TravelPal - Developer Guide

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

1. Setting up

Refer to the guide here.

2. Design

2.1. Architecture

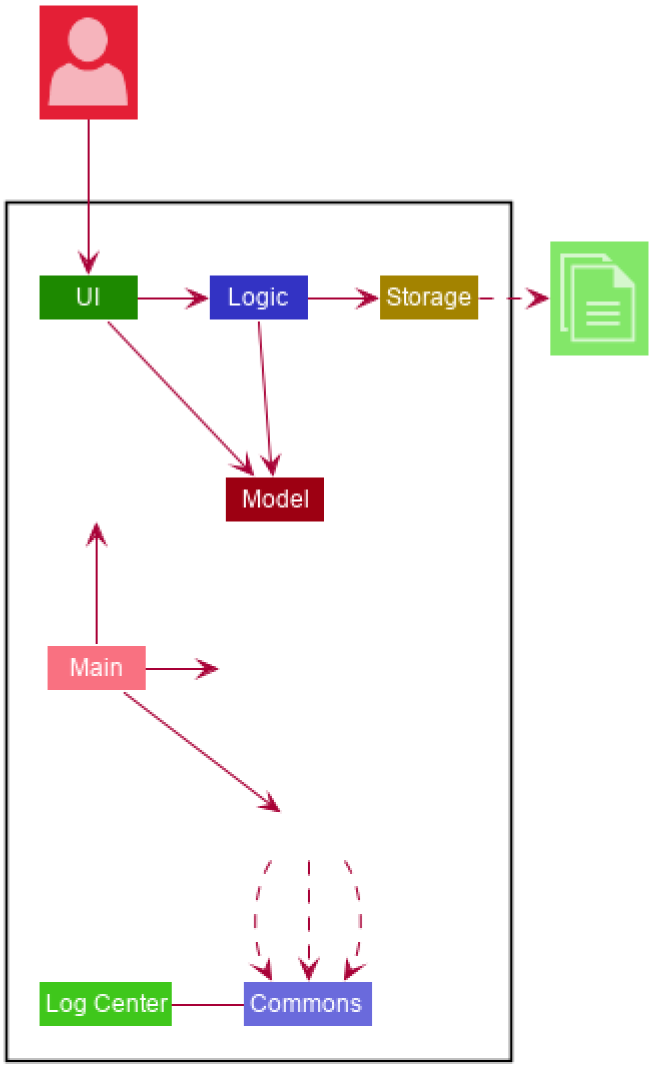


Figure 1. Architecture Diagram

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

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

`Main` has two classes called `Main` and `MainApp`. It is responsible for,

- At app launch: Initializes the components in the correct sequence, and connects them up with each other.
- At shut down: Shuts down the components and invokes cleanup method where necessary.

`Commons` represents a collection of classes used by multiple other components. The following class plays an important role at the architecture level:

- `LogsCenter` : Used by many classes to write log messages to the App's log file.

The rest of the App consists of four components.

- **`UI`**: The UI of the App.
- **`Logic`**: The command executor.
- **`Model`**: Holds the data of the App in-memory.
- **`Storage`**: Reads data from, and writes data to, the hard disk.

Each of the four components

- Defines its *API* in an `interface` with the same name as the Component.
- Exposes its functionality using a `{Component Name}Manager` class.

For example, the `Logic` component (see the class diagram given below) defines it's API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

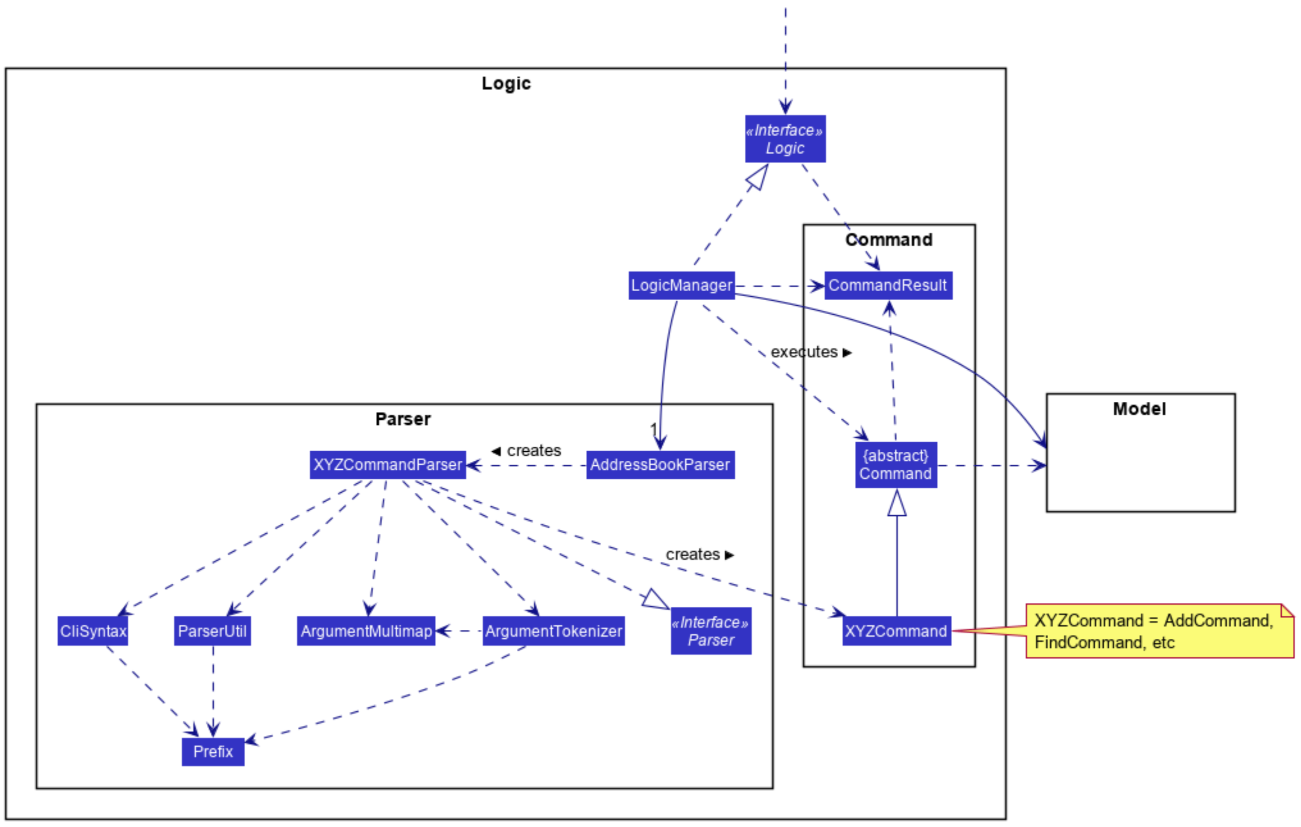


Figure 2. Class Diagram of the Logic Component

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`.

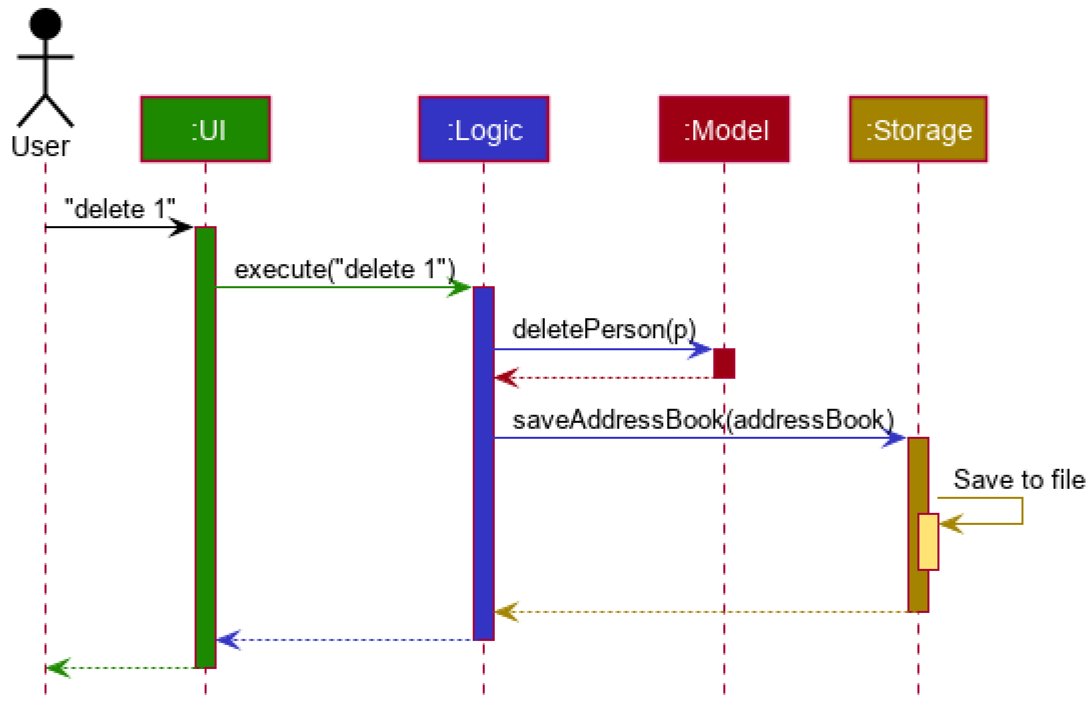


Figure 3. Component interactions for `delete 1` command

The sections below give more details of each component.

2.2. UI component

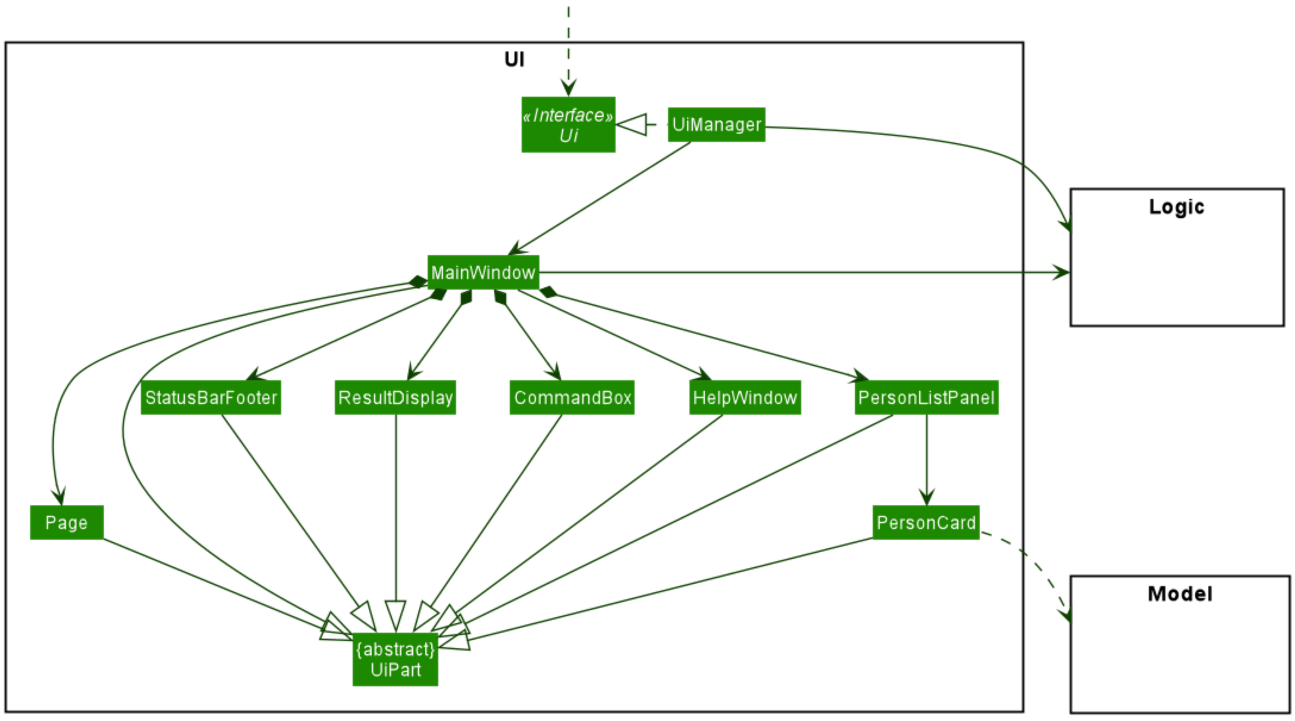


Figure 4. Structure of the UI Component

API : `Ui.java`

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

Overall, 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.

2.2.1. Page API

The `Page` `UiPart`, as contained by `MainWindow`, is the key component that reflects the differences in the data displayed across different pages of the application.

Each class extending `Page` is able to implement its various components and root JavaFX node types separately, allowing for much flexibility in terms of the different user interfaces of the pages.

Due to its flexibility, the `Page` abstract class is mainly only responsible for :

1. Providing its child classes with the supporting instances of `Model` and `MainWindow` in order to,
   - populate the classes' components with data
   - execute commands from the user interface, outside of the `CommandBox` (e.g. an *add* button).
2. Providing a way to execute any callback function (such as one to update display data), through use of the abstract method `fillPage`. The `fillPage` method is registered inside `MainWindow`, such that it runs after each command execution.

2.3. Logic component

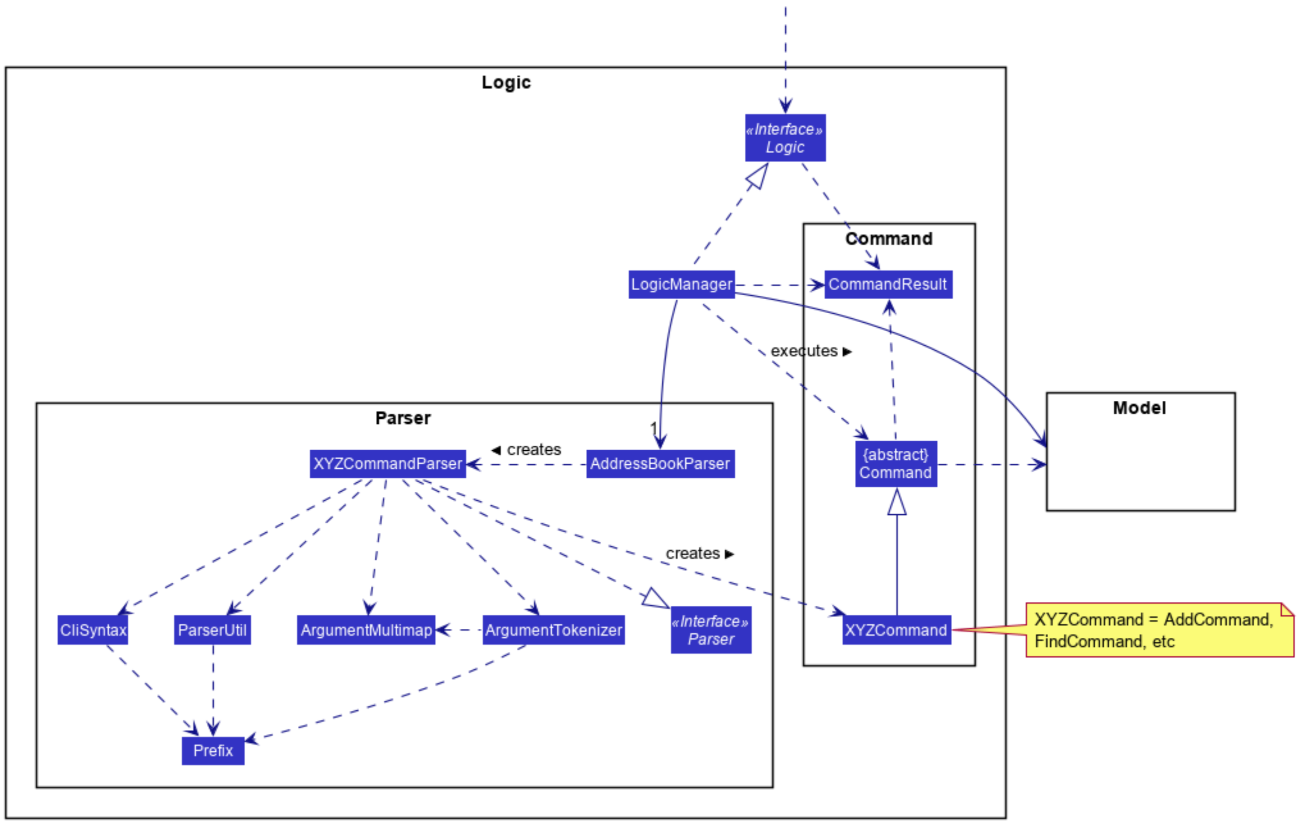


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `AddressBookParser` 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 person).
4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.
5. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.

