Plan²travel - Developer Guide

1. Setting up . . . 1
2. Design . . . 1
   2.1. Architecture . . . 2
   2.2. UI component . . . 4
   2.3. Logic component . . . 5
   2.4. Model component . . . 6
   2.5. Storage component . . . 8
   2.6. Common classes . . . 11
3. Implementation . . . 11
   3.1. Undo/Redo feature . . . 11
   3.2. Schedule optimisation . . . 16
   3.3. [Proposed] Schedule Activity feature . . . 18
   3.4. Logging . . . 19
   3.5. Configuration . . . 20
   3.6. Timetable . . . 20
4. Documentation . . . 20
5. Testing . . . 21
6. Dev Ops . . . 21
Appendix A: Product Scope . . . 21
Appendix B: User Stories . . . 21
Appendix C: Use Cases . . . 23
Appendix D: Non Functional Requirements . . . 24
Appendix E: Glossary . . . 25
Appendix F: Product Survey . . . 25
Appendix G: Instructions for Manual Testing . . . 25
   G.1. Launch and Shutdown . . . 25
   G.2. Deleting a contact . . . 26
   G.3. Saving data . . . 26

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

1. Setting up

Refer to the guide here.

2. Design

2.1. Architecture

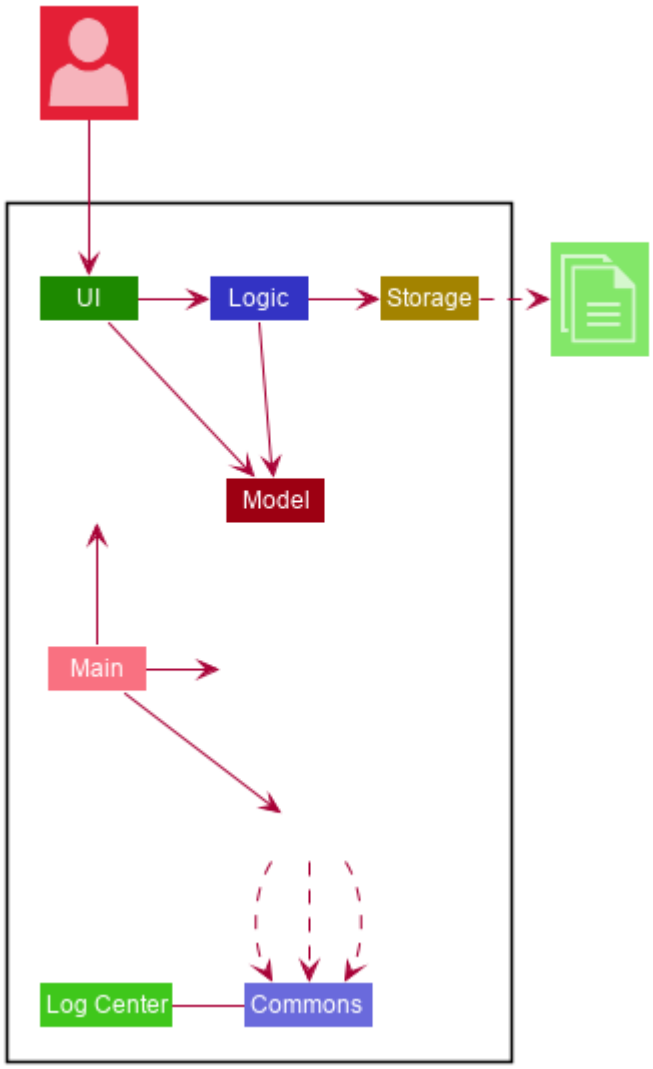


Figure 1. Architecture Diagram

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

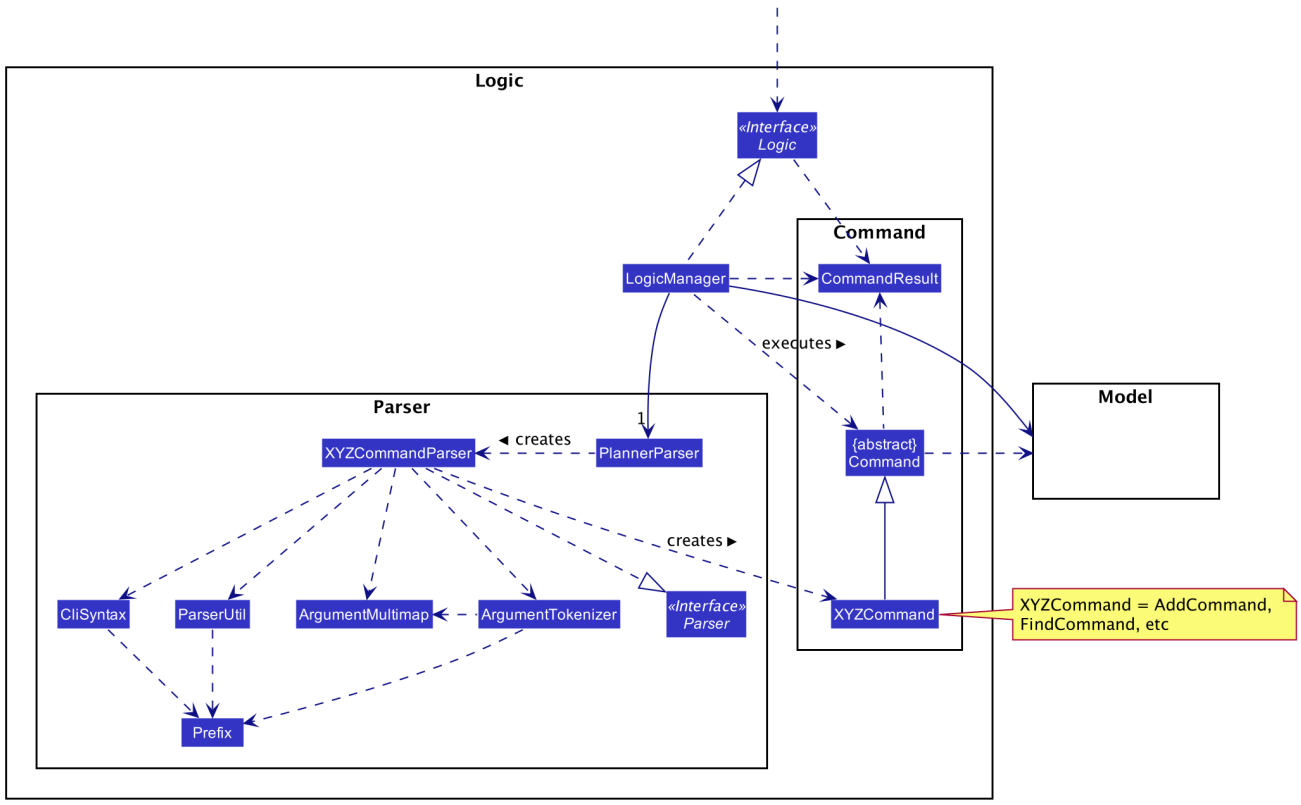


Figure 2. Class Diagram of the Logic Component

How the architecture components interact with each other

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

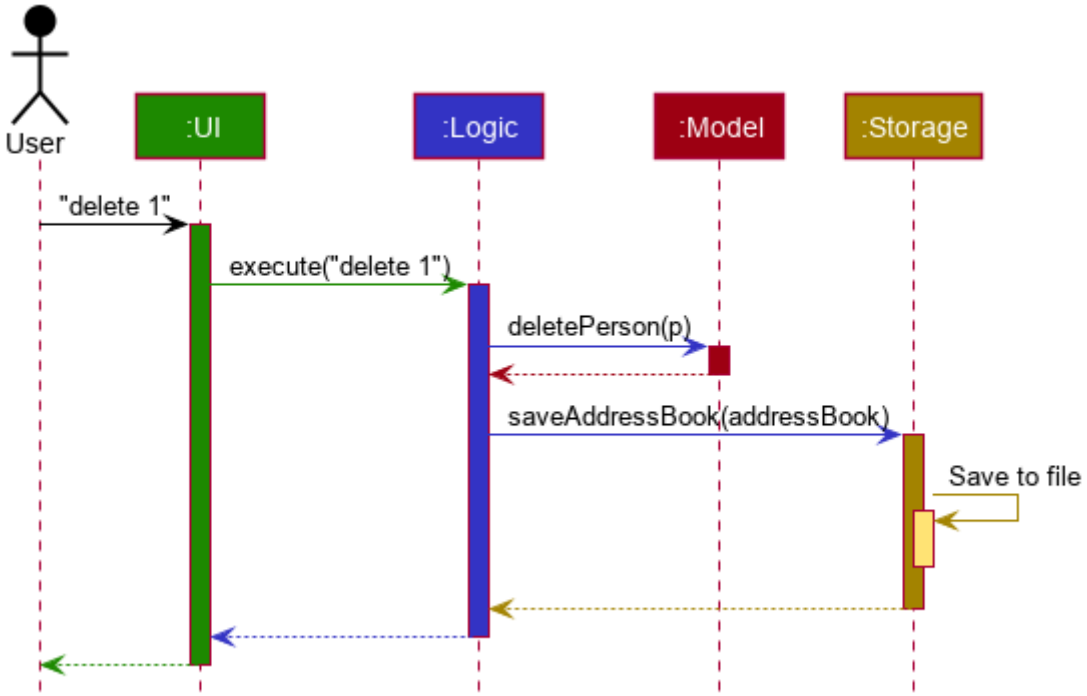


Figure 3. Component interactions for `delete 1` command

The sections below give more details of each component.

