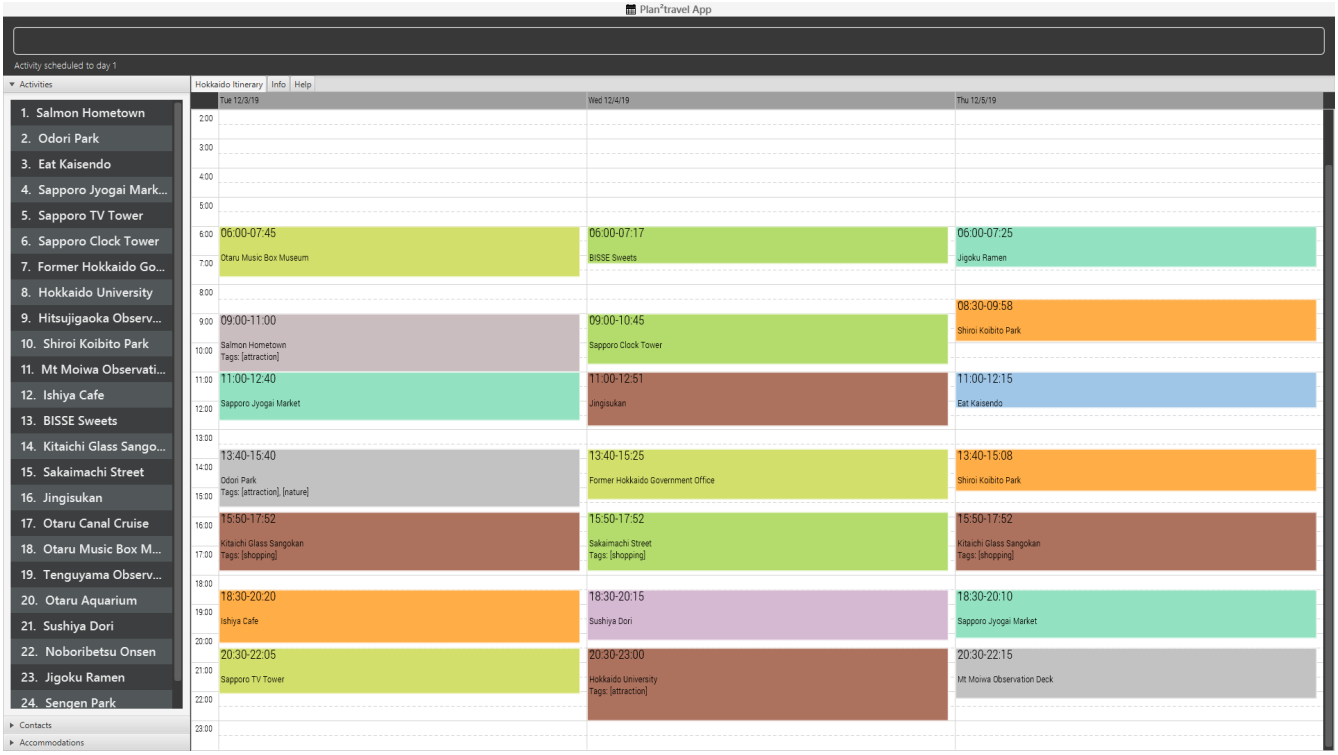
Leong Sheu Xiang - Project Portfolio

PROJECT: *Plan²travel*

Overview

Plan²travel is a travel planning application built for desktops. The program supports a GUI interface built with JavaFX but most user interactions should happen from a Command Line Interface(CLI). It is a Java application intended for students travelling to an overseas country. The program helps users quickly organise their travel information, schedule their activities and create a visually appealing itinerary.

Below is a screenshot of our application:



Summary of contributions

- **Major enhancement 1**: added **autocompletion of commands**
  - What it does: gives the user suggestions of how to complete the command input. Users can select the suggestions and the input will have the suggestion added.
  - Justification: There are many commands to learn to be proficient in the program. This gives new users more confidence when using the program and also allows advanced users to save time when typing commands. Also it prevents mistakes in spelling, especially when it comes to words like "accommodation" and "itinerary".

