TA-Tracker - Developer Guide

By: `Team SE-EDU` Since: `Jun 2016` Licence: `MIT`

1. Setting up

Refer to the guide here.

2. Design

2.1. Architecture

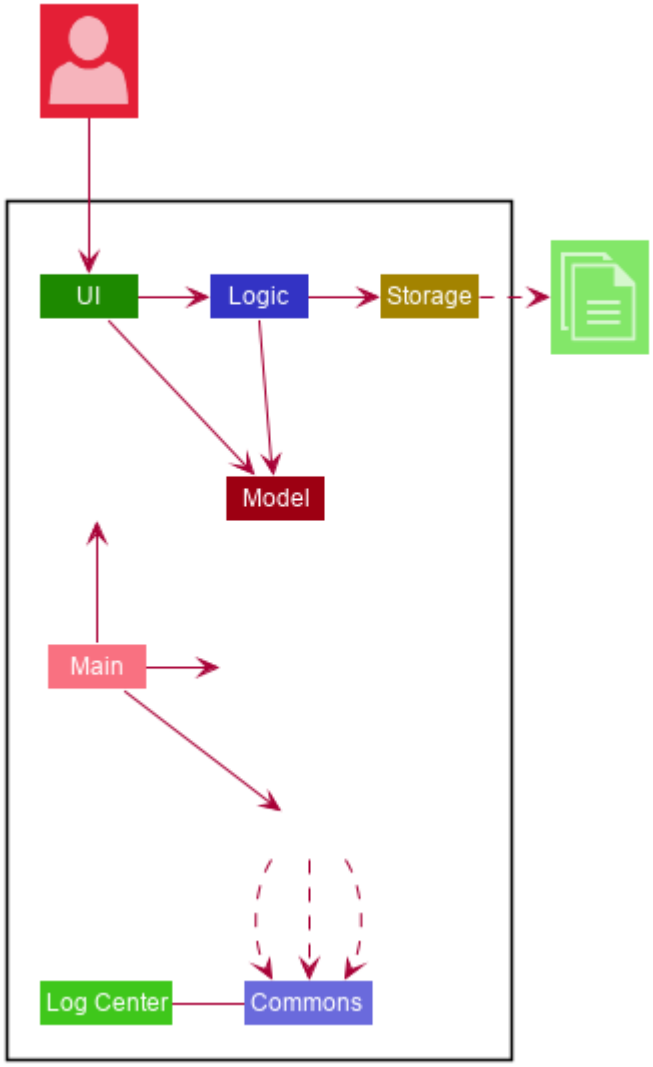


Figure 1. Architecture Diagram

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

TIP	The `.puml` files used to create diagrams in this document can be found in the diagrams folder. Refer to the Using PlantUML guide to learn how to create and edit diagrams.

`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 its API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

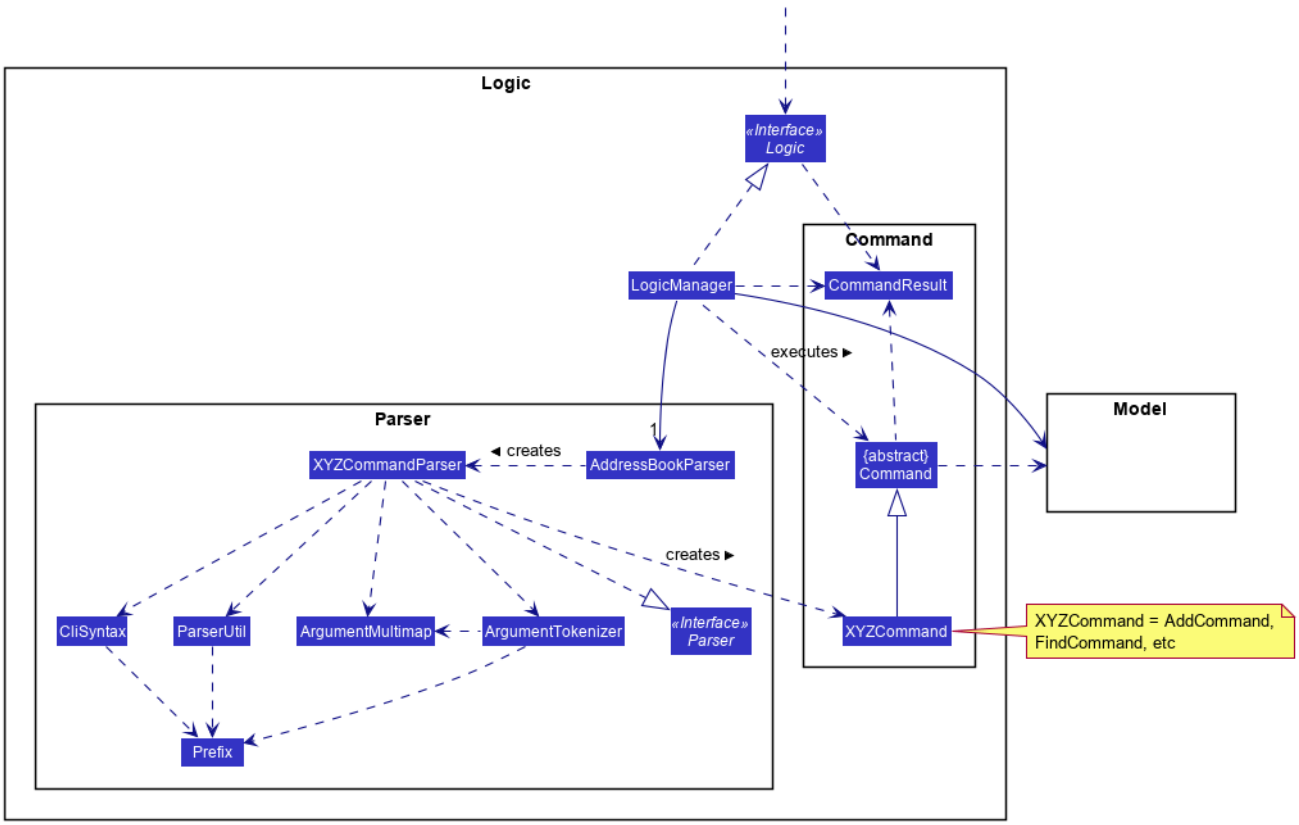


Figure 2. Class Diagram of the Logic Component

How the architecture components interact with each other

The *Sequence Diagram* below shows how the components interact with each other for the scenario

where the user issues the command `delete 1`.

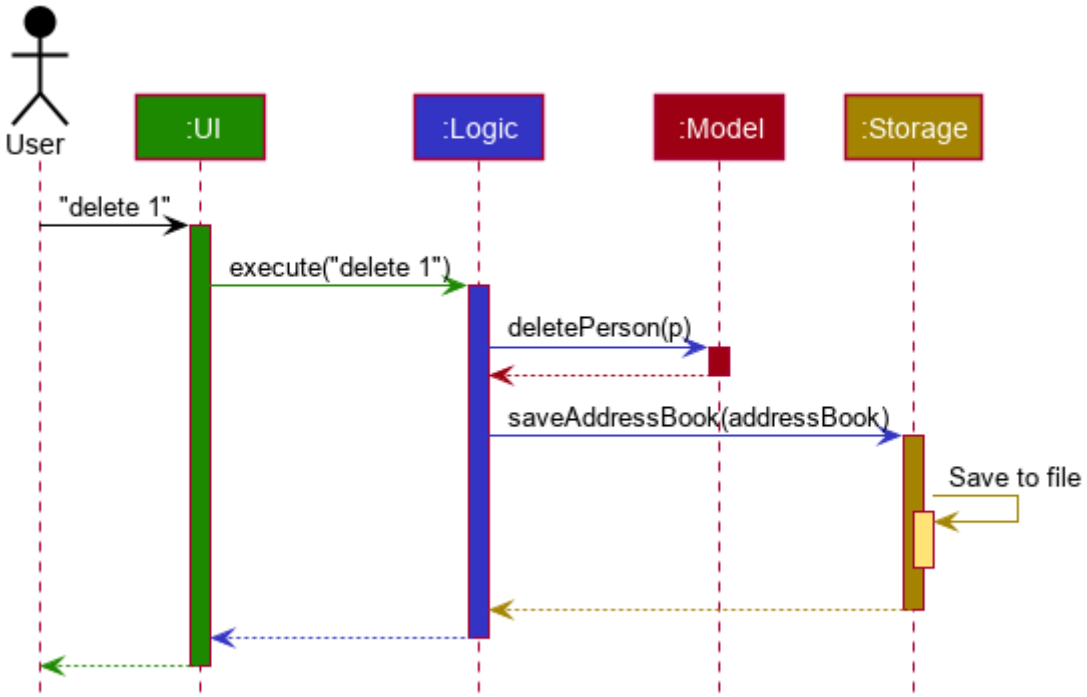


Figure 3. Component interactions for `delete 1` command

The sections below give more details of each component.

2.2. UI component

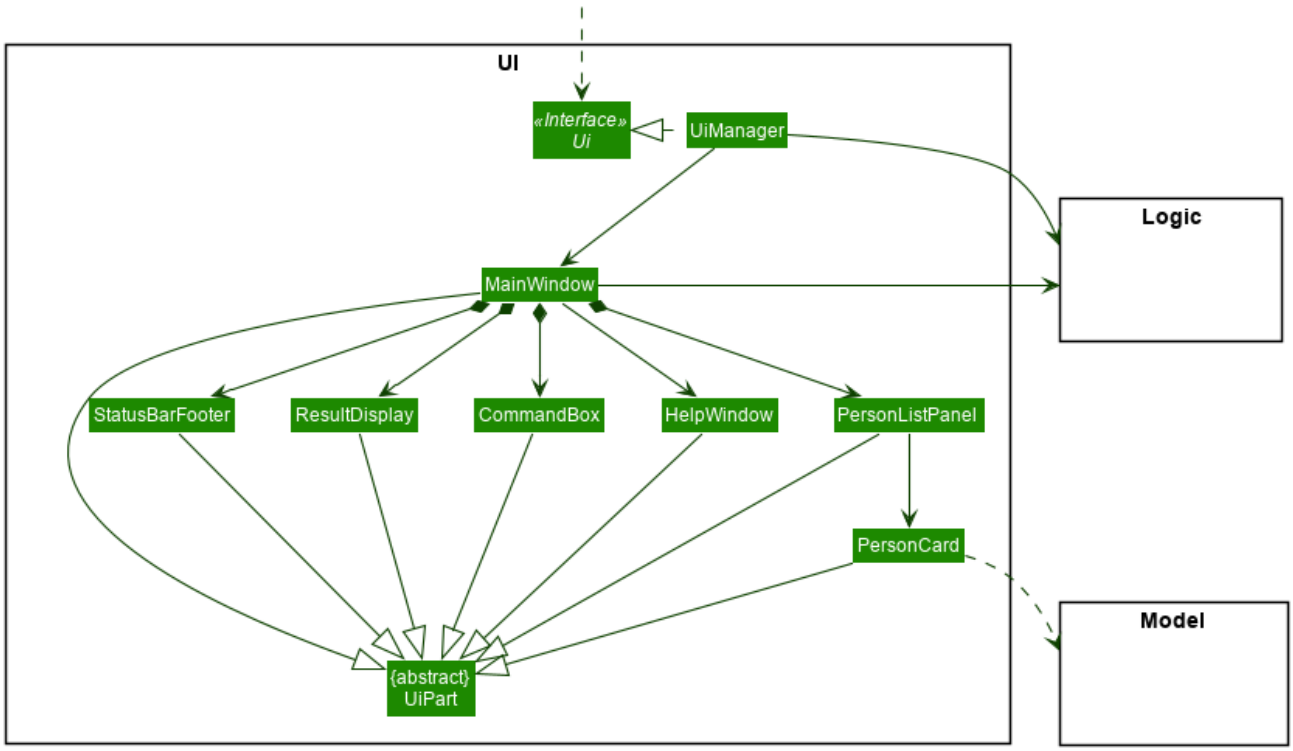


Figure 4. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `StudentListPanel`, `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.

