpdateFilteredFacilitatorList(PREDICATE_SHOW_ALL_FACILITATORS)
:CommandResult
result

Figure 12. Sequence diagram for `facil add` command

NOTE The lifeline for `FacilitatorCommandParser`, `FacilitatorAddCommandParser` and `FacilitatorAddCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

The following activity diagram summarizes what happens when a user executes a facilitator add command:

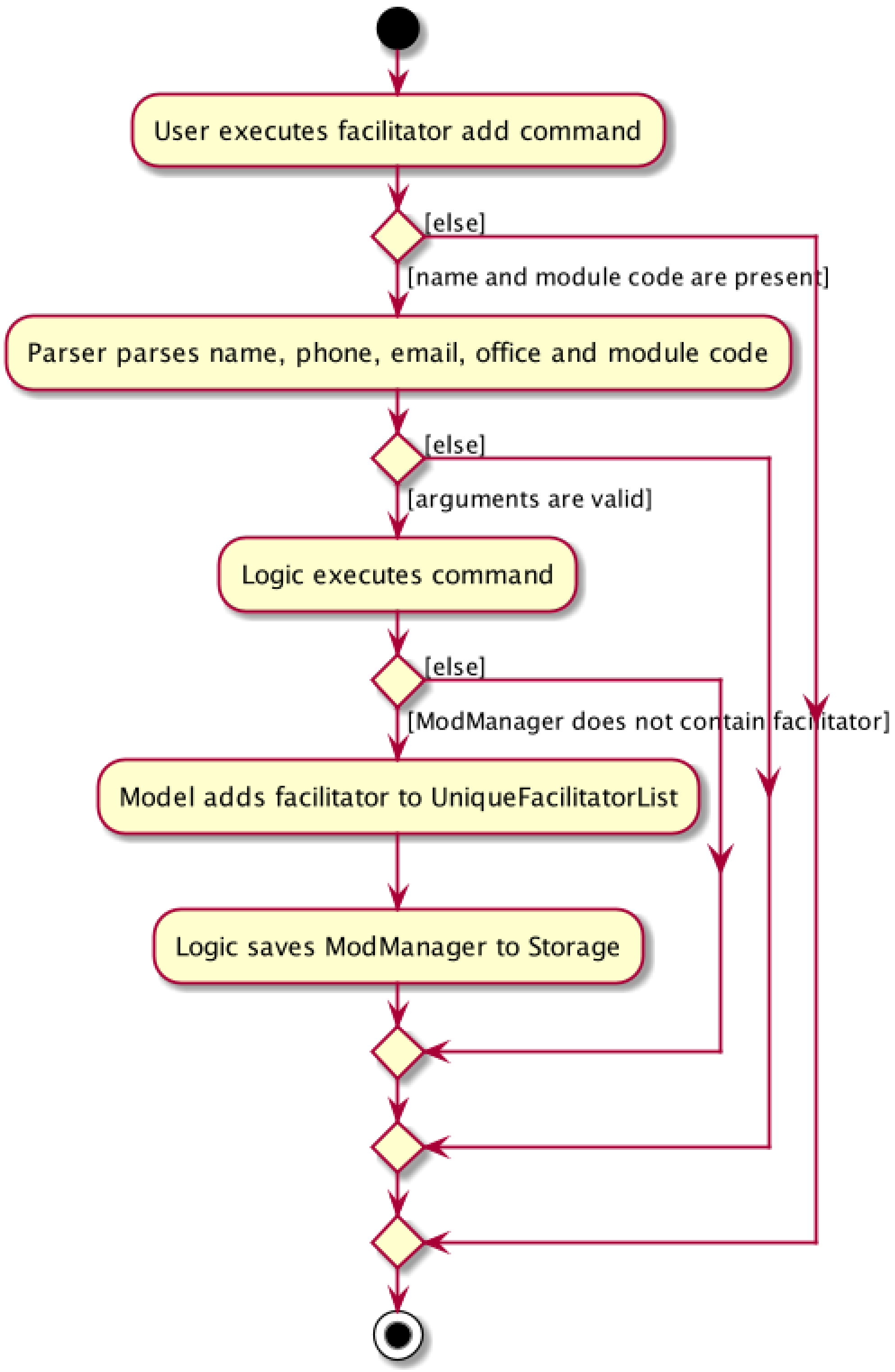

User executes facilitator add command
[else]
[name and module code are present]
Parser parses name, phone, email, office and module code
[else]
[arguments are valid]
Logic executes command
[else]
[ModManager does not contain facilitator]
Model adds facilitator to UniqueFacilitatorList
Logic saves ModManager to Storage

Figure 13. Activity diagram for `facil add` command

Listing all facilitators

The list facilitator feature allows users to list all facilitators in Mod Manager. This feature is facilitated by `FacilCommandParser` and `FacilListCommand`. The operation is exposed in the `Model` interface as `Model#updateFilteredFacilitatorList()`.

Given below is an example usage scenario and how the facilitator list mechanism behaves at each step:

1. The user executes the facilitator list command.
2. `FacilCommandParser` creates a new `FacilListCommand`.
3. `LogicManager` executes the `FacilListCommand`.
4. `ModelManager` updates the `filteredFacilitators` in `ModelManager`.

The following sequence diagram shows how the facilitator list command works:

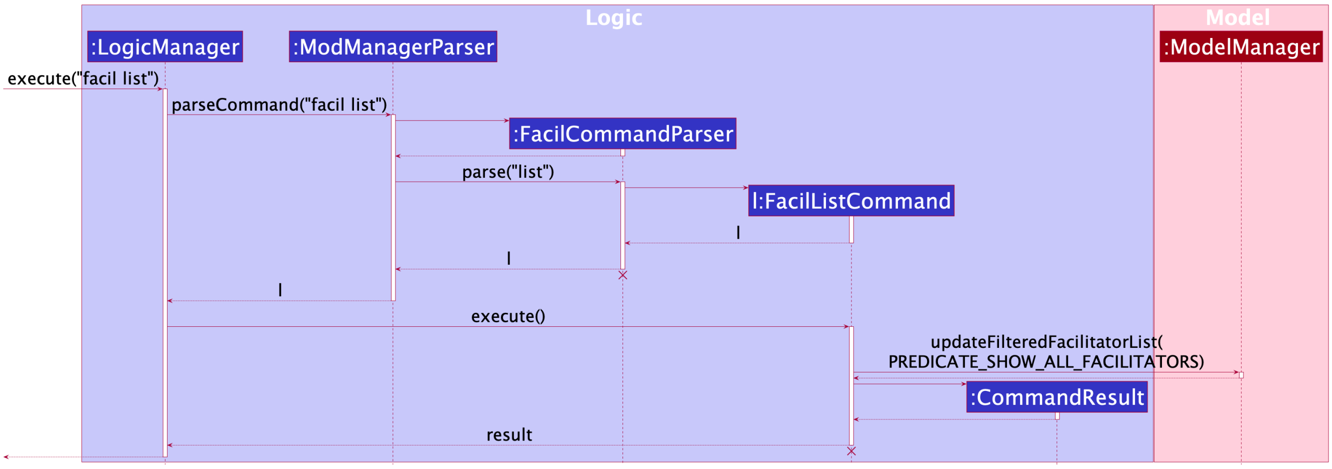


Figure 14. Sequence diagram for `facil list` command

NOTE The lifeline for `FacilCommandParser` and `FacilListCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of the diagram.

The following activity diagram summarizes what happens when a user executes a facilitator list command:

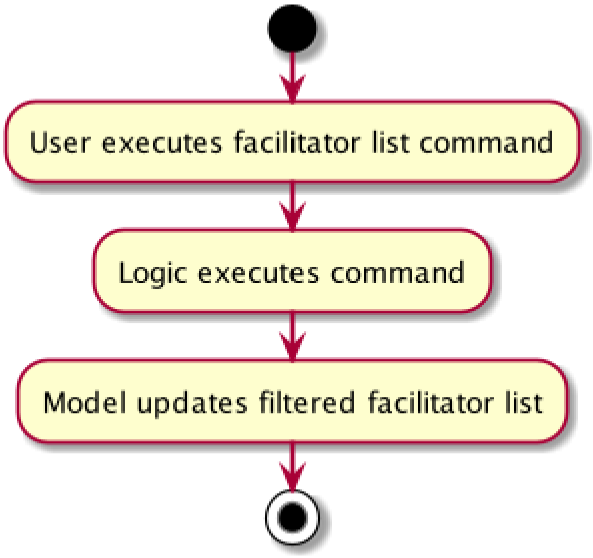

User executes facilitator list command
Logic executes command
Model updates filtered facilitator list

Figure 15. Activity diagram for `facil list` command

Finding facilitators

The find facilitator feature allows users to find a facilitator by name in Mod Manager. This feature is facilitated by `FacilCommandParser`, `FacilFindCommandParser` and `FacilFindCommand`. The operation is exposed in the Model interface as Model#updateFilteredFacilitatorList().

Given below is an example usage scenario and how the facilitator find mechanism behaves at each step:

1. The user executes the facilitator find command and provides the names of the facilitators to search for.
2. FacilFindCommandParser creates a new FacilFindCommand based on the names.
3. LogicManager executes the FacilFindCommand.
4. ModelManager updates the filteredFacilitators in ModelManager.

The following sequence diagram shows how the facilitator find command works:

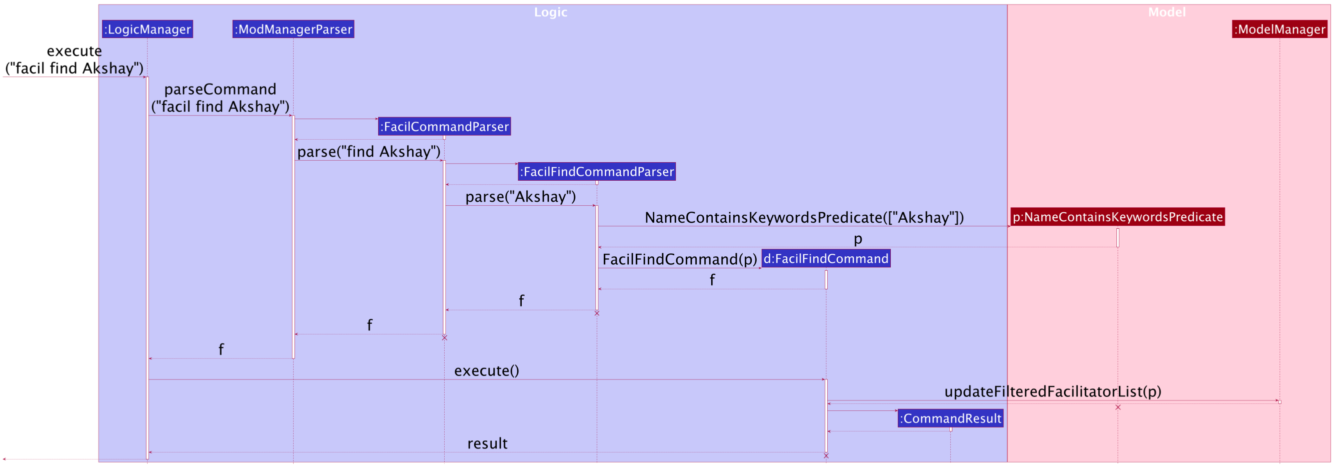


Figure 16. Sequence diagram for facil find command

NOTE The lifeline for FacilCommandParser, FacilFindCommandParser, FacilFindCommand and NameContainsKeyword should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of the diagram.

The following activity diagram summarizes what happens when a user executes a facilitator find command:

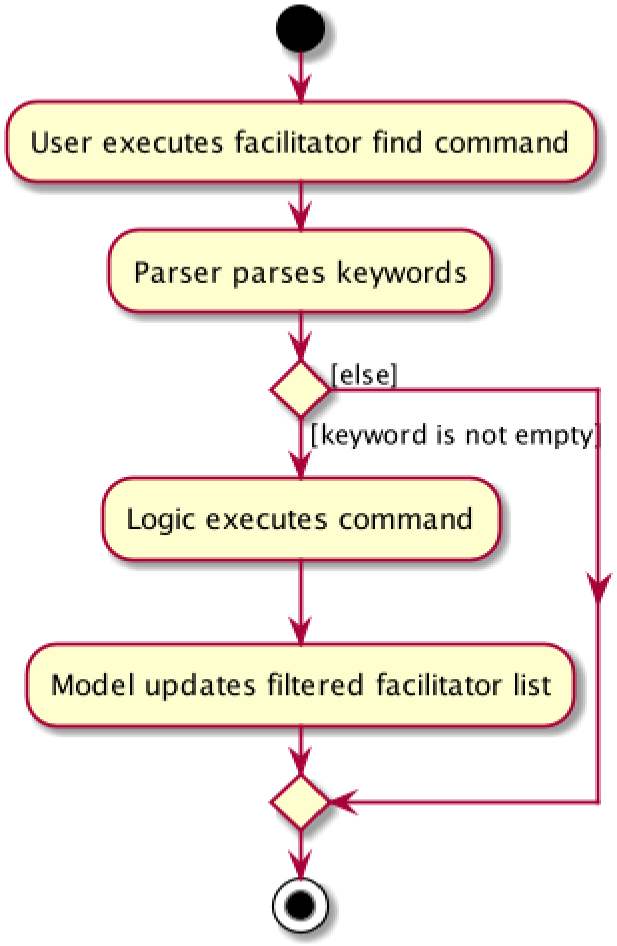

User executes facilitator find command
Parser parses keywords
[else]
[keyword is not empty]
Logic executes command
Model updates filtered facilitator list

Figure 17. Activity diagram for `facil find` command

Editing a facilitator

The edit facilitator feature allows users to edit a facilitator from Mod Manager. This feature is facilitated by `FacilCommandParser`, `FacilEditCommandParser` and `FacilEditCommand`. The operation is exposed in the `Model` interface as `Model#setFacilitator()`.