    - Highlights: This enhancement requires a good understanding of the different variations of command syntax in the program to be able to come up with a general way to narrow down the possible suggestions.
    - Credits: Caleb Brinkman for providing a code base to start the autocomplete feature at https://gist.github.com/floralvikings/10290131. The code had to be modified as the original autocomplete feature by Brinkman was more suited for dictionary words rather than for commands.
- **Major enhancement 2**: added the **itinerary**
    - What it does: Allow the itinerary to update and show the current activities on the itinerary.
    - Justification: Allow the user to be able to immediately see how the scheduled activities look like on a timetable and be able to better plan their itineraries.
    - Highlights: This enhancement required a good understanding of Agenda from the JFXtras library. Since the default timetable provided by Agenda was edited mostly by mouse clicks, modifications had to be made to Agenda. Also, the default Agenda displayed a static amount of days and further modifications had to be made for it to adapt to changing number of days.
- **Minor enhancement**: added the model for **Day** and **ActivityWithTime**
    - What it does: Provides the underlying data structure for each day in the itinerary. ActivityWithTime is a higher-level wrapper of Activity that contains the starting time of the activity.
    - Justification: An itinerary is naturally composed of days, hence it was helpful when using it to store activities that have been scheduled on a particular day.
    - Highlights: This enhancement interacts with the itinerary UI and different considerations had to be made to ensure the UI will be able to accurately display what was saved in the each day.
- **Other contributions**: Coded the redesign from AddressBook to plan2travel. Also, added functionality to allow information about activity, accommodation, contact and day to be displayed in the info tab via `list activity/accommodation/contact/day` or through `view activity/accommodation/contact`.
    - Enhanced the GUI (Pull requests #109, #126, #207, #212)
    - Added tests to increase coverage: #247
    - Community:
        - Over 40 Pull Requests on Github
        - PRs reviewed (with non-trivial review comments): #72, #74, #82, #87, #94, #99, #100, #106, #117, #122, #134, #193, #216, #241
        - Reported bugs and suggestions for other teams in the class (examples: 1, 2, 3, 4, 5, 6, 7)
        - Overview of Code Contribution here
    - Documentation:
        - Contributed to the User Guide and Developer Guide for this project. See below for more details.

Contributions to the User Guide

> *Given below are sections I contributed to the User Guide. They showcase my ability to write documentation targeting end-users.*

View specific tab `view`

Displays the tab specified. The available tabs to view are `itinerary`, `info` and `help`.
Format: `view TAB_NAME`

Examples:

- `view itinerary`
- `view info`
- `view help`

Each command will display the tab specified.

Set name/start date : `set`

Sets the trip's name or start date
Format: `set [n/NAME] [sd/START_DATE]`

- The name provided cannot exceed 30 characters long.
- The start date provided needs to be in dd-mm-yyyy format.

List activities in a day: `list day`

Lists the activities within day DAY_INDEX of the itinerary.
Format: `list day DAY_INDEX`

Examples:

- `list day 3`

Lists activities within day 3 of the itinerary. == Contributions to the Developer Guide

> *Given below are sections I contributed to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.*

Auto Complete feature

Rationale

There are many commands to learn in our program and some of the commands can be wordy. In order to ease the learning curve and to help the user be productive as quickly as possible,

autocomplete was implemented.

Implementation

KEY IDEA:

Every command in the planner can be broadly broken down into a few categories, namely:

1. Command Word
2. Preamble
3. Required Prefix
    a. That are used once
    b. That are used multiple times
4. Optional Prefix
    a. That are used once
    b. That are used multiple times

AutoCompleteSuggester makes use of this and checks the input for these categories one by one.

- If the input lacks the command word, then the best matching command word will be suggested.
- If the command word is provided, but the preamble is not (and it is required by the command), then the preamble will be suggested.
- If the input contains the command word and contains the preamble(when necessary), then the required prefixes and optional prefixes will be suggested. Since there are some prefix that should only be used, the input will be checked to see if these prefixes are present and remove them from the suggestions.

The `AutoCompleteParser` supports the `AutoCompleteSuggester` by parsing the user input into the command word, the preamble and a list of prefixes used in the program.

