ResuMe - Developer Guide

By: `CS2103T F10-1` Since: `Feb 2020` Licence: `MIT`

{ start of `introduction` section written by: Pham Thuy Duong }

1. Introduction

This developer guide documents the system design and implementation of **ResuMe**. We hope that it would be a useful reference for software developers who wish to contribute to the project, or to simply gain deeper insight into our development process and considerations.

{ end of `introduction` section written by: Pham Thuy Duong }

2. Setting up

Refer to the guide here.

3. Design

3.1. Architecture

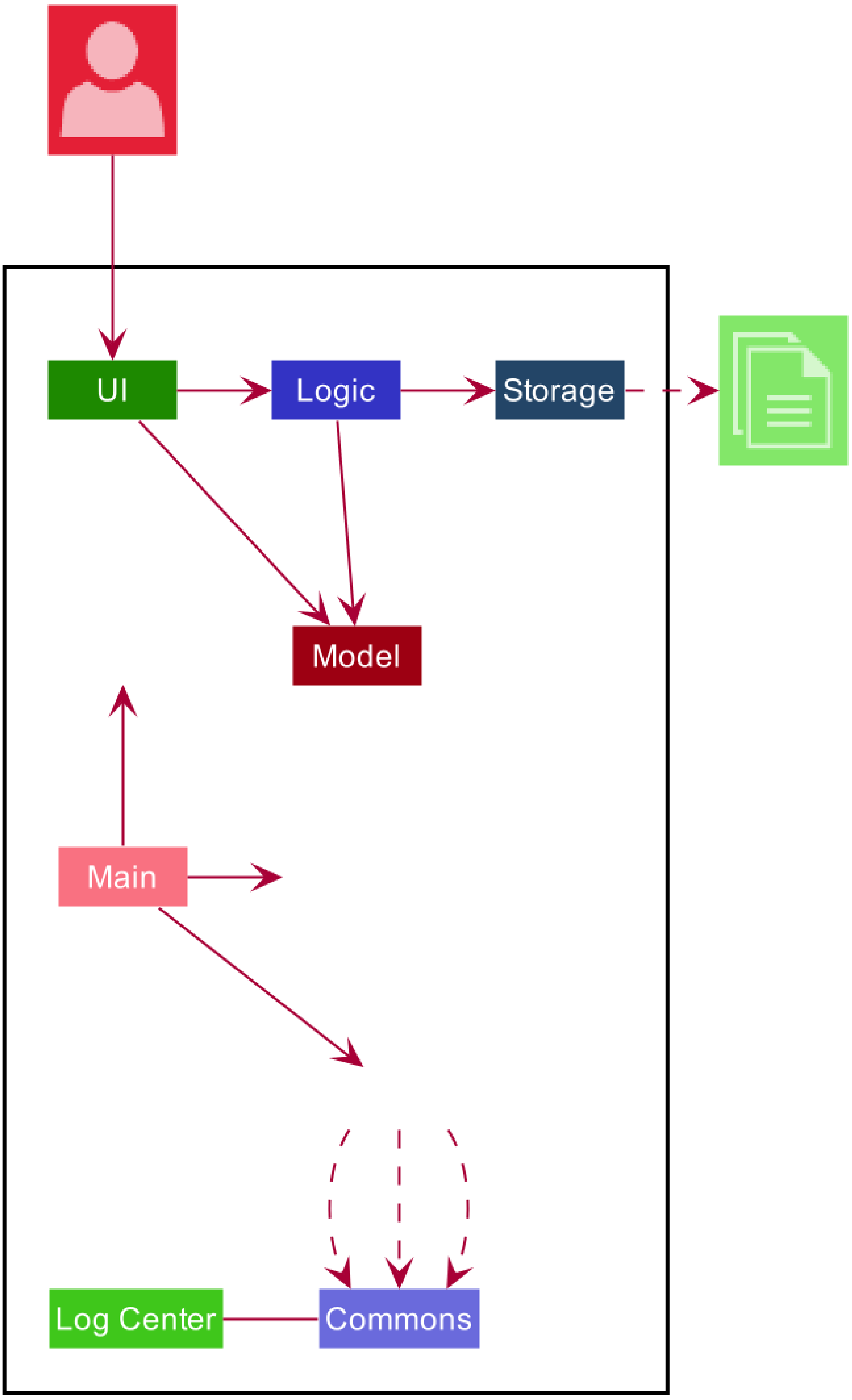

UI
Logic
Storage
Model
Main
Log Center
Commons

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.

> 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.

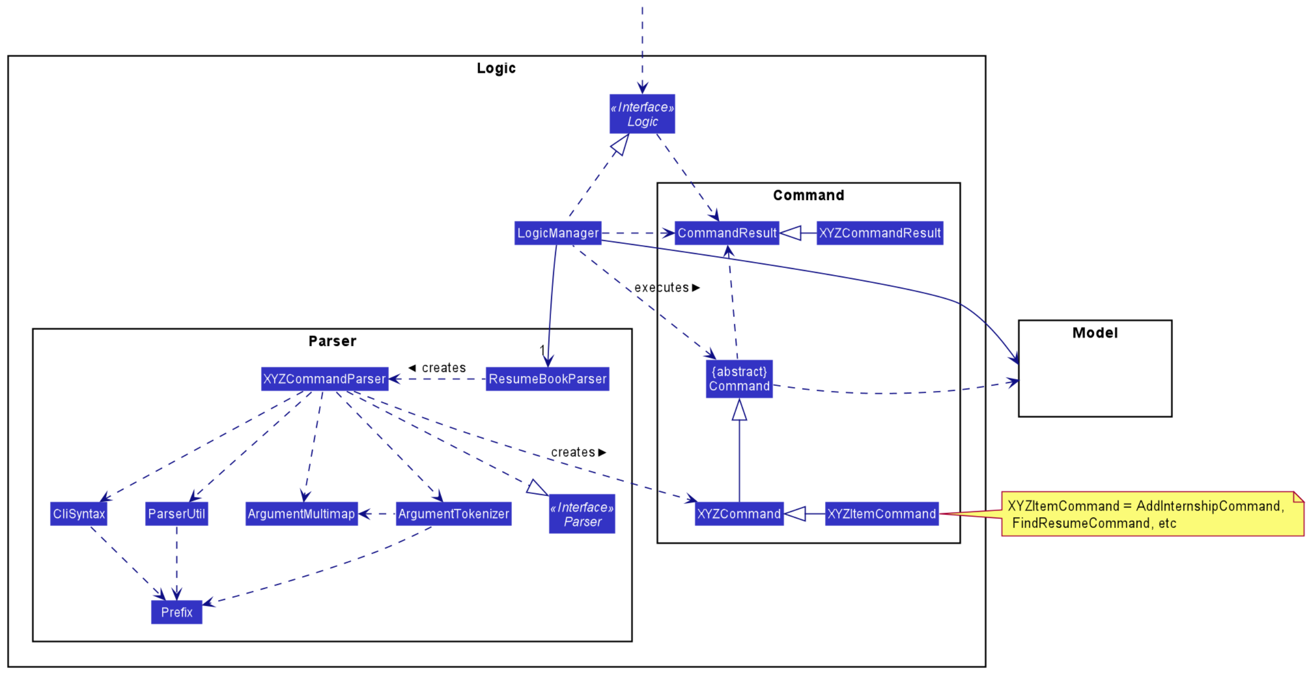
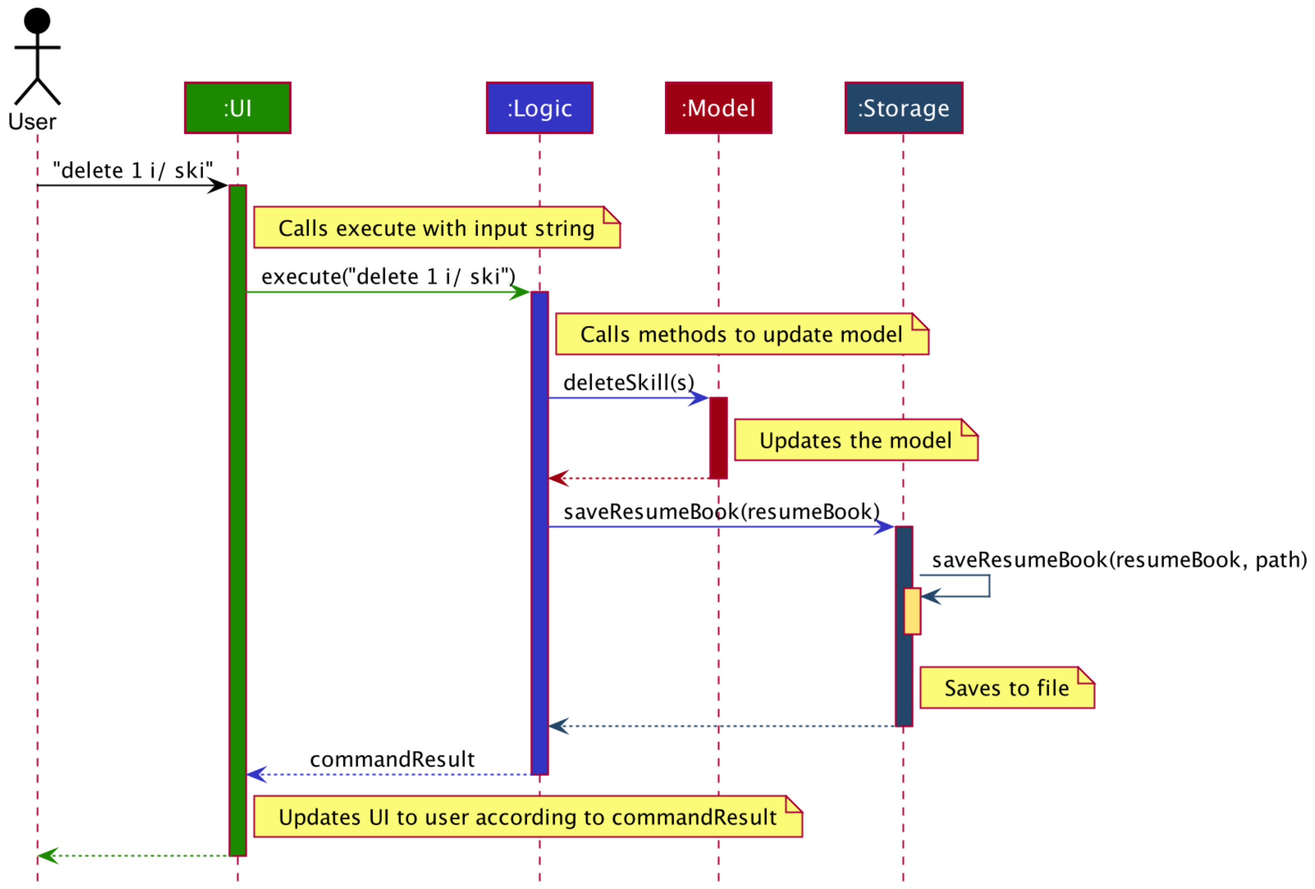


Figure 2. Class Diagram of the Logic Component

{ start of Overall Architecture section written by: Nguyen Chi Hai }

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 i/ ski`, delete skill at index 1.



