NOVA - Developer Guide

By: `CS2103T-F10-3` Since: `Apr 2020` Licence: `MIT`

1. Setting up

Refer to the guide here.

2. Design

2.1. Architecture

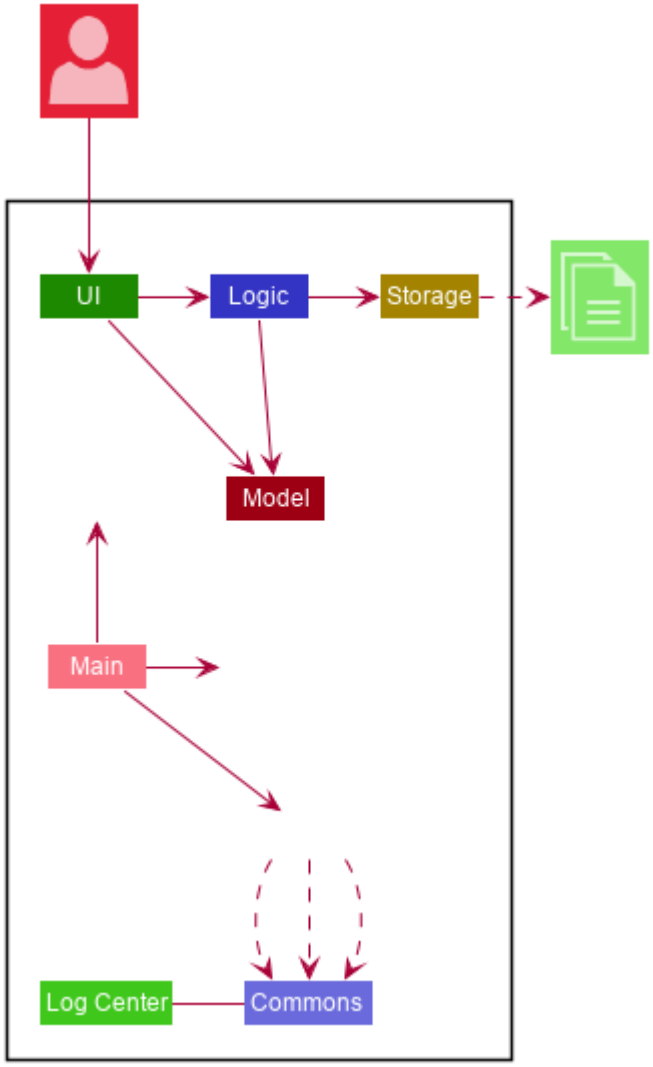


Figure 1. Architecture Diagram

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

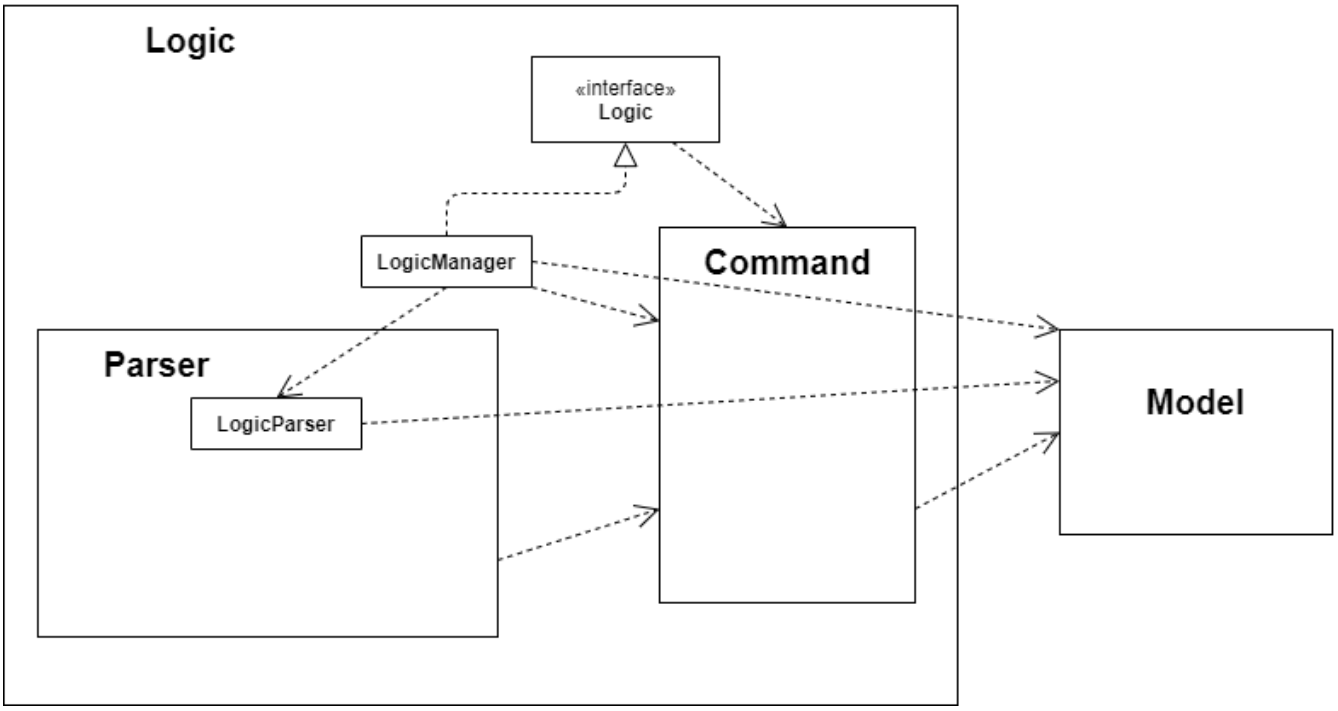


Figure 2. Architecture Diagram of the Logic Component

How the architecture components interact with each other (Loh Sze Ying)

The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `delete i\1` after navigating to Address Book mode with `nav ab`. The same type of interaction applies to other mode within NOVA, such as Schedule mode, Planner mode or Progress Tracker mode.

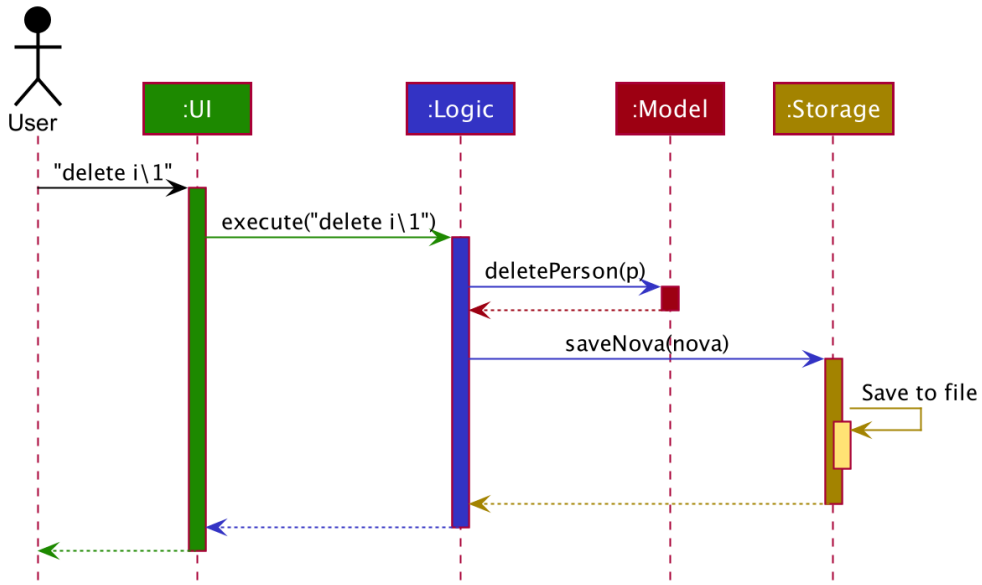


Figure 3. Component interactions for `delete i\1` command

The sections below give more details of each component.

2.2. UI component (Yap Wen Jun Bryan)

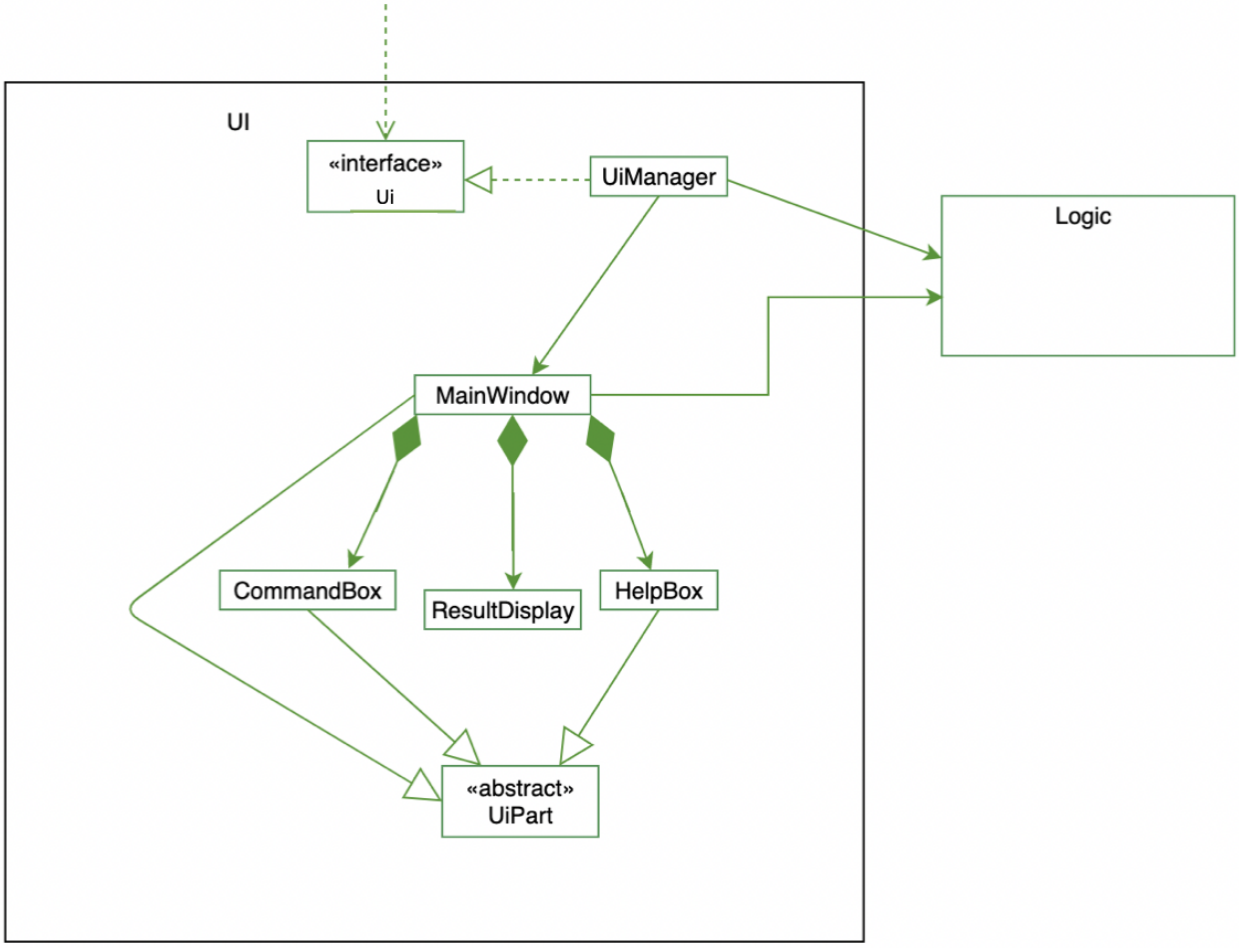


Figure 4. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay` and `HelpBox`. 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 (HelpBox does not have a `.fxml` file) 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 (Chua Huixian)