2.3. Logic component

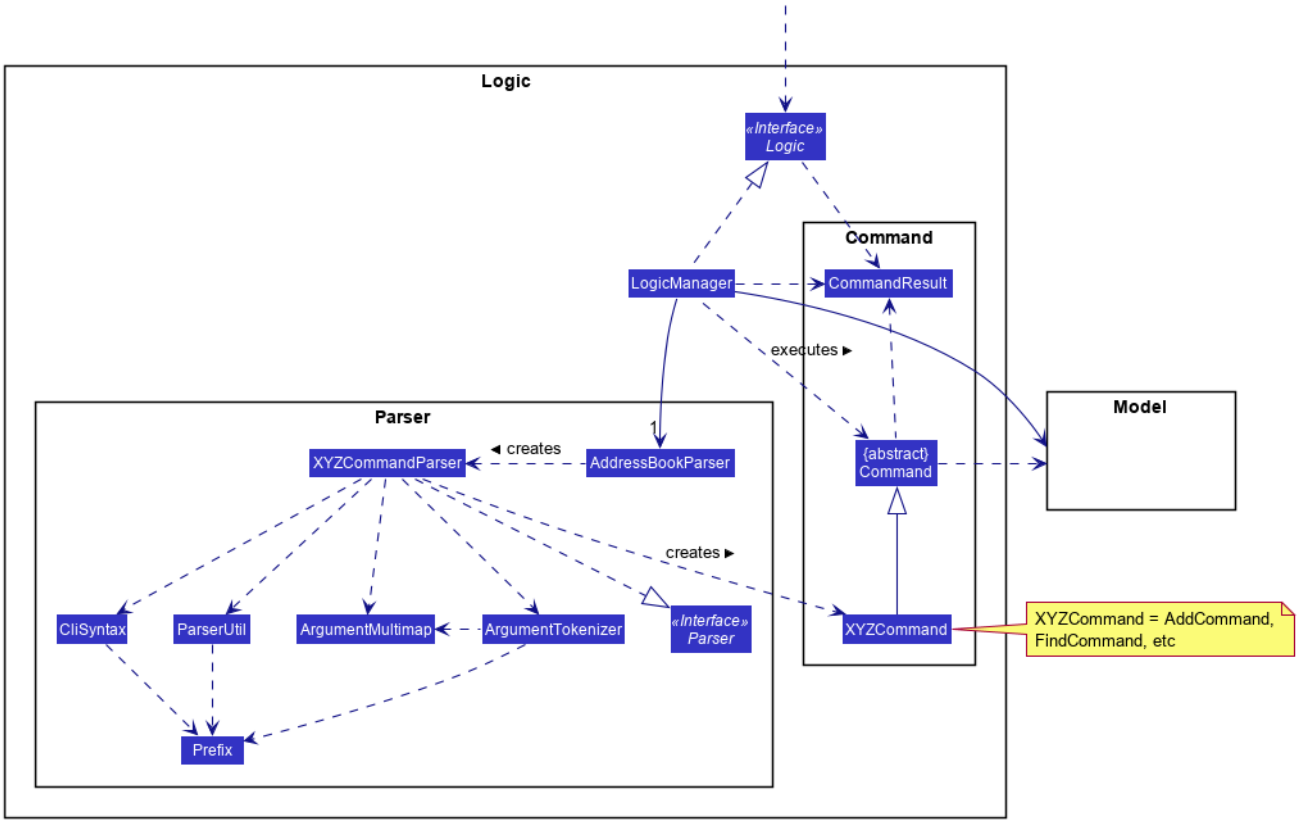


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `TaTrackerParser` 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 student).
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.

Given below is the Sequence Diagram for interactions within the `Logic` component for the `execute("delete 1")` API call.

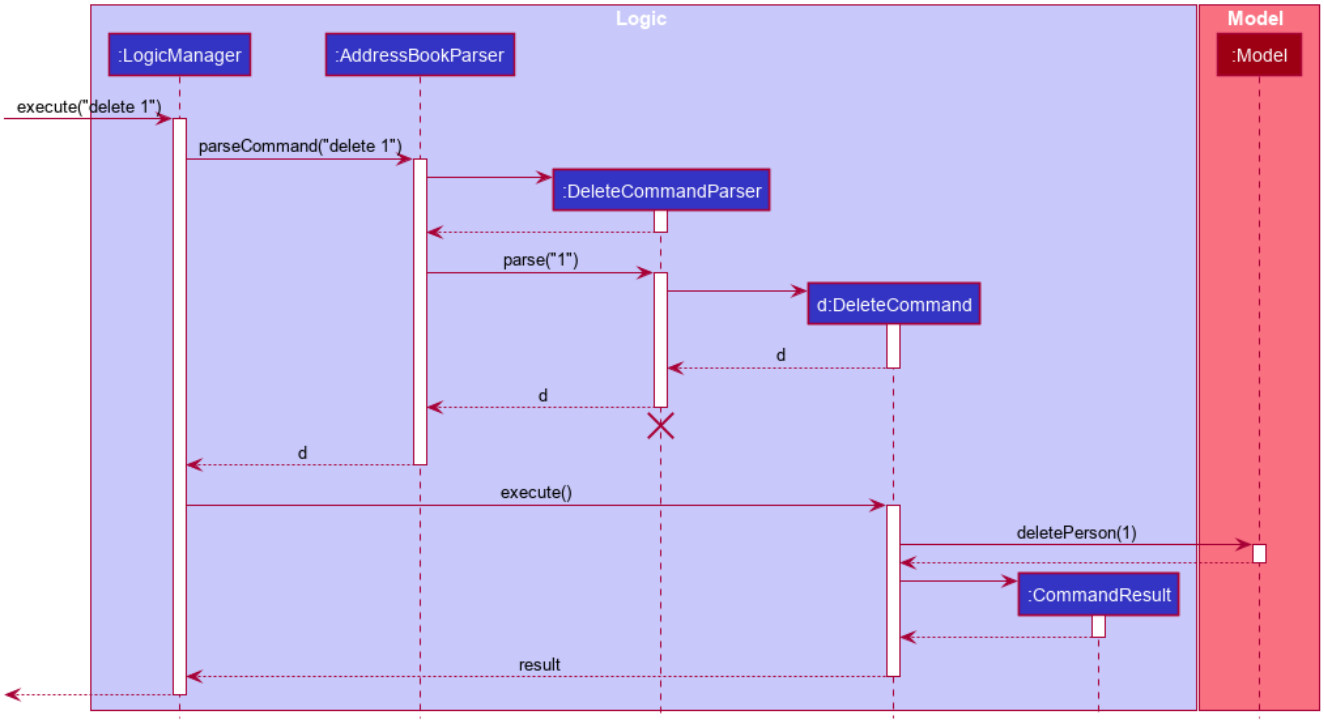


Figure 6. Interactions Inside the Logic Component for the `delete 1` Command

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

2.4. Model component

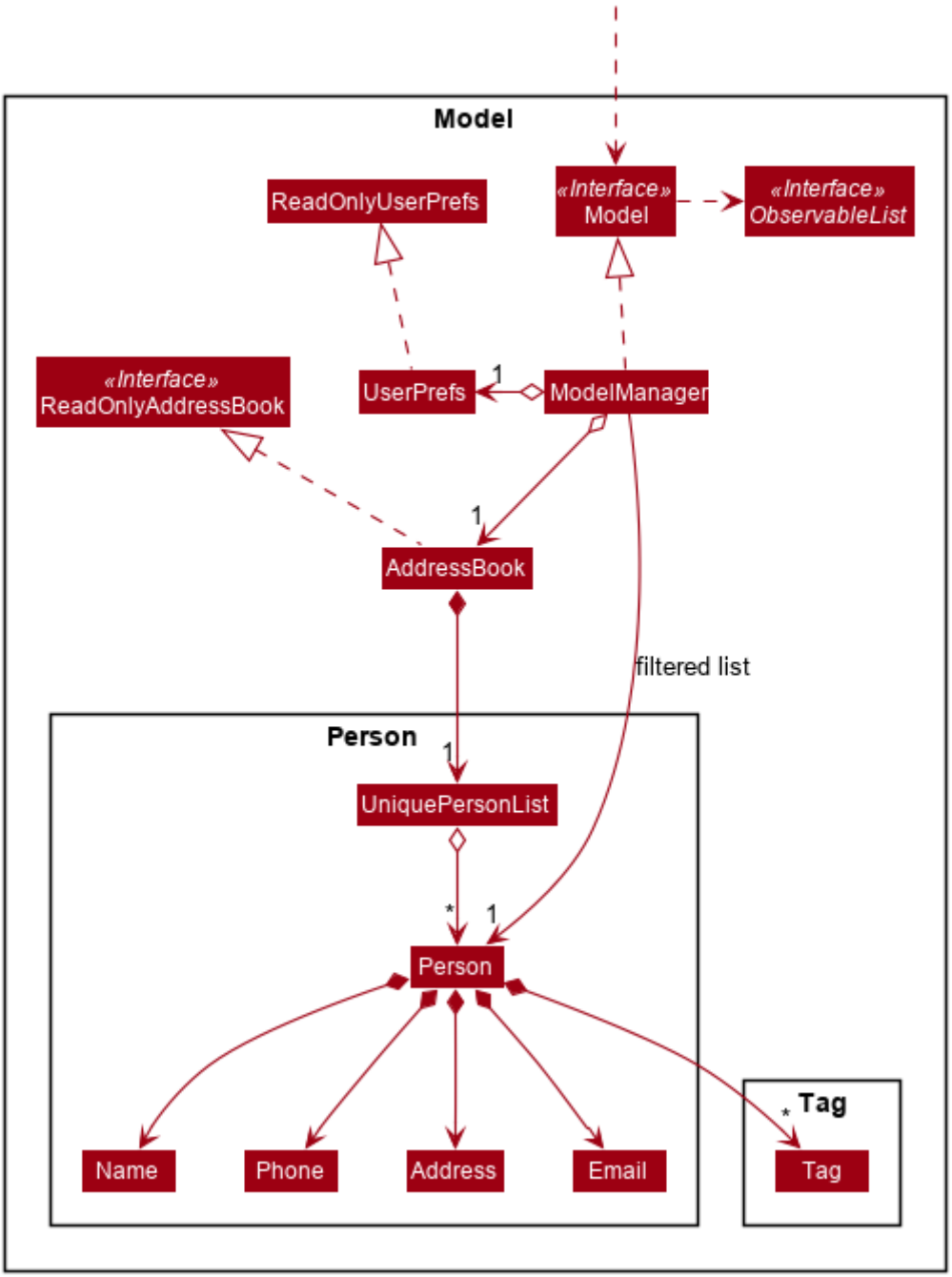


Figure 7. Structure of the Model Component

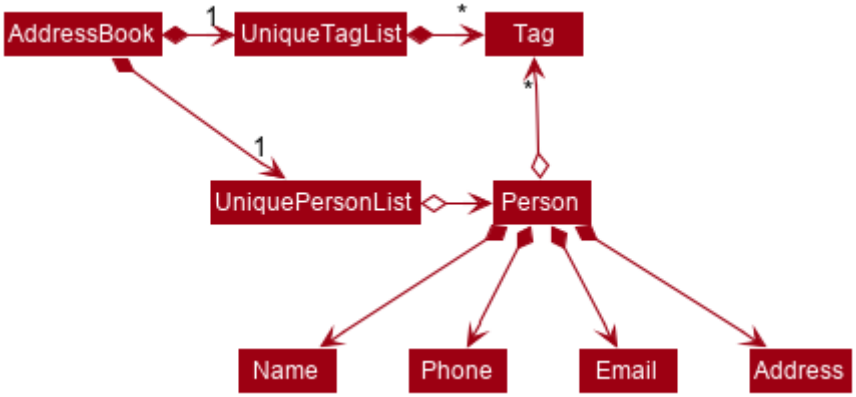
API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the TA-Tracker data.
- exposes an unmodifiable `ObservableList<Student>` 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.

