Insurelytics - Developer Guide

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

1. Setting up

Refer to the guide here.

2. Design

2.1. Architecture

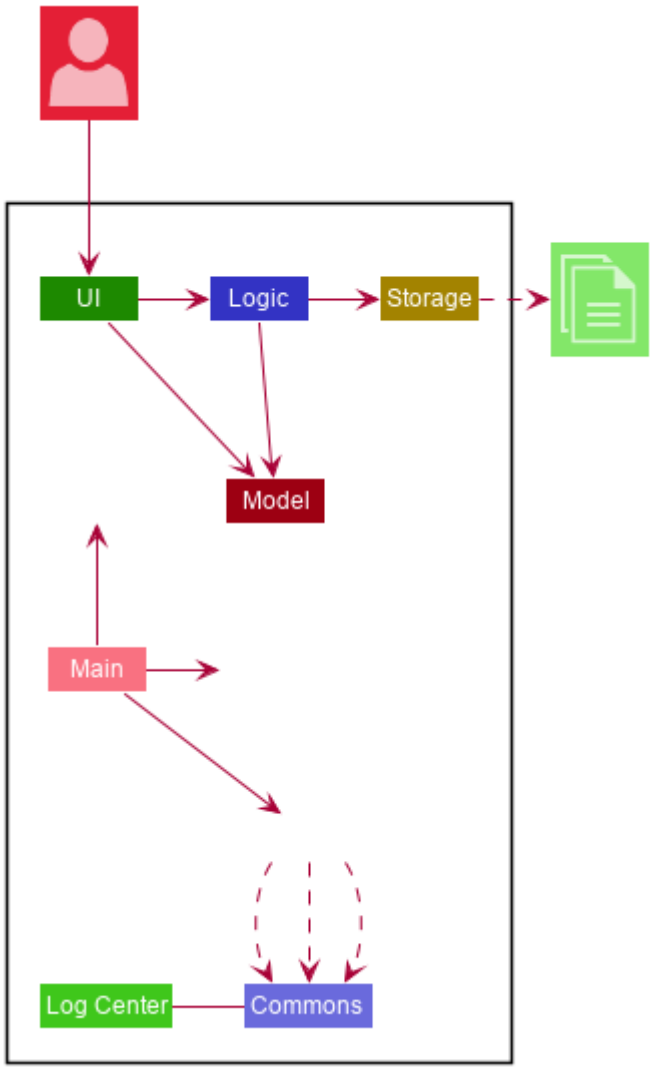


Figure 1. Architecture Diagram

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.

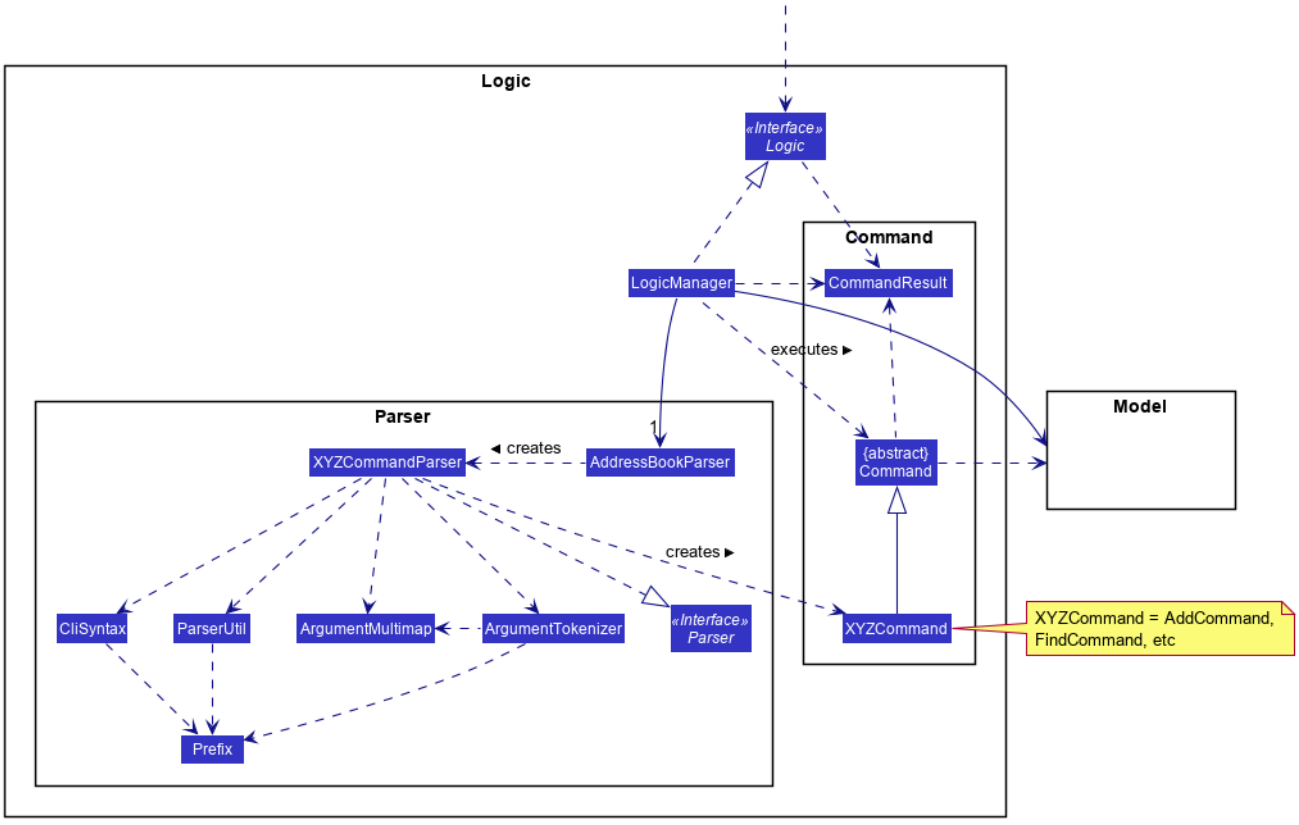


Figure 2. Class Diagram of the Logic Component

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`.

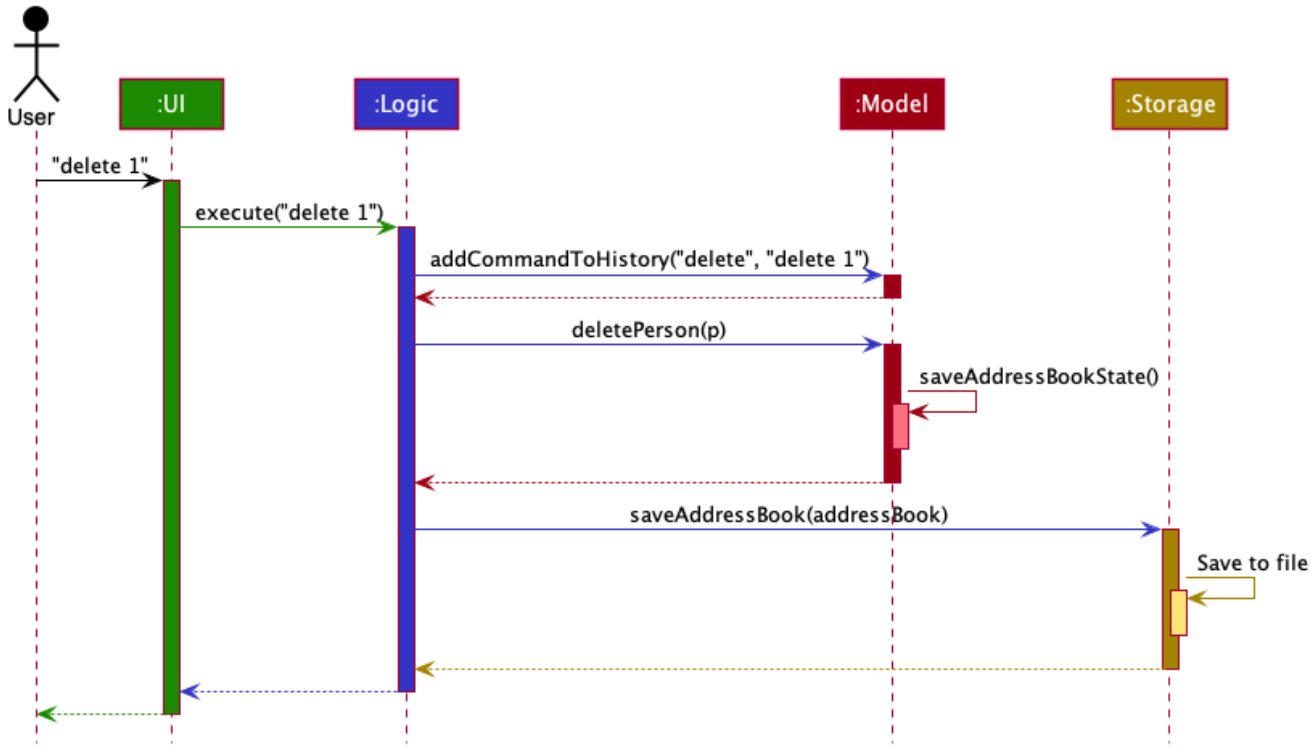


Figure 3. Component interactions for `delete 1` command

The sections below give more details of each component.