User
:UI
:Logic
:Model
:Storage
"delete 1 i/ ski"
Calls execute with input string
execute("delete 1 i/ ski")
Calls methods to update model
deleteSkill(s)
Updates the model
saveResumeBook(resumeBook)
saveResumeBook(resumeBook, path)
Saves to file
commandResult
Updates UI to user according to commandResult

Figure 3. Component interactions for `delete 1 i/ ski` *command*

The figure above explains quite clearly how the input of the user is passed into the system through `Ui` that calls `Logic` to make changes in the `Model` and saves to `Storage`. However, it may not be immediately clear how the `Ui` is then updated to give visual feedback to the User. The updating of the Ui is actually done through a combination of observable items, lists, and updating of the Ui with data passed through `commandResult`. Head to Section 3.2, “UI component” to read more.

How the architecture components interact at start up

The *Sequence Diagram* below shows how the components interact with each other at start up.

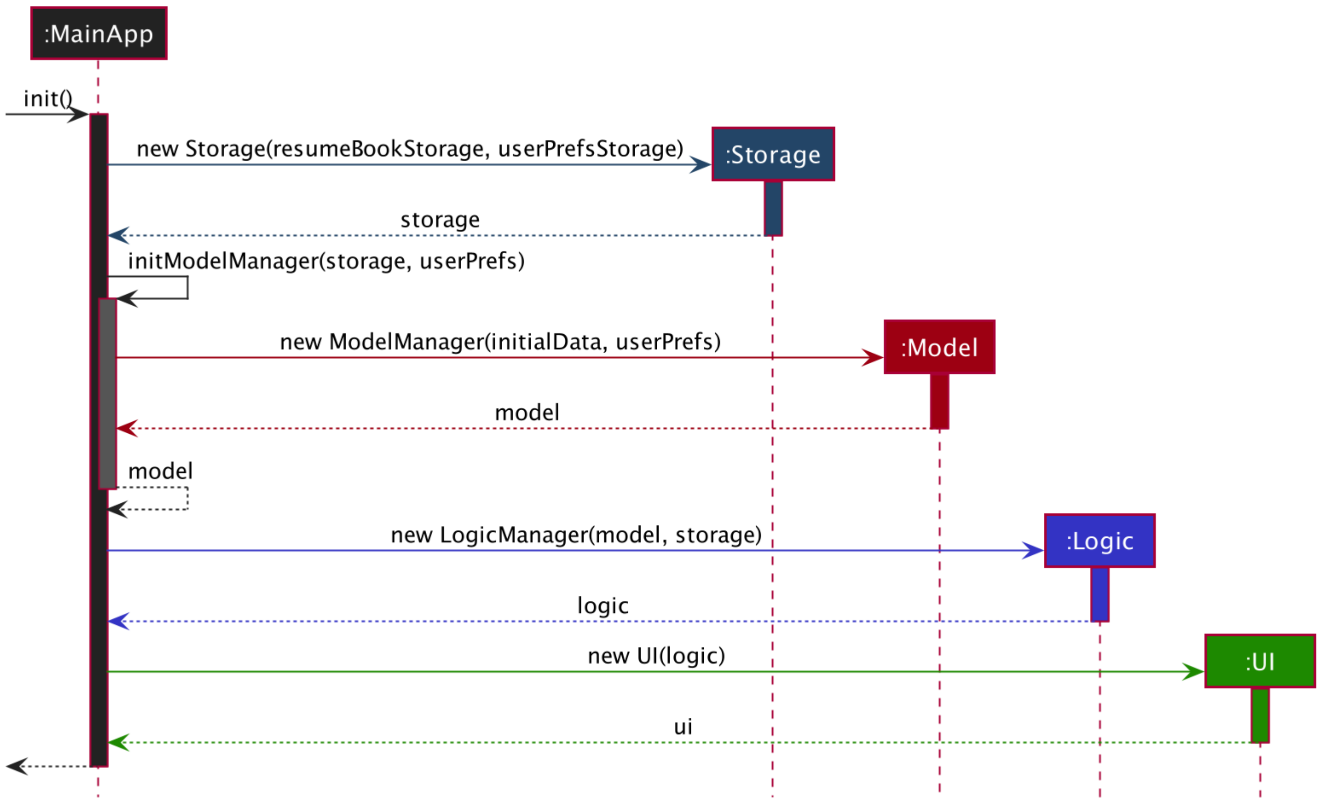

:MainApp
init()
new Storage(resumeBookStorage, userPrefsStorage)
:Storage
storage
initModelManager(storage, userPrefs)
new ModelManager(initialData, userPrefs)
:Model
model
model
new LogicManager(model, storage)
:Logic
logic
new UI(logic)
:UI
ui

Figure 4. Component interactions for initialisation

The sections below give more details of each component.

{ end of Overall Architecture section written by: Nguyen Chi Hai }

{ start of `ui` section written by: Nham Quoc Hung }

3.2. UI component

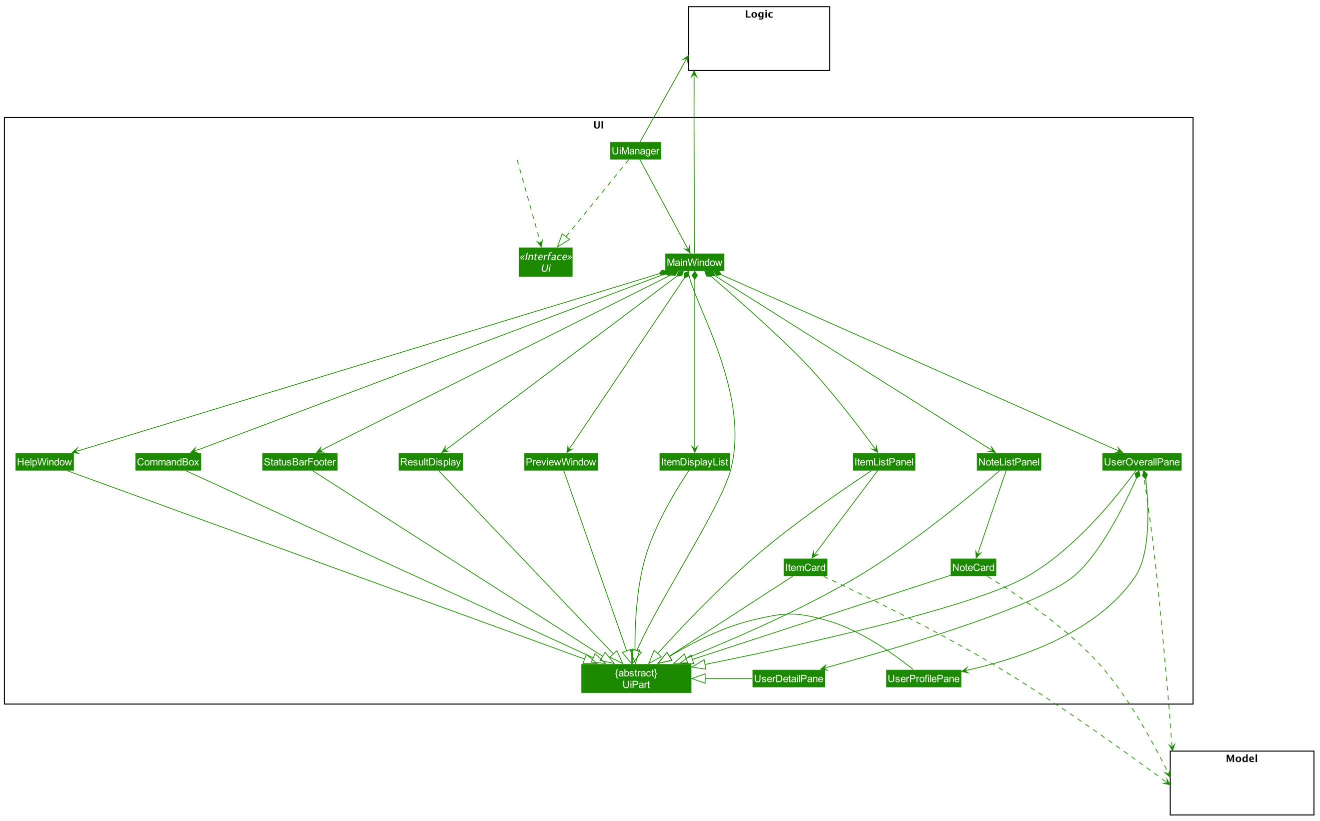


Figure 5. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `ItemDisplayList`, `ItemListPanel`, `NoteListPanel`, `UserOverallPane` and `StatusBarFooter`. 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.
- Responds to events raised by various commands and the UI can be updated accordingly.

{ end of `ui` section written by: Nham Quoc Hung }

3.3. Logic component

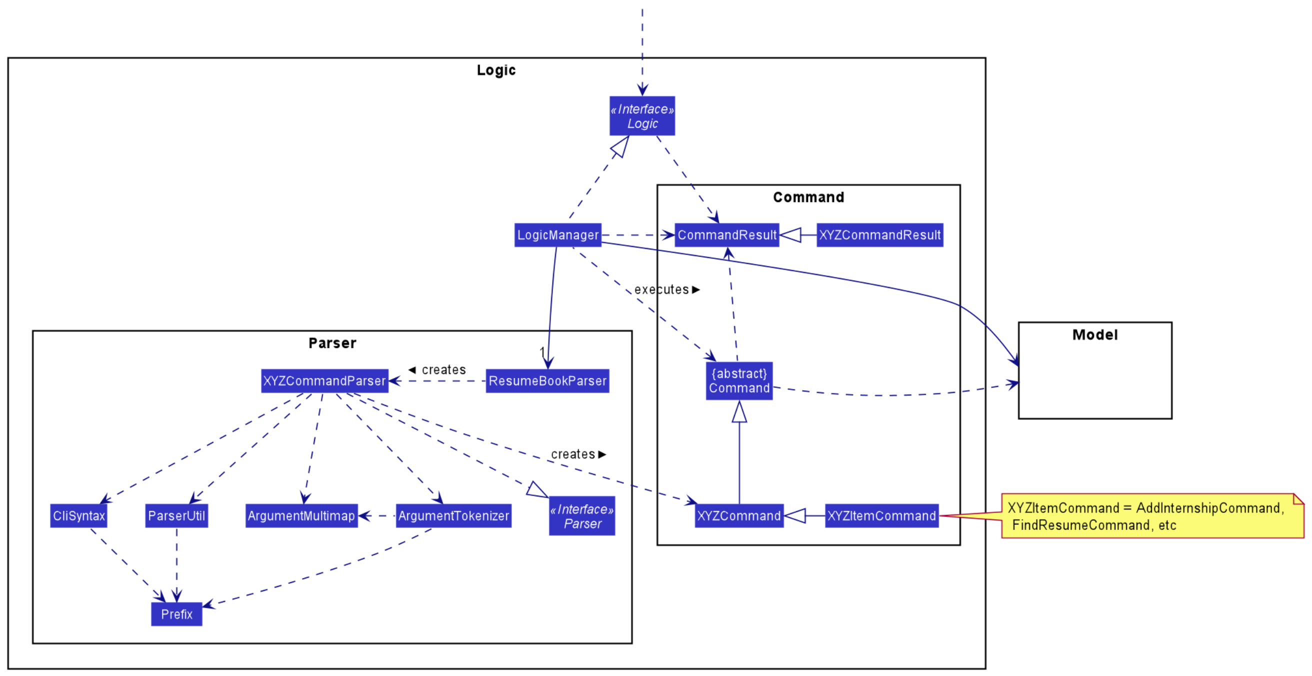


Figure 6. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `ResumeBookParser` 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 new resume).
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 i/ res")` API call.