Given below is an example usage scenario and how the facilitator edit mechanism behaves at each step:

1. The user executes the facilitator edit command and provides the index of the facilitator to be edited and the fields to be edited.
2. `FacilEditCommandParser` creates a new `EditFacilitatorDescriptor` with the fields to be edited.
3. `FacilEditCommandParser` creates a new `FacilEditCommand` based on the index and `EditFacilitatorDescriptor`.
4. `LogicManager` executes the `FacilEditCommand`.
5. `FacilEditCommand` retrieves the facilitator to be edited.
6. `FacilEditCommand` creates a new `Facilitator`.
7. `ModManager` sets the existing facilitator to the new facilitator in the `UniqueFacilitatorList`.
8. `ModelManager` updates the `filteredFacilitators` in `ModelManager`.

The following sequence diagram shows how the facilitator edits command works:

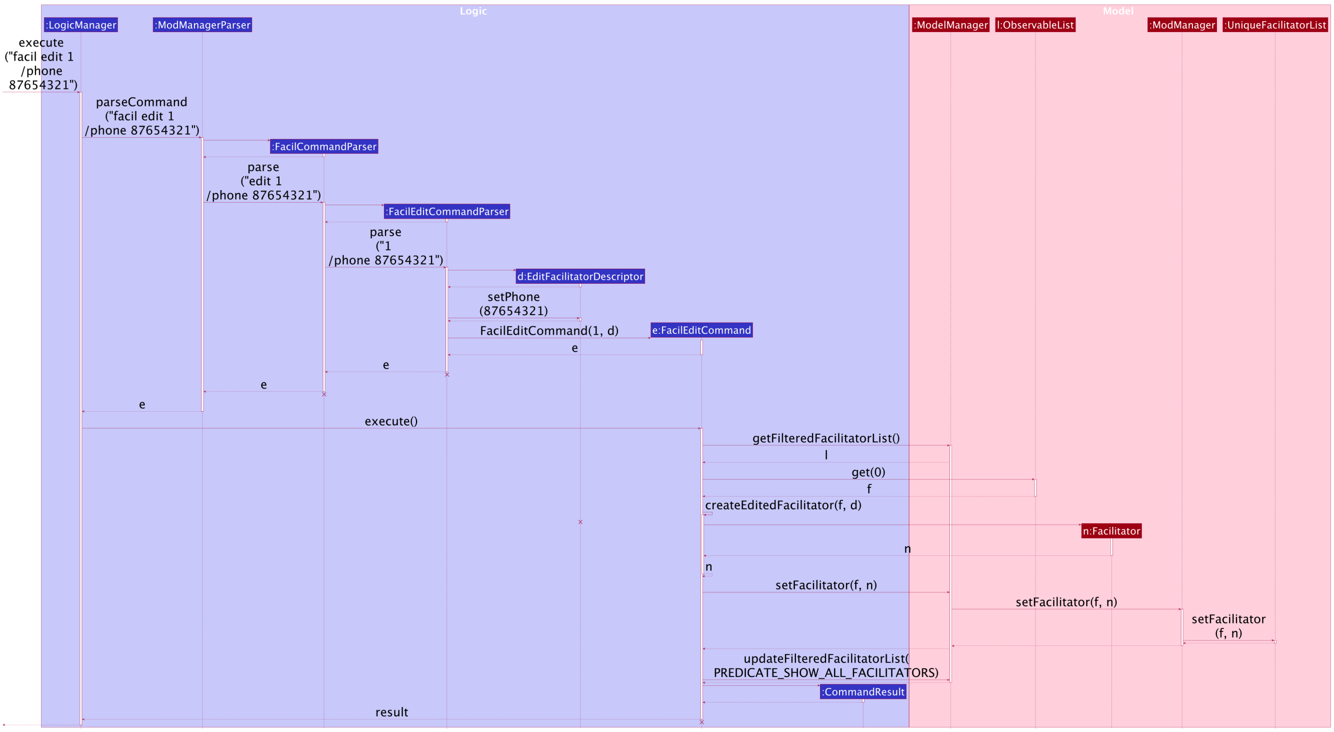


Figure 18. Sequence diagram for `facil edit` command

NOTE The lifeline for `FacilCommandParser`, `FacilEditCommandParser`, `EditFacilitatorDescriptor` and `FacilEditCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of the diagram.

The following activity diagram summarizes what happens when a user executes a facilitator edit command:

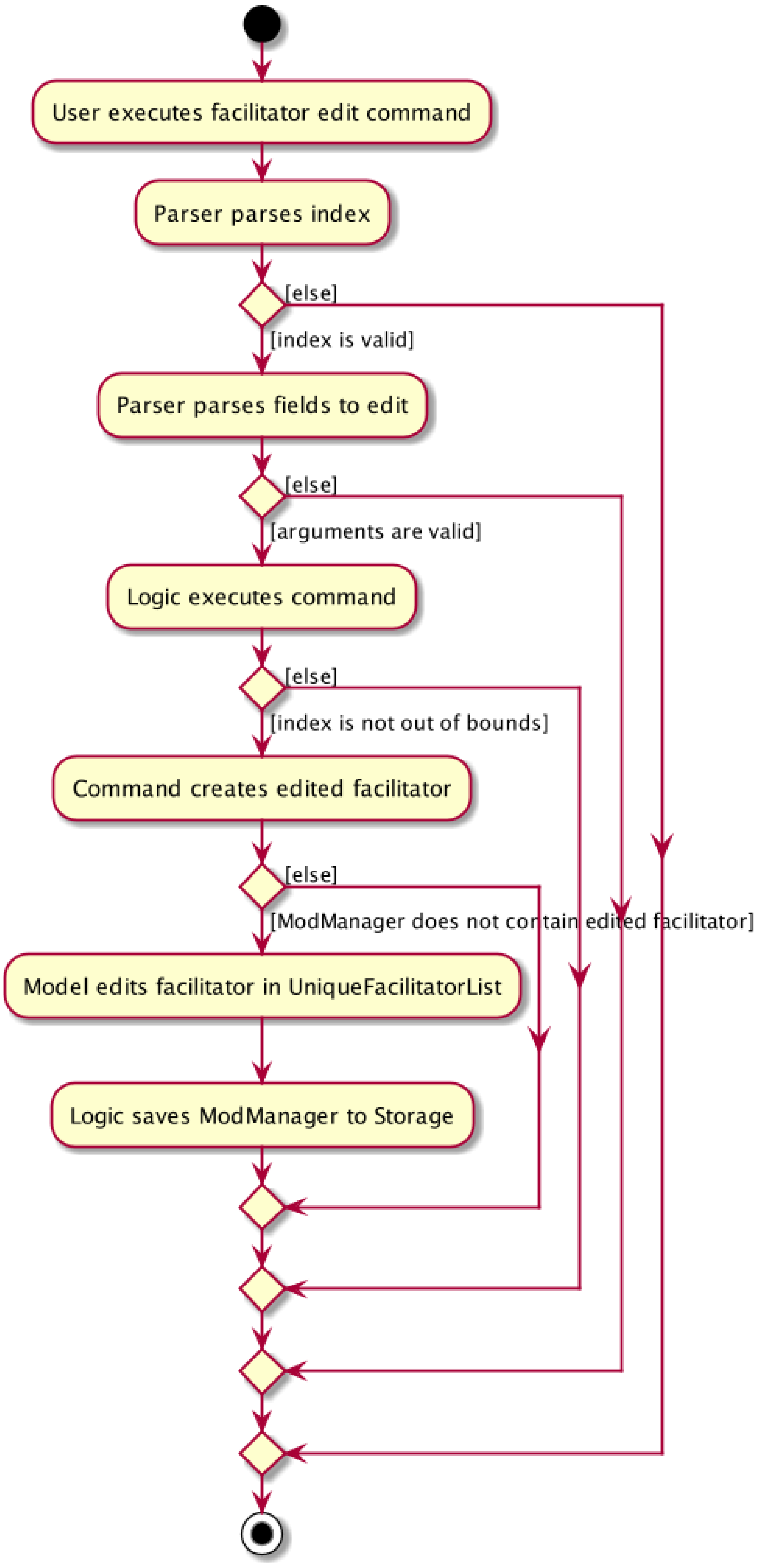

User executes facilitator edit command
Parser parses index
[else]
[index is valid]
Parser parses fields to edit
[else]
[arguments are valid]
Logic executes command
[else]
[index is not out of bounds]
Command creates edited facilitator
[else]
[ModManager does not contain edited facilitator]
Model edits facilitator in UniqueFacilitatorList
Logic saves ModManager to Storage

Figure 19. Activity diagram for `facil edt` command

Deleting a facilitator

The delete facilitator feature allows users to delete a facilitator from Mod Manager. This feature is facilitated by `FacilCommandParser`, `FacilDeleteCommandParser` and `FacilDeleteCommand`. The operation is exposed in the `Model` interface as `Model#deleteFacilitator()`.

Given below is an example usage scenario and how the facilitator delete mechanism behaves at each step:

1. The user executes the facilitator delete command and provides the index of the facilitator to be deleted.
2. `FacilDeleteCommandParser` creates a new `FacilDeleteCommand` based on the index.
3. `LogicManager` executes the `FacilDeleteCommand`.
4. `FacilDeleteCommand` retrieves the facilitator to be deleted.
5. `ModManager` deletes the facilitator from the `UniqueFacilitatorList`.

The following sequence diagram shows how the facilitator delete command works:

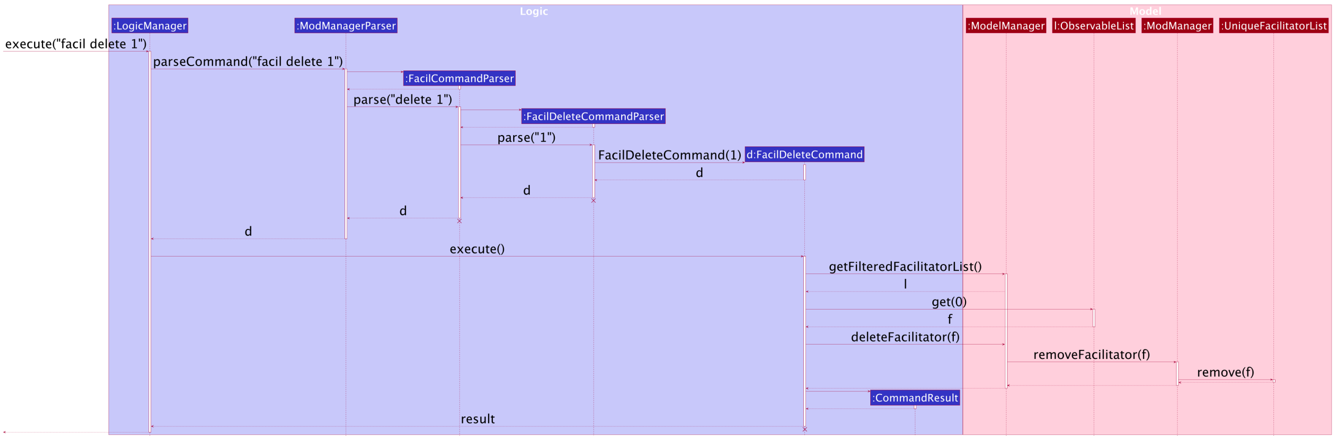


Figure 20. Sequence diagram for `facil delete` command

> **NOTE** The lifeline for `FacilCommandParser`, `FacilDeleteCommandParser` and `FacilDeleteCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of the diagram.

The following activity diagram summarizes what happens when a user executes a facilitator delete command:

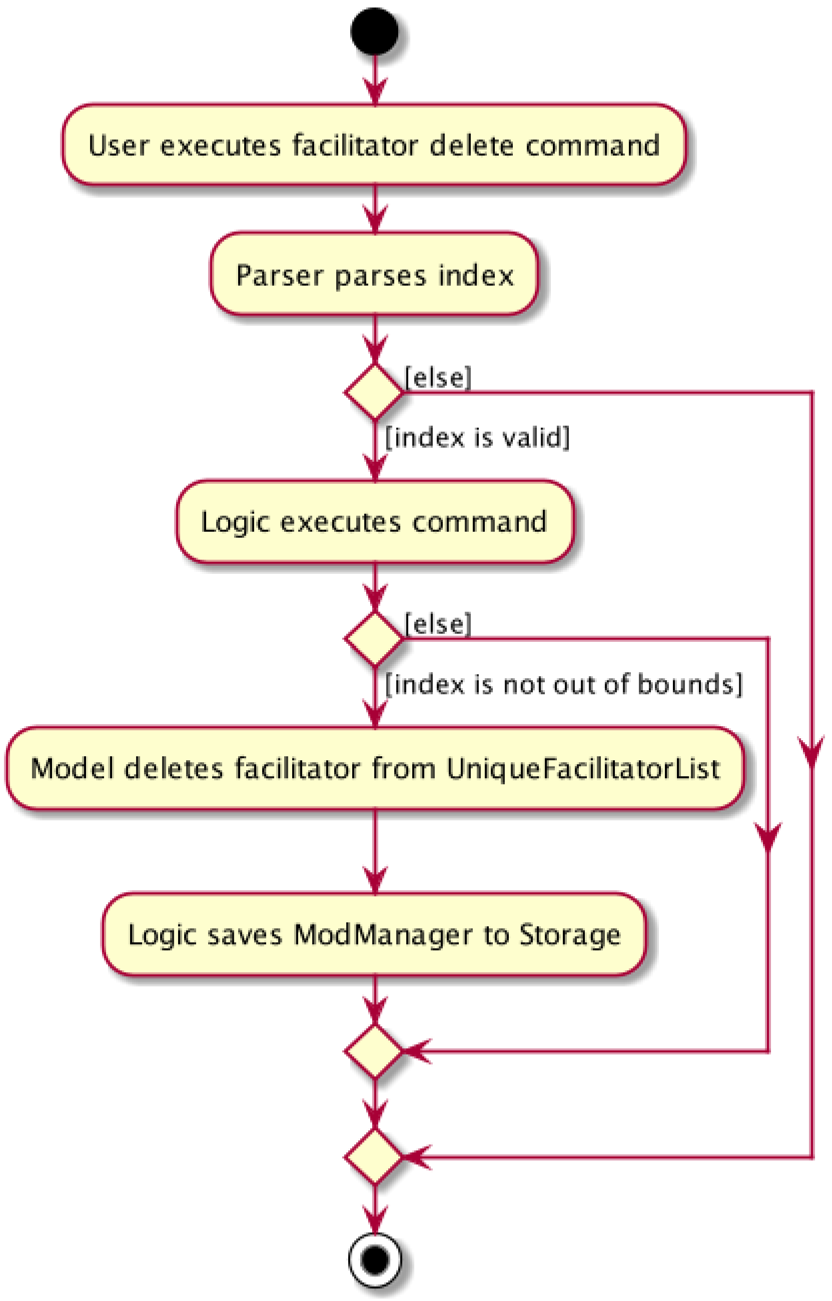

User executes facilitator delete command
Parser parses index
[else]
[index is valid]
Logic executes command
[else]
[index is not out of bounds]
Model deletes facilitator from UniqueFacilitatorList
Logic saves ModManager to Storage