2.2. UI component

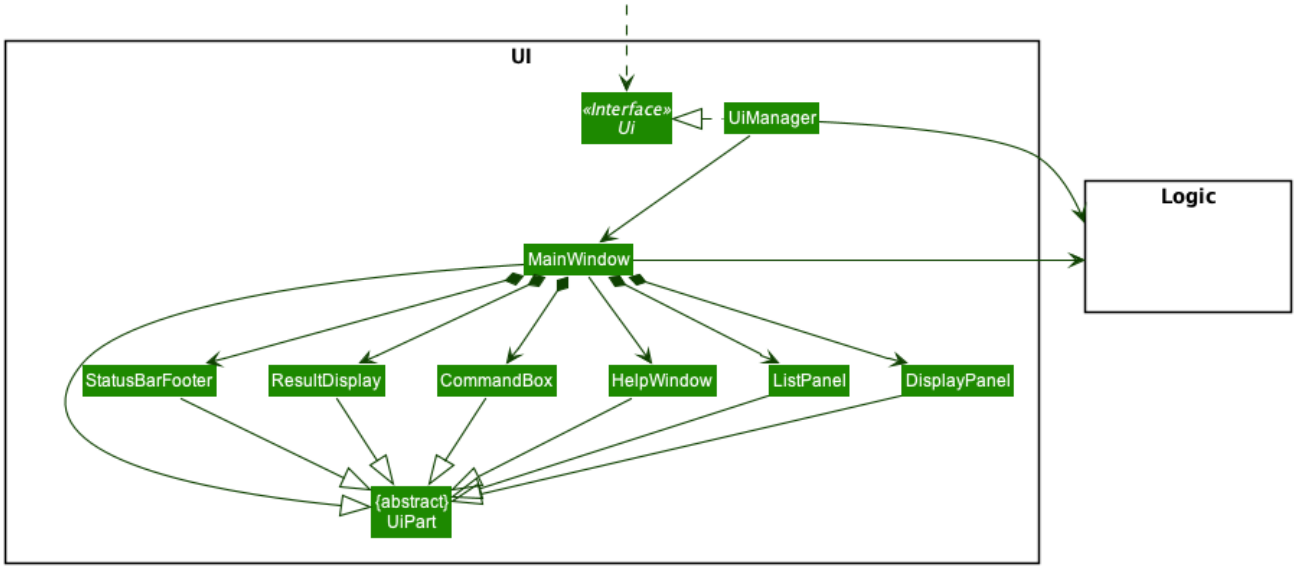


Figure 4. Structure of the UI Component

API : `Ui.java`

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

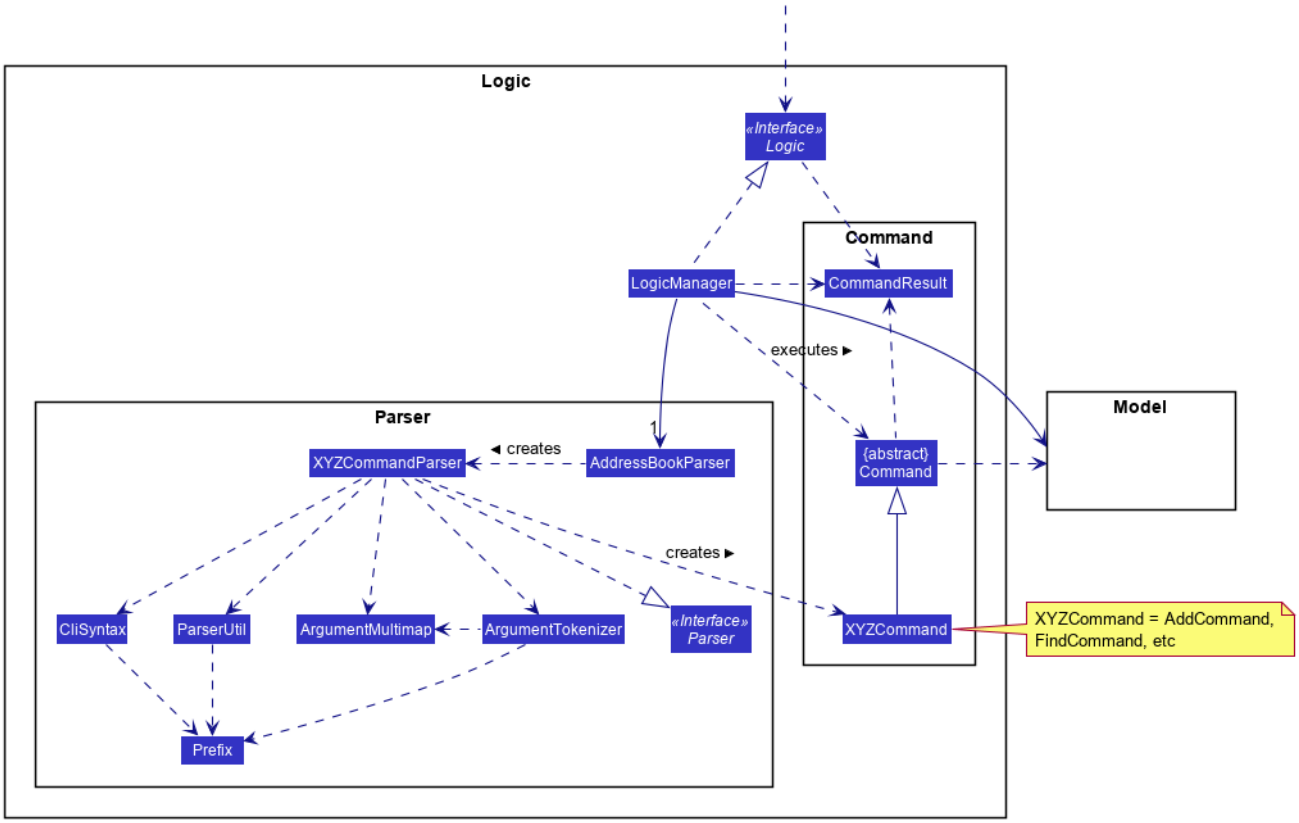


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `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.

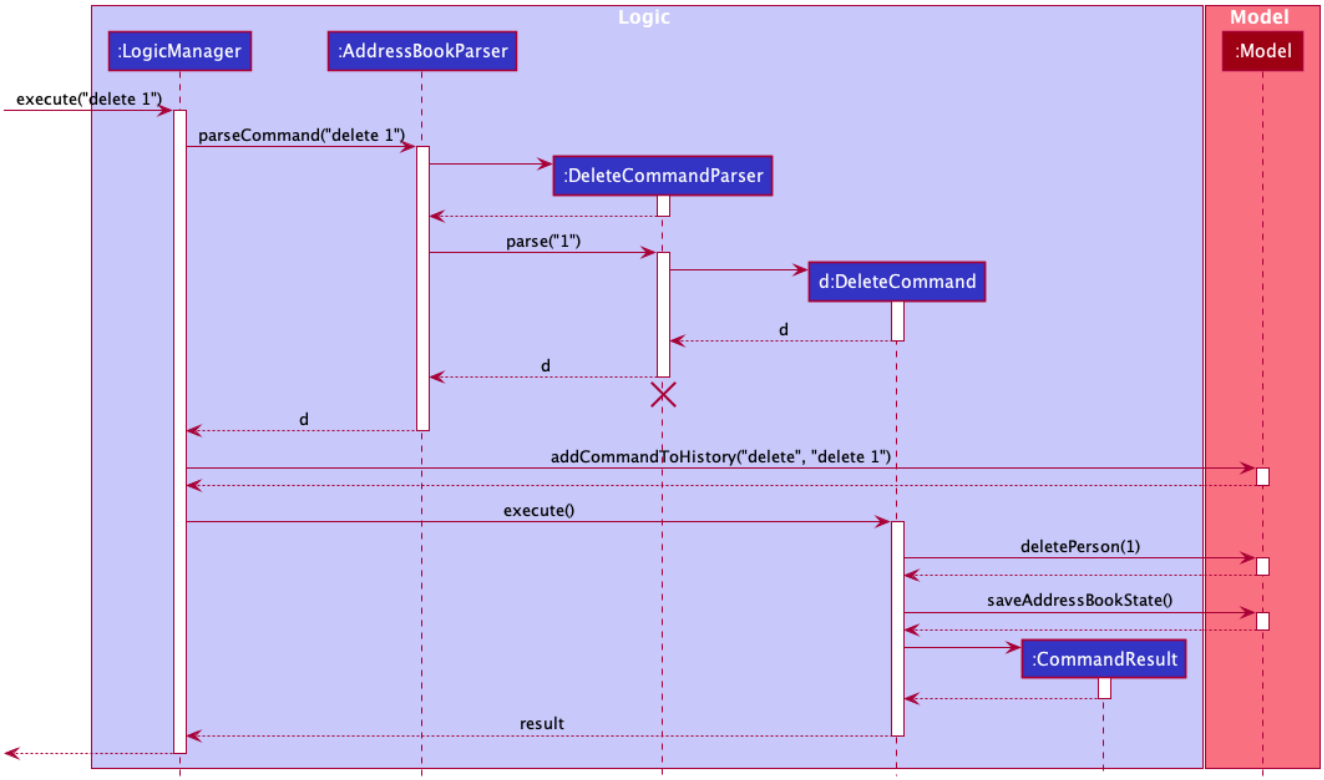


Figure 6. Interactions Inside the Logic Component for the `delete 1` Command

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

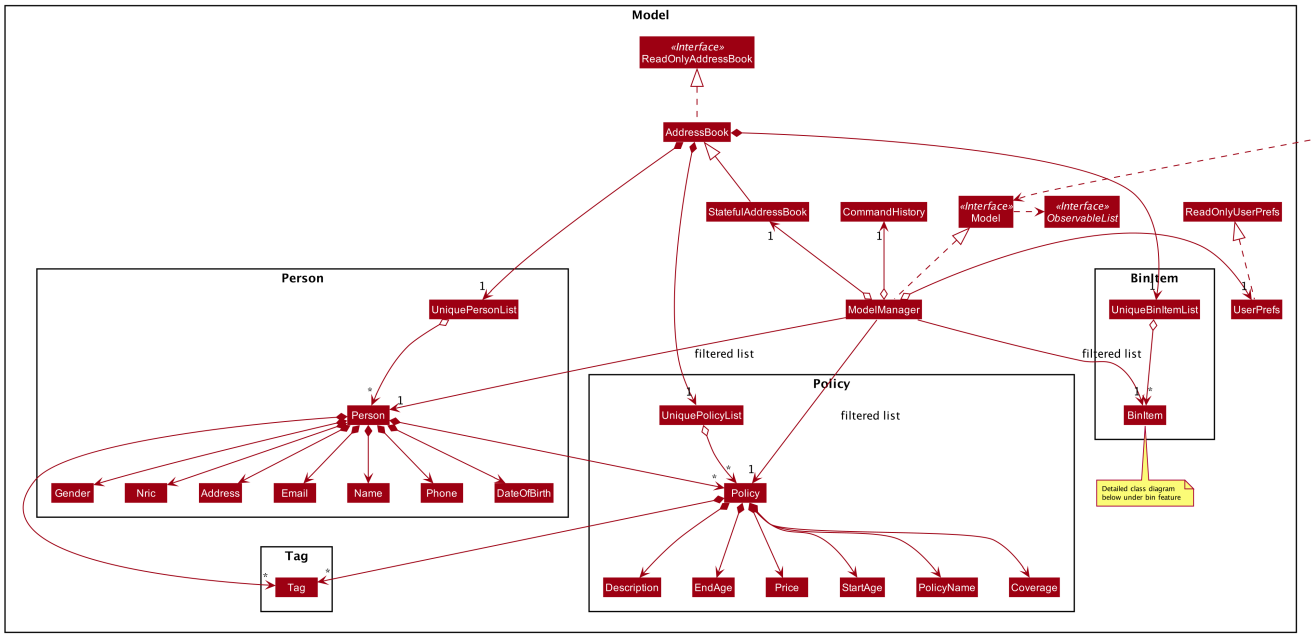


Figure 7. Structure of the Model Component

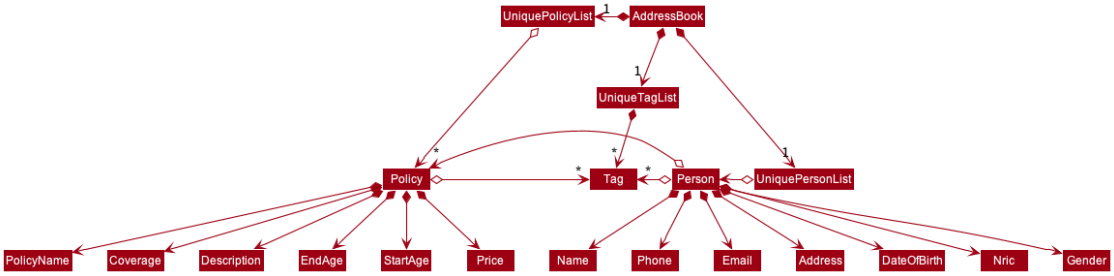
API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the Address Book data, in the form of a stateful address book which maintains a list of address books as states.
- stores the list of previously entered commands, in the form of a `CommandHistory` object.
- exposes an unmodifiable `ObservableList<Person>` and `ObservableList<Policy>` 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 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