Given below is a sequence diagram showing the generation of suggestions:

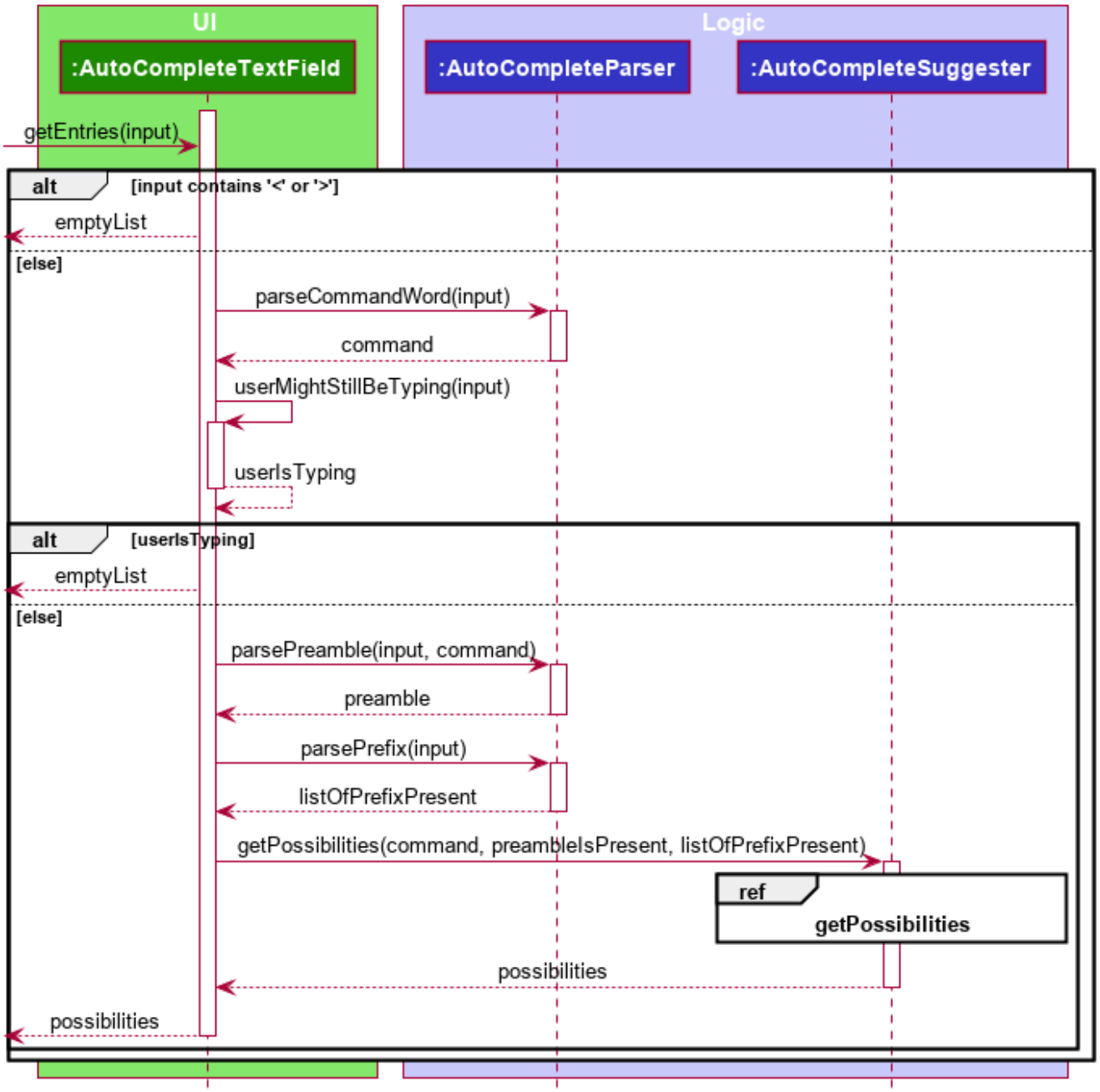


Figure 1. Sequence diagram showing the generation of suggestions.

Figure 2. Sub diagram for getPossibilities

Below is how the `autocomplete` feature is triggered:

Step 1. Modification in the text of `AutoCompleteTextField` is detected.

Step 2. If the input is empty, hide the pop-up box which contains the suggestions. If not empty, get entries (suggestions) for the pop-up box.

Step 3. If there are entries, populate the pop-up box with it. Else, hide the pop-up box.