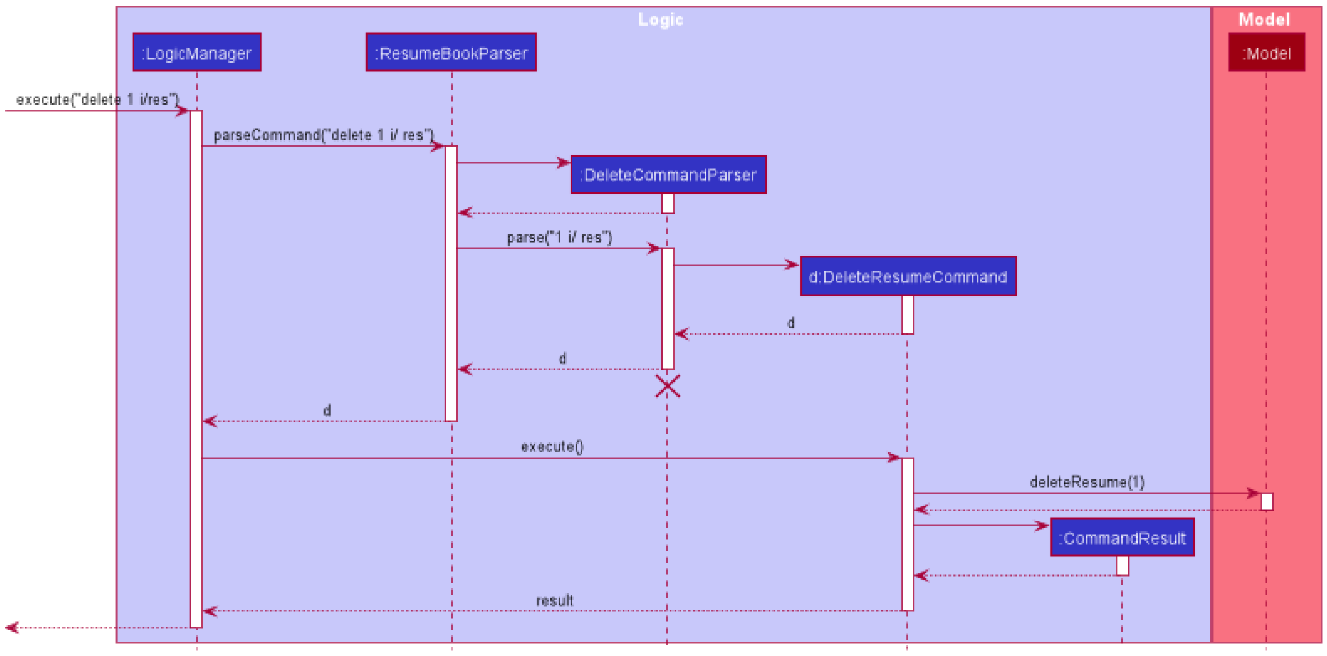


Figure 7. Interactions Inside the Logic Component for the `delete 1` Command

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.

{ start of `model` section written by: Pham Thuy Duong }

3.4. Model component

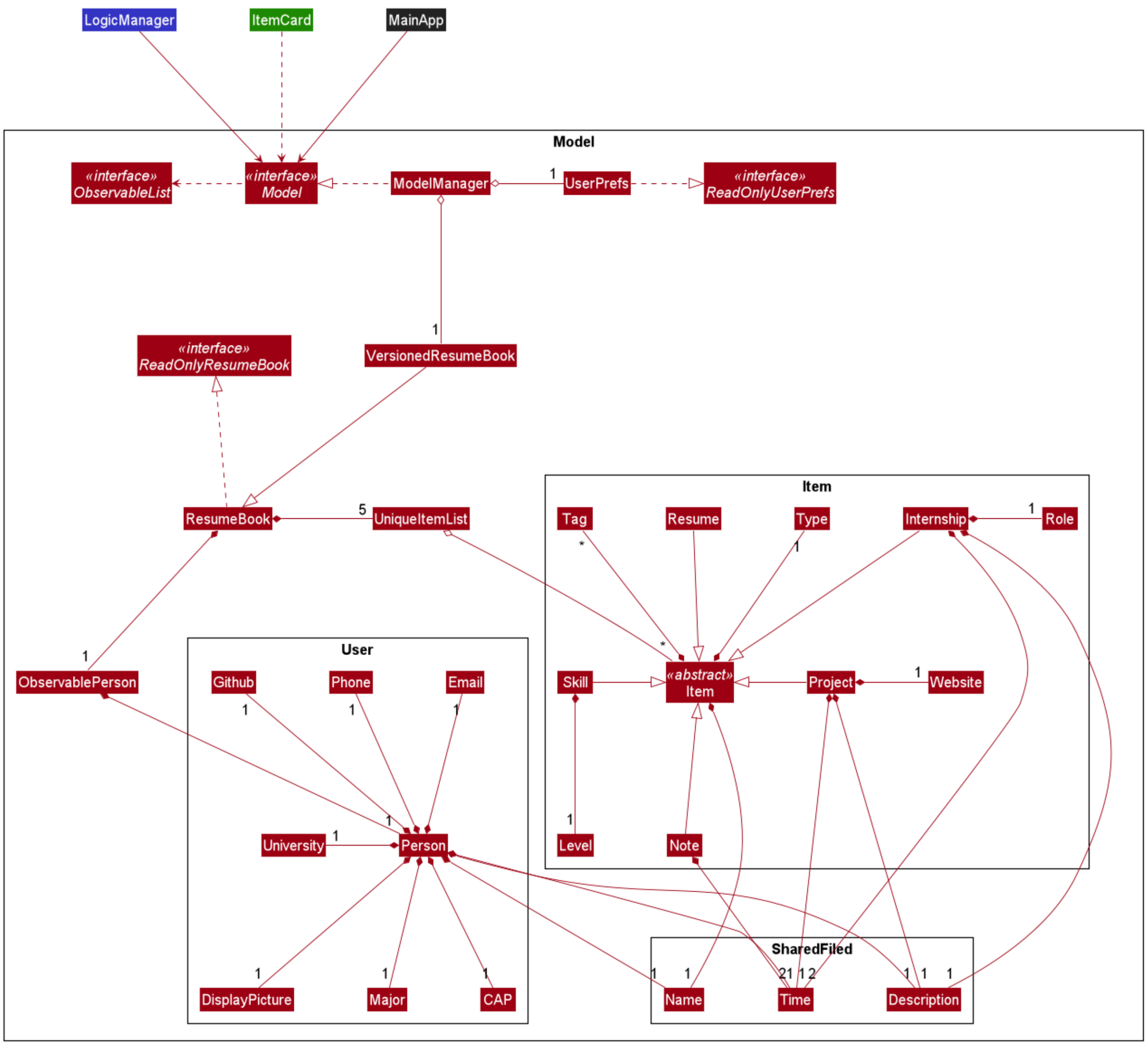


Figure 8. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the Resume Book data.
- stores the Resume Book state using `VersionedResumeBook` to facilitate `undo`/`redo`.
- exposes an `Observable` that contains an internal `Person` with user's data, and two unmodifiable `ObservableList<Item>`, one for `Note` and one for `Internship`, `Project`, `Skill` and `Resume`.
- has the UI bound to its observables so that the UI automatically updates when the data change.
- does not depend on any of the other three components.