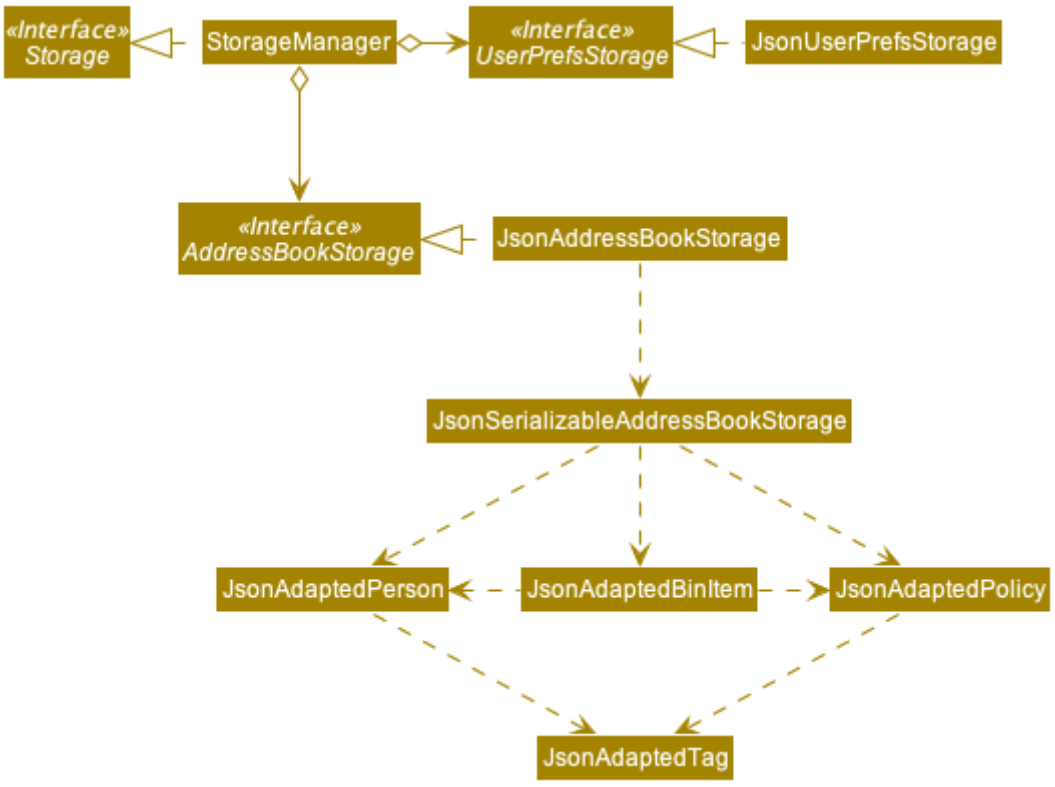


Figure 8. Structure of the Storage Component

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. Merging Feature

3.1.1. Implementation

The merging mechanism is facilitated by abstract classes `MergeCommand`, `DoNotMergeCommand`, `MergeConfirmedCommand` and `MergeRejectedCommand` and their child classes, which implement the merging of profiles and policies respectively. These classes extend `Command`. The child classes of `MergeCommand` are `MergePersonCommand` and `MergePolicyCommand`. A `MergePersonCommand` object will store the `Person` created by the input and the corresponding `Person` that is stored in the model. Additionally, the main crucial operations implemented by this class are:

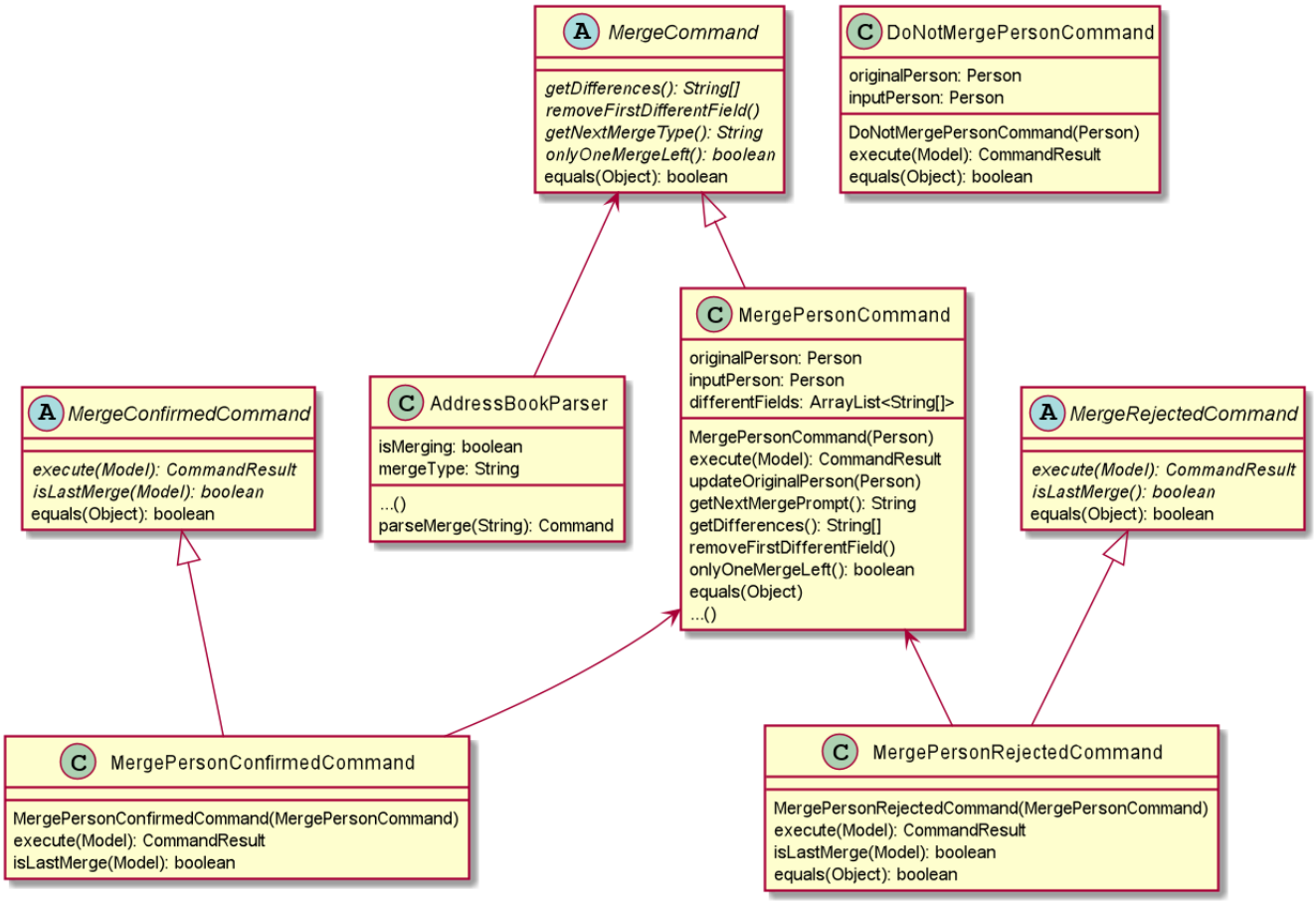
- `MergeCommand#getDifferences()` — Finds all the different fields between the input person and the original person.
- `MergeCommand#removeFirstDifferentField()` — Removes the first different field in the list of differences. This method is called after a merge decision has been input by the user and executed.
- `MergeCommand#getNextMergeFieldType()` — Returns the type of the field for the next merge.
- `MergeCommand#onlyOneMergeLeft()` — Checks whether there is only one merge left.

The implementation of `MergePolicyCommand` is similar.

The child classes of `MergeConfirmedCommand` are `MergePersonConfirmedCommand` and `MergePolicyConfirmedCommand`, while the child classes of `MergeRejectedCommand` are `MergePersonRejectedCommand` and `MergePolicyRejectedCommand`. They all implement `#execute(Model)`. Additionally, these classes implement an `#isLastMerge()` command to indicate if this is the last possible merge for the entity being merged.

AddressBookParser stores a boolean flag to indicate whether a merge is currently taking place. When it is set as true, all other commands will not be parsed and will be treated as invalid commands. The AddressBookParser object also stores the MergeCommand object during a merge process. This object is then used by `MergeConfirmedCommand` objects and `MergeRejectedCommand` objects

in their execution.



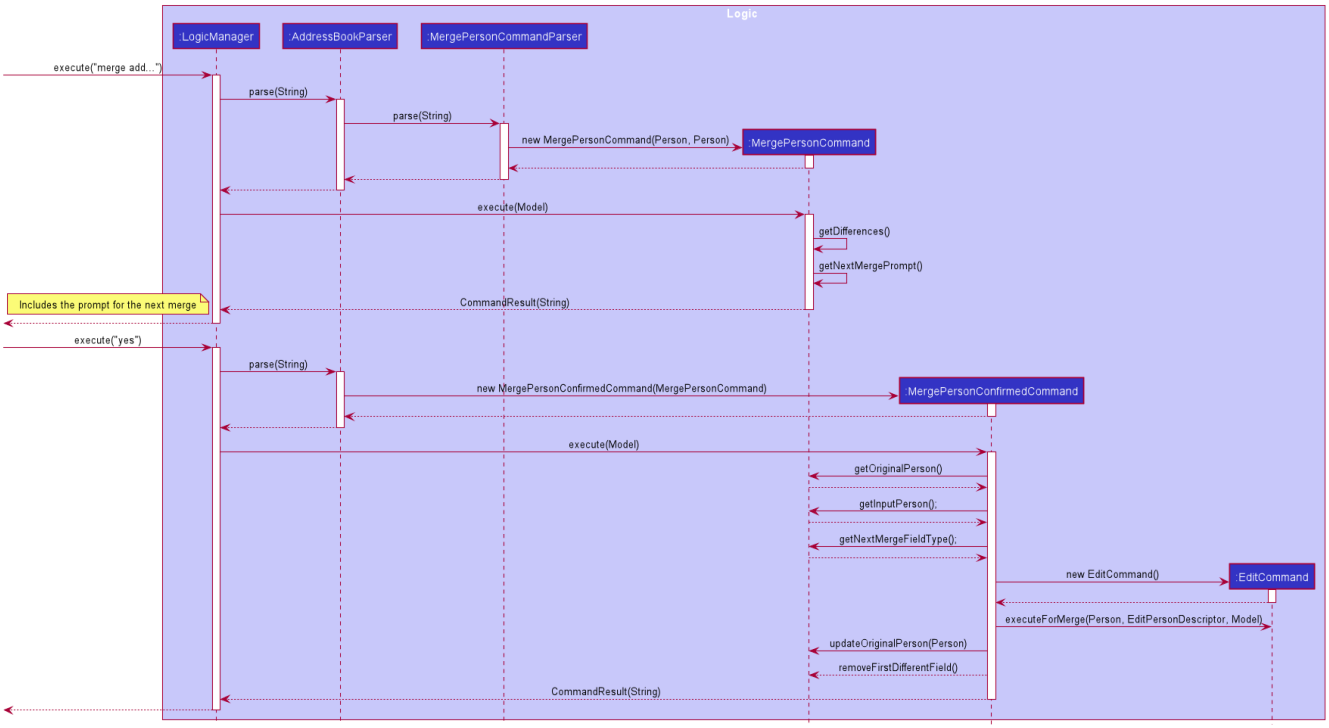
Given below is an example usage scenario and how the merge mechanism behaves at each step.