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

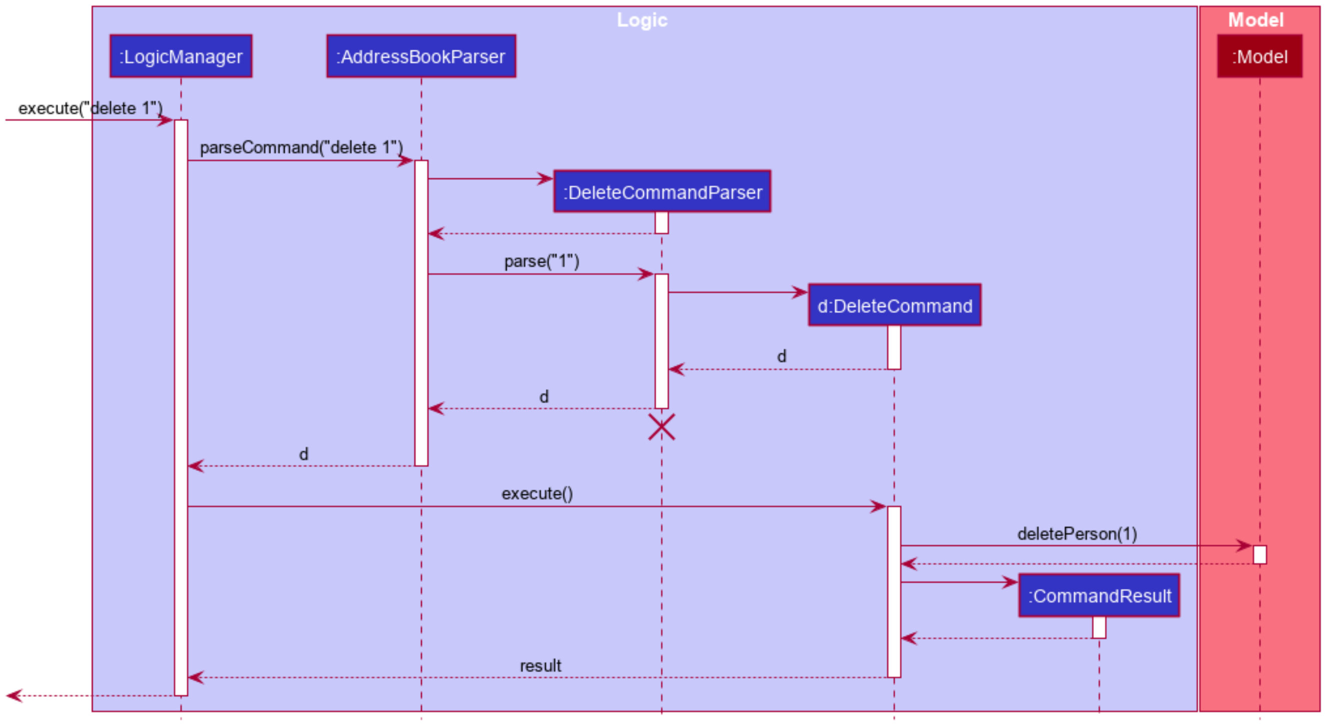


Figure 6. Interactions Inside the Logic Component for the `delete 1` *Command*

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

2.4. Model component

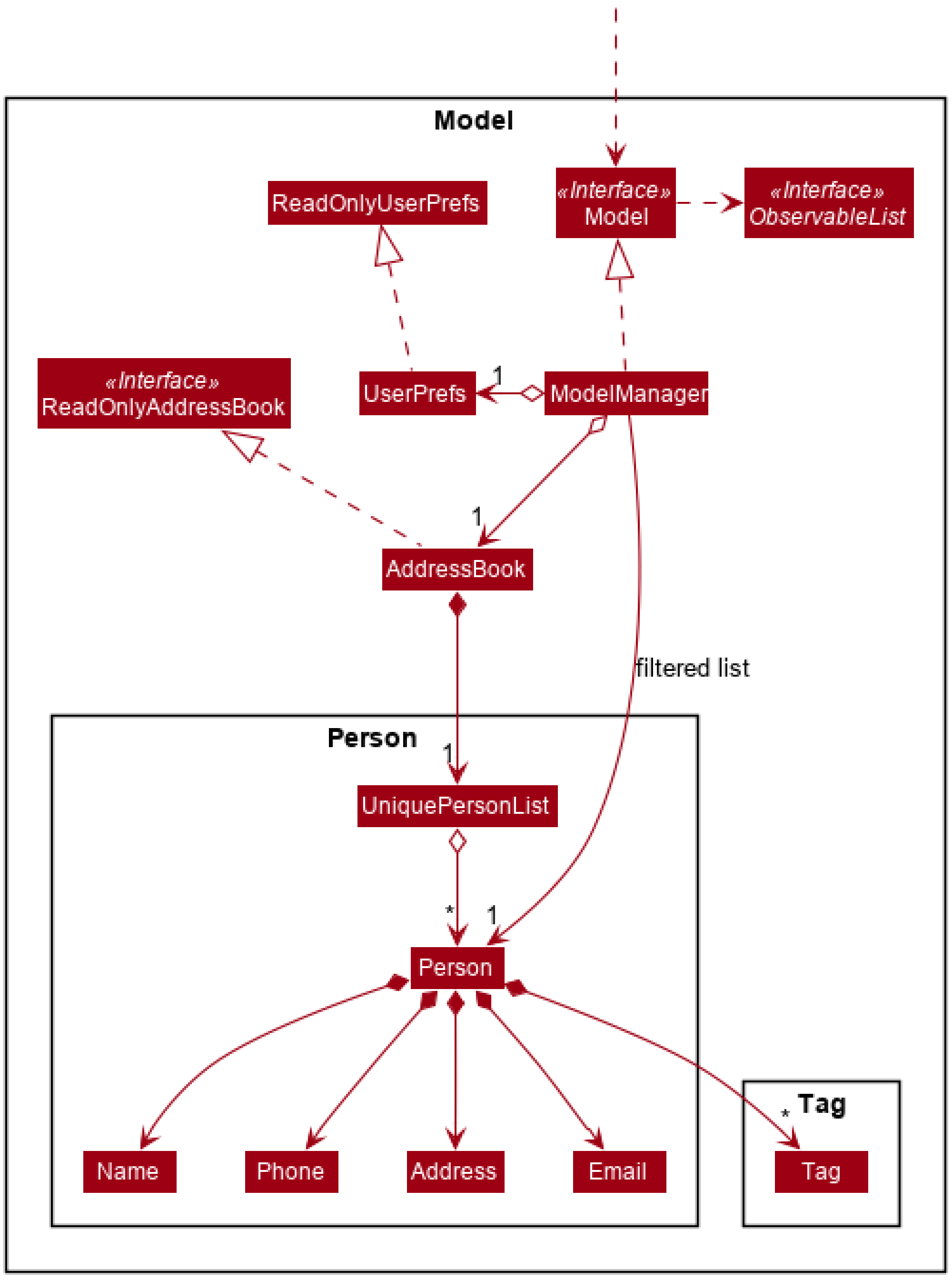


Figure 7. Structure of the Model Component

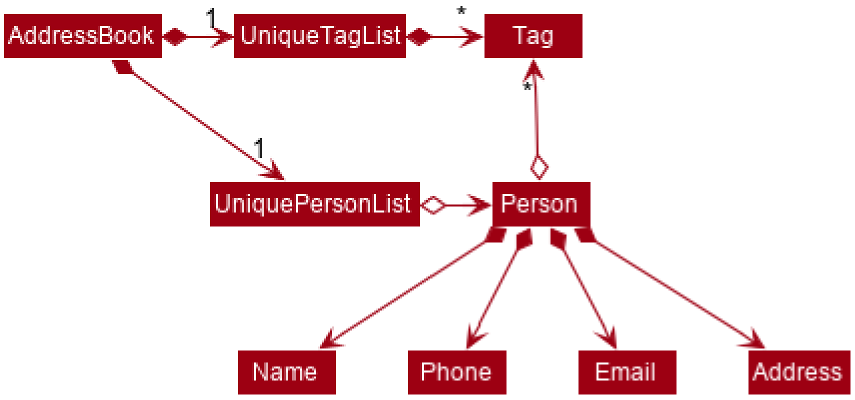
API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user’s preferences.
- stores the Address Book 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

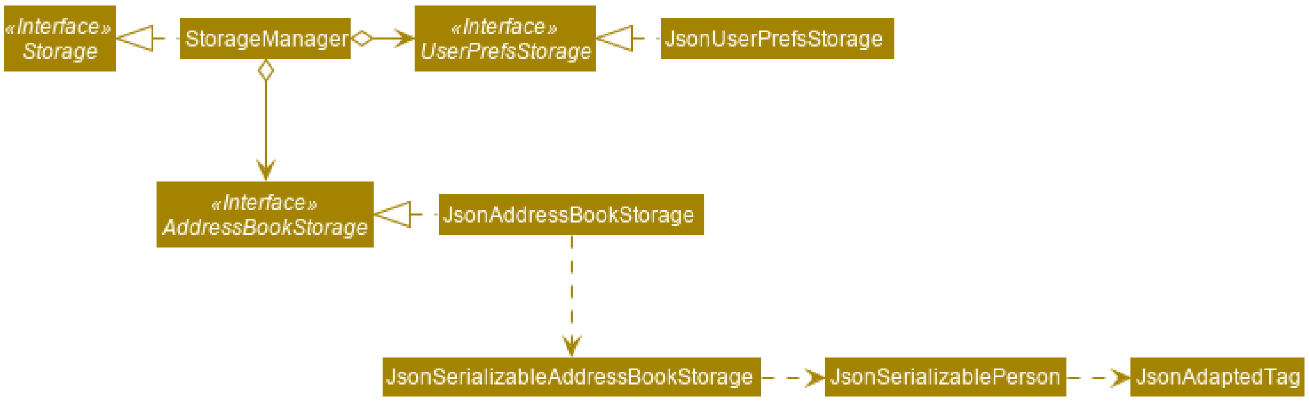


Figure 8. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

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

2.6. Common classes

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

3. Implementation

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

3.1. [Itinerary] Edit trip/day/event feature

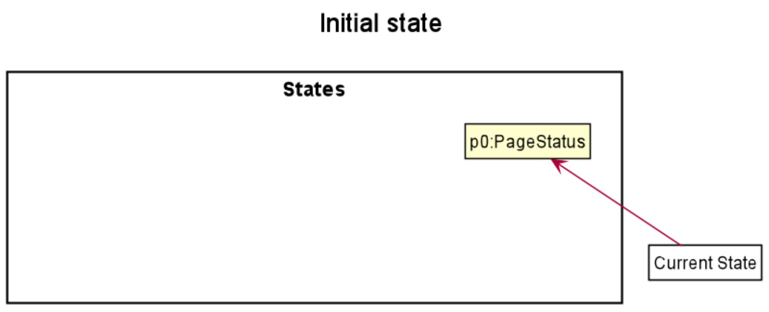
3.1.1. Aspect: Logic

Editing of trip/day/event can be accessed from `TripsPage/DaysPage/EventsPage` respectively. The execution of commands in the each page is facilitated by `TripManagerParser/DayViewParser/EventViewParser` which extends from the `PageParser`. This class serves as the abstraction for all parsers related to each *Page*.

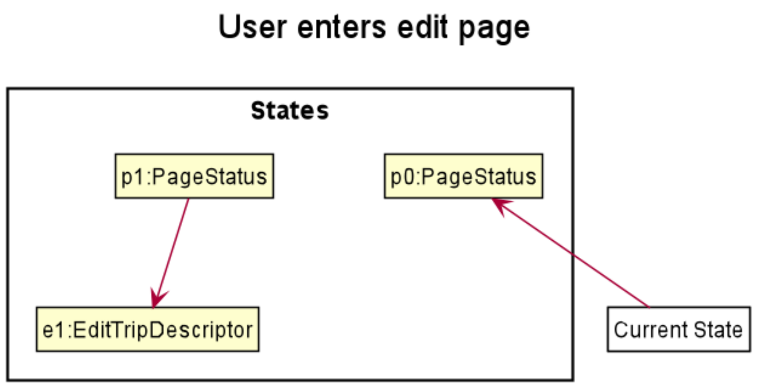
The operations are exposed to the `Model` interface through the `Model#getPageStatus()` method that returns the `PageStatus` containing the all information regarding the current state of application. This includes the *descriptors* (explained in Step 1 below) which stores all information about the edit.

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

Step 1. When the user launches the application. The `PageStatus` is initialized under along with other `Model` components. `PageStatus` at launch does not contain any `EditTripDescriptor/EditDayDescriptor/EditEventDescriptor` responsible for storing information for the edit.



Step 2. The user currently on the `TripsPage/DaysPage/EventsPage` is displayed a list of `Trip/Day/Event` respectively. The user executes the edit command `EDIT1` using the `OneBasedIndex` on the list to edit it.This executes the `EnterEditTripFieldCommand/EnterEditDayFieldCommand/EnterEditEventFieldCommand` that initializes a new descriptor within `PageStatus` before switching over to the `EditTripPage/EditDayPage/EditEventPage` containing to perform the editing.

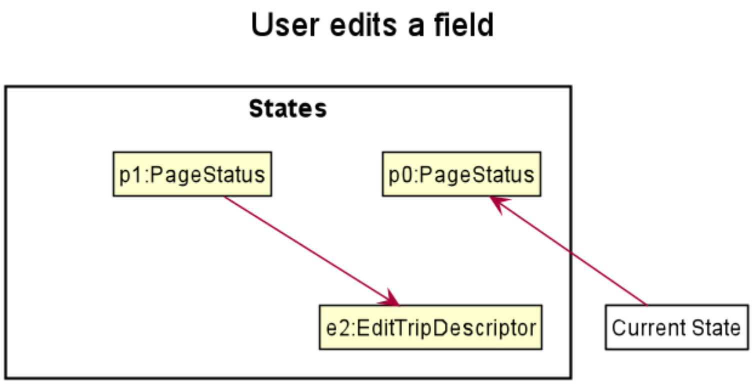


Step 3. The user is now on the edit page displaying a list of fields that the user can edit in the `Trip/Day/Event`. Commands on each page differs based on the fields they contain.

The following is an example list of commands available in `DaysPage` and the execution of the program when a field is edited in `DaysPage`:

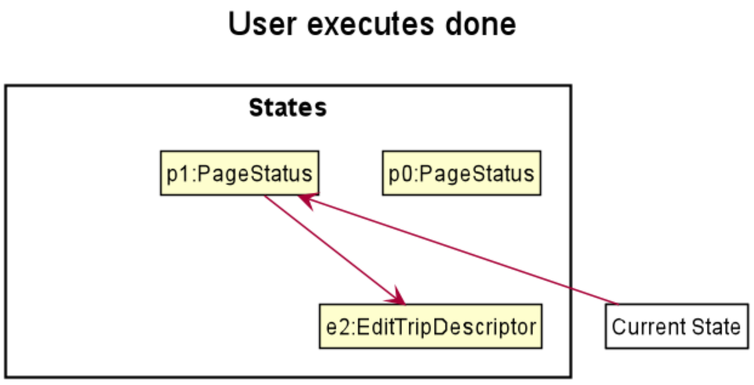
- `edit n/<name> ds/<startDate> de/<endDate> b/<totalBudget> l/<destination> d/<description>` - Edits the relevant fields
- `done` - Completes the edit and returns to the *Overall View*
- `cancel` - Discards the edit and returns to the *Overall View*

When user executes the command `edit n/EditedName` on the `DaysPage`. The command creates a new descriptor from the contents of the original, replacing the fields only if they are edited. The new descriptor is then assigned to `PageStatus` replacing the original `EditDayDescriptor`. The result of the edit is then displayed to the user.

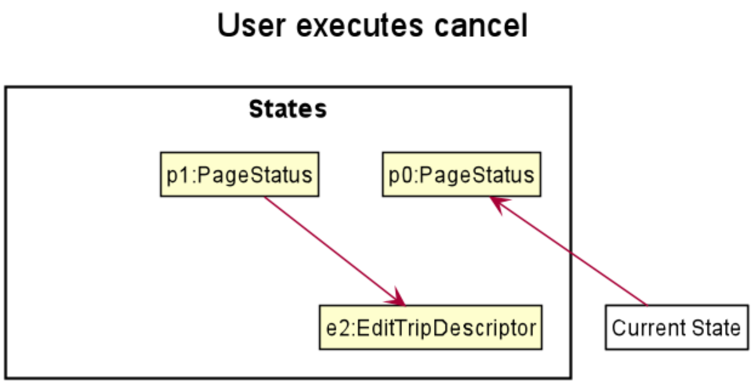


Step 4. The user has completed editing the `Trip/Day/Event` and executes `done`/`cancel` to confirm/discard the edit. The execution of the two cases are as follows:

- The user executes `done` to confirm the edit. This executes the `DoneEditTripCommand/DoneEditDayCommand/DoneEditEventCommand` and a `Trip/Day/Event` is built from the descriptor respective to the type it describes. `DayList#set(target, edited)` proceeds to be executed which accesses the `Day` to edit from the `day` field in `PageStatus` as the target. This method replaces the original day with the newly built day from the descriptor. The descriptor in `PageStatus` is then reset to contain empty fields (See figure below).



- The User executes `cancel` to discard the edit. This executes the `CancelEditTripCommand/CancelEditDayCommand/CancelEditEventCommand` which resets the descriptor in `PageStatus` to contain all empty fields.



- Upon completion of the edit, the user is returned to the `TripPage/DaysPage/EventsPage` depending on where the user entered the edit page from.

Below is a sequence diagram illustrating the execution of the command "edit ds/10/10/2019" on *Days Page*:

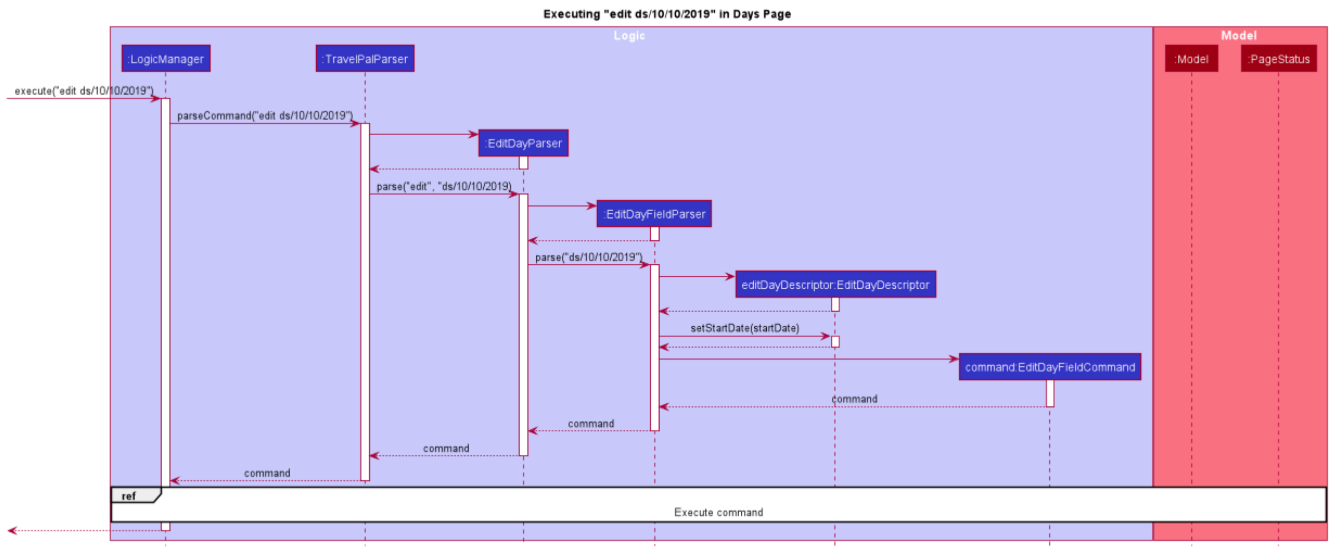

Executing "edit ds/10/10/2019" in Days Page
Logic
Model
:LogicManager
:TravelPalParser
:EditDayParser
:EditDayFieldParser
editDayDescriptor:EditDayDescriptor
command:EditDayFieldCommand
:Model
:PageStatus
execute("edit ds/10/10/2019")
parseCommand("edit ds/10/10/2019")
parse("edit", "ds/10/10/2019)
parse("ds/10/10/2019")
setStartDate(startDate)
command
command
command
command
ref
Execute command

Figure 9. Sequence diagram for execution of `edit ds/10/10/2019`

- When the command is executed, TravelPal uses a series of parsers to parse entire command
    - `TravelPalParser`: Parses the command. In execution above, it is identified that the first word is the command.
    - `EditDayParser`: Parses the type of command. The string "edit" is parsed and correctly identifies `EditDayParser` should be used to continue parsing further tokens
    - `EditDayFieldParser`: Parses the details of the edit. In this execution, the date is parsed by the `DateParserUtil` class and creates a descriptor as mentioned in the section above

After executing the parsers above, the last parser instantiates and recursively returns the command (e.g. `commandEditDayFieldCommand`) up to the `LogicManager`. `LogicManager` then executes the command as the sequence diagram below:

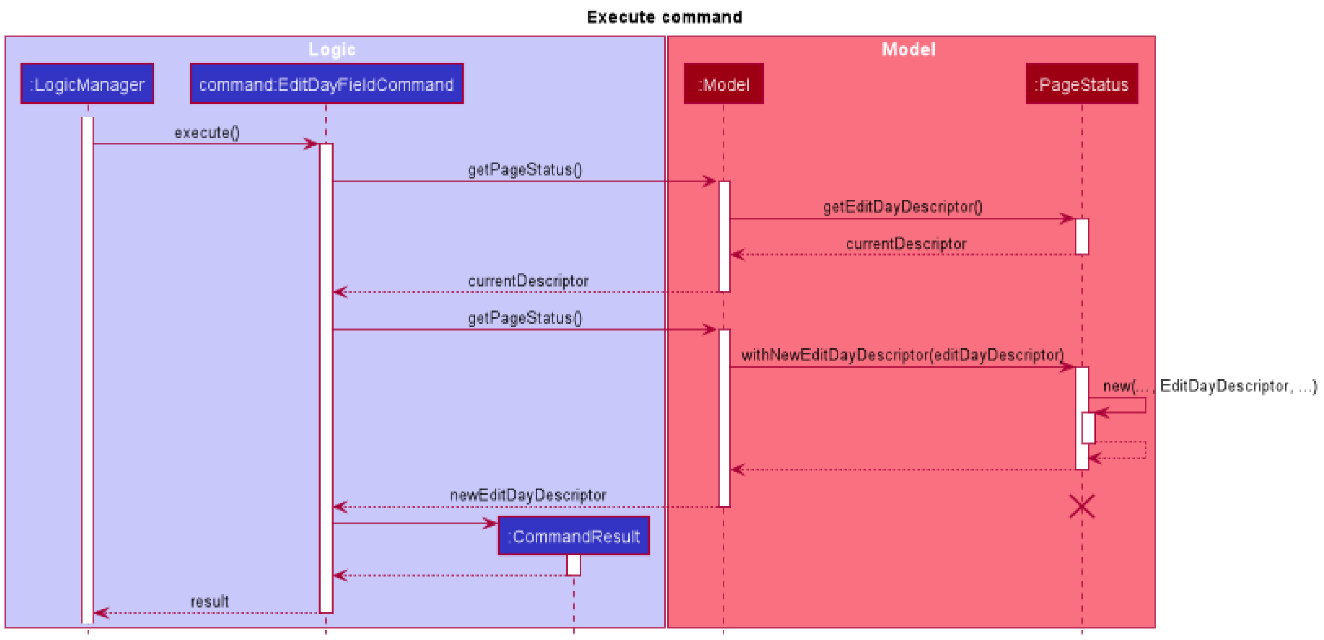


Figure 10. Reference frame for execution of command

The execution of the command is explained above (refer to Section 3.1.1, “Aspect: Logic”).

3.1.2. Aspect: User Interface

The UI for to edit fields are associated with the `EditTripPage/EditDayPage/EditEventPage` respectively.