NOTE

As a more OOP model, we can store a `Tag` list in `TaTracker`, which `Student` can reference. This would allow `TaTracker` to only require one `Tag` object per unique `Tag`, instead of each `Student` needing their own `Tag` object. An example of how such a model may look like is given below.



2.5. Storage component

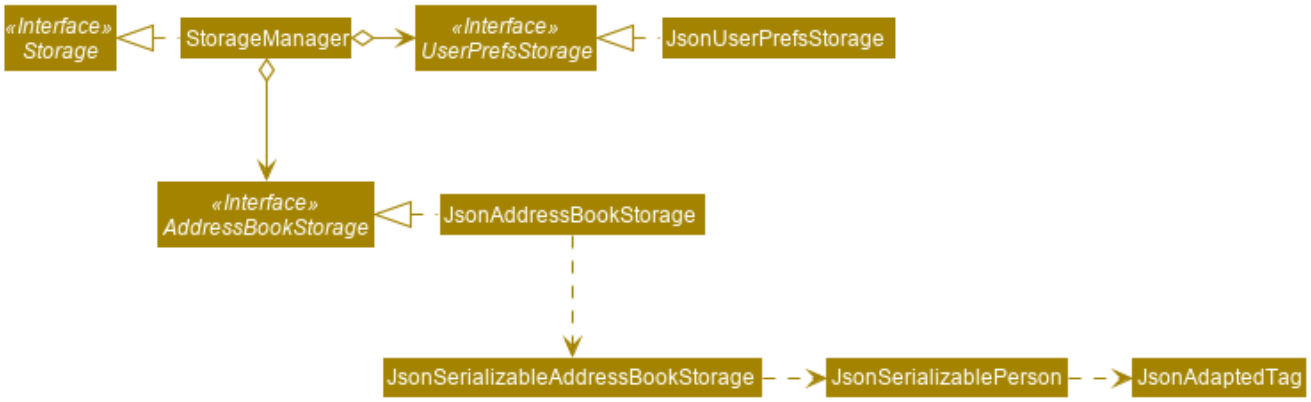


Figure 8. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

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

2.6. Common classes

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

3. Implementation

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

3.1. [Proposed] Undo/Redo feature

3.1.1. Proposed Implementation

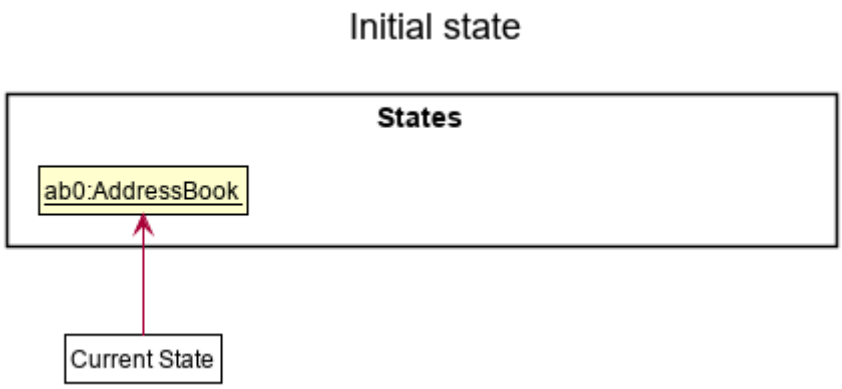
The undo/redo mechanism is facilitated by `VersionedTaTracker`. It extends `TaTracker` with an undo/redo history, stored internally as an `taTrackerStateList` and `currentStatePointer`. Additionally, it implements the following operations:

- `VersionedTaTracker#commit()` — Saves the current TA-Tracker state in its history.
- `VersionedTaTracker#undo()` — Restores the previous TA-Tracker state from its history.
- `VersionedTaTracker#redo()` — Restores a previously undone TA-Tracker state from its history.

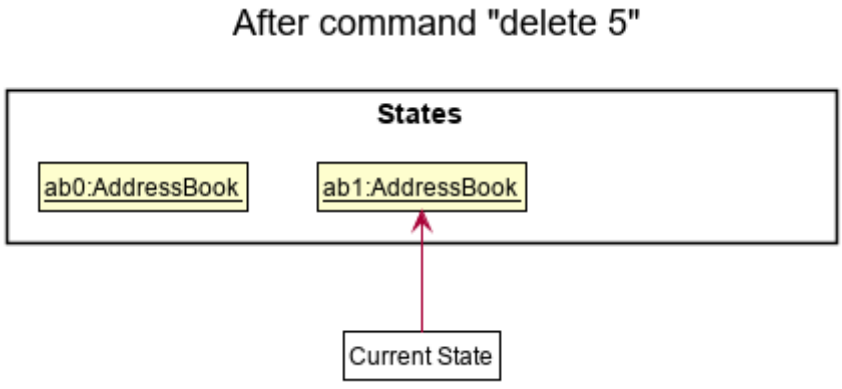
These operations are exposed in the `Model` interface as `Model#commitTaTracker()`, `Model#undoTaTracker()` and `Model#redoTaTracker()` respectively.

Given below is an example usage scenario and how the undo/redo mechanism behaves at each step.

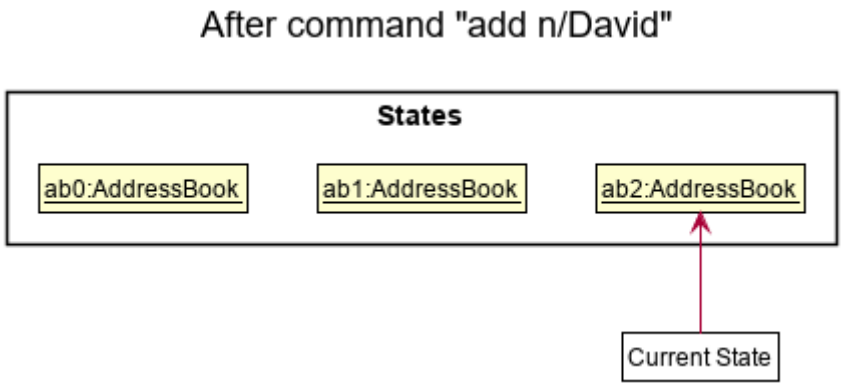
Step 1. The user launches the application for the first time. The `VersionedTaTracker` will be initialized with the initial TA-Tracker state, and the `currentStatePointer` pointing to that single TA-Tracker state.



Step 2. The user executes `delete 5` command to delete the 5th student in the TA-Tracker. The `delete` command calls `Model#commitTaTracker()`, causing the modified state of the TA-Tracker after the `delete 5` command executes to be saved in the `taTrackerStateList`, and the `currentStatePointer` is shifted to the newly inserted TA-Tracker state.

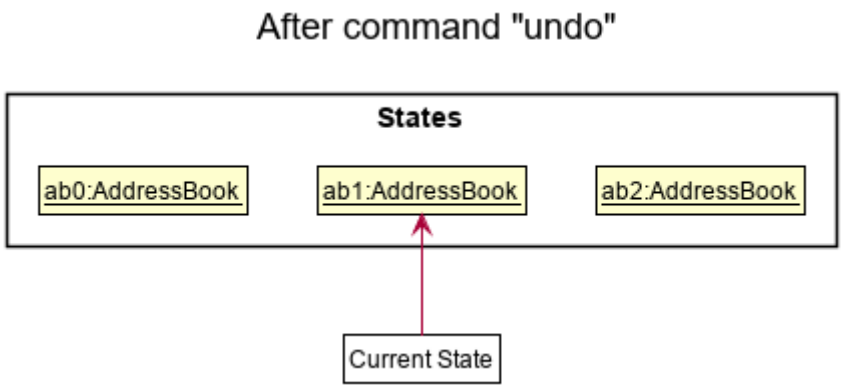


Step 3. The user executes `add n/David ...` to add a new student. The `add` command also calls `Model#commitTaTracker()`, causing another modified TA-Tracker state to be saved into the `taTrackerStateList`.



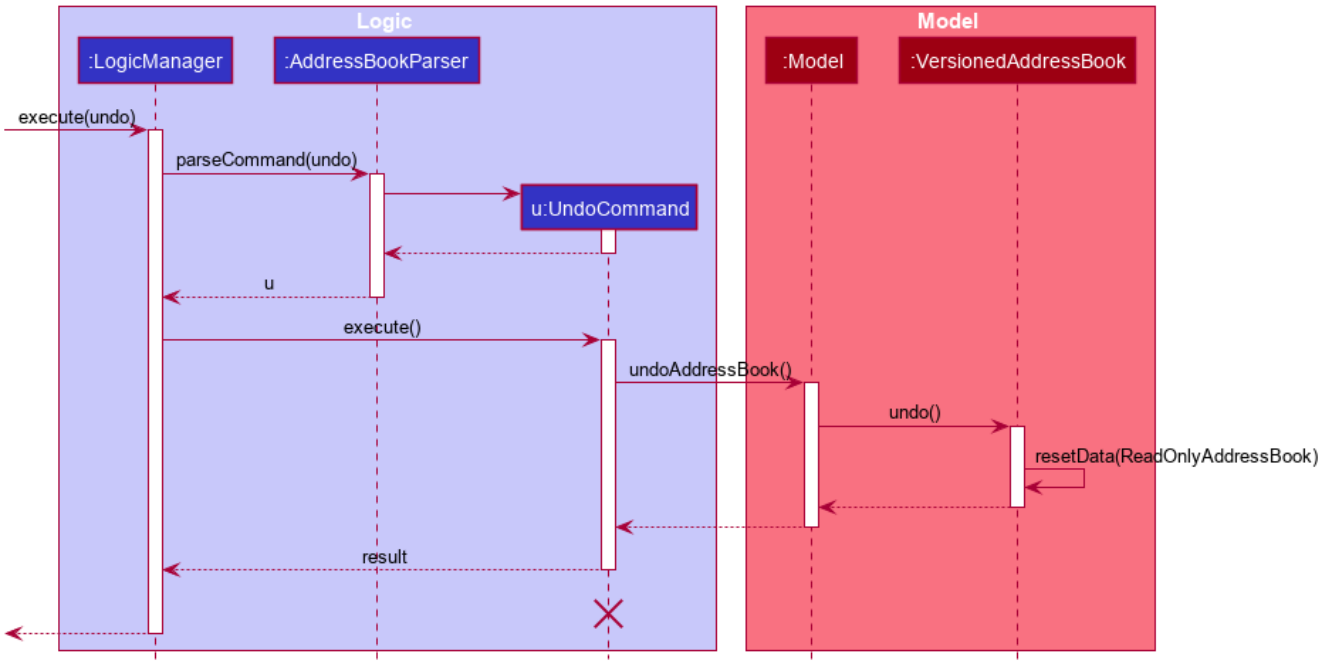
NOTE If a command fails its execution, it will not call `Model#commitTaTracker()`, so the TA-Tracker state will not be saved into the `taTrackerStateList`.

Step 4. The user now decides that adding the student was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoTaTracker()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous TA-Tracker state, and restores the TA-Tracker to that state.



NOTE If the `currentStatePointer` is at index 0, pointing to the initial TA-Tracker state, then there are no previous TA-Tracker states to restore. The `undo` command uses `Model#canUndoTaTracker()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

The following sequence diagram shows how the undo operation works:

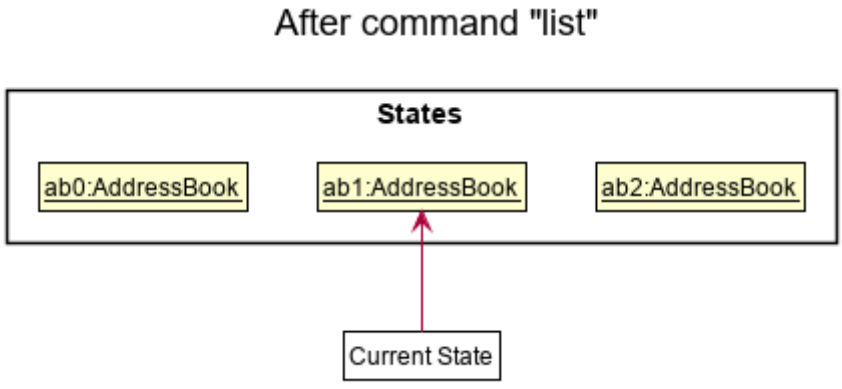


NOTE The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

The `redo` command does the opposite — it calls `Model#redoTaTracker()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores the TA-Tracker to that state.