2.2. UI component

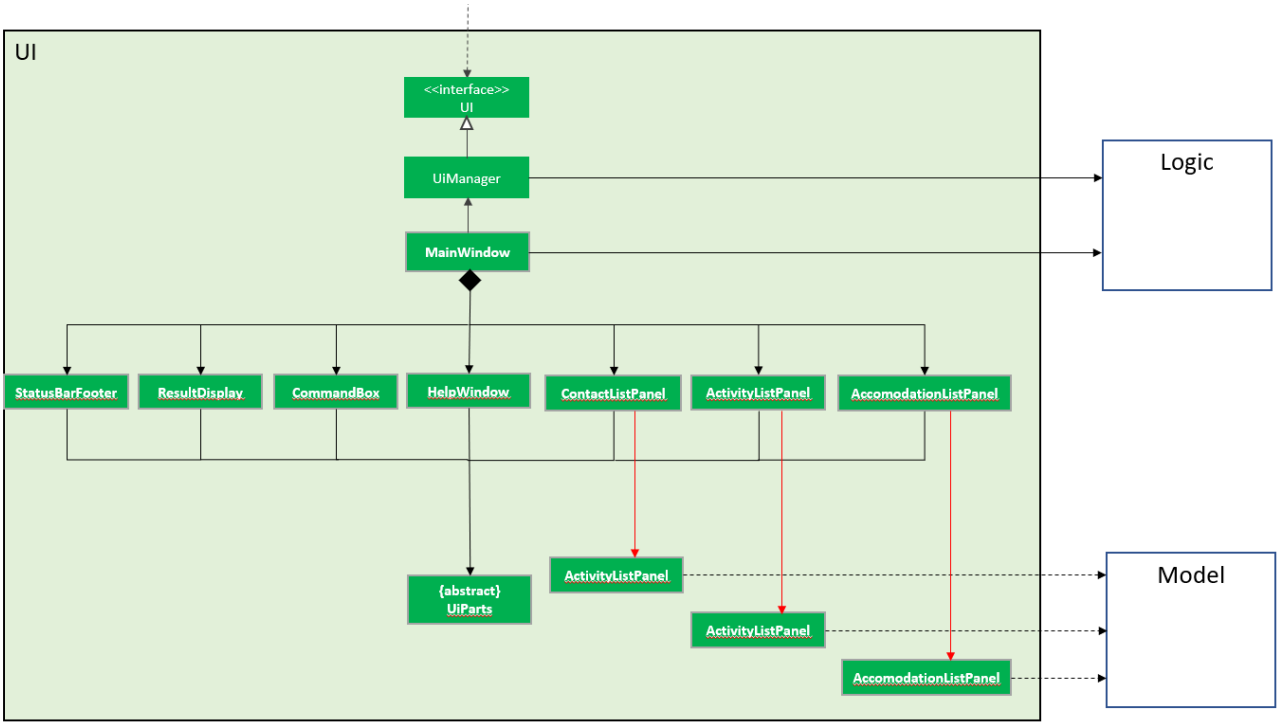


Figure 4. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `FeedbackDisplay`, `ContactListPanel`, `ActivityListPanel`, `AccomodationListPanel`, `StatusBarFooter`, `HelpWindow` 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

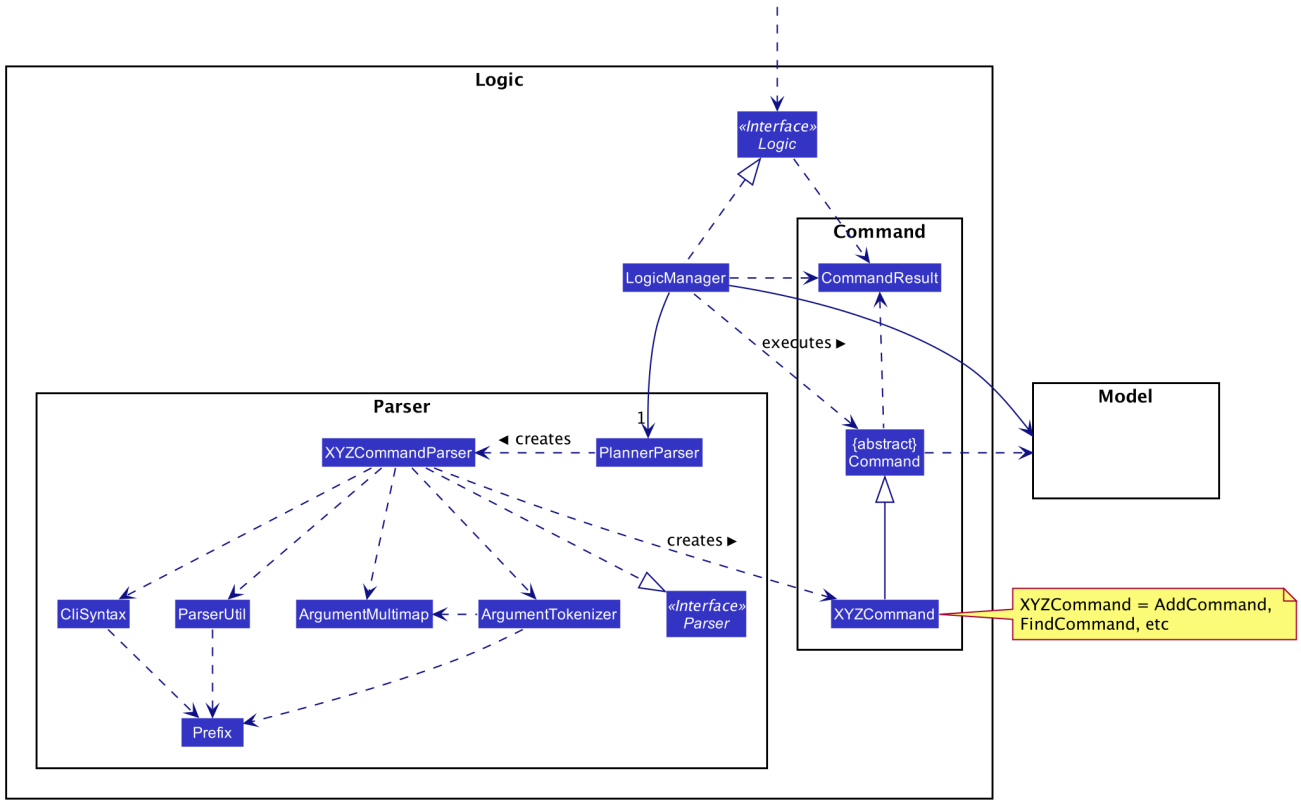


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `PlannerParser` 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 contact).
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("add activity n/Climb Fuji a/Mount Fuji")` API call.