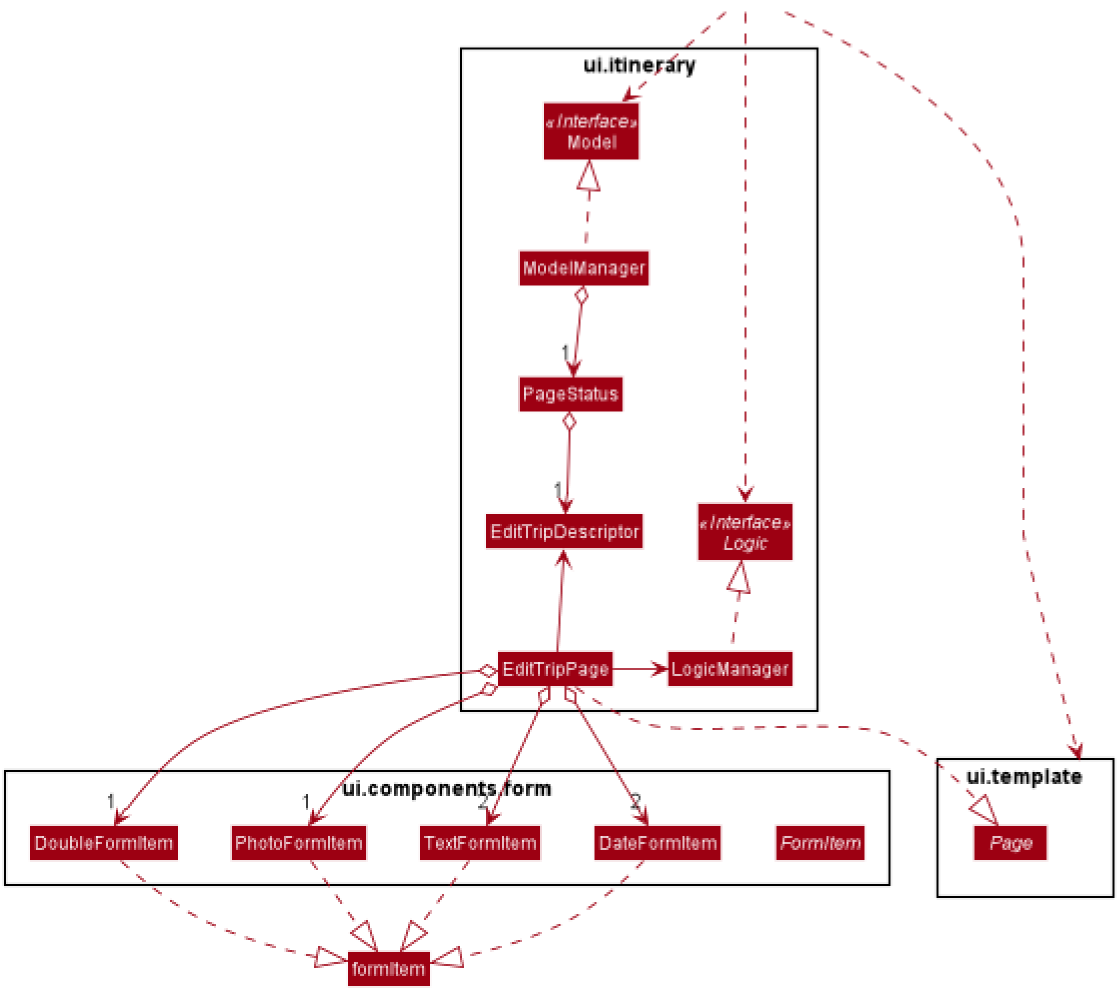


Figure 11. Class diagram showing EditTripPage's associations

The execution of the edit command involves the `Model`, `Logic` and `Ui` components of the application. Listed here are the packages used in the execution of the `edit` commands that are found in the figure above:

- **ui.itinerary**: This package contains all the Ui classes for the *Itinerary feature*.
- **ui.components.form**: Contains all the form items
- **ui.template**: This package contains the `Page` class which all pages extend from

The class at large in the diagram above is the `EditTripPage` of the 3 pages explained. It extends from the `Page` class and is associated with the following:

- Contains `formItems` from the `ui.components.form` generate a form
  - The `FormItems` (e.g. `DateFormItem`) are instantiated by the `EditTripPage#initFormWithModel` method called by the constructor of `EditTripPage` . Each `FormItem` contains an `executeChangeHandler` that executes whenever the `onChange` property is modified by the user. These are initialized as execution of the various edit commands (e.g. EditTripFieldCommand/EditDayFieldCommand/EditEventFieldCommand) using the value in the `FormItem`.
- Navigable to the `ModelManager` and `LogicManager` for execution of commands using Ui interactions.

The contents of the fields are updated by the execution of the commands above. When the user edits any of the `FormItems`, the commands are executed which will cause the `EditTripPage/EditDayPage/EditEventPage#fillPage()` to execute again. `fillPage` retrieves the updated fields from `PageStatus` and displays them as the values in the `FormItems`.

3.1.3. Aspect: Workflow of execution

The logic of editing a field and committing it to memory is a simple process of validating each field. If any field fails to meet the specifications, the `Trip/Day/Event` will not be created/edited. Below is an example execution of validating the edit:

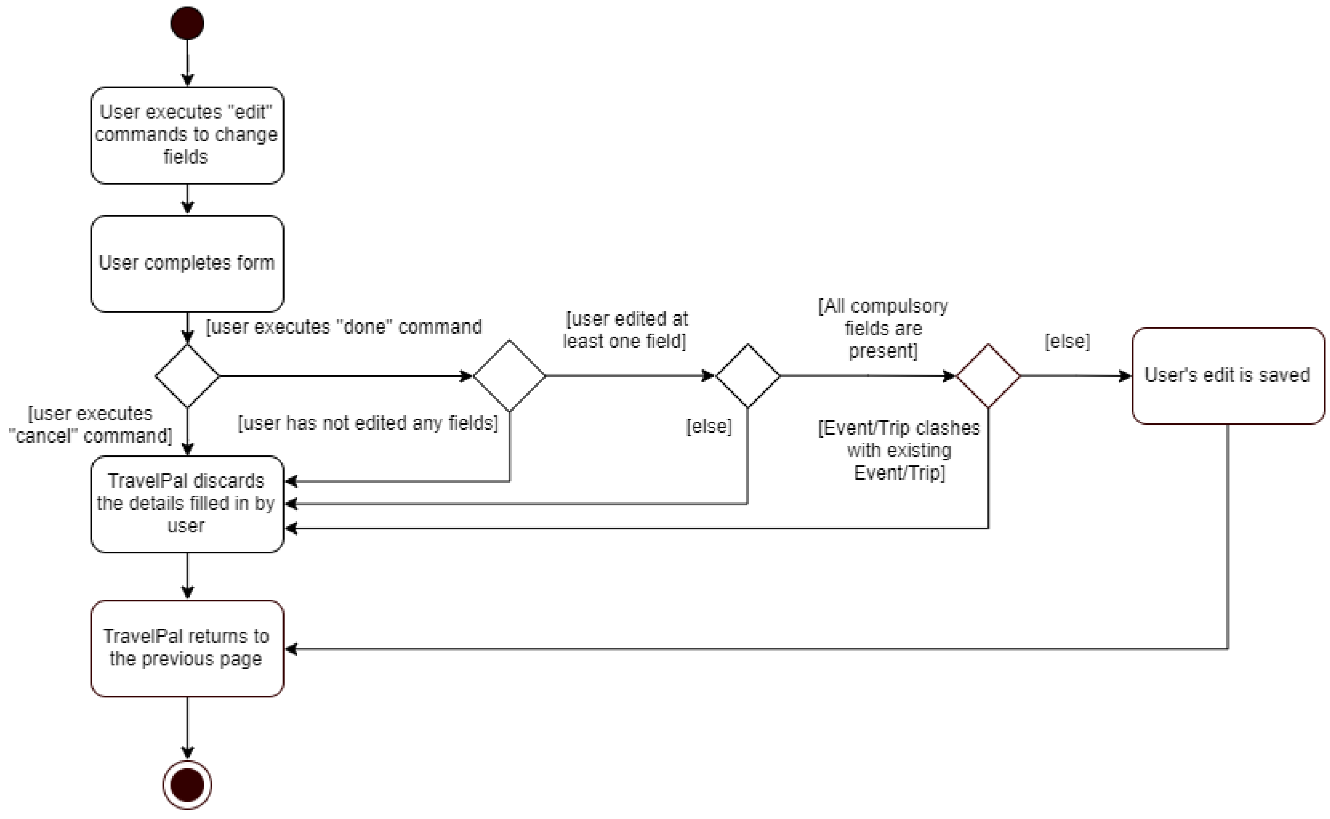

User executes "edit" commands to change fields
User completes form
[user executes "done" command
[user executes "cancel" command]
[user edited at least one field]
[user has not edited any fields]
[All compulsory fields are present]
[else]
[else]
[Event/Trip clashes with existing Event/Trip]
User's edit is saved
TravelPal discards the details filled in by user
TravelPal returns to the previous page

Figure 12. Execution of the done command on any edit page

3.1.4. Aspect: Design considerations

When designing this feature, there were several challenges involved while working with the existing code base especially to adhere to strict Object Orientated Programming Principles. Below are two such design challenges that and how they were resolved:

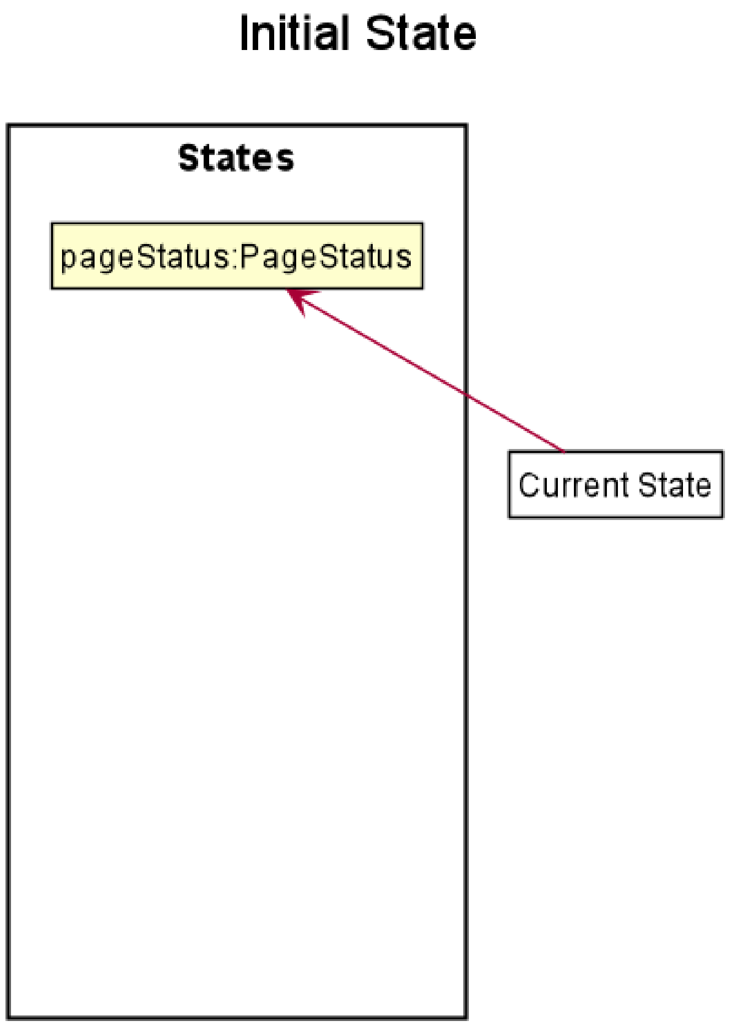
Challenge	Alternative 1	Alternative 2	Chosen Option
Handling Dynamic UI Changes	The first alternative was to consider the updating of ui as a state of the program. The `PageStatus` class includes an `ObservableValue<Command>`, `uiChangeCommand` and design each `Page` to implement `ChangeListener`. When a command is executed, if it involves changing the UI but without switching pages, the `Pages` implementing `ChangeListener` would perform checks on the command executed and execute the correct UI change.	The second alternative was to let pages that can change by execution of commands (dynamic pages) to extend the class `DynamicUiPart` extending from the provided `UiPart` class that contains an abstract method `uiChange` that handle the ui changes. The identification of what ui change should execute is then placed in the `CommandResult` with new fields `CommandWord` and `doChangeUi`	Alternative 2 was chosen due better Object Orientated Programming (OOP) principles. The second method was good practice of the Interface Segregation Principle where classes do not need to depend on methods it did not need. Static pages in the program does not inherit `DynamicUiPart`. However limitations of Java arose as classes cannot inherit more than one class at once. Instead of using the class `DynamicUiPart`, the interface `UiChangeConsumer` was used instead.
Storing of the user's edit information	The first alternative was the straight forward implementation of using the `Logic` interface and its accessors to edit the information in memory directly. This method was however incoherent with out intended design of having forms for users to edit.	The second alternative was using `PageStatus` to store the current state of editing by the user (*edit descriptors*). This method creates a separate place in memory to store the information of the edit. Only after the user confirms/cancels the edit, then the information in the *descriptors* are validated and committed to memory	The second alternative was chose mainly due to the coherence to the design of using forms. The descriptors also serve as minor validations (e.g. Only alphanumeric characters, up to 40 characters etc.). Users can be informed earlier of mistakes in filling forms before submitting.

3.2. [Itinerary] Delete Trip/Day/Event

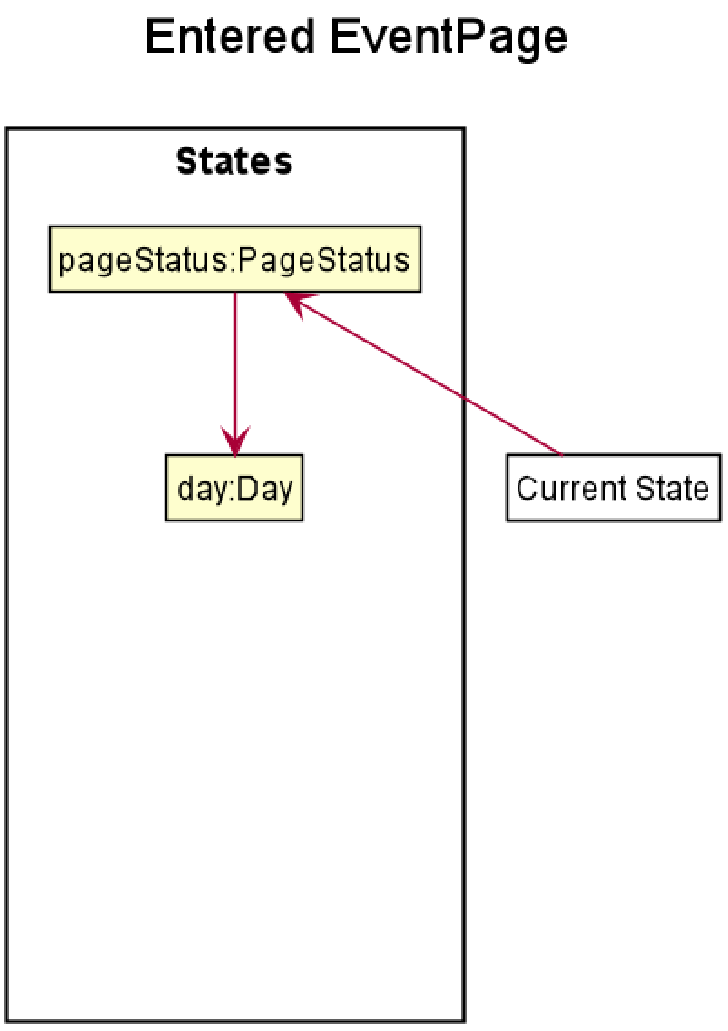
3.2.1. Implementation

Deletion of `Trip/Day/Event` is facilitated by `PageStatus`. `PageStatus` stores the current state of execution of the user program. Upon initial startup of the program `Model` is initialized with `PageStatus` with the `PageType` set to enum `PageType#TRIP_MANAGER`. This indicates the current page displayed to the user. `PageStatus` is initialized with empty references to the `Trip/Day/Event` the user executes an action for.

Step 1. When the user launches the application. `PageStatus` is initialized along with other `Model` components with empty references.



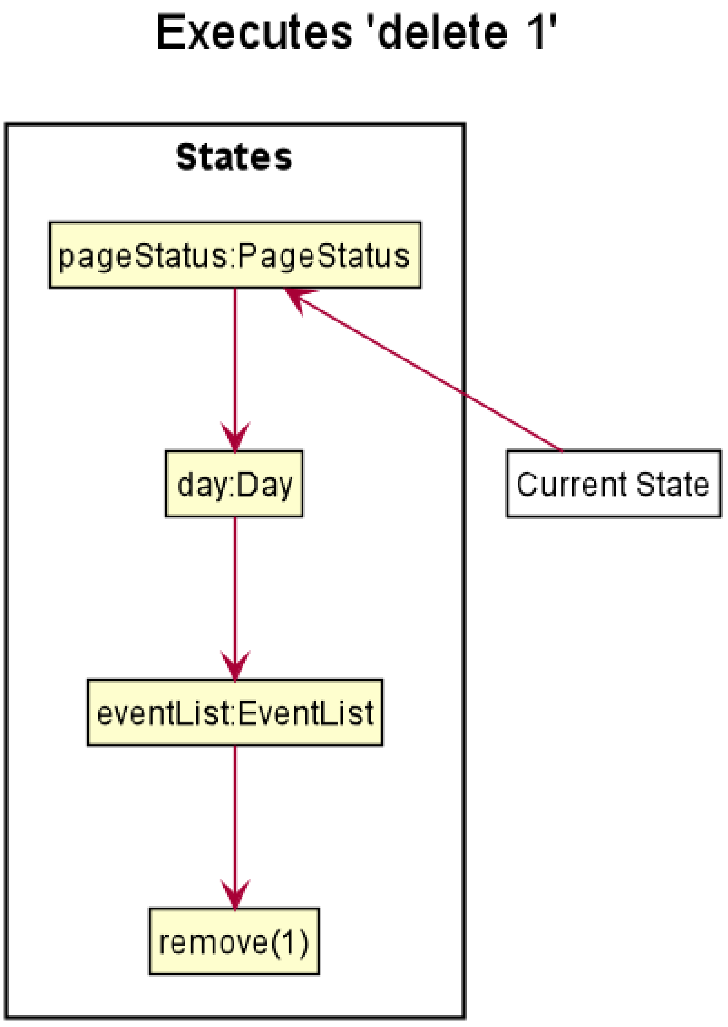
Step 2. The user enters the `DaysPage/EventsPage` using the goto command. This instantiates a new `PageStatus` object from the the existing `PageStatus` with a modified `Day/Trip`, providing the context for subsequent actions. Below is an example execution of the command:



Step 3. The user is now on the `TripManager/DaysPage/EventsPage`, the user can execute the `delete` command in accordance to the display ordered index on any of the aforementioned pages.

When the command `delete <index>` is executed, `DeleteTripCommand/DeleteDayCommand/DeleteEventCommand` is executed. This command accesses `Trip/Day` reference in `PageStatus` assigned by the previous step. (Note: deleting `Trips` do not require `PageStatus`, it being directly accessible to `Model` using `TripList` accessors).

The `Day/Trip` reference contains the list of `Events/Days` in memory respectively (`DayList/EventList`). `DayList#remove/EventList#remove` are methods in the respective list classes used to delete the day/event. These are executed, modifying the in memory `TravelPal` and `Trip/Event/Day` is removed.



3.3. [Expense] Expense Manager

The *Expense Manager* is one of the main features of TravelPal, it maintains a list of `Expense` stored in an `ExpenseList`. *Expense Manager* is also capable of calculating and displaying budget, sorting expenses and toggling display options.

3.3.1. Aspect : Model

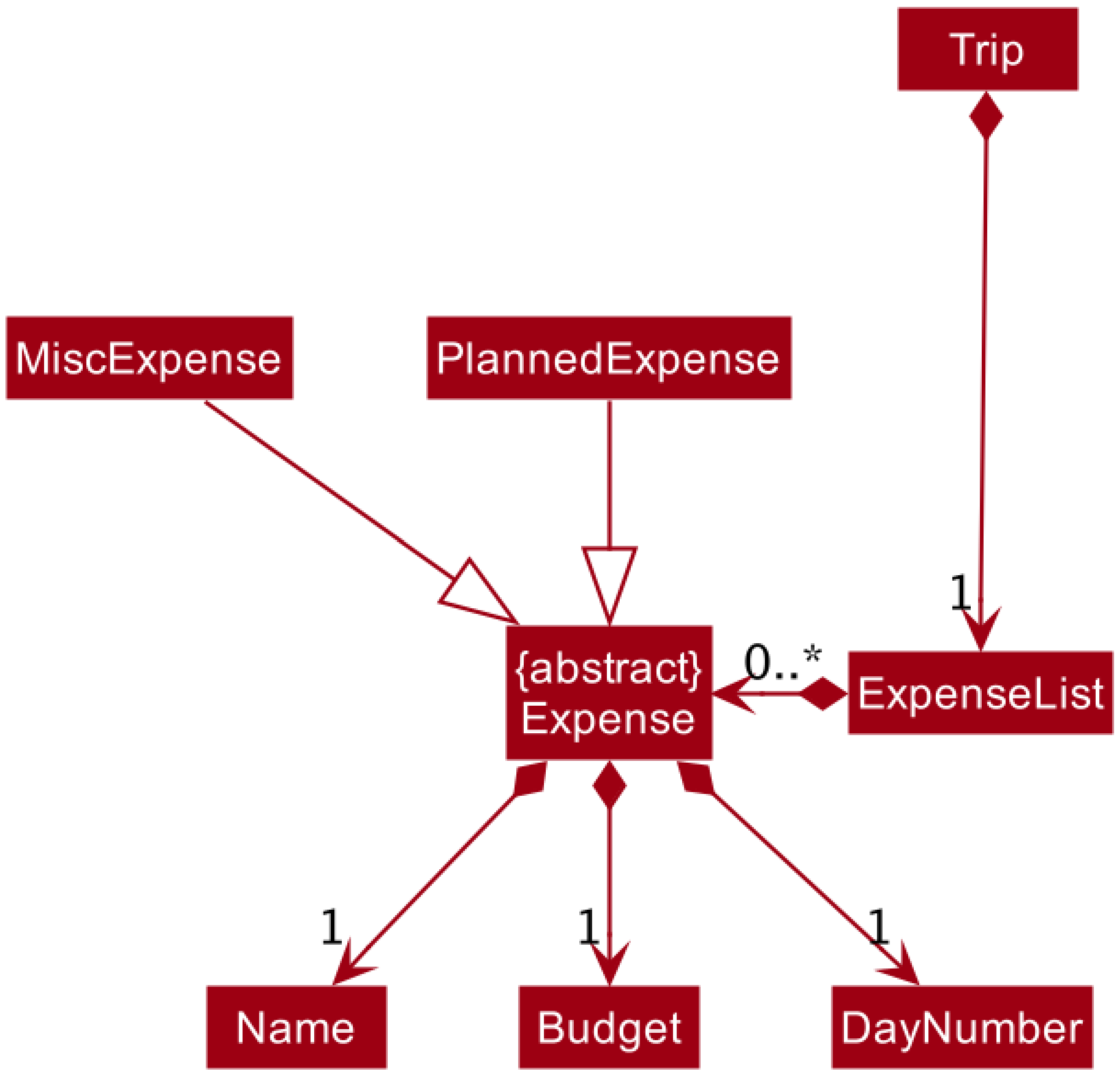


Figure 13. Class diagram showing the expense model

Expense

`Expense` is an abstract class storing expense model.