Step 4. If pop-up box in previous step was populated, then show the pop-up box to the user.

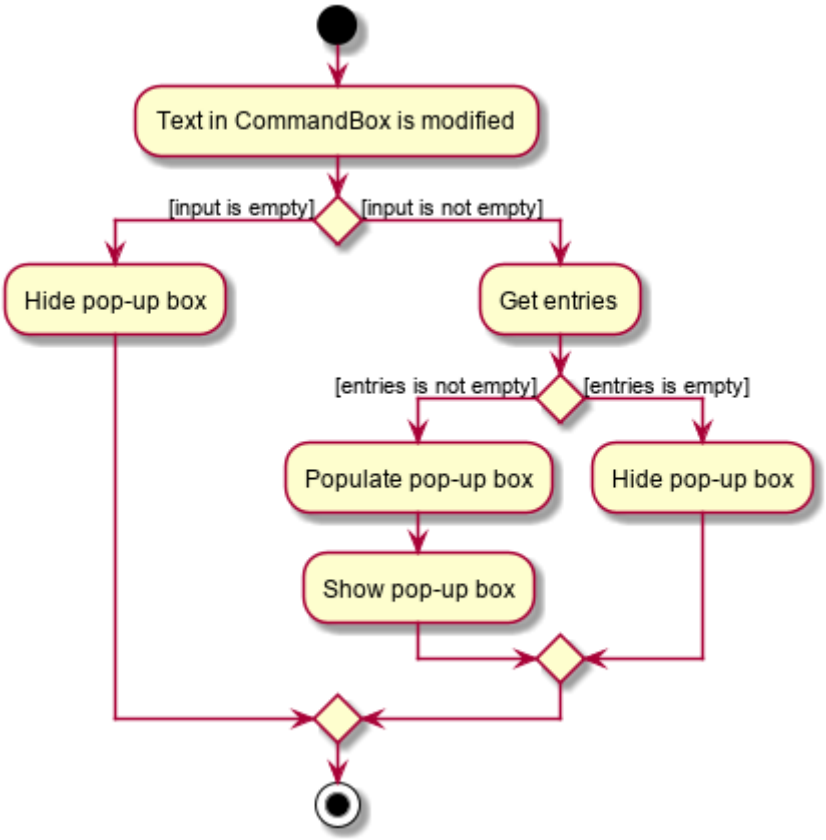


Figure 3. Activity diagram showing how the pop-up box containing the suggestions appears:

Below is how entries for the pop-up box is obtained:

Step 1. Check if input contains '<' or '>', as that would indicate that the user has accepted the suggestion of preamble, which has description enclosed with < and > (for e.g. <INDEX>) but has not replaced it with the actual preamble. If it has '<' or '>', then don't give suggestions to user.

Step 2. If the input does not contain '<' or '>', parse the command word.

Step 3. Next check if command word can be found. If it is not found, suggest command words that partially matches what the user has already inputted. If command word was found, do a check if there is a space at the end of the input.

Step 3. Next check if command word can be found. If it is not found, suggest command words that partially matches what the user has already inputted. If command word was found, do a check if there is a space at the end of the input.

Step 4. If there is no space, don't give suggestions to the user as it is assumed that the user is still typing. If there is a space, then parse preamble, parse prefix and make suggestions based on the command word, preamble and prefix present in the input.