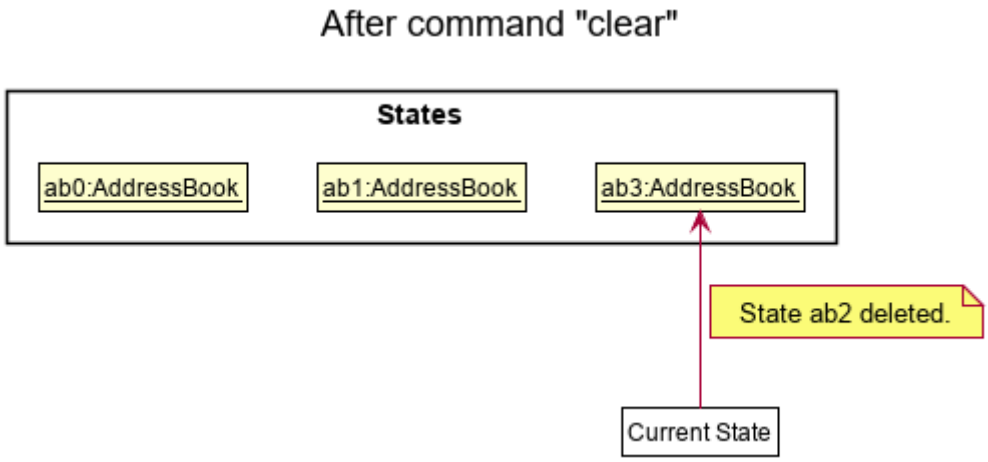
NOTE If the `currentStatePointer` is at index `taTrackerStateList.size() - 1`, pointing to the latest TA-Tracker state, then there are no undone TA-Tracker states to restore. The `redo` command uses `Model#canRedoTaTracker()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the redo.

Step 5. The user then decides to execute the command `list`. Commands that do not modify the TA-Tracker, such as `list`, will usually not call `Model#commitTaTracker()`, `Model#undoTaTracker()` or `Model#redoTaTracker()`. Thus, the `taTrackerStateList` remains unchanged.

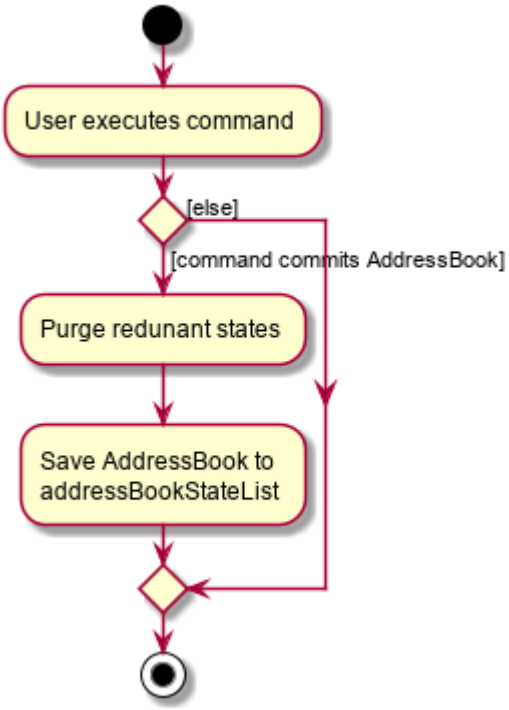


Step 6. The user executes `clear`, which calls `Model#commitTaTracker()`. Since the `currentStatePointer` is not pointing at the end of the `taTrackerStateList`, all TA-Tracker states after the `currentStatePointer` will be purged. We designed it this way because it no longer makes sense to redo the `add n/David …` command. This is the behavior that most modern desktop applications

follow.



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



3.1.2. Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire TA-Tracker.
  - Pros: Easy to implement.
  - Cons: May have performance issues in terms of memory usage.
- **Alternative 2:** Individual command knows how to undo/redo by itself.
  - Pros: Will use less memory (e.g. for `delete`, just save the student being deleted).
  - Cons: We must ensure that the implementation of each individual command are correct.

Aspect: Data structure to support the undo/redo commands

- **Alternative 1 (current choice):** Use a list to store the history of TA-Tracker states.
  - Pros: Easy for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project.
  - Cons: Logic is duplicated twice. For example, when a new command is executed, we must remember to update both `HistoryManager` and `VersionedTaTracker`.
- **Alternative 2:** Use `HistoryManager` for undo/redo
  - Pros: We do not need to maintain a separate list, and just reuse what is already in the codebase.
  - Cons: Requires dealing with commands that have already been undone: We must remember to skip these commands. Violates Single Responsibility Principle and Separation of Concerns as `HistoryManager` now needs to do two different things.

3.2. Module View

Module view is the term used to characterise the different functionalities related to the modules and groups that the user is affiliated with.

3.2.1. Model Framework

The following class diagram shows how different classes are related in the functioning of the module view.

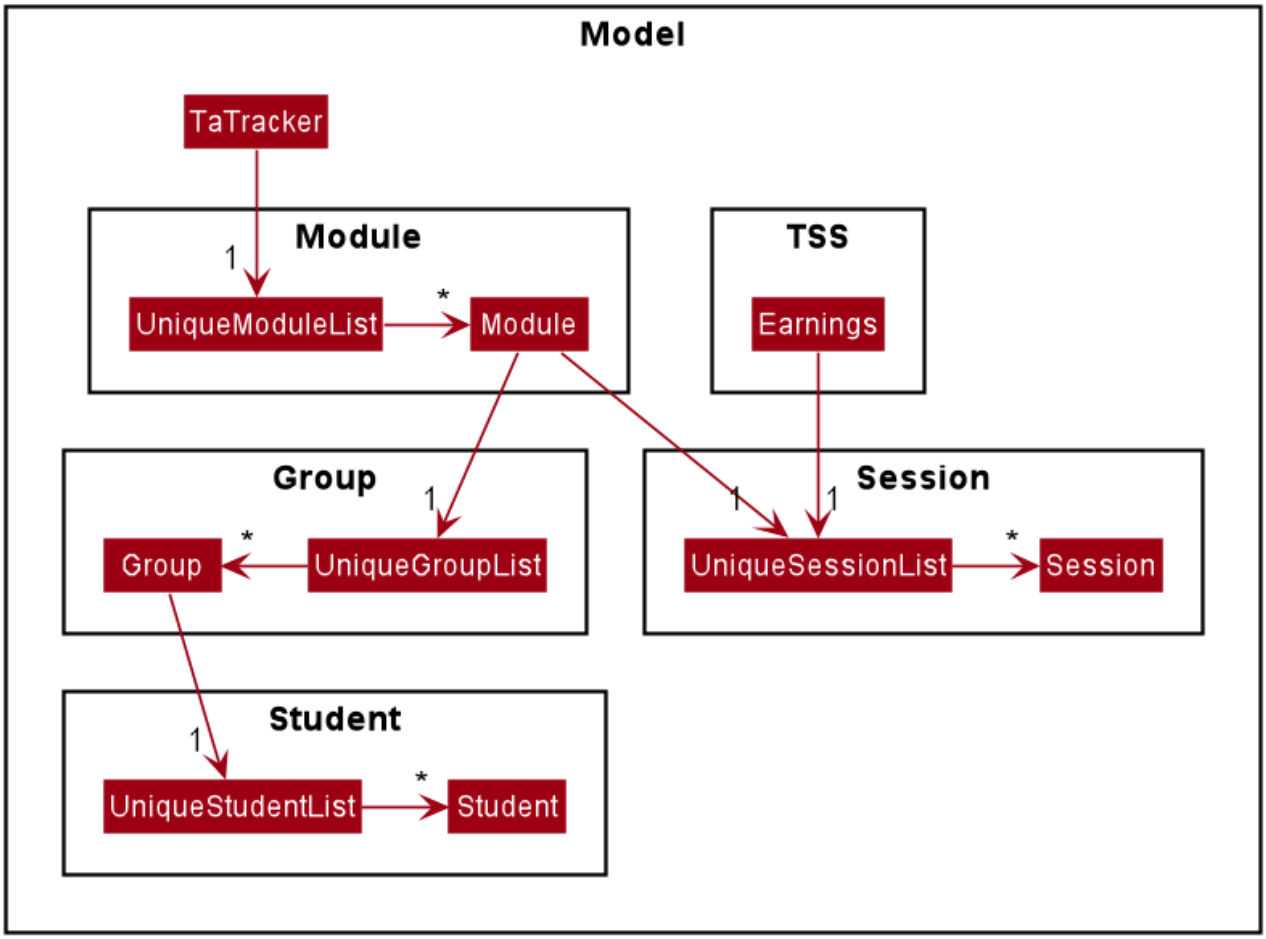


Figure 9. Module View - Class Diagram

The TaTracker model class contains a UniqueModuleList which helps it keep track of the different modules the user is associated with. Each module contains a UniqueGroupList and a UniqueSessionList.

The UniqueGroupList contains a list of all the groups of a module that the user is affiliated with. Each group contains a UniqueStudentsList that contains the students in that group.

The UniqueSessionList contains a list of all the sessions associated with the module that have been marked as done. This list is used in the TSS view.

3.2.2. Implementation of the Module Add and Delete Commands

The following sequence diagram shows the sequence of commands that take place between the logic and model components of the TA-Tracker when the user enters the command 'module add m/CS2103 n/Software Engineering'.

Note: This diagram assumes that there is no module with the module code 'CS2103' pre-existing in the TA-Tracker.

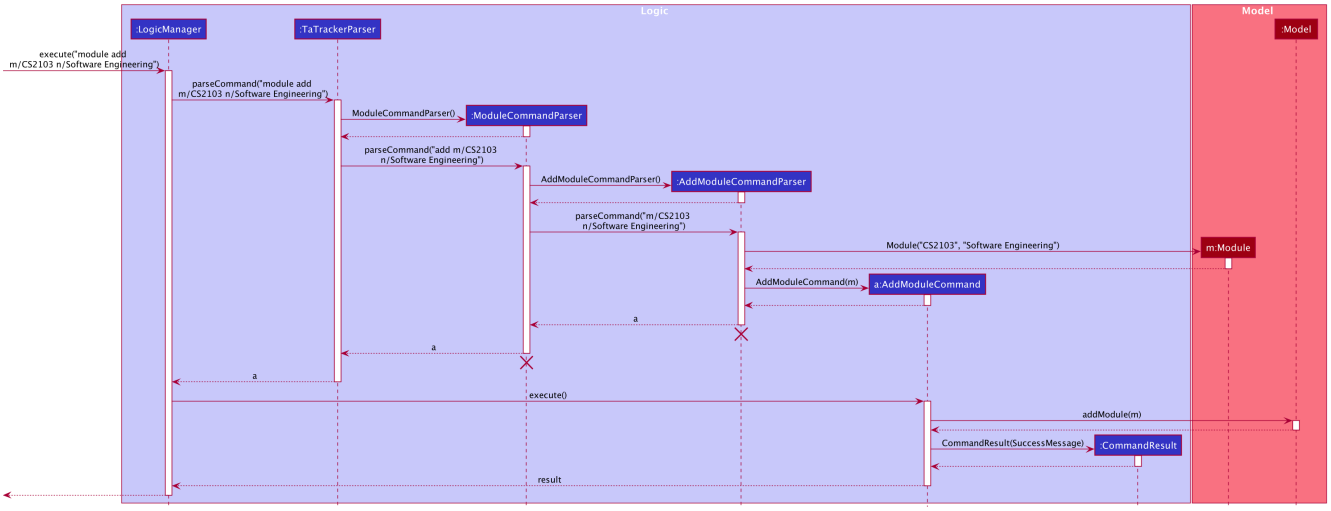


Figure 10. Module Add - Sequence Diagram

1. LogicManager uses the TATrackerParser to first parse the user command.
2. The TATrackerParser sees that the command is of type module and passes the command to the ModuleCommandParser.
3. The ModuleCommandParser sees that the command is of type add and passes the arguments to the AddModuleCommandParser.
4. The AddModuleCommandParser creates a Module with the given module code and name.
5. The AddModuleCommandParser then creates an AddModuleCommand object and passes it the created module. The parser then returns the AddModuleCommand
6. LogicManager calls AddModuleCommad's execute method. The AddModuleCommand object checks whether a module with the given module code already exists in TA-Tracker. If it does, a command exception is thrown saying that a module with the given module code already exists in the TA-Tracker.
7. If no such module exists, the module is added to the TA-Tracker.