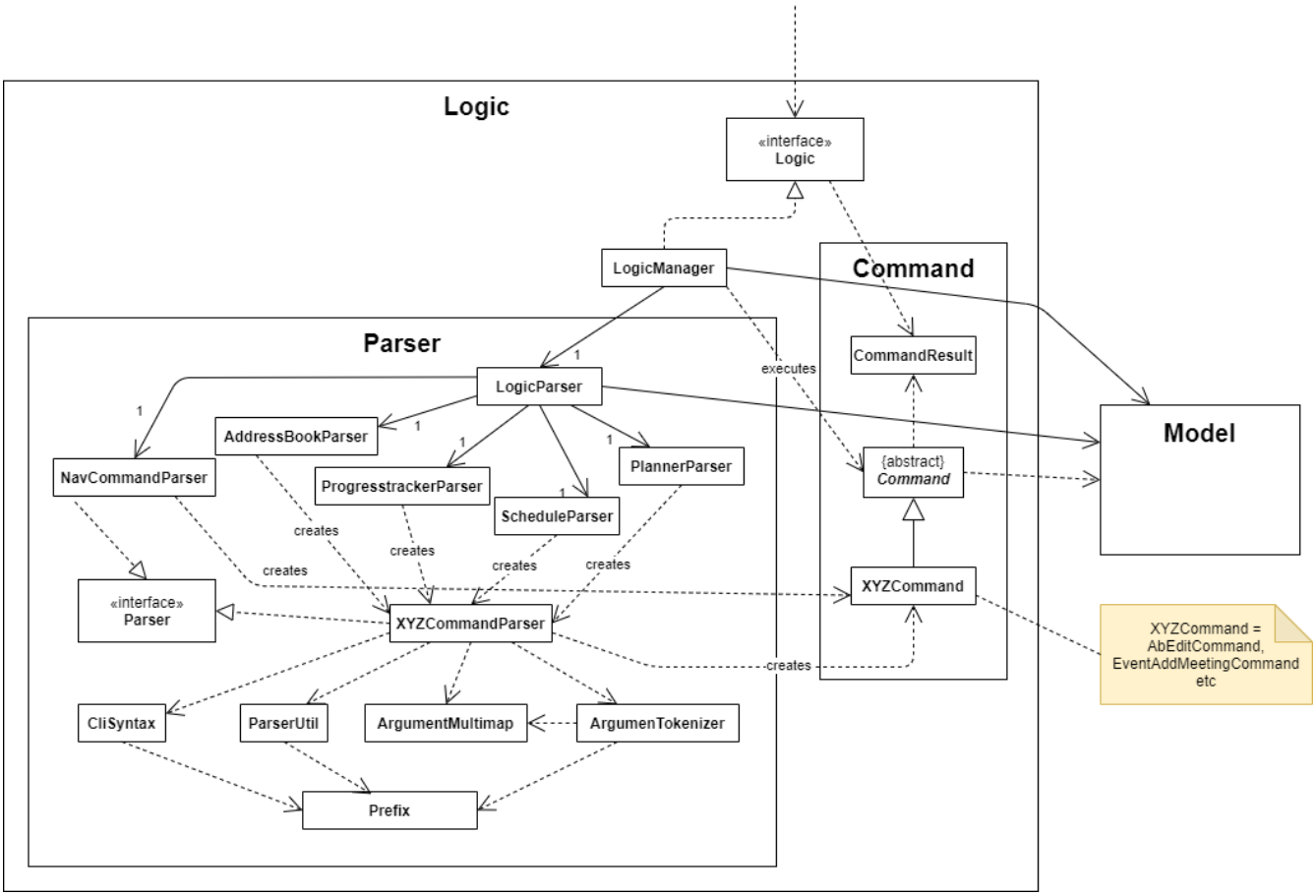


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `LogicParser` class to determine which mode the user is in when they input a command.
2. After which, the relevant parser is called (e.g. `AddressBookParser`).
3. This results in a `Command` object which is executed by the `LogicManager`.
4. The command execution can affect the `Model` (e.g. adding a person).
5. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.

6. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.

2.4. Model component (Terence)

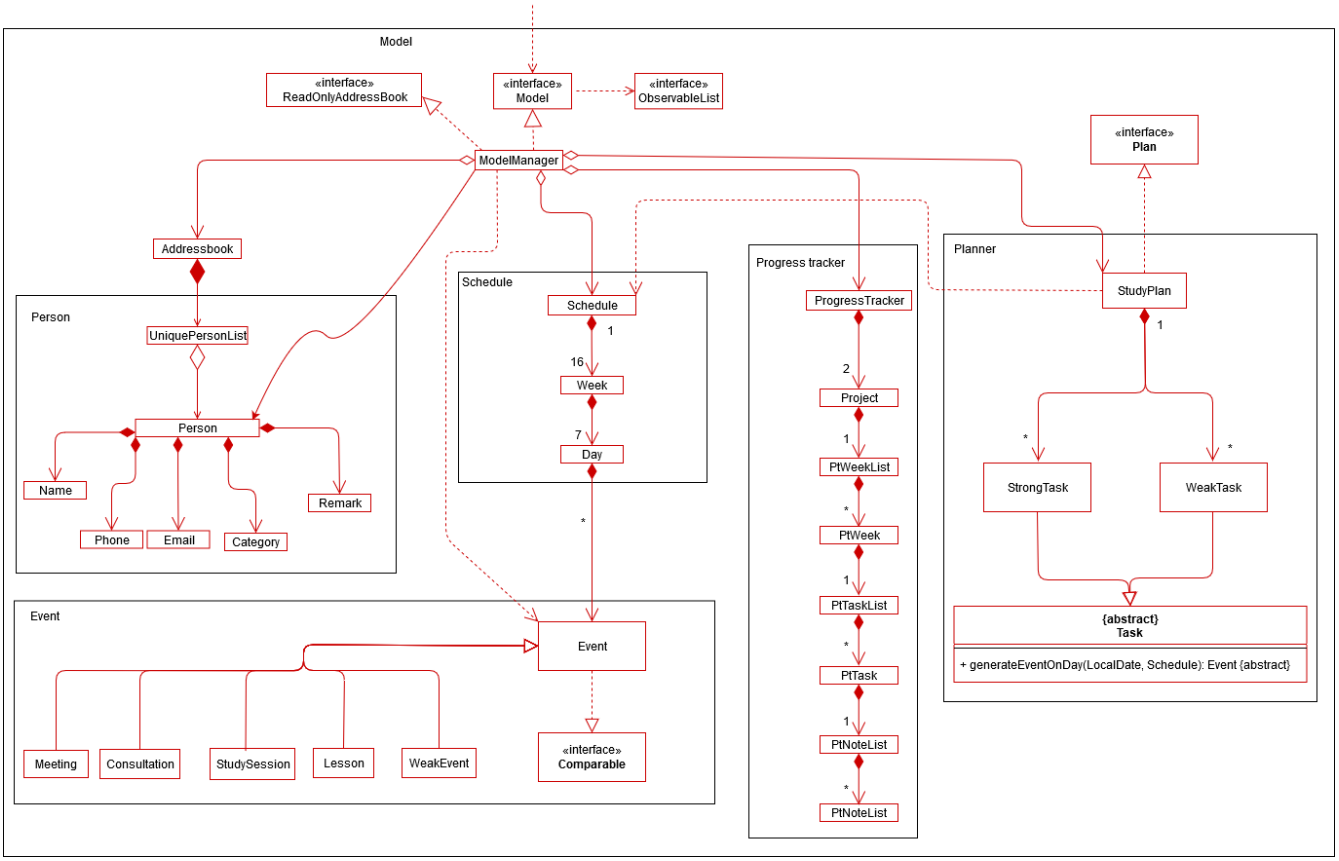


Figure 6. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the Address Book data.
- stores a 'Schedule' object that represents the user's schedule.
- stores a 'ProgressTracker' object that represents the user's progress in their project tasks.
- exposes 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.
- does not depend on any of the other three components.

2.5. Storage component

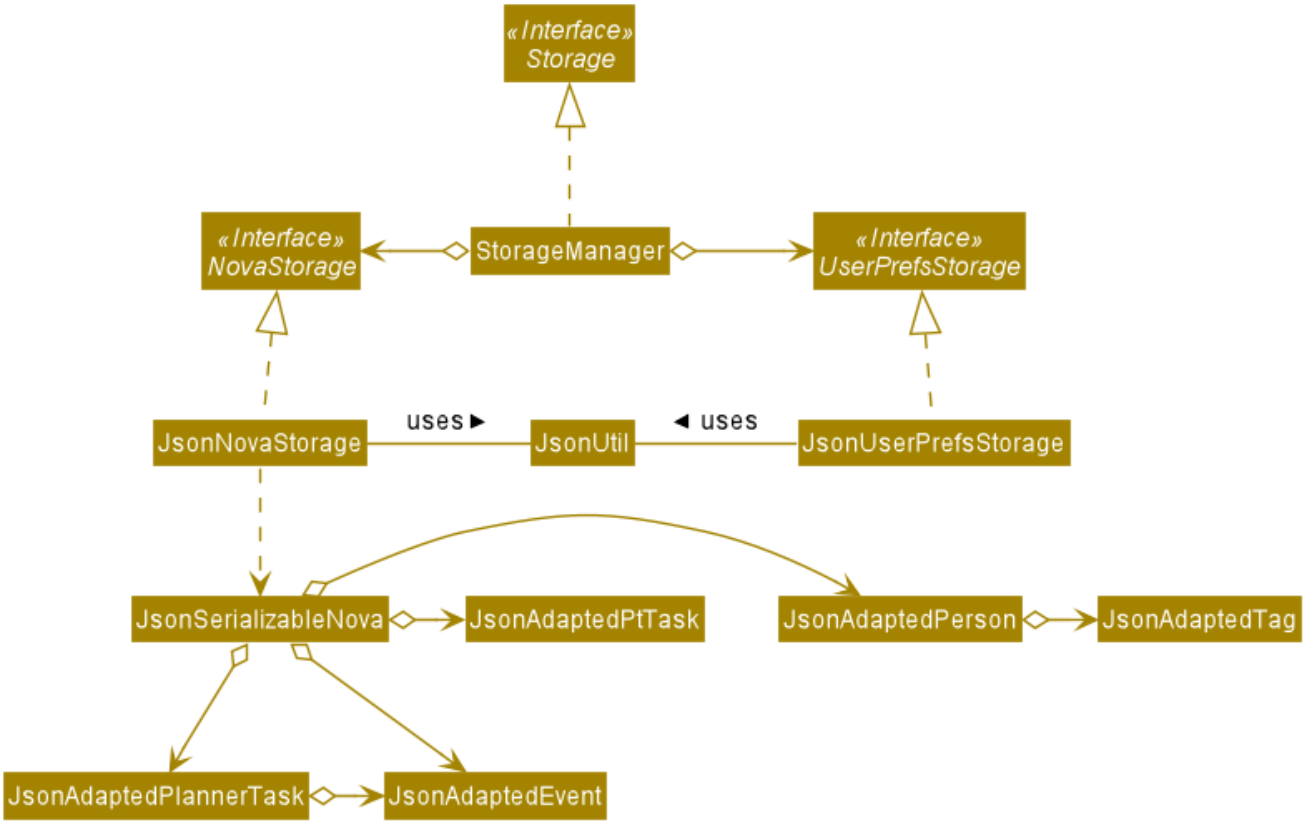


Figure 7. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

- can save `UserPref` objects in json format and read it back.
- can save NOVA's data in json format and read it back.

2.6. Common classes (Loh Sze Ying)

Classes used by multiple components are in the `seedu.nova.commons` package. All of the classes under `Commons` work independently.

Most notably,

- **API** : `LogsCenter.java`
  The `LogsCenter` is used by NOVA to display logs when running NOVA in terminal.
- **API** : `Messages.java`
  The `Messages` is used by NOVA and deals with messages to display relating to NOVA's features.