Figure 21. Activity diagram for `facil delete` command

4.2.2. Design Considerations

Aspect: How the facilitator is edited

- **Alternative 1 (current choice):** Create a new facilitator with the edited fields and replace the existing facilitator with the new facilitator.
  - Pros: Preserves the immutability of the `Facilitator` object.
  - Cons: Overhead in creating a new `Facilitator` object for every edit operation.
- **Alternative 2:** Modify the existing facilitator directly.
  - Pros: More convenient and lower overhead to edit a facilitator by setting the relevant fields without creating a new `Facilitator` object.
  - Cons: The `Facilitator` object has to be mutable and may be edited unintentionally.

Aspect: How the facilitator list is stored

- **Alternative 1 (current choice):** Store all facilitators in a single facilitator list.
  - Pros: Will not have to maintain multiple lists. Less memory usage as each facilitator is represented once. Will not have to iterate through multiple lists to find all instances of a particular facilitator when executing facilitator commands.
  - Cons: Have to iterate through the whole list to find facilitators for a particular module when executing module commands.
- **Alternative 2:** Store facilitators for each module in a separate list.
  - Pros: Able to find facilitators for a particular module easily when executing module commands.
  - Cons: May contain duplicates as some facilitators may have multiple module codes. Have to iterate through multiple lists when executing facilitator commands.

4.3. Task feature

The task feature manages the tasks in Mod Manager and is represented by the `Task` abstract class with implementing class `ScheduledTask` for a `Task` with a time period and `NonScheduledTask` for a `Task` with no specified time period. A task has a `Description`, an optional `TaskDateTime`, and one and only one `ModuleCode`. A `Module` with that `ModuleCode` of the task should exist in Mod Manager.

It supports the following operations:

- `add` - Adds a task to a `Module` in Mod Manager.
- `list` - Shows a list of all tasks across all `Module` s in Mod Manager.
- `find` - Finds a task in Mod Manager by its description.
- `upcoming` - Finds upcoming tasks (for tasks with a specified time period) in Mod Manager.
- `search`- Searches for tasks that occur on your specified date, month, or year in Mod Manager.
- `edit` - Edits the information of a task in Mod Manager.
- `delete` - Deletes a task from the `Module` and Mod Manager.

4.3.1. Implementation

Adding a task

The add task feature allows users to add a task to Mod Manager. This feature is taskitated by `FacilCommandParser`, `FacilAddCommandParser` and `FacilAddCommand`. The operation is exposed in the `Model` interface as `Model#addFacilitator()`.

Given below is an example usage scenario and how the taskitator add mechanism behaves at each step:

1. The user executes the taskitator add command and provides the name, phone, email, office and module code of the taskitator to be added.

2. `FacilitatorAddCommandParser` creates a new `Facilitator` based on the name, phone, email, office and module code.
3. `FacilitatorAddCommandParser` creates a new `FacilitatorAddCommand` based on the taskitator.
4. `LogicManager` executes the `FacilitatorAddCommand`.
5. `ModManager` adds the taskitator to the `UniqueFacilitatorList`.
6. `ModelManager` updates the `filteredFacilitators` in `ModelManager`.

The following sequence diagram shows how the taskitator add command works:

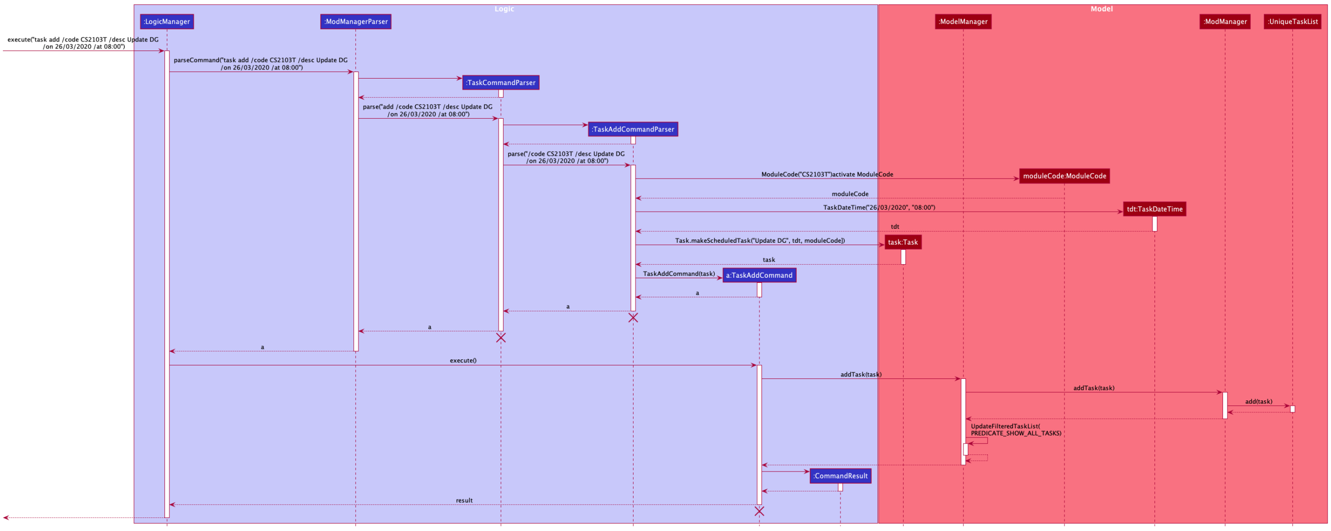

Logic
Model
:LogicManager
:ModManagerParser
:ModelManager
:ModManager
:UniqueTaskList
execute("task add /code CS2103T /desc Update DG /on 26/03/2020 /at 08:00")
parseCommand("task add /code CS2103T /desc Update DG /on 26/03/2020 /at 08:00")
:TaskCommandParser
parse("add /code CS2103T /desc Update DG /on 26/03/2020 /at 08:00")
:TaskAddCommandParser
parse("/code CS2103T /desc Update DG /on 26/03/2020 /at 08:00")
ModuleCode("CS2103T")activate ModuleCode
moduleCode:ModuleCode
moduleCode
TaskDateTime("26/03/2020", "08:00")
tdt:TaskDateTime
tdt
Task.makeScheduledTask("Update DG", tdt, moduleCode])
task:Task
task
TaskAddCommand(task)
a:TaskAddCommand
a
a
a
a
execute()
addTask(task)
addTask(task)
add(task)
UpdateFilteredTaskList(
PREDICATE_SHOW_ALL_TASKS)
:CommandResult
result

Figure 22. Sequence diagram for `facil add` command

NOTE The lifeline for `FacilitatorCommandParser`, `FacilitatorAddCommandParser` and `FacilitatorAddCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

The following activity diagram summarizes what happens when a user executes a facilitator add command:

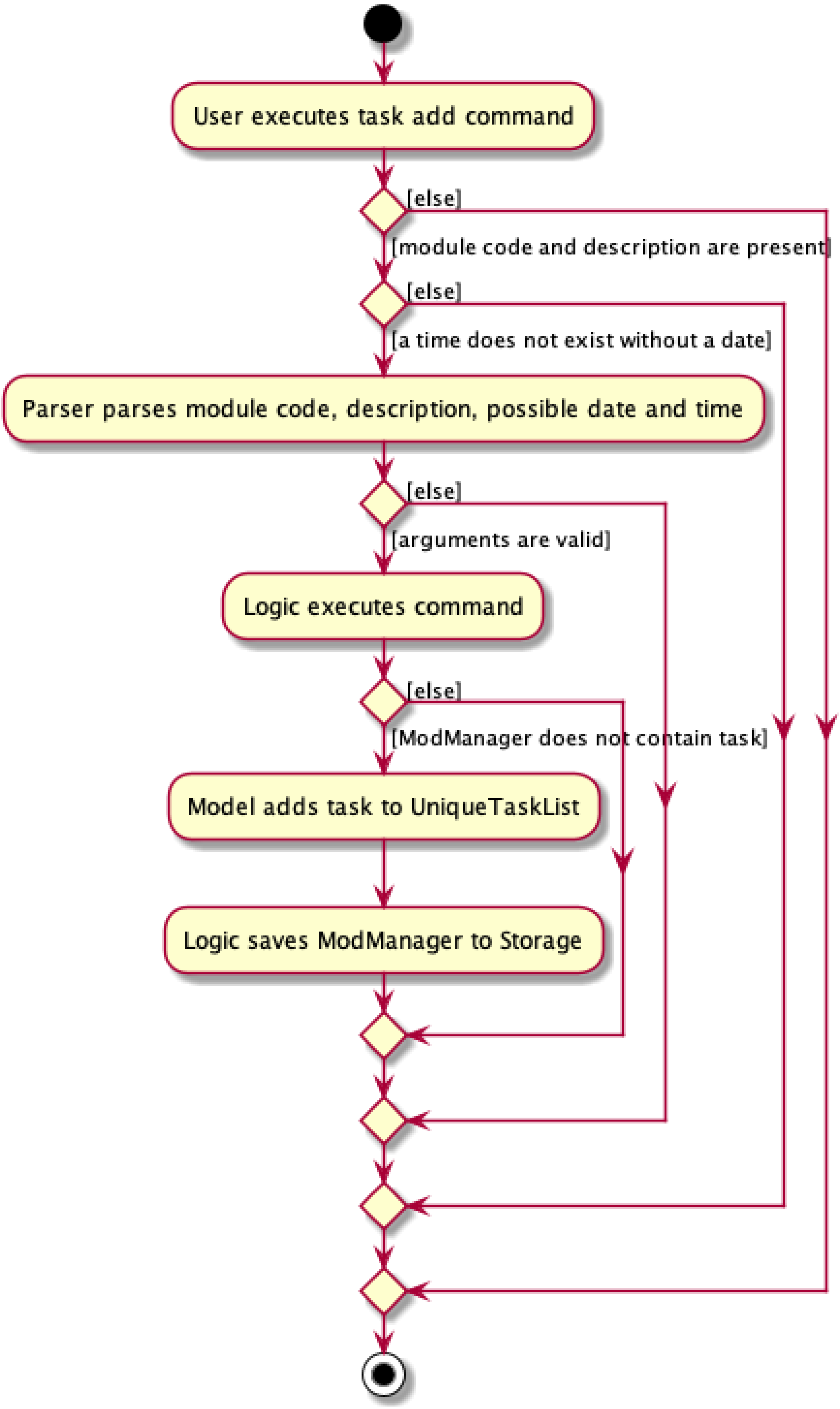

User executes task add command
[else]
[module code and description are present]
[else]
[a time does not exist without a date]
Parser parses module code, description, possible date and time
[else]
[arguments are valid]
Logic executes command
[else]
[ModManager does not contain task]
Model adds task to UniqueTaskList
Logic saves ModManager to Storage

Figure 23. Activity diagram for `facil add` command

Listing all facilitators

The list facilitator feature allows users to list all facilitators in Mod Manager. This feature is facilitated by `FacilCommandParser` and `FacilListCommand`. The operation is exposed in the `Model` interface as `Model#updateFilteredFacilitatorList()`.

Given below is an example usage scenario and how the facilitator list mechanism behaves at each step:

1. The user executes the facilitator list command.
2. `FacilCommandParser` creates a new `FacilListCommand`.

3. `LogicManager` executes the `FacilListCommand`.
4. `ModelManager` updates the `filteredFacilitators` in `ModelManager`.