The command used to delete a module has been implemented in a similar way. Tha main difference is that the DeleteModuleCommand checks whether an object with the given module code exists in the TA-Tracker. If no such module exists, a command exception is thrown saying that a module with the given module code doesn't exist. If it does exist, the module is removed from the TA-Tracker.

3.2.3. Implementation of the Group Add and Delete Commands

The following activity diagram shows the steps taken by the AddGroupCommand object when the execute method is called.

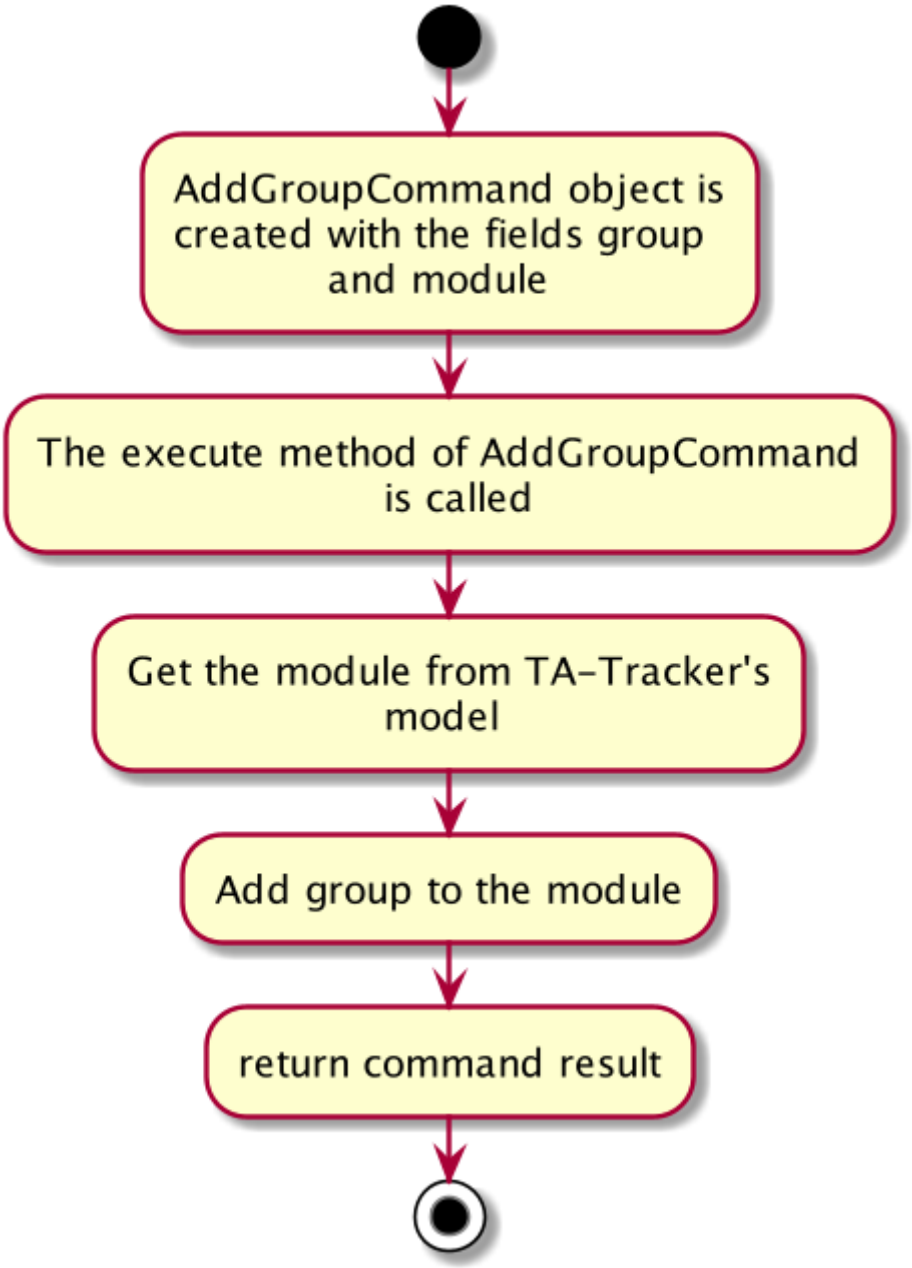


Figure 11. Group Add - Activity Diagram

It should be noted that these are the steps followed assuming that no exception is thrown. Before getting the module from TA-Tracker's model, the DeleteGroupCommand object checks whether such a module even exists. If it doesn't exists, it throws a command exception saying that no such module exists. Before adding a group to the module, the object even checks whether the module already has a group with the given group code. If it exists, a command exception is thrown saying that there is already a group with the given group code.

The interactions between the logic and model components when adding a group are similar to the interactions when deleting a group as shown below.

The following sequence diagram shows the interactions between the logic and model components when the user inputs the command 'group delete m/CS2103 g/G03'.

Note: This diagram is under the case where a group with the group code G03 does exist in the module with module code CS2103 inside the TA-Tracker.

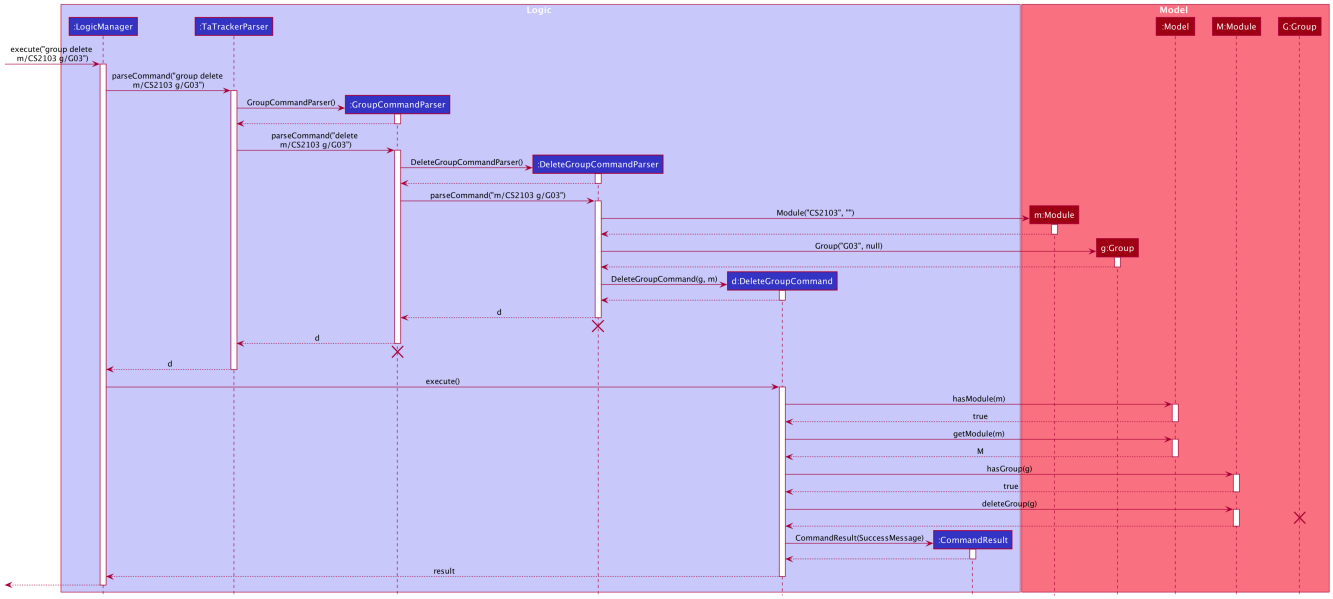


Figure 12. Group Delete - Sequence Diagram

1. LogicManager uses the TATrackerParser to first parse the user command.
2. The TATrackerParser sees that the command is of type group and passes the command to the GroupCommandParser.
3. The GroupCommandParser sees that the command is of type delete and passes the arguments to the DeleteGroupCommandParser.
4. The DeleteGroupCommandParser creates a Module with the given module code and a group with the given group code.
5. The DeleteGroupCommandParser then creates a DeleteGroupCommand object and passes it the created module and group. The parser then returns the DeleteGroupCommand
6. LogicManager calls DeleteGroupCommad's execute method. The DeleteGroupCommand object checks whether a module with the given module code already exists in TA-Tracker. If it doesn't, a command exception is thrown saying that a module with the given module code doesn't exist in the TA-Tracker.
7. If the module exists, the DeletGroupCommand object retrieves the module from the model and checks whether the module has a group with the given group code. If it doesn't, a command exception is thrown saying that no such group exists. If the group does exist, it is removed from the module.

3.2.4. Implementation of the Sort Command

The sort command allows the user to sort the students in the module view either alphabetically or by rating.

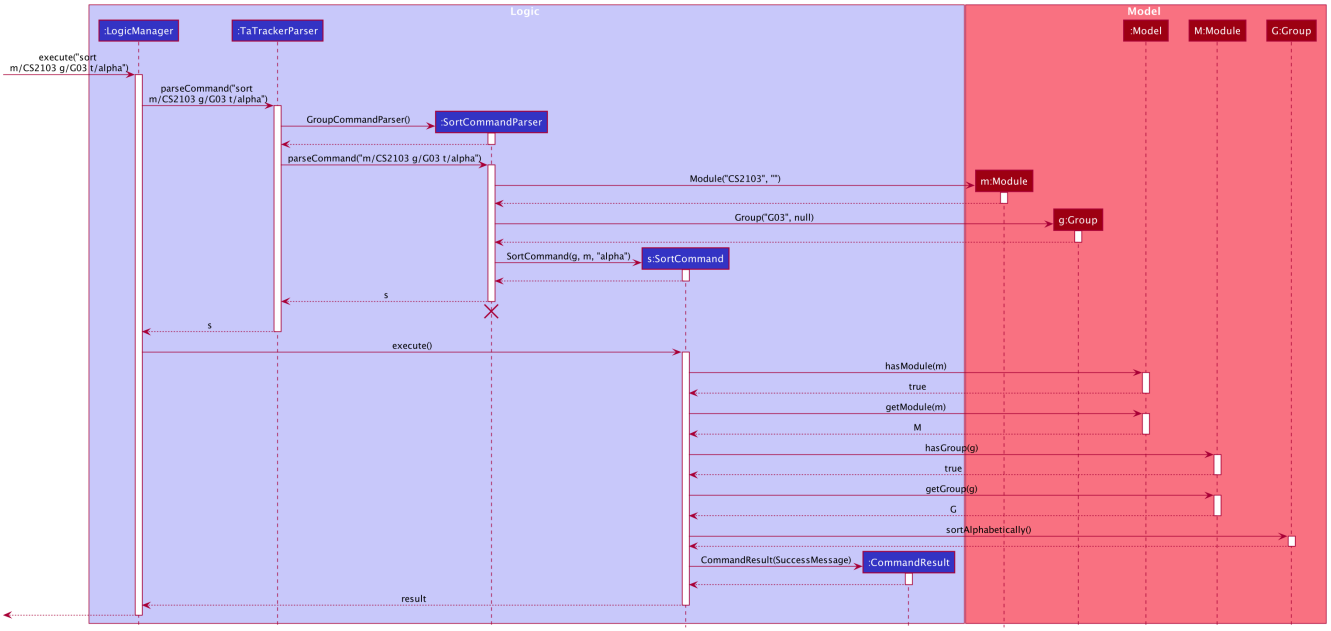


Figure 13. Sort - Sequence Diagram

3.3. Student View

Student view is the term used to characterise the different functionalities related to the students that the user is affiliated with.