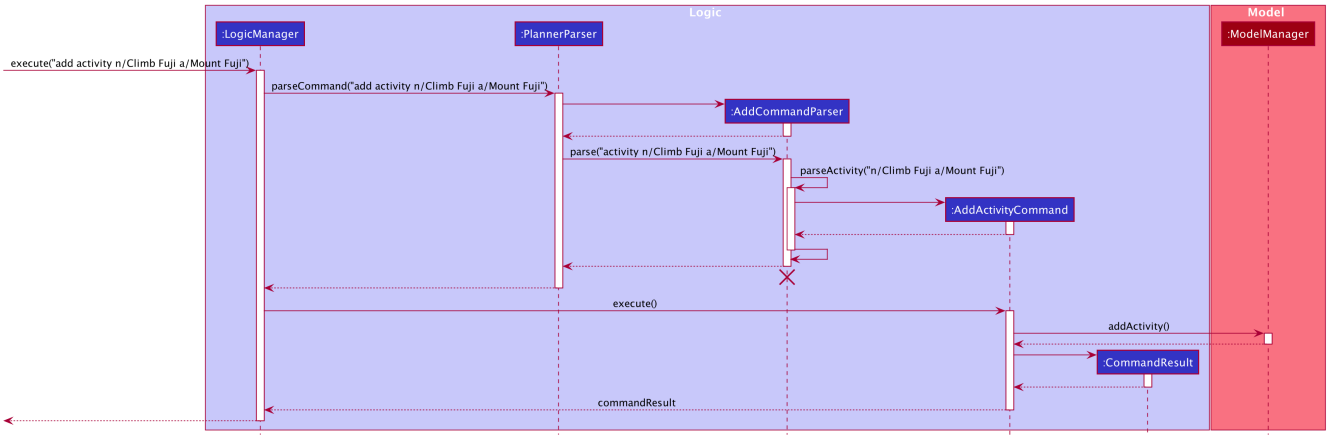


Figure 6. Interactions Inside the Logic Component for the `add activity n/Climb Fuji a/Mount Fuji` Command

NOTE	The lifeline for `AddCommandParser` 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

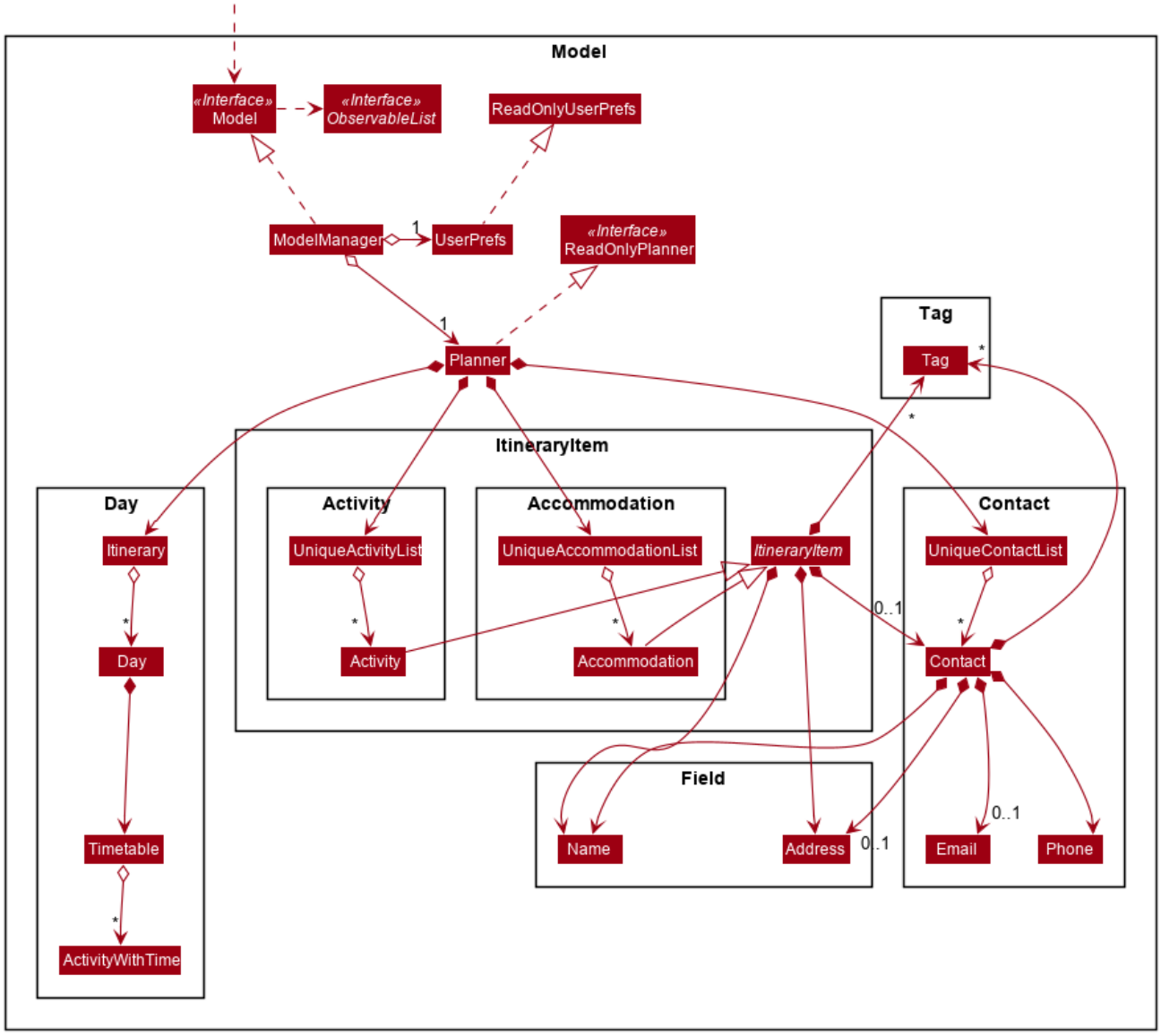


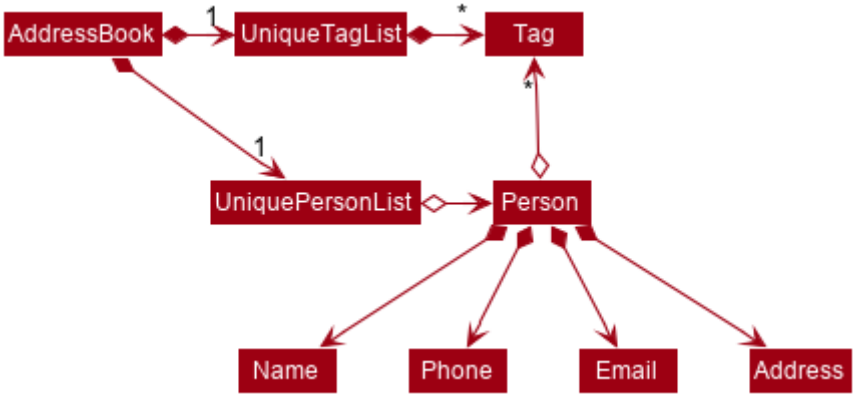
Figure 7. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the Planner data.
- 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.

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



2.5. Storage component

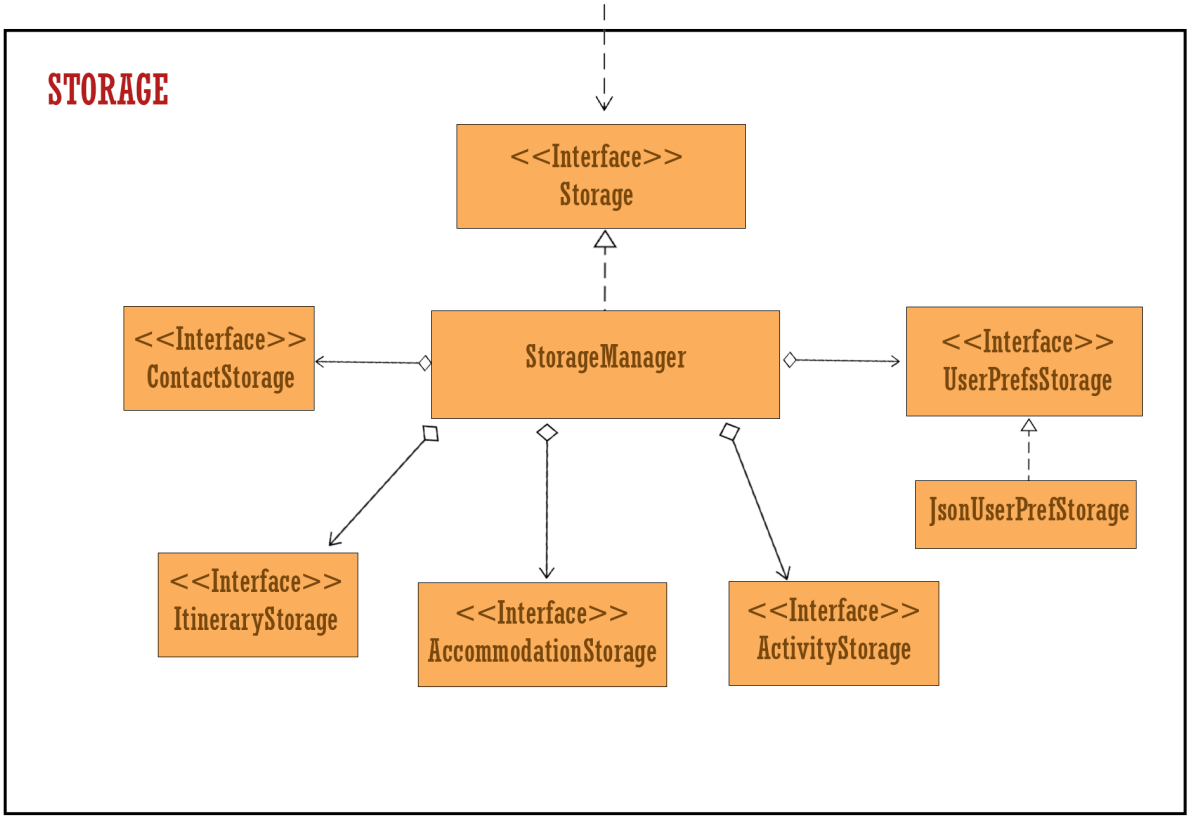


Figure 8. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

- can save `UserPref` objects in json format and read it back.
- application data is split into 4 different components, `Accommodation`, `Activity`, `Contact`, `Itinerary`.