The following sequence diagram shows how the facilitator list command works:

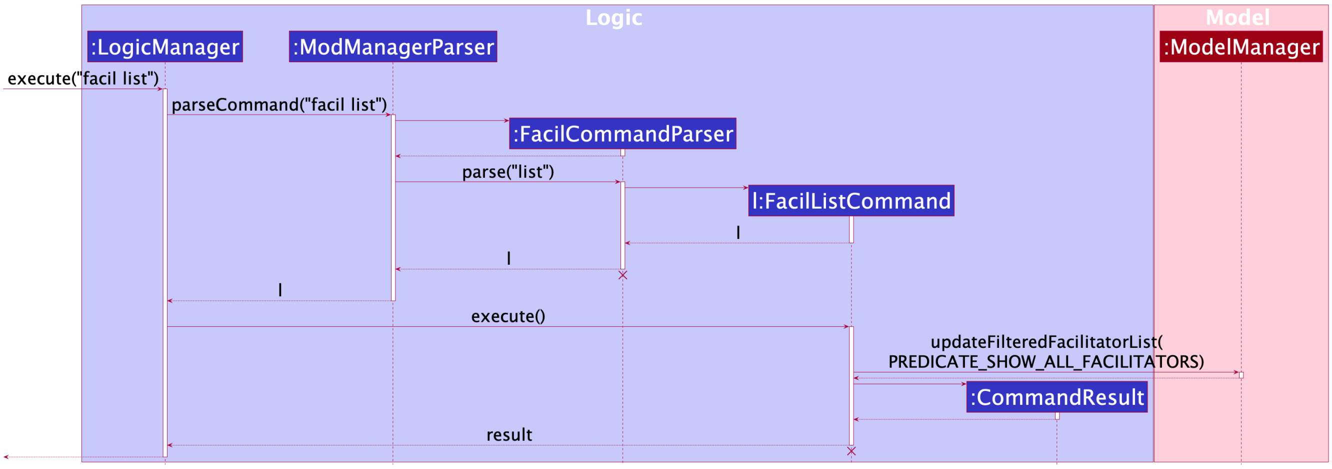


Figure 24. Sequence diagram for `facil list` command

NOTE The lifeline for `FacilCommandParser` and `FacilListCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of the diagram.

The following activity diagram summarizes what happens when a user executes a facilitator list command:

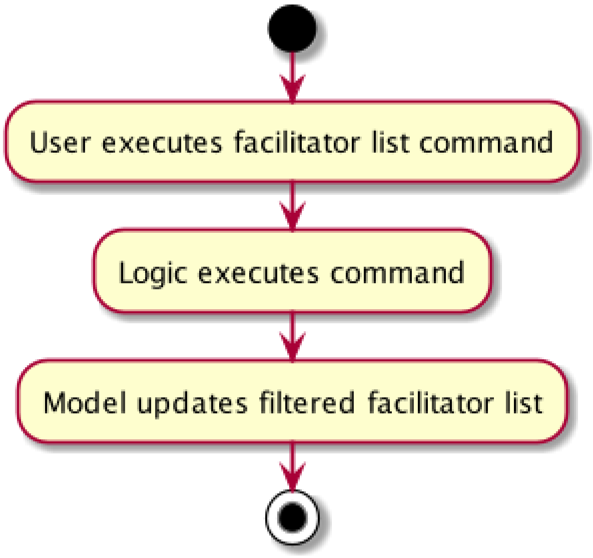

User executes facilitator list command
Logic executes command
Model updates filtered facilitator list

Figure 25. Activity diagram for `facil list` command

Finding facilitators

The find facilitator feature allows users to find a facilitator by name in Mod Manager. This feature is facilitated by `FacilCommandParser`, `FacilFindCommandParser` and `FacilFindCommand`. The operation is exposed in the `Model` interface as `Model#updateFilteredFacilitatorList()`.

Given below is an example usage scenario and how the facilitator find mechanism behaves at each step:

1. The user executes the facilitator find command and provides the names of the facilitators to search for.
2. `FacilFindCommandParser` creates a new `FacilFindCommand` based on the names.
3. `LogicManager` executes the `FacilFindCommand`.
4. `ModelManager` updates the `filteredFacilitators` in `ModelManager`.

The following sequence diagram shows how the facilitator find command works:

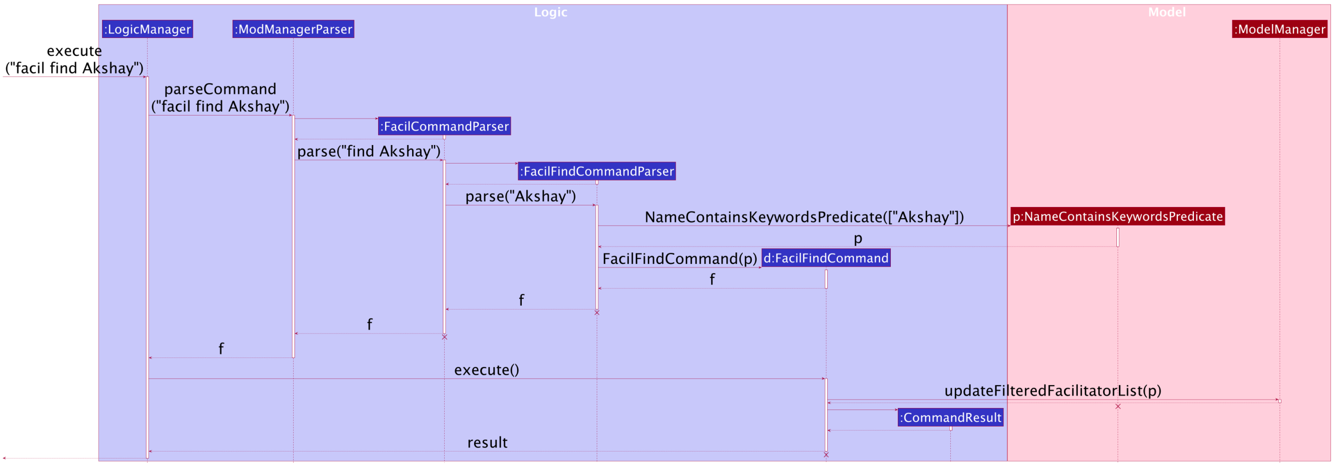


Figure 26. Sequence diagram for `facil find` command

NOTE The lifeline for `FacilCommandParser`, `FacilFindCommandParser`, `FacilFindCommand` and `NameContainsKeyword` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of the diagram.

The following activity diagram summarizes what happens when a user executes a facilitator find command:

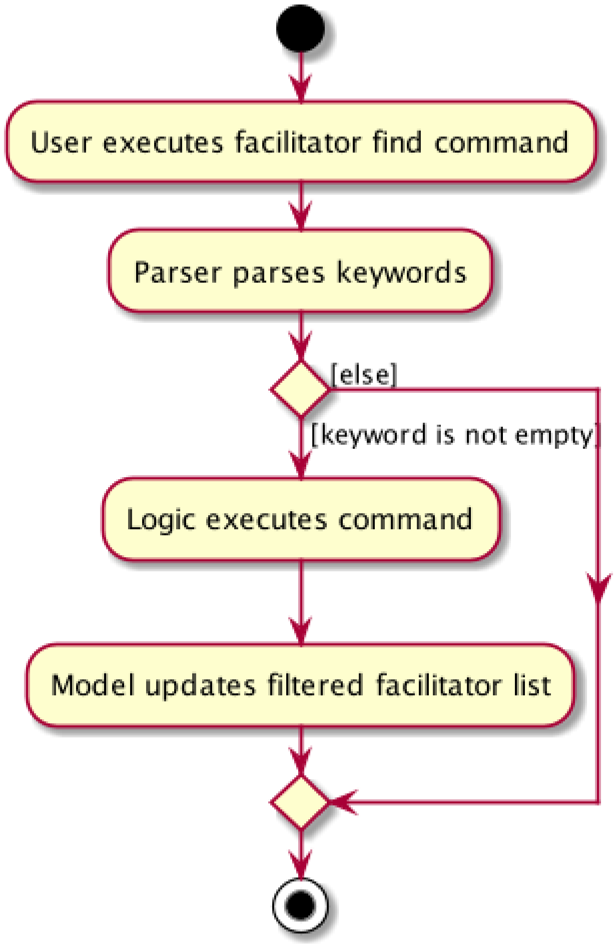

User executes facilitator find command
Parser parses keywords
[else]
[keyword is not empty]
Logic executes command
Model updates filtered facilitator list

Figure 27. Activity diagram for `facil find` command

Editing a facilitator

The edit facilitator feature allows users to edit a facilitator from Mod Manager. This feature is facilitated by `FacilCommandParser`, `FacilEditCommandParser` and `FacilEditCommand`. The operation is exposed in the `Model` interface as `Model#setFacilitator()`.

Given below is an example usage scenario and how the facilitator edit mechanism behaves at each step:

1. The user executes the facilitator edit command and provides the index of the facilitator to be edited and the fields to be edited.
2. `FacilEditCommandParser` creates a new `EditFacilitatorDescriptor` with the fields to be edited.
3. `FacilEditCommandParser` creates a new `FacilEditCommand` based on the index and `EditFacilitatorDescriptor`.
4. `LogicManager` executes the `FacilEditCommand`.
5. `FacilEditCommand` retrieves the facilitator to be edited.
6. `FacilEditCommand` creates a new `Facilitator`.
7. `ModManager` sets the existing facilitator to the new facilitator in the `UniqueFacilitatorList`.
8. `ModelManager` updates the `filteredFacilitators` in `ModelManager`.

The following sequence diagram shows how the facilitator edits command works:

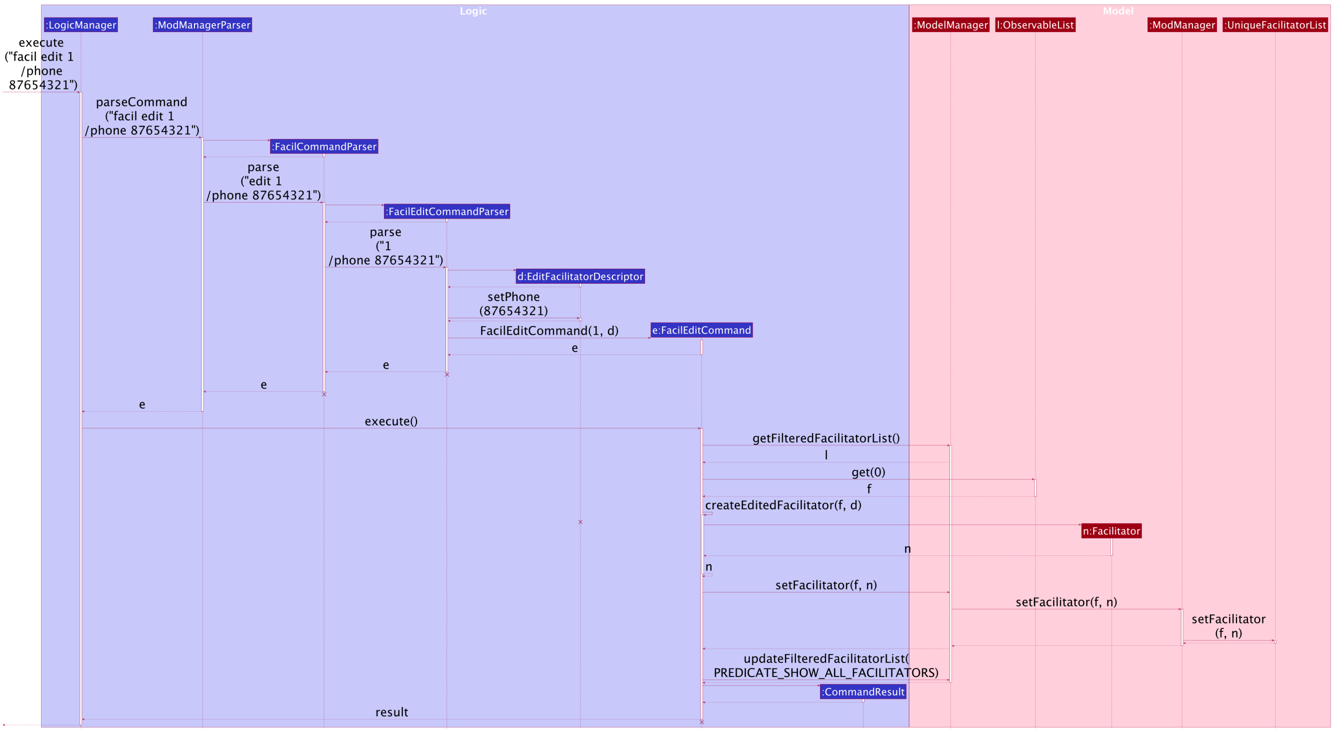


Figure 28. Sequence diagram for `facil edit` command

NOTE The lifeline for `FacilCommandParser`, `FacilEditCommandParser`, `EditFacilitatorDescriptor` and `FacilEditCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of the diagram.