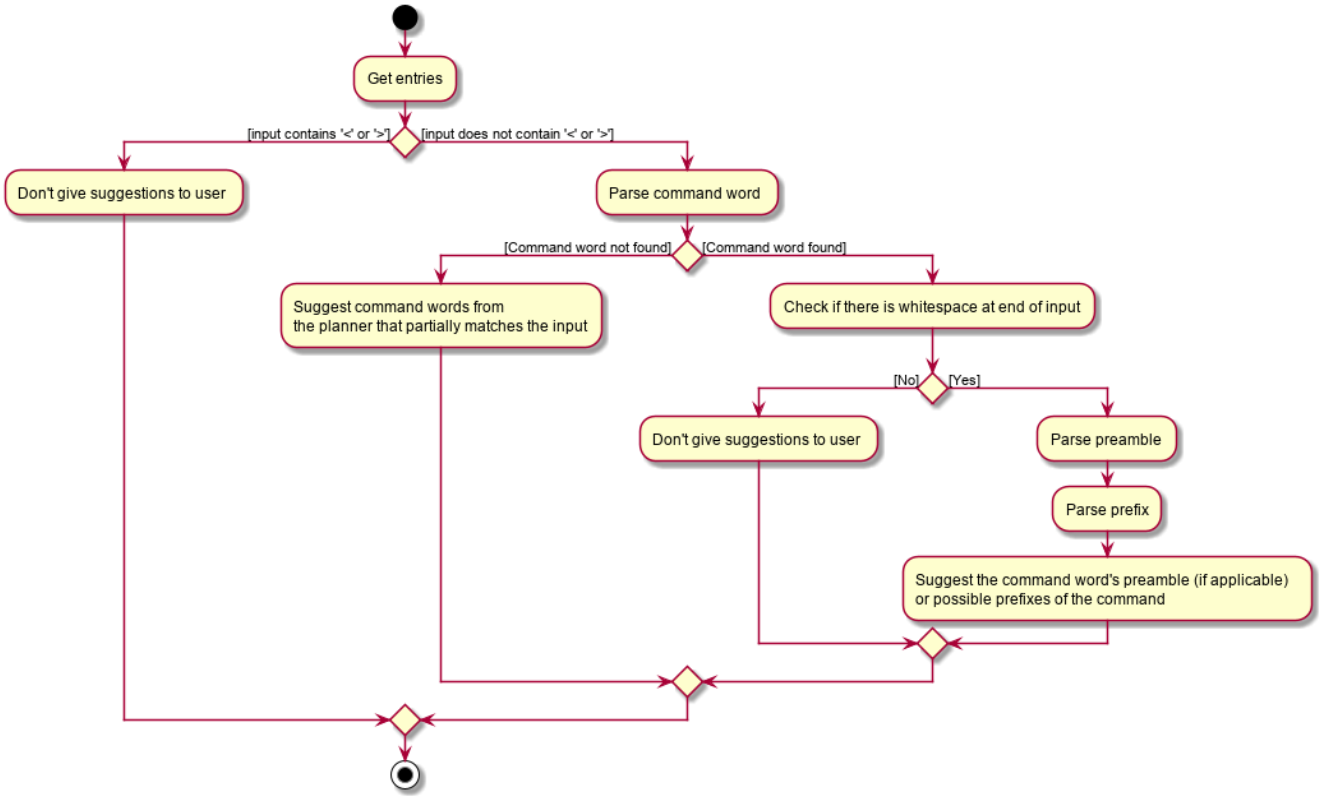


Figure 4. Activity diagram showing how entries for the pop-up box is generated:

Design Considerations

Aspect: Underlying data structure to store the command words

- **Current Choice:** Use an ArrayList
  - Pros: Simple to implement.
  - Cons: Might be slow in providing suggestion for the command word, as in the worse case, it needs to iterate through every command word entry and check with every entry.
- **Alternative:** Use a Trie
  - Pros: Faster in providing suggestion for command word as it only has to iterate through the length of the input.
  - Cons: More complicated to implement. The Trie also needs to be built during startup.
- **Reason for choice**: Given that the program does not have many commands and unlikely to have a significant number more commands in the future, search times are not an issue.

Aspect: Checking if the user has completed typing the command word/preamble/prefix content

- **Current Choice:** Assumes the user has finished the command word/preamble/prefix content when there is a whitespace at the end of input
  - Pros: Simple to implement. Works in most cases.
  - Cons: Might be wrong when suggesting new prefix as there are some commands like autoschedule that have prefixes that can have 2 arguments (autoschedule t/argument1 argument2 d/argument1 argument2...)

- **Alternative:** Waits for a specific duration of time before giving user suggestions
    - Pros: More likely to provide suggestions when the user needs it.
    - Cons: There might be a noticeable delay in the suggestions appearing if the specific duration is too long. More complicated to implement.

Timetable

Current Implementation

Internally, Timetable is a List of ActivityWithTime. This gives the Timetable flexibility to accept Activities that start and end at any time instead of fixed intervals (e.g. 30 minute intervals) while allow for conflicting ActivityWithTime.