2.5.1. Accommodation Storage component

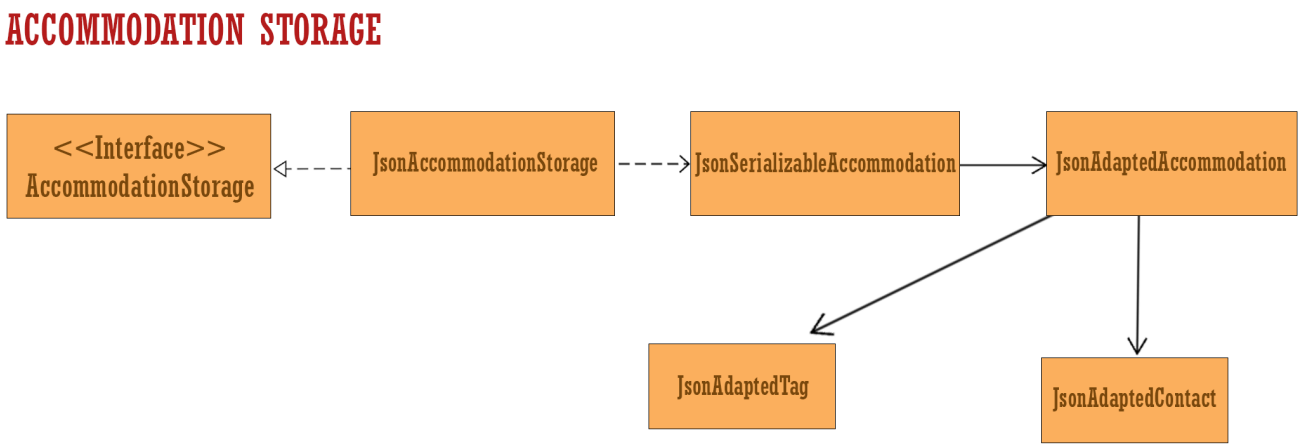


Figure 9. Structure of Accommodation Storage component

API : `AccommodationStorage.java`

The `Accommodation Storage` component,

- can save `Accommodation` objects in json format and read it back.
- as `Tag is a field in Accommodation`, similarly JsonAdaptedTag is a field within JsonAdaptedAccommodation. Hence the association as shown in the diagram above.
- as `Contact is a field in Accommodation`, similarly JsonAdaptedContact is a field within JsonAdaptedAccommodation. Hence the association as shown in the diagram above.

2.5.2. Activity Storage component

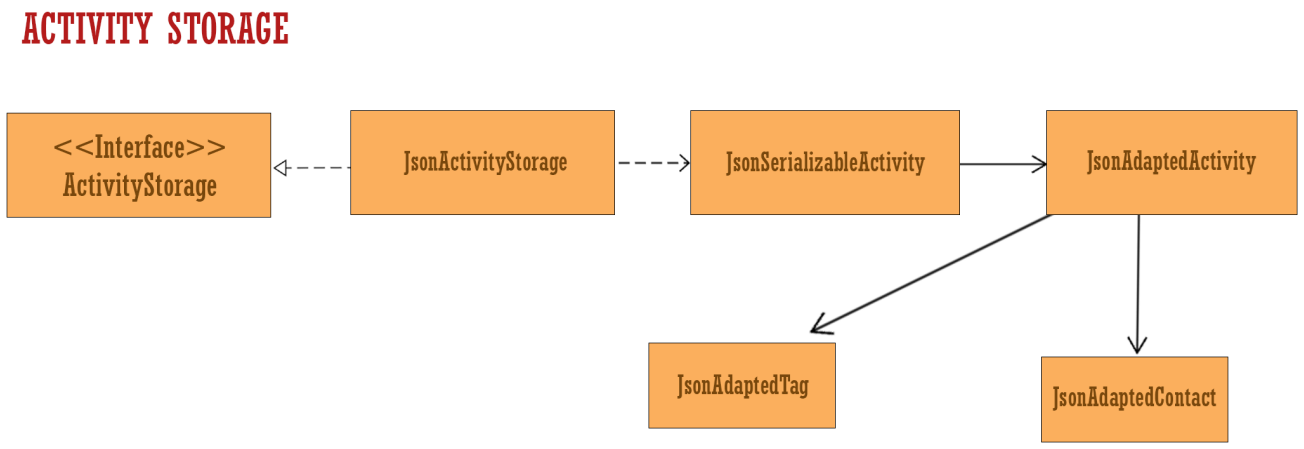


Figure 10. Structure of Activity Storage component

API : `ActivityStorage.java`

- can save `Activity` objects in json format and read it back.
- as `Tag is a field in Activity`, similarly JsonAdaptedTag is a field within JsonAdaptedActivity. Hence the association as shown in the diagram above.
- as `Contact is a field in Activity`, similarly JsonAdaptedContact is a field within

JsonAdaptedActivity. Hence the association as shown in the diagram above.

2.5.3. Contact Storage component

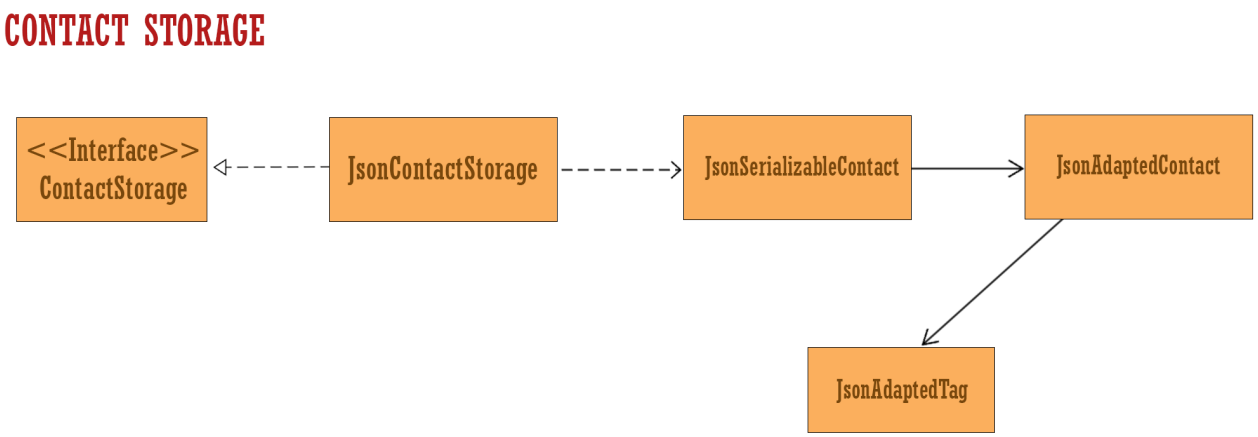


Figure 11. Structure of Contact Storage component

API : `ContactStorage.java`

- can save `Contact` objects in json format and read it back.
- as `Tag is a field in Contact`, similarly JsonAdaptedTag is a field within JsonAdaptedContact. Hence the association as shown in the diagram above.