The following activity diagram summarizes what happens when a user executes a facilitator edit command:

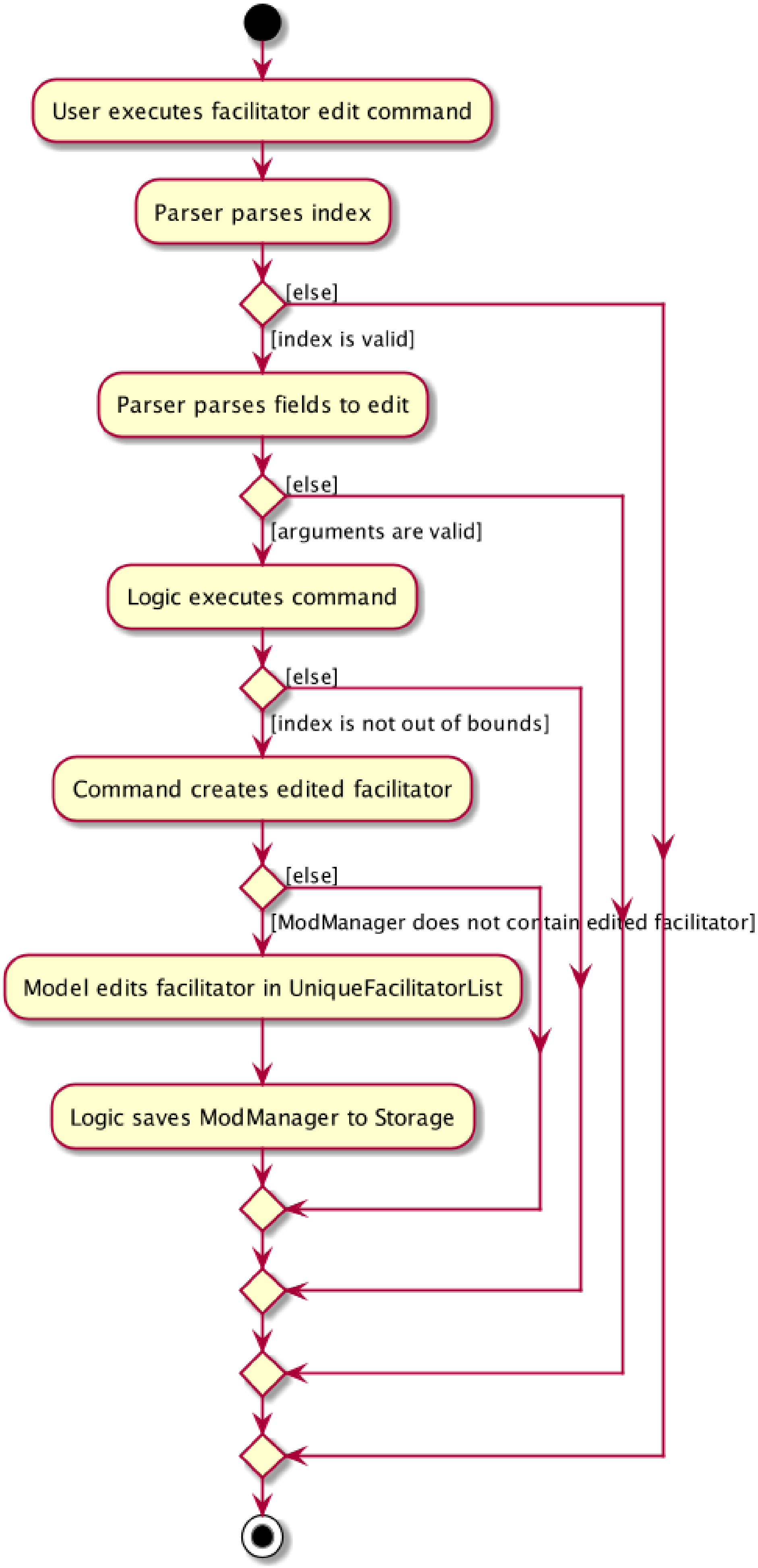

User executes facilitator edit command
Parser parses index
[else]
[index is valid]
Parser parses fields to edit
[else]
[arguments are valid]
Logic executes command
[else]
[index is not out of bounds]
Command creates edited facilitator
[else]
[ModManager does not contain edited facilitator]
Model edits facilitator in UniqueFacilitatorList
Logic saves ModManager to Storage

Figure 29. Activity diagram for `facil edt` command

Deleting a facilitator

The delete facilitator feature allows users to delete a facilitator from Mod Manager. This feature is facilitated by `FacilCommandParser`, `FacilDeleteCommandParser` and `FacilDeleteCommand`. The operation is exposed in the `Model` interface as `Model#deleteFacilitator()`.

Given below is an example usage scenario and how the facilitator delete mechanism behaves at each step:

1. The user executes the facilitator delete command and provides the index of the facilitator to be deleted.
2. `FacilDeleteCommandParser` creates a new `FacilDeleteCommand` based on the index.
3. `LogicManager` executes the `FacilDeleteCommand`.
4. `FacilDeleteCommand` retrieves the facilitator to be deleted.
5. `ModManager` deletes the facilitator from the `UniqueFacilitatorList`.

The following sequence diagram shows how the facilitator delete command works:

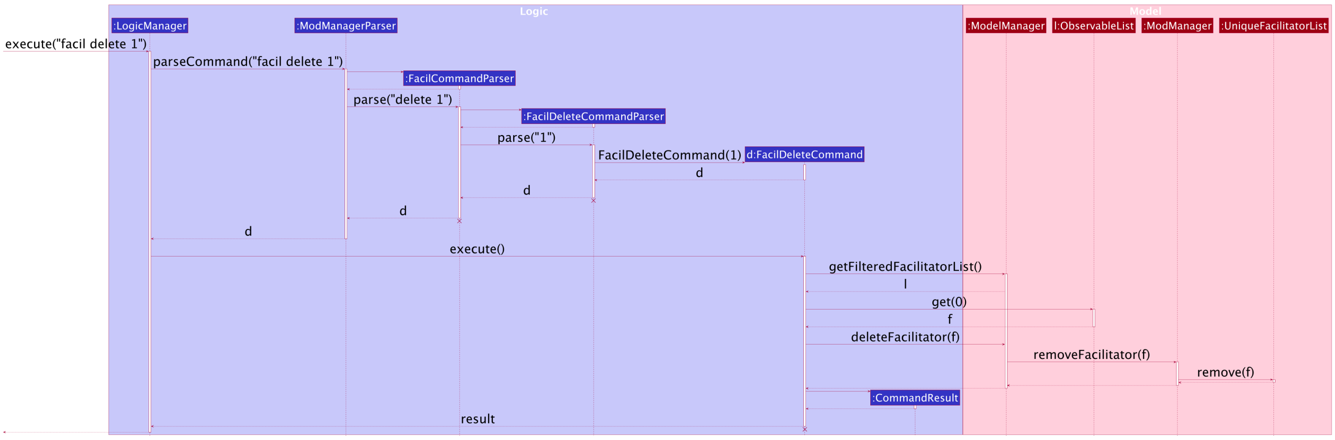


Figure 30. Sequence diagram for `facil delete` command

NOTE The lifeline for `FacilCommandParser`, `FacilDeleteCommandParser` and `FacilDeleteCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of the diagram.

The following activity diagram summarizes what happens when a user executes a facilitator delete command:

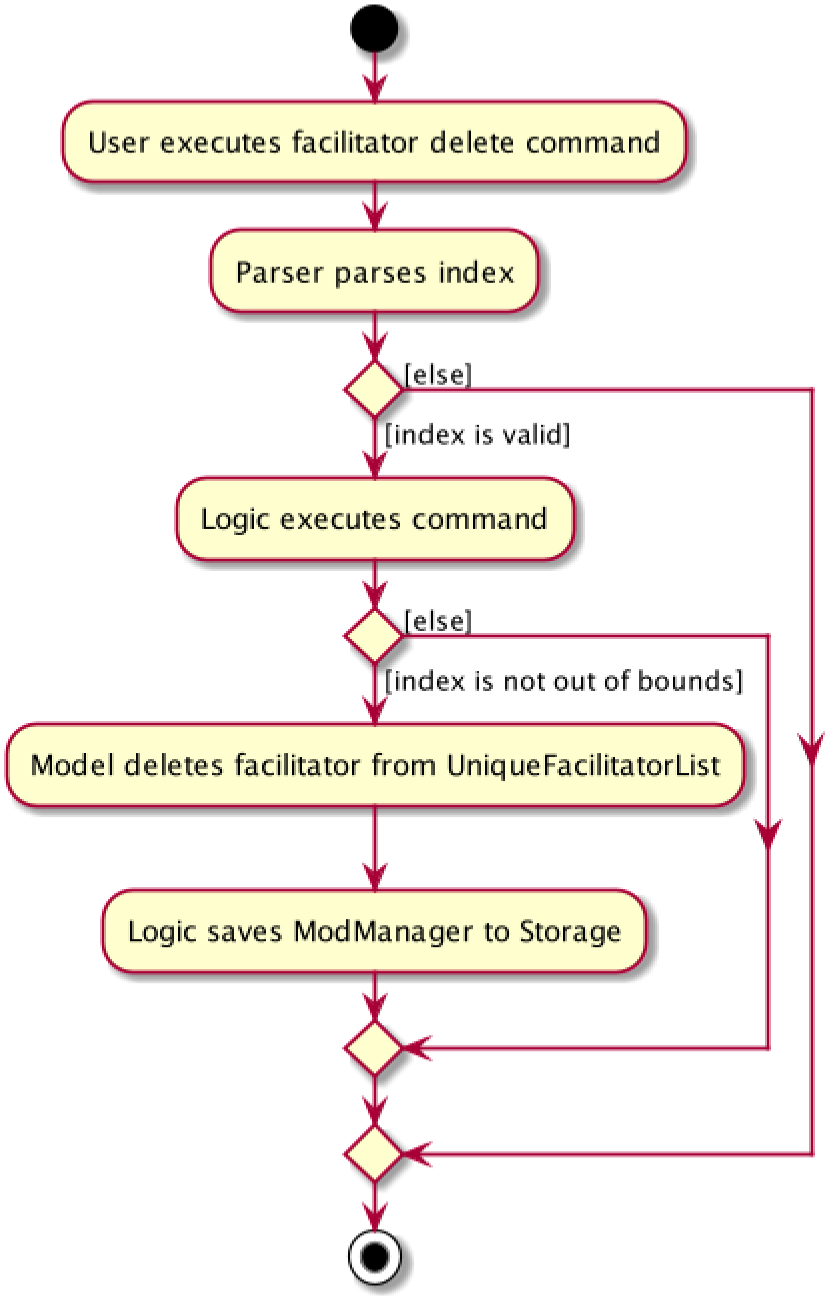

User executes facilitator delete command
Parser parses index
[else]
[index is valid]
Logic executes command
[else]
[index is not out of bounds]
Model deletes facilitator from UniqueFacilitatorList
Logic saves ModManager to Storage

Figure 31. Activity diagram for `facil delete` command

4.3.2. Design considerations

Aspect: How the optional attribute of `TaskDateTime` is managed

- **Alternative 1 (current choice):** Implement `Task` as an abstract class for Mod Manager. A task with a specified time period will be created as a `ScheduledTask`, while a task with no time period specified will be created as a `NonScheduledTask`, with both `ScheduledTask` and `NonScheduledTask` are concrete subclasses of `Task`.
  - Pros: Utilises Object-Oriented Programming. Easy to implement **`search`** functionality, which we need to search for tasks that occur on a specified date, month, or year, and **`upcoming`** functionality, which we need to find the upcoming tasks in Mod Manager. For these two features, we only need to work on `ScheduledTask` instances, which reduces the burden of checking for `null` `TaskDateTime` instances as the second approach below.

    - Cons: More difficulty in implementation due to time constraints. Moreover, command **`edit`** that allows us to edits the information of the task will be troublesome, when a user decides to add a time period to a `NonScheduledTask`. In this case, we have to re-create a new `ScheduledTask` with the same description and its time provided. If we need to maintain a `List<ScheduledTask>` or `List<Task>` somewhere in the code, for example, in our `Module` instance, we also have to update the list contents in our `Module` s too. This requires the association between `Module` and `Task` to be bi-directional, which increases content and data coupling and make it harder for us to maintain and conduct tests. There is also extra overhead time communicating and collaborating with another member in our team who is doing the `Module` component, Because of these challenges, we decide to weaken the association between `Task` and `Module`, which is elaborated in our next aspect.

- **Alternative 2:** Implement `Task` as a concrete class in Mod Manager. `Task` s without a specified time period will have its `TaskDateTime` set to `null`, while `Task` s with a given time period will be assign a `TaskDateTime` attribute, which is a wrapper class for Java's `LocalDateTime`.
    - Pros: Easier to implement, as we only need to create one class `Task`.
    - Cons: We must handle `null` cases every time we query something about the time of a `Task`. For example, it's more challenging to implement the **`search`** and **`upcoming`** command, since we have to handle the cases when the `TaskDateTime` instance is `null`. It's very complex to implement the method `compareTo` of `Comparable` interface for `Task` to compare the time between tasks, when one, or both of the `TaskDateTime` attributes can be `null`.

Aspect: The association between `Module` and `Task`

- **Alternative 1 (current choice):** Aggregation: Each `Task` has an unique `ModuleCode` tag, which uniquely identifies which `Module` the task belongs to. This is a aggregation relationship, which is weaker than composition in our second approach. image::ModuleTaskAggregationDiagram.png[]
    - Pros: Easier to implementation, and weak coupling with `Module` implementation. The `Module` need not to be aware that there are a list of `Task` s for it.
    - Cons: The association between `Module` and `Task` cannot be extensive and fully descriptive as in our second approach, but this is a trade-off given the time constraints.
- **Alternative 2:** Composition: each `Module` has a list of `Task` s corresponding to it. If the `Module` is deleted, all of the related `Task` s for the `Module` will also be removed. image::ModuleTaskCompositionDiagram.png[]
    - Pros: This design choice better simulates the real-life interactions between `Module` and `Task`. For example, if we drop a `Module` in NUS, we will also drop all the `Task` s related to the `Module`, such as assignments, homework, term tests, and exams.
    - Cons: Difficulty in implementation due to time constraints, as well as strong content and data coupling. More overhead in communicating and collaborating with the team member responsible for the `Module` component, as mentioned above.

4.4. Calendar Feature

The calendar feature manages the calendar in Mod Manager and is represented by the Calendar class. A calendar has a LocalDate.

It supports the following operations:

- view - Views the schedules and tasks in a whole week in Mod Manager.
- find - Finds empty slots in a week from current day to end of the week in Mod Manager.

4.4.1. Implementation Details

Viewing the calendar

The view calendar feature allows users to view the calendar for a week in Mod Manager. This feature is facilitated by `CalCommandParser`, `CalViewCommandParser` and `CalViewCommand`. The calendar is exposed in the `Model` interface in `Module#updateCalendar()` and it is retrieved in `MainWindow` to show the timeline for the specified week to users.

Given below is an example usage scenario and how the calendar view mechanism behaves at each step:

1. The user executes the calendar view command and provides which week to be viewed. The week to be viewed can be this or next week.
2. `CalViewCommandParser` creates a new `Calendar` based on the specified week.
3. `CalViewCommandParser` creates a new `CalViewCommand` based on the `Calendar`.
4. `LogicManager` executes the `CalViewCommand`.
5. `ModelManager` updates the calendar in `ModelManager`.
6. `MainWindow` retrieves the calendar from `LogicManager` which retrieves from `ModelManager`.
7. `MainWindow` shows the calendar.