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: 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. Does not allow for conflicting startDateTime between ActivityWithTime.
- Alternative 3(current choice): a List of ActivityWithTime
    - Pros: Allows flexible start times and end times. ActivityWithTime can have the same startDateTime, thus allowing conflicting ActivityWithTime.
    - 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. Slower access time if the size of the List is large.

UI component

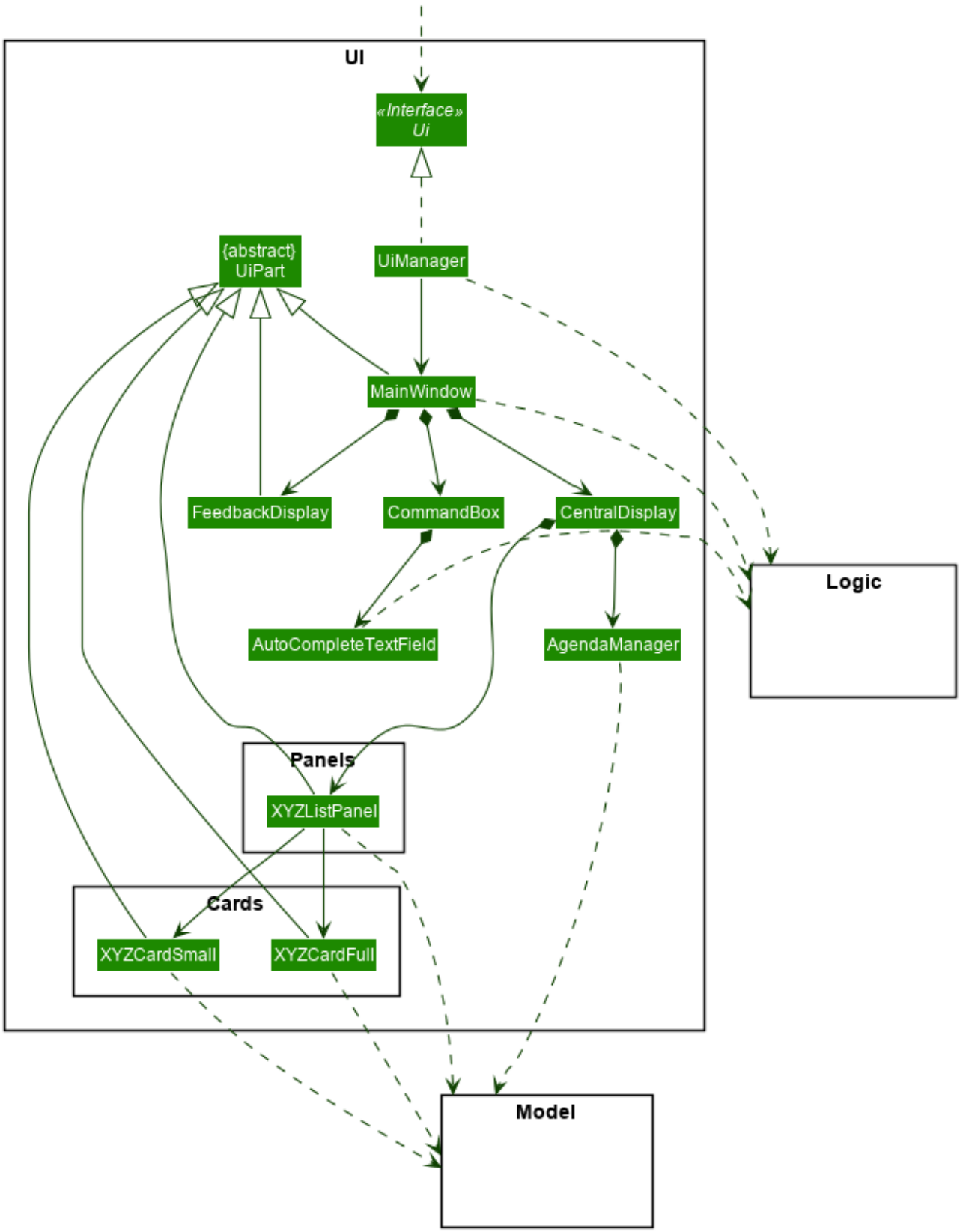


Figure 5. Structure of the UI Component

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.