2.5.4. Itinerary Storage component

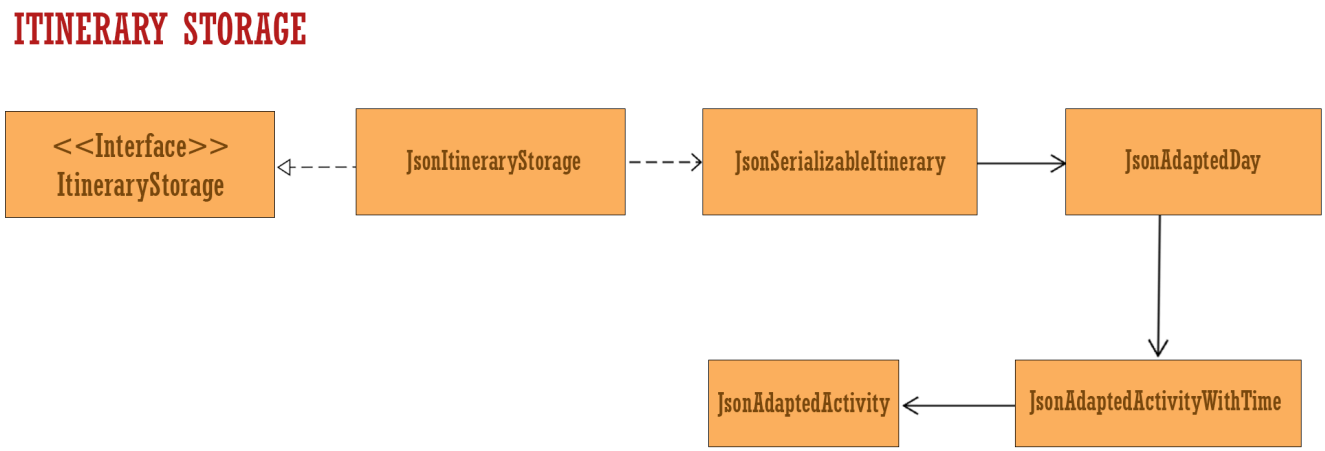


Figure 12. Structure of Itinerary Storage component

API : `ItineraryStorage.java`

- can save `Itinerary` objects in json format and read it back.
- as `ActivityWithTime is a field in Day`, similarly JsonAdaptedActivityWithTime is a field within JsonAdaptedDay. Hence the association as shown in the diagram above.
- as `Activity is a field in ActivityWithTime`, similarly JsonAdaptedActivity is a field within

JsonAdaptedActivityWithTime. Hence the association as shown in the diagram above.

2.6. Common classes

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

3. Implementation

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

3.1. Undo/Redo feature

The undo command allows user to `undo by one command` (if the command is `undoable`). The redo command allows user to `return to the original state before the latest undo is called`.

3.1.1. Implementation

The undo/redo feature utilizes various classes to operate, such as `CommandHistory` class and the classes within the `events package` in the logic component.

`CommandHistory` is a `static` class that contains the undoEventStack and redoEventStack of the application, each `containing Events`.

KEY IDEA:

`Event` — Every UndoableCommand can be wrapped into its own unique Event. `undo/redo` — Every Event has an undo and redo method. `EventFactory` — An Event object is generated by the EventFactory when executing the UndoableCommand.

List of UndoableCommand:

add activity/ accommodation/ contact/ days
delete activity/ accommodation/ contact/ day
edit activity/ accommodation/ contact
schedule
unschedule
autoschedule
optimise
clear

Step 1. The user executes an `UndoableCommand`.

Step 2. The `UndoableCommand` is executed, generating an `Event` in the process.

Step 3. `EventFactory` will parse the `UndoableCommand` to generate an Event.
(eg. DeleteActivityCommand will result in DeleteActivityEvent generated)

NOTE	EventFactory is a `static class` that will parse an UndoableCommand and generate the corresponding Event.

Step 4. `Event` is added to `undoEventStack` stored in CommandHistory. The `redoEventStack` in CommandHistory is also `cleared` upon generating a new Event.

Step 5. The `UndoableCommand` has been executed, returning a `CommandResult` to be shown.

Step 6. To undo the previous UndoableCommand, the user executes **`undo`** command. An `UndoCommand` is generated.

Step 7. `UndoCommand is not an UndoableCommand`. Executing the UndoCommand gets the Event from the top of undoEventStack and calls the undo method of `Event`.

NOTE	Both UndoCommand and RedoCommand are not UndoableCommands, no Events are generated.

Step 8. The `Event` is popped from the undoEventStack and pushed to redoEventStack in CommandHistory. A CommandResult is returned and the `Event is undone`.

Step 9. To redo the command that has been undone, the user executes **`redo`** command. A `RedoCommand` is generated. This execution is `similar to steps 6 and 7`, except Event is popped from `redo stack instead and pushed to undo stack`.

The **following two sequence diagrams** shows how the user's input is handled for **`'delete activity 1'`**.

The first diagram below shows the execution of `delete activity 1` command. It shows how the `Event` is generated and how the `undoEventStack and redoEventStack is updated`.

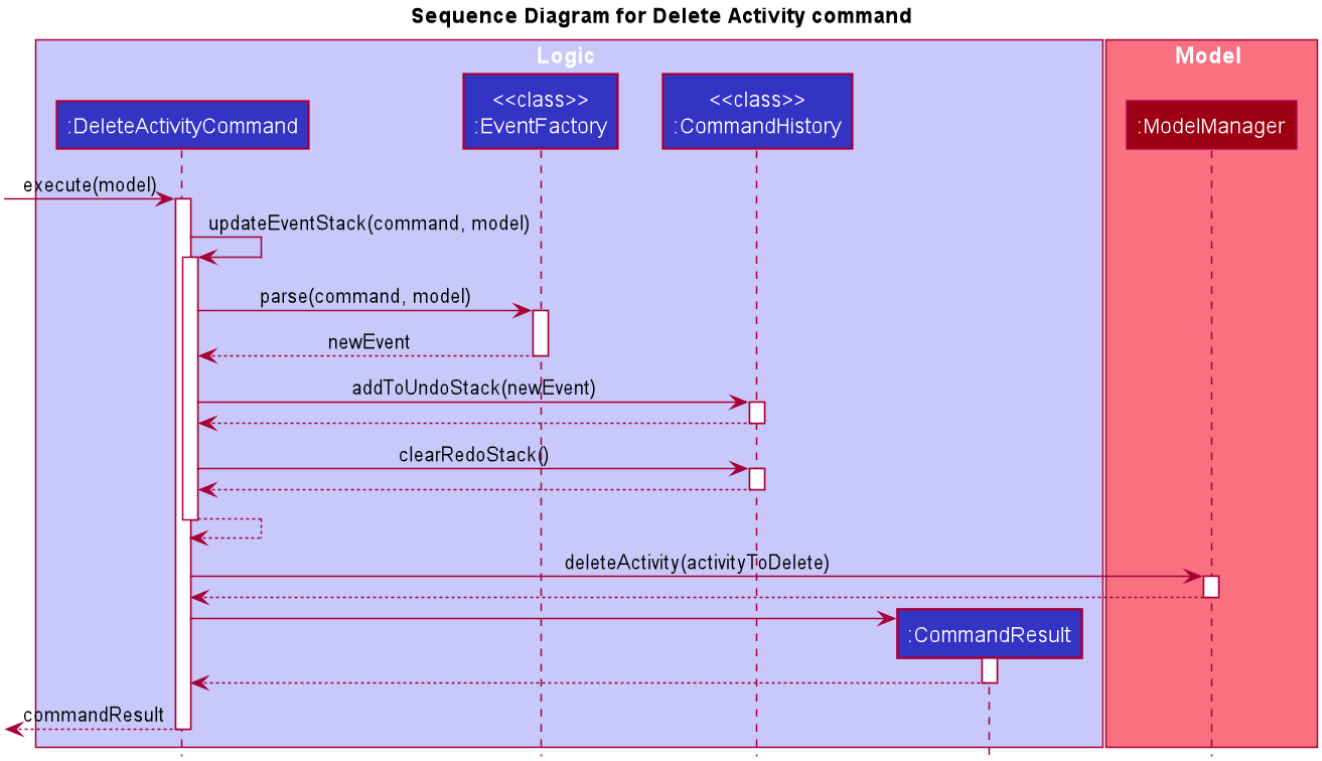


Figure 13. Executing delete activity

NOTE

When DeleteActivityCommand is executed, it generates the `activityToDelete` by extracting the `Activity` to be deleted from the Model's list of activities based on the index specified in user's command input. It then calls the Model's `deleteActivity` method.

The second diagram below shows the execution of `undo` command. Executing `UndoCommand` calls undo of the `DeleteActivityEvent`, which returns an `AddActivityCommand` with the `Activity (initially deleted)` to be added back at the `correct index`. Both Activity and Index were stored in `DeleteActivityEvent`.

This new `AddActivityCommand` is executed, and the `DeleteActivityCommand` is successfully undone.