3. Implementation

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

3.1. Manage Events (Chua Huixian)

The manage events feature handles the events of the user, including meetings, consultations, study sessions and lessons. Users are able to:

- add events
- delete events
- add notes to events

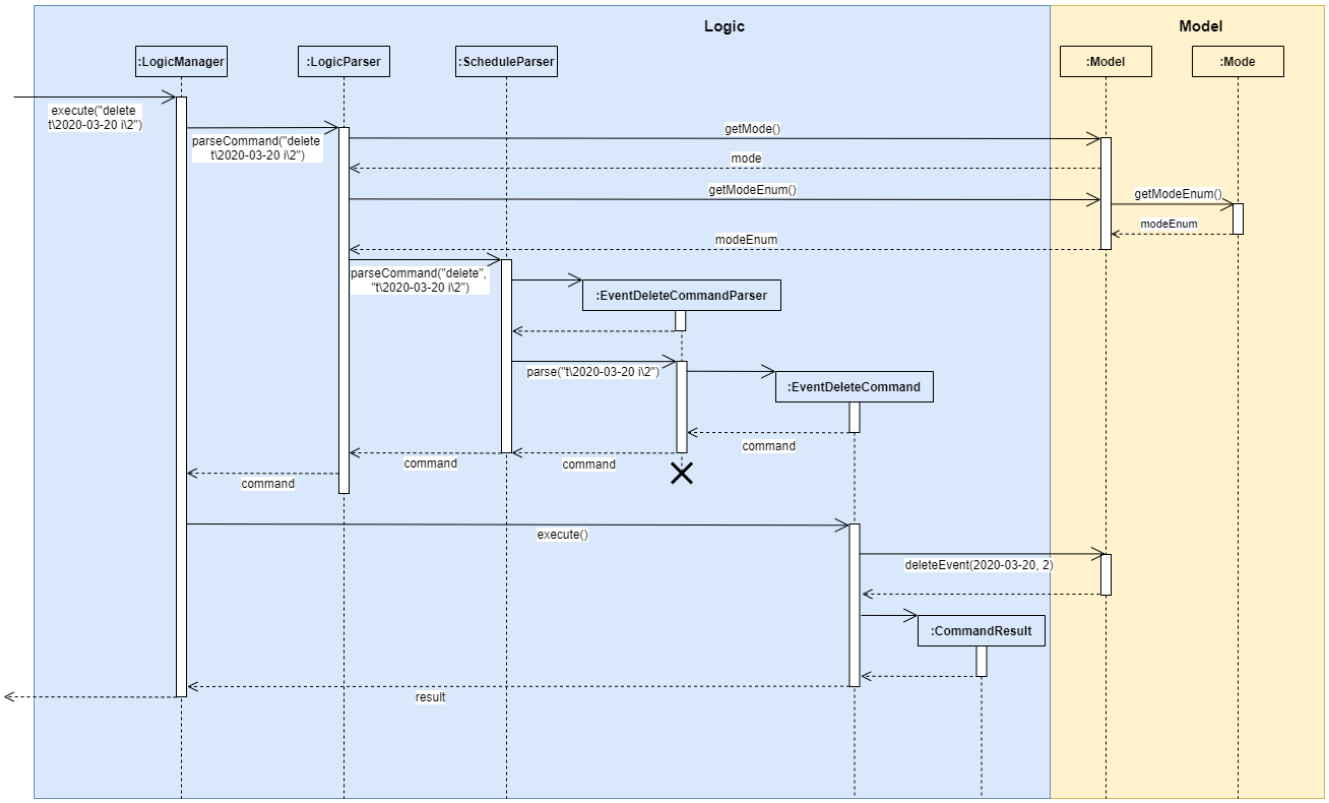
3.1.1. Implementation - Deleting an event

The delete feature allows users to remove events from the schedule. This feature is facilitated by `ScheduleParser`, `EventDeleteCommandParser` and `EventDeleteCommand`. The operation is exposed in the `Model` interface as `Model#deleteEvent()`.

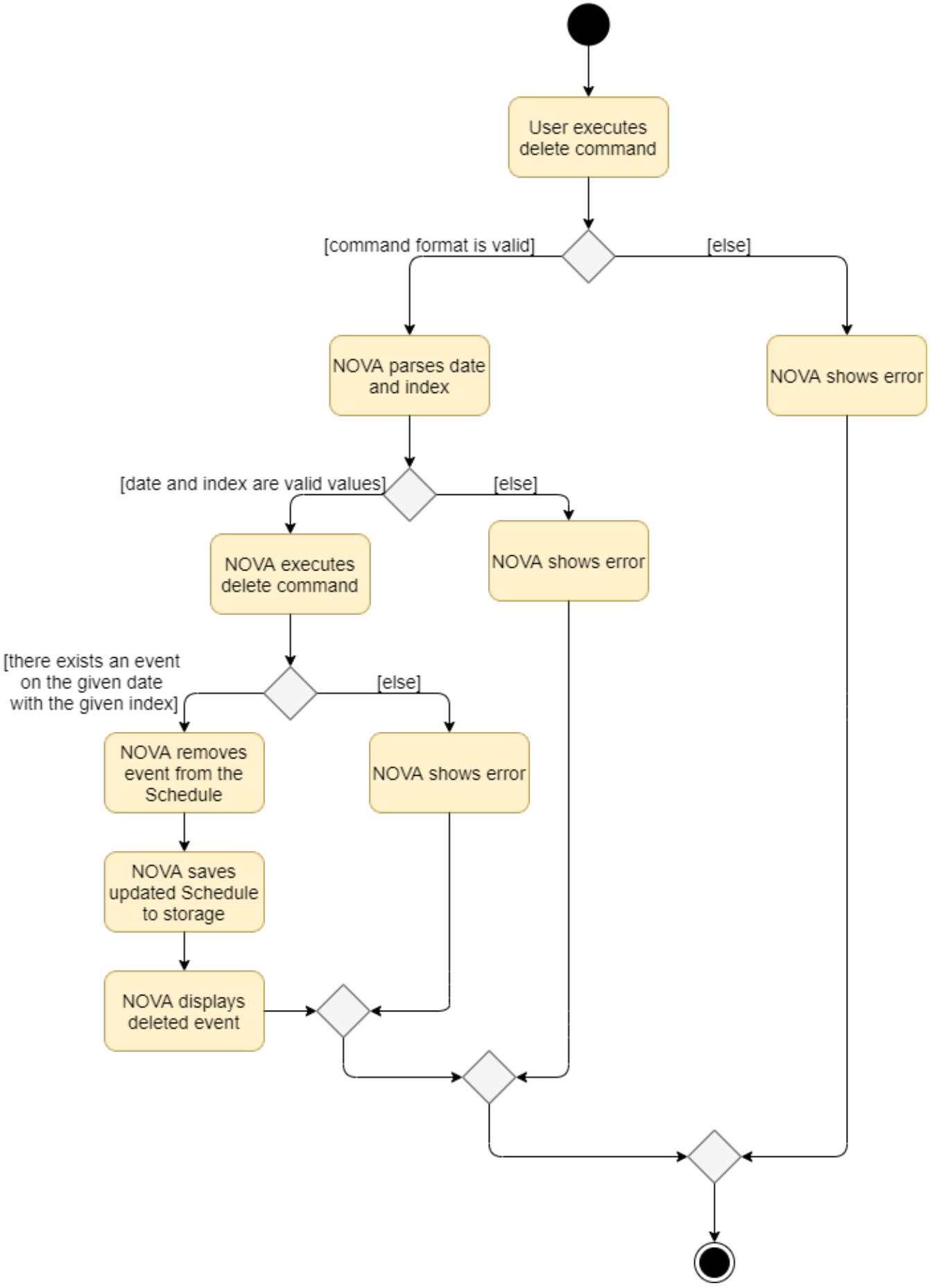
Given below is an example usage scenario and how the delete mechanism behaves at each step.

1. The user does `view t\2020-03-20` to view their events on 20th March 2020.
2. The user executes `delete t\2020-03-20 i\2` command to delete the second event on 20th March 2020.
3. `EventDeleteCommandParser` creates a new `EventDeleteCommand`.
4. `LogicManager` executes the `EventDeleteCommand`.
5. `Model#deleteEvent()` is called, and the `Schedule` object in `ModelManager` is updated.

The following sequence diagram shows how the delete operation works:



The following activity diagram shows what happens when a user inputs a delete command:



3.1.2. Design Considerations

Aspect: Syntax of Deleting an Event

- **Alternative 1 (current choice):** choosing the event by its date and its index in the list of events on that date
    - Pros: relatively short to type, greater ease of implementation
    - Cons: users have to view the list of events on that date before determining which event to mark as done
- **Alternative 2:** choosing the event by description
    - Pros: more recognisable for users
    - Cons: difficulty in implementing as certain events may have the exact same descriptions

3.2. Address Book (Loh Sze Ying)

The address book feature handles the contact list of the users. To enter address book mode, users need to enter `nav ab` command. Users are able to:

- add contacts
- edit contacts
- delete contacts
- find contacts
- list all contacts
- list category specific contacts
- add category specific remark for contacts
- edit category specific remark for contacts
- delete category specific remark for contacts
- undo or redo command
- add profile picture to contacts
- delete profile picture to contacts

3.2.1. Implementation - Undo or redo command

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

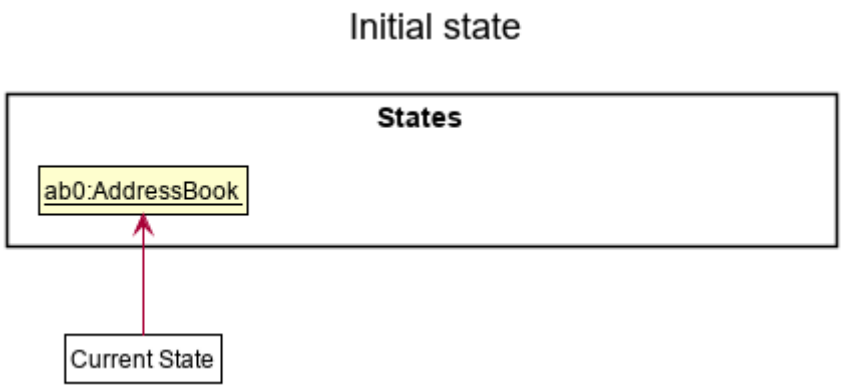
- `VersionedAddressBook#commit()` — Saves the current address book state in its history.
- `VersionedAddressBook#undo()` — Restores the previous address book state from its history.
- `VersionedAddressBook#redo()` — Restores a previously undone address book state from its history.

These operations are exposed in the `Model` interface as `Model#commitAddressBook()`, `Model#undoAddressBook()` and `Model#redoAddressBook()` respectively.

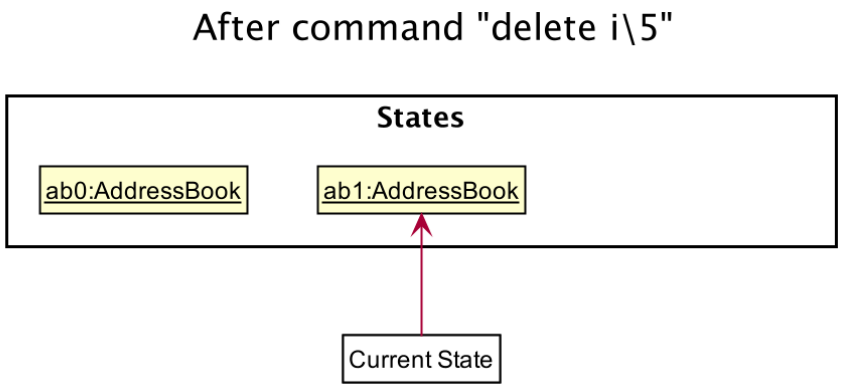
Given below is an example usage scenario and how the undo/redo mechanism behaves at each

step.

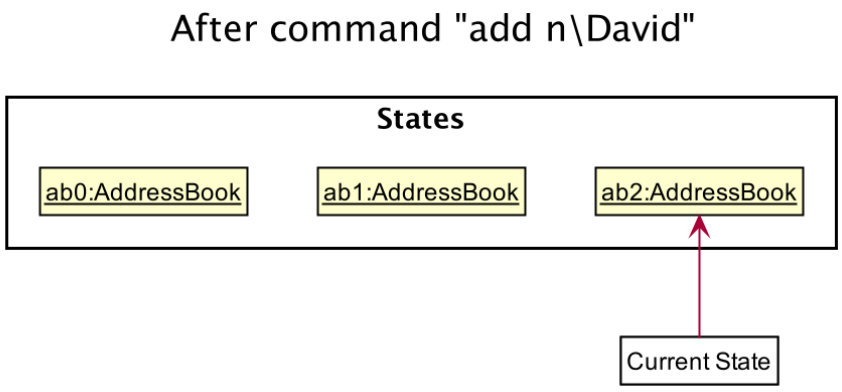
Step 1. The user launches the application for the first time. The `VersionedAddressBook` will be initialized with the initial address book state, and the `currentStatePointer` pointing to that single address book state.



Step 2. The user executes `delete i\5` command to delete the 5th person in the address book. The `delete` command calls `Model#commitAddressBook()`, causing the modified state of the address book after the `delete i\5` command executes to be saved in the `addressBookStateList`, and the `currentStatePointer` is shifted to the newly inserted address book state.