It has three compulsory fields, the *name of expense*, the *amount of expense*, and a *day number*. These fields are used to store information related to an expense. They are implemented as instance of class `Name` `Budget` and `DayNumber` respectively.

`MiscExpense` and `PlannedExpense` are the child classes extending from `Expense` class, they are used to represent the two types of expenses: *miscellaneous expense* and *planned expense*.

ExpenseList

`ExpenseList` is an class that stores the `Expense` models. It supports wrapper methods around the underlying `ObservableList` to facilitate the use in the logic components.

3.3.2. Aspect : UI

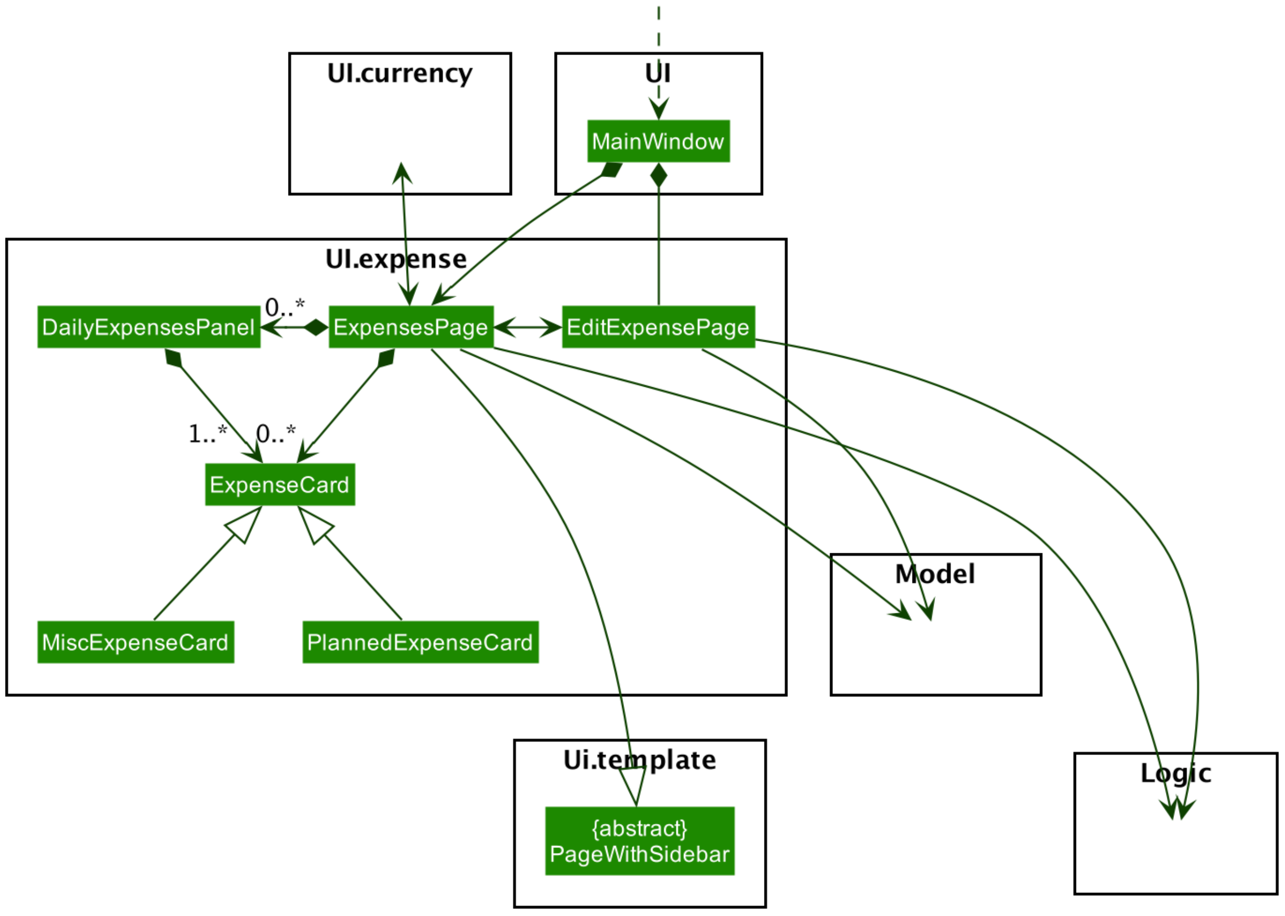


Figure 14. Class diagram showing the user interface of expense

NOTE The `ExpensesPage` implements an `UiChangeConsumer` interface to facilitate the toggling between the *Days View* and *List View* of *Expense Manager*, which is not shown in the diagram above.

The UI of expense manager mainly consists of two `Page`: `ExpensesPage` and `EditExpensePage`.

`ExpensePage` is the component in charge of displaying expense and budget information. It has a list of `ExpenseCard`, a component that contains individual expense details. `ExpensePage` extends `PageWithSidebar` as it contains navigation bar that helps user to navigate between different features.

In `ExpensePage`, user can toggle between *Days View* and *List View*. In *Days View*, a list of `DailyExpensesPanel` is used to group `ExpenseCard` according to date.

`EditExpensePage` is the main page for creating and editing of expense. Both `ExpensesPage` and `EditExpensePage` have access to `Model` and `Logic` of the application, for handling of stored data and parsing commands.

From `ExpensePage`, user can navigate to *Currency* feature of application through CLI or GUI.

3.3.3. Aspect : Logic

Create an expense

The creation of a new `Expense` is done in two ways:

1. The creation of a `PlannedExpense` is created when a new `Event` with a `Budget` is created. The execution happens in `DoneEditingEventCommand`.
   - When the `Name` or `Budget` field of `Event` is modified, a method call replaces the current `Expense` associated with the `Event` with an updated `Expense`.
2. The creation of a `MiscExpense` is done by calling `EnterCreateExpenseCommand`, which brings user to an *Expense Setup Page*.

Edit an expense

Editing of expense can be accessed from `ExpensePage`. The execution of command is handled by `ExpenseManagerParser` and the command accesses the model through `Model#getPageStatus()` method. The details of execution is similar to *Edit trip/day/event* feature (see Section 3.1, “[Itinerary] Edit trip/day/event feature”)

Only the *amount of expense* field of a `PlannedExpense` can be edited with an `edit` command. When `done` command is executed after the *amount of expense* is edited, both the expense in `ExpenseList` and the `Event` will be updated.

The following sequence diagram shows the sequence of method call when `DoneEditCommand#execute(model)` is called in `LogicManager`.

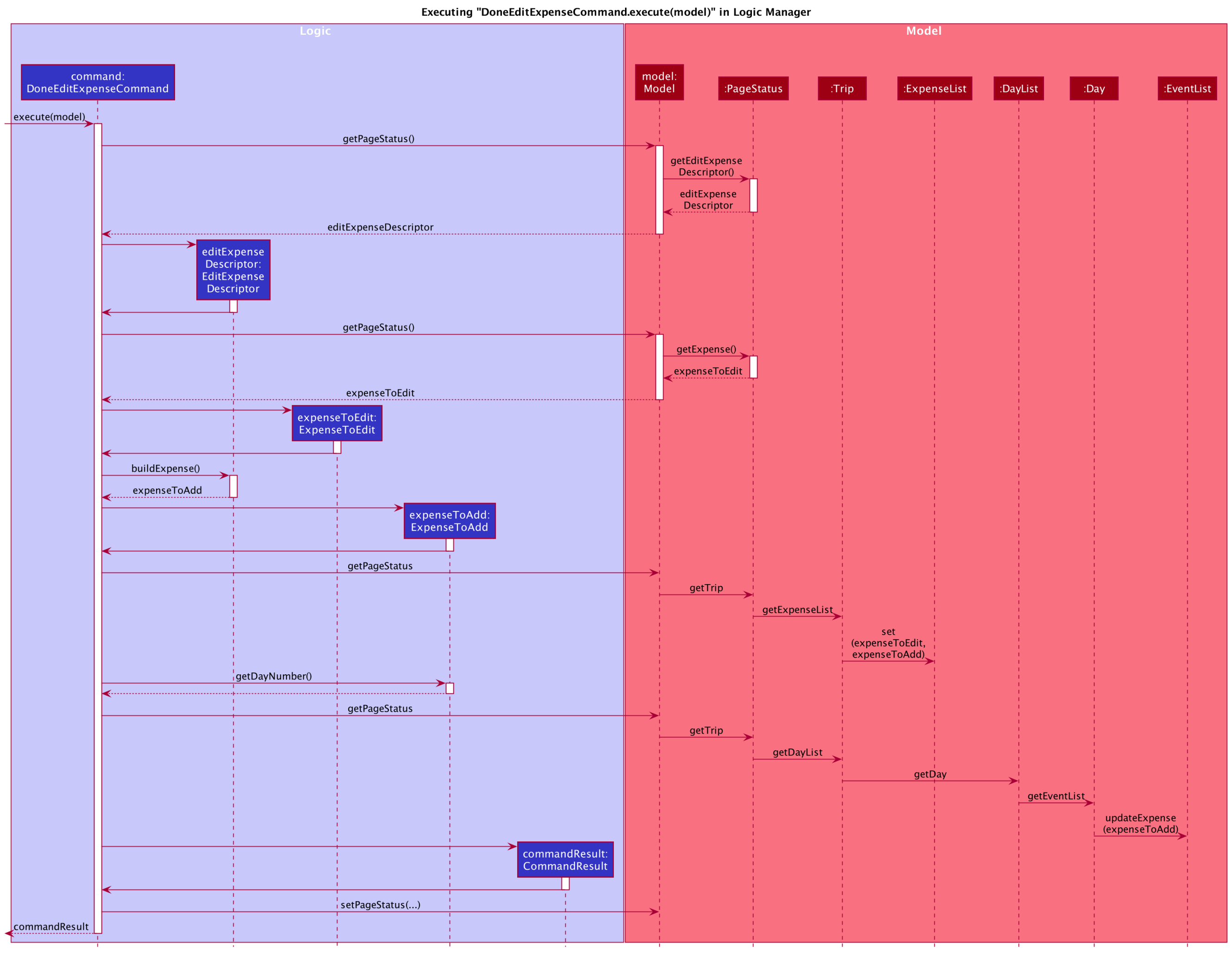


Figure 15. Sequence diagram showing the execution of DoneEditCommand

When `DoneEditCommand#execute(model)` is called, `getPageStatus()` is used to fetch information / update information from `model`. The following steps shows the sequence of event happened within the method:

1. The current instance of `EditExpenseDescriptor` and `Expense` in `model` are returned and stored as `editExpenseDescriptor` and `expenseToEdit` in `logic`.
2. A new instance of `Expense`, `expenseToAdd` is created by calling `buildExpense()` in `editExpenseDescriptor`.
3. Through `getPageStatus()`, `set(expenseToEdit, expenseToAdd)` is called on `EventList`, which updates the unedited `expenseToEdit` by replacing it with the edited `expenseToAdd`.
4. The `DayNumber` in `expenseToAdd` is returned so that logic can update the associated `Event` by going to the corresponding `Day` in `DayList`.
5. `updateExpense(expenseToAdd)` is called on `EventList` so that the target `Event` will have its `Expenditure` updated.
6. By calling `setPageStatus()`, the current `EditExpenseDescriptor` and `Expense` will be reset, the current page will be set to *Expense Manager Page*.
7. `CommandResult` is returned to give user feedback and update UI.

Delete an Expense

Deletion of expense is similar to the *Delete Trip/Day/Event* feature (see Section 3.2, "[Itinerary] Delete Trip/Day/Event").

The `DeleteExpenseCommand` in logic checks for index of deletion and type of expense. Only `MiscExpense` can be deleted through this command. Below is an example execution of deleting an expense:

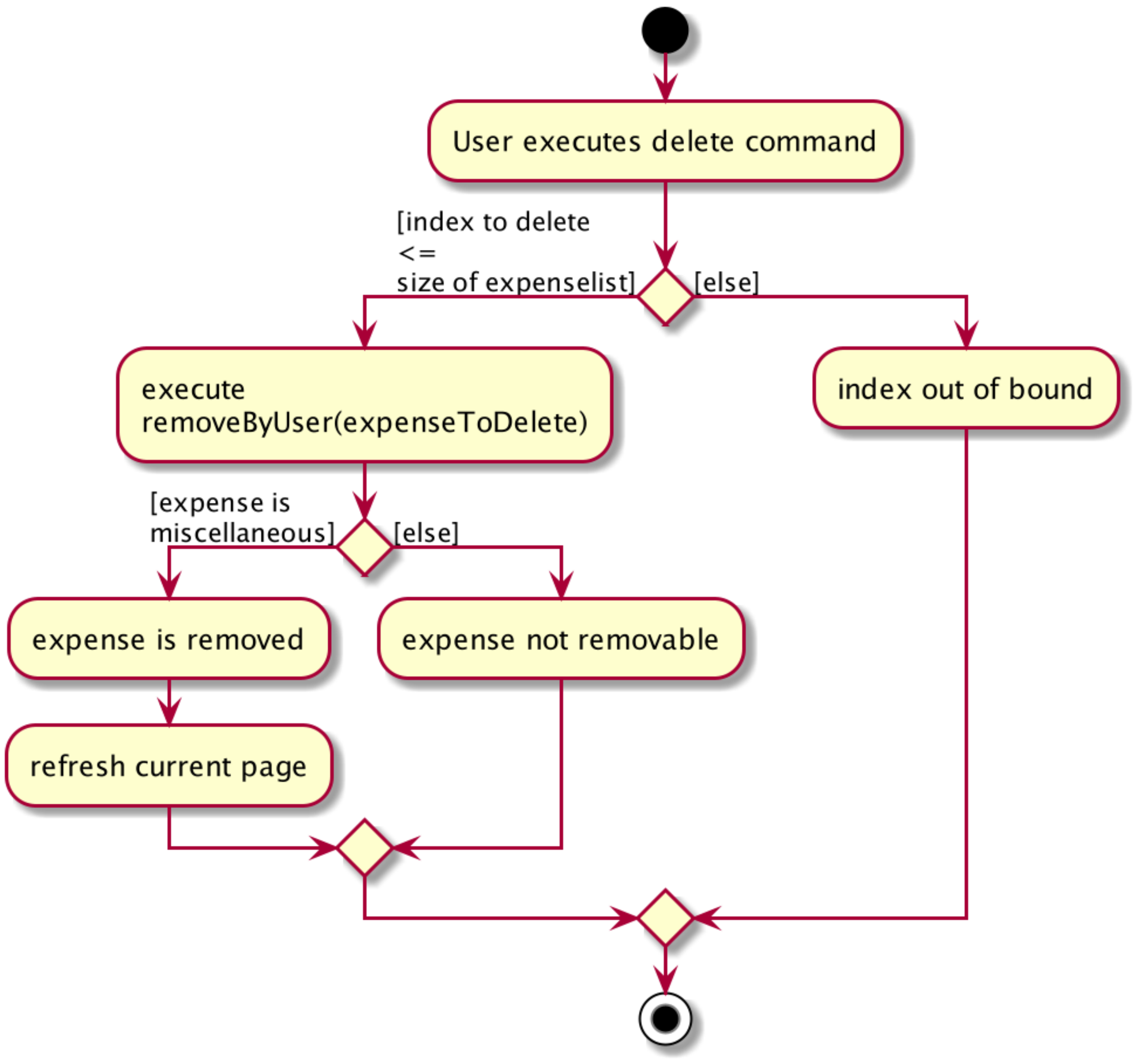

User executes delete command
[index to delete <= size of expenselist]
[else]
execute removeByUser(expenseToDelete)
index out of bound
[expense is miscellaneous]
[else]
expense is removed
expense not removable
refresh current page

Figure 16. Activity diagram showing the execution of deleting an expense

3.4. [Currency] Currency

The *Currency* maintains a list of `CustomisedCurrency` stored in an `CurrencyList`. *Currency* supports creation, deletion and selection of currency.

3.4.1. Aspect : Model

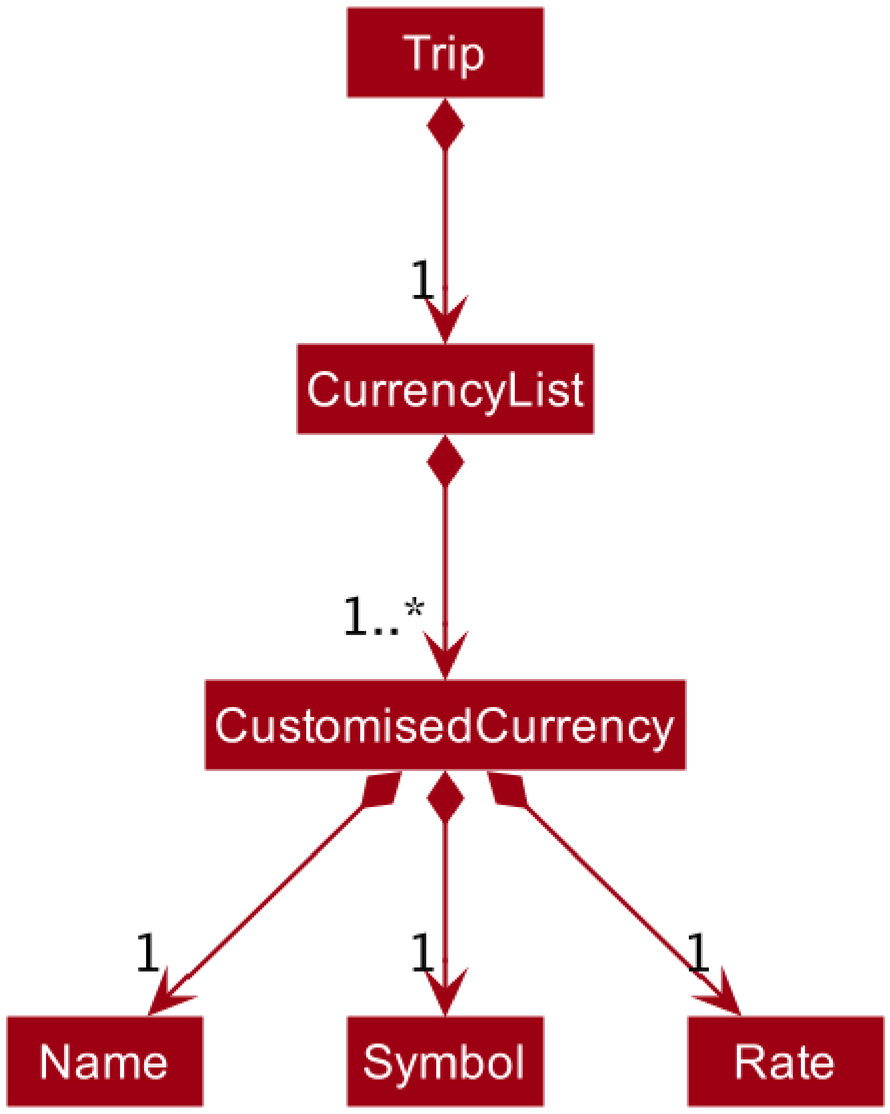


Figure 17. Class diagram showing the currency model

CustomisedCurrency

`CustomisedCurrency` is the generic abstraction of a user defined currency.

A `CustomisedCurrency` contains exactly one instance `Name` `Symbol` and `Rate` for each of these three classes, which represents the *name of currency*, *currency symbol*, and *exchange rate of the currency (to Singapore Dollars)* respectively.

CurrencyList

`CurrencyList` is an class that stores the `CustomisedCurrency` instances. It supports wrapper methods around the underlying `ObservableList` for the use of logic components.