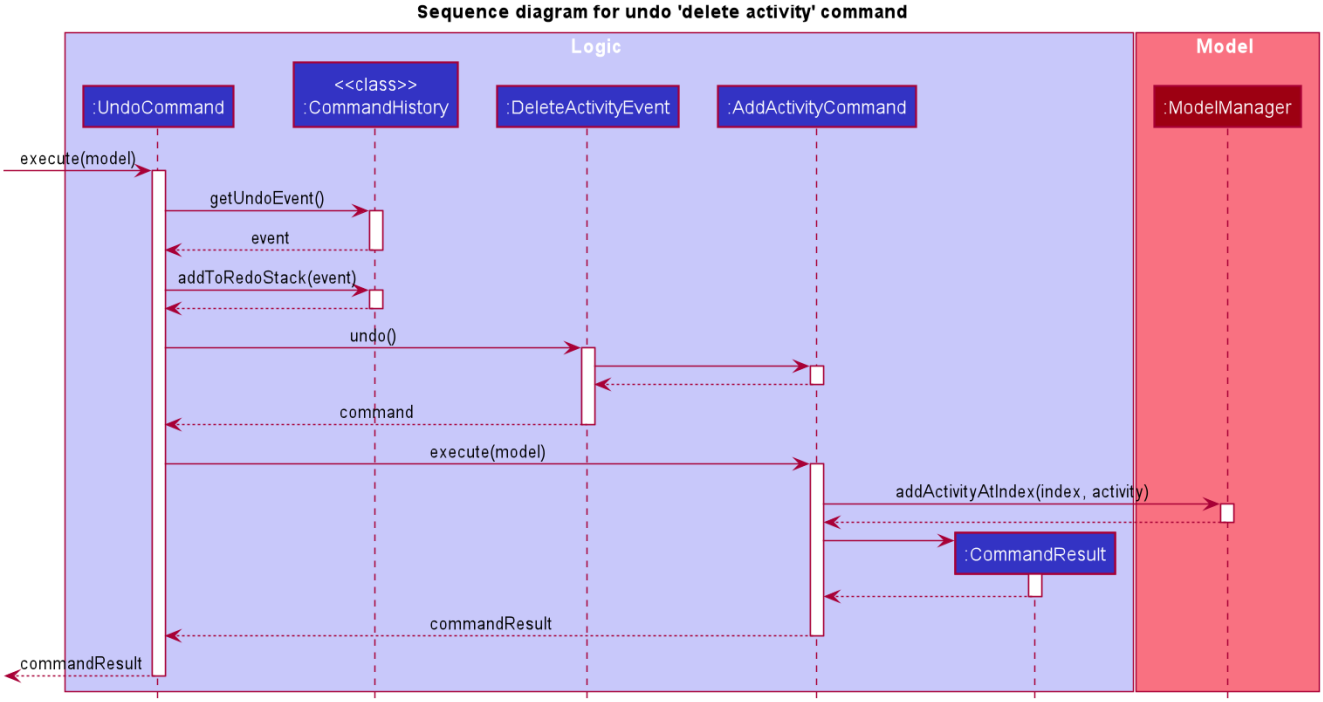
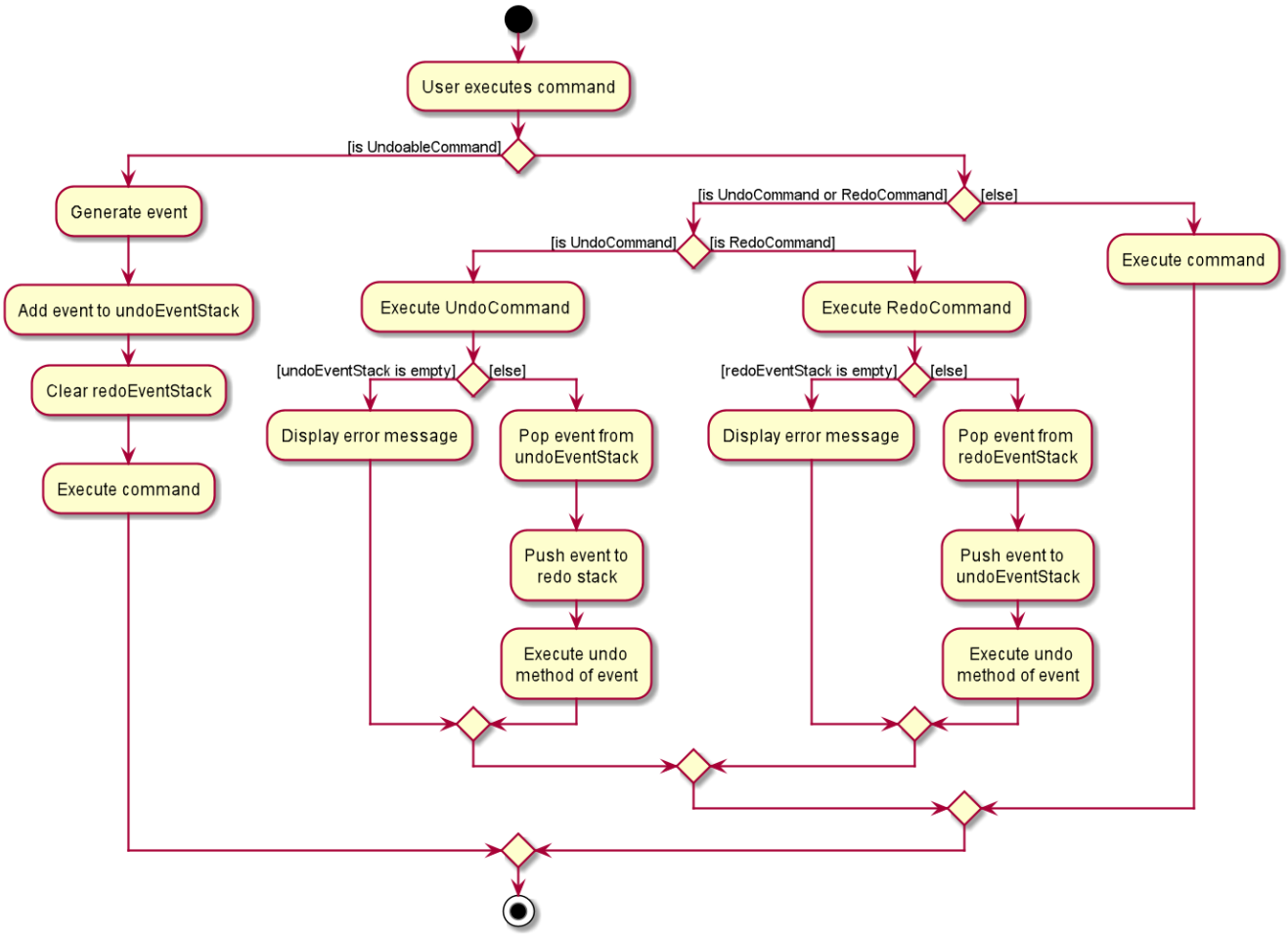


Figure 14. Executing undo for 'delete activity'

NOTE	Each Event stores the `necessary data` required by the reverse Command to undo the effects of the initial UndoableCommand. (eg. AddActivityEvent stores the `Activity added`, as `DeleteActivityCommand` requires the Activity to undo the initial `AddActivityCommand`'s changes)

The following **activity diagram** summarizes what happens when a user executes an UndoableCommand, an UndoCommand, a RedoCommand, or any other Commands.



NOTE	If the user does not execute an UndoableCommand, UndoCommand or RedoCommand, the stack of Events in CommandHistory will **not be affected**. (eg. view, list commands)

3.1.2. Design Considerations

Aspect: How undo & redo executes

- **Option 1 :** Wrap every UndoableCommand in an Event class, which has undo and redo methods.
  - Pros:
    - Uses less memory by storing Event objects rather than storing every state of the Model.
    - Convenient for future extensions for new Commands added. Just need to ensure for every UndoableCommand, there must a Command that is able to undo its changes.
    - Command classes obey Single Responsibility Principle, they do not need to know how to undo or redo itself, as it is abstracted to their corresponding Event classes.
  - Cons:
    - Every UndoableCommand requires another Command to undo its changes. Might be difficult to manage if more UndoableCommands are added.
- **Option 2 :** Saves the entire Model data (comprising of accommodations, activities, contacts and days).

    - Pros:
        - Easy to implement.
    - Cons:
        - May have performance issues in terms of memory usage. Expensive to store the various objects at every state.
- **Option 3:** Each UndoableCommand knows how to undo/redo by itself.
    - Pros:
        - Will use less memory
    - Cons:
        - We must ensure that the implementation of each individual command are correct.
        - Commands do not obey Single Responsibility Principle.
        - Commands will also violate Law of Demeter principle in the process of undo implementation.