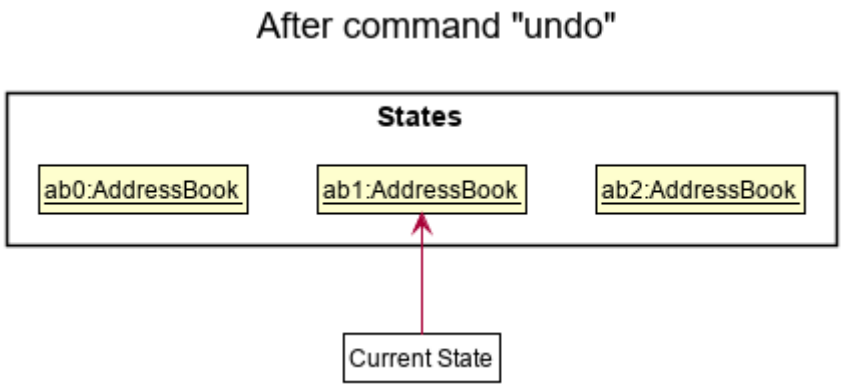


Step 3. The user executes `add n\David …` to add a new person. The `add` command also calls `Model#commitAddressBook()`, causing another modified address book state to be saved into the `addressBookStateList`.



NOTE If a command fails its execution, it will not call `Model#commitAddressBook()`, so the address book state will not be saved into the `addressBookStateList`.

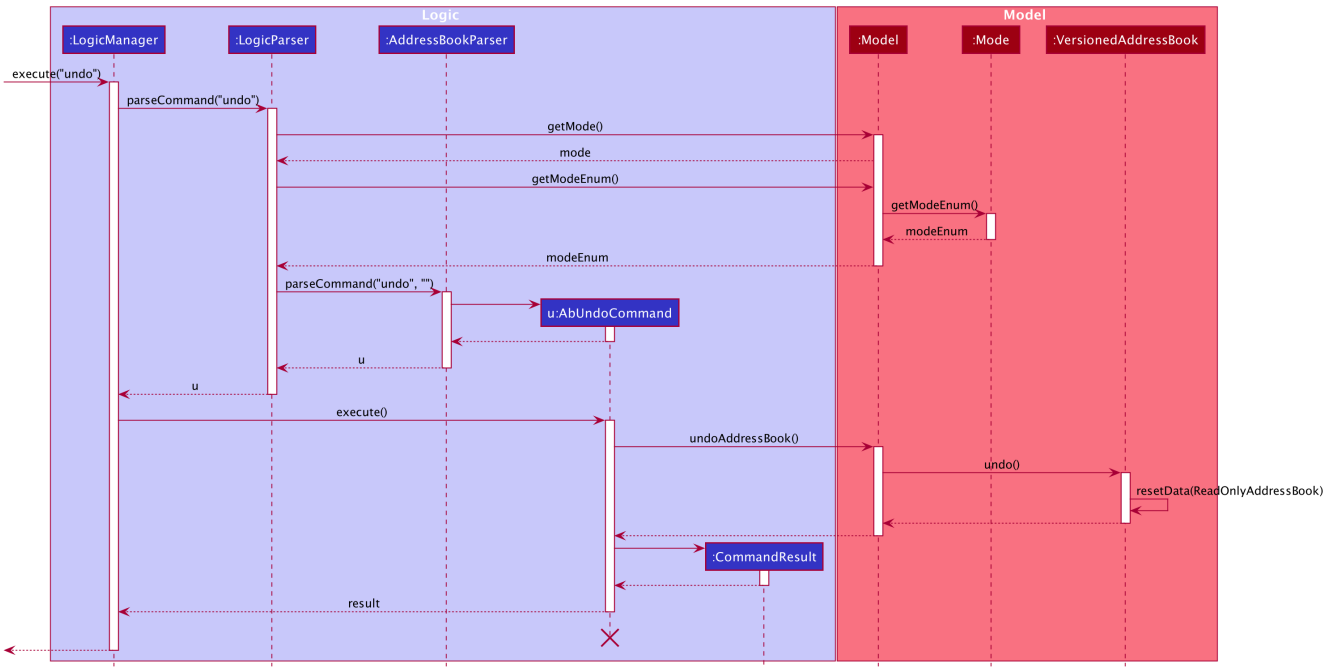
Step 4. The user now decides that adding the person was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoAddressBook()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous address book state, and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



NOTE

If the `currentStatePointer` is at index 0, pointing to the initial address book state, then there are no previous address book states to restore. The `undo` command uses `Model#canUndoAddressBook()` 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:

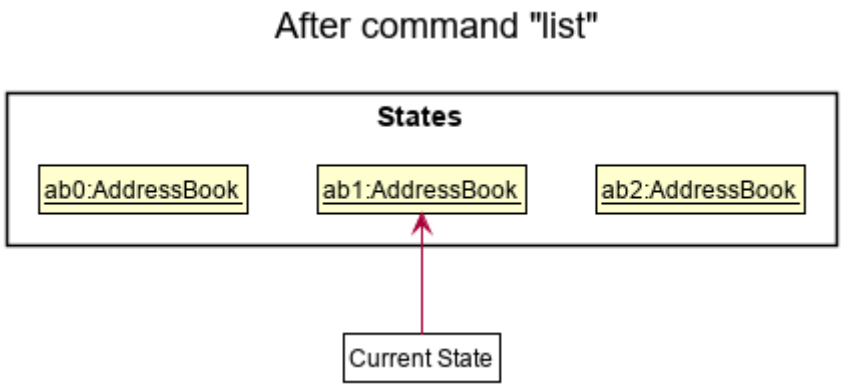


The `redo` command does the opposite — it calls `Model#redoAddressBook()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores the address book to that state.

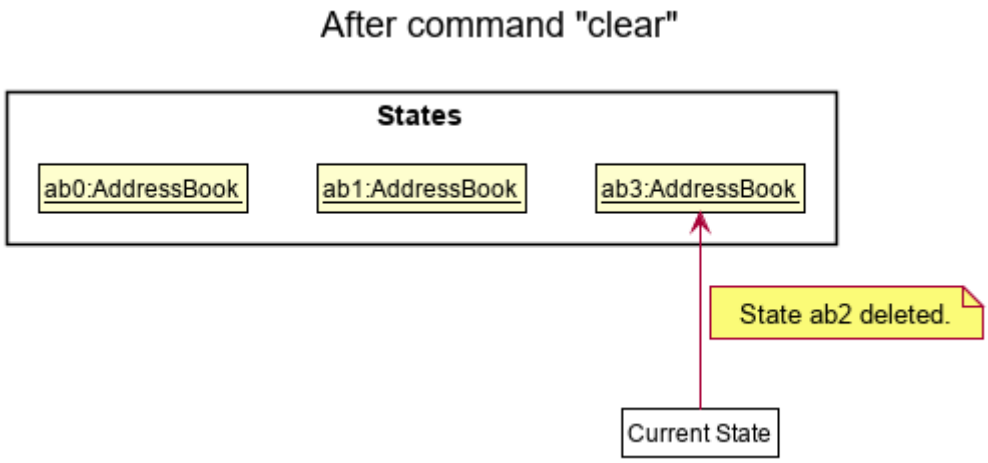
NOTE

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

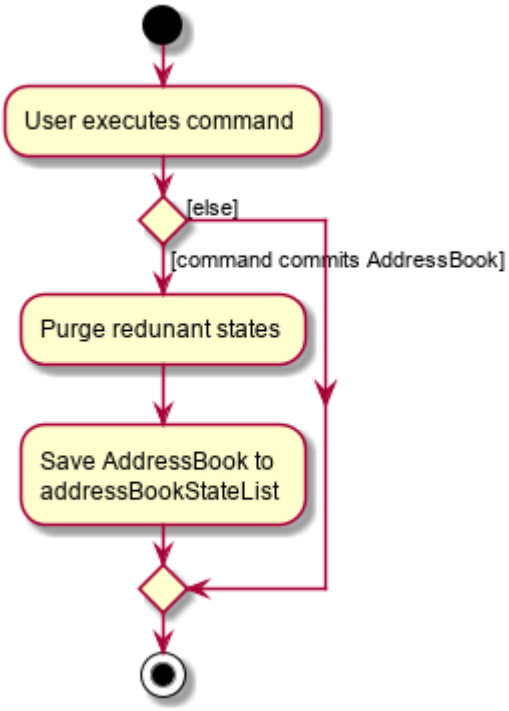
Step 5. The user then decides to execute the command `list`. Commands that do not modify the address book, such as `list`, `list c\classmate`, `list c\teammate` or `find`, will usually not call `Model#commitAddressBook()`, `Model#undoAddressBook()` or `Model#redoAddressBook()`. Thus, the `addressBookStateList` remains unchanged.



Step 6. The user executes `clear`, which calls `Model#commitAddressBook()`. Since the `currentStatePointer` is not pointing at the end of the `addressBookStateList`, all address book 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:



3.2.2. Design Considerations - Undo or redo command

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire address book.
  - 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 person 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 address book 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 `VersionedAddressBook`.
- **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.

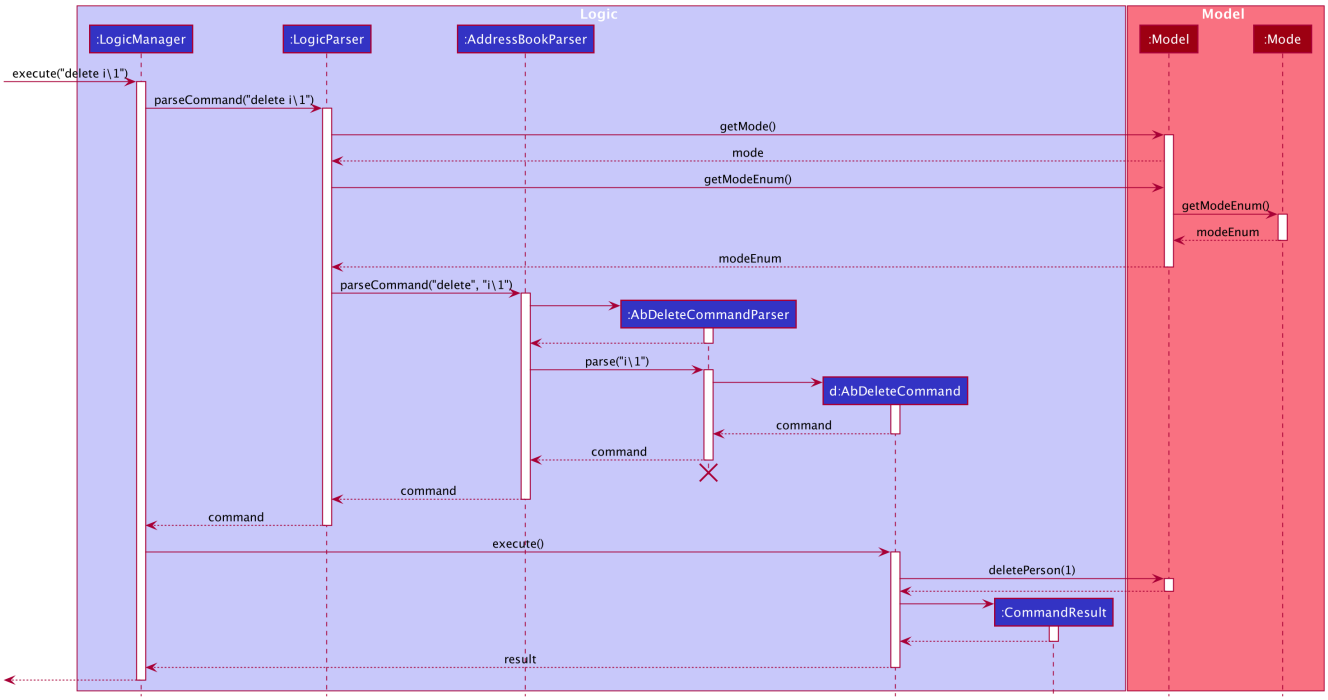
3.2.3. Implementation - Delete a contact

The edit feature allows users to edit a contact from Address Book. This feature is facilitated by `AddressBookParser`, `AbDeleteCommandParser` and `AbDeleteCommand`. The operation is exposed in the `Model` interface as `Model#deletePerson()`.