The following sequence diagram shows how the calendar view command works:

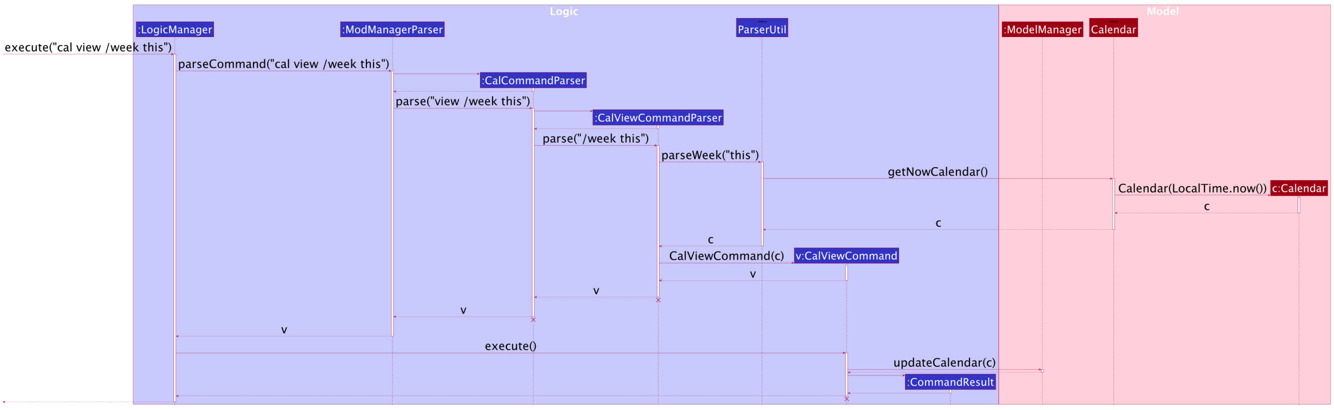


Figure 32. Sequence diagram for `cal view` command

NOTE The lifeline for `CalCommandParser`, `CalViewCommandParser` and `CalViewCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of the diagram.

The following activity diagram summarizes what happens when a user executes a calendar view command:

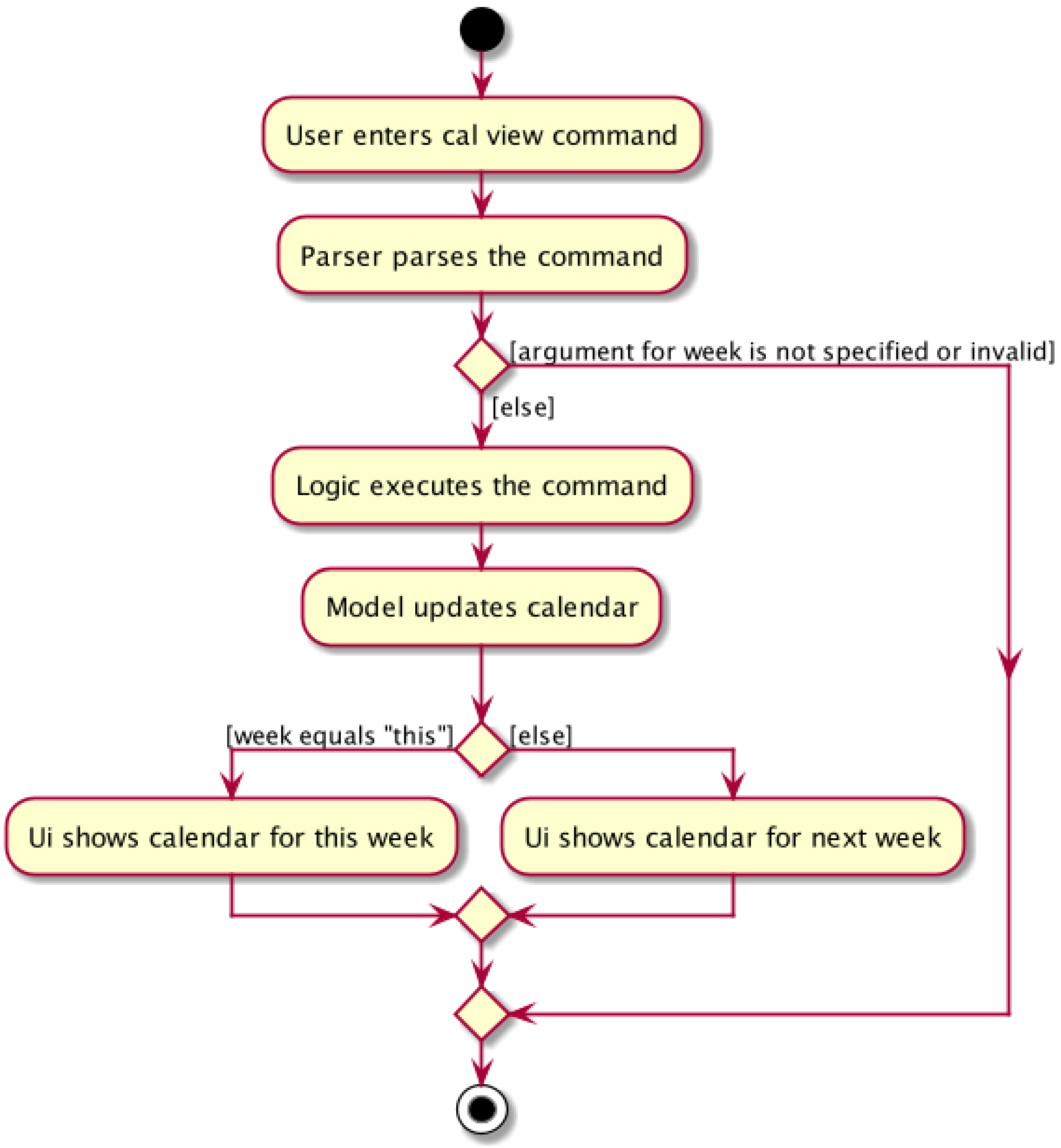

User enters cal view command
Parser parses the command
[argument for week is not specified or invalid]
[else]
Logic executes the command
Model updates calendar
[week equals "this"]
[else]
Ui shows calendar for this week
Ui shows calendar for next week

Figure 33. Activity diagram for 'cal view command'

Finding empty slots in calendar

The find empty in calendar feature allows users to know the empty slots they have in the calendar from the current day to the end of the week in Mod Manager. This feature is facilitated by CalCommandParser, CalFindCommandParser and CalFindCommand.

Given below is an example usage scenario and how the calendar find mechanism behaves at each step:

1. The user executes the calendar find command.
2. CalFindCommandParser creates a new CalFindCommand.
3. LogicManager executes the CalFindCommand.

The following sequence diagram shows how the calendar find command works:

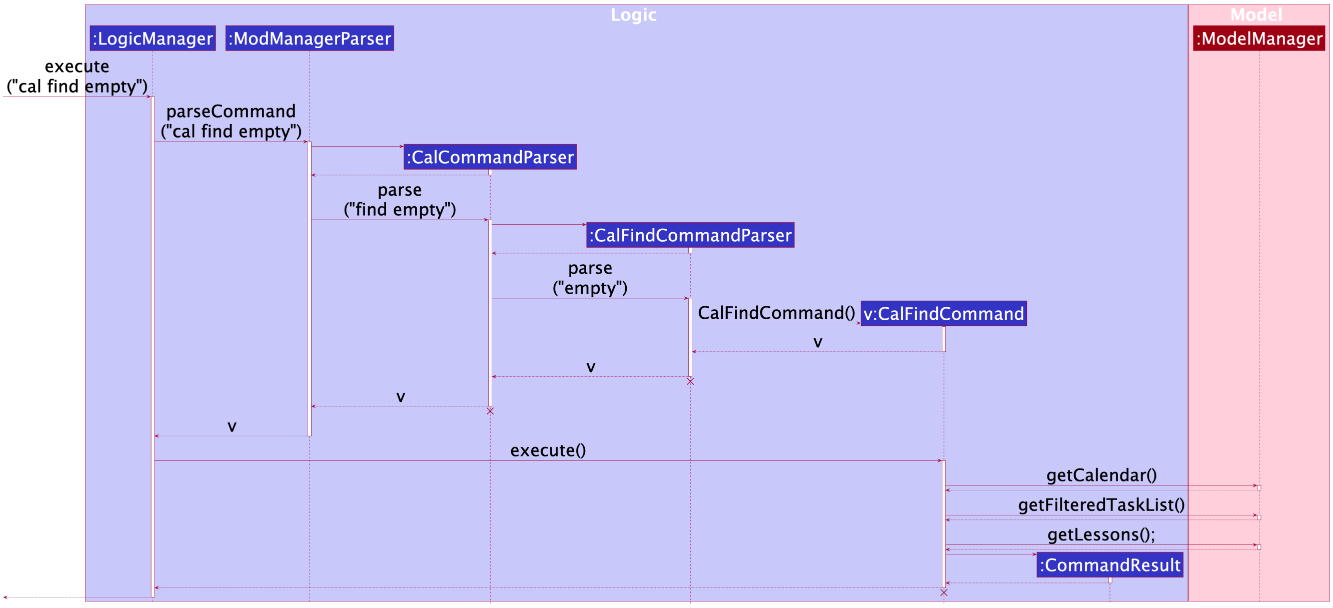


Figure 34. Sequence diagram for `cal find` command

NOTE The lifeline for CalCommandParser, CalFindCommandParser and CalFindCommand should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of the diagram.

The following activity diagram summarizes what happens when a user executes a calendar find command:

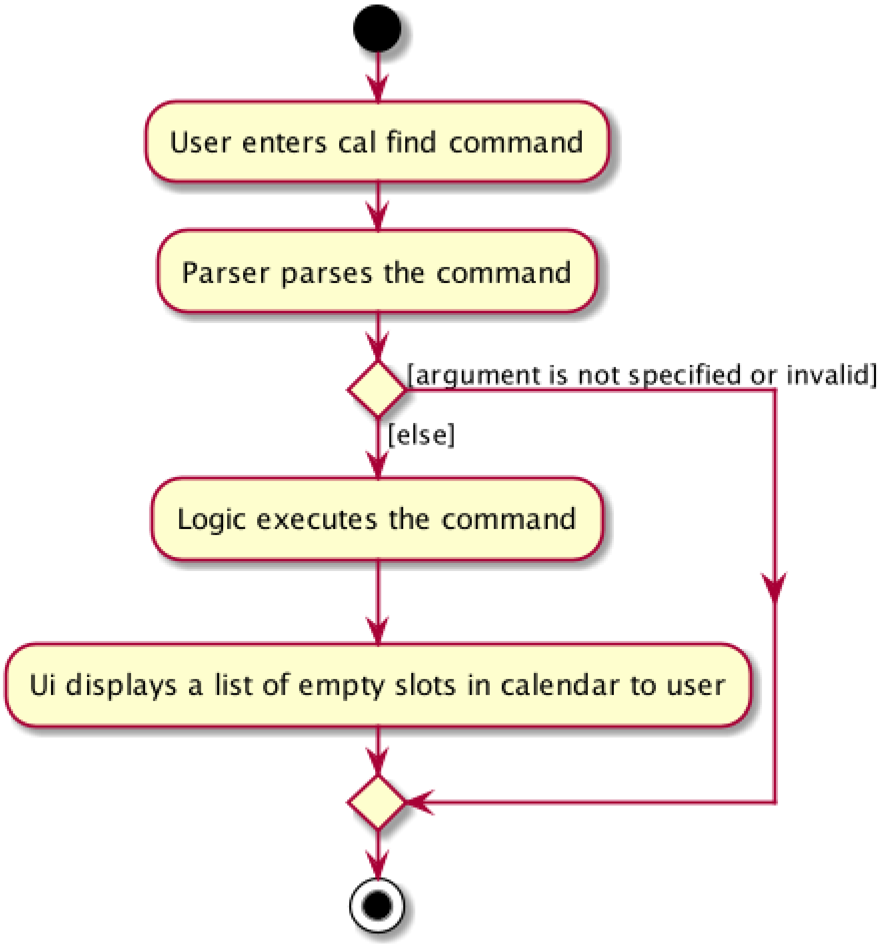
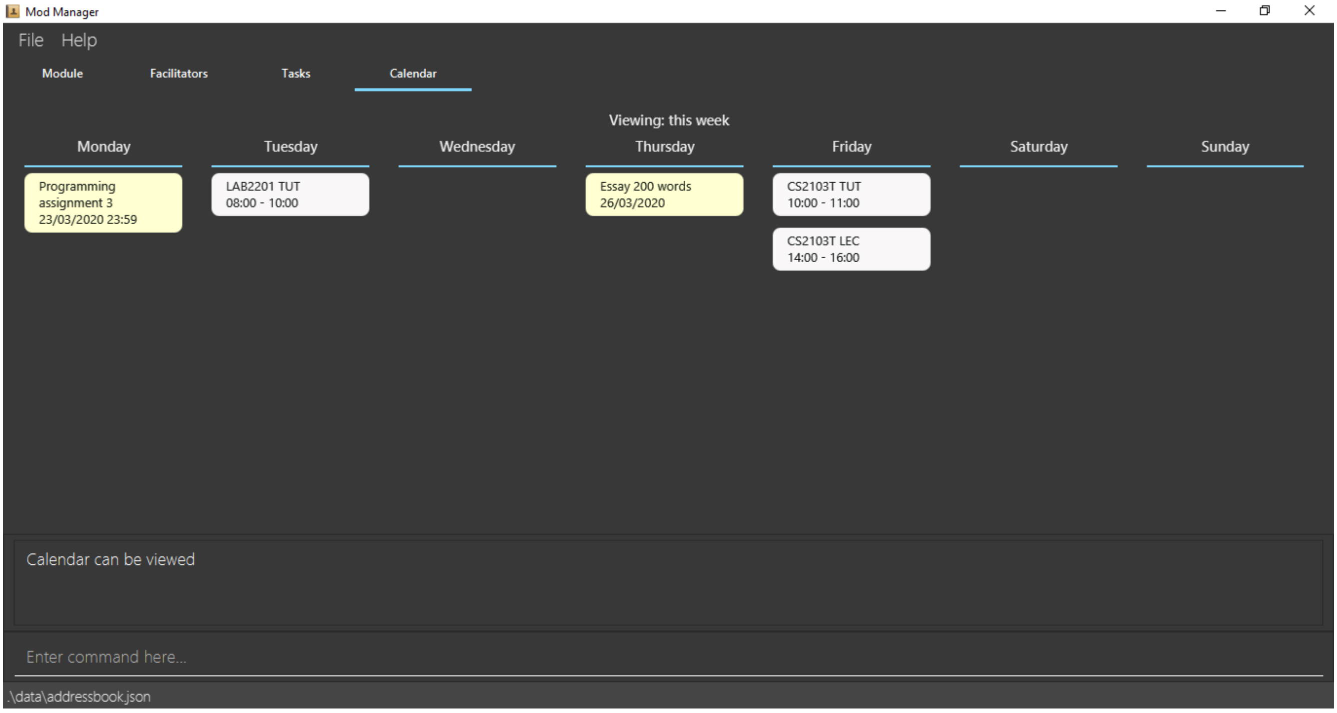