Aspect: Data structure to support the undo/redo commands

- **Option 1 (current choice):** Use of static class CommandHistory to store a stack of Events to undo, and a stack of Events to redo.
    - Pros:
        - Easy to implement and understand. Every Event object is generated through EventFactory and stored in CommandHistory.
    - Cons:
        - Static class is used instead of Singleton implementation. No single instance of CommandHistory is created, cannot be passed as a parameter to other methods and treated as a normal object, hence might pose a difficulty during extensions.
- **Option 2:** Create a HistoryManager class to store a single list of Model objects for undo/redo
    - Pros:
        - Straightforward and easy implementation, storing a deep copy of Model whenever an UndoableCommand is executed.
    - Cons:
        - Need to keep track of the Model object obtained from the list to set during Undo/Redo. Difficult to manage the pointer in the list.

3.2. Schedule optimisation

3.2.1. Implementation

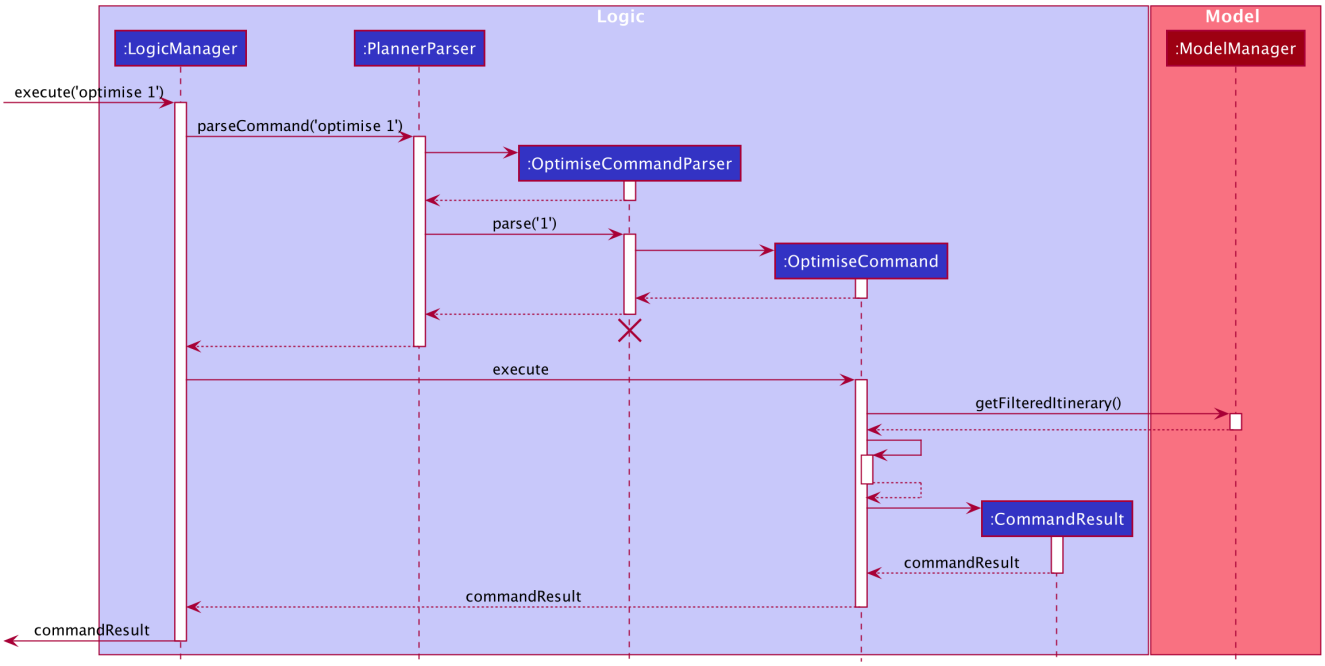
The schedule optimisation feature utilises `TimeTable` and allows the user to resolve any overlapping activities. Resulting in a schedule with the lowest overall cost and the most number of activities such that the user's time is maximised. The following command is supported:

- optimise INDEX_OF_DAY — optimises and de-conflicts the schedule. This operation supports activities with or without a cost. Activities without a cost have a default cost of $0.00.

Given below is a sample usage of the feature:

The scheduled activities in a day can be viewed as a Directed Acyclic Graph (DAG), where a path represents a possible schedule. When a user executes the `Optimise` command on a day with conflicting activities, a utility method creates an adjacency list. Another utility method then traverses the adjacency list to create all possible paths. The paths are then sorted to obtain the one with the lowest total cost and most number of activities (nodes).

Shown below is a summary of the execution of the command.



3.2.2. Design considerations

Aspect: Execution

- Alternative 1 (current): Find all paths
  - Pros:
    - Able to compare both budget and number of activities.
    - Can reuse to find different methods of optimisation besides cost, by replacing comparator.
  - Cons:
    - Slow runtime.
- Alternative 2: Find shortest path
  - Pros:
    - Faster runtime.
  - Cons:

- Less flexible optimisation.

3.3. [Proposed] Schedule Activity feature

Plan²travel allows user to schedule an activity from the activity list to a specified time of a day. This is accomplished by executing the Schedule Command using `activity index`, `day index`, `start time` and `end time` of the activity.

Eg. `schedule activity ACTIVITY_INDEX st/START_TIME et/END_TIME d/DAY_INDEX`

3.3.1. Current Implementation

The keywords from the command given by the user is parsed using `ScheduleCommandParser` which converts the string variable of start time and end time into a LocalTime format and wraps activity index and day index with an Index class. These are then passed to the `ScheduleActivityCommand` for execution later on.

Given below is a sequence diagram showing the creation of `ScheduleActivityCommand`:

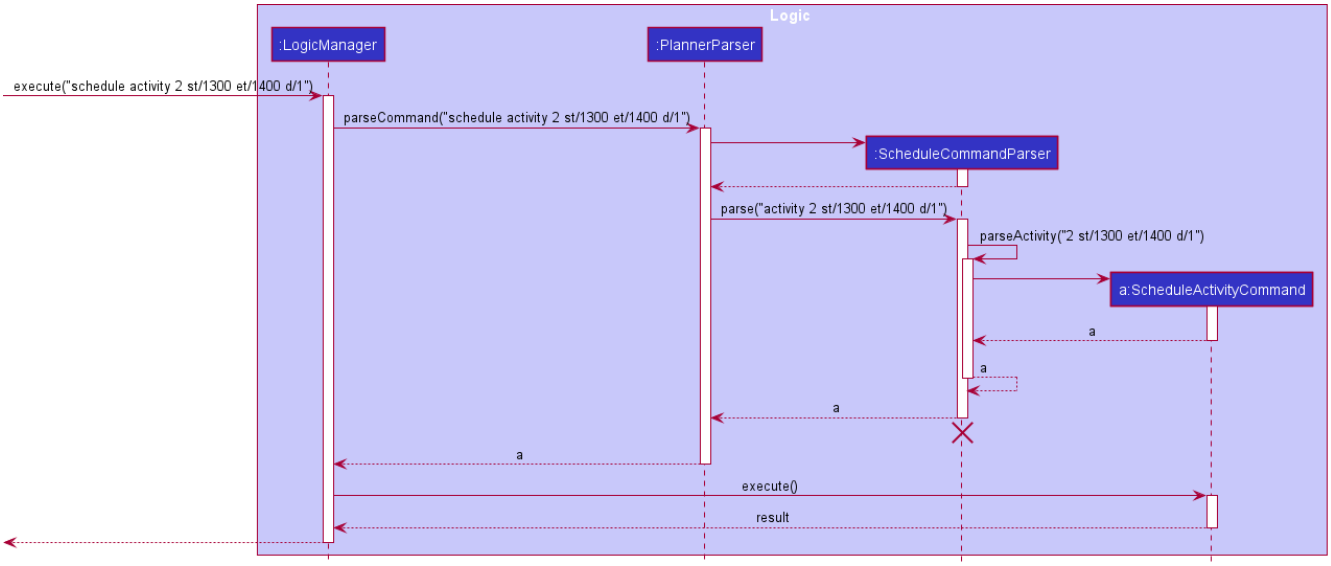


Figure 15. Sequence diagram showing how ScheduleActivityCommand is created.

After the creation of `ScheduleActivityCommand`, `LogicManager` will proceed to call the `execute()` method of `ScheduleActivityCommand`.