{ end of `model` section written by: Pham Thuy Duong }

{ start of `storage` section written by: Nguyen Minh Hoang }

3.5. Storage component

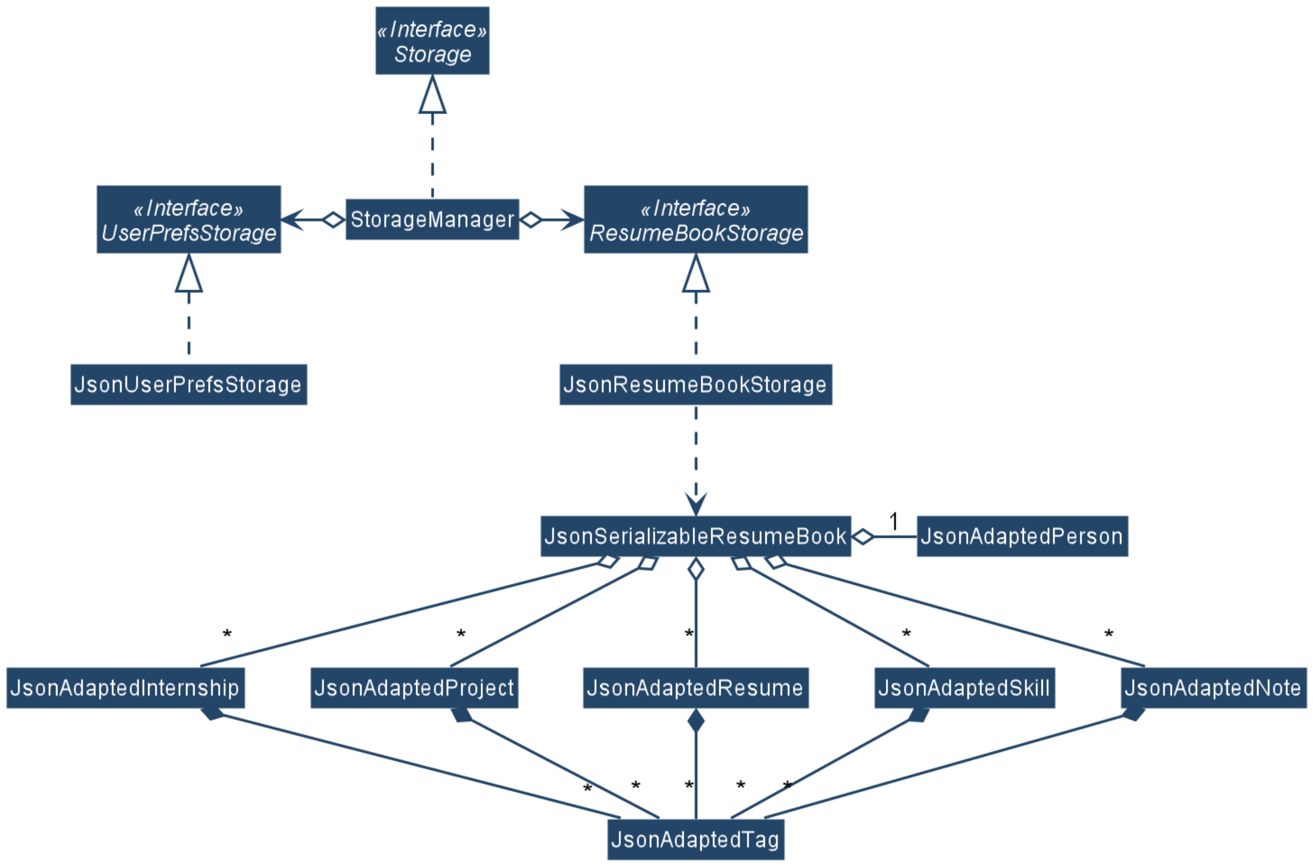


Figure 9. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

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

{ end of `storage` section written by: Nguyen Minh Hoang }

3.6. Common classes

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

4. Implementation

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

{ start of Resume Edit feature implementation written by: Christian James Welly }

4.1. Resume Edit feature

The Resume Edit feature or `redit` allows user to modify the content items of the `Resume` (for example, adding a `Skill` item or removing an `Internship` item). It is not to be confused with the `edit` command, which simply modifies the fields of an `Item` (such as name).

4.1.1. Current Implementation

The `redit` command is facilitated by `ResumeEditCommand`, which extends `Command`. Therefore, like any other `Command` classes, it will have an `execute` method.

Given below is an example usage scenario and how the `redit` works at each step.

Step 1. The user launches the application, and uses the `add` command to add several `Resume`, `Internship`, `Project`, and `Skill` items.

Step 2. The user executes `redit 1 int/ 2` command to add the second `Internship` in the list of `Internship` items to the first `Resume` in the list of `Resume` items.

Step 3. This calls `ResumeBookParser#parseCommand()`, which would create a new `ResumeEditCommandParser` object and call the `ResumeEditCommandParser#parse()` method.

Step 4. A new `ResumeEditCommand` object is created. It contains the index of the `Resume` that is to be modified, and three `Optional<List<Integer>>` representing the indices of `Internship`, `Project` and `Skill` to be modified into the `Resume`. In this example, the `Project` and `Skill` indices are represented by empty `Optional` because the user did not specify any project or skill indices. (This will be further elaborated in the next section)

Step 5. The `ResumeEditCommand#execute()` method is called with the current `model`. A copy of the `Resume` is created and its content is set to refer to the `Internship`, `Project` and `Skill` items specified by the user.

Step 6. A new `ResumeEditCommandResult` object, which contains the edited copy of the `Resume`, is created and returned.

The following sequence diagram shows the process of invokation for `redit`:

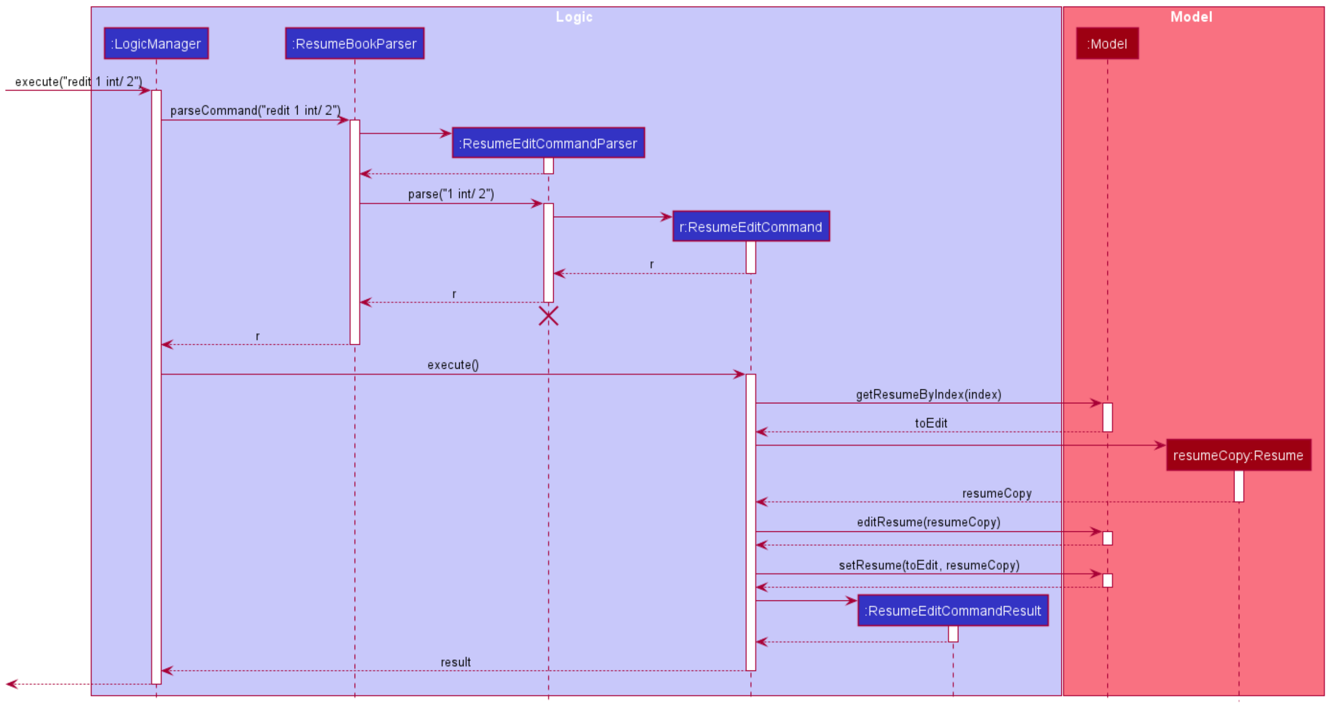


Figure 10. Sequence diagram for ResumeEdit.

Representation of indices after parsing

In Step 4. above, it is mentioned that `Optional<List<Integer>>` is used to represent the indices of `Internship`, `Project`, and `Skill` items. This section elaborates further on the representation.

To explain the various representations, we will use the example of executing `redit 1 int/ 2 3 proj/`:

- A non-empty `List<Integer>` wrapped with `Optional` is used to represent the indices when the user specifies both the item prefix and the item indices. In the above example, indices of `Internship` items will be represented by a `List<Integer>` of `2` and `3`, wrapped with `Optional`.
- An empty `List<Integer>` wrapped with `Optional` is used to represent the indices when the user specifies the item prefix, but no item indices are given. In the above example, indices of `Project` items will be represented by an empty `List<Integer>` wrapped with `Optional`.
- An empty `Optional` is used to represent the indices when the user does not specify the item prefix. In the above example, indices of `Skill` items will be represented with an empty `Optional`.

The three representations are used because `redit` facilitates the following:

- If the prefix and indices are both present, the resume will be modified to contain the content items of that prefix at the specified indices. In the same example above, `Resume` at index 1 will be modified to contain `Internship` items at indices 2 and 3.
- If the prefix is specified but no indices are present, the resume will be modified to remove all the items of that prefix. In the example above, `Resume` at index 1 will be modified to have all its `Project` items removed.
- If the prefix is not specified, the resume will have the items of that prefix unmodified. In the example above, `Resume` at index 1 will not have its `Skill` items modified. If originally there were 4 `Skill` items, then after the command execution, it will still have 4 `Skill` items.

The following activity diagram summarises the execution of ResumeEditCommand:

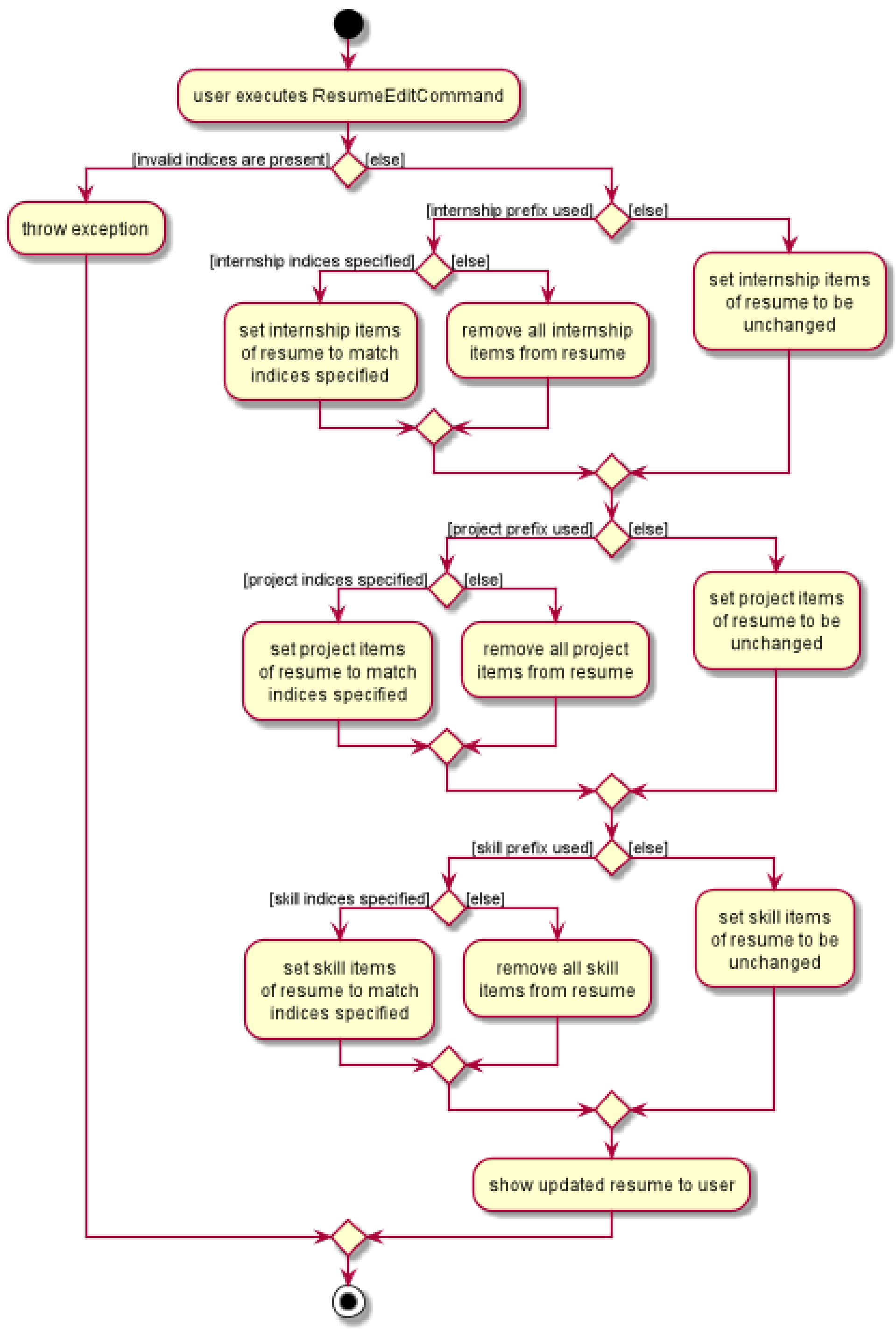

user executes ResumeEditCommand
[invalid indices are present]
[else]
throw exception
[internship prefix used]
[else]
[internship indices specified]
[else]
set internship items of resume to match indices specified
remove all internship items from resume
set internship items of resume to be unchanged
[project prefix used]
[else]
[project indices specified]
[else]
set project items of resume to match indices specified
remove all project items from resume
set project items of resume to be unchanged
[skill prefix used]
[else]
[skill indices specified]
[else]
set skill items of resume to match indices specified
remove all skill items from resume
set skill items of resume to be unchanged
show updated resume to user

Figure 11. Activity Diagram for ResumeEdit.

4.1.2. Design Considerations

Aspect: Whether ResumeEditCommand should extend EditCommand

- **Alternative 1 (current choice):** ResumeEditCommand does not extend EditCommand, but extends Command.
  - Pros: Since redit modifies the content items of the Resume and not the Resume details, this reduces the size of responsibility for EditCommand. Each command class now does one and only one thing so Single Responsibility Principle is observed.

    - Cons: Unable to exploit polymorphism if there is similarity with the `EditCommand`. From user's point-of-view, it may also be confusing to have both `redit` and `edit`.

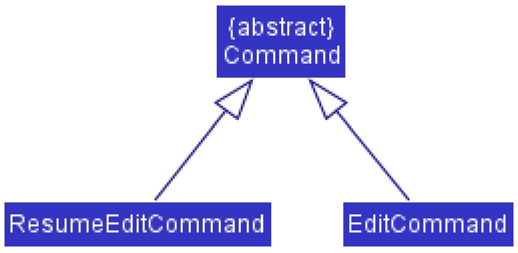


Figure 12. ResumeEditCommand and EditCommand extends Command.

- **Alternative 2:** `ResumeEditCommand` extends `EditCommand`
    - Pros: Some methods in `EditCommand` may be able to be inherited by `ResumeEditCommand`, reducing code duplication.
    - Cons: If the functionality of `ResumeEditCommand` is limited, it could have been combined with `EditCommand` entirely. If the intention of `EditCommand` is to change the `Item` *details* (such as name), and `ResumeEditCommand` only modifies the content items of the `Resume` (without changing any *details*), then this is also a violation of the Liskov Substitution Principle.

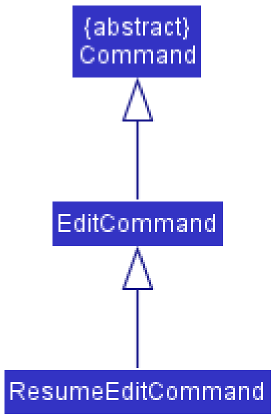


Figure 13. ResumeEditCommand extends EditCommand.

Conclusion: The first design is chosen because `redit` is sufficiently different from `edit`. An `edit` command is intended to change the details of the `Resume`, such as its name, while `redit` is supposed to change the content items that the `Resume` holds.

This also reduces bloating of code and increases the flexibility of `ResumeEditCommand` class if the behaviour of `redit` needs to be changed or added on in the future.

Aspect: Representation of indices after parsing

- **Alternative 1 (current choice):** Usage of `Optional<List<Integer>>`
    - Pros: The 3 different cases is naturally represented when `List<Integer>` is wrapped with `Optional`. There is also an enhanced safety, reducing risk of `NullPointerException`.
    - Cons: More checks are required to ensure that the `Optional` is not empty before getting its

value.

- **Alternative 2:** Usage of `null` and `List<Integer>`
    - Pros: Implementation is much simpler, and code becomes much more concise.
    - Cons: High risk of getting a `NullPointerException` if `null` is not handled carefully.

Conclusion: We went with `Optional` as it is more expressive than using `null`: it has a clearer semantic when checking whether the value of `Optional` is empty or not than to check whether the variable is a `null` value.

Additionally, using `Optional` provides much less risk to getting `NullPointerException`. The reduced risk allows the developers to potentially save some debugging time, and developers worry less about handling the `NullPointerException`.

{ end of Resume Edit feature implementation written by: Christian James Welly }

{ start of Tag Pull feature implementation written by: Christian James Welly }

4.2. Tag Pull feature

The Tag Pull feature is similar to Resume Edit Feature in the sense that it modifies the content items of the `Resume`. Unlike Resume Edit which modifies using the content item indices, the Tag Pull feature modifies the resume by *adding* items with the specified tags on top of existing items in the `Resume`.

4.2.1. Current Implementation

The `tagpull` command is facilitated by `TagPullCommand`, which extends `Command`. Therefore, like any other `Command` classes, it will have an `execute` method.

Given below is an example usage scenario and how the `tagpull` works at each step.

Step 1. The user launches the application, and uses the `add` command to add several `Resume`, `Internship`, `Project`, and `Skill` items.

Step 2. The user executes `tagpull 2 #/ tech` command to add all items that have been tagged with `tech` to the first `Resume` in the list of `Resume` items.

Step 3. This calls `ResumeBookParser#parseCommand()`, which would create a new `TagPullCommandParser` object and call the `TagPullCommandParser#parse()` method.

Step 4. A new `TagPullCommand` object is created. It contains the index of the `Resume` that is to be modified, and the tags of the items which the user wants to add. In this example, it will only have the `tech` tag.

Step 5. The `TagPullCommand#execute()` method is called with the current `model`. A copy of the `Resume` is created and all the items with matching tags are retrieved from `model`. The content of the copied `Resume` is updated to now contain all the items with matching tags, on top of existing ones.

Step 6. A new `TagPullCommand` object, which contains the edited copy of the `Resume`, is created and returned.

The following sequence diagram shows the process of invokation for `tagpull`:

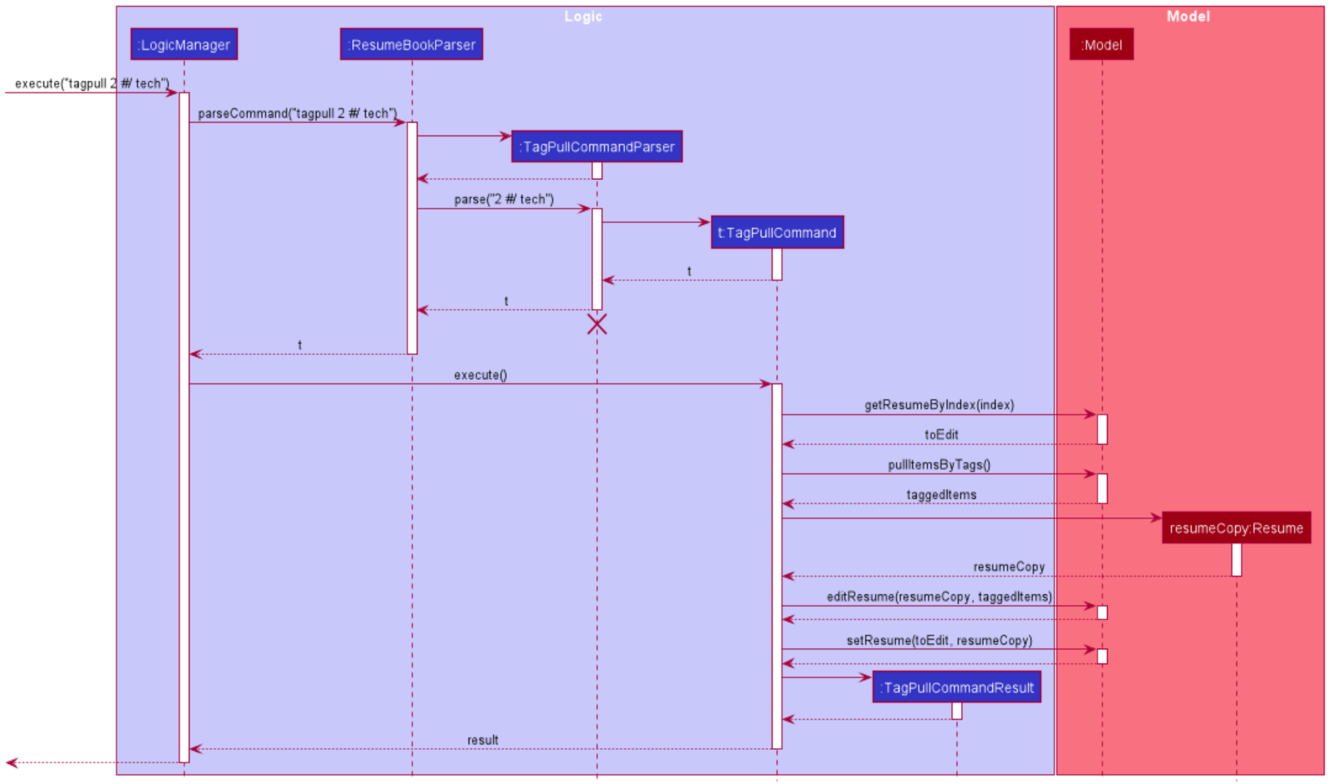
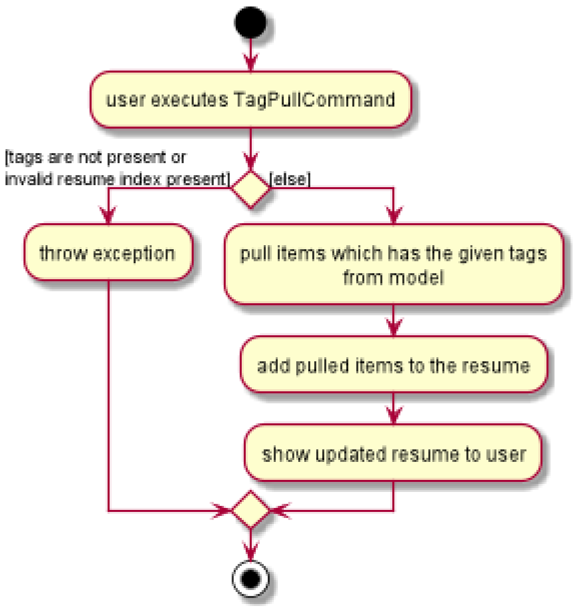


Figure 14. Sequence Diagram for Tag Pull.