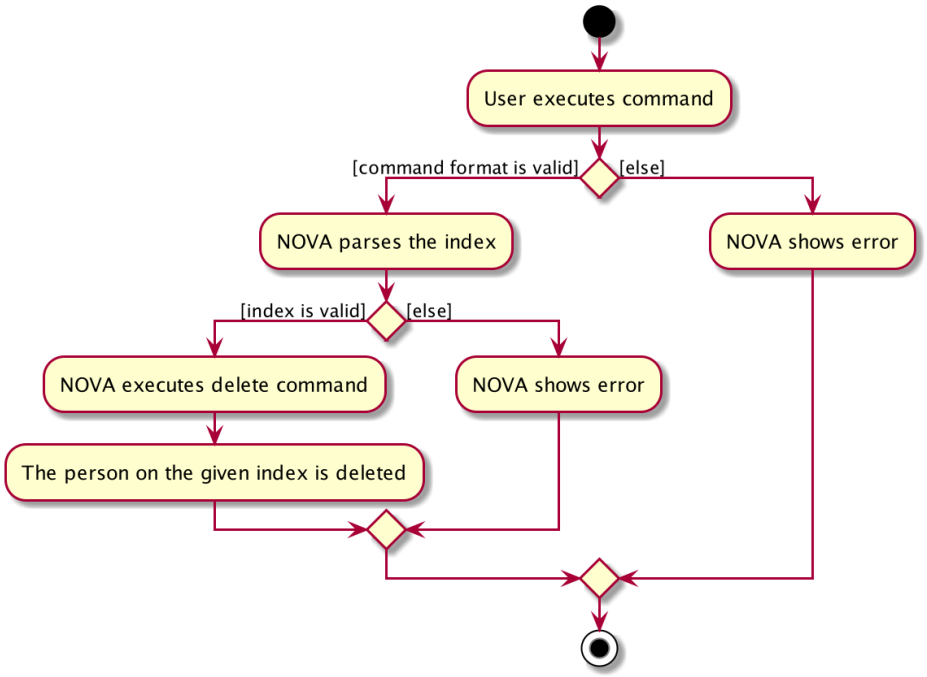
Given below is an example usage scenario and how the delete mechanism behaves at each step.

1. The user does `list`, `list c\classmate`, `list c\teammate`, or `find n\[name]` to view their contacts.
2. The user executes `delete i\1` command to delete the first contact.
3. `AbDeleteCommandParser` creates a new `AbDeleteCommand`.
4. `LogicManager` executes the `AbDeleteCommand`.
5. `Model#deletePerson()` is called, and the `AddressBook` object in `ModelManager` is updated.

The following sequence diagram shows how the delete operation works:



The following activity diagram shows how the delete mechanism works:



3.2.4. Design Considerations - Delete a contact

Aspect: Deleting a contact by index or name

- **Alternative 1 (current choice):** Use index tag to delete contact
  - Pros: Shorter command to type by using index
  - Cons: Users need to use `list`, `list c\classmate`, `list c\teammate` or `find n\[name]` command before deleting contact
- **Alternative 2:** Use name to delete contact
  - Pros: No need to use `list`, `list c\classmate`, `list c\teammate` or `find n\[name]` prior to deleting contact
  - Cons: Need to handle deletion of contacts with the same name

3.3. List tasks under IP project (Yap Wen Jun Bryan)

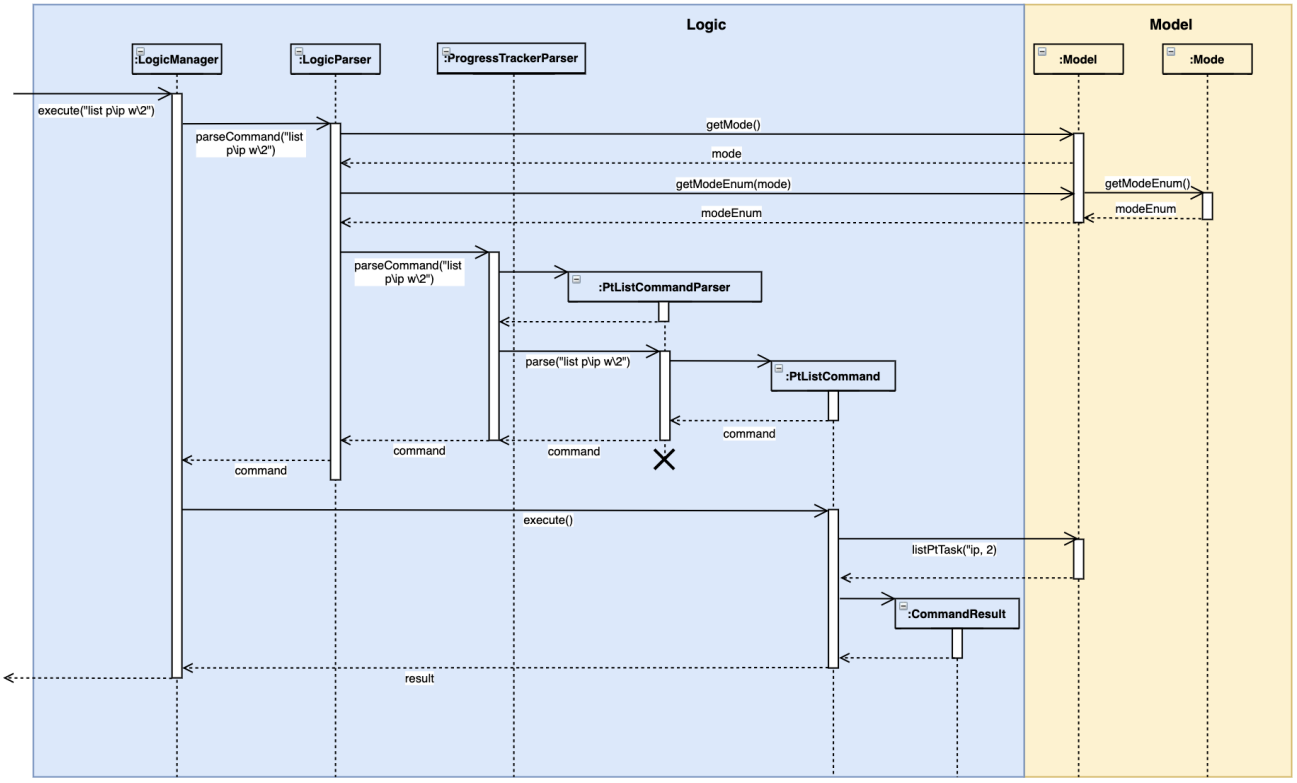
The list tasks feature for the IP project allows the user to view a list of tasks that were added.

3.3.1. Implementation

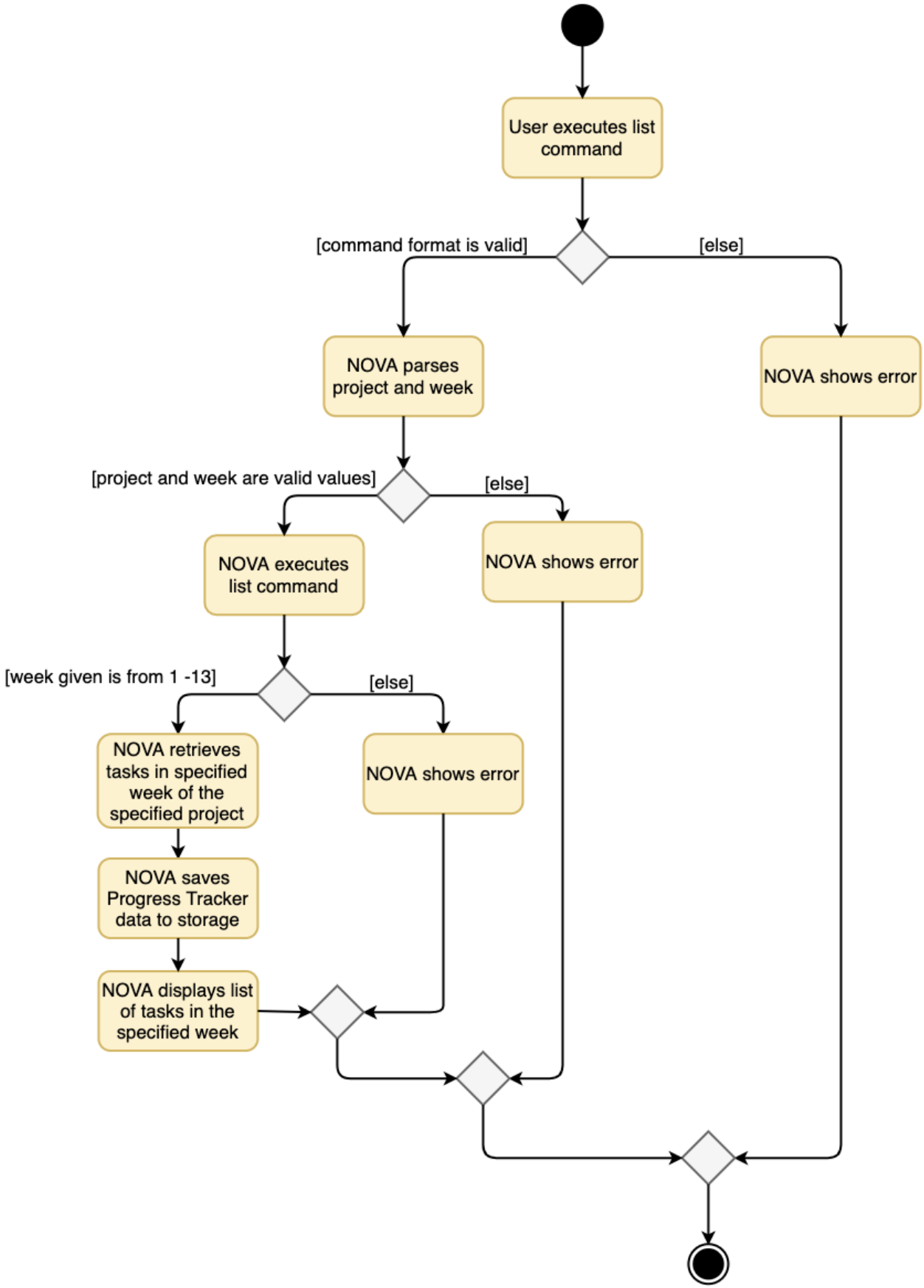
Given below is an example usage scenario and how the list tasks mechanism behaves at each step.

1. The user keys in `list p\ip w\2` into the command box.
2. The user executes `list p\ip w\2'` to view the list of tasks in week 2 of the IP project.
3. `PtListCommandParser` creates a new `PtListCommand`.
4. `LogicManager` executes the `PtListCommand`.
5. `Model#listTasks()` is called and the list of tasks is retrieved.

The following sequence diagram shows how the list tasks operation works:



The following activity diagram shows what happens when a user inputs a list command:



3.3.2. Design Considerations

Aspect: Adding choice of week to view tasks

- **Alternative 1 (current choice):** adding in choice of week to view tasks

- Pros: more freedom to choose what to see as a user
- Cons: longer command to type