3.4.2. Aspect : UI

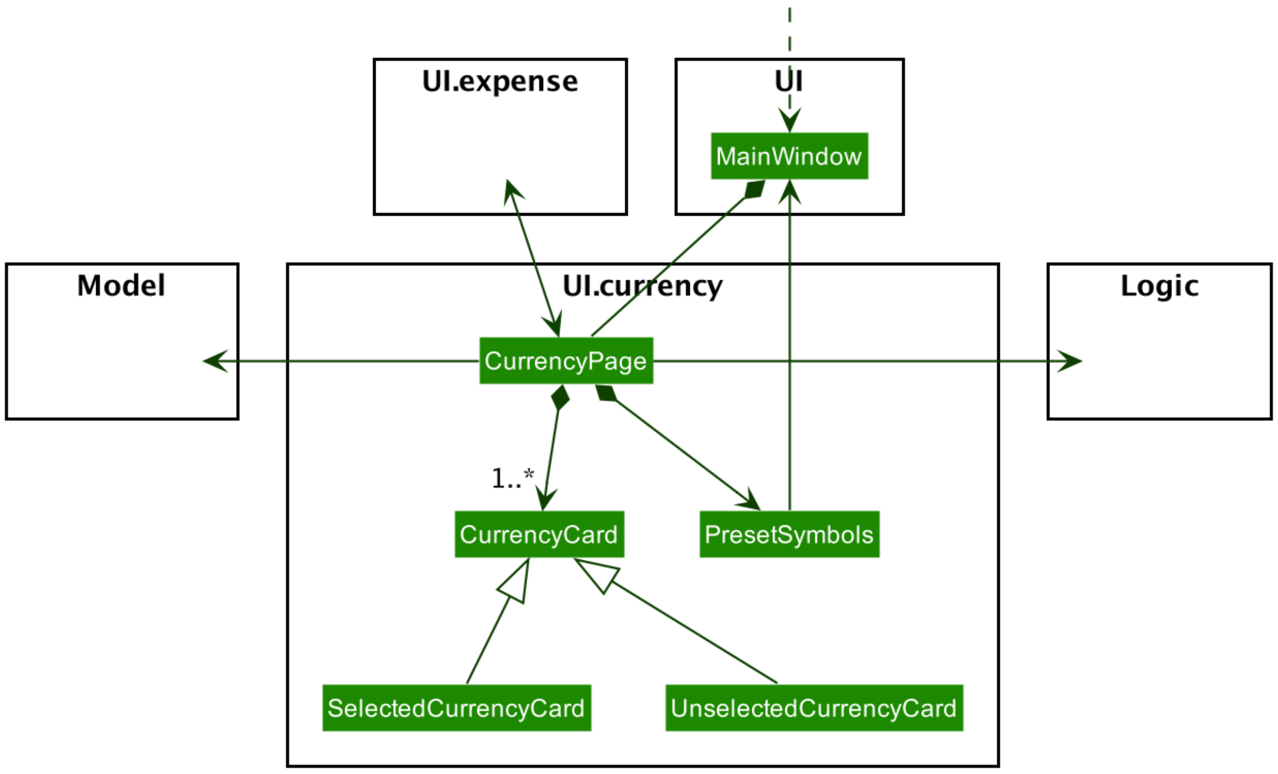


Figure 18. Class diagram showing the user interface of currency

The main `Page` for displaying and creating currency is `CurrencyPage`. The page contains a list of `CurrencyCard` for displaying individual currency details. Two types of `CurrencyCard`: `SelectedCurrencyCard` and `UnelectedCurrencyCard` are used to indicate the currency in use. Upon launching of the application, a default `CustomisedCurrency` — Singapore Dollars is automatically added to the `CurrencyList`, which is not deletable. Thus, there will always be at least one `CurrencyCard` in `CurrencyPage`.

`CurrencyPage` also contains text fields for input of new currency information. It has a `PresetSymbols` instance which contains a group of `ToggleButton` which updates the symbol in `MainWindow`. `CurrencyPage` have access to `Model` and `Logic` of the application, for handling of stored data and parsing commands.

From `ExpensePage`, user can navigate to *Expense Manager* feature of application through CLI or GUI.

3.4.3. Aspect : Logic

The executions of add / delete of `CustomisedCurrency` is similar to those of `Trip` / `Day` / `Event` / `Expense`. More details can be found in the previous sections: Section 3.1, “[Itinerary] Edit trip/day/event feature”

Select a Currency

Selection of `CustomisedCurrency` is achieved by promoting it to the front of `CurrencyList`, while the first `CustomisedCurrency` in the list is to be used to display `Budget` with the selected currency symbol and rate conversion.

All the monetary values in the application is displayed through calling the method from UI:

`Budget#getValueStringInCurrency(CustomisedCurrency)`

3.5. [Proposed] Undo/Redo feature

3.5.1. Proposed Implementation

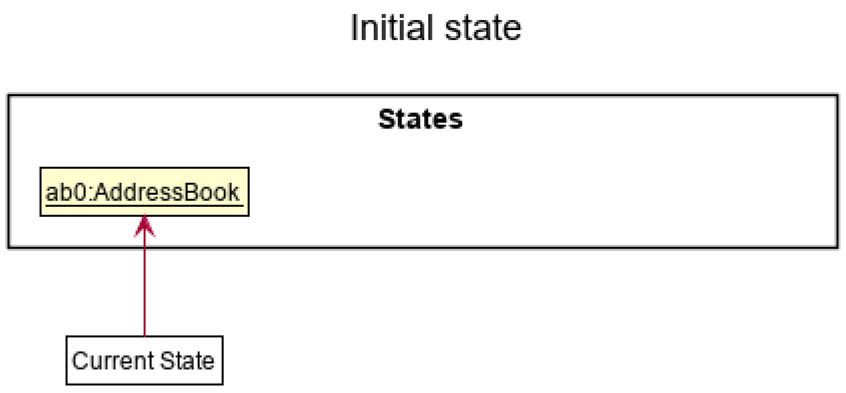
The undo/redo mechanism is facilitated by `VersionedAddressBook`. It extends `AddressBook` with an undo/redo history, stored internally as an `addressBookStateList` and `currentStatePointer`. Additionally, it implements the following operations:

- `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.

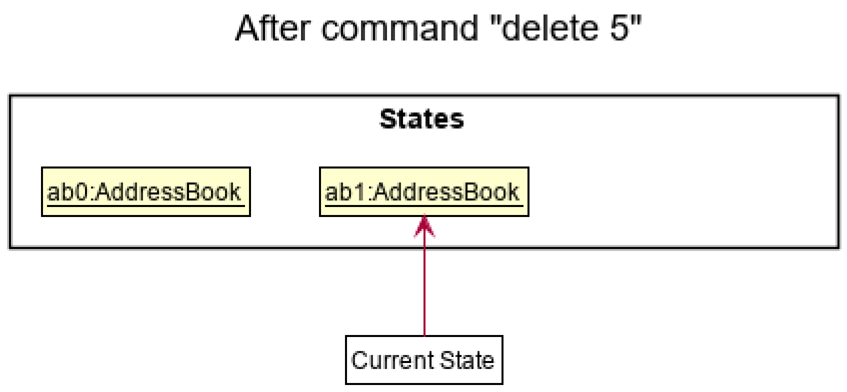
These operations are exposed in the `Model` interface as `Model#commitAddressBook()`, `Model#undoAddressBook()` and `Model#redoAddressBook()` respectively.

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

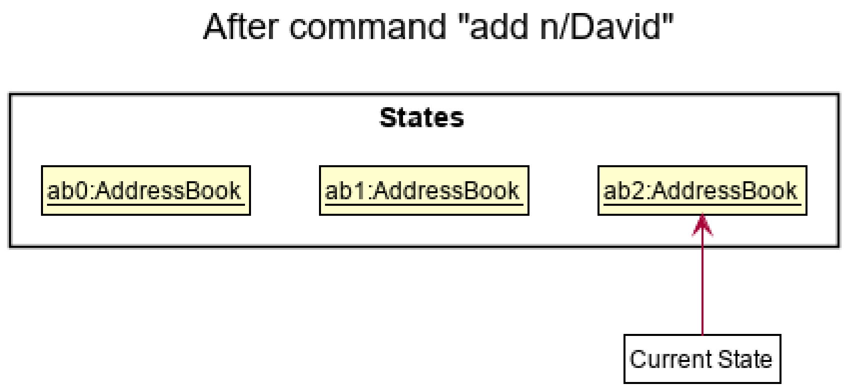
Step 1. The user launches the application for the first time. The `VersionedAddressBook` will be initialized with the initial address book state, and the `currentStatePointer` pointing to that single address book state.



Step 2. The user executes `delete 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 5` command executes to be saved in the `addressBookStateList`, and the `currentStatePointer` is shifted to the newly inserted address book state.

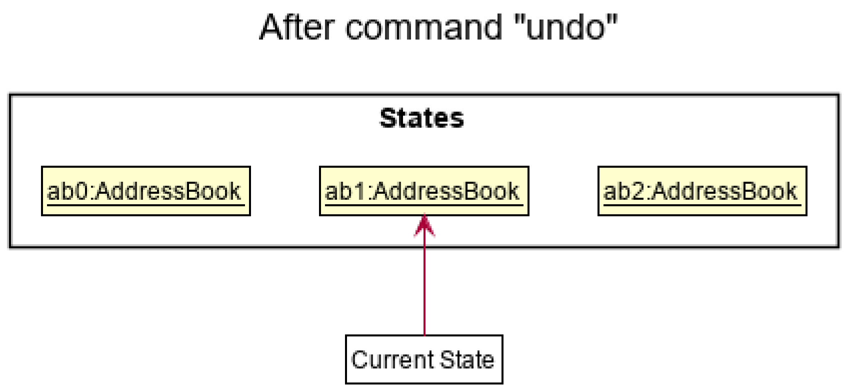


Step 3. The user executes `add n/David …` to add a new person. The `add` command also calls `Model#commitAddressBook()`, causing another modified address book state to be saved into the `addressBookStateList`.



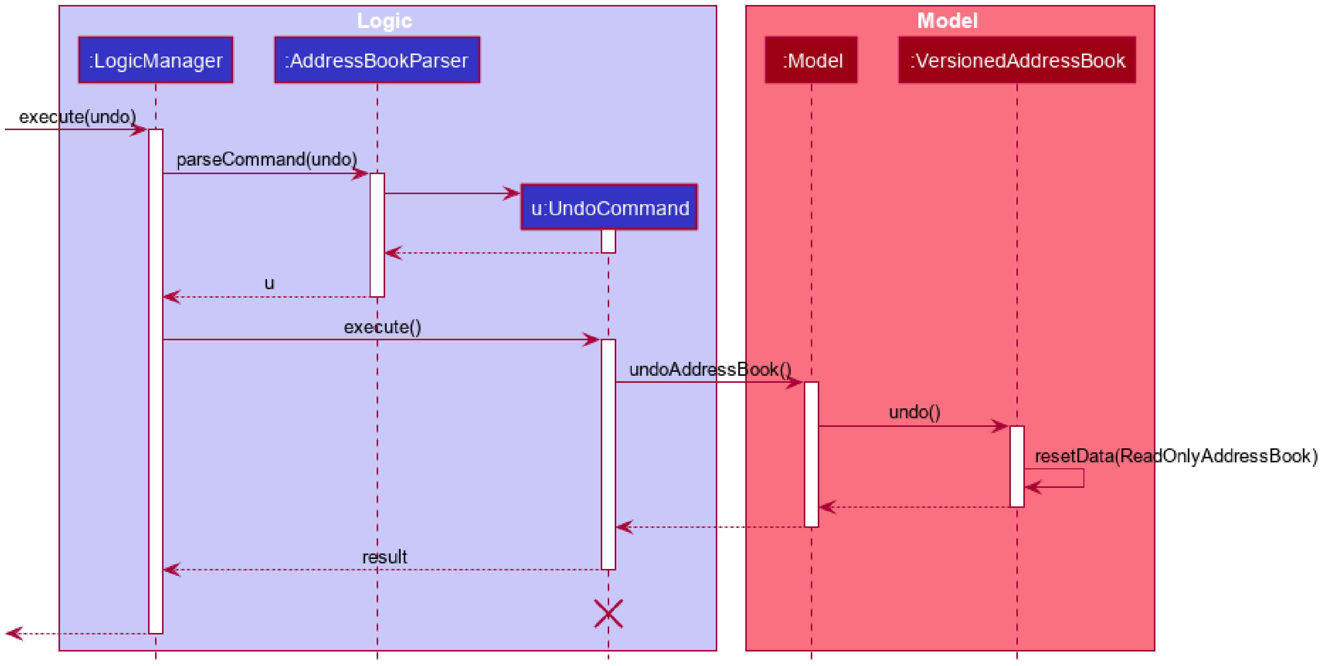
NOTE If a command fails its execution, it will not call `Model#commitAddressBook()`, so the address book state will not be saved into the `addressBookStateList`.

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:

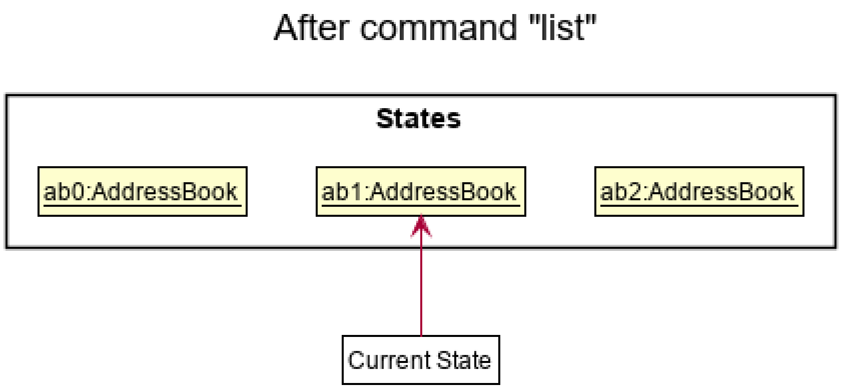


NOTE	The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

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

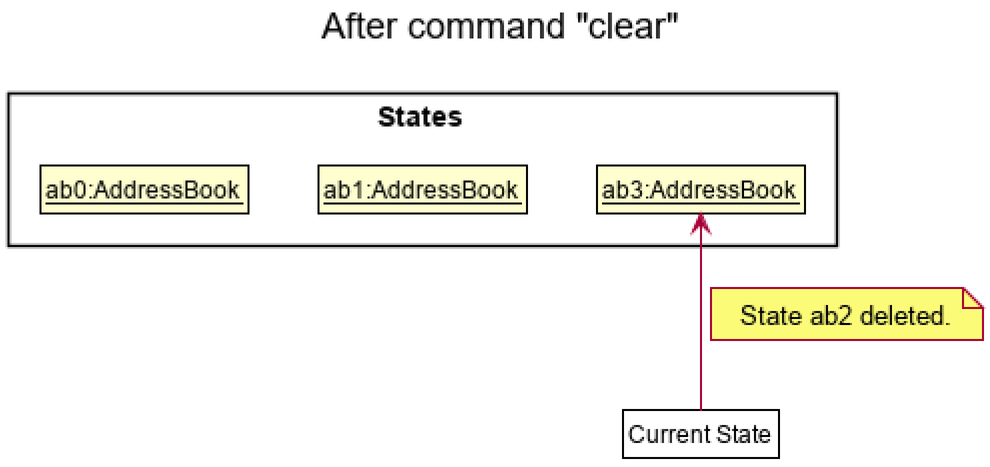
NOTE	If the `currentStatePointer` is at index `addressBookStateList.size() - 1`, pointing to the latest address book state, then there are no undone address book states to restore. The `redo` command uses `Model#canRedoAddressBook()` to check if this is the case. If so, it will return an error to the 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`, will usually not call `Model#commitAddressBook()`, `Model#undoAddressBook()` or `Model#redoAddressBook()`. Thus, the `addressBookStateList` remains unchanged.

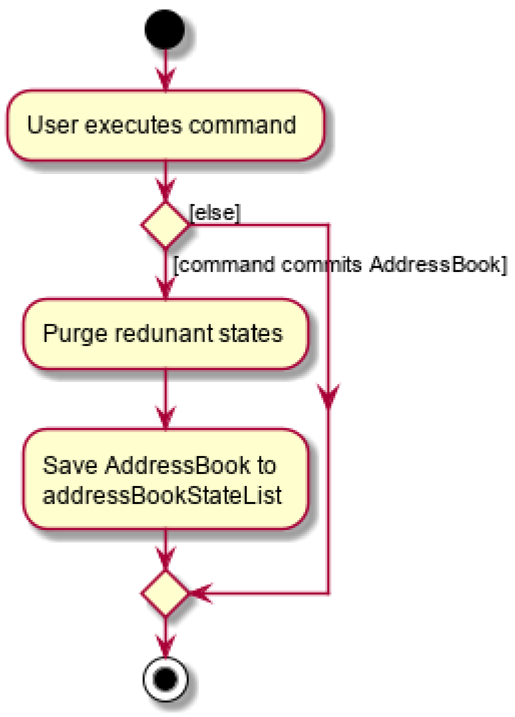


Step 6. The user executes `clear`, which calls `Model#commitAddressBook()`. Since the `currentStatePointer` is not pointing at the end of the `addressBookStateList`, all address book states

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:


User executes command
[else]
[command commits AddressBook]
Purge redunant states
Save AddressBook to addressBookStateList

3.5.2. Design Considerations

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.

3.6. [Proposed] Data Encryption

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

3.7. [Diary] Photo Manager

The photo manager pertains to components for storing, and displaying user specified photos on the disk.

3.7.1. Aspect: Models

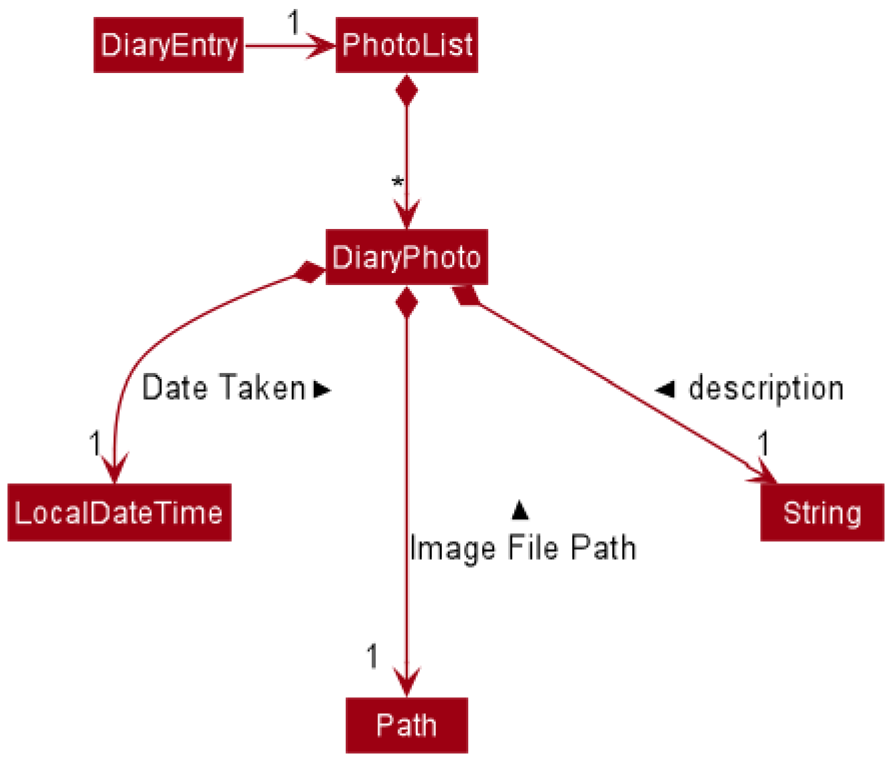


Figure 19. Class diagram of a PhotoList *as contained by a diary entry, and its contained models*

Photo

The model for a photo stored in memory is stored in the DiaryPhoto class.

It contains three key fields, that is, the imagePath, description, and dateTaken fields which are used to display key information of the image to the user. The imagePath and dateTaken were implemented respectively with the robust java apis of Path and LocalDateTime, while description is simply a String.

In addition, a JavaFX Image is also stored inside the DiaryPhoto (not shown in Figure 19, "Class diagram of a PhotoList as contained by a diary entry, and its contained models" for brevity), which holds the Image to use for displaying in an ImageView inside the user interface. The Image is cached this way, as the Image construction directly in the user interface involves costly I/O operations.

NOTE

Restrictions on fields during DiaryPhoto instance construction:

- Several restrictions on the description are enforced by class level Pattern matchers, such as the length of the description.
- While the image file path is parsed and checked using the java Files api, it is non-strict in that a path to an invalid image will result in the Image field referring to the default class level variable that specifies a placeholder image.
  - However, the original user entered file path is still stored inside the Model, to guard against accidental file deletion.

PhotoList

On the other hand, the DiaryPhoto models are contained within a PhotoList. It stores the photos in a

JavaFX `ObservableList`, so that changes are registered with the user interface. (see Section 3.7.2, “Aspect: User interface of photo manager”)

It also supports several convenience wrapper methods around the underlying `ObservableList`, tailored for use for the logic components.

3.7.2. Aspect: User interface of photo manager

The main `UiPart` component that displays photos is the `DiaryGallery`. It abides by the `Page` implementation (see Section 2.2.1, “Page API”), and is thus contained within, in one of `DiaryPage`s placeholders.