User enters cal find command
Parser parses the command
[argument is not specified or invalid]
[else]
Logic executes the command
Ui displays a list of empty slots in calendar to user

Figure 35. Activity diagram for `cal find command`

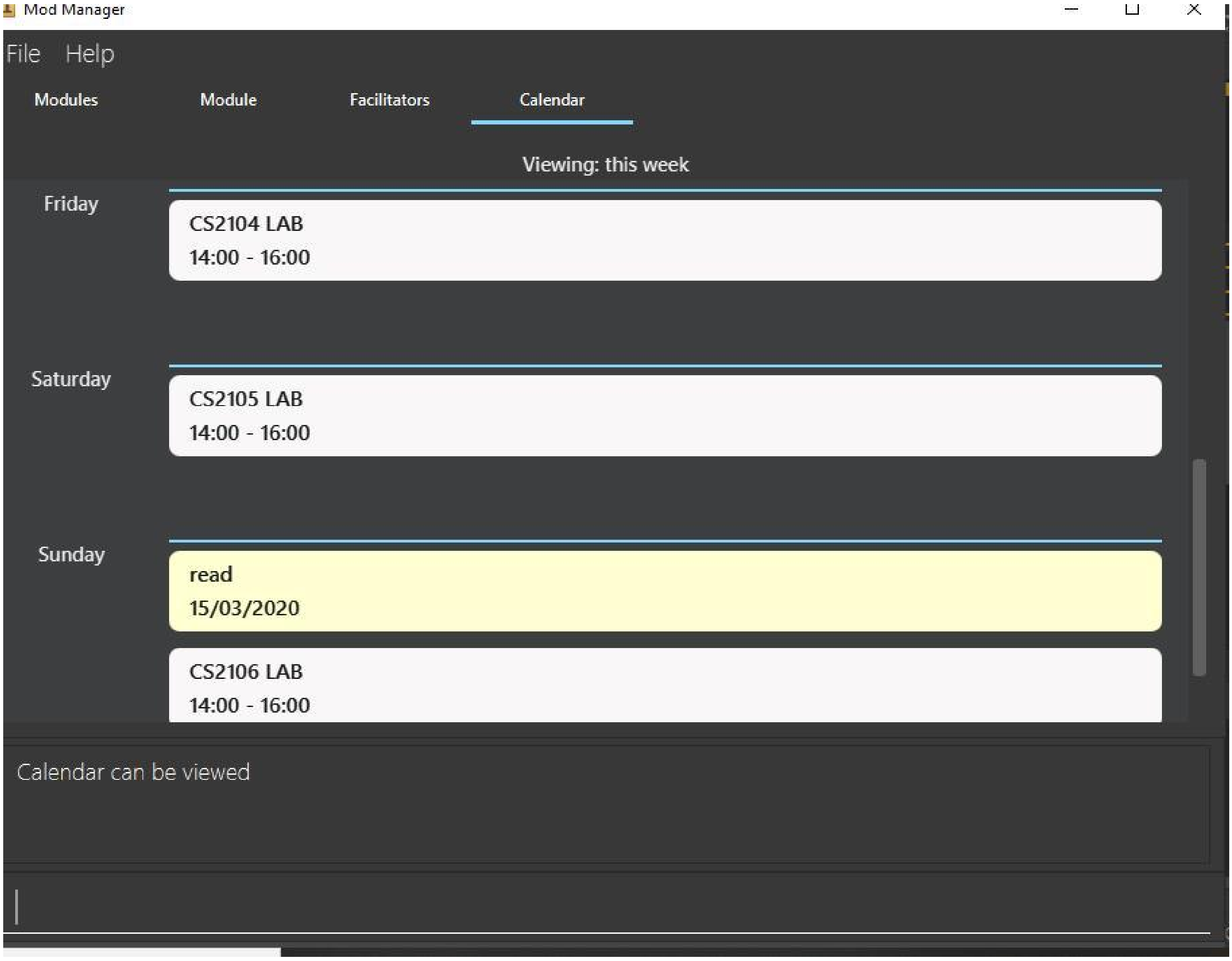
4.4.2. Design Considerations

Aspect: Calendar appearance



Calendar can be viewed

Figure 36. New design for calendar appearance (alternative 1)



Viewing: this week

Friday

CS2104 LAB
14:00 - 16:00

Saturday

CS2105 LAB
14:00 - 16:00

Sunday

read
15/03/2020

CS2106 LAB
14:00 - 16:00

Calendar can be viewed

Figure 37. Old design for calendar appearance (alternative 2)

- **Alternative 1 (current choice):** Displaying the days of a week in calendar from left to right.
  - Pros: The whole week can be seen on one screen without having users to scroll down for a particular day.
  - Cons: Words that are long in number of characters may not be able to be displayed in a single line.
- **Alternative 2:** Displaying the days of a week in the calendar from top to bottom.
  - Pros: Tasks and schedules that have description that are long can be displayed in a single line.
  - Cons: There is a need for users to scroll down to see a particular day. If there are many tasks and schedules in a day, the other days after it will be pushed downwards and this requires even more scrolling for users.

Alternative 1 is chosen as it is better that people are able to see their whole schedules and tasks for a week in one look. It makes better use of space than alternative 2 where the right side is usually not used.

Aspect: Command syntax for calendar find command

- **Alternative 1 (current choice):** User is required to input `cal find empty`.

    - Pros: It is short in command length.
    - Cons: Since there is only one type of calendar find, `empty` may seem redundant.
- **Alternative 2:** User is required to input `cal find /type empty`.
    - Pros: With the need to input `/type`, it can be clear about the type of find the command is trying to do. This is because without the `/type`, it is possible that users thought that the command is finding the word `empty`.
    - Cons: It can be tedious for users to type `/type` and this increases the command length.

Alternative 1 is chosen because it is shorter than alternative 2 and hence it can be easier for users to type. It is easier to implement too. The word `empty` is kept to allow users to know what the find command is for.

4.5. Classes Management feature

The class feature manages the classes in Mod Manager and is represented by the `Lesson` class. A class has a `ModuleCode`, `LessonType`, `day` which is a `DayOfWeek` object, `startTime`, `endTime` which are `LocalTime` objects and `venue` which is a `String`.

It supports the following operations:

- add - Adds a class to Mod Manager.
- list - Lists all classes in Mod Manager.
- edit - Edits a class in Mod Manager.
- delete - Deletes a class in Mod Manager.

4.5.1. Implementation Details

Adding a class

The add class command allows user to add a class to ModManager. This feature is facilitated by `LessonCommandParser`, `LessonAddCommandParser` and `LessonAddCommand`. The operation is exposed in the `Model` interface as `Model#addLesson()`.

Given below is an example usage scenario and how the lesson add mechanism behaves at each step.

1. The user executes the lesson add command and provides the module code, lesson type, day, start time, end time and venue of the lesson to be added.
2. `LessonAddCommandParser` creates a new `Lesson`, then a new `LessonAddCommand`.
3. `LogicManager` executes the `LessonAddCommand`.
4. `ModManager` adds the `Lesson` to `LessonList`.

The following sequence diagram shows how the lesson add command works:

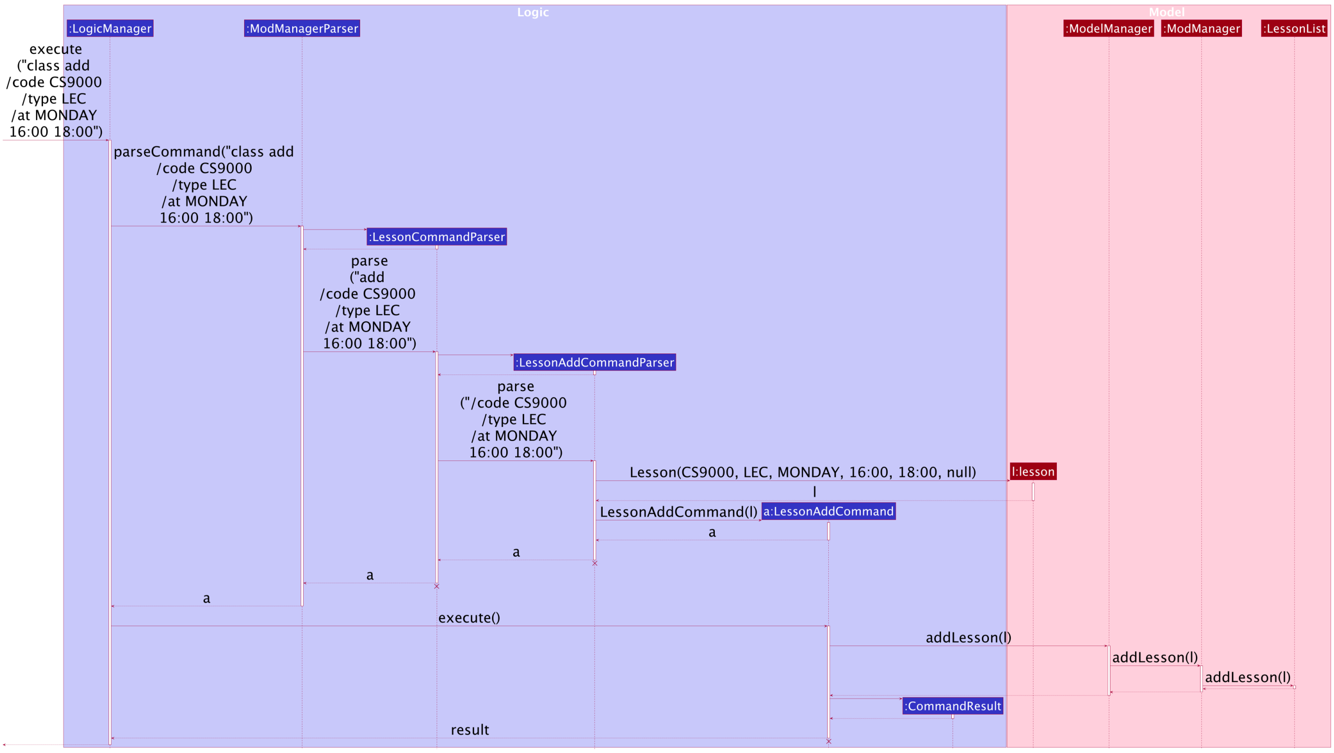

Logic
Model
:LogicManager
:ModManagerParser
:ModelManager
:ModManager
:LessonList
execute ("class add /code CS9000 /type LEC /at MONDAY 16:00 18:00")
parseCommand("class add /code CS9000 /type LEC /at MONDAY 16:00 18:00")
:LessonCommandParser
parse ("add /code CS9000 /type LEC /at MONDAY 16:00 18:00")
:LessonAddCommandParser
parse ("/code CS9000 /type LEC /at MONDAY 16:00 18:00")
Lesson(CS9000, LEC, MONDAY, 16:00, 18:00, null)
l:lesson
l
LessonAddCommand(l)
a:LessonAddCommand
a
a
a
a
execute()
addLesson(l)
addLesson(l)
addLesson(l)
:CommandResult
result

Figure 38. Sequence diagram for `class add` command

NOTE The lifeline for LessonCommandParser, LessonAddCommandParser and LessonAddCommand should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

The following activity diagram summarizes what happens when a user executes a lesson add command:

