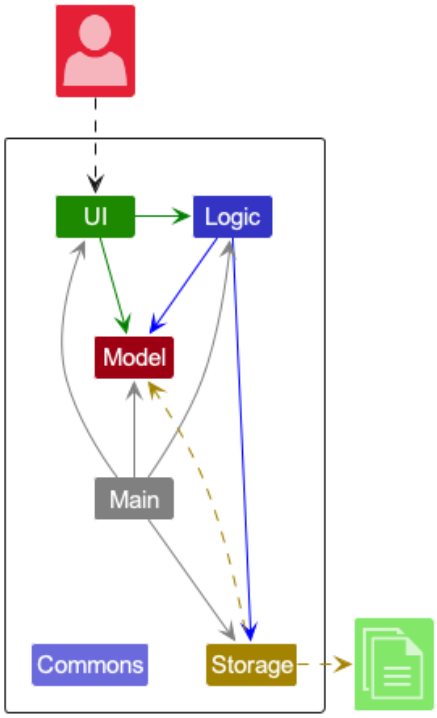
TeamTracker Developer Guide

Setting up, getting started

Refer to the guide *Setting up and getting started*.

Design

Architecture



The ***Architecture Diagram*** given above explains the high-level design of the App.

Given below is a quick overview of main components and how they interact with each other.

Main components of the architecture

`Main` (consisting of classes `Main` and `MainApp`) is in charge of the app launch and shut down.

- At app launch, it initializes the other components in the correct sequence, and connects them up with each other.
- At shut down, it shuts down the other components and invokes cleanup methods where necessary.

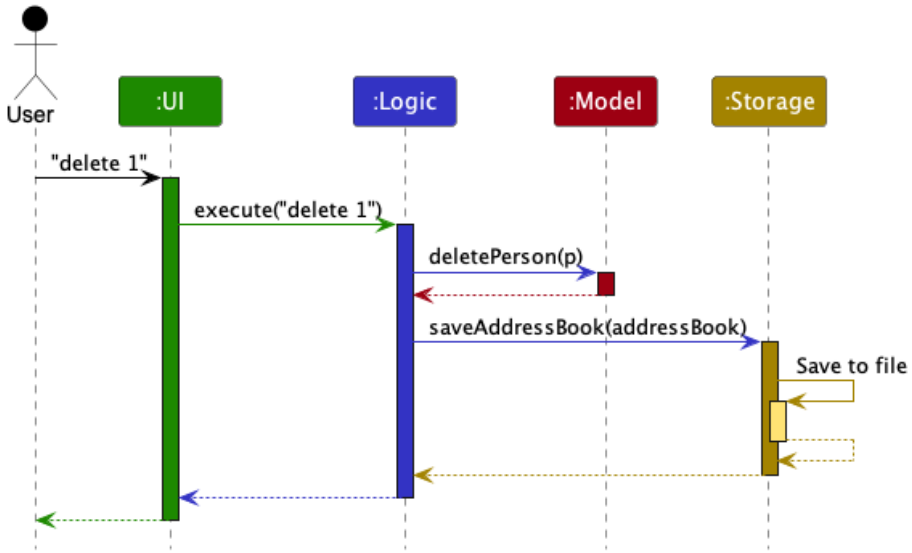
The bulk of the app's work is done by the following four components:

- `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.

`Commons` represents a collection of classes used by multiple other components.

How the architecture components interact with each other

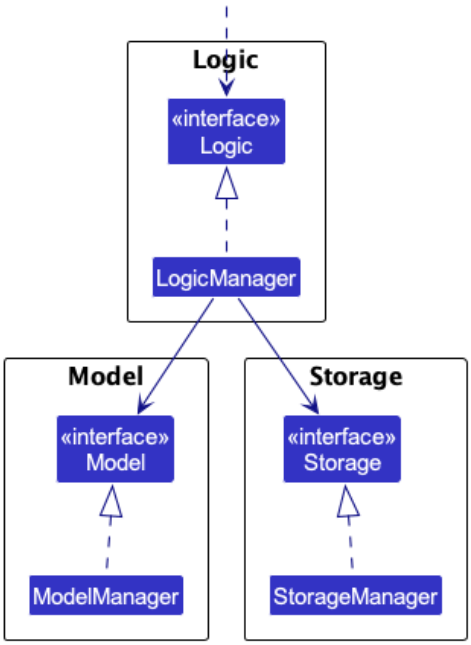
The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `delete 1`.



Each of the four main components (also shown in the diagram above),

- defines its *API* in an `interface` with the same name as the Component.
- implements its functionality using a concrete `{Component Name}Manager` class (which follows the corresponding API `interface` mentioned in the previous point.

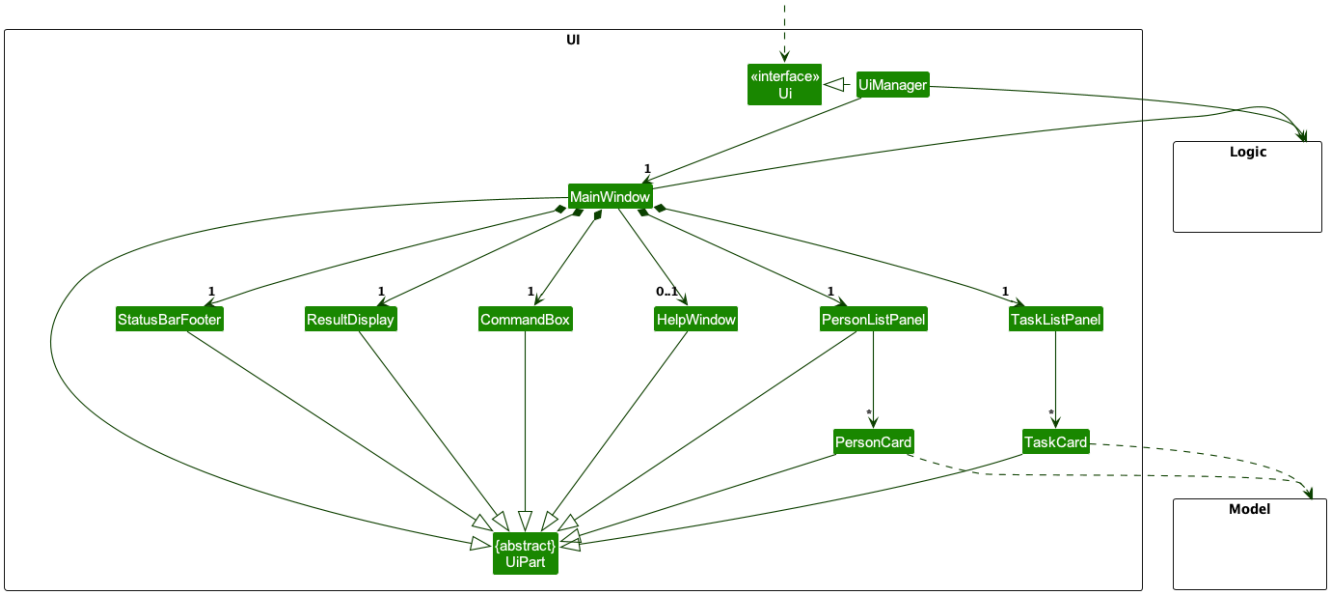
For example, the `Logic` component defines its API in the `Logic.java` interface and implements its functionality using the `LogicManager.java` class which follows the `Logic` interface. Other components interact with a given component through its interface rather than the concrete class (reason: to prevent outside component's being coupled to the implementation of a component), as illustrated in the (partial) class diagram below.