Step 1. The user adds a duplicate profile. The `AddCommand` will throw a `DuplicatePersonWithMergeException` during its execution. This exception is thrown if there is at least one different field between the input person and the original person stored in the model. Else, a `DuplicatePersonWithoutMergeException` will be thrown. The `DuplicatePersonWithMergeException` will finally be caught in the `CommandBox`. UI outputs the error message and a prompt to start a merge. `CommandBox` then constructs two command strings: one to proceed with the merge and one to reject the merge. This is done via `#standByForMerge(String, String)`. This string is then stored.

Step 2. The user inputs yes or presses enter to proceed with the merge. `CommandBox` then calls `CommandExecutor#execute()` to execute the merge command it constructed previously. When the command is being parsed in the `AddressBookParser` object, a new `MergeCommand` object is created and stored. The `isMerging` flag is also set to true. The execution of this command then returns a CommandResult that prompts the next merge.

Step 3. The user inputs yes or presses enter to update the field that was displayed in the prompt. The `AddressBookParser` parses the input and creates a new `MergePersonConfirmedCommand` object. The `MergePersonConfirmedCommand` object obtains information for the merge from the `MergeCommand` object that was passed in as a parameter in the constructor. In the execution, a new `EditCommand` is created and `EditCommand#executeForMerge()` is used to update the person in the model. If the `MergePersonConfirmedCommand#isLastMerge` returns false, `MergeCommand#removeFirstDifferentField` is called and the command result then shows a success message and the next prompt.

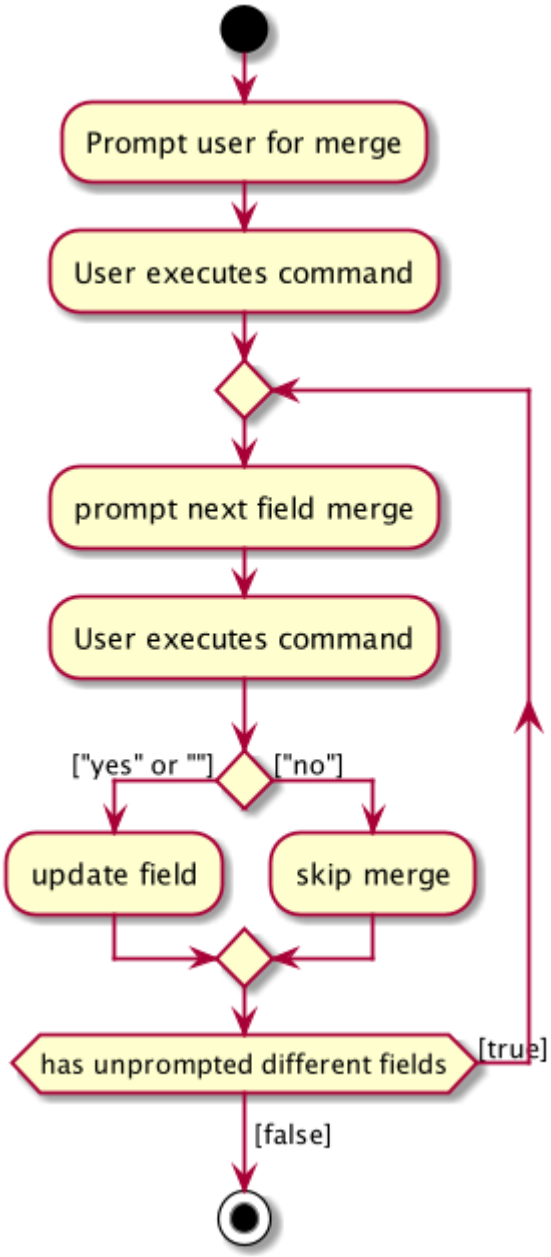
This process is shown in the sequence diagram below.



NOTE	If the user inputs an invalid command, the prompt will be displayed again along with an error message.

Step 4. The user inputs no to reject the update of the field that was displayed in the prompt. The input gets parsed in the `AddressBookParser` object and creates a new MergePersonRejectedCommand. If it is not the last merge, `MergeCommand#removeFirstDifferentField` is called. The command result then shows the next prompt. Else, it will show a success message of successfully updating the profile.

This is repeated until all merges have been prompted.



3.1.2. Design Considerations

Aspect: How merge command executes

- **Alternative 1 (current choice):** Stores the `MergeCommand` object in the `AddressBookParser` to be accessed by `MergeConfirmedCommand` and `MergeRejectedCommand` objects.
  - Pros: Finding of different fields is only executed once and can be used by future commands.
  - Cons: More coupling between `MergeCommand` and other classes.
- **Alternative 2:** Update the field in the command string and pass it on in the command result.
  - Pros: Less coupling between `MergeCommand` and other classes.
  - Cons:
    - User has to see the updated command (information that user does not need to see is displayed).
    - Command still has to be stored somewhere to be accessed by other future merge

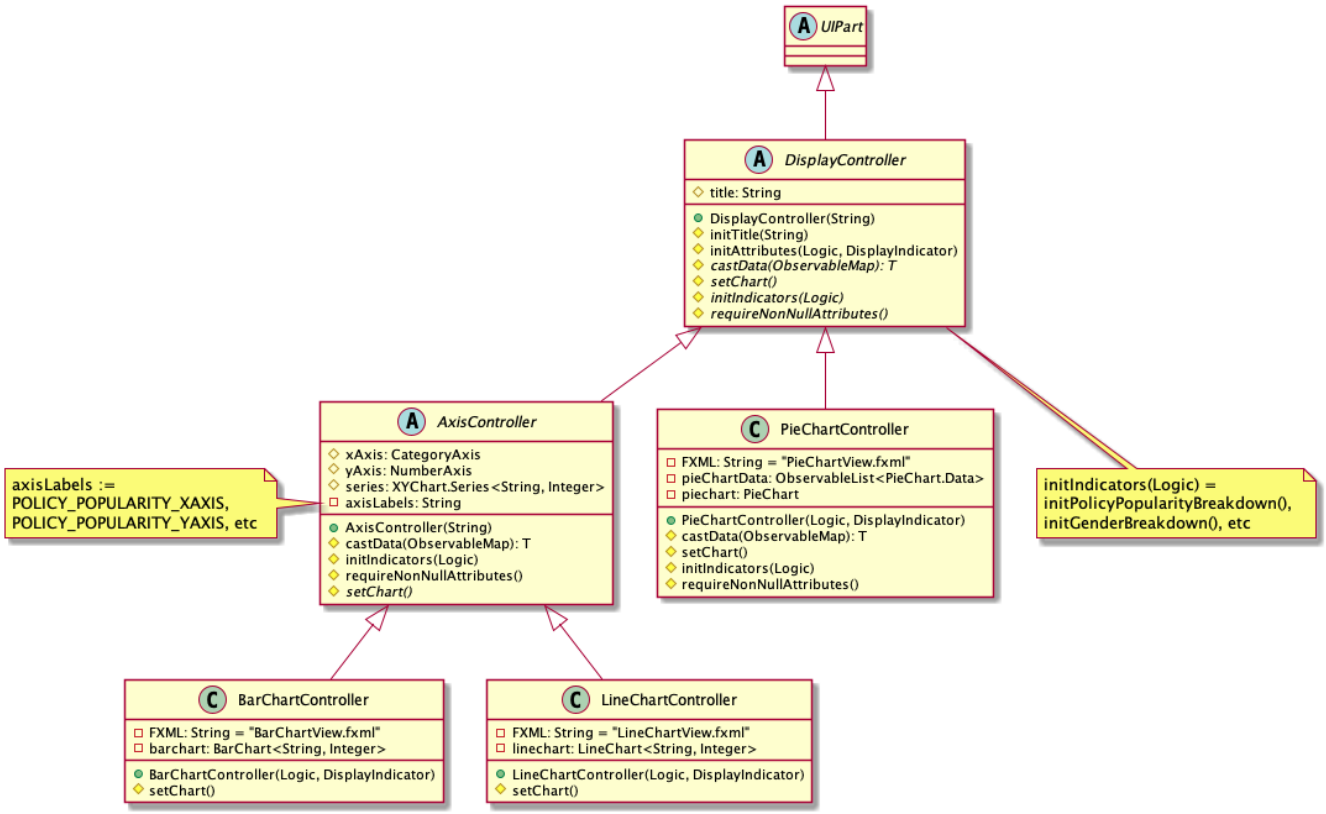
commands.

3.2. Display Feature

3.2.1. Implementation

The `display` mechanism follows the Model-View-Controller design pattern. The model is facilitated by the `AddressBook` instance, which provides the data the controller needs. The controller is facilitated by an abstract class `DisplayController`, which extends `UIPart<Region>`. Every supported format controller extends this abstract class.

The following class diagram shows the OOP solution for `display`:



NOTE Every controller needs to support every indicator. In the event a controller cannot display a particular indicator, it will throw a `parseException` error, which provides suggestions of which visual controllers are supported by the particular indicator.

The view is facilitated by the associated FXML. These views share a common CSS, and also have their individual CSS file.