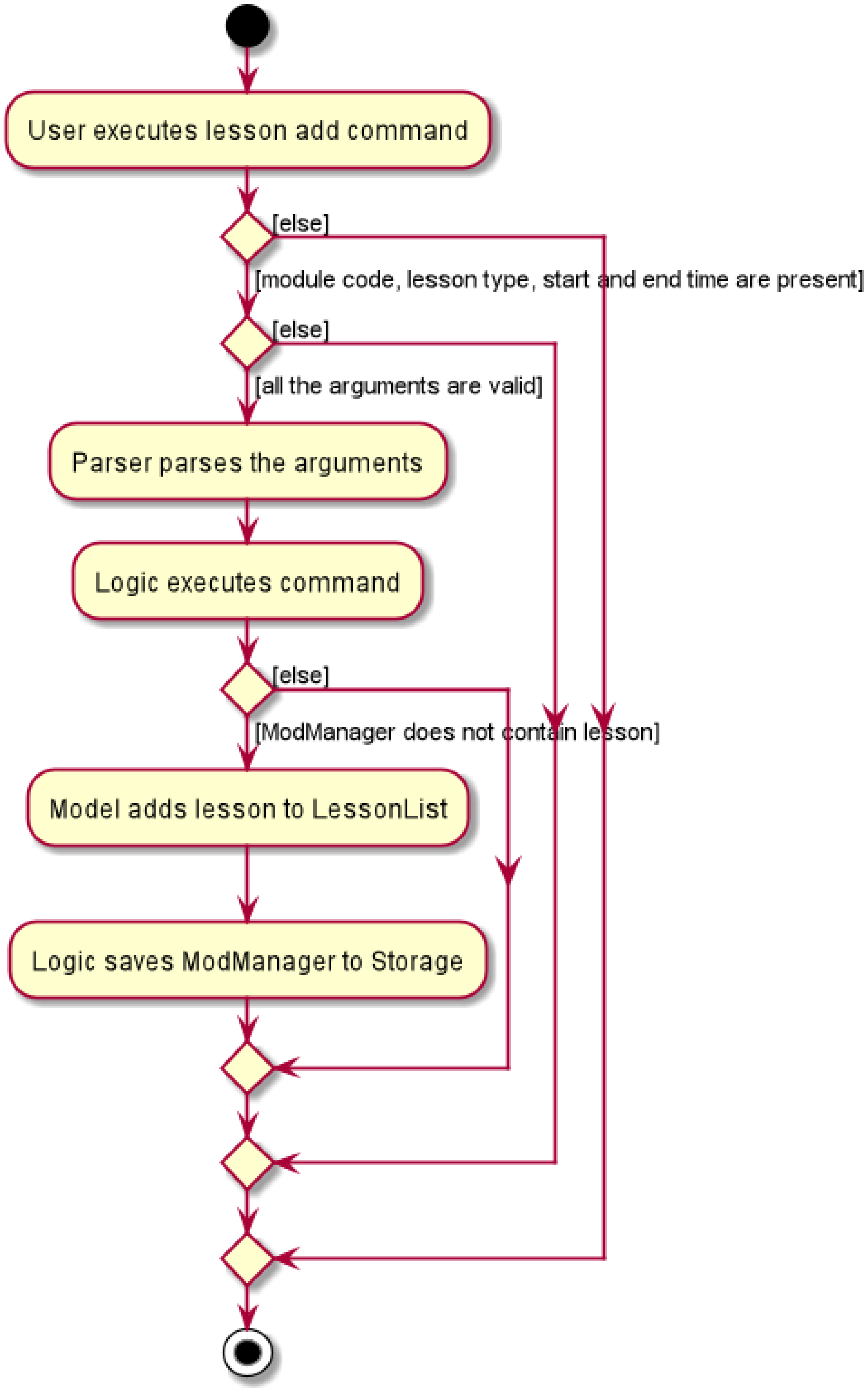

User executes lesson add command
[else]
[module code, lesson type, start and end time are present]
[else]
[all the arguments are valid]
Parser parses the arguments
Logic executes command
[else]
[ModManager does not contain lesson]
Model adds lesson to LessonList
Logic saves ModManager to Storage

Figure 39. Activity diagram for `class add` command

4.5.2. Design Considerations

Aspect: Prefix of day and time

- **Alternative 1: (current choice)** Have one prefix for all three `day`, `startTime` and `endTime` fields.
  - Pros: User types less.
  - Cons: When user wants to edit one field only, user have to key in other unnecessary details.

- **Alternative 2:** Have one prefix each for `day`, `startTime` and `endTime` fields.
    - Pros: Easier to parse and less invalid inputs to take note of. User can also edit any field.
    - Cons: More prefixes to remember and command will be very lengthy.

4.6. Logging

We are using `java.util.logging` package for logging. The `LogsCenter` class is used to manage the logging levels and logging destinations.

- The logging level can be controlled using the `logLevel` setting in the configuration file (See Section 4.7, "Configuration")
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

- `SEVERE` : Critical problem detected which may possibly cause the termination of the application
- `WARNING` : Can continue, but with caution
- `INFO` : Information showing the noteworthy actions by the App
- `FINE` : Details that is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

4.7. Configuration

Certain properties of the application can be controlled (e.g user prefs file location, logging level) through the configuration file (default: `config.json`).

5. Documentation

Refer to the guide here.

6. Testing

Refer to the guide here.

7. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile:

- is a NUS student
- has a need to manage modules taken in a semester
- has a need to manage classes, tasks and facilitators for each module
- has a need to visualize schedule and tasks of the week in a calendar
- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition:

- manage school-related modules faster than a typical mouse/GUI driven app
- view schedule and tasks for the current and upcoming week easily
- navigate easily with the command assistant for quicker management

Appendix B: User Stories

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

Priority	As a …	I want to …	So that I can…
* * *	new user	see usage instructions	refer to instructions when I forget how to use the App
* * *	student	add modules I am taking	keep track of the information related to the module
* * *	student	add classes	keep track of the classes I have for a particular module
* * *	student	add tasks	keep track of the tasks I have for a particular module

Priority	As a …	I want to …	So that I can…
* * *	student	add facilitators' information	keep track of the information of the facilitators
* * *	student	view information related to a module	prepare for each module
* * *	student	view tasks	complete them
* * *	student	view facilitators' information	contact them when I need help
* * *	student	edit a module’s description	modify the module’s description
* * *	student	edit classes	keep my classes up to date
* * *	student	edit a task	keep my tasks up to date
* * *	student	edit a facilitator’s information	keep their contact details up to date
* * *	student	delete a module	use the App for different semesters
* * *	student	delete a class	remove classes that I am no longer in
* * *	student	delete a task	remove tasks that I no longer need to track
* * *	student	delete a facilitator’s information	remove information that I no longer need
* * *	busy student	view schedule for the current week	prepare for them

Priority	As a …	I want to …	So that I can…
* * *	busy student	view schedule for the upcoming week	prepare for them
* * *	new user	view all commands	learn how to use them
* * *	new user	view commands for a specific feature	learn how to use them
* * *	user	get reminded about how commands work	recall the commands
* * *	user	import and export data	easily migrate the data to another computer
* *	student	find a facilitator by name	locate details of facilitators without having to go through the entire list
* *	student	find tasks by date	keep track of tasks on a particular date
* *	student	find upcoming tasks	prioritise them
* *	busy student	find empty slots in my schedule	manage my time easily
*	student	mark a task as done	not take note of them anymore
*	student	add a priority level to a task	prioritise my tasks
*	student	tag my tasks	categorise them

Priority	As a …	I want to …	So that I can…
*	student	see countdown timers	be reminded of deadlines
*	busy student	receive reminders about deadlines and events the next day	take note of them
*	student	mass delete the modules	delete them quickly once the semester is over
*	advanced user	use shorter versions of a command	type a command faster
*	careless user	undo my commands	undo the mistakes in my command
*	visual user	see a clear GUI	navigate the App more easily

Appendix C: Use Cases

(For all use cases below, the **System** is the `Mod Manager` and the **Actor** is the `user`, unless specified otherwise)

Use case: UC01 - Add module

MSS

1. User requests to add a module and provides the module code and description of the module.
2. Mod Manager adds the module.

   Use case ends.

Extensions

1a. Compulsory fields are not provided.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

1b. The module code or description is invalid.

1b1. Mod Manager shows an error message.

Use case resumes at step 1.

Use case: UC02 - List modules

MSS

1. User requests to list all modules.
2. Mod Manager shows the list of all the modules.

   Use case ends.

Extensions

1a. The list of modules is empty.

Use case ends.

Use case: UC03 - View module

MSS

1. User requests to view a module and provides the module code.
2. Mod Manager shows all information related to the module.

   Use case ends.

Extensions

1a. The given module code is invalid.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

Use case: UC04 - Edit module

MSS

1. User requests to edit a module and provides the index or module code and the new description.
2. Mod Manager edits the module.

   Use case ends.

Extensions

1a. The given index or module code is invalid.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

1b. The new description is invalid.

1b1. Mod Manager shows an error message.

Use case resumes at step 1.

Use case: UC05 - Delete module

MSS

1. User requests to delete a module and provides the index or module code.
2. Mod Manager deletes the module.

Use case ends.

Extensions

1a. The given index or module code is invalid.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

Use case: UC06 - Add class

MSS

1. User request to add a class and provides the details of the new class.
2. Mod Manager adds a class.

Use case ends.

Extensions

1a. Compulsory fields are not provided or fields provided are invalid.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

Use case: UC07 - List classes

MSS

1. User request to list all the classes.
2. Mod Manager replies with the list of all classes.

   Use case ends.

Extensions

1a. The list of classes is empty.

Use case ends.

Use case: UC08 - Find class by day

MSS

1. User request to list all the classes by day and provides the day.
2. Mod Manager replies with the list of classes.

   Use case ends.

Extensions

1a. Day provided is invalid.

  1a1. Mod Manager shows an error message.

  Use case resumes at step 1.

1b. No class on the day provided.

Use case ends.

Use case: UC09 - Find next class

MSS

1. User request to find the next class.
2. Mod Manager replies with the next class.

   Use case ends.

Extensions

1a. No next class.

Use case ends.

Use case: UC10 - Edit class

MSS

1. User request to edit a class and provides the index and necessary details to be edited.
2. Mod Manager edits the class.

   Use case ends.

Extensions

1a. Index is not provided or invalid, or details are not provided or invalid.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

Use case: UC11 - Delete class

MSS

1. User requests to delete a class and provides the index.
2. Mod Manager deletes the class.

   Use case ends.

Extensions

1a. Index is not provided or is invalid.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

Use case: UC18 - Add facilitator

MSS

1. User requests to add a facilitator and provides the details of the facilitator.
2. Mod Manager adds the facilitator.

   Use case ends.

Extensions

1a. Compulsory fields are not provided or none of the optional fields provided.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

1b. Fields provided are invalid.

1b1. Mod Manager shows an error message.

Use case resumes at step 1.

Use case: UC19 - List facilitators

MSS

1. User requests to list all facilitators.
2. Mod Manager shows the list of all the facilitators.

Use case ends.

Extensions

1a. The list of facilitators is empty.

Use case ends.

Use case: UC20 - Find facilitator

MSS

1. User requests to find a facilitator and provides a keyword.
2. Mod Manager shows the list of facilitators whose names contain the keyword.

Use case ends.

Extensions

1a. None of the names of the facilitators contain the keyword.

Use case ends.

Use case: UC21 - Edit facilitator

MSS

1. User requests to edit a facilitator and provides the index or module code and new details.
2. Mod Manager edits the facilitator.

Use case ends.

Extensions

1a. The given index or module code is invalid.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

1a. Fields provided are invalid.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

Use case: UC22 - Delete facilitator

MSS

1. User requests to delete a facilitator and provides the index or module code.
2. Mod Manager deletes the facilitator.

Use case ends.

Extensions

1a. The given index or module code is invalid.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

Use case: UC23 - View calendar

MSS

1. User requests to view the calendar for a specified week.
2. Mod Manager shows the calendar for the specified week.

Use case ends.

Extensions

1a. The specified week is invalid.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

Use case: UC24 - Find empty slots in calendar

MSS

1. User requests to find empty slots in the calendar.
2. Mod Manager shows the list of empty slots available.

   Use case ends.

Extensions

1a. The given input is invalid.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

2a. The list of empty slots is empty.

Use case ends.

Use case: UC25 - Clear all entries in Mod Manager

MSS

1. User requests to clear all entries.
2. Mod Manager clears all entries.

   Use case ends.

Extensions

1a. The given input is invalid.

1a1. Mod Manager shows an error message.

Use case resumes at step 1.

Appendix D: 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 250 classes, 250 tasks and 250 facilitators and without a noticeable sluggishness in performance for typical usage.
3. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.
4. Should work without any internet required.
5. Should be for a single user.

6. Data should be stored locally and should be in a human editable file.

Appendix E: Glossary

CLI

Command-line interface: processes commands to a computer program in the form of lines of text

Extensions

"Add-on"s to the MSS that describe exceptional or alternative flow of events, describe variations of the scenario that can happen if certain things are not as expected by the MSS

GUI

Graphical user interface: a form of user interface that allows user to interact with electronic devices through graphical icons

Mainstream OS

Windows, Linux, Unix, OS-X

MSS

Main Success Scenario: describes the most straightforwards interaction for a given use case, which assumes that nothing goes wrong

Appendix F: Product Survey

Product Name

Author: …

Pros:

- …
- …

Cons:

- …
- …

Appendix G: Instructions for Manual Testing

Given below are instructions to test the app manually.

NOTE These instructions only provide a starting point for testers to work on; testers are expected to do more *exploratory* testing.

G.1. Launch and Shutdown

1. Initial launch
   a. Download the jar file and copy into an empty folder
   b. Double-click the jar file
      Expected: Shows the GUI with a set of sample contacts. The window size may not be optimum.
2. Saving window preferences
   a. Resize the window to an optimum size. Move the window to a different location. Close the window.
   b. Re-launch the app by double-clicking the jar file.
      Expected: The most recent window size and location is retained.

{ more test cases … }

G.2. Deleting a facilitator

1. Deleting a facilitator while all facilitators are listed
   a. Prerequisites: List all facilitators using the `list` command. Multiple facilitators in the list.
   b. Test case: `delete 1`
      Expected: First contact is deleted from the list. Details of the deleted contact shown in the status message. Timestamp in the status bar is updated.
   c. Test case: `delete 0`
      Expected: No facilitator is deleted. Error details shown in the status message. Status bar remains the same.
   d. Other incorrect delete commands to try: `delete`, `delete x` (where x is larger than the list size) *{give more}*
      Expected: Similar to previous.

{ more test cases … }

G.3. Saving data

1. Dealing with missing/corrupted data files
   a. *{explain how to simulate a missing/corrupted file and the expected behavior}*

{ more test cases … }