The sections below give more details of each component.

UI component

The **API** of this component is specified in `Ui.java`



The UI consists of a `MainWindow` that is made up of parts e.g. `CommandBox`, `ResultDisplay`, `PersonListPanel`, `StatusBarFooter`, `TaskListPanel` etc. All these, including the `MainWindow`, inherit from the abstract `UiPart` class which captures the commonalities between classes that represent parts of the visible GUI.

The `UI` component uses the 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`

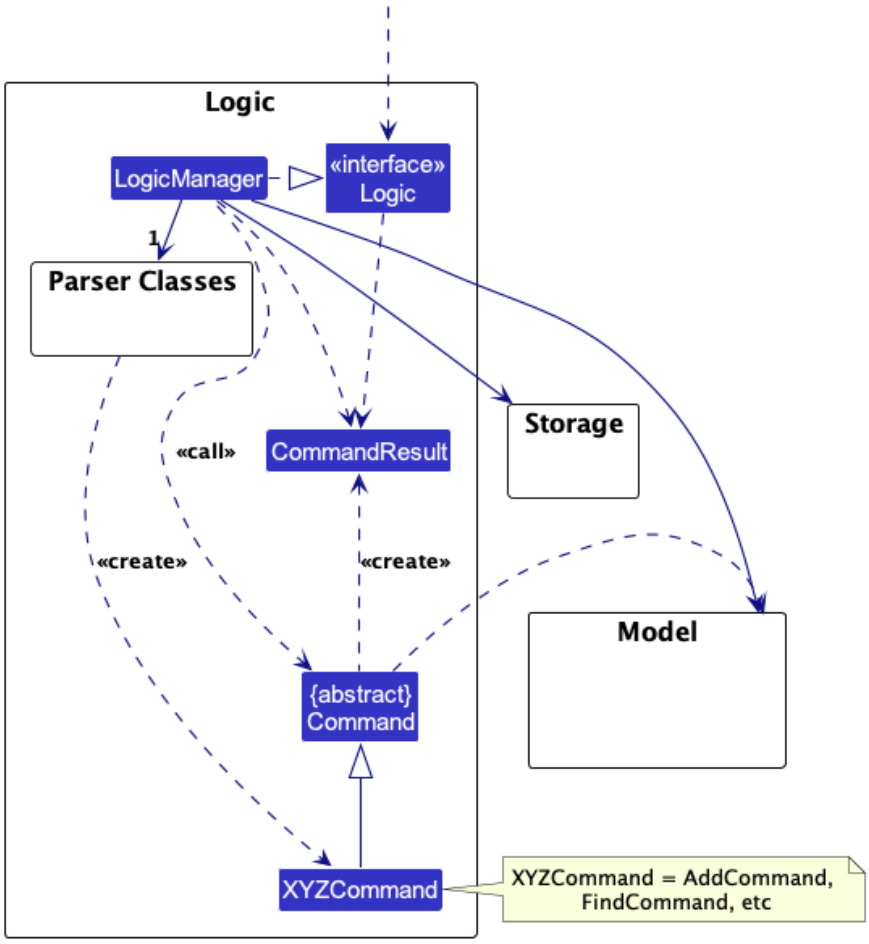
The `UI` component,

- executes user commands using the `Logic` component.
- listens for changes to `Model` data so that the UI can be updated with the modified data.
- keeps a reference to the `Logic` component, because the `UI` relies on the `Logic` to execute commands.
- depends on some classes in the `Model` component, as it displays `Person` object residing in the `Model`.

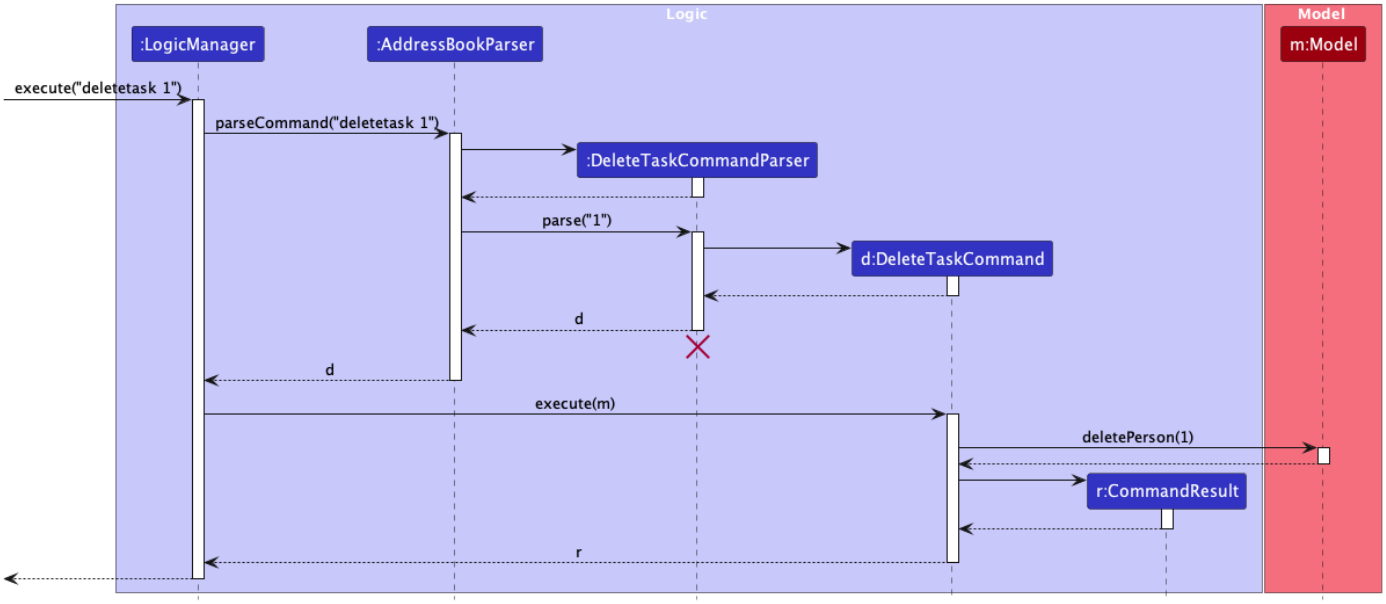
Logic component

API : `Logic.java`

Here's a (partial) class diagram of the `Logic` component:



The sequence diagram below illustrates the interactions within the `Logic` component, taking `execute("delete 1")` API call as an example.