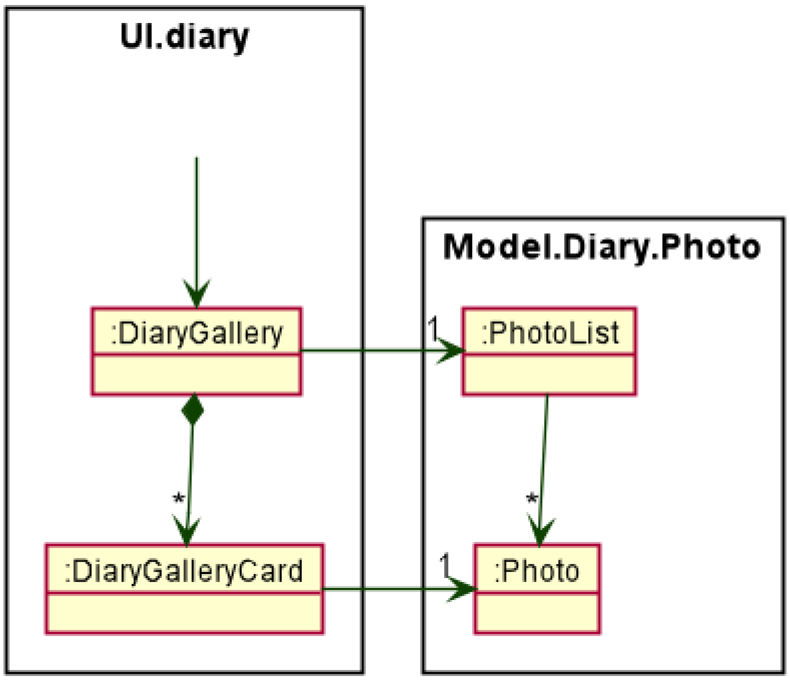


Figure 20. Object diagram of the diary gallery component, as contained by `DiaryPage` *(not shown)*

The main JavaFX component responsible for displaying the photos is a `ListView<DiaryPhoto>` component. The `ListView` obtains its data from the `PhotoList` of the `DiaryGallery`, which is automatically observed by the `ListView`.

Hence, changes in the `PhotoList`, such as the addition of a `DiaryPhoto` are immediately communicated to the user interface.

The `ListView` uses a simple custom `cell factory`, which sets the `ListCells` of the `ListView` to use `DiaryGalleryCards` as its graphic. `DiaryGalleryCards` are in turn generated in the `cell factory` using the `ListCell`s index and a `DiaryPhoto` instance.

`DiaryGalleryCards` display the information as supplied by the `DiaryPhoto` model using a series of `Labels` and one `ImageView`. Additionally, the index of the card as ordered in the `DiaryGallery` is also displayed, but not stored in the model.

3.7.3. Aspect: Logic of photo manager operations

The logic for photo manager plays to the same `PageParser` structure of parsing commands, that is, `DiaryParser` returns either `AddPhotoParser`, `DeletePhotoParser` when the appropriate command word is parsed, which in turn returns instances of `AddPhotoCommand` and `DeletePhotoCommand` respectively.

Logic aspect 1: Adding photos (through command line file path or os file chooser)

Following `DiaryParser` returning an instance of `AddPhotoParser` that calls `parse()` on the user specified arguments, a number of operations happen, as per the UML sequence diagram below ([AddPhotoParser parse sequence diagram]). The specifics of `getFilePath`, `parseDescription`, `parseDateTime` are detailed further down below.

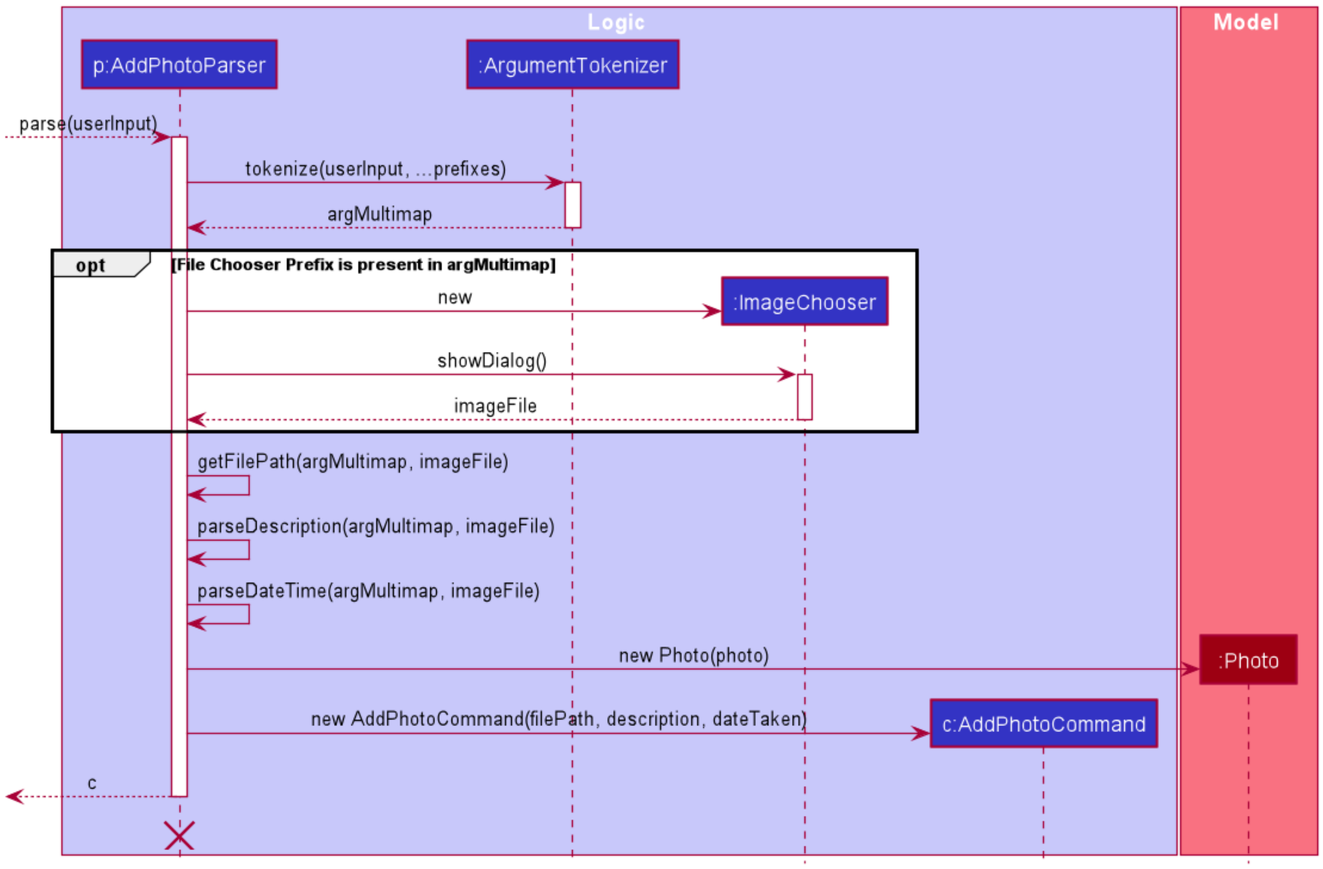

Logic
Model
p:AddPhotoParser
:ArgumentTokenizer
parse(userInput)
tokenize(userInput, ...prefixes)
argMultimap
opt
[File Chooser Prefix is present in argMultimap]
new
:ImageChooser
showDialog()
imageFile
getFilePath(argMultimap, imageFile)
parseDescription(argMultimap, imageFile)
parseDateTime(argMultimap, imageFile)
new Photo(photo)
:Photo
new AddPhotoCommand(filePath, description, dateTaken)
c:AddPhotoCommand
c

Figure 21. Sequence diagram of the parse method in AddPhotoParser

---

Parsing the image file path

- Using `ArgumentMultimap`, the file chooser prefix, "fc/", is checked for. If present, the OS file choosing gui is opened using `ImageChooser` (a simple extension of JavaFX's `FileChooser` enforcing image file extensions), and the data file path prefix is ignored.
- Otherwise, the presence of the data file prefix is checked, and its subsequent argument is validated as a valid image file.
- If the file chooser prefix is unspecified and the data file path is invalid, `AddPhotoParser` throws a `ParseException`

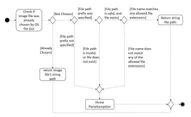

Check if image file was already chosen by OS file GUI
[Not Chosen]
[File path prefix was specified]
[File path is valid, and file exists]
[File name matches any allowed file extensions]
Return string file path
[Already Chosen]
[File path prefix not specified]
[File path is invalid, or file does not exist]
[File name does not match any of the allowed file extensions]
return image file's string path
throw ParseException

Figure 22. Activity diagram of getFilePath subroutine

Parsing the description of the photo

- If the description prefix is present, `AddPhotoParser` tries to construct the `DiaryPhoto` instance with the specified input. If validation of the description, as described in the Section 3.7.1, "Aspect: Models" fails, then a ParseException is thrown during the instance construction.
- Otherwise, the file name of the validated file from Section 3.7.3.1.1, "Parsing the image file path" (truncated to match `DiaryPhoto□s` description constraints) is used.

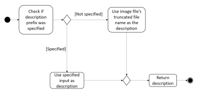

Check if description prefix was specified
[Not specified]
Use image file's truncated file name as the description
[Specified]
Use specified input as description
Return description

Figure 23. Activity diagram of parseDescription subroutine

Parsing the date of the photo

- If the date time prefix is present, `ParserDateUtil` is used to parse the date time as per the app level date formats. A `ParseException` is automatically thrown in the case of date parsing failure, by `ParserDateUtil`.
- Otherwise, the last modified date of the validated file from Section 3.7.3.1.1, "Parsing the image file path" is used.

The `DiaryPhoto` instance is then constructed, and passed to `AddPhotoCommand` which simply adds the `DiaryPhoto` to the current `PhotoList` of the `DiaryEntry`.

Logic aspect 2: Deleting photos

Following `DiaryParser` parsing the 'delphoto' command from the user, an instance of `DeletePhotoParser` is created, which parses the received arguments.

1. The `DeletePhotoParser` simply parses the arguments for a valid integer, failing which a `ParseException` is thrown.
2. An instance of `DeletePhotoCommand` is returned, which attempts a delete operation on the current `PhotoList` of the `DiaryEntry` with the specified index. A `CommandException` is thrown to alert the user if the index was out of bounds.

3.7.4. Design considerations

Feature	Alternative 1	Alternative 2
Validation of image file path	The first option is to implement the file path validation directly inside the `DiaryPhoto` model. This would have enforced a stricter level of validation on the image file path throughout the code, if an instance of `DiaryPhoto` needed to be instantiated somewhere else other than the `AddPhotoParser` for future use. However, since the storage model for `DiaryPhoto` (which is `JsonAdaptedDiaryPhoto`), initializes the model through deserializing the saved file path, this would have led to needing to a separate constructor for `DiaryPhoto` in the case that the file path was invalidated on app startup in order to create a placeholder image.	The second, chosen option, was to implement the file path validation inside the parser itself. Although this option limited the validation to only the 'addphoto' command, it allowed for leeway in image path validation in other areas such as `JsonAdaptedDiaryPhoto`, where it is possible for deletion of an image file by the user, outside of the application, to invalidate the stored file path and erroneous data to be loaded on application start. Moreover, Since the function for parsing the image file can and was abstracted into a single utility function, any other areas in future development needing this functionality can simply reuse this code. Overall, this leads to a more robust behaviour of the application, while providing the same level of extensibility as the first option.

3.8. [Diary] Diary Entry Text Editing

The diary entry is capable of displaying text with inline images, or lines consisting of only images.

There are two primary facets of input styles to this feature, one being commands that edits a part or the whole of the entry through the command line input, and the other being the JavaFX text editor.

3.8.1. Aspect: Models

The main model abstraction holding the data of an entry is the `DiaryEntry` class.

It stores three key fields, namely:
1. An `Index` denoting the day the entry is for
2. A `String` written by the user in the domain specific language (see Section 3.8.2.1, “Entry text parsing”) required by the user interface.
3. A `PhotoList` storing the photos of the entry, as described in Section 3.7.1, “Aspect: Models”.

The `DiaryEntry` models are contained within a `DiaryEntryList`, which enforces the uniqueness of the `Index` (denoting the day index) of each `DiaryEntry`, and supports common list operations.

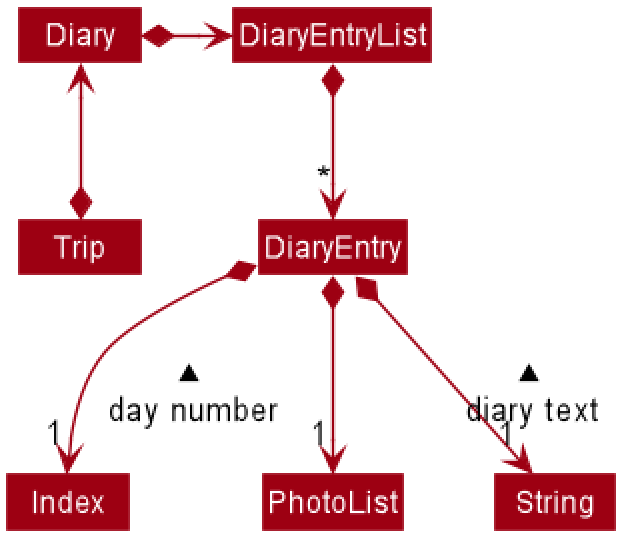


Figure 24. Class diagram of the models used in diary text editing

As one of the desired specifications of our application was to allow the user commands, and edits made directly to the edit box to be non final until the `done` command is executed, a separate buffer model, `EditDiaryEntryDescriptor`, was needed to store the edit information.

This buffer model stores the same `PhotoList` and `Index` as the initial `DiaryEntry` it is constructed from, but the diary text references a different String, that is, the buffered diary text String.

3.8.2. Aspect: User interface

Multiple `UiPart` components come into play in displaying the diary entry. However, `Page` implementation (see Section 2.2.1, “Page API”) is still followed, and all components are thus contained within, in one of `DiaryPage`s placeholders.

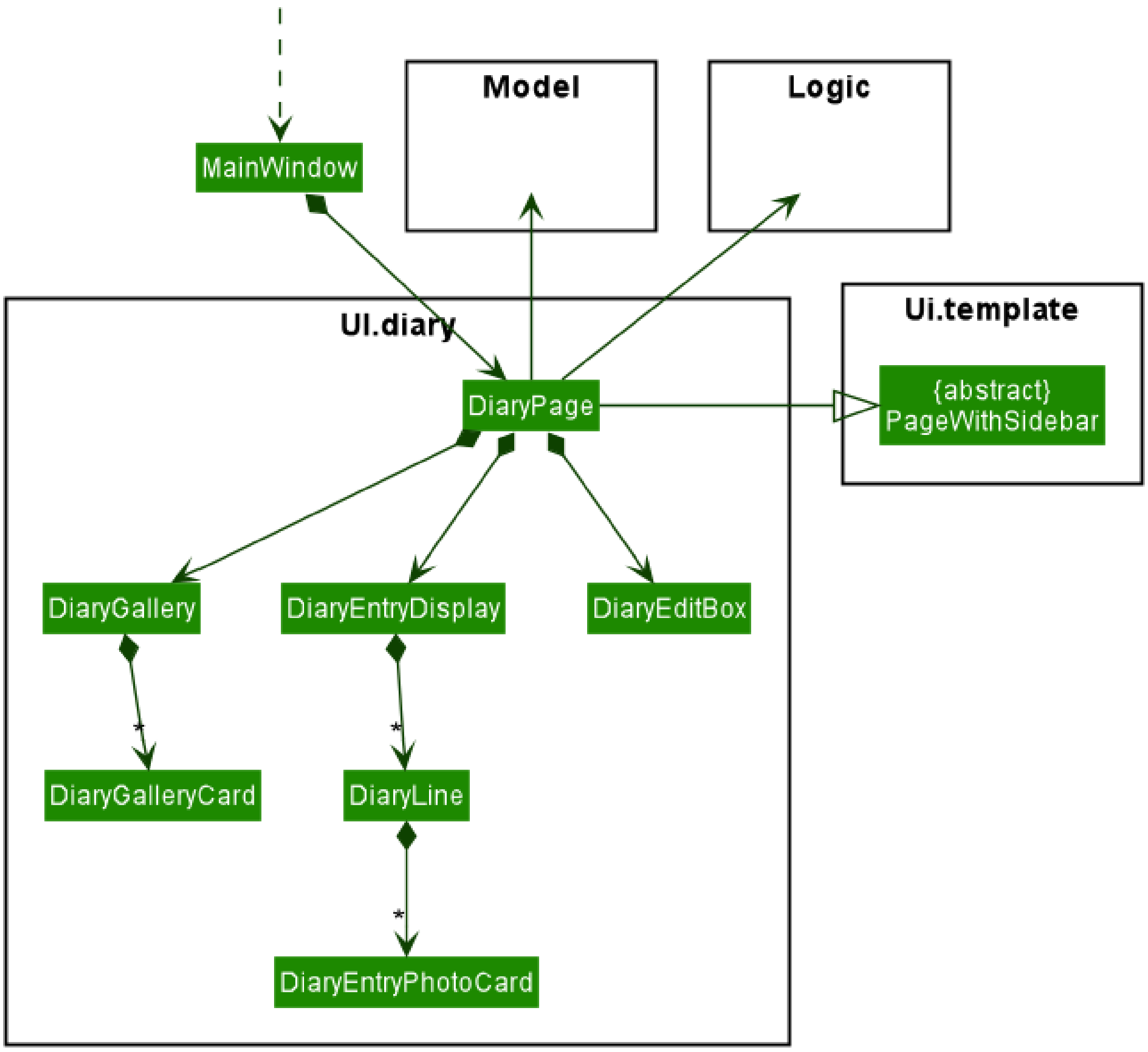


Figure 25. Class diagram showing the user interface of the main diary entry text display

NOTE In the diagram above, all parts and subparts of the composition of `DiaryPage` extend from `UiPart`, although not shown.

The `DiaryEntryDisplay` is the component responsible for displaying the content of the `DiaryEntry` model. Internally, it uses a JavaFX `ListView<CharSequence>` with a custom cell factory that returns `DiaryTextLineCell` (as detailed in Section 3.8.2.1.3, "Graphic of `ListView` cells in `DiaryEntryDisplay`"). `DiaryTextLineCells` in turn uses the `DiaryLine UiPart` as its graphic.

Entry text parsing

In both facets of input styles, special entry text parsing is required to display the various formats of lines, and dynamic text updates that occur when the text in the text editor is changed should propagate to the display immediately.

To accomplish this, the internal `ListView` is set to observe the paragraphs of the `DiaryEditBox`, which is done in the constructor of `DiaryEntryDisplay` during the initialisation of `DiaryPage`.

The two facets of inputs dictate *two separate ways the paragraphs can change*.

1. Changes as a result of edits by the user in the text edit box

In this case, the edits to the `TextArea` input in `DiaryEditBox` are immediately propagated to the observable paragraphs, since the `ListView` was set to observe the same list provided by `DiaryEditBox`.

2. Changes as a result of user commands

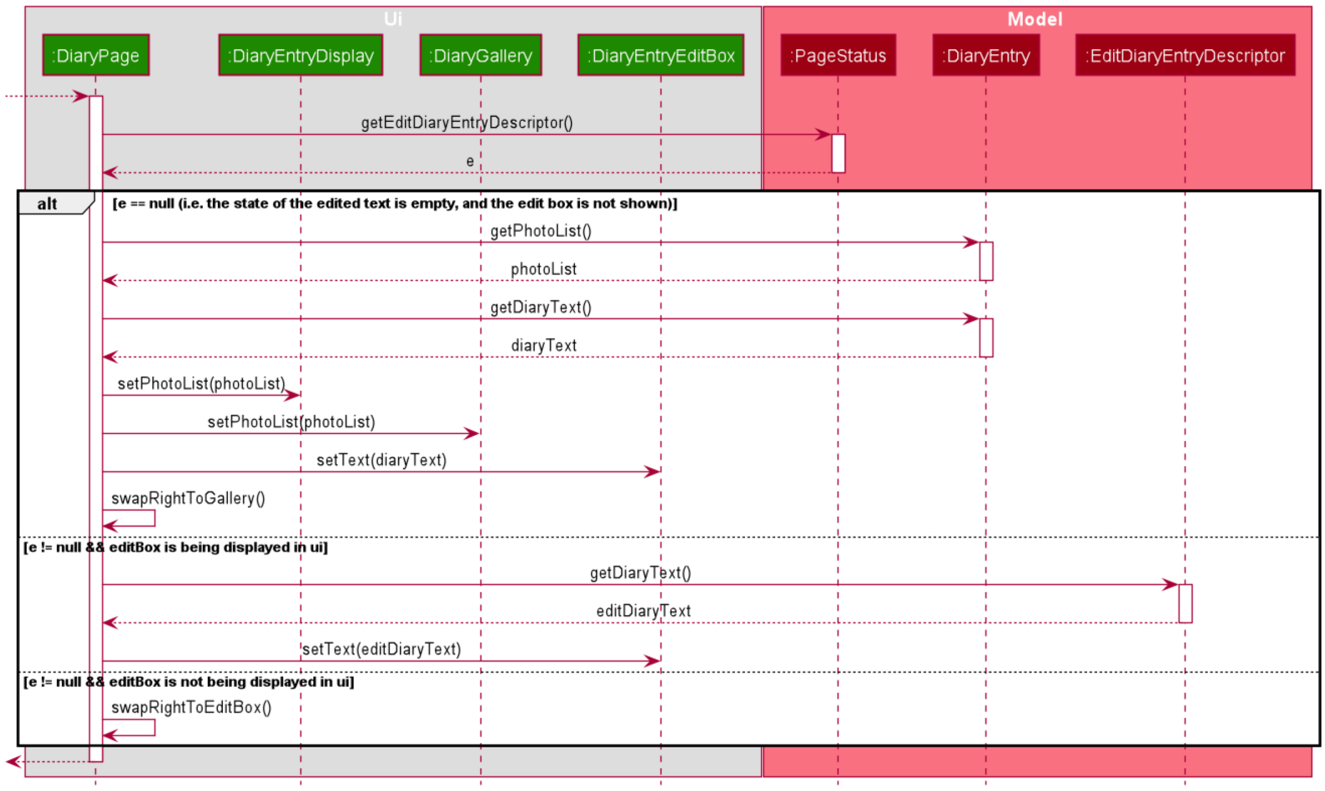


Figure 26. Sequence diagram of updating of DiaryPage UI post command execution

1. The `model` is updated, depending on whether the edit box is currently shown to the user.
   1.1. The edited but uncommitted text stored in the current `EditDiaryEntryDescriptor` will be updated if the edit box is shown. (second branch in the diagram Figure 26, “Sequence diagram of updating of DiaryPage UI post command execution”)
   1.2. Otherwise, the current `DiaryEntry` in the `PageStatus` of the `model` is updated immediately. (first branch in the diagram Figure 26, “Sequence diagram of updating of DiaryPage UI post command execution”)
2. The text in the `DiaryEntryEditBox` is then refreshed with the updated `model` in the `fillPage` callback function executed by `MainWindow` (as per the `Page` api), resulting in the changes reflecting in the observable paragraphs.