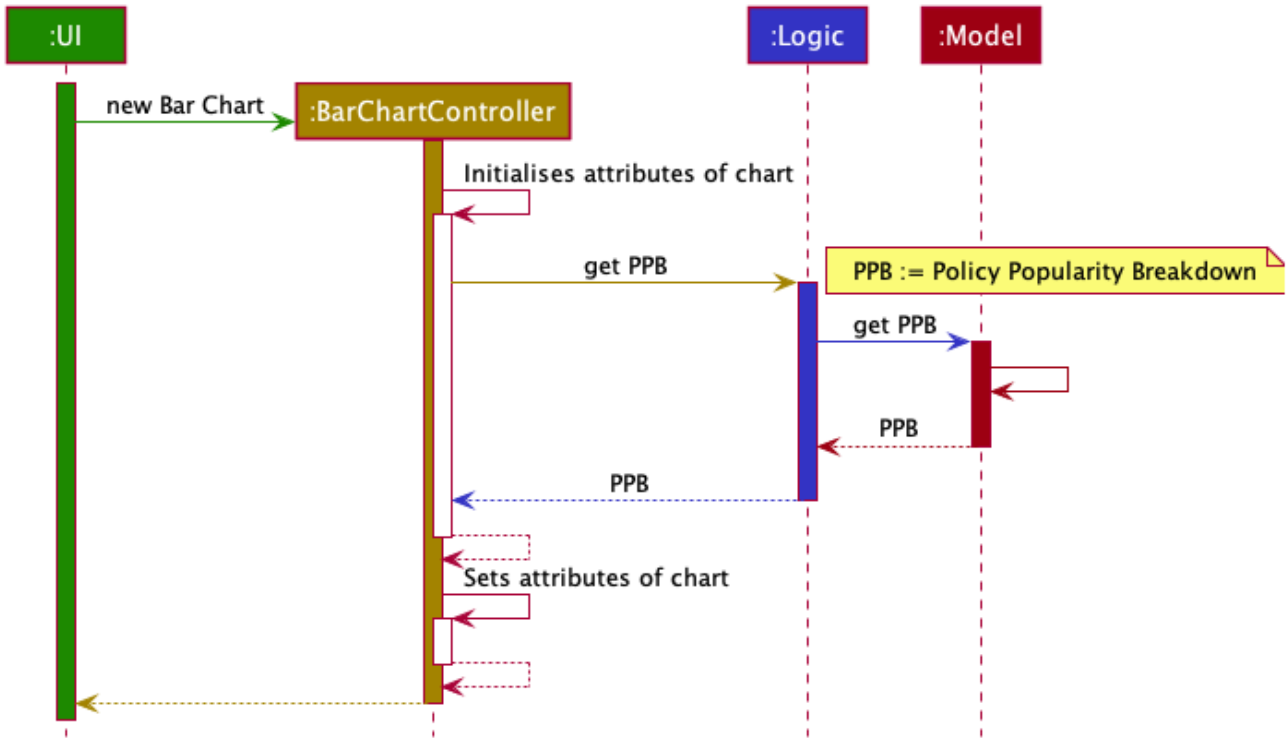
Given below is an example usage scenario and how the display mechanism behaves at each step.

Step 1. The user executes the `display i/policy-popularity-breakdown f/barchart` command to display the policy popularity breakdown indicator in bar chart format. The execution of a `display` command determines what will be shown (`displayIndicator`) and how it will be shown (`displayFormat`).

Step 2. `displayFormat` specifies that the controller `BarChartController` will be instantiated.

Step 3. The `BarChartController` initialises all the attributes of its associated FXML in its construction. Let us take a closer look at the initialisation of the `series` attribute. The controller utilises the display indicator `policy-popularity-breakdown` to retrieve the data in the model it needs. The controller then casts the model's data type to the data type supported by bar charts. The result is assigned to `series` attribute.

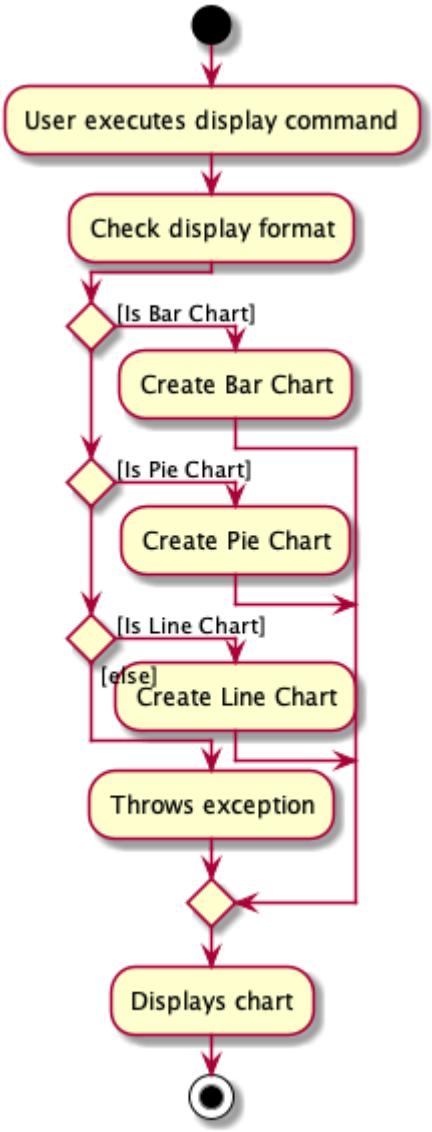
The following sequence diagram shows the interaction between UI, Controller and Model for steps 2 and 3:



Step 4. The bar chart controller then sets all the attributes of its associated FXML.

Step 5. Finally, the `MainWindow` calls `DisplayController#getRoot()` and displays the view.

The following activity diagram summarizes what happens when a user executes the display command:



3.2.2. Design Considerations

Aspect: How should controllers interact with model

- **Alternative 1 (current choice):** Within controllers (by passing `logic`, which accesses the model, as an argument to the instantiation of a controller.)
  - Pros: Every controller handles their own interaction with the model.
  - Cons: Inconsistent with current implementation (alternative 2).
- **Alternative 2:** Within `MainWindow`
  - Pros: Consistent with current implementation
  - Cons: The controllers are fully dependent on `MainWindow` for the data from the model. This entails that

Aspect: OOP solution for visual controllers

- **Alternative 1 (current choice):** Display controllers to extend from abstract class `DisplayController`
  - Pros:

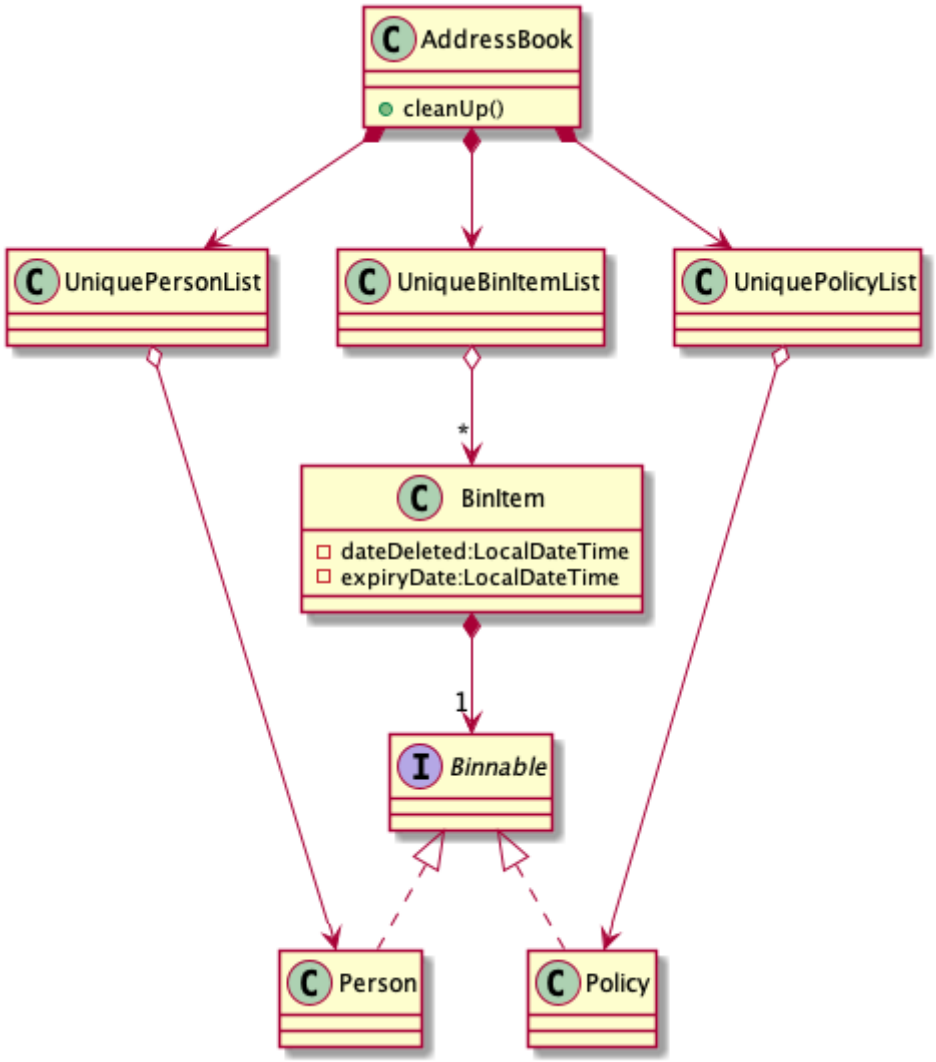
        - Allows display controller subclasses to share code. (refer to class diagram above)
        - Subclasses have many common methods (`initIndicators(Logic)`)
- **Alternative 2:** Display controllers to implement interface `DisplayController`
    - Pros: Satisfies the can-do relationship of an interface.
    - Cons: Restricted to public access modifiers. This violates Law of Demeter.

3.3. Bin Feature

3.3.1. Implementation

The bin feature is facilitated by `BinItem`, `UniqueBinItemList` classes and the interface `Binnable`. Objects that can be "binned" will implement the interface `Binnable`. When a `Binnable` object is deleted, it is wrapped in a wrapper class `BinItem` and is moved into `UniqueBinItemList`.

The follow class diagram shows how bin is implemented.



`BinItem` has 2 key attributes that is wrapped on top of the `Binnable` object, namely: `dateDeleted` and `expiryDate`. Objects in the bin stays there for 30 days, before it is automatically deleted forever. Both attributes are used in the auto-deletion mechanism of objects in the bin.