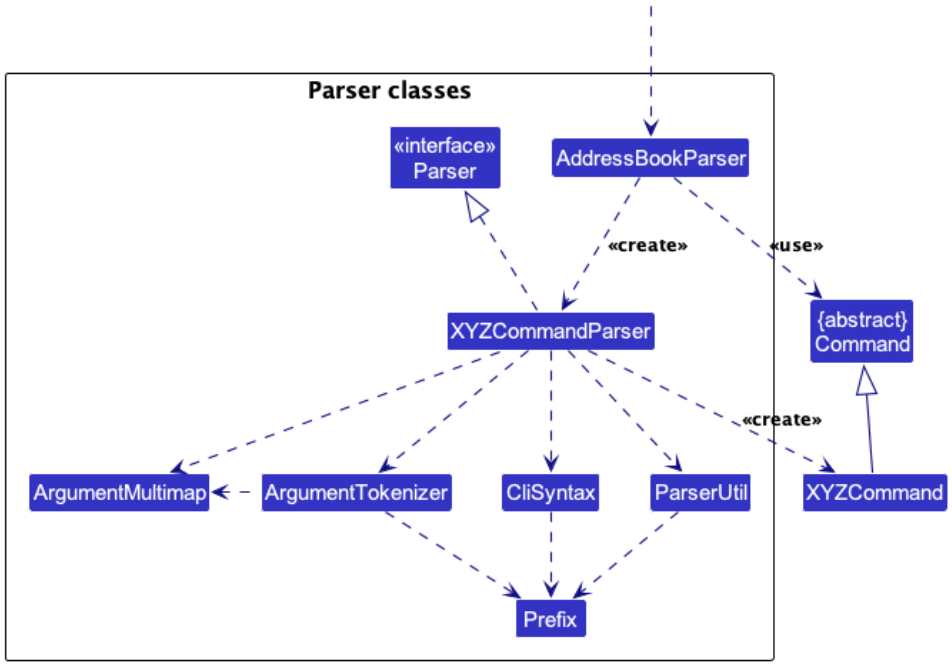


> **Note:** The lifeline for `DeleteTaskCommandParser` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline continues till the end of diagram.

How the `Logic` component works:

1. When `Logic` is called upon to execute a command, it is passed to an `AddressBookParser` object which in turn creates a parser that matches the command (e.g., `DeleteTaskCommandParser`) and uses it to parse the command.
2. This results in a `Command` object (more precisely, an object of one of its subclasses e.g., `DeleteTaskCommand`) which is executed by the `LogicManager`.
3. The command can communicate with the `Model` when it is executed (e.g. to delete a task).
   Note that although this is shown as a single step in the diagram above (for simplicity), in the code it can take several interactions (between the command object and the `Model`) to achieve.
4. The result of the command execution is encapsulated as a `CommandResult` object which is returned back from `Logic`.

Here are the other classes in `Logic` (omitted from the class diagram above) that are used for parsing a user command:

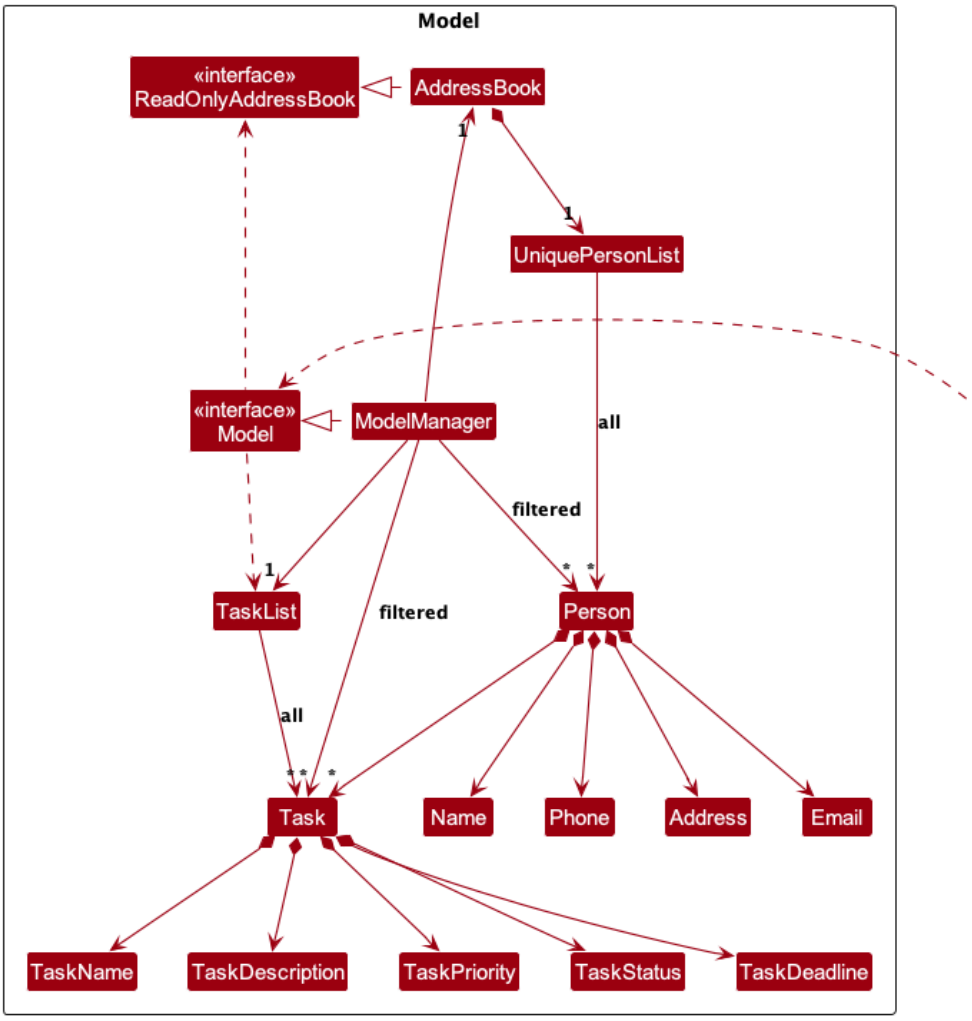


How the parsing works:

- When called upon to parse a user command, the `AddressBookParser` class creates an `XYZCommandParser` (`XYZ` is a placeholder for the specific command name e.g., `AddTaskCommandParser`) which uses the other classes shown above to parse the user command and create a `XYZCommand` object (e.g., `AddTaskCommand`) which the `AddressBookParser` returns back as a `Command` object.
- All `XYZCommandParser` classes (e.g., `AddTaskCommandParser`, `DeleteTaskCommandParser`, ...) inherit from the `Parser` interface so that they can be treated similarly where possible e.g, during testing.