The following activity diagram summarises the execution of `tagpull`:



user executes TagPullCommand
[tags are not present or invalid resume index present]
[else]
throw exception
pull items which has the given tags from model
add pulled items to the resume
show updated resume to user

Figure 15. Activity Diagram for Tag Pull.

4.2.2. Design Considerations

Aspect: Integrating Tag Pull to Resume Edit

- **Alternative 1 (current choice):** Separating `redit` and `tagpull`
    - Pros: Separation of concerns. `redit` handles updates of `Resume` using indices and `tagpull` handles updates of `Resume` using tags.
    - Cons: There is some code duplication due to similarities in behaviour.
- **Alternative 2:** Combine `redit` with `tagpull`, making use of `#/` as prefix for `redit` command
    - Pros: It may be intuitive for user to only have a single command that handles modification of `Resume`
    - Cons: The implementation of `redit` becomes much more complicated as various combinations of input has to be considered. For example, considerations of what the expected behaviour should be if both indices and tags are given as arguments.

Conclusion: We decided to separate the two commands in order to have a simpler implementation of the commands. By separating the two, there is a separation of concerns and there is no need to consider the behaviour when both indices and tags are given as arguments.

It may also save the user some confusion since the `redit` feature specifically handles only updates using indices while the `tagpull` feature handles only updates using tags.

As we have separated the two commands, we can then also vary the behaviour of the two commands slightly. We have implemented `redit` to be able to *modify* (adding, changing, and removing) the `Resume` item, while `tagpull` modifies only by *adding* onto existing content items in the `Resume`.

{ end of Tag Pull feature implementation written by: Christian James Welly }

{ start of Implementation of `Command` section written by: Nguyen Chi Hai }

4.3. Implementation of `Command` classes

4.3.1. Current Implementation

Currently, there are several object `Type` which are subclasses of `Item`, namely `Resume`, `Internship`, `Skill`, `Note` and `Project`.

Commands that are dependent on item `Type`, namely `AddCommand`, `DeleteCommand`, `EditCommand`, `FindCommand`, `ListCommand`, `SortCommand`, and `ViewCommand` are implemented as abstract classes that inherits from `Command` and would have a concrete classes that corresponds to each item `Type`. For example, `AddCommand` is an abstract class that `AddInternshipCommand` and `AddSkillCommand` inherits from.

Commands that are not dependent on item `Type` (eg. `EditUserCommand`, `ResumeEditCommand`) are implemented as concrete classes that inherits directly from `Command`.

From this point onwards, for the sake of clarity in our discussion, commands that are dependent on type will be called `ABCCommand` whereas those who are independent of type will be called `XYZCommand`.

The following is the class diagram for `Command` and its subclasses.

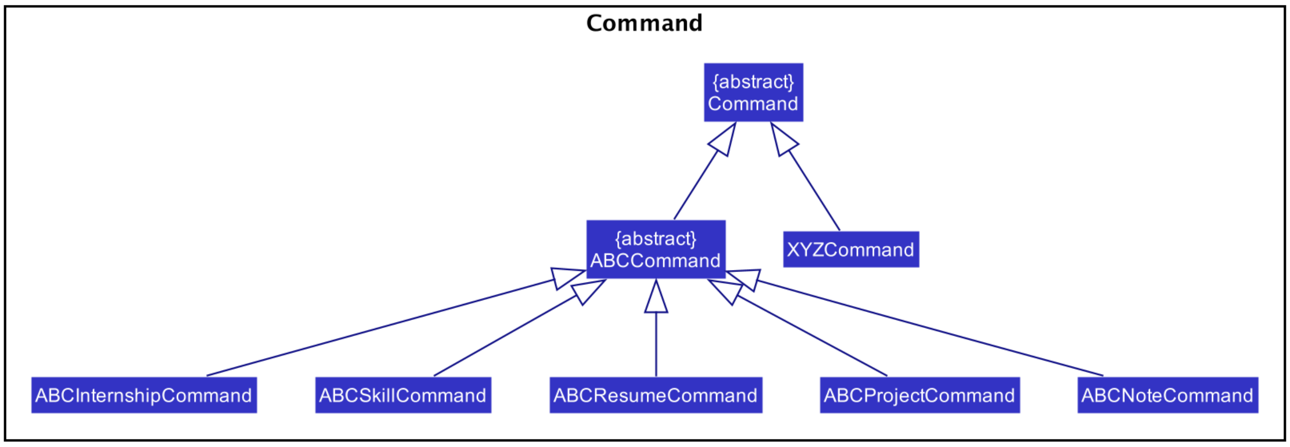


Figure 16. Component `XYZCommand` is independent of `Type` whereas `ABCCommand` is dependent on `Type`.

4.3.2. Design Considerations

Aspect: Whether to separate the `ABCCommand` that is dependent on type into many `ABCItemCommand`

Alternative 1 (current choice): `ABCCommand` is separated into many `ABCItemCommand`. Parser will parse user input and create the exact `ABCItemCommand`. The following is the activity diagram for execution of `AddResumeCommand` when the user adds a resume.

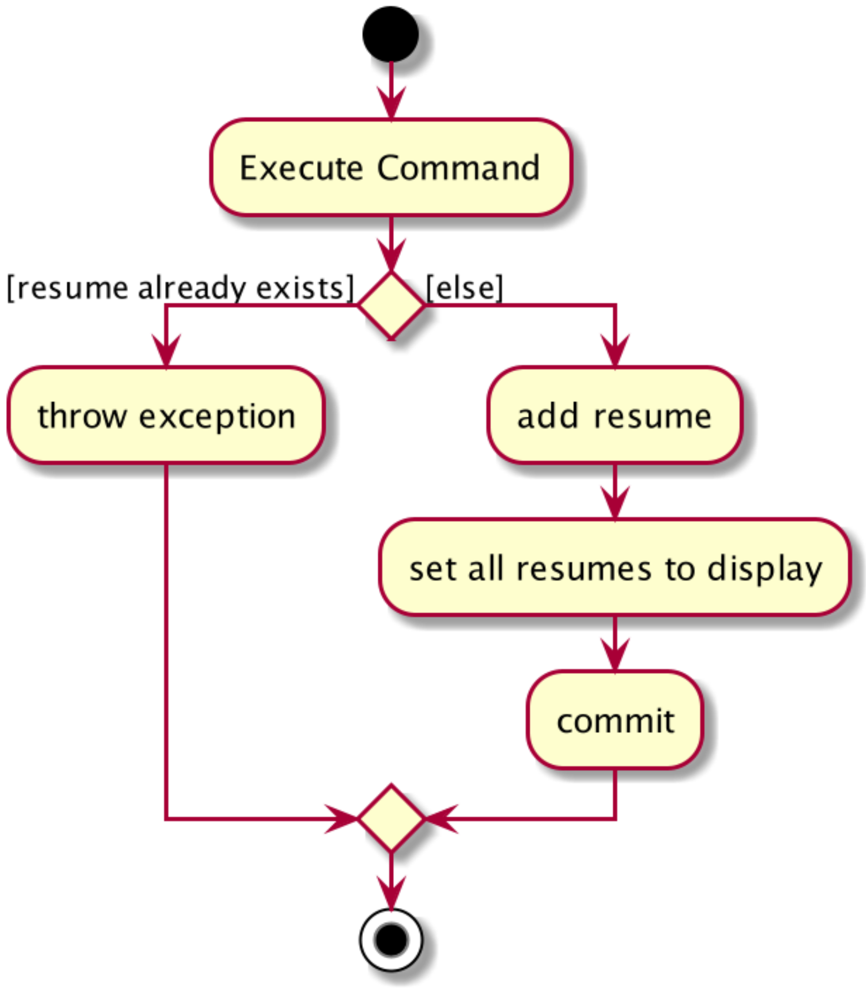


Figure 17. Activity diagram for execution of `AddResumeCommand`

This leads to a cleaner execution method of each ABCItemCommand as each command class has a clear goal.

- Pros: More OOP. Each `ABCItemCommand` has its own and distinct functionality. Each `ABCItemCommand` has more flexible behaviour and can be easily changed as required.
- Cons: Many classes have to be maintained.

Alternative 2: `ABCCommand` is not separated into many `ABCItemCommand`. The following is the activity diagram for execution of `AddCommand` if `AddCommand` is not separated into `AddResumeCommand`, `AddNoteCommand`, `AddInternshipCommand`, `AddProjectCommand`, and `AddSkillCommand` when the user adds a resume.