Graphic of `ListView` cells in `DiaryEntryDisplay`

The `ListView` of `DiaryEntryDisplay` uses a custom cell factory and cell implementation, that is, `DiaryTextLineCell`.

Once the data has been updated in the above two ways, the `ListView` receives the notification for which cell(s) to update.

The parsing is done in the inner class `DiaryTextLineCell` based on the text line received, using a customised regex pattern. `DiaryTextLineCell` then creates new instances of `DiaryLines` based on the parsed input, setting them as the `graphic` for the `ListCell`.

NOTE For `DiaryLines` with photos, the parsing process uses the photoList as set in the `DiaryPage's fillPage` method. (see branch 1 in Figure 26, “Sequence diagram of updating of DiaryPage UI post command execution”)

3.8.3. Design considerations

Numerous design decisions and comprimised had to be made due to the desired specifications of text editing and displaying.
Specifically, the following had to be achieved :

- Changes to text in the `DiaryEntryEditBox` must reflect immediately in the `DiaryEntryDisplay` to provide visual cue to the user.
- While the `DiaryEntryEditBox` is active, commands that edit the entry must behave like they edit the `DiaryEntryEditBox` directly. That is, the changes should not be committed immediately.
- In general, where mentioned below, performance was favoured because of how a singular diary line can present both multimedia and text to the user, which puts a considerable strain on the system.

Aspect	Option	Implementation
Updating of UI		

Subsequent iterations of development and testing showed that this erased the performance benefit of implementing the observable list, presumably due to the overhead of firing commands whenever the text in the `DiaryEntryEditBox` changed.

Aspect	Option	Implementation
	3	The last option was to set the `ListView` to only observe `ObservableList` of paragraphs already provided by the `TextArea` JavaFX component located in `DiaryEntryEditBox`. Edits to the paragraphs in the `DiaryEntryEditBox` would be directly reflected in the `DiaryEntryDisplay`, without the additional overhead of executing commands whenever the text in the edit box changes. Instead, the text edit command is only executed when the edit box loses focus. On the other hand, edits using commands would reflect in the UI through setting the text of the `DiaryEntryEditBox`. A hybrid solution built upon **alternatives 2 and 3** was also considered, in that the `DiaryEntryDisplay` s would be alternate between observing the `DiaryEntryEditBox` and the `DiaryEntry model`. However, this also proved to be costly, as changes from the edit box cannot be communicated on a per paragraph basis to the model when focus is lost, defeating the performance benefit of the `ObservableList`. Ultimately, this also required maintaining as many `` ObservableList`s `` as there were diary entries in memory, presenting a significant memory overhead to the application. Having considered the performance impacts of **alternatives 1 and 2**, and the desired specifications of the application, the chosen solution was thus **alternative 3**.
High level composition of `DiaryEntry Display` component	1	The first solution to was to make `DiaryEntryDisplay` hold a JavaFX `TextFlow` component, which supports displaying images alongside text. Although it supports various apis to format and position text, displaying multimedia with it required complex parsing logic of the `DiaryEntry` text to achieve desired positioning. Moreover, the parsing would be re run on the entire text of the `DiaryEntry` for any form of user input, posing a clear performance downside.
	2	The second solution is to use a wrapper (`` DiaryEntryDisplay) around a `ListView `` containing `DiaryLine` s. (see Figure 25, "Class diagram showing the user interface of the main diary entry text display") On one hand, this increases extensibility, as the the graphic of a `ListViewCell` (`DiaryTextLineCell`) is not fixed. This allows *building other variants of diary lines easily*, such as a diary line containing a playable audio file. Secondly, `ListViews` render only the visible cells on the screen. Apart from the reducing the amount of nodes loaded in the JavaFX scene graph, it also allows running the parsing logic on only parts (paragraphs) of the text in the `DiaryEntry` model. This results in a considerable performance benefit.

3.9. [Inventory] The Inventory List

In the application, there can be two types of Inventory Lists:

1. Overall Trip Inventory List

An overall trip inventory list is the inventory list of all the items the user needs for the trip.

Each trip only has **one** overall trip inventory list.

2. An Event inventory list

An event inventory list is the inventory list of all the items the user needs for the event.

Each trip can have **mutiple** event inventory lists.

Therefore, in a trip, all the events inventory lists are subsets of one overall trip inventory list.

3.9.3. Aspect : Model

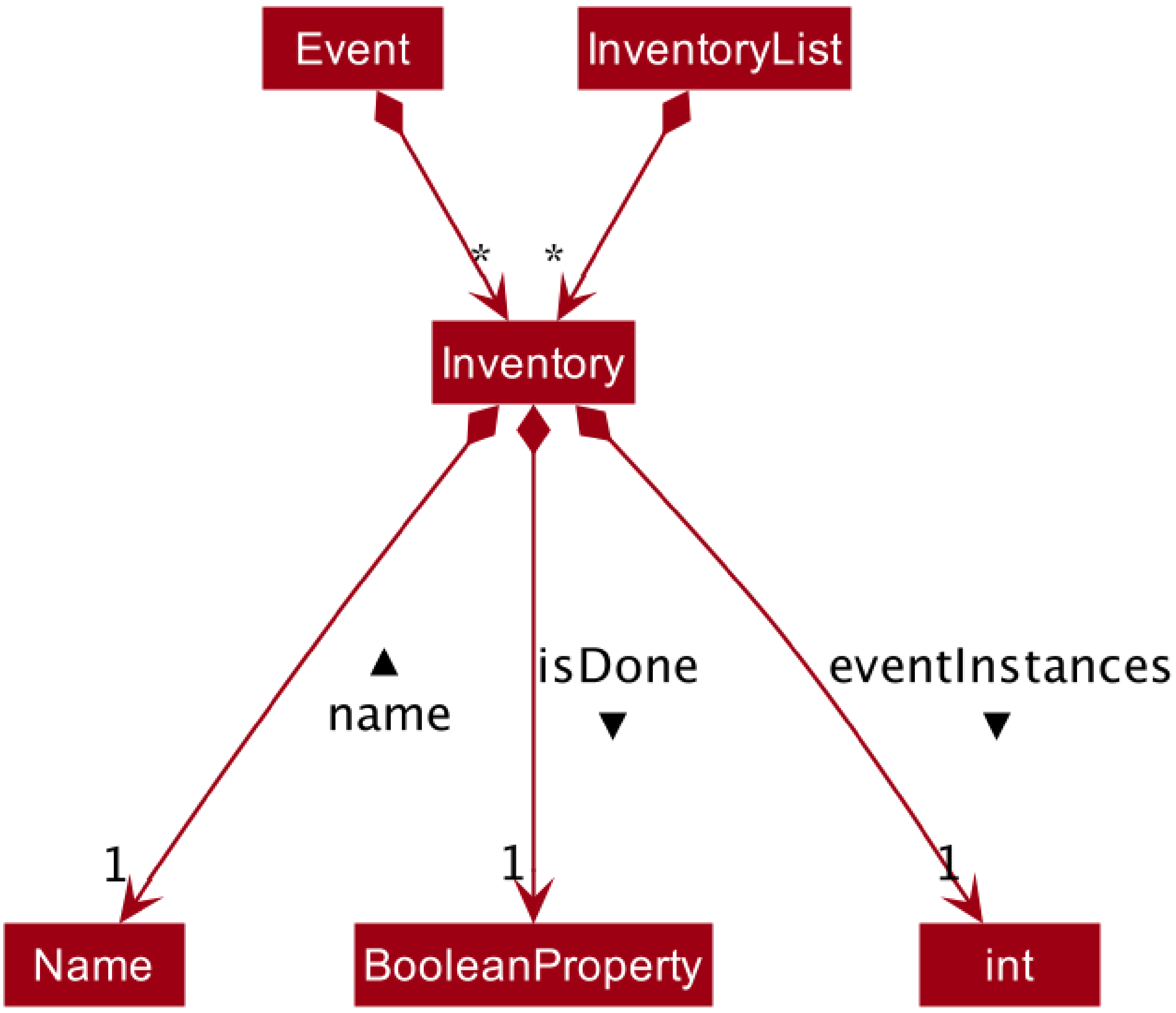


Figure 27. Class diagram of a 'InventoryList' and an `Event`, and its contained models

Inventory

The model for an inventory item stored in memory is stored in the `Inventory` class.

It contains three attributes: the `name`, `isDone` and `eventOccurances`. The `name` attribute is implemented using a custom built `Name` class which ensures that the `name` is valid. `isDone` was implemented using a BooleanProperty and indicates whether the inventory item has been packed. While, `eventInstances` is an integer which represents the total number of `Event` models that have the same inventory item.

Event

`Inventory` models are contained within an `Event`, if the corresponding inventory items are part of an event. `Event` stores the `Inventory` models in an `ArrayList`. Therefore, each `Event` model contains an Event Inventory List (as defined above).

Since, currently, the application does allow the user to alter the 'isDone' toggle of an `Inventory` model contained in an event, the `isDone` attribute is never set once it is is initialised (as `false`).

Furthermore, as the `eventInstances` attribute is only relevant to the Overall Trip Inventory List, the `eventInstances` attribute is set as `-1`.

InventoryList

All unique (by name) `Inventory` models in any `Event` model is contained within an `InventoryList`. It stores the inventory items in a JavaFX `ObservableList`. Therefore, the `InventoryList` contains the Overall Trip Inventory List (as defined above).

An `ObservableList` was chosen as the user interface requires the changes in the inventory list to be displayed instantly. Furthermore, `isDone` was implemented using a `BooleanProperty` instead of a simple `boolean` to instantly detect and display changes in any inventory item’s `isDone` value.

Additionally, an `InventoryList` can also contain miscellaneous inventory items (not contained in any event).

3.9.4. Aspect : User Interface of Overall Trip Inventory List (with Model `InventoryList`)

The Overall Trip Inventory List (with Model `InventoryList`) is displayed in the `TableView<Inventory>` of the `InventoryPage`, as seen in the object diagram:

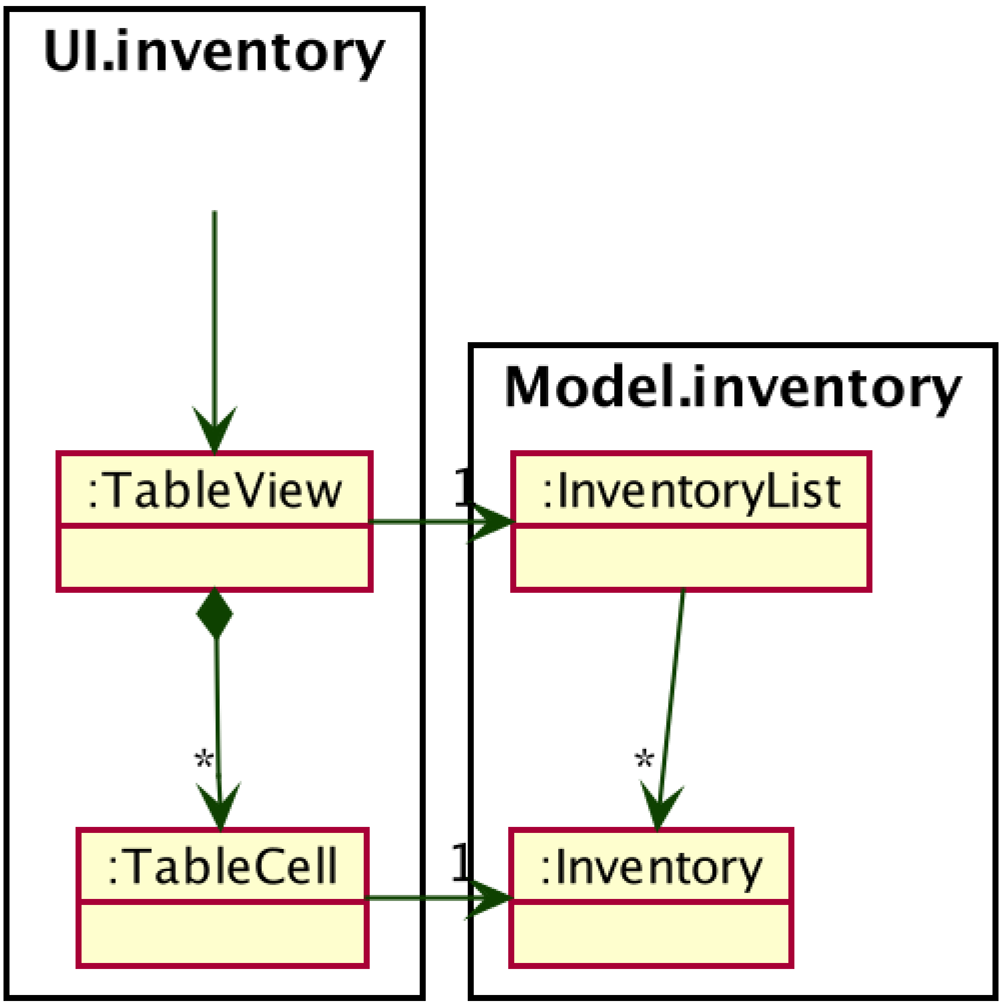


Figure 28. Object diagram of table view component, as contained by `InventoryPage` *(not shown)*

The TableView has 3 columns. The first column numbers the items. The second column displays the item’s name. And the thrid column displays whether the item’s `isDone` property is true or false (through a checkbox).

Both the first and the second column’s `cellValueFactory` were set to `new PropertyValueFactory(STRING)` with String `STRING`. However, the third column was set to `CheckBoxTableCell.forTableColumn(BOOLEAN)` with Boolean `BOOLEAN`.

3.9.5. Aspect : User Interface of Event Inventory List (with Model `Event`)

The Event Inventory List (with Model `Event`) is displayed in the `ListView<Inventory>` of the `EventPage`, as seen in the object diagram:

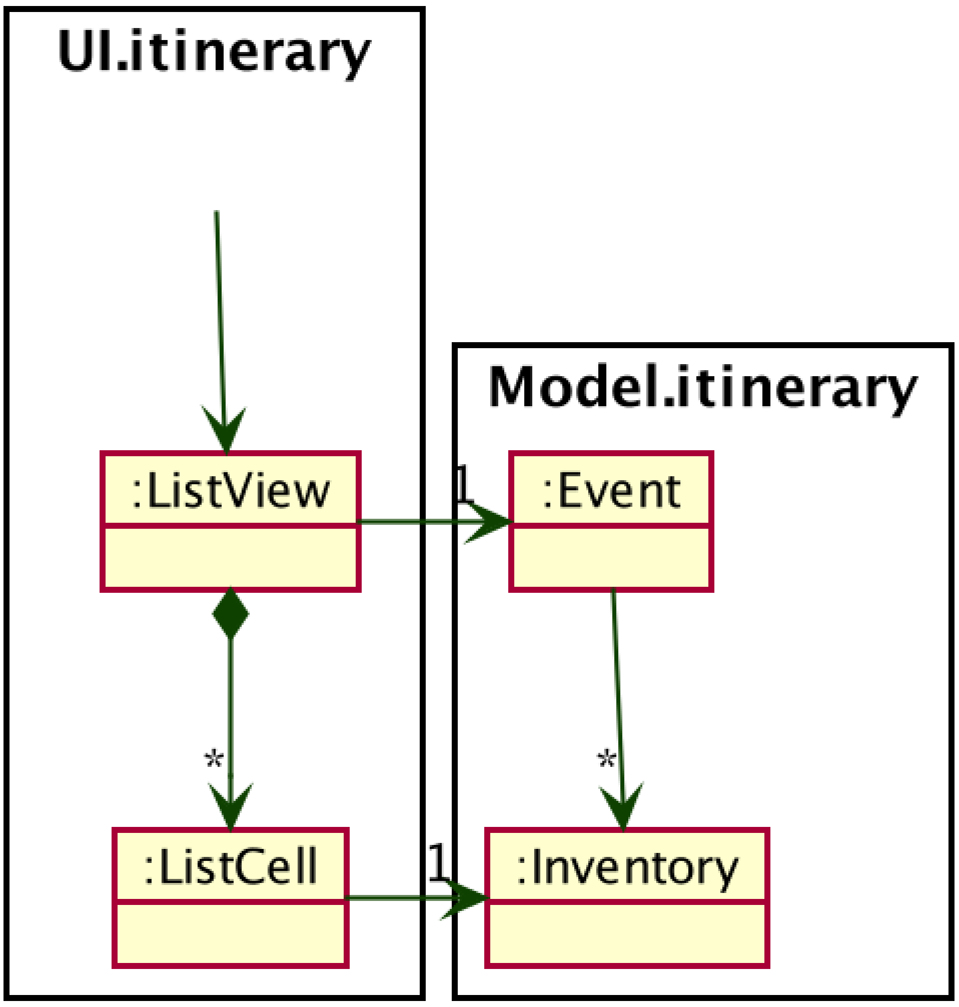


Figure 29. Object diagram of list view component, as contained by `EventPage` *(not shown)*

The `ListView` uses the CSS styling of `resources/view/Inventory/InventoryListViewTheme.css' by calling `listView.getStylesheets().add(CSS_FILE_PATH)` with the File Path `CSS_FILE_PATH`)

Furthermore, the reason why `ListView` and `ListCell` are in package `Ui.itinerary` is because `EventPage` is in itinerary `Ui.itinerary`. Similarly, `Event` is in package `Model.itinerary`

3.9.6. Aspect : Logic of adding item to Overall Trip Inventory List (with Model `InventoryList`)

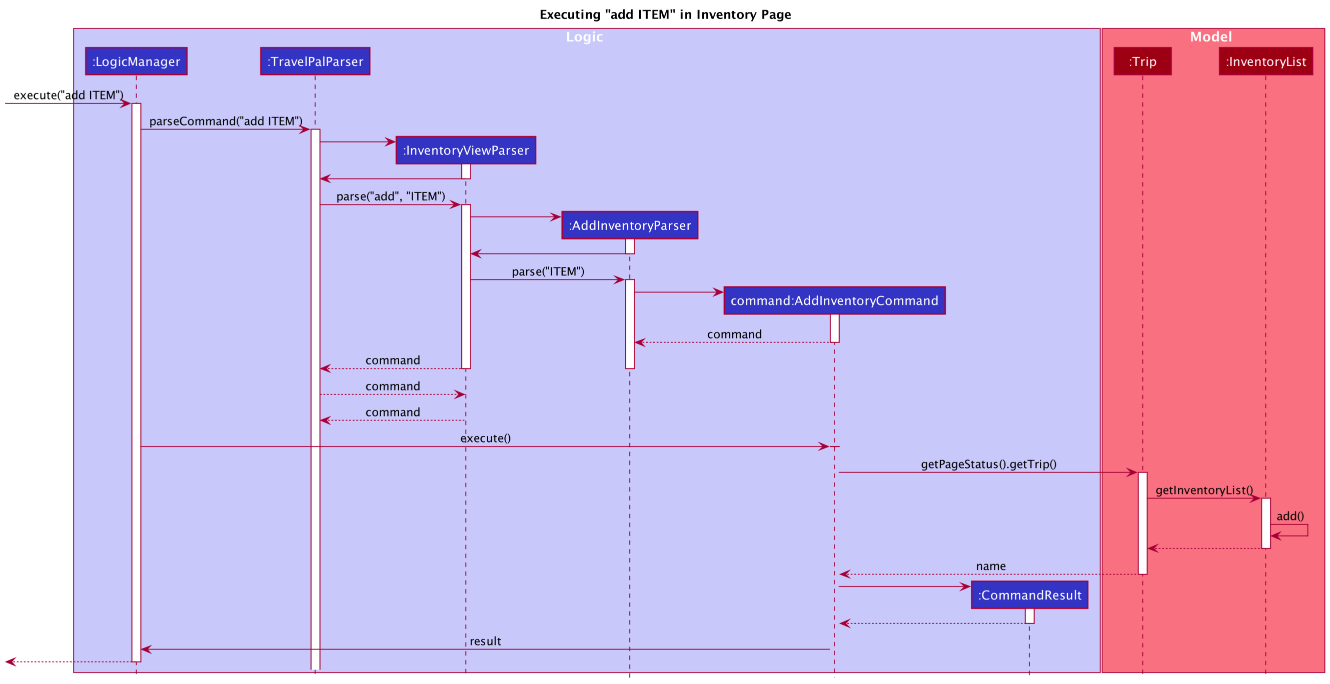


Figure 30. Sequence diagram for adding an item to the Overall Trip Inventory List

From the sequence diagram, it can be observed that only if `InventoryViewParser` and `AddInventoryParser` returns a Command, will the Command be executed. This ensures the AddCommand is only executed when intended.