Model component

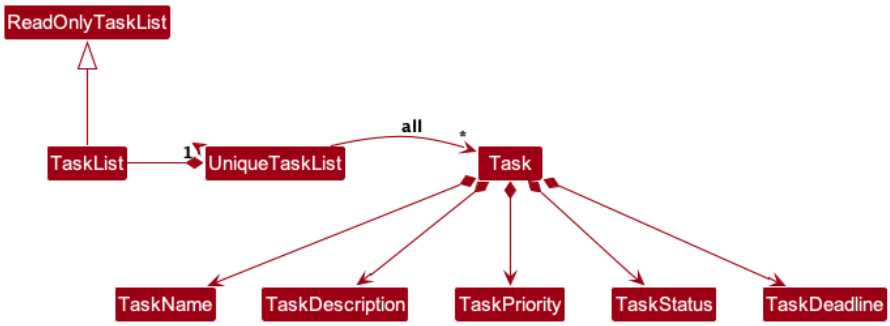
API : `Model.java`



The `Model` component,

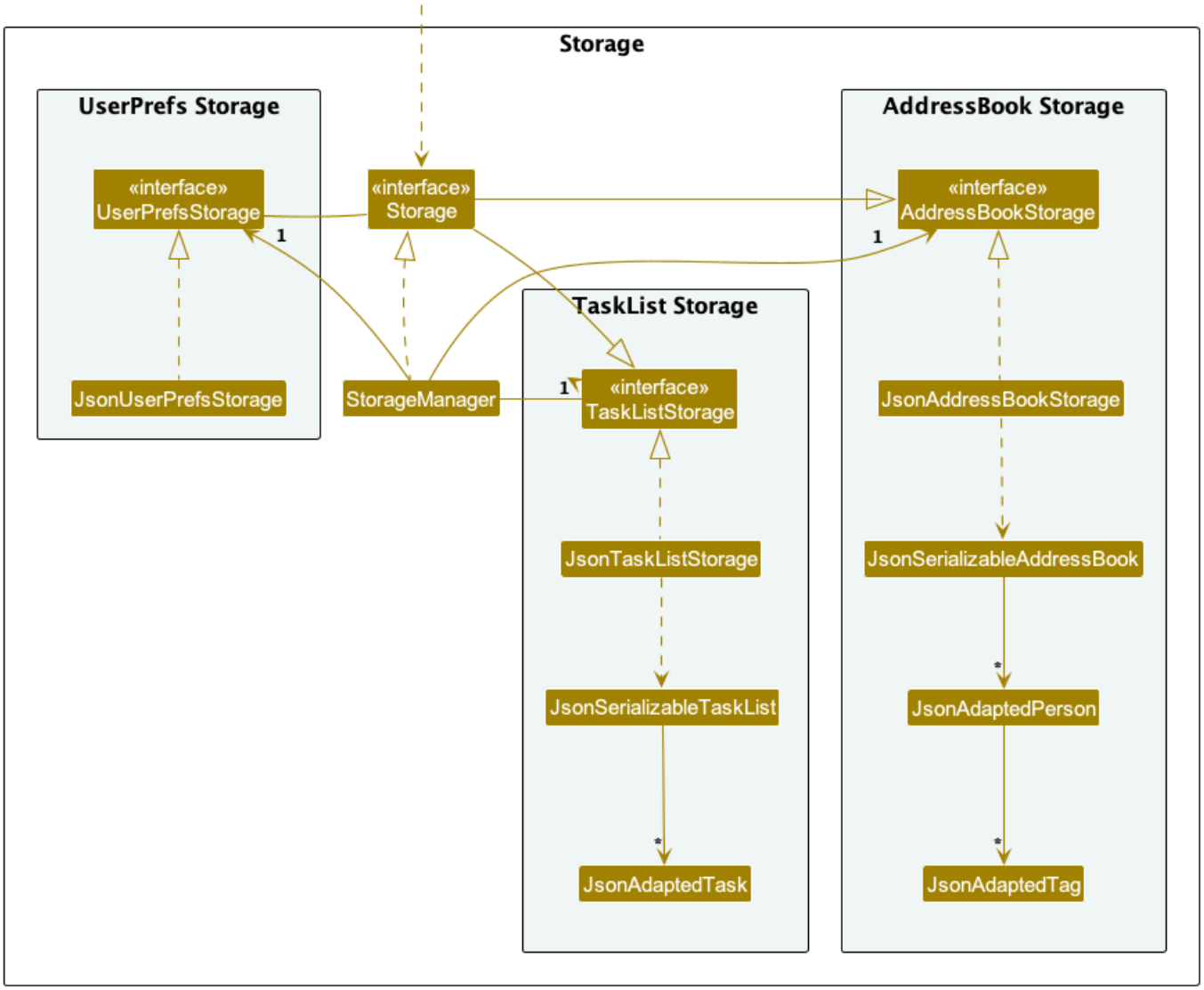
- stores the address book data i.e., all `Person` objects (which are contained in a `UniquePersonList` object).
- stores the currently 'selected' `Person` objects (e.g., results of a search query) as a separate *filtered* list which is exposed to outsiders as an unmodifiable `ObservableList<Person>` 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.
- stores the task list data i.e., all `Task` objects (which are contained in a `TaskList` object).
- stores the currently 'selected' `Task` objects (e.g., results of a search query) as a separate *filtered* list which is exposed to outsiders as an unmodifiable `ObservableList<Task>` 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.
- stores a `UserPref` object that represents the user's preferences. This is exposed to the outside as a `ReadOnlyUserPref` objects.
- does not depend on any of the other three components (as the `Model` represents data entities of the domain, they should make sense on their own without depending on other components)

Note: An alternative (arguably, a more OOP) model is given below. `TaskList` implements the `ReadOnlyTaskList` interface, and has a `UniqueTaskList` that contains all `Task` s. This allows `TaskList` to be implemented in a way that is consistent to how `AddressBook` is implemented, thus any benefits arising from the design decisions of `Person` also applies to `Task` . We are currently not adopting this model due to time constraints and the benefits are not immediately obvious.


ReadOnlyTaskList
TaskList
1
UniqueTaskList
all
*
Task
TaskName
TaskDescription
TaskPriority
TaskStatus
TaskDeadline

Storage component

API : `Storage.java`



The `Storage` component,

- can save address book data, task list data and user preference data in JSON format, and read them back into corresponding objects.
- inherits from AddressBookStorage, TaskListStorage and UserPrefStorage, which means it can be treated as one of the three classes (if only the functionality of only one is needed).
- depends on some classes in the `Model` component (because the `Storage` component's job is to save/retrieve objects that belong to the `Model`)

Common classes

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

Implementation

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

[Proposed] Grouping Feature

Proposed Implementation

The proposed grouping mechanism is facilitated by `GroupedUniquePersonList`. It extends `UniquePersonList` with task that are linked between the people of the same group. Additionally, it implements the following operations:

- `GroupedUniquePersonList#assignTask(Task)` - Add task to everyone in the group
- `GroupedAddressBook#markTask(Index)` - Mark task of everyone in the group
- `GroupedAddressBook#unmarkTask(Index)` - Unmark task of everyone in the group

These operations are exposed in the `Model` interface as `Model#assignTaskToGroup(String, Task)`, `Model#markTaskOfGroup(String, Index)` and `Model#unmarkTaskofGroup(String, Index)` respectively.