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

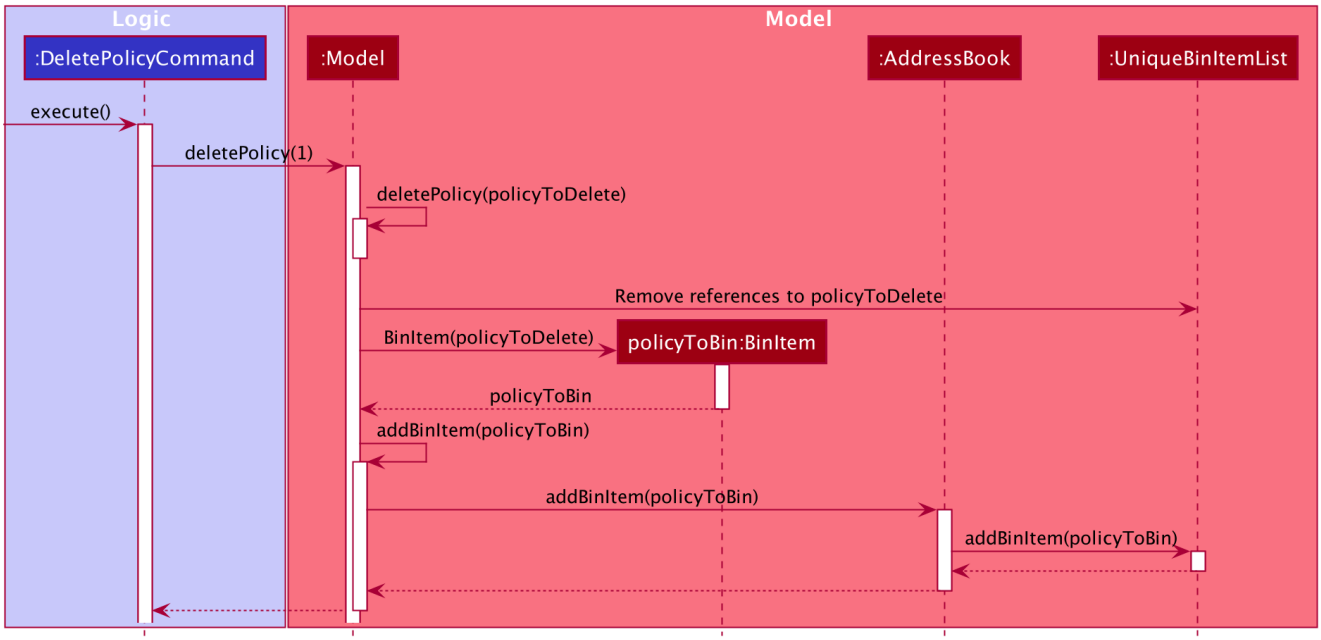
Step 1. When the user launches **Insurelytics**, `ModelManager` will run `ModelManager#binCleanUp()`, which will check the `expiryDate` of all objects in `UniqueBinItemList` against the system clock. If the system clock exceeds `expiryDate`, `UniqueBinItemList#remove()` is called and deletes the expired object forever.

Step 2. The user executes `deletepolicy 1` command to delete the first policy in the address book. The `deletepolicy` command calls the constructor of `BinItem` with the deleted policy to create a new `BinItem` object. At this juncture, the attribute `dateDeleted` is created, and `expiryDate` is generated by adding `TIME_TO_LIVE` to `dateDeleted`. At the same time, references to the policy that was just deleted will also be removed from any `BinItem` that has them.

NOTE	Removing references of deleted policies in items inside the bin only happens for `deletepolicy`. Removing of references does not happen for deleted persons, since policies don't keep track of the persons that bought them.

Step 3. The `deletepolicy` command then calls `Model#addBinItem(policyToBin)` and shifts the newly created `BinItem` to `UniqueBinItemList`.

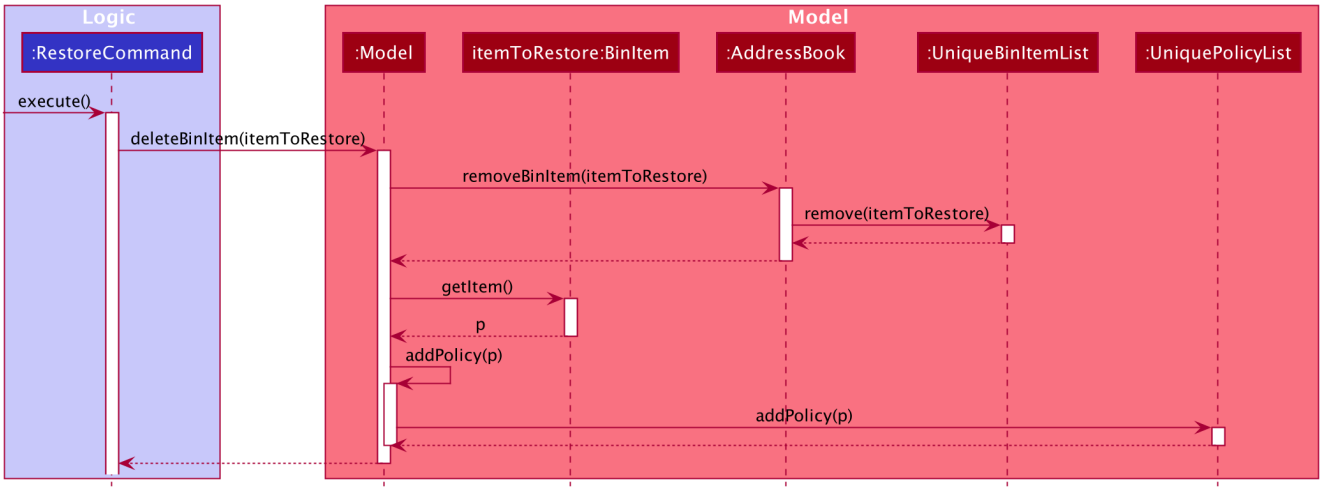
The following sequence diagram shows how a `deletepolicy` operation involves the bin.



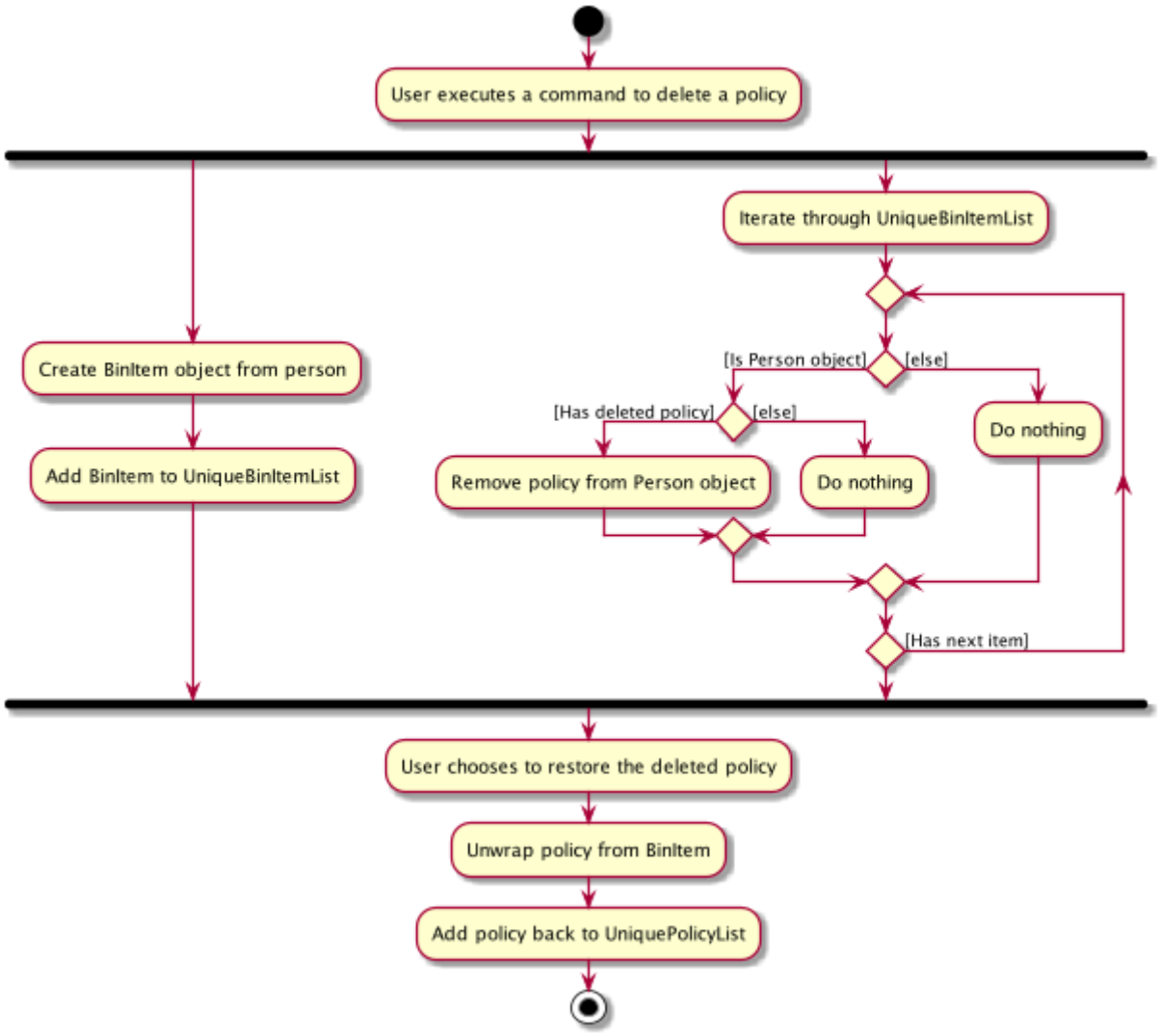
Step 4. The user quits the current session and starts a new session some time later. He/she then realises that he/she needs the person that was deleted and wants it back, so he/she executes `restore 1` to restore the deleted person from the bin.

Step 5. The `restore` command then calls `Model#deleteBinItem(itemToRestore)`, which removes `itemToRestore` from `UniqueBinItemList`. The wrapper class `BinItem` is then stripped and the internal policy item is added back to `UniquePolicyList`.

The following sequence diagram shows how a restore command operates.



The following activity diagram summarizes the steps above.



3.3.2. Design Considerations

Aspect: Which part of the architecture does Bin belong

- **Alternative 1 (current choice):** As part of AddressBook
  - Pros: Lesser repeated code and unnecessary refactoring. Other features at the AddressBook

level such as undo/redo will not be affected with a change/modification made to Bin as it is not dependent on them.

  - Cons: From a OOP design point of view, this is not the most direct way of structuring the program.

- **Alternative 2:** Just like AddressBook, as part of Model
  - Pros: More OOP like and lesser dependencies since Bin is extracted out from AddressBook. Methods related to bin operations are called only from within Bin.
  - Cons: Many sections with repeated code since it is structurally similar to AddressBook.