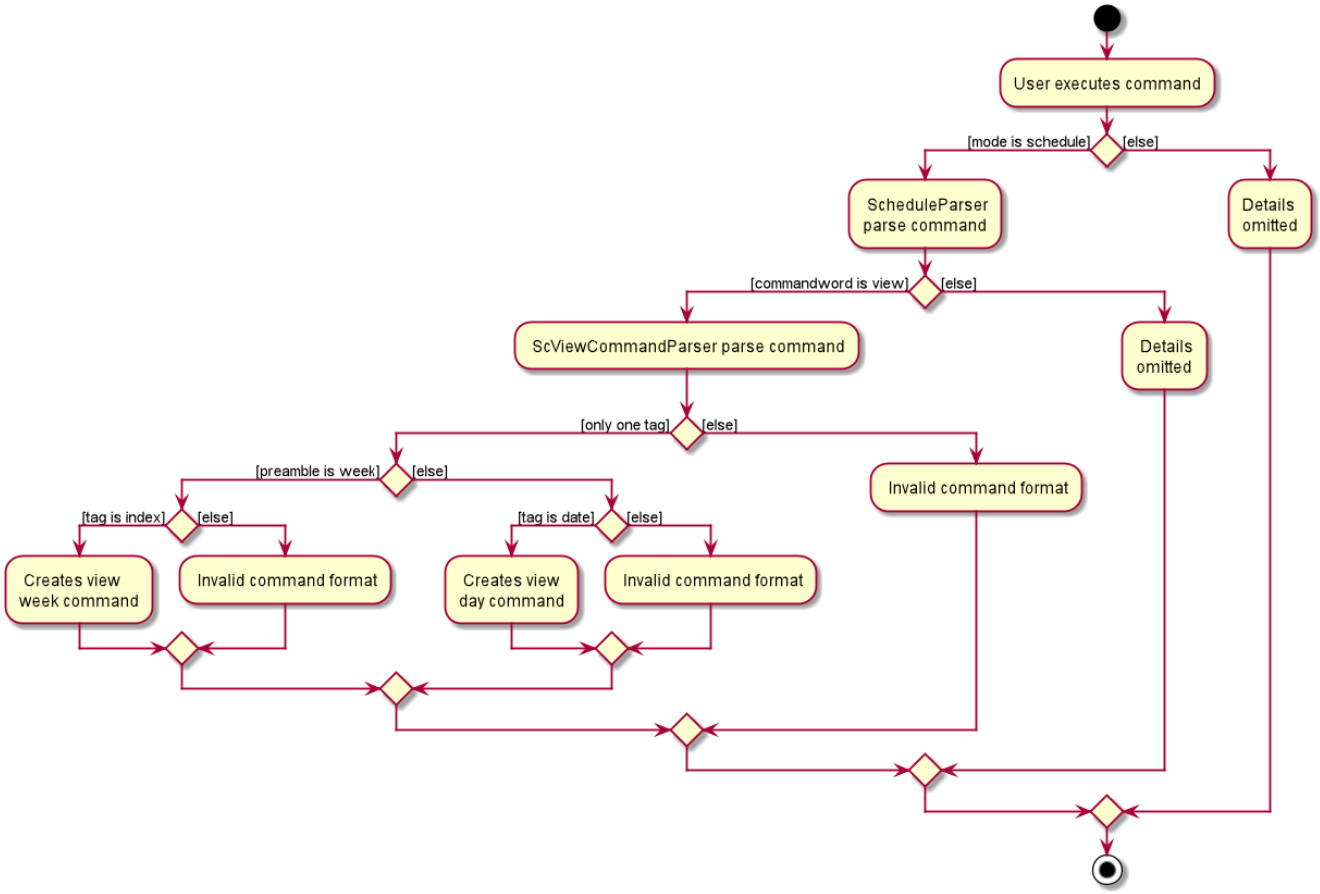
- **Alternative 2:** listing out the whole project tasks rather than letting user choose based on week
  - Pros: shorter command to type and user can see all their tasks at once
  - Cons: if user wants to see tasks only for a specific week will be harder to scroll and find

3.4. View schedule (Terence)

The view schedule feature allows users to view the events they have added into the schedule. Users are able to view the schedule by two time frames:

- By date
- By week

The following activity diagram shows what happens when a user inputs a view command:



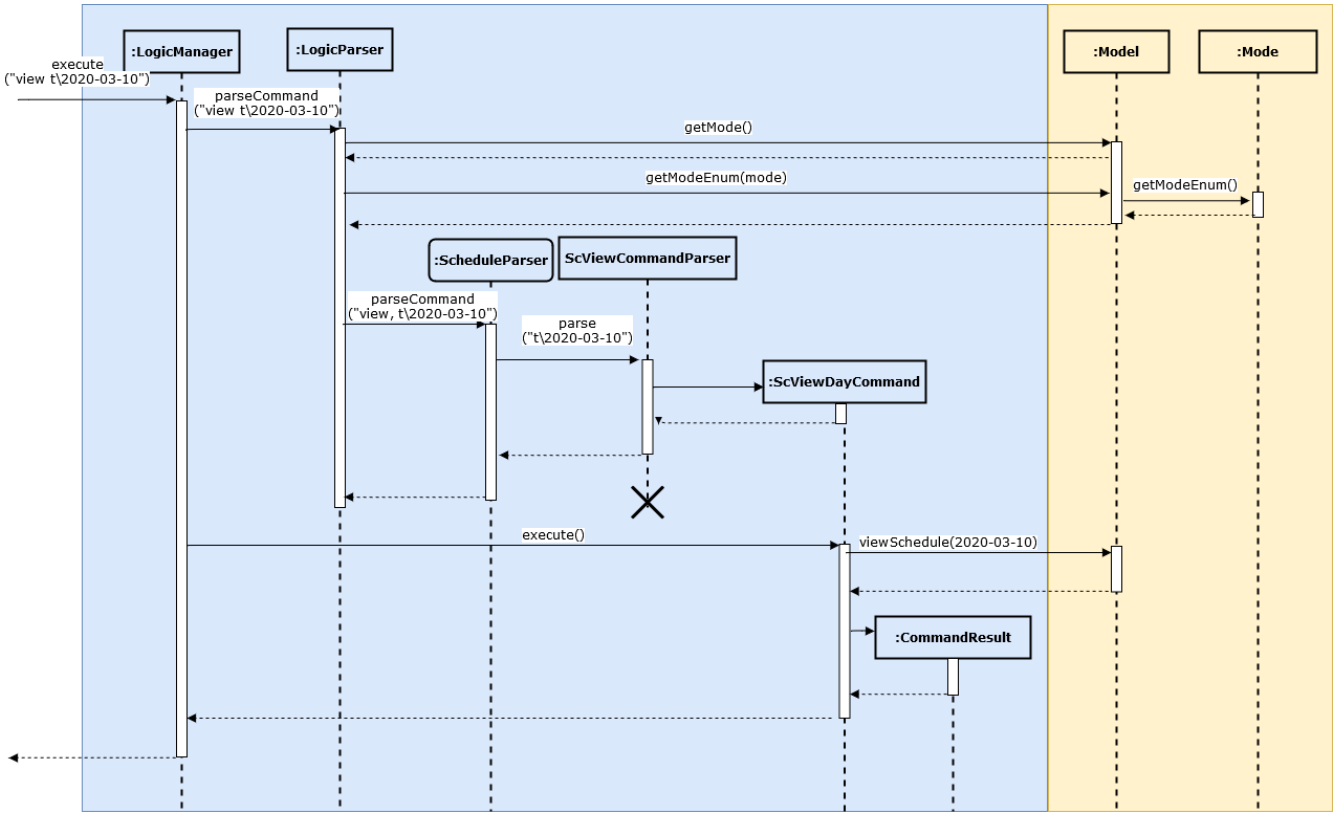
3.4.1. Implementation - View schedule by date

The view feature allows users to see the events happening on the specified date. This feature is facilitated by `ScheduleParser`, `ScViewCommandParser` and `ScViewDayCommand`. The operation is exposed in the `Model` interface as `Model#view(LocalDate)`.

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

1. The user keys in 'view t\2020-03-10' into the command box.
2. The user executes 'view t\2020-03-10' to view their schedule on the 10 Mar 2020.
3. `LogicManager` calls LogicParser to parse the command.
4. `LogicParser` gets the mode from Model and passes the command word and the argument to ScheduleParser.
5. `ScheduleParser` checks the command word and calls ScViewDayCommandParser.
6. 'ScViewDayCommandParser' creates a new 'ScViewDayCommand'.
7. 'LogicManager' executes the 'ScViewDayCommand'.
8. 'ModelManger#viewSechdule(LocalDate)' is called and the schedule for the day is retrieved.

The following sequence diagram shows how the view tasks operation works:



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

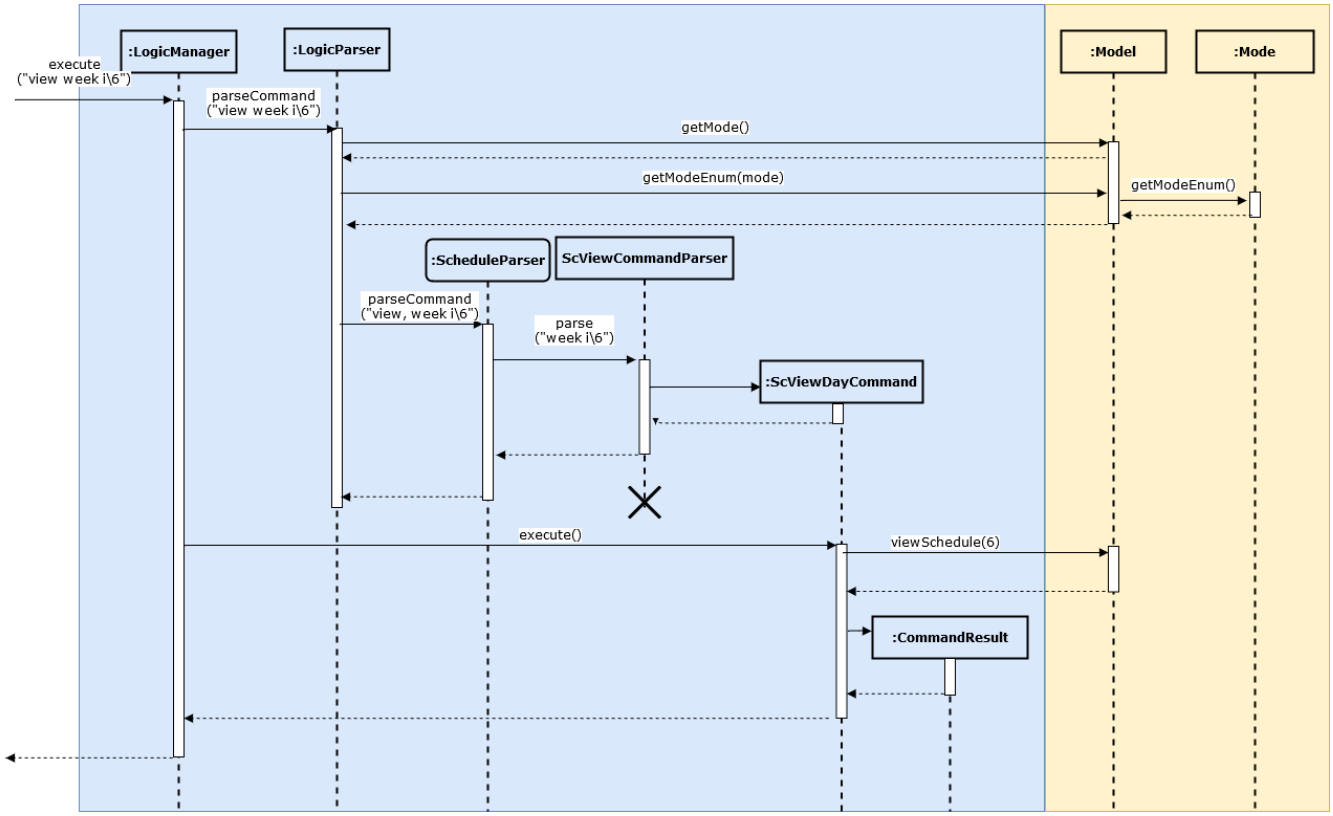
1. The user keys in 'view week i\6' into the command box.
2. The user executes 'view week i\6' to view their schedule on the 6th week.
3. `LogicManager` calls LogicParser to parse the command.
4. `LogicParser` gets the mode from Model and passes the command word and the argument to ScheduleParser.
5. `ScheduleParser` checks the command word and the preamble and calls ScViewWeekCommandParser.
6. 'ScViewWeekCommandParser' creates a new 'ScViewWeekCommand'.

7. 'LogicManager' executes the 'ScViewWeekCommand'.
8. 'ModelManger#viewSechdule(int)' is called and the schedule for the week is retrieved.

3.4.2. Implementation - View schedule by week

The view feature allows users to see the events happening throughout the specified week. This feature is facilitated by `ScheduleParser`, `ScViewCommandParser` and `ScViewWeekCommand`. The operation is exposed in the `Model` interface as `Model#view(int)`.

The following sequence diagram shows what happens when a user inputs a view week command:



3.4.3. Design Considerations

Aspect: View schedule by at most week and not month.

- **Alternative 1 (current choice):** View schedule up to week
  - Pros: Easier to fit the events into the display.
  - Cons: Less ways for user to view schedule.
- **Alternative 2:** View schedule up to month
  - Pros: User can see their whole month's schedule at once.
  - Cons: Might be too long and cannot fit into the display box. Time to gather all the events may be too long.