`GroupedUniquePersonList` adds a new string called `groupName` to label each of their groups.

A new list of `GroupedUniquePersonList` will be added to the `Model` interface.

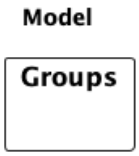
To add to the list of `GroupedUniquePersonList`, the Model interface includes `Model#addGroup(String, List<Person>)` and `Model#addListOfGroups(List<Group>)`.

To remove to the list of `GroupedUniquePersonList`, the Model interface includes `Model#removeGroup(String)`.

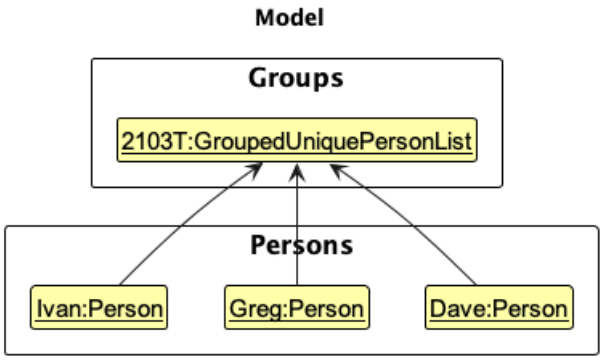
New operation are exposed in the `Model` interface are `Model#addPersonToGroup(String, Person)`, `Model#removePersonFromGroup(String, Person)` and `Model#deleteAssignedTaskGroup(String, Task)` which would call `UniquePersonList#add(Person)`, `UniquePersonList#remove(Person)` and `UniquePersonList#deleteAssignedTask(Person)` respectively.

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

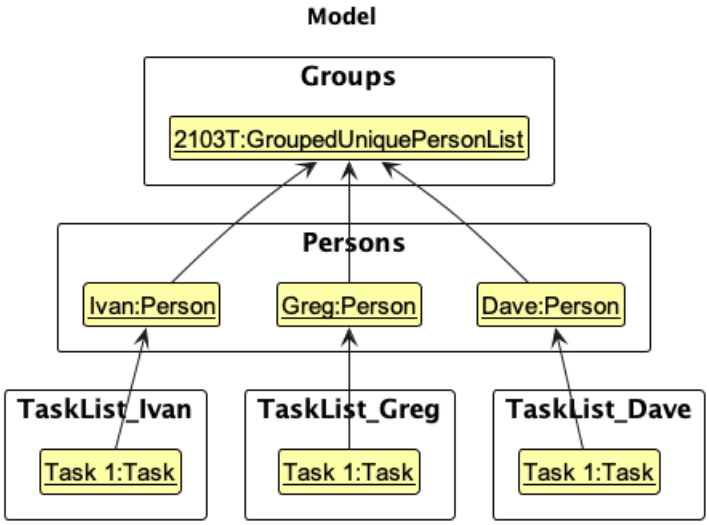
Step 1. The user launches the application for the first time. The list of the `GroupedUniquePersonList` will be empty if there are no groups stored in the storage.



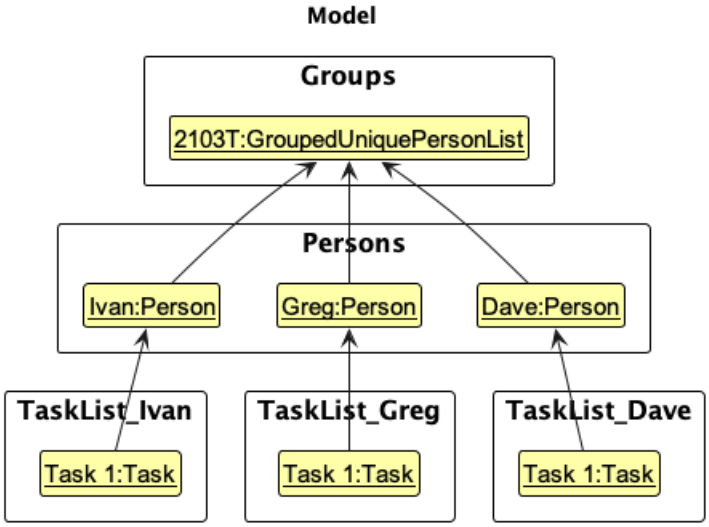
Step 2. The user executes `group gn/2103T gp/Ivan gp/Greg gp/Dave` command to group Ivan, Greg and Dave from the displayed person list to one group. The `group` command calls `Model#addGroup(String, List<Person>)`, which creates a new group with that contains the list of people that was indicated by the user.



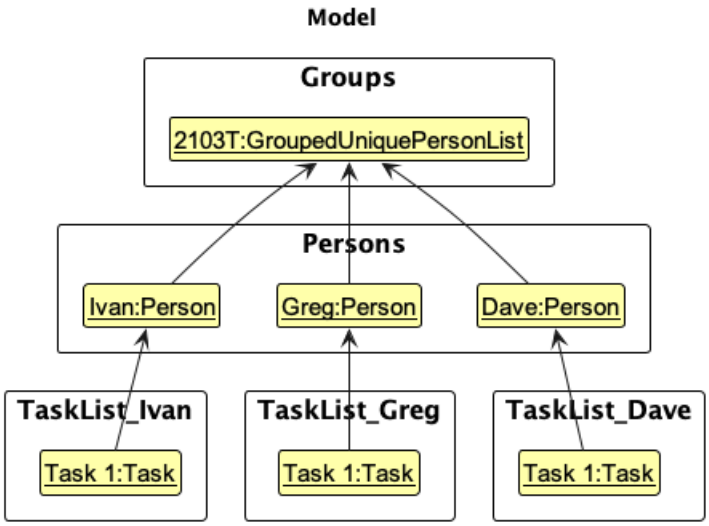
Step 3. The user executes `assigngroup gn/2103T gt/Task 1` command to assign a task named "Task 1" to the group named "2103T" from the group list. The `assigngroup` command calls `Model#assignTaskToGroup(String, Task)`, which finds the group with the same name and assign that task to everyone that is in the group.