3.3.1. Model Framework

The following class diagram shows how different classes are related in the functioning of the module view.

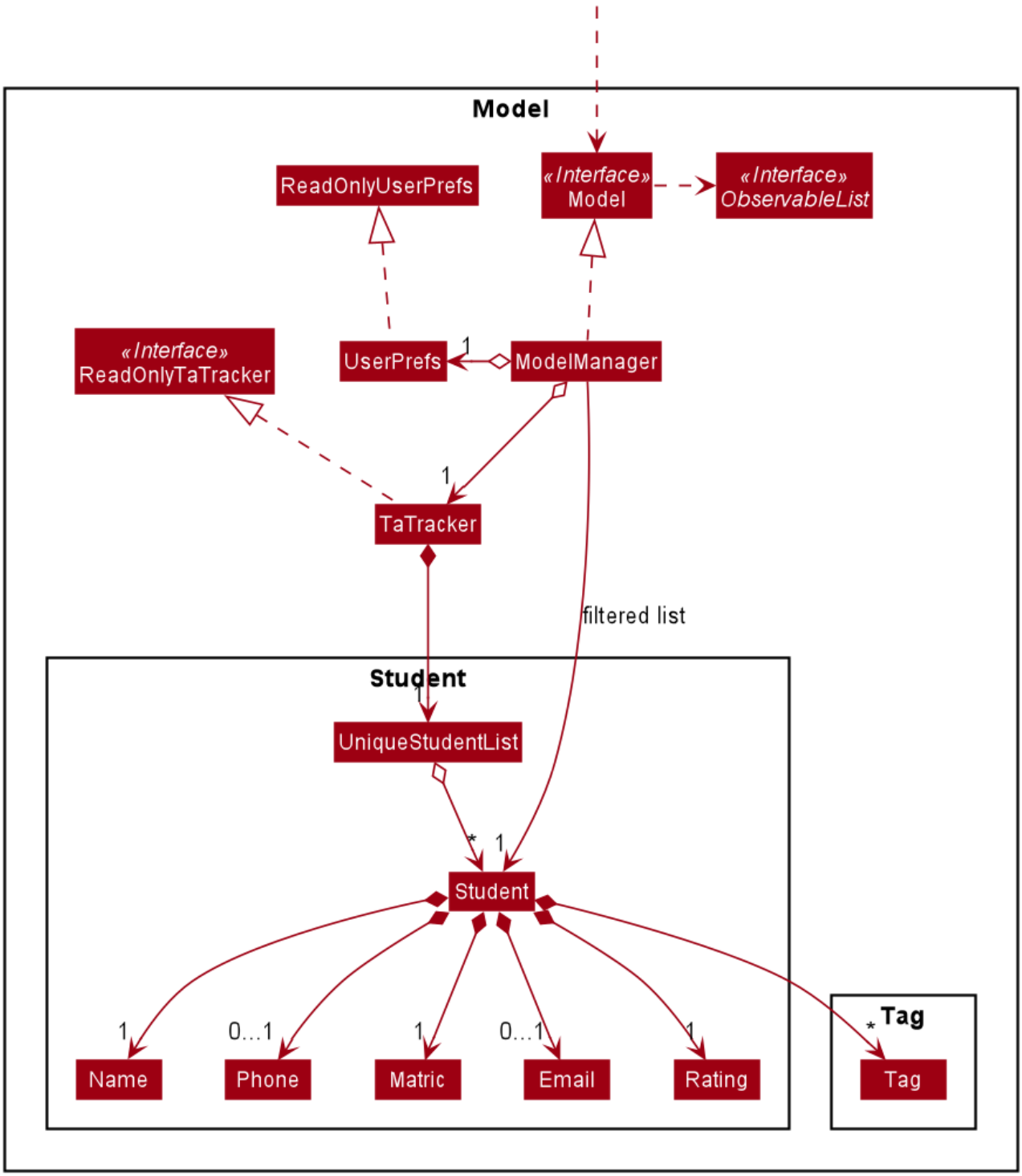


Figure 14. Student View - Class Diagram

The TaTracker model class contains a UniqueModuleList which helps it keep track of the different modules the user is associated with. Each module contains a UniqueGroupList and a UniqueSessionList.

The UniqueGroupList contains a list of all the groups of a module that the user is affiliated with. Each group contains a UniqueStudentsList that contains the students in that group.

The UniqueSessionList contains a list of all the done sessions associated with the module. This list is used in the TSS view.

3.3.2. Implementation of the Module Add and Delete Commands

The following sequence diagram shows the sequence of commands that take place between the logic and model components of the TA-Tracker when the user enters the command 'module add m/CS2103 n/Software Engineering'.

Note: This diagram assumes that there is no module with the module code 'CS2103' pre-existing in the TA-Tracker.

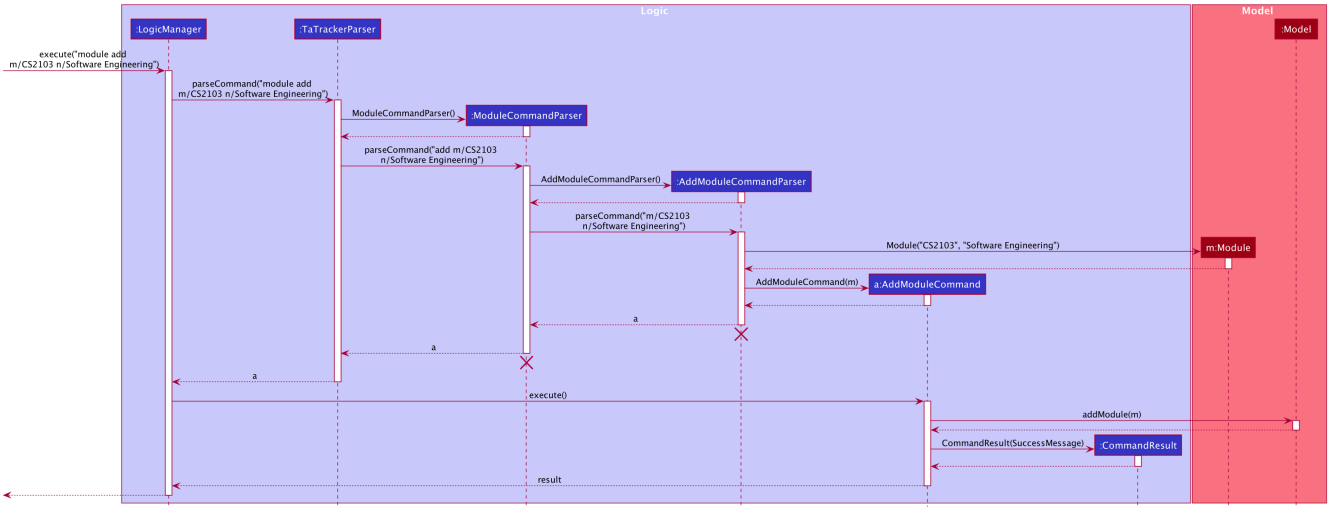


Figure 15. Module Add - Sequence Diagram

1. LogicManager uses the TATrackerParser to first parse the user command.
2. The TATrackerParser sees that the command is of type module and passes the command to the ModuleCommandParser.
3. The ModuleCommandParser sees that the command is of type add and passes the arguments to the AddModuleCommandParser.
4. The AddModuleCommandParser creates a Module with the given module code and name.
5. The AddModuleCommandParser then creates an AddModuleCommand object and passes it the created module. The parser then returns the AddModuleCommand
6. LogicManager calls AddModuleCommad's execute method. The AddModuleCommand object checks whether a module with the given module code already exists in TA-Tracker. If it does, a command exception is thrown saying that a module with the given module code already exists in the TA-Tracker.
7. If no such module exists, the module is added to the TA-Tracker.
8. The SortGroupCommand returns a CommandResult with a success message.

The command used to delete a module has been implemented in a similar way. Tha main difference is that the DeleteModuleCommand checks whether an object with the given module code exists in the TA-Tracker. If no such module exists, a command exception is thrown saying that a module with the given module code doesn't exist. If it does exist, first all the sessions linked to that module are removed. Then the module is removed from the TA-Tracker.

3.3.3. Implementation of the Group Add and Delete Commands

The following activity diagram shows the steps taken by the AddGroupCommand object when the execute method is called.

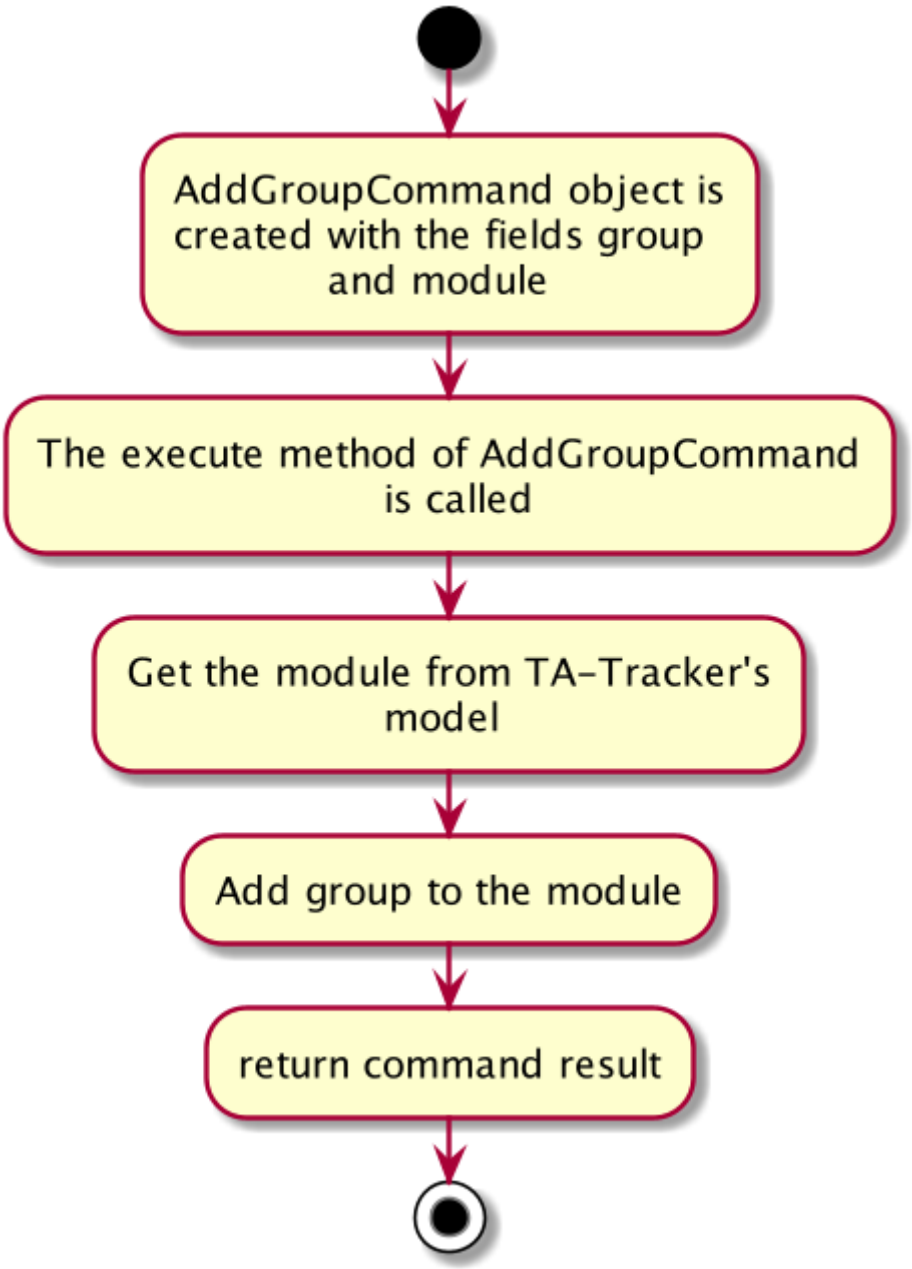


Figure 16. Group Add - Activity Diagram

It should be noted that these are the steps followed assuming that no exception is thrown. Before getting the module from TA-Tracker's model, the DeleteGroupCommand object checks whether such a module even exists. If it doesn't exists, it throws a command exception saying that no such module exists. Before adding a group to the module, the object even checks whether the module already has a group with the given group code. If it exists, a command exception is thrown saying that there is already a group with the given group code.