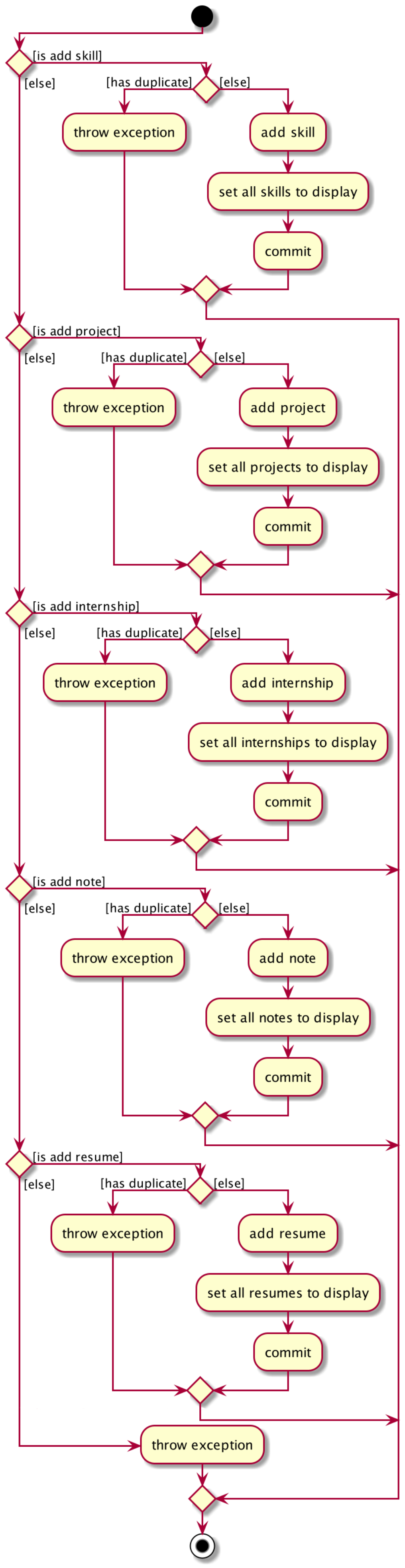

[is add skill]
[else]
[has duplicate]
[else]
throw exception
add skill
set all skills to display
commit
[is add project]
[else]
[has duplicate]
[else]
throw exception
add project
set all projects to display
commit
[is add internship]
[else]
[has duplicate]
[else]
throw exception
add internship
set all internships to display
commit
[is add note]
[else]
[has duplicate]
[else]
throw exception
add note
set all notes to display
commit
[is add resume]
[else]
[has duplicate]
[else]
throw exception
add resume
set all resumes to display
commit
throw exception

Figure 18. Activity diagram for execution of `AddCommand`

Implementing `ABCCommand` this way forces execute to be switch-cased. Functionality of execute would vary depending on the item `Type`.

- Pros: Only one command is required, regardless of number of items. Low overhead.
- Cons: Long `execute` method due to the need for handling the different item types as seen from the logic of the activity diagram. Item `Type` would also need to be stored. Undesirable variable functionality of `execute` command depending on the `Type` field despite it being from the same class. ie. `AddItem` can add `Internship` to the `Internship` list, or add `Skill` to `Skill` list.

Conclusion: We went with our current design because it allows for each command type to only have one distinct job which is more in line with the object oriented programming paradigm of Single Responsibility Principle. Instead of having one single class that that would need to change if implementation of any of the `Type` changes, our implementation ensures that our many command classes would only have a single reason to change. Moreover, our current implementation also reduces double work as `Parser` will not have to parse `Type` in the user input to create the `ABCCommand`, then only to be switch-cased again in `ABCCommand`.

{ start of `observable-user` section written by: Nguyen Minh Hoang }

{ end of Implementation of `Command` section written by: Nguyen Chi Hai }

4.4. Usage of `Observable` and `Observer`

4.4.1. Reason for Implementation

ResuMe inherits a lot of implementations from AddressBook Level 3, one of which is the usage of `ObservableList` that allows a list of item to be automatically updated in the UI every time a change is made to the underlying model.

When we added functionality for a user profile and sequentially a user profile window, the need for auto UI update surfaced. We could not use the existing implementation since it is only for `ObservableList`. The initial rectification is to explicitly call a UI update, but that requires the UI to directly reach out to the Model to get the user profile data. This seriously breached many Software Engineering principles and we decided to amend it.

4.4.2. Design Considerations

- **Alternative 1**: Use `Observable` and `Observer`
  - Pros: Is way easier to implement and does not disrupt the existing code base much.
  - Cons: The two classes are deprecated in `Java 9` due to various shortcomings.
- **Alternative 2**: Use `java.beans` package
  - Pros: Is designed to replace `Observable` and `Observer` and is currently the industry standard.
  - Cons: Is harder to implement and would require a lot of refactoring of the existing codes.

4.4.3. Conclusion

After deliberation we decided to go with the first alternative, due to time constraint and to avoid the potential havoc that would arise from refactoring the code base. In addition, the limitations of `Observable` and `Observer` are not likely to manifest considering the usage scenarios for our application (offline and not multi-threaded).

We would, however, make it a priority to change the implementation to Alternative 2, since we want to scale up our application and it is not recommendable to build it on top of something that is no longer supported.

{ end of `observable-user` section written by: Nguyen Minh Hoang }

{ start of `me` section written by: Nham Quoc Hung }

4.5. Me feature

This feature intends to serve a single user of the application to sets and updates his/her user profile. The profile is then reflected in the user's profile panel.

4.5.1. `me`: Edit User Profile

Implementation

`me` is supported by the `EditUserCommand`, where it allows the main user to modify and update user information that includes `display picture`, `name`, `description`, `phone`, `email`, `github`, `university`, `major`, `from`, `to`, `cap`.

Given below is an example usage scenario:

Step 1. User launches the ResuMe application for the first time. The user profile data is not yet edited and will thus be initialized with the initial json data stored.

Step 2. User executes `me dp/ FILEPATH n/ NAME d/ DESCRIPTION p/ PHONE e/ EMAIL g/ GITHUB u/ UNIVERSITY m/ MAJOR f/ FROM t/ TO c/ CAP` so as to update the Person object currently stored in Model as well as Storage.

```
me dp/ /Users/nhamquochung/Desktop/test.png n/ HUNG d/ I am an aspiring software
engineer. p/ 91648888 e/ nhamhung.gttn@gmail.com g/ nhamhung u/ National University of
Singapore m/ Computer Science f/ 08-2018 t/ 05-2022 c/ 4.0 5.0
```

Step 3. The user profile panel will be updated accordingly.

Note: To set customised user picture, the file path of your display picture has to be absolute and from the root directory of your computer.

Command sequence:

1. User type `me [dp/ FILEPATH] [n/ NAME] …` command in the command box.

2. Command is executed by Logic Manager.
3. Command is parsed by `ResumeBookParser` which identifies what type of command it is. An `EditUserParser` is returned accordingly.
4. `EditUserParser` extracts out different fields specified in the command based on their prefixes and returns an `EditUserCommand` with an `EditUserDescriptor` object parameter which contains information on which attributes of user data is updated or kept unchanged.
5. `EditUserCommand` then calls `execute()` which first gets the existing `Person` in Model as the `userToEdit`. It then creates a new `editedUser` based on `EditUserDescriptor` and set the current `userToEdit` in Model to `editedUser`. Afterwards, a `CommandResult` is returned to Logic with data and feedback to be displayed to the user.
6. Feedback acknowledgement is displayed by `ResultDisplay`. User profile changes are displayed automatically as the user `Person` is wrapped around by a JavaFx Observable as an `ObservablePerson` so that the user profile 's display is always updated after execution of every command.

The following sequence diagram shows how the `me` feature allows user to edit his/her user profile:

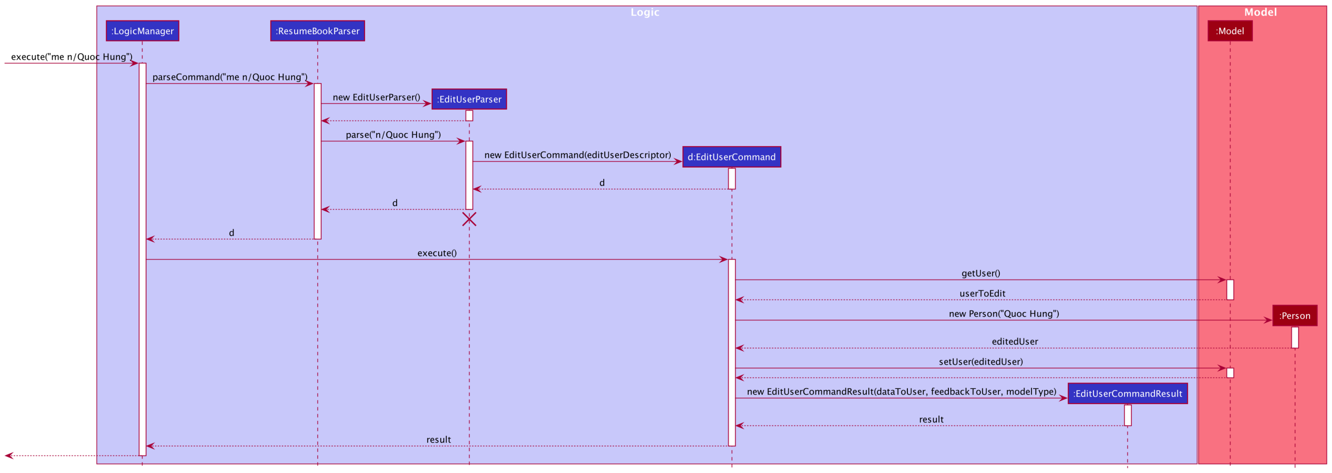


Figure 19. Sequence Diagram for EditUserCommand

Design Considerations

Aspect: Whether `EditUserCommand` should extend `EditCommand`

- **Alternative 1 (current choice):** `EditUserCommand` does not extend `EditCommand`, but extends `Command`.

This design is chosen because while `EditCommand` takes into account the item index as all items are stored in a list in Model, `EditUserCommand` only concerns with a single `Person` who is the main user.

- Pros: Reduces unnecessary overhead for `EditUserCommand`.
- Cons: Unable to exploit polymorphism if there is similarity with the `EditCommand`.
  - **Alternative 2: `EditUserCommand` extends `EditCommand`**
- Pros: Better utilise polymorphism and perhaps can be more intuitive as it is also a command to edit.
- Cons: Does not treat it as an entirely separate command with a distinct keyword `me` which is

more intuitive for the user.

Aspect: Whether to have both `AddUserCommand` and `EditUserCommand`

- **Alternative 1 (current choice):** A default user data is initialized and displayed at first start-up. User can update it afterwards. This design is chosen because `EditUserCommand` only concerns with a single `Person` object in the Model as the sole user. Hence there is no need for `AddUserCommand` as `EditUserCommand` when executed always creates a new `Person` object to replace the existing one and update the Model and Ui accordingly.
    - Pros: Reduces unnecessary code duplication with AddUserCommand is present.
    - Cons: User may expect to have `add` command intuitively.
- Alternative 2: Have both `AddUserCommand` and `EditUserCommand`
    - Pros: User can intuitively treat `add` as adding in a new `user` and `edit` as just modifying an existing `user`.
    - Cons: There will be code duplication and the one single user logic is not fully utilised to reduce code.

Conclusion: We went with our current design because it only concerns with a single target user whose usage of the application can help him/her manage and craft multiple resume versions. As such, only a single user profile which includes essential biography and educational background needs to be managed to be included in every generated resume. This user profile must thus be made clearly, constantly visible and to be updated with a simple and powerful command.

{ end of `me` section written by: Nham Quoc Hung }

{ start of `note` section written by: Nham Quoc Hung }

4.5.2. Note taking feature: take simple notes or reminders

Implementation

This feature utilises a `Note` class that extends `Item`. It provides necessary functionality related to note taking in order to support the user in his/her resume building and management.