Below are the steps taken during the execution of `ScheduleActivityCommand`:

1. `Model` will retrieve the list of days from the `Itinerary` and the list of activities from `UniqueActivityList`.
2. The `activityIndex` and `dayIndex` will then be used to obtain the target `Activity` from activity list and target `Day` from list of days.
3. `Activity` will be converted to `ActivityWithTime` using the start time and time given.
4. This `ActivityWithTime` is then added to the list of `ActivityWithTime` for the target `Day`.
5. A new `Day` is created with the updated list of `ActivityWithTime`.

6. `Model` will replace the old `Day` with the new `Day` created in the `Itinerary`.

Given below is a sequence diagram showing the execution of `ScheduleActivityCommand`:

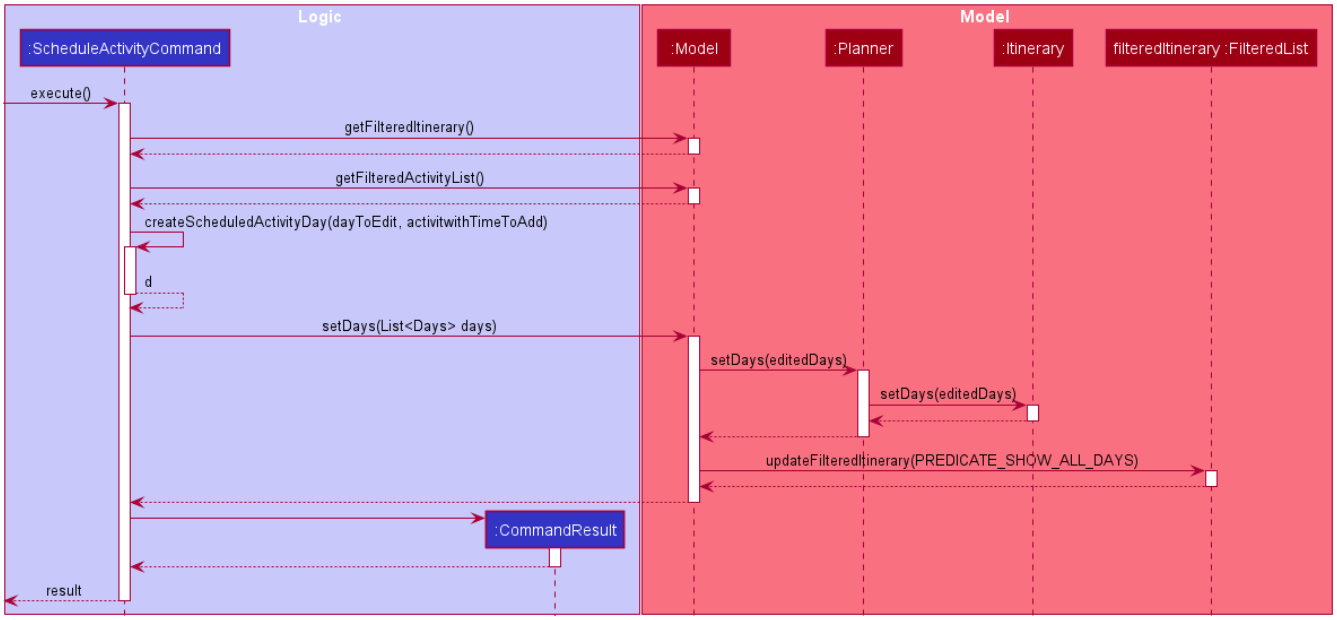


Figure 16. Sequence diagram showing how ScheduleActivityCommand executes.

3.3.2. Design Considerations

Aspect: Update activities for that particular day

- **Alternative 1 (current choice):** Create a new `Day` class with updated activity list for that day.
  - Pros: Easier for developer to test the code.
  - Cons: Might create unnecessary overheads in the code by creating new object every time we schedule an activity.
- **Alternative 2:** Directly updates the activity list in the current `Day` class.
  - Pros: Seem more intuitive and simple to implement.
  - Cons: Might make it harder to debug if other functions/classes also depends on the same `Day` class.

3.4. 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 3.5, “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

3.5. Configuration

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

3.6. Timetable

3.6.1. Current Implementation

Internally, Timetable is a TreeSet of ActivityWithTime. This allows fast access of ActivityWithTime stored in Timetable as well as giving Timetable the flexibility to accept Activities that start and end at any time instead of fixed intervals (e.g. 30 minute intervals).

To be precise, comparison between ActivityWithTime in the TreeSet is done by comparing their start times. To prevent overlapping activities, checks will be conducted before the adding. The start time of an ActivityWithTime that is added will not be before the end time of its floor neighbour and it's end time will not be after the start time of its ceiling neighbour.

3.6.2. Design considerations

- Alternative 1: an Array of time slots that stores Activity
  - Pros: Simple and intuitive to implement. UI for itinerary is easier to implement too. Very fast access to each Activity in the Timetable.
  - Cons: Constrained to fixed intervals. Hence, Activity start times and end times have to be in multiples of the fixed interval.
- Alternative 2(current choice): a TreeSet of ActivityWithTime
  - Pros: Allows flexible start times and end times. Fast access to Activity in Timetable. Does not allow ActivityWithTime objects to have the same start times.
  - Con: UI for itinerary might be difficult to implement as each the size of each block of ActivityWithTime in the UI is not the same.

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:

- a student who is inexperienced in planning for overseas trips
- has a need to manage and schedule planner items
- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: Many students wish to go for overseas trips during their holidays. They may be inexperienced in trip planning. These students would benefit from having a template as a way to organise the information they have for their trip. Plan²travel can organise information faster than a typical mouse/GUI driven app.

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…
* * *	traveller	refer to a list of attractions	I can decide on what activities to do for the day
* * *	traveller	add activities that I want to do	I can plan my trip
* * *	traveller	save my contact list	I can review/access them again
* * *	traveller	access a list of accommodations	I can better plan for where to stay at

Priority	As a …	I want to …	So that I can…
* * *	organised traveller	plan my daily planner	I can make better use of my travel time
* * *	lightweight traveller	save the planner to my phone	I can pack light
* * *	infrequent traveller	add contacts	I can get in touch with the hotel concierge
* * *	new user	view a help guide	I can familiarise myself with the application
* *	traveller	categorise activities by interest	I can prioritise certain activities
* *	solo traveller	keep a list of emergency contacts	I know who to contact in times of emergency
* *	eager traveller	follow an accommodation checklist	I will not miss out on anything
* *	indecisive traveller	plan for multiple trips	I can decide on a later date
* *	messy planner	quickly organise my travel research	I can streamline my planning processes
* *	fast typist	be auto-corrected for my typos	I won't break my train of thought while planning
* *	advanced user	use command shortcuts	I can improve my planning efficiency
* *	advanced user	set where to save my itineraries	I can organise my itineraries
* *	careless user	undo my mistakes	I don't have to retype if I make one

Priority	As a …	I want to …	So that I can…
*	traveller	rate activities that I have done	I can make a better recommendation to my friends
*	traveller on a tight budget	estimate my budget for a trip	I can minimise my spendings
*	traveller	organise and record my travel experiences	I can share them online
*	inexperienced planner	receive planning recommendations	I can improve my planner

Appendix C: Use Cases

(For all use cases below, the **System** is the `Plan²travel` application and the **Actor** is the `user`, unless specified otherwise)

Use case: Schedule activity

MSS

1. User requests to schedule activity
2. System shows a list of days and activities
3. User requests to add a specific accommodation to a specific day
4. System adds accommodation under selected day

   Use case ends.

Extensions

2a. The list of days is empty.

- Use case ends.

2b. The list of activities is empty.

- Use case ends.

3a. The day number is invalid.

3a1. System shows an error message.

Use case resumes at step 2.

3b. The accommodation index is invalid.

3b1. System shows an error message.

Use case resumes at step 2.

Use case: Add Contact

MSS

1. User requests to add a new Contact
2. System adds the new Contact into the database

Use case ends.

Extensions

1a. The new Contact's syntax is not entered correctly.

1a1. System shows a feedback to the user that the Contact was not entered correctly.

Use case ends.

Use case: Undo command

MSS

1. User requests to undo last possible command
2. System reverts to state before the last possible command

Use case ends.

Extensions

1a. There is no last possible command.

- Use case ends.

{More to be added}

Appendix D: Non Functional Requirements

Availability

1. Application should work on any mainstream OS as long as it has Java 11 or above installed.

Performance

1. Application should respond within 2 seconds of client's query.

Usability

1. Application should be easy to use for new user when following the User Guide.
2. Application's interface should be intuitive and easy to understand for the user.
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.

Reliability

1. Application should be able to execute all user's commands without failing.

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

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 contact

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