3.5. View available free slot of a specific day

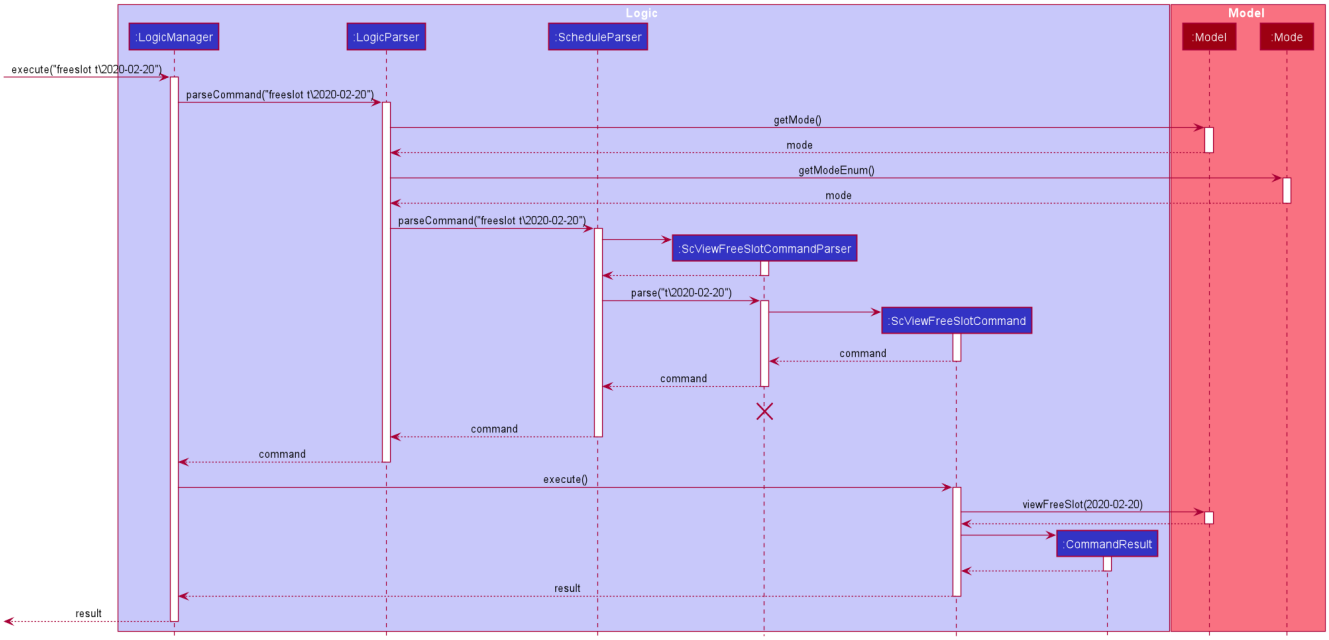
The view free slots feature allows the user to view their available free slots on their schedule.

3.5.1. Implementation

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

1. The user keys in 'freeslot t\2020-03-10' into the command box.
2. The user executes 'freeslot t\2020-03-10' to view the free slots on their schedule on the 10th of March 2020.
3. 'ScViewFreeSlotCommandParser' creates a new 'ScViewFreeSlotCommand'.
4. 'LogicManager' executes the 'ScViewFreeSlotCommand'.
5. 'ModelManger#viewFreeSlot()' is called and the free slots for the day is retrieved.

The following sequence diagram shows how the view tasks operation works:



3.5.2. Design Considerations

Aspect: Calculating free slots given a schedule

- **Alternative 1 (current choice):** Embeds a free slot data structure to keep track of the free slots whenever events are added
  - Pros: no need to calculate free slots whenever user execute freeslot.
  - Cons: overhead to add event commands, making its execution slower.
- **Alternative 2:** Calculates free slot based on the events whenever user executes freeslot
  - Pros: easier to implement.
  - Cons: slower freeslot execution.

3.6. Schedule events based on a task in plan.

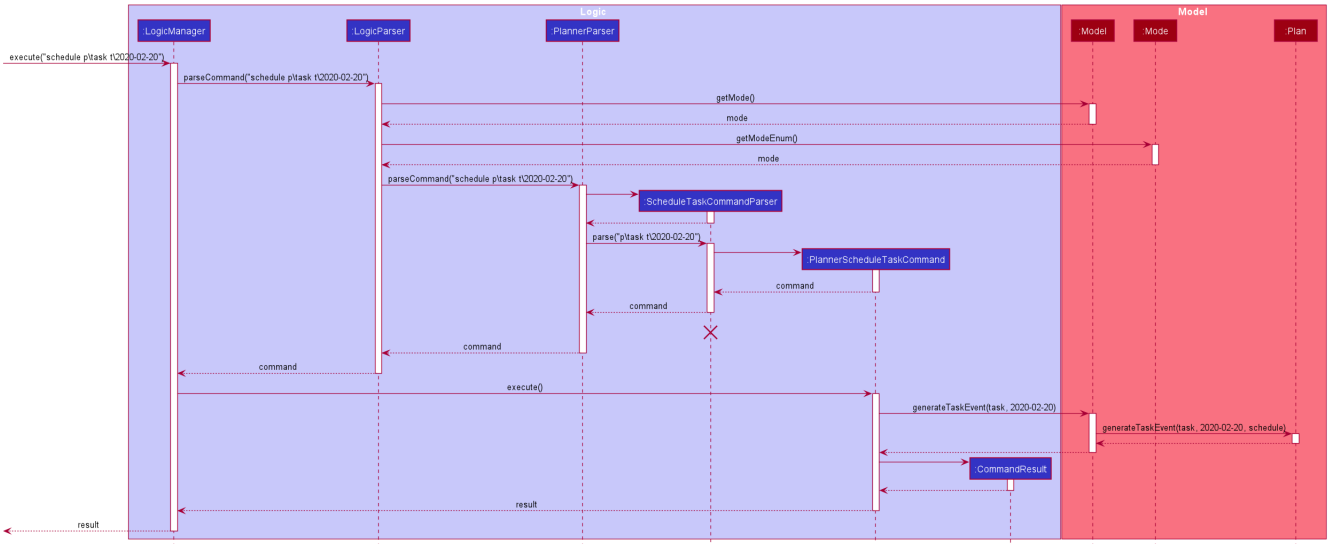
The plan feature allows the user to create an event based on the task user created in the plan.

3.6.1. Implementation

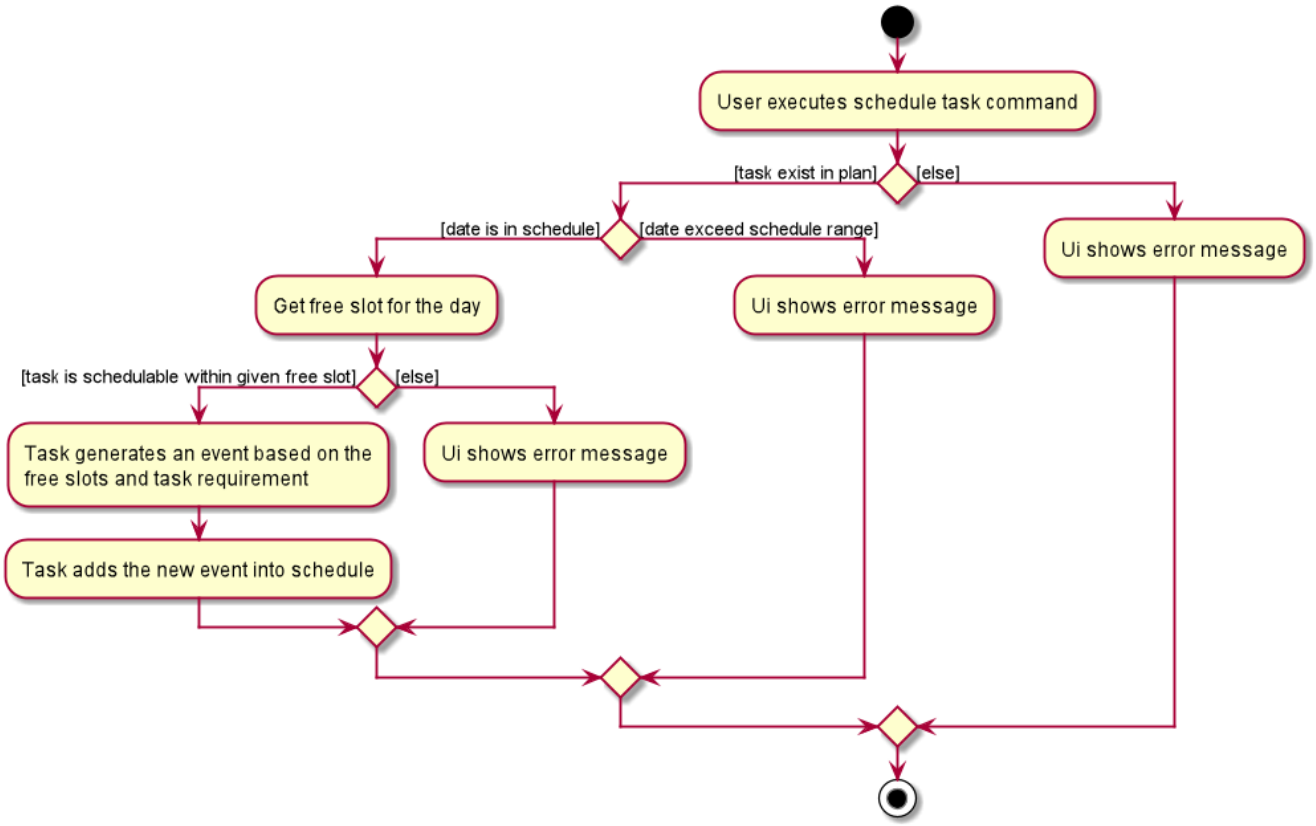
Given below is an example usage scenario and how the plan task mechanism behaves at each step.

1. The user keys in 'schedule p\task name t\2020-03-10' into the command box.
2. The user executes 'schedule p\task name t\2020-03-10' to create an event "task name" on their schedule on the 10th of March 2020.
3. 'PlannerScheduleTaskCommandParser' creates a new 'PlannerScheduleTaskCommand'.
4. 'LogicManager' executes the 'PlannerScheduleTaskCommand'.
5. 'ModelManger#searchTask()' is called to search for the task user specified.
6. 'ModelManger#generateTaskEvent()' is called and one event with time determined by algorithm is created on the day in schedule.

The following sequence diagram shows how the schedule task operation works:



The following activity diagram summarizes what happens when a user schedules a task:



3.6.2. Design Considerations

Aspect: Calculating best fit time frame for a task

- **Alternative 1 (current choice):** Plan gets free slot from schedule and generate event based on it.
  - Pros: Isolation of modules.
  - Cons: Redundant code.
- **Alternative 2:** Schedule decides whether to schedule or discard an event generated from task.
  - Pros: more robust schedule.
  - Cons: more difficult to implement.

3.7. 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 3.8, “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

3.8. 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:

- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps
- is a CS2103T student

Value proposition: a one-stop study aid platform for CS2103T students

Appendix B: User Stories

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

Priority	As a …	I can …	So that I …
* * *	student	add classmate or teammate's name and contact information	contact them easily
* * *	student	edit classmate or teammate's name and contact information	Edit them if the information are changed
* * *	student	delete classmate or teammate's contact	delete if not necessary anymore
* * *	student	categorise contacts into teammate or classmate	sort contacts according to category
* * *	student	add category specific remark	filter out a contact's remark according to type of contact
* * *	student	edit category specific remark	edit remark if necessary
* * *	student	delete category specific remark	delete remark if it is not needed
*	student	add profile picture to added contact	know at a glance who is the person in my contact
*	student	edit profile picture to added contact	edit the picture if changes are necessary
*	student	delete profile picture to added contact	don't need the contact picture within NOVA anymore
* * *	student	create meeting events	can keep track of my schedule
* * *	student	create study session events	can keep track of my schedule
* * *	student	create consultation events	can keep track of my schedule
* * *	student	create lesson events	can keep track of my schedule
* * *	student	note down the location of the meeting	know where to go
* * *	student	delete events	can get rid of events that I do not need anymore
* * *	student	add notes to events	can jot down additional details about the events
* *	student	mark events as done	know which events I have completed
* *	student	find events	can check if I have any specific events according to keyword(s)
* *	student	repeat events	can add multiple similar events at one go
* * *	forgetful student	keep track of my project tasks	make sure all my project tasks are completed on time
* * *	student	mark tasks as done	track how many tasks I have finished

Priority	As a …	I can …	So that I …
* * *	student	add notes to project tasks	keep track of details regarding the tasks
* * *	student	add project tasks	keep track of those project tasks
* * *	student	delete project tasks	remove unwanted tasks from the tracker
* *	lazy student	edit project tasks	can correct mistakes made to task descriptions with little effort
* *	lazy student	edit notes to project tasks	can correct mistakes made to notes with little effort
* * *	Student	View my schedule for a day	Know the flow of events on that day
* * *	Student	View my schedule for a week	Know what will happen for that week
*	Student	View my timetable	Can check when my classes are
* *	Student	Easily find my free slots without looking at my schedule	Do not need to strain my eyes
* *	Forgetful student	Set reminders for upcoming events	Will remember to attend them
* * *	Student	Add tasks to study plan	Can add study tasks to my study plan
* * *	Student	Delete tasks on study plan	Can delete study tasks if I don't need it anymore
* * *	Student	Generate event from a task	If I feel like I want to do a study task today, I can generate an event on today's schedule so that I can keep up with my study plan.
* *	Student	View statistics of my task progress	Can see how much I've done for each task on my study plan.