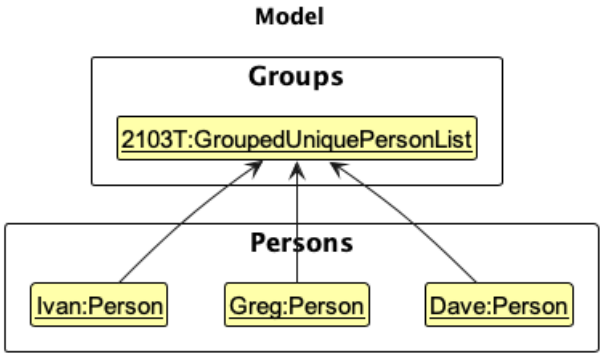
Step 4. The user executes `addpersontogroup gn/2103T gp/Bob` command to add Bob to the group named "2103T" from the group list. The `addpersontogroup` command calls `Model#addPersonToGroup(String, Person)`, which finds the group with the same name and add the person to the group.



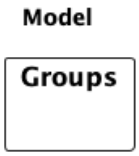
Step 5. The user executes `removepersonfromgroup gn/2103T gp/4` command to remove Bob from the group named "2103T" from the group list. The `removepersonfromgroup` command calls `Model#removePersonFromGroup(String, Person)`, which finds the group with the same name and remove the person to the group.



Step 6. The user executes `deletetaskgroup gn/2103T gt/Task 1` command to remove a task named "Task 1" from the group named "2103T" from the group list. The `deletetaskgroup` command calls `Model#deleteAssignedTaskGroup(String, Task)`, which finds the group with the same name and remove that task from everyone that is in the group.



Step 7. The user executes `deletegroup gn/2103T` command to remove the group from the list. The `deletegroup` command calls `Model#removeGroup(String)`, which finds the group with the same name and remove that group from the list.



[Proposed] Undo/Redo Feature

The proposed undo/redo mechanism is facilitated by `VersionedAddressBook` and `VersionedTaskList`. The `VersionedAddressBook` extends `AddressBook` with an undo/redo history, stored internally as an `addressBookStateList` and `addressBookStatePointer`. The `VersionedTaskList` extends `TaskList` with an undo/redo history, stored internally as an `taskListStateList` and `taskListStatePointer`.

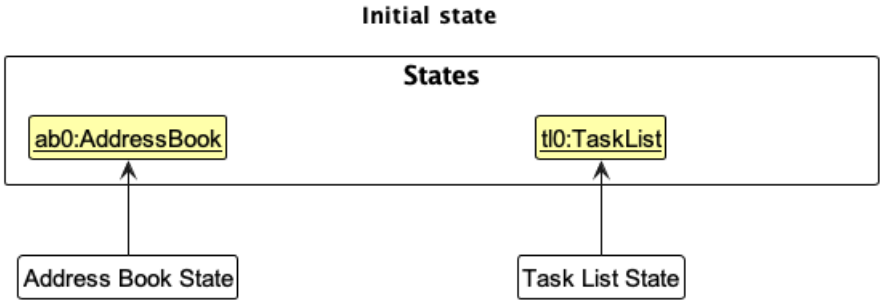
Additionally, they implements the following operations:

- `VersionedAddressBook#commit()` — Saves the current address book state in its history.
- `VersionedAddressBook#undo()` — Restores the previous address book state from its history.
- `VersionedAddressBook#redo()` — Restores a previously undone address book state from its history.
- `VersionedTaskList#commit()` — Saves the current task list state in its history.
- `VersionedTaskList#undo()` — Restores the previous task list state from its history.
- `VersionedTaskList#redo()` — Restores a previously undone task list state from its history.

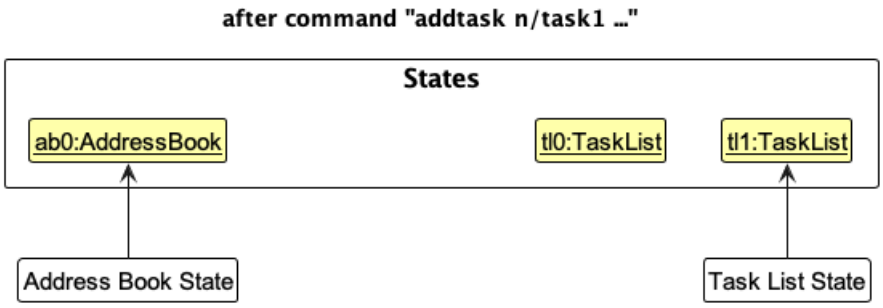
These operations are exposed in the `Model` interface as `Model#commit()`, `Model#undo()` and `Model#redo()` respectively.

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 `VersionedAddressBook` and `VersionedTaskList` will be initialized with the initial address book and task list state, with the `addressBookStatePointer` and `taskListStatePointer` respectively.

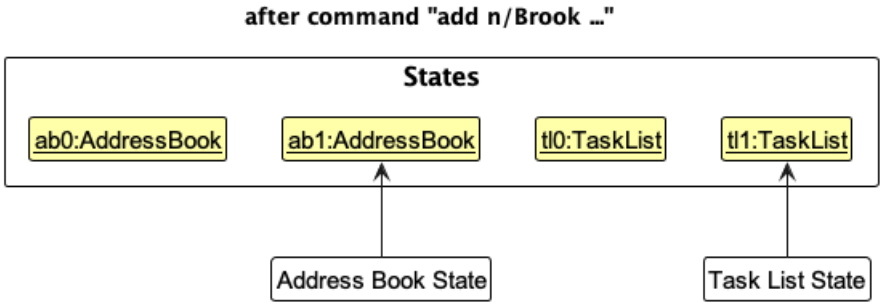


Step 2. The user executes `addtask n/task1 …` to add a new task. The `addtask` command also calls `Model#commit()`, causing another modified task list state to be saved into the `taskListStateList`.

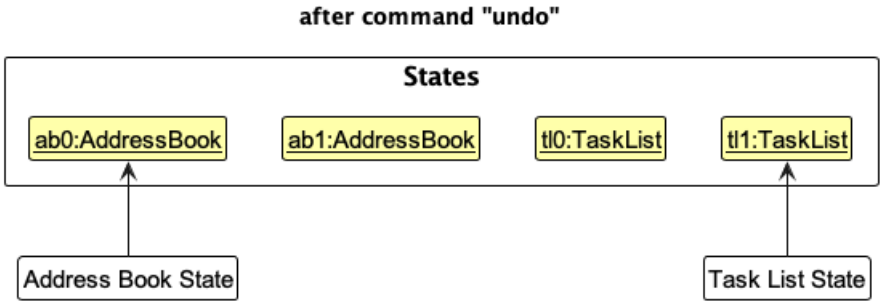


ℹ **Note:** If a command fails its execution, it will not call `Model#commit()`, so the state will not be saved.

Step 3. The user executes `add n/Brook …` to add a new person. The `add` command also calls `Model#commit()`, causing another modified task list state to be saved into the `addressBookStateList`.