3.4. Command History Feature

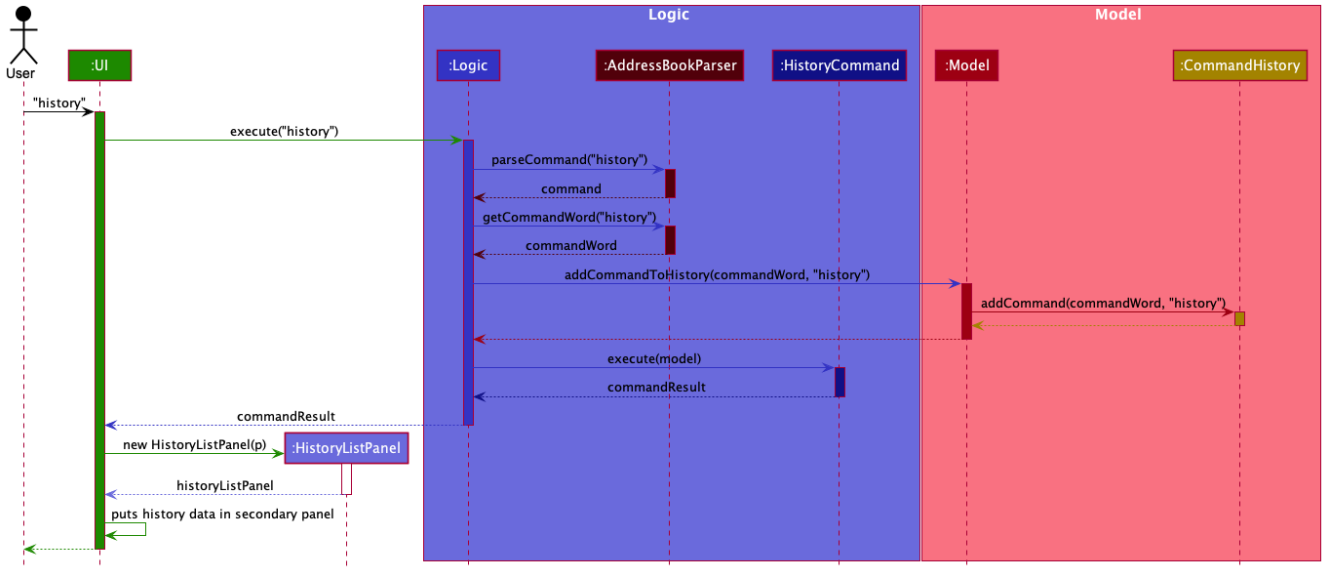
3.4.1. Implementation

To allow users to view the list of previously entered commands, a command history mechanism is implemented, which lists down all the previous (valid) commands entered by the user from the point of starting the application. Commands entered while merging are not added. The feature is supported by the `CommandHistory` class, an instance of which is stored as one of the memory objects inside `ModelManager`. The main operations implemented by this class are:

- `addCommand(commandWord, commandText)` — Adds the command with the command word `commandWord` and full command text `commandText` into the list of previously entered commands `userInputHistory`.
- `getHistory()` — Reverses the list of previously entered commands and returns the list.

The history view is accompanied by its associated `HistoryCard` and `HistoryListPanel` FXML. These views share a common CSS with other FXML files.

Following is the sequence diagram that shows the interaction between different components of the app when the command `history` is typed in by the user:



3.4.2. Design Considerations

Aspect: Where to store the CommandHistory object

- **Alternative 1:** Place the `CommandHistory` object directly in the `LogicManager` class
  - Pros: Since the user input is being parsed in `LogicManager`, storing the command history ensures that data is not being passed around between the `LogicManager` and the `ModelManager`.
  - Cons: By right the `LogicManager` is only concerned with handling logic, any data storage should be performed in `ModelManager`. Storing command history here would violate Single Responsibility Principle and Separation of Concerns.
- **Alternative 2 (current choice):** Place the `CommandHistory` object in model manager, parse user input in `LogicManager` and pass it to `ModelManager`.
  - Pros: Separation of concerns and Single Responsibility Principle for `LogicManager` and `ModelManager` is maintained.
  - Cons: Increases coupling between `LogicManager` and `ModelManager`, thereby violating the Law of Demeter.

Aspect: Which commands to display in command history

- **Alternative 1:** Display only those commands which will be relevant to user if she is considering undo/redo (therefore, display only data changes)
  - Pros: Only commands which are more relevant shown to user.
  - Cons: Which commands are relevant depends on the user, moreover limits the application of command history feature to undo/redo application.
- **Alternative 2:** Display every valid command entered by the user
  - Pros: More accurate representation of command history, every command entered is a part of the history
  - Cons: Clutters the command history with unnecessary commands like `history` and `undo`.

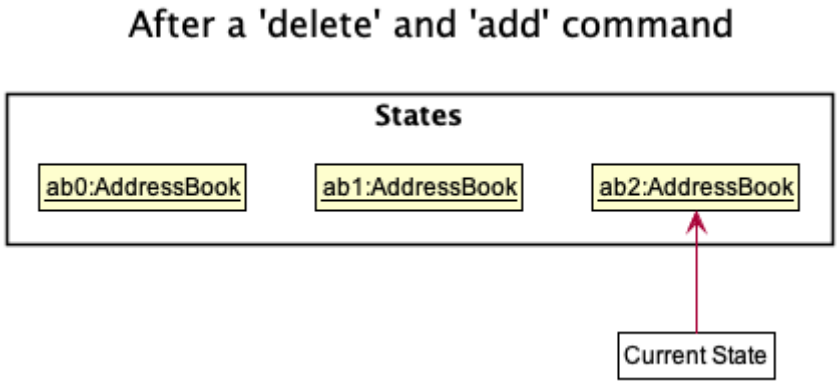
3.5. Undo/Redo Feature

3.5.1. Implementation

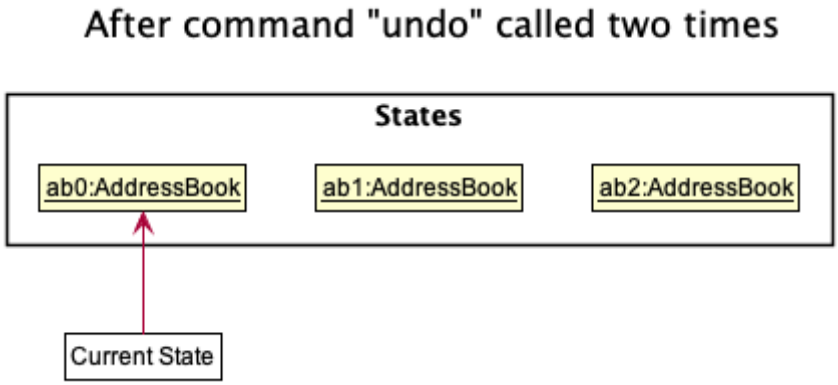
To allow users to revert/return to a previous/future state of the address book, an undo/redo mechanism is implemented, which undoes/redoes the last data change made in the application. The feature is supported by the `StatefulAddressBook` class, an instance of which is stored as one of the memory objects inside `ModelManager`.

Given below is an example usage scenario and how the undo-redo mechanism works at each step. Let us assume that `StatefulAddressBook` has just been initialised.

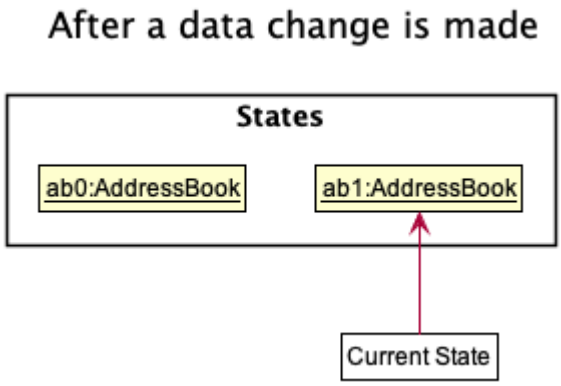
Step 1. The user makes some data changes. This adds a list of states to our `StatefulAddressBook`, and updates the `currentStatePointer`.



Step 2. User types `undo`. This invokes the `UndoCommand::execute` method, which further invokes the `StatefulAddressBook::undo` method. The `currentStatePointer` is decremented, and the address book is reset to the one being pointed to by `currentStatePointer`.

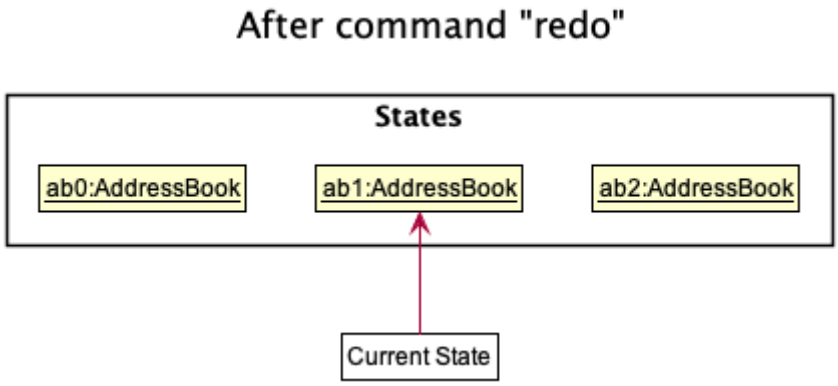


Step 3a. The use can perform another data change, following which states after `currentStatePointer` are erased, and a new state is added.



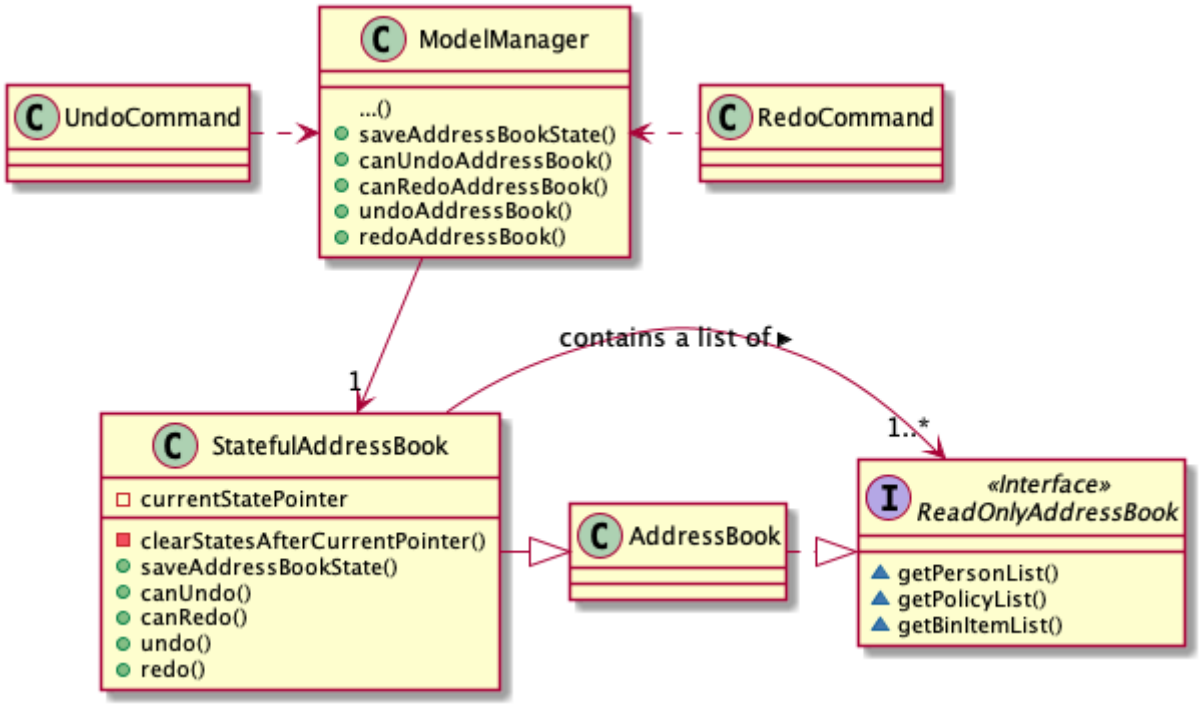
Step 3b. If a command which does not perform a data change is called, nothing happens in the `StatefulAddressBook`, and it stays the same as seen in `Step 2.`

Step 3c. If a `redo()` is called, then the `currentStatePointer` is incremented, and the address book is reset to the one being pointed to by `currentStatePointer`.