{More to be added}

Appendix C: Use Cases

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

Use case 1: Add a contact (Loh Sze Ying)

MSS

1. Student enters add command with the contact's name, phone number, email and category
2. NOVA saves the contact

   Use case ends.

Extensions

1a. Student did not include compulsory field

1a1. NOVA informs student to include compulsory field

Use case resumes at step 1

1b. Student did not adhere to format required for adding contact

1b1. NOVA informs student that the format is invalid, and provides an example of a correct format

Use case resumes at step 1

Use case 2: List all contacts (Loh Sze Ying)

MSS

1. Student enters list command
2. NOVA list all the contacts

   Use case ends

Extensions

1a. There is no contacts saved

1a1. NOVA informs student that the list is empty

Use case end

Use case 3: List category specific contacts (Loh Sze Ying)

MSS

1. Student enters list category command
2. NOVA list all the contacts under that category

   Use case ends

Extensions

1a. There is no contacts saved under that category

1a1. NOVA informs student that the list of that category is empty

Use case end

Use case 4: Find saved contacts (Loh Sze Ying)

MSS

1. Student enters find command
2. NOVA finds the name of contact that the student typed and list all the matching names

Use case ends

Extensions

1a. There is no contact that matches what the student type

1a1. NOVA prints an empty list

Use case end

Use case 5: Edit a contact (Loh Sze Ying)

MSS

1. Student used `list`, `list c\classmate`, `list c\teammate` or `find` command
2. Student enters edit command with index, and at least 1 field to edit
3. NOVA saves the edited contact

Use case ends

Extensions

1a. Student did not use `list`, `list c\classmate`, `list c\teammate` or `find` command before using `edit` command

1a1. Student edits information of the wrong contact, and NOVA informs the student to use `undo` command if wrong contact is edited

Use case resumes at step 1

2a. Student did not adhere to format required for editing contact

2a1. NOVA informs student that the format is invalid, and provides an example of a correct format

Use case resumes at step 2

2b. Student did not include at least 1 compulsory field

2b1. NOVA informs student to include at least 1 compulsory field

Use case resumes at step 2

Use case 6: Delete a contact (Loh Sze Ying)

MSS

1. Student used `list`, `list c\classmate`, `list c\teammate` or `find` command
2. Student enters delete command
3. NOVA deletes the contact

Use case ends

Extensions

1a. Student did not use `list`, `list c\classmate`, `list c\teammate` or `find` command before using `delete` command

1a1. Student deletes the wrong contact, and NOVA informs the student to use `undo` command if wrong contact is deleted

Use case resumes at step 1

2a. NOVA cannot find the contact in the contact list

2a1. NOVA informs student that the contact to delete does not exist

Use case resumes at step 2

2b. Student provides a wrong format to delete

2b1. NOVA informs student that the format is invalid, and provides an example of a correct format

Use case resumes at step 2

Use case 7: Add, edit or delete remark to a contact (Loh Sze Ying)

MSS

1. Student used `list`, `list c\classmate`, `list c\teammate` or `find` command
2. Student enters remark command
3. NOVA adds, edits or deletes remark to a contact

Use case ends

Extensions

1a. Student did not use `list`, `list c\classmate`, `list c\teammate` or `find` command before using `remark` command

1a1. Student adds, edits or deletes remark of the wrong contact, and NOVA informs the student to use `undo` command if student add, edit or delete remark on the wrong contact

Use case resumes at step 1

2a. NOVA cannot find the contact in the contact list

2a1. NOVA informs student that the contact to add, edit or delete mark does not exist

Use case resumes at step 2

2b. Student provides a wrong format to add, edit or delete

2b1. NOVA informs student that the format is invalid, and provides an example of a correct format

Use case resumes at step 2

Use case 8: Undoing in address book (Loh Sze Ying)

MSS

1. Student used `add`, `edit`, `delete` or `remark` prior
2. Student enters undo command
3. NOVA undone the changes that the student made

Use case ends

1a. Student did not use `add`, `edit`, `delete` or `remark` prior to using `undo`

1a1. NOVA informs the student that there are no more commands to undo

resumes at step 1

Use case 9: Redoing in address book (Loh Sze Ying)

MSS

1. Student used `undo` successfully prior to using `redo`
2. Student enters redo command
3. NOVA redo the changes that the student made

Use case ends

Extensions

1a. Student did not use `undo` prior to using `redo`

1a1. NOVA informs the student that there are no more commands to redo

Use case resumes at step 1

Use case 10: Adding a consultation event (Chua Huixian)

MSS

1. Student enters consultation command with details of the consultation
2. NOVA adds consultation event to the schedule

Use case ends.

Extensions

1a. NOVA detects error in data inputted

- 1a1. NOVA informs student of the error

Use case ends.

Use case 11: Delete an event (Chua Huixian)

MSS

1. Student enters delete command with details of the event
2. NOVA deletes the event

Use case ends.

Extensions

1a. NOVA cannot find the event given

- 1a1. NOVA informs student that the event does not exist

Use case ends.

Use case 12: Adding a note to an event (Chua Huixian)

MSS

1. Student enters note command with details of the event
2. NOVA adds note to the event

Use case ends.

Extensions

1a. NOVA cannot find the event given

- 1a1. NOVA informs student that the event does not exist

Use case 13: Add task to a project of progress tracker (Yap Wen Jun Bryan)

MSS

1. User enter command to add task to a project.
2. Progress tracker adds task to the project.

Use case ends.

Extensions

1a. No such project exist.

1a1. NOVA shows an error message.

Use case ends.

Use case 14: Edit a task (Yap Wen Jun Bryan)

MSS

1. User enter command to edit task.
2. Progress tracker replaces old task description with new description.

Use case ends.

Extensions

1a. No such task exist.

1a1. NOVA shows an error message.

Use case ends.

C.1. Use case 15: Delete a task (Yap Wen Jun Bryan)

MSS

1. User enter command to delete task.

2. Progress tracker deletes task.

   Use case ends.

Extensions

   1a. Task to be deleted does not exist.

   1a1. NOVA shows error message.

   Use case ends.

Use case 16: List tasks in a week of a project (Yap Wen Jun Bryan)

MSS

1. User enter command to list tasks.
2. Progress tracker lists task.

   Use case ends.

Extensions

   1a. No such tasks exists in the week specified.

   1a1. NOVA shows error message.

   Use case ends.

Use case 17: Set an added task as done (Yap Wen Jun Bryan)

MSS

1. User enter command to set task as done.
2. Progress tracker sets task as done.

   Use case ends.

Extensions

   1a. No such tasks exists.

   1a1. NOVA shows error message.

   Use case ends.

Use case 18: Add notes to a task in progress tracker (Yap Wen Jun Bryan)

MSS

1. User enter command to add notes to the project task.
2. Progress tracker adds notes to the project task.

   Use case ends.

Extensions

   1a. No such project task exist.

   1a1. NOVA shows an error message.

   Use case ends.

Use case 19: Edit a note (Yap Wen Jun Bryan)

MSS

1. User enter command to edit note.
2. Progress tracker replace old note with new note.

   Use case ends.

Extensions

   1a. No prior note was added.

   1a1. NOVA shows an error message.

   Use case ends.

Use case 20: Delete a note (Yap Wen Jun Bryan)

MSS

1. User enter command to delete note.
2. Progress tracker deletes note.

   Use case ends.

Extensions

   1a. Note to be deleted does not exist.

1a1. NOVA shows error message.

Use case ends.

Use case 21: View the schedule for a day

MSS

1. User requests for the schedule of a day.
2. NOVA shows the schedule for the day.

Use case ends.

Extensions

1a. User enters the wrong format.

1a1. NOVA displays the correct format for the command.

2a. The schedule for the day is empty.

2a1. NOVA displays that day does not have any events.

Use case ends.

Use case 22: View the schedule for a week

MSS

1. User requests for the schedule of a week.
2. NOVA shows the schedule for the week.

Use case ends.

Extensions

1a. User enters the wrong format.

1a1. NOVA displays the correct format for the command.

2a. The schedule for the week is empty.

2a1. NOVA displays that week does not have any events.

Use case ends.

Use case 23: User add a task into study plan.

MSS

1. User enter command to create a task with name specified by user.

2. Study Planner of NOVA adds the task into study plan.

   Use case ends.

Extensions

   1a. There is already a task with the same name.

   1a1. NOVA shows error message.

   Use case ends.

Use case 24: User add a task into study plan

MSS

1. User enter command to delete a task with name specified by user.
2. Study Planner of NOVA deletes the task.

   Use case ends.

Extensions

   1a. No task with the name specified exists in study plan.

   1a1. NOVA shows error message.

   Use case ends.

Use case 25: User view statistics of every tasks in study plan

MSS

1. User enter command to view statistics of every tasks in study plan.
2. NOVA calculates and shows all the statistics of every task.

   Use case ends.

Extensions

   1a. No task in study plan.

   1a1. NOVA shows error message.

   Use case ends.

Use case 26: User schedules a task into a particular

day.

MSS

1. User enter command to schedules a task into a particular day.
2. NOVA generates and adds the event into schedule.

   Use case ends.

Extensions

   1a. Unable to generate event.

   1a1. NOVA shows error message.

   Use case ends.

{More to be added}

Appendix D: Non Functional Requirements

1. The application should work on any mainstream OS provided that Java 11 or above is installed.
2. The application should work on both 32-bit and 64-bit environments.
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. The application should work without internet connection.
5. The application should respond to every command within one second.
6. The application should be easily modifiable to meet changing curriculum of CS2013T.

{More to be added}

Appendix E: Glossary

Mainstream OS

Windows, Linux, OS-X

Appendix F: 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.

F.1. Launch

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.

F.2. Shutdown (Yap Wen Jun Bryan)

1. Exiting the application
    a. Enter `exit` in the command line.

F.3. Deleting a person in Address Book (Loh Sze Ying)

1. Enter address book mode via `nav ab`
2. Deleting a person while all persons are listed
    a. Prerequisites: List contacts using the `list`, `list c\classmate`, `list c\teammate` or `find` command. There are multiple contacts in the list.
    b. Test case: `delete i\1`
    Expected: First contact is deleted from the list. Details of the deleted contact shown in the status message.
    c. Test case: `delete i\0`
    Expected: No person is deleted. Error details shown in the status message. Status bar remains the same.
    d. Other incorrect delete commands to try: `delete`, `delete i\x` (where x is larger than the list size), `delete x` (where x is number or letter) Expected: Similar to previous.

F.4. Adding a project task in Progress Tracker (Yap Wen Jun Bryan)

1. Enter progress tracker mode via `nav progresstracker`
2. Adding a project task
    a. Test case: `add p\ip w\1 d\new task`
    Expected: New project task with description "new task" is added to IP project week 1. Message showing successful execution is shown.

b. Test case: `add p\ip w\14 d\new task`
   Expected: No new project task is added. Error details shown in the status message.

c. Other incorrect add commands to try: `add`, `add p\ppp w\1 d\new task`.
   Expected: Similar to previous.

F.5. Saving data (Yap Wen Jun Bryan)

1. Dealing with corrupted data file

   a. Edit the Nova data file to contain invalid data.
      Example: Adding a @ to a week in a ptTask.

   b. Run the application. The application should start with an empty Nova data file.

2. Dealing with missing data file

   a. Delete the current Nova data file.

   b. Run the application. The application should start with a sample Nova data file with pre-included data.

F.6. View schedule (Terence)

1. Enter schedule mode via `nav schedule`
2. View the schedule

   a. Prerequisite: An event has been added to 13 Jan 2020.

   b. Test case: `view t\2020-01-13`
      Expected: The event that was pre-added will be displayed.

   c. Test case: `view t\2020-01-12`
      Expected: Error message is shown to tell user that the date is out of the range of schedule.