Model component

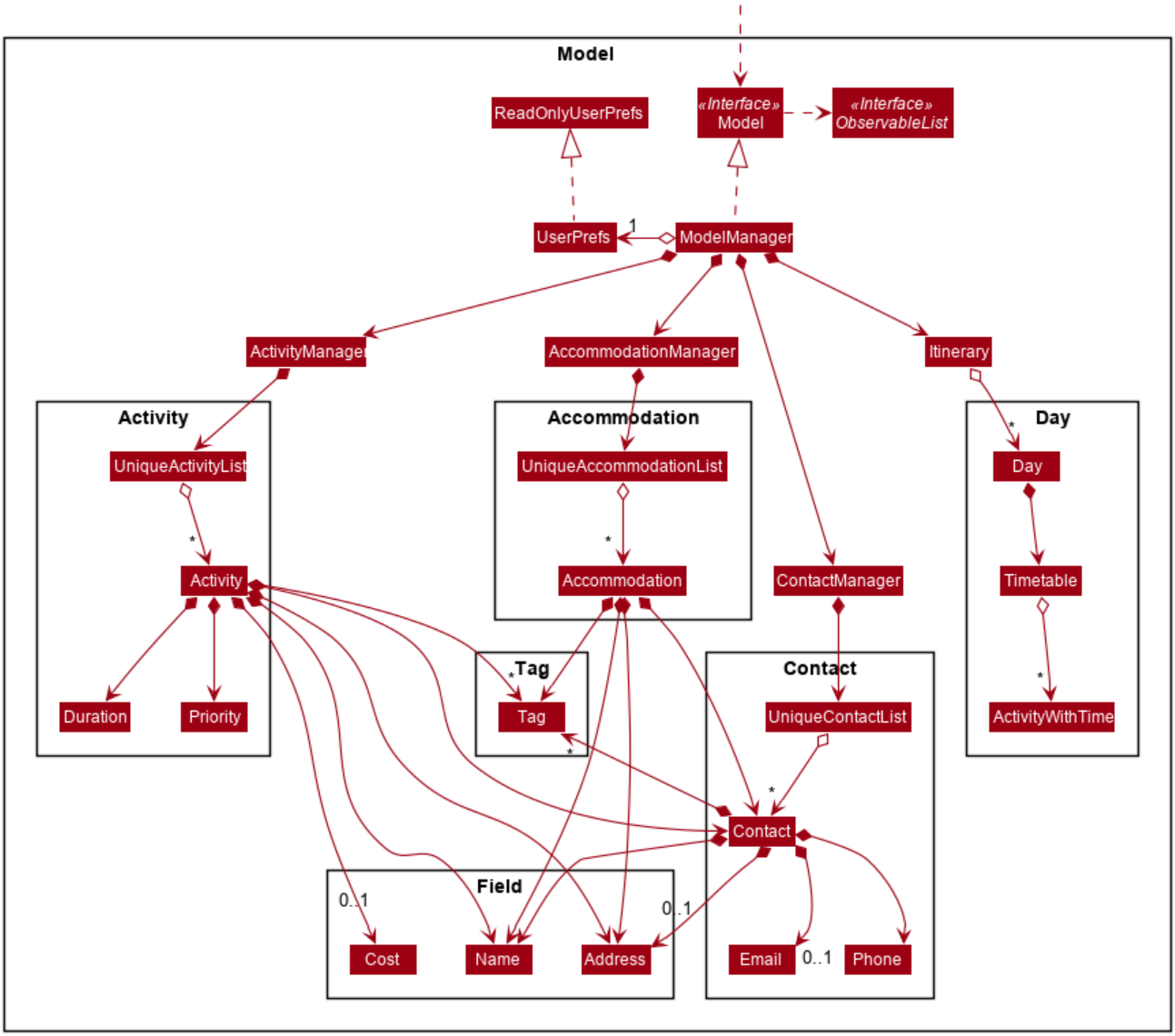


Figure 6. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores a `ActivityManager`, `AccommodationManager`, `ContactManager` and `Itinerary` that represents the

managers for activity, accommodation, contact and day respectively.

- exposes an unmodifiable `ObservableList<Activity>`, `ObservableList<Accommodation>`, `ObservableList<Contact>` and `ObservableList<Day>` 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.