Moreover, as seen from the sequence diagram, whenever the `AddInventoryCommand` is executed, it will always return the `InventoryView` that corresponds to the Trip that the user is correctly on - ensuring that there is only one Overall Trip Inventory List.

Furthermore, the `InventoryList` calls its own internal function of `add` to add an item to its list - which loops through the entire list to make sure that the item to be added does not already exist - otherwise it does not add it . Therefore, this also allows all the items in InventoryList to be unique.

3.9.7. Possible Improvement in Future Release

Currently, both the Event Inventory List and the Overall Trip Inventory List utilise a common `Inventory` class to store inventory items.

However, an inventory item in the Event Inventory List does not need to keep count of the `eventInstances` attribute. Although, the current solution of setting `eventInstances` to -1 allows both classes to share the `Inventory` class, it might not be scalable and there may lead to bugs in the future.

Therefore, in a future update, it would be rational to remove make the `Inventory` class an abstract class with just a `Name` attribute. And have 2 different classes extend it for the 2 different use cases.

3.10. 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 3.11, “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

3.11. 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.

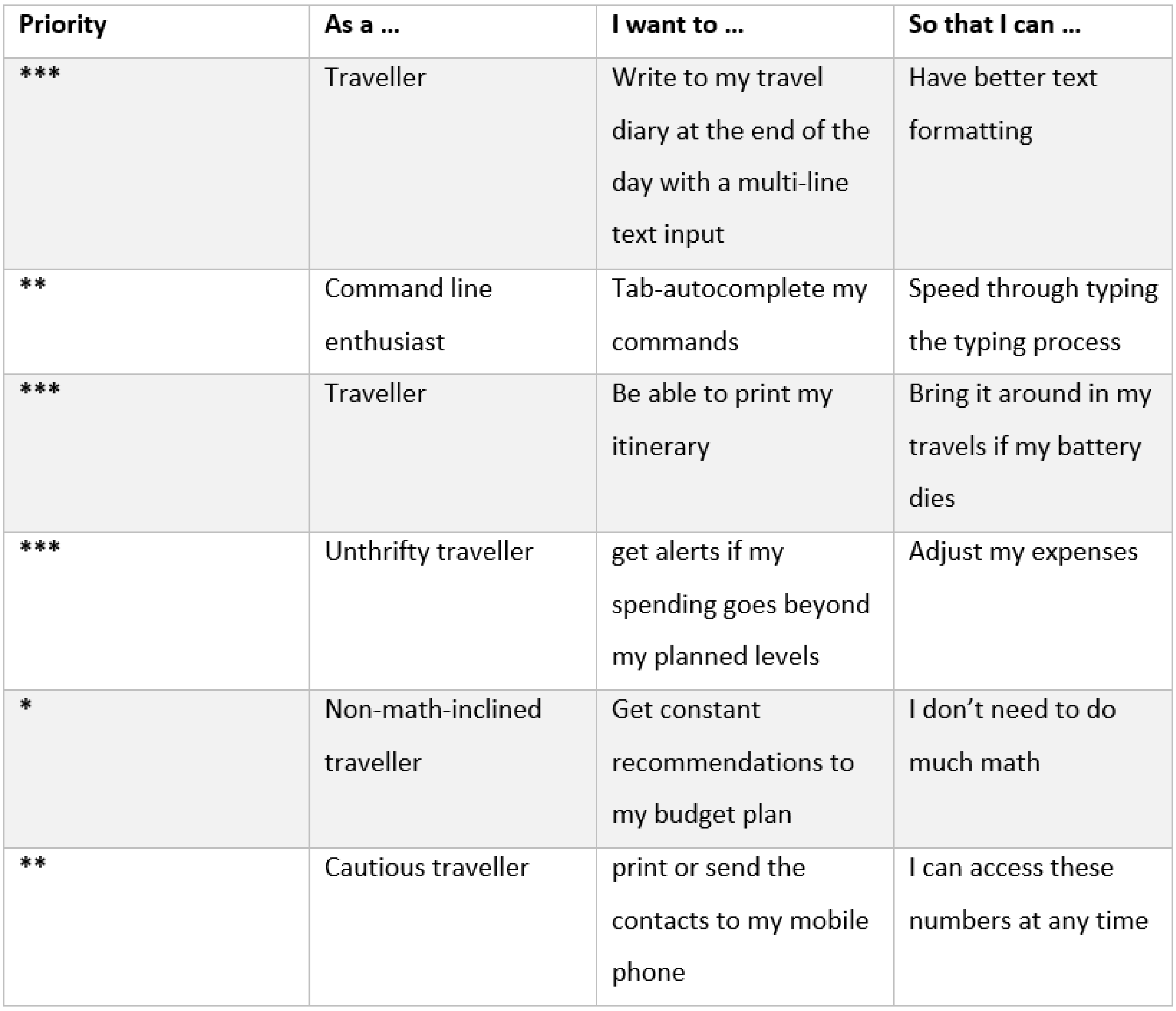
Appendix A: Product Scope

Target user profile:

- Has a need to manage multiple trips
- Prefers using a notebook to other types
- Frequently uses the computer while overseas
- Wants to micromanage all parts of their trips
- Wants to plan all details of the trip before leaving
- Wants to manage a trip even without an internet connection

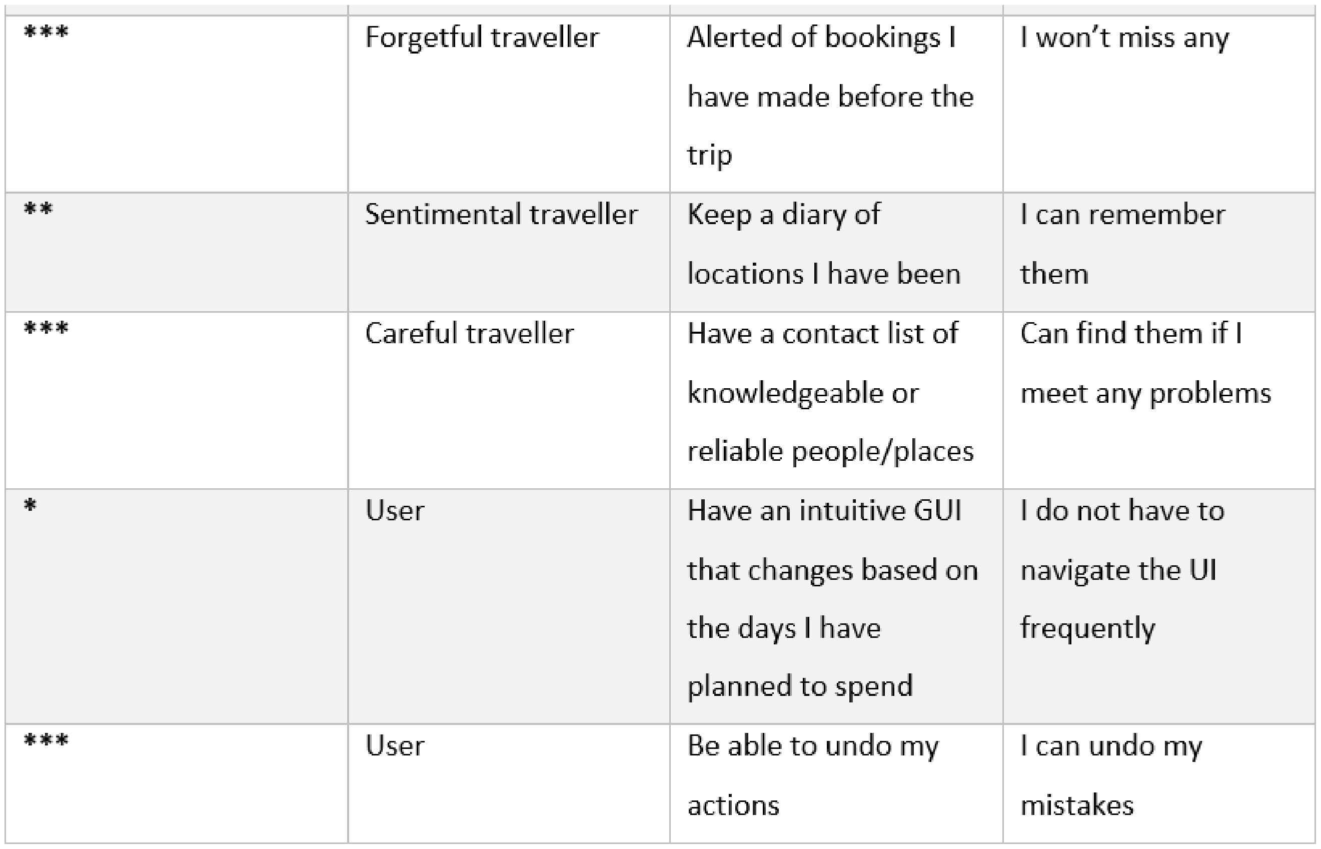
Value proposition: Able to micromanage a trip and access one’s plans more conveniently than traditional forms of trip planning

Appendix B: User Stories

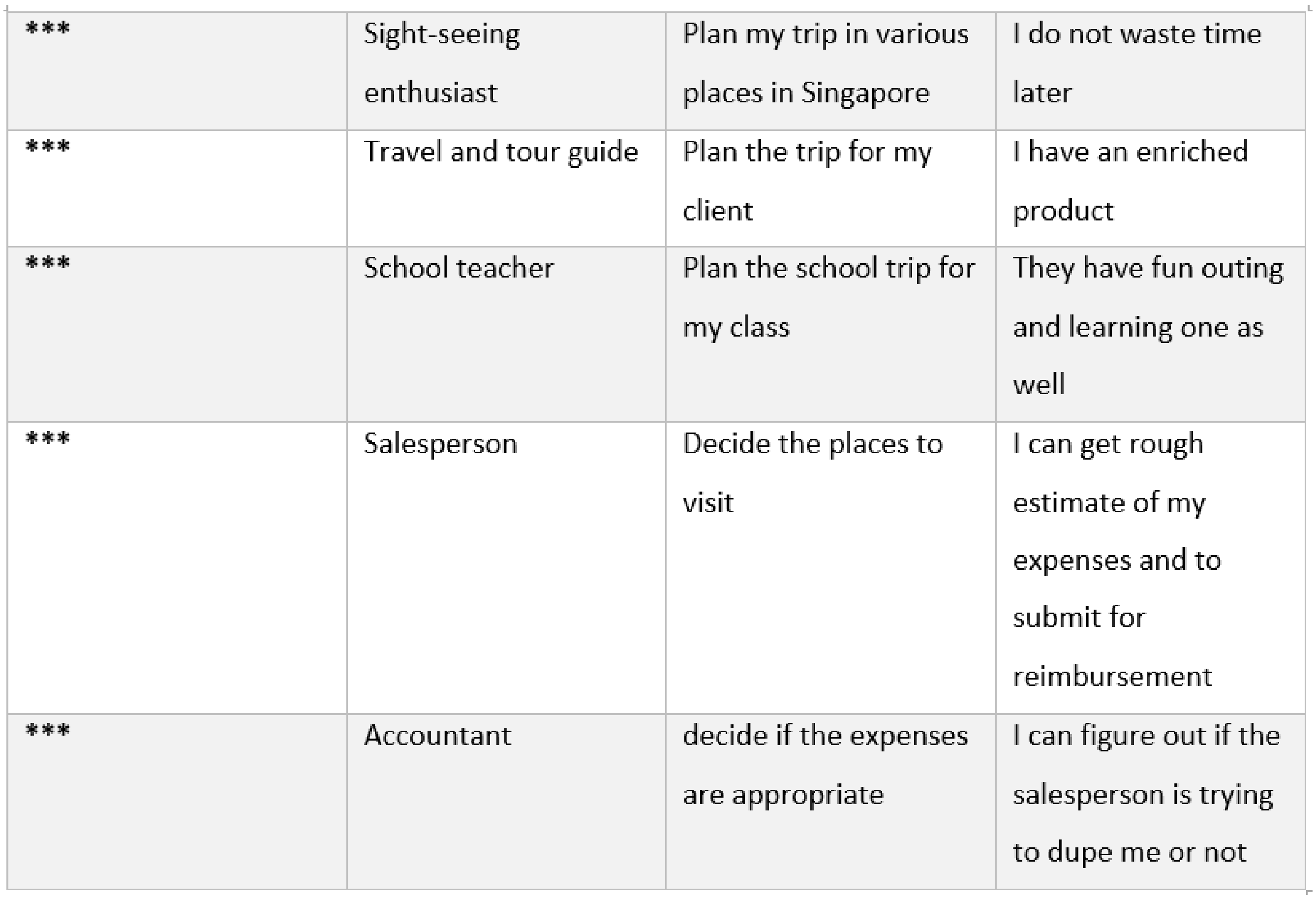


Priority	As a …	I want to …	So that I can …
***	Traveller	Write to my travel diary at the end of the day with a multi-line text input	Have better text formatting
**	Command line enthusiast	Tab-autocomplete my commands	Speed through typing the typing process
***	Traveller	Be able to print my itinerary	Bring it around in my travels if my battery dies
***	Unthrifty traveller	get alerts if my spending goes beyond my planned levels	Adjust my expenses
*	Non-math-inclined traveller	Get constant recommendations to my budget plan	I don’t need to do much math
**	Cautious traveller	print or send the contacts to my mobile phone	I can access these numbers at any time

***	Traveller	Make a detailed schedule for my trip	I don’t have to waste time deciding which places to visit next
*	Traveller	Visualise my routes on a map	Use my time more efficiently
***	Forgetful traveller	Make a checklist for items to bring	I won’t forget any
***	Traveller	Have a copy of my trip schedule in my mobile devices	Carry it around easily
***	Traveller	To be easy to remember	
***	User	Not need to edit multiple areas of the app after a single command	
*	Traveller	Manage my expenses based on the country I am visiting	I don’t need to convert the currency myself



***	Forgetful traveller	Alerted of bookings I have made before the trip	I won’t miss any
**	Sentimental traveller	Keep a diary of locations I have been	I can remember them
***	Careful traveller	Have a contact list of knowledgeable or reliable people/places	Can find them if I meet any problems
*	User	Have an intuitive GUI that changes based on the days I have planned to spend	I do not have to navigate the UI frequently
***	User	Be able to undo my actions	I can undo my mistakes



***	Sight-seeing enthusiast	Plan my trip in various places in Singapore	I do not waste time later
***	Travel and tour guide	Plan the trip for my client	I have an enriched product
***	School teacher	Plan the school trip for my class	They have fun outing and learning one as well
***	Salesperson	Decide the places to visit	I can get rough estimate of my expenses and to submit for reimbursement
***	Accountant	decide if the expenses are appropriate	I can figure out if the salesperson is trying to dupe me or not

Appendix C: Use Cases

Use case: UC1 - Add Trip

MSS

1. User requests to **Trip Manager** to list trips
2. TravelPal shows a list of **Trips**
3. User requests to add a specific **Trip** to the list
4. User <span class="underline">edits the **Trip** (UC2)</span>
5. TravelPal adds the **Trip**
6. TravelPal shows the list of **Trips**. Use case ends.

Extensions

5a. The trip added clashes with another trip

5a1. TravelPal shows an error message

5a2. TravelPal does not discard information the user has provided

5a3. TravelPal displays the **Edit Trip** page containing the user’s previous input

5a4. TravelPal requests the user to change the dates of the **Trip**

Steps 5a1-5a2 are repeated until no clashes occur between trips

<span class="underline">Use case: UC2 – Edit Trip</span>

MSS

1. User chooses to edit specific **Trip**
2. Travelpal shows **Edit Trip Screen** with fields to edit/enter
3. User edits the information in the specified **Trip**
4. User submits the details and confirms the edit. Use case ends.

Extensions

3a. User enters an invalid field

3a1. TravelPal shows an error message

3a2. TravelPal does not edit invalid field

Use case continues at step 2

3b. User requests to list of **Days** in the trip

3b1. TravelPal shows a list of days to the user (can be empty)

3b2. User chooses to <span class="underline">add/edit/delete (UC4/5/6) **Day**</span>

Use case continues at step 4

4b. User leaves necessary information empty

4a1. TravelPal shows an error message

4a2. TravelPal does not submit the details and does not confirm the edit

4a3. User enters new data

Steps 4a1-4a3 are repeated until the data entered are non empty

Use case ends.

Use case: UC3 – Delete Trip

MSS

1. User requests to **Trip Manager** to list **Trips**
2. TravelPal shows a list of **Trips**
3. User requests to delete a specific **Trip** in the list
4. TravelPal deletes the **Trip**

> Use case ends ` **Extensions**

2a. The list is empty

Use case ends

3a. The **Name** provided is invalid

3a1. TravelPal shows an error message

3a2. TravelPal does not delete any trips

Use case ends

Use case: UC4 – Add Day

MSS

1. User chooses to add a **Day** to a specified **Trip**
2. User *edits the day (UC5)*
3. TravelPal saves the **Day**

Extensions

3a **Day** added clashes with other days in the **Trip**

3a1. TravelPal shows an error message

3a2. TravelPal does not discard information the user has provided

3a3. TravelPal displays the **Edit Day** page containing the user's input

3a4. TravelPal requests the user to change the date of the **Day**

Steps 3a1 – 3a4 are repeated until the user provided non clashing date

Use case: UC5 – Edit Day

MSS

1. User requests to edit specific **Day**
2. TravelPal shows the **Edit Day** page with fields to enter
3. User edits information in the specified **Day**
4. User submits and confirms the edit

> Use case ends

Extensions

3a. User enters an invalid field

3a1. TravelPal shows an error message

3a2. TravelPal does not edit invalid field

Use case continues at step 2

3b. User requests to list of **Events** in the trip

3b1. TravelPal shows a list of **Events** to the user (can be empty)

3b2. User chooses to *add/edit/delete (UC 7/8/9)* ***Event***

Use case continues at step 4

4b. User leaves necessary information empty

4a1. TravelPal shows an error message

4a2. TravelPal does not submit the details and does not confirm the edit

4a3. User enters new data

Steps 4a1-4a3 are repeated until the data entered are correct

Use case ends.

User case: UC6 – Delete Day

MSS

1. User requests to delete a specific **Day** in the list

2. TravelPal deletes the **Day**

> Use case ends

Extensions

2a. The list is empty

Use case ends

3a. The **Name** provided is invalid

3a1. TravelPal shows an error message

3a2. TravelPal does not delete any **Day**

Use case ends

User case: UC7 – Add Event

MSS

1. User chooses to add a **Event** to a specified **Day**
2. User *edits the event (UC5)*
3. TravelPal saves the **Event**

Extensions

3a **Event** added clashes with other **Events** in the **Day**

3a1. TravelPal shows an error message

3a2. TravelPal does not discard information the user has provided

3a3. TravelPal displays the **Edit Event** page containing the user's input

3a4. TravelPal requests the user to change the date of the **Event**

Steps 3a1 – 3a4 are repeated until the user provided non clashing date

User case UC8 – Edit Event

MSS

1. User requests to edit specific **Day**
2. TravelPal shows the **Edit Day** page with fields to enter
3. User edits information in the specified **Day**
4. User submits and confirms the edit

> Use case ends

Extensions

3a. User enters an invalid field

3a1. TravelPal shows an error message

3a2. TravelPal does not edit invalid field

Use case continues at step 2

3b. User requests to list of **Events** in the **trip**

3b1. TravelPal shows a list of **Events** to the user (can be empty)

3b2. User chooses to *add/edit/delete (UC 7/8/9)* ***Event*** Use case continues at step 4

4b. User leaves necessary information empty

4a1. TravelPal shows an error message

4a2. TravelPal does not submit the details and does not confirm the edit

4a3. User enters new data

Steps 4a1-4a3 are repeated until the data entered are non empty

Use case ends.

User case UC9 – Delete Event

MSS

1. User requests to delete a specific **Event** in the list
2. TravelPal deletes the **Event**

> Use case ends

Extensions

2a. The list is empty

Use case ends

3a. The **Name** provided is invalid

3a1. TravelPal shows an error message

3a2. TravelPal does not delete any **Event**

Use case ends

Appendix D: Non Functional Requirements

1. Should work on any [mainstream OS] as long as it has Java 11 or above installed.
2. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.
3. Should be able to hold up to 30 trips without a noticeable sluggishness in performance for typical usage.
4. A user familiar with travelling should be able to navigate the app easily
5. A novice user should be able to navigate without prior experience
6. Application does not depend on online resources to operate
7. Products is not required to make decisions for the user

Appendix E: Glossary

TravelPal – Our cross-platform desktop application for those who love to plan and micromanage their travels

CLI – Command Line Interface. CLI is a command line program that accepts text input to execute operating system functions.

GUI – Graphical User Interface. The graphical user interface is a form of user interface that allows users to interact

OS - An operating system, or "OS," is software that communicates with the hardware and allows other programs to run

Mainstream OS - Windows, Linux, Unix, 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 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 … }

F.2. Deleting a person

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

F.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 … }