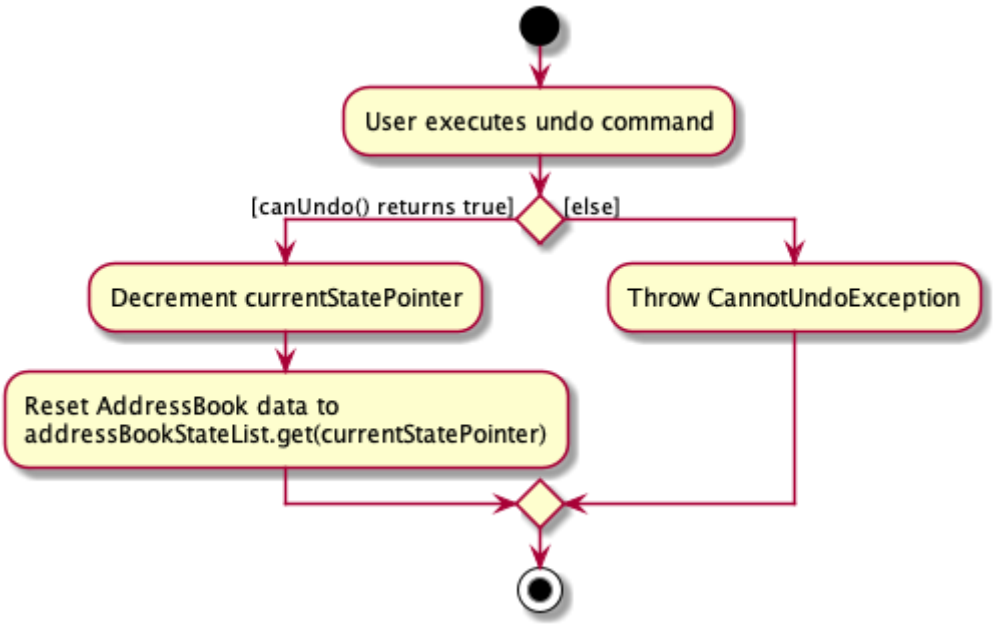


Step 4. If the address book data is reset to another state's, then all the data inside the application is reloaded, with the resulting changes now reflected.

Following is the class diagram for the undo/redo feature.



Following is the activity diagram. The diagram for a `redo` command will be similar.



3.5.2. Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire address book.
  - Pros: Easy to implement, takes way lesser code.
  - 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.
- **Alternative 3:** Do not store entire address book, but a PersonList, PolicyList and BinList depending on what is changed.
  - Pros: Will use less memory (e.g. for delete, just save the person list).
  - Cons: We must ensure that the implementation of each individual command are correct. Several cases to consider when we try to undo a command.

3.6. 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 3.7, “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

3.7. 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:

- is an insurance agent
- is always meeting new clients, so needs to manage a significant number of contacts
- needs to manage a significant number of insurance policies
- always offering insurance schemes to client base
- finds it easier to understand visual data
- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: manages large number of contacts and insurance policies 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…
* * *	new user	see usage instructions	refer to instructions when I forget how to use the App
* * *	user	add a new contact/policy	
* * *	user	delete a contact/policy	remove entries that I no longer need
* * *	insurance agent	find a person/policy by name	locate details of persons without having to go through the entire list
* * *	insurance agent	predefine a custom set of policies	so I can select policies in this predefined set and make data entry faster
* * *	insurance agent	tag or untag clients	so I can target people belonging to different target groups
* * *	insurance agent	generate basic statistics and data about my sales for the month	so I can visualise and keep track of them
* * *	busy user	quickly check what commands are available	so I do not have to remember all commands of the app

Priority	As a …	I want to …	So that I can…
* * *	user handling many profiles and contacts	can use mass operations to change the same details across different people/policies	so I can speed up my workflow
* * *	insurance agent with many clients	filter and sort people according to their policies and tags	so I can get information about a particular group of people quickly
* * *	user who prefers typing over using a mouse	interact with all aspects of GUI using commands only	
* * *	insurance agent	view the key KPIs of my address book	track my performance
* *	user with not a strong memory	view all commands as a dropdown list	
* *	insurance agent prone to making mistakes during manual entry	I want to be sure that the details of buyers are valid without having to manually check every record	
* *	insurance agent	start-up page to reflect key KPIs (e.g: sales in current quarter)	

Priority	As a …	I want to …	So that I can…
* *	insurance agent with many contacts	disallow creating duplicate profiles	so I need not worry about accidentally creating duplicate profiles
* *	clumsy insurance agent	retrieve deleted contacts from an archive of recently deleted contacts	
* *	insurance agent	want to see which policies a new contact is eligible for	so I can quickly check eligible policies while constantly adding new contacts
* *	clumsy user	I can undo and redo my previous commands	to amend mistakes made by entering incorrect commands
* *	insurance agent	I can export the data as an Excel document for easier sharing of data	so I can generate reports and and send these reports to authorities/other agents
* *	insurance agent	I can group families who are under the same insurance	so it is easier to sell/manage plans for these people
* *	insurance agent with new policies	I want to be able to filter people based on eligibility for these policies	for faster data entry

Priority	As a …	I want to …	So that I can…
* *	insurance agent	I can have details auto-filled into business/government forms	so I can save time keying in details I already have
* *	insurance agent	receive reminders for clients whose policies are almost due for renewal	so I can contact them to renew their insurance policy
* *	insurance agent	receive reminders when clients pass a certain age group	so I can contact them about the new policies that they are now eligible for.
* *	user	hide private contact details by default	minimize chance of someone else seeing them by accident
* *	insurance agent who prefers visualisation	view key performance indicators as diagrams	
*	insurance agent with many clients	configure automatic greeting emails to policyholders	so I can maintain a good relationship with clients without manually sending individual emails
*	user with personal preferences	configure the CLI	so I can speed up my workflow

Priority	As a …	I want to …	So that I can…
*	insurance agent with many clients	want to contact my policyholders with ease (such as email)	so I have a convenient method of communication
*	busy user	auto-complete my commands	so I can perform operations and find the data I need quickly
*	user with many persons in the address book	sort persons by name	locate a person easily

{More to be added}

Appendix C: Use Cases

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

Use case: Add person

Guarantees:

1. Person is added even if input fields might be invalid (see 1a).

MSS

1. User requests to add a person.
2. Insurelytics adds the person.

   Use case ends.

Extensions

1a. Either of the given NRIC, contact number, or email address is invalid.

  1a1. Insurelytics adds the person into address book.

  1a1. Insurelytics shows a warning.

1b. Duplicate profile is added.

  1b1. Insurelytics shows an error message and will attempt to merge the profile.

Use case: Edit person

MSS

1. User requests to list persons.
2. Insurelytics shows a list of persons.
3. User requests to edit a specific person in the list.
4. Insurelytics edits the person.
5. The person’s edited details are now visible in the address book.

   Use case ends.

Extensions

   2a. The list is empty.

   Use case ends.

   3a. The given index is invalid.

   3b. The given edited details are invalid.

      3*1. Insurelytics shows an error message.

      Use case resumes at step 2.

Use case: Delete person

MSS

1. User requests to list persons.
2. Insurelytics shows a list of persons.
3. User requests to delete a specific person in the list.
4. Insurelytics deletes the person.
5. Person appears in the recycling bin.

   Use case ends.

Extensions

   2a. The list is empty.

   Use case ends.

   3a. The given index is invalid.

      3a1. Insurelytics shows an error message.

Use case resumes at step 2.

Use case: Assigning a policy to a person

MSS

1. User requests to list persons.
2. Insurelytics shows a list of persons.
3. User requests to assign a policy to a specific person.
4. The policy gets assigned to the person.

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3b. The person is not eligible for the policy.

3c. The policy is not present in the global list of policies.

3*1. Insurelytics shows an error message.

Use case resumes at step 2.

Use case: Undoing a data change

MSS

1. User enters a command.
2. Insurelytics performs the entered command.
3. User requests to undo the previously entered command.
4. The previously entered command gets undone and the data is reverted back to the previous state.

   Use case ends.

Extensions

3a. The previously entered command did not perform a data change.

3a1. Insurelytics shows an error message.

Use case ends.

Use case: Restoring recently deleted items

MSS

1. User requests to list recently deleted items from bin.
2. Insurelytics shows a list of bin items.
3. User requests to restore a specific item in the list.
4. AddressBook restores the item to the list it belongs to.

   Use case ends.

Extensions

   2a. The list is empty.

   Use case ends.

   3a. The given index is invalid.

      3a1. Insurelytics shows an error message.

      Use case resumes at step 2.

Use case: Merging a duplicate person with different fields

MSS

1. User requests to add a person.
2. Insurelytics indicates that this person already exists and prompts a merge.
3. User indicates whether or not to edit this profile.
4. A different field is displayed and asks the user whether or not to update this field.
5. Steps 3 and 4 repeat until decisions whether or not to merge different fields have been completed.

   Use case ends.

Extensions

   *a. User indicates to stop the merging process.

      3a1. The user inputs an invalid command.

      3a2. The Insurelytics indicates an error and prompts the merge again.

      Use case resumes at 4.

{More to be added}

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be able to hold up to 1000 persons without a noticeable sluggishness in performance for typical usage.
3. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.
4. Should display visual representations as long as Excel is installed.

{More to be added}

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, macOS

Private contact detail

A contact detail that is not meant to be shared with others

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 persons and policies. 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 person is deleted from the list. Details of the deleted person 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 … }

F.4. Displaying indicators

1. Indicators are up to date and shown in the correct format
    a. Test case:
    `display i/policy-popularity-breakdown f/piechart`
    `addpolicy n/Child Care d/Care for children c/days/20 months/11 years/5 pr/$50000 sa/0 ea/10`
    Expected: Popularity of child care policy should be 0. Format of indicator should be piechart.
    b. Test case:
    `display i/age-group-breakdown f/linechart`
    `add n/Norman ic/S0000001J p/98765432 e/johnd@example.com a/311, Clementi Ave 2, #02-25 dob/12.12.2000 g/Male`
    Expected: Number of people in the age group below 20 years old should increase by 1. Format of indicator should be linechart.
    c. Test case:
    `display i/gender-breakdown f/barchart`
    `add n/Sally ic/S0000001J p/98765432 e/johnd@example.com a/311, Clementi Ave 2, #02-25 dob/12.12.2000 g/Female`
    Expected: Number of females should increase by 1. Format of indicator should be barchart.