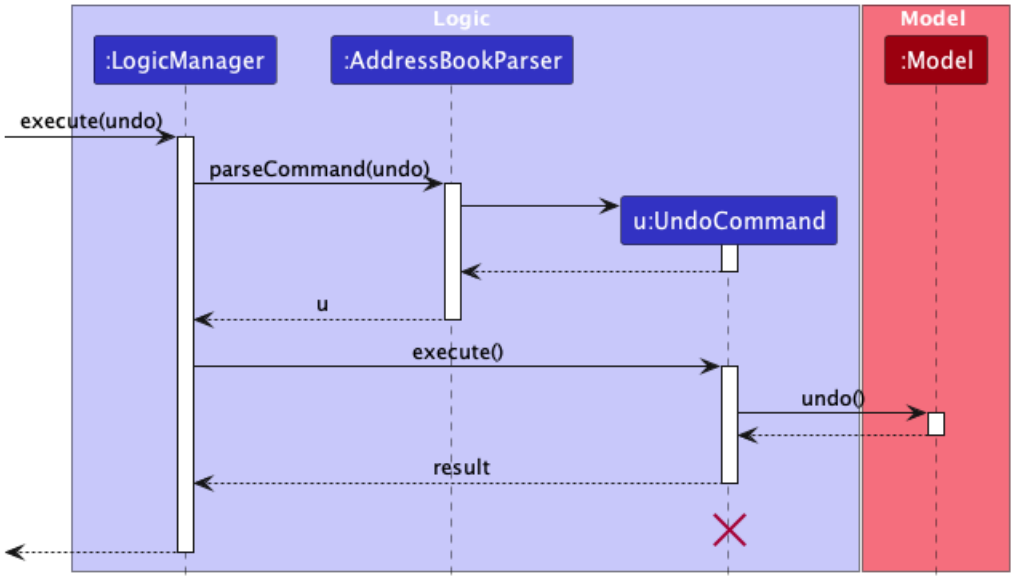


Step 4. The user now decides that adding the person was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undo()`, which will shift the `addressBookStatePointer` once to the left, pointing it to the previous address book state, and restores the address book to that state.



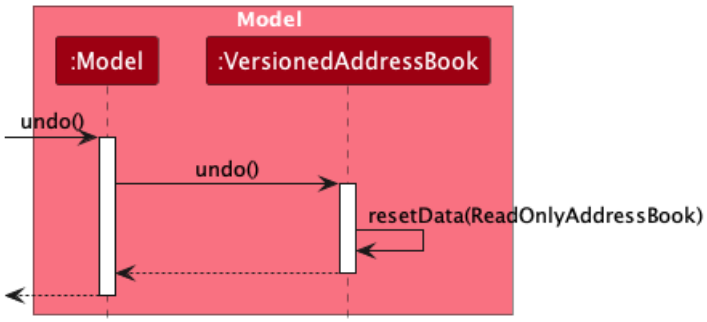
ℹ **Note:** If the pointers are at index 0, pointing to the initial state, then there are no previous states to restore. The `undo` command uses `Model#canUndo()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

The following sequence diagram shows how an undo operation goes through the `Logic` component:



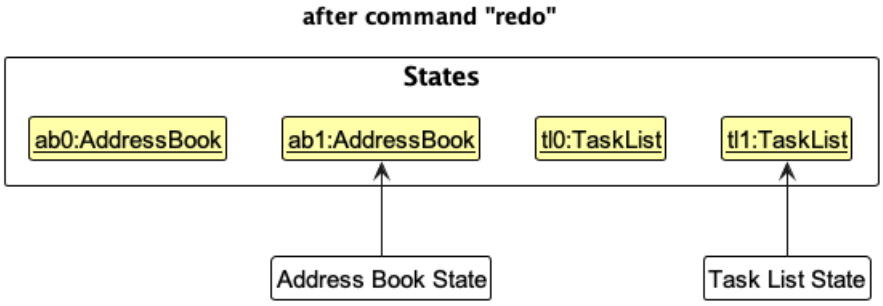
ℹ **Note:** The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

Similarly, how an undo operation goes through the `Model` component is shown below:



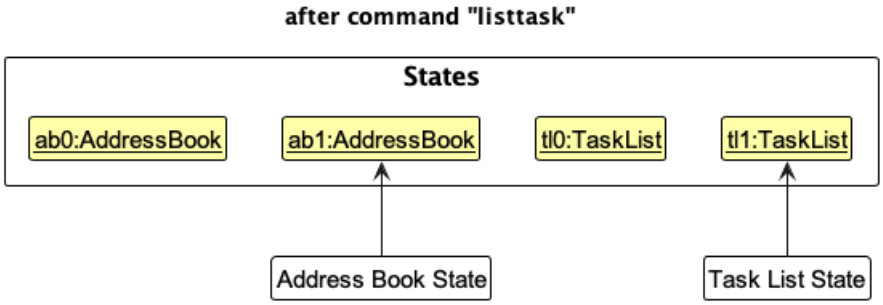
The `redo` command does the opposite — it calls `Model#redo()`, which shifts the pointers once to the right, pointing to the previously undone state, and restores the address book or task list to that state.

Step 5. The user decides that adding the person was not a mistake, and decides to redo that action by executing the `redo` command. The `redo` command will call `Model#redo()`, which will shift the `addressBookStatePointer` once to the right, pointing it to the next address book state, and restores the address book to that state.



ℹ **Note:** If the `addressBookStatePointer` is at index `addressBookStateList.size() - 1` or `taskListStatePointer` is at index `taskListStateList.size() - 1`, pointing to the latest state, then there are no undone states to restore for the respective commands. The `redo` command uses `Model#canRedo()` 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 6. The user then decides to execute the command `listtask`. Commands that do not modify the address book task list, such as `listtask`, will usually not call `Model#commit()`, `Model#undo()` or `Model#redo()`. Thus, the state lists and state pointers remains unchanged.



Design considerations:

Aspect: How undo & redo executes:

- **Alternative 1 (current choice):** Saves the entire address book and task list.
    - 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 `deletetask`, just save the task being deleted).
    - Cons: We must ensure that the implementation of each individual command are correct.

Documentation, logging, testing, configuration, dev-ops

- Documentation guide
- Testing guide
- Logging guide
- Configuration guide
- DevOps guide

Appendix: Requirements

Product scope

Target user profile:

- has a need to manage a significant number of contacts
- prefer desktop apps over other types
- can type fast
- prefers typing to mouse interactions
- is reasonably comfortable using CLI apps
- are currently managing people for groups
- has a tendency to forget tasks to complete

Value proposition: This app aims to help leaders to keep track of members of formed groups and their contact information. This app helps to keep track of individual and group tasks, deadlines and meetings, thus allowing them to have a better overview of the structure.