The interactions between the logic and model components when adding a group are similar to the interactions when deleting a group as shown below.

The following sequence diagram shows the interactions between the logic and model components when the user inputs the command 'group delete m/CS2103 g/G03'.

Note: This diagram is under the case where a group with the group code G03 does exist in the module with module code CS2103 inside the TA-Tracker.

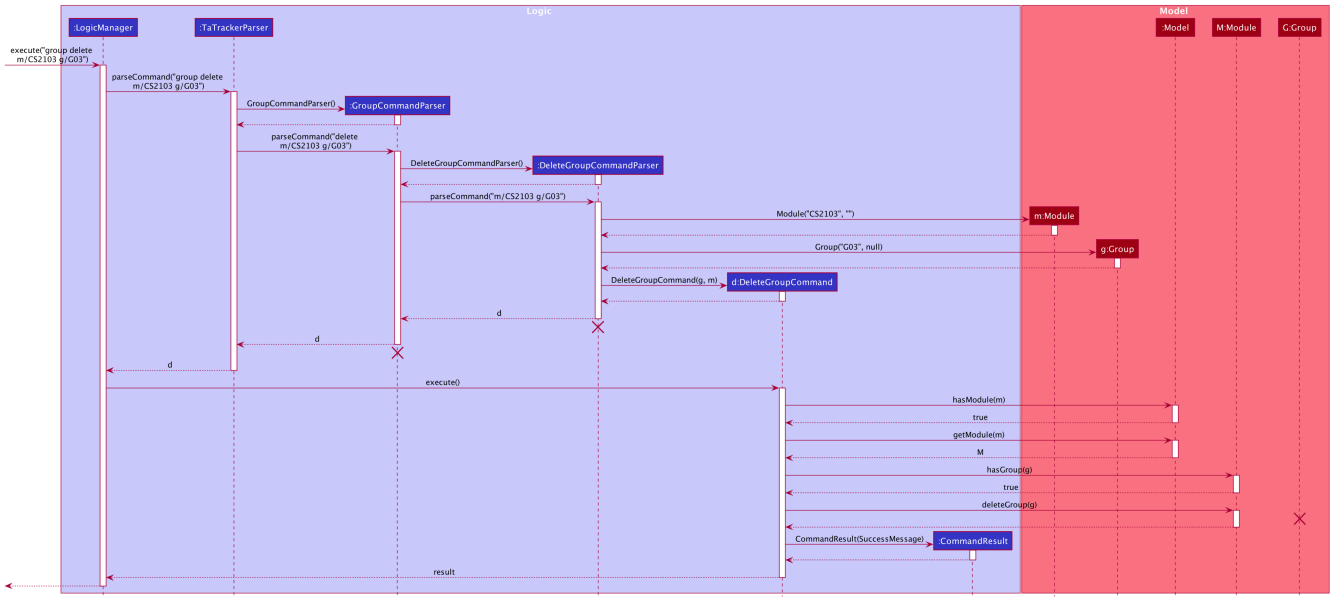


Figure 17. Group Delete - Sequence Diagram

1. LogicManager uses the TATrackerParser to first parse the user command.
2. The TATrackerParser sees that the command is of type group and passes the command to the GroupCommandParser.
3. The GroupCommandParser sees that the command is of type delete and passes the arguments to the DeleteGroupCommandParser.
4. The DeleteGroupCommandParser creates a Module with the given module code and a group with the given group code.
5. The DeleteGroupCommandParser then creates a DeleteGroupCommand object and passes it the created module and group. The parser then returns the DeleteGroupCommand
6. LogicManager calls DeleteGroupCommand’s execute method. The DeleteGroupCommand object checks whether a module with the given module code already exists in TA-Tracker. If it doesn’t, a command exception is thrown saying that a module with the given module code doesn’t exist in the TA-Tracker.
7. If the module exists, the DeleteGroupCommand object retrieves the module from the model and checks whether the module has a group with the given group code. If it doesn’t, a command exception is thrown saying that no such group exists. If the group does exist, it is removed from the module.
8. The SortGroupCommand returns a CommandResult with a success message.

3.3.4. Implementation of the Sort Command

The sort command allows the user to sort the students in the module view either alphabetically or by rating.

The sort command can be used in three ways:

1. sort g/GROUP_CODE m/MODULE_CODE t/TYPE : When a user enters the command in this manner, they are sorting all the students of the given group in the given module by type TYPE (which can be either alphabetical or by rating).
2. sort g/MODULE_CODE t/TYPE : When a user enters a command in this manner, they are sorting all the students of all the groups in the given module by type TYPE (which can be either alphabetical or by rating).
3. sort t/TYPE : When a user enters a command in this manner, they are sorting all students of all groups of all the modules in the TA-Tracker by the type TYPE (which can be either alphabetical or by rating).

Since these sort commands function differently but use the same parser, the following class structure is used.

[SortCommandsClassDiagram] | *SortCommandsClassDiagram.png*

Figure 18. Sort Commands - Class Diagram

Since the different commands use the same parser, the SortCommandParser needs to check which prefixes have been passed and return the appropriate command accordingly. The following activity diagram shows the steps the SortCommandParser takes once its parse command is called (assuming that no exception is thrown).

If the user enters a sort command with no valid prefix, a command exception is thrown thats explains the usage of the sort command.

[SortParserActivityDiagram] | *SortParserActivityDiagram.png*

Figure 19. SortCommandParser - Activity Diagram

The following sequence diagram illustrates the interactions between the logic and model components when the user enters the command 'sort m/CS2103 g/G03 t/alpha'.

Note: To allow the user to type quickly, for type both 'alpha' and 'alphabetically' sort the students lexicographically.

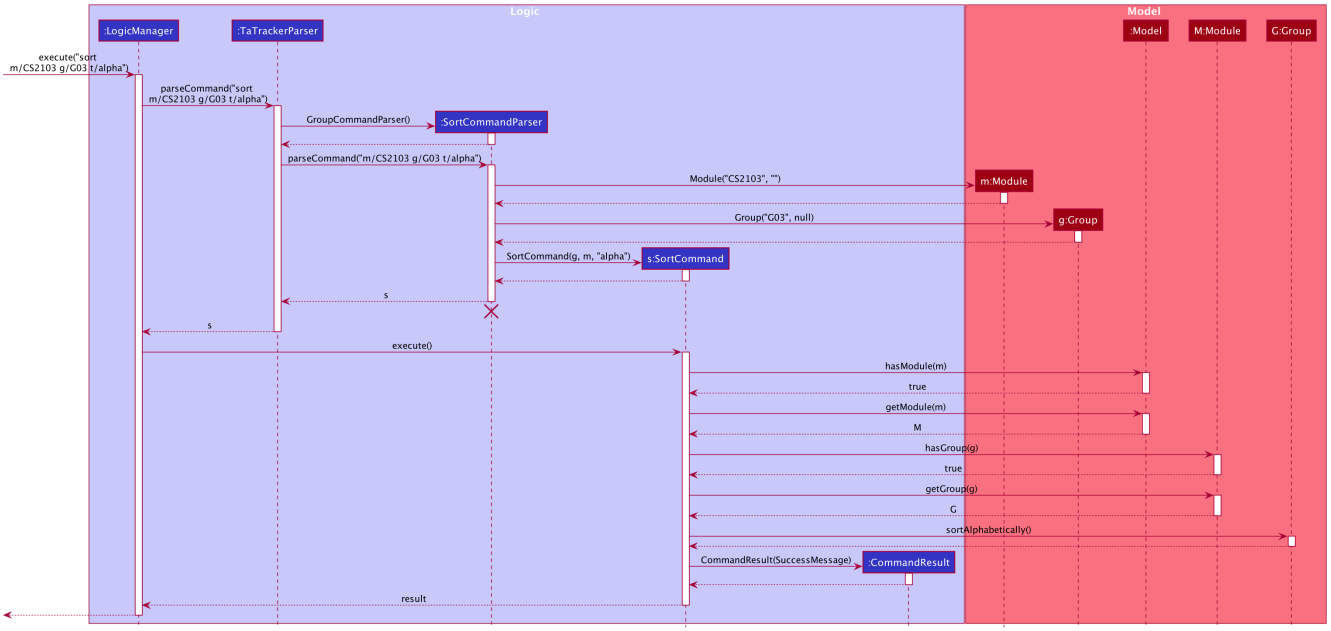


Figure 20. Sort - Sequence Diagram

1. LogicManager uses the TATrackerParser to first parse the user command.
2. The TATrackerParser sees that the command is of type sort and passes the command to the SortCommandParser.
3. The SortCommandParser performs the steps shown in the previous activity diagram and creates and returns a SortGroupCommand.
4. LogicManager calls SortGroupCommand's execute method.
5. The SortGroupCommand creates a Module with the given module code and a group with the given group code. The SortGroupCommand object checks whether a module with the given module code already exists in TA-Tracker. If it doesn't, a command exception is thrown saying that a module with the given module code doesn't exist in the TA-Tracker.
6. If the module exists, the SortGroupCommand object retrieves the module from the model and checks whether the module has a group with the given group code. If it doesn't, a command exception is thrown saying that no such group exists.
7. If the group does exist, it is sorted according to the type of sort specified.
8. The SortGroupCommand returns a CommandResult with a success message.

3.4. [Proposed] Data Encryption

{Explain here how the data encryption feature will be implemented}

3.5. 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 3.6, "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

3.6. Configuration

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

4. Documentation

Refer to the guide here.

5. Testing

Refer to the guide here.

6. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile:

- targets NUS Computing Teaching Assistants
- has a need to track and manage all their claimable hours of teaching
- has a need to keep track of their tasks and reminders (TA-related and/or personal)
- prefer apps on desktop over other platforms
- types quickly and prefers it over mouse
- experiences no discomfort with CLI navigation

Value proposition:

- congregates all information regarding claimable hours of teaching in a single location
- provides desired (TSS) format back to users for convenient viewing

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
* * *	TA	see an overview of events in a week	know what I have that week in a glance
* *	TA	give students ratings	keep a track of student participation in class
* *	TA	delete tasks and events	remove cancelled tasks and events from my session tracker
*	TA	get a message when a new task clashes with an old one	prevent clashes in my schedule
* * *	TA	store and retrieve details of my students	I can get details relating to students whenever necessary
* *	TA	be able to get tasks on a particular date	

Priority	As a …	I want to …	So that I can…
* *	TA	filter by a module	see events relating to a particular module clearly
*	TA	state that a task is recurring	prevent the need to put a recurring task in my schedule each week
* * *	TA	see all my claimable hours in the TSS format	type my claims easily at the end of the semester
* * *	TA	set my hourly rate	get the value of my estimated pay according to the latest rate of the semester
* *	TA	get information on how many hours I've worked so far	keep track of how much work I've done
* * *	user	change between the different pages	view the information on the different pages
* * *	TA	add students to a particular module	
* *	TA	store my students' email ids	retrieve their email ids when I need to contact them
* * *	TA	add multiple modules	keep track of the different modules I am a TA for

Priority	As a …	I want to …	So that I can…
* * *	TA	add a tutorial/lab group	keep track of the different tutorial and lab groups I conduct
* *	TA	delete a tutorial group	remove tasks relating to a tutorial group I am no longer the TA of
* *	TA	delete a module	remove tasks relating to a module I am no longer the TA of
* * *	TA	edit student details	
* * *	TA	remove students from a tutorial or lab group	no longer have details of students that are no longer in my tutorial/lab group
* * *	TA	mark a session as done	automatically get filled in my TSS claim section.
* *	TA	give students ratings	keep a track of student participation in class
* *	TA	delete tasks and events	remove cancelled tasks and events from my session tracker
* * *	TA	schedule consultation sessions with my students	keep track of claimable hours spent in consultations

Priority	As a …	I want to …	So that I can…
*	TA	get a message when a new task clashes with an old one	prevent clashes in my schedule
* * *	TA	store and retrieve details of my students	I can get details relating to students whenever necessary
* *	TA	be able to get tasks on a particular date	
* *	TA	filter by a module	see events relating to a particular module clearly
*	TA	state that a task is recurring	prevent the need to put a recurring task in my schedule each week
* * *	TA	see all my claimable hours in the TSS format	type my claims easily at the end of the semester
* * *	TA	set my hourly rate	get the value of my estimated pay according to the latest rate of the semester
* *	TA	get information on how many hours I've worked so far	keep track of how much work I've done
* * *	user	change between the different pages	view the information on the different pages

Priority	As a …	I want to …	So that I can…
* * *	TA	add students to a particular module	
* *	TA	store my students' email ids	retrieve their email ids when I need to contact them
* * *	TA	add multiple modules	keep track of the different modules I am a TA for
* * *	TA	add a tutorial/lab group	keep track of the different tutorial and lab groups I conduct
* *	TA	delete a tutorial group	remove tasks relating to a tutorial group I am no longer the TA of
* *	TA	delete a module	remove tasks relating to a module I am no longer the TA of
* * *	TA	edit student details	
* * *	TA	remove students from a tutorial or lab group	no longer have details of students that are no longer in my tutorial/lab group
* * *	TA	mark a session as done	automatically get filled in my TSS claim section.
*	user	change the default view of the application	

Appendix C: Use Cases

(For all use cases below, the **System** is the `TA-Tracker` and the **Actor** is the `user`, unless specified otherwise)

Use case: Viewing a page

MSS

1. User requests to view a different page.
2. TA-Tracker layout changes to show the new page.

   Use case ends.

Extensions

- 1a. The requested page is invalid.

  1a1. TA-Tracker shows an error message.

  Use case resumes at step 1.

Use case: Viewing the help menu

MSS

1. User requests to view the help menu.
2. TA-Tracker shows the list of commands.

   Use case ends.

Use case: Change default view

MSS

1. User requests to change the default view to a specified page.
2. TA-Tracker changes the default view.
3. TA-Tracker shows the default view.

   Use case ends.

Extensions

- 1a. The given page is invalid.

  1a1. TA-Tracker shows an error message.

Use case resumes at step 1.

Use case: Change the hourly pay rate

MSS

1. User requests to change the hourly pay rate to a specified amount.
2. TA-Tracker changes the pay rate.
3. TA-Tracker shows an edited TSS claims page the total pay adjusted to reflect the new pay rate.

   Use case ends.

Extensions

1a. The given rate is invalid.

1a1. TA-Tracker shows an error message.

Use case resumes at step 1.

Use case: Add student

MSS

1. User requests to add a student.
2. TA-Tracker adds new student.
3. TA-Tracker layout changes to show the student list page.

   Use case ends.

Extensions

- 1a. The input required (eg. Matric Number) to add a student is invalid.

  1a1. TA-Tracker shows an error message.

  Use case resumes at step 1.

Use case: Add module

MSS

1. User requests to add a new module.
2. TA-Tracker adds a new module.
3. TA-Tracker layout changes to show the session list page.

   Use case ends.

Extensions

- 1a. The given module code is invalid.

    1a1. TA-Tracker shows an error message.

    Use case resumes at step 1.

Use case: Add tutorial

MSS

1. User requests to add a new tutorial.
2. TA-Tracker shows adds a new tutorial linked to the specified module.
3. TA-Tracker layout changes to show the session list page.

   Use case ends.

Extensions

1a. The given module code is invalid.

    1a1. TA-Tracker shows an error message.

    Use case resumes at step 1.

1a. The given class code is invalid.

    1a1. TA-Tracker shows an error message.

    Use case resumes at step 1.

Use case: Edit Student

MSS

1. User requests to list students.
2. TA-Tracker shows a list of students.
3. User requests to edit a specific student in the list.
4. TA-Tracker edits the student according to the specified parameters.

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given matric number is invalid.

3a1. TA-Tracker shows an error message.

Use case resumes at step 3.

3a. The given new input for the parameter(s) are invalid.

3a1. TA-Tracker shows an error message.

Use case resumes at step 3.

Use case: Delete student

MSS

1. User requests to show students page.
2. TA-Tracker shows a list of students categorised by tutorial.
3. User requests to delete a specific student in the list.
4. TA-Tracker deletes the student.

Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given matric number is invalid.

3a1. TA-Tracker shows an error message.

Use case resumes at step 3.

Use case: Delete module

MSS

1. User requests to show sessions page.
2. TA-Tracker shows a list of sessions categorised by modules.
3. User requests to delete a specific module in the list.
4. TA-Tracker deletes the module and all of the sessions and tutorials in it.

Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given module code is invalid.

3a1. TA-Tracker shows an error message.

Use case resumes at step 3.

Use case: Delete tutorial

MSS

1. User requests to show students page.
2. TA-Tracker shows a list of students categorised by tutorial.
3. User requests to delete a specific tutorial in the list.
4. TA-Tracker deletes the tutorial and all of the students in it.

Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given class code is invalid.

3a1. TA-Tracker shows an error message.

Use case resumes at step 3.

Use case: Add session

MSS

1. User requests to add a session.
2. TA-Tracker creates the new session.
3. TA-Tracker adds the new session into the corresponding session group.
4. TA-Tracker switches to the Schedule View in order to display the new session.

Use case ends.

Extensions

1a. The user requests to add a recurring session.

1a1. TA-Tracker creates a new session, and labels it as recurring.

Use case resumes at step 3.

2a. The module does not exist.

2a1. TA-Tracker shows an error message.

Use case resumes at step 2.

2b. The session group does not exist.

2b1. TA-Tracker shows an error message.

Use case resumes at step 2.

Use case: Edit session

MSS

1. User requests to view the Schedule View.
2. TA-Tracker switches to the Schedule View.
3. User requests to edit a specific session in the view.
4. TA-Tracker edits the session.
5. TA-Tracker replaces the session in the current view with the new version.

Use case ends.

Extensions

2a. The Schedule View is empty.

Use case ends.

3a. The given session UID is invalid.

- 3a1. TA-Tracker shows an error message.

Use case resumes at step 2.

Use case: Delete session

MSS

1. User requests to view the Schedule View.
2. TA-Tracker switches to the Schedule View.
3. User requests to delete a specific session in the view.

4. TA-Tracker deletes the session.
5. TA-Tracker removes the session from the current view.

   Use case ends.

Extensions

   2a. The Schedule View is empty.

   Use case ends.

   3a. The given session UID is invalid.

   - 3a1. TA-Tracker shows an error message.

     Use case resumes at step 2.

Use case: Mark a session as done

MSS

1. User requests to view the Schedule View.
2. TA-Tracker switches to the Schedule View.
3. User requests to mark a specific session in the view as done.
4. TA-Tracker marks the session as done.
5. TA-Tracker shows a tick next to the session in the current view.

   Use case ends.

Extensions

   2a. The Schedule View is empty.

   Use case ends.

   3a. The given session UID is invalid.

   - 3a1. TA-Tracker shows an error message.

     Use case resumes at step 2.

Use case: Enroll student into class

MSS

1. User requests to enroll a student in a session group.
2. TA-Tracker registers the student in the session group.

3. TA-Tracker switches to the Student View.
4. TA-Tracker shows the student in the student list for the session group.

   Use case ends.

Extensions

2a. The student does not exist.

   2a1. TA-Tracker shows an error message.

   Use case resumes at step 2.

2b. The module does not exist.

   2b1. TA-Tracker shows an error message.

   Use case resumes at step 2.

2c. The session group does not exist.

   2c1. TA-Tracker shows an error message.

   Use case resumes at step 2.

Use case: Kick student from class

MSS

1. User requests to withdraw a student from a session group.
2. TA-Tracker removes the student from the session group.
3. TA-Tracker switches to the Student View.
4. TA-Tracker shows that the student is removed from the student list for the session group.

   Use case ends.

Extensions

2a. The student does not exist.

   2a1. TA-Tracker shows an error message.

   Use case resumes at step 2.

2b. The module does not exist.

   2b1. TA-Tracker shows an error message.

   Use case resumes at step 2.

2c. The session group does not exist.

2c1. TA-Tracker shows an error message.

Use case resumes at step 2.

Use case: Filter information for a specific module

MSS

1. User requests to filter information for a specific module.
2. TA-Tracker hides unrelated information from the current view.

Use case ends.

Extensions

1a. The module does not exist.

1a1. TA-Tracker shows an error message.

Use case resumes at step 2.

Use case: Find sessions matching a keyword

MSS

1. User requests to find sessions related to a specific keyword.
2. TA-Tracker retrieves a list of sessions containing the keyword in any of their fields.
3. TA-Tracker shows the list of sessions.

Use case ends.

Extensions

2a. The search did find any matches.

2a1. TA-Tracker shows an empty list.

Use case resumes at step 2.

Use case: Exit the app

MSS

1. User requests to exit the app.
2. App window closes.

Use case ends.

Appendix D: Non Functional Requirements

1. `TAT` should be able to run on any mainstream OS as long as it has `Java 11` installed.
2. 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.
3. `TAT` should be able to run with or without internet connection.
4. `TAT` should work for a single user only.
5. `TAT` should not require user to install.
6. Features implemented should be testable using manual testing and automated testing.
7. `TAT` should support screen resolution of 1920 x 1080 or higher.
8. `TAT` should support the English locale only. Any locale from this link that starts with "en" will be supported.

Appendix E: Glossary

TAT	Stands for "Teaching Assistant Tracker". It is the application this developer guide is for.
TA	Stands for "Teaching Assistant", and in our context limited to undergraduate and graduate teaching assistants in the National University of Singapore. A teaching assistant is an individual who assists a teacher with instructional responsibilities such as holding tutorials, labs, consultations, etc.
NUS	Stands for "National University of Singapore".
Module	Refers to one of multiple academic courses in NUS.
Tutorial	A tutorial is a regular meeting between a tutor and one or several students, for discussion of a subject that is being studied.
API	Stands for "Application Programming Interface" which simplifies programming by abstracting the underlying implementation and only exposing objects or actions the developer needs.
Locale	Stands for a setting on the user's computer that defines the user's language and region.

PlantUML	Stands for a software tool that we use to render the diagrams used in this document.
NFR	Stands for "Non-functional Requirement"
Mainstream OS	Stands for commonly used Operating Systems (OS) such as Windows, Linux, Unix, OS-X
Regular English Text	Stands for text with ordinary english grammar structures and vocabulary generally used by the public. It excludes syntax related to programming and system administration.
System Administration	Stands for the field of work in which someone manages one or more systems, be they software, hardware, servers or workstations with the goal of ensuring the systems are running efficiently and effectively.
MSS	Stands for Main Success Scenario that describes the 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 student

1. Deleting a student while all students are listed
    a. Prerequisites: List all students using the `list` command. Multiple students 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 student 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 … }