User stories

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

Priority	As a ...	I want to ...	So that I can...
`* * *`	student	add personal tasks	keep up to date with the different tasks to complete
`* * *`	student	delete tasks	
`* * *`	student	mark/unmark the tasks as done/not done	keep track of tasks that are completed
`* *`	group leader	assign tasks to individuals within the group	manage individual tasks
`* *`	busy group leader	see an overview of all the saved task	save time
`* *`	student	set deadline for my tasks	see which task need to be done earlier

Use cases

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

Use case: Assigns a task

MSS

1. User requests to list of contacts
2. TeamTracker shows a list of contacts
3. User requests to assign a task to a contact
4. TeamTracker assigns the task to the contact

   Use case ends.

Extensions

- 2a. The list is empty.

  Use case ends.

- 4a. The task given does not exist.

- 4a1. TeamTracker shows an error message.

  Use case ends.

Use case: Delete a task

MSS

1. User requests to list tasks
2. TeamTracker shows a list of tasks
3. User requests to delete a specific task in the list
4. TeamTracker deletes the task

   Use case ends.

Extensions

- 2a. The list is empty.

  Use case ends.

- 3a. The given index is invalid.
- 3a1. TeamTracker shows an error message.

  Use case resumes at step 2.

Use case: Add a task

MSS

1. User requests to add a task to the list of tasks
2. TeamTracker adds to the list

   Use case ends.

Extensions

- 2a. The given parameters is invalid.
- 2a1. TeamTracker shows an error message.

  Use case ends.

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 be used for long periods 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 respond within one seconds.
5. Documentation should be easy for users that are inexperienced in command line to follow.
6. User Interface should be straightforward and intuitive to navigate even for first-time users.
7. Should be able to store more than 10,000 tasks.
8. Should lose no more than 1 command worth of work in case of an app crash.
9. Should work without internet access.

Glossary

- **Mainstream OS**: Windows, Linux, Unix, MacOS
- **System admin commands**: commands or instructions that are used through a command-line interface (CLI) or a terminal window
- **Action**: TeamTracker processing a user command

Appendix: Planned Enhancements

Team size: 5

1. **Loosen the constraints for a valid name**: The current constraints for a valid name that it should only contain alphanumeric characters and spaces, and it should not be blank, are too restricted as certain actual names (such as those containing `s/o` or the character `'`) are not supported under these constraints. We plan to loosen these constraints by accepting any character, as long as the given name is not blank.
2. **Loosen the constraints for a valid phone number**: The current constraints for a valid phone number that it should only contain numbers, and it should be at least 3 digits long, are too restricted as users may want to enter a phone number with the `-` characters and spaces. We plan to loosen these constraints by accepting these characters as well.
3. **Tighten the definition of two contacts being the same**: The current definition that two contacts are the same if and only if the two contacts have the same name, is too restricted because it is extremely common that two different people share the same name, yet they need to be treated as different contacts. We plan to tighten this definition to also require the two contacts to have the same phone number, email and address.
4. **Change the handling of invalid deadlines**: The current behaviour of `addtask` and `edittask` against invalid dates, where for example dates like `30-02-2024` is implicitly replaced by `29-02-2024`, is counter-intuitive to the users. We plan to explicitly display an error message like `Invalid date` to the user whenever a non-existent date is given.
5. **Make 'failed task addition due to invalid priority' message more specific**: The current error message for a failed task addition when the given priority is invalid `Task priority can take either an integer or low, medium, high, and it should not be blank` is inaccurate. We plan to make the error message also mention the range of accepted integers: `Task priority can take either an integer between 1 and 3, or low, medium, high, and it should not be blank`.
6. **Make 'invalid index' message more specific**: The current error message displayed when a given index is out of range `The person index provided is invalid` or `The task index provided is invalid` is too general. We plan to make the error message also mention which indices are invalid: `The person index 2, 3, 4 are invalid`, since mass ops fails whenever at least one of the provided indices is invalid.
7. **Support the clearing of all tasks in the task list**: The current lack of such a command makes the feature of 'clearing all entries' rather incomplete. We plan to add a command that can clear all tasks in the task list, correspondingly similar to the `clear` command which clears all persons in the address book.
8. **Support the removal of deadline from tasks by means of editing the task**: The current method of removing deadline from tasks, which requires users to delete and add the same task, inconveniences users. We plan to support the removal of deadline from tasks using the `edittask` command, where deadline can be removed from tasks by providing a blank `by/` parameter.
9. **Enhance the colour combination of priority level blocks in the UI**: The current UI display for a low priority level, involving a striking green colour block with a white colour font, makes it difficult for users to see the label for the block. We plan to make the green colour of the block less striking, e.g. using dark green (`#006400`) or forest green (`#228B22`) instead.

Appendix: Instructions for manual testing

Given below are instructions to test the app manually.

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

Launch and shutdown

1. Initial launch

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

    2. Double-click the jar file Expected: Shows the GUI with a set of sample contacts. The window size may not be optimum.

2. Saving window preferences

    1. Resize the window to an optimum size. Move the window to a different location. Close the window.

    2. Re-launch the app by double-clicking the jar file.
       Expected: The most recent window size and location is retained.

Deleting a person

1. Deleting a person while all persons are being shown

    1. Prerequisites: List all persons using the `list` command. Multiple persons in the list.

2. Test case: `delete 1`
   Expected: First person is deleted from the list. Name of the deleted person shown in the status message. Timestamp in the status bar is updated.

3. Test case: `delete 0`
   Expected: No person is deleted. Error details shown in the status message. Status bar remains the same.

4. Other incorrect delete commands to try: `delete`, `delete x`, `...` (where x is larger than the list size)
   Expected: Similar to previous.

Deleting a task

1. Deleting a task while all tasks are being shown
   1. Prerequisites: List all tasks using the `listtask` command. Multiple tasks in the list.

   2. Test case: `deletetask 1`
      Expected: First task is deleted from the list. Name of the deleted task shown in the status message. Timestamp in the status bar is updated.

   3. Test case: `deletetask 0`
      Expected: No task is deleted. Error details shown in the status message. Status bar remains the same.

   4. Other incorrect deletetask commands to try: `deletetask`, `deletetask x` (where x is larger than the list size)
      Expected: Similar to previous.

Editing a task

1. Editing a task while all tasks are being shown
   1. Prerequisites: List all tasks using the `listtask` command. Multiple tasks in the list.

   2. Test case: `edittask 1 p/low`
      Expected: First task is edited. Name of the edit task shown in the status message. The priority will be updated to `LOW` in this case.

   3. Test case: `edittask 0 p/low`
      Expected: No task is edited. Error details shown in the status message. Status bar remains the same.

   4. Other incorrect edittask commands to try: `edittask`, `edittask 1`, `edittask 1 p/very high`, `edittask x` (where x is larger than the list size)
      Expected: Similar to previous.

Saving data

1. Dealing with missing data files
   1. Prerequisites: One or more of the data files does not exist in the data folder.

   2. Test case: tasklist.json is missing from the data folder on launch
      Expected: TeamTracker should still launch and function as normal.

   3. Other missing files: addressbook.json or both the data files
      Expected: Similar to previous.