Given below is an example usage scenario:

Scenario 1. Add a reminder note: `add i/ note`

Step 1. The user launches the ResuMe application. Data will be loaded from storage to fill the note list in model.

Step 2. The user executes `add i/ note n/ NAME t/ TIME #/ TAG`. In `ResumeBook`, the note list is implemented as a `UniqueItemList` which implements an `add()` method that will always check if this note already exists in current note list. This check is done by iterating through every note in the list and compare to this note using an `isSame()` method that checks for the same note name and time. If a same note already exists, ResuMe throws a duplicate error message.

Step 3. If no error is thrown, the note will be created, defaulted as `not done` and added to the current note list with according `Ui` update.

Scenario 2. Edit an existing note: `edit i/ note`

Step 1. Once data has been loaded from `storage` to `model`, the list of notes in the ResumeBook could either contain some notes, or is empty.

Step 2. The user executes `edit INDEX i/ Note [n/ NAME] [t/ TIME] [#/ TAG]`. If the specified note index is invalid or out of bound, ResuMe will throw an invalid index error message.

Step 3. If no index error is thrown, edited values will be captured by an `EditNoteDescriptor` object and the note at the specified index will be extracted to be updated according to the fields captured by this descriptor. However, if this note to be edited becomes another similar note in the list, a duplicate item error will be thrown.

Step 4. If no duplicate error is thrown, ResuMe will replace the note at this index with its edited version.

The following activity diagram summarises this process when user executes `edit i/ note` command:

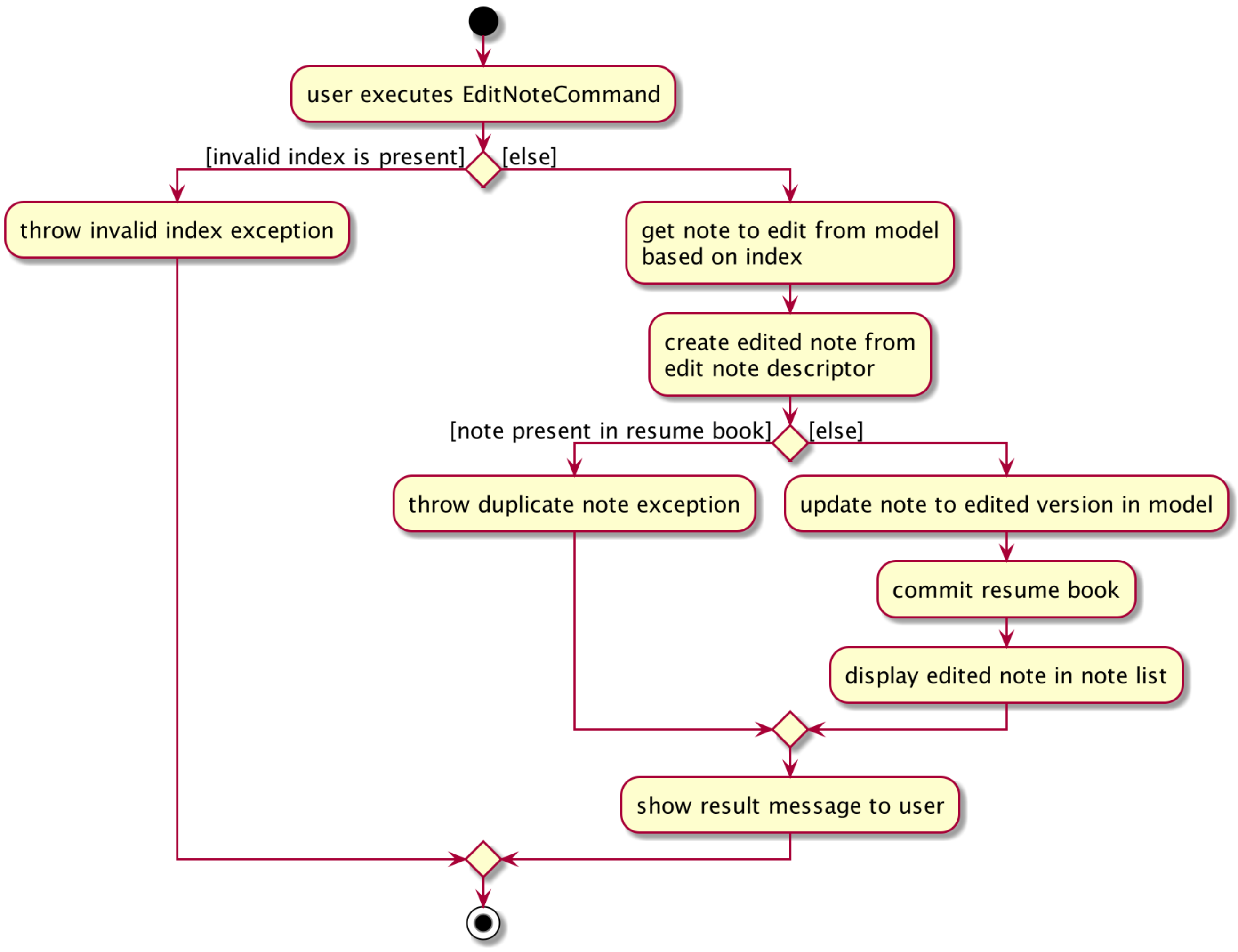

user executes EditNoteCommand
[invalid index is present]
[else]
throw invalid index exception
get note to edit from model based on index
create edited note from edit note descriptor
[note present in resume book]
[else]
throw duplicate note exception
update note to edited version in model
commit resume book
display edited note in note list
show result message to user

Figure 20. Activity Diagram for EditNoteCommand

Scenario 3. Mark an existing note as done: `done`

Step 1. Given the currently displaying list of notes, the user executes `done INDEX`. If the specified note index is invalid or out of bound, ResuMe throws an error message.

Step 2. The corresponding note at this index is marked as done with an Ui update from a `tick` to `cross`. If the note has already been marked as done, a user feedback message is displayed to notify the user.

Design Considerations

Aspect: Whether this feature is necessary in supporting the user

- **Alternative 1 (current choice):** Note taking is implemented with functionality to `add`, `edit`, `view`, `list`, `find`, `delete`, `sort` and `done`.

This design is chosen because it can be an important part of overall user experience in managing his/her resumes. It is an enhancement to existing features that deal strictly with building resumes, by allowing the user to jot down short entries which can serve as simple reminders for them.

- Pros: User may work on crafting his/her `Internship`, `Project` and `Skill` with a lot of writing and summarising past experiences. As such, this brainstorming process tends to be over a long time. Note taking thus makes it easier for user to resume his/her work.
- Cons: Note taking may seem like an unrelated feature to building resumes. Thus, it may be underutilised if the user only focuses on managing resumes.
- **Alternative 2:** Remove note taking feature from the application
- Pros: Make ResuMe more inline with being a resume building application.
- Cons: May miss out on a portion of users who would appreciate this feature, especially those with a habit of jotting down notes.

Aspect: Whether `Note` class should extend `Item` class

- **Alternative 1 (current choice):** `Note` is also an `Item`

This design is chosen because note taking feature is intended to have similar `Command` to a typical `Item` such as `AddCommand`, `EditCommand` and `SortCommand`. As such, by extending `Item`, `Note` can inherit attributes such as `Index` and `Tag` as well as being able to kept as a `UniqueItemList` in `Model`.

- Pros: Reduce code duplication in achieving the same functionality between `Note` commands and other `Item` commands. `Note` can also inherit important attributes such as `index` and `tags` which it intends to have.
- Cons: Right now other subclasses of `Item` are `Internship`, `Project`, `Skill`, `Person` and `Resume` which are all relevant to building a `Resume`. Details from these items will be included in the actual resume PDF generated. As such, `Note` as a subclass of `Item` can add confusion because it is not part of a resume.
- **Alternative 2:** Implement a `Note` class which does not inherit from `Item`
- Pros: Make it more independent and do not interfere with the design considerations for other resume-related items.

- Cons: However, this would lead to a significant code duplication to achieve the same purpose. This could violate `Don□t Repeat Yourself` principle which increases the amount of work required to test the application.

> **Conclusion:** We went with our current design because we feel that note taking feature is helpful for user in managing multiple resume versions as it allows him/her to interact in more ways with the process of logging their experiences to include in resumes. We foresee that crafting resumes can be prolonged and thus this helps them to resume with ease. With regards to inheritance consideration, we decided that it would be faster and more reliable to make `Note` an `Item` so as to minimise double work and potential bugs. This is hidden from the user’s perspective and so this design suits our needs given the short time frame that we have.

{ end of `note` section written by: Nham Quoc Hung }

{ start of `export` section written by: Pham Thuy Duong }

4.6. Export Feature

The Export Feature supports two main actions: previewing the content of a `Resume` item, and then generating a `.pdf` file from it.

4.6.1. Previewing a resume: `rpreview`

Implementation

`rpreview` is supported by the new `Command`, `ResumePreviewCommand` and the additional method `toPreview()` implemented in `Internship`, `Project` and `Skill`, which return the content of the item in textual format.

Given below is an example usage scenario:

Step 1. The user launches the **ResuMe** application. After loading data from storage to the application memory, the list of `resumes` in the `ResumeBook` could either contain some resumes, or is empty.

Step 2. The user executes `rpreview INDEX`. If the specified resume `INDEX` is out of bound, **ResuMe** throws an error message.

Step 3. The application retrieves the correct `Resume` item and calls `toPreview()` on the items contained in that resume.

The following activity diagram summarises what happens when a user executes `rpreview` command:

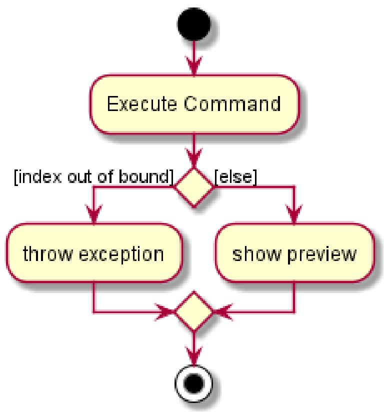


Figure 21. Activity Diagram for ResumePreviewCommand

Design Considerations

Aspect: Where `rpreview` is displayed

- **Alternative 1 (current choice):** Displays in a separate preview pop-up window.
  - Pros: The preview is separated from the application and does not clutter the application view. It is also no longer confined to a small space and hence improves readability.
  - Cons: Additional non-command-line action needed to close the pop-up window after previewing.
- **Alternative 2:** Display in the same panel as `view`
  - Pros: No significant change to UI component.
  - Cons: Multiple commands needed if user finds out about a typo in an item, wants to view the item in details, fix the typo and then check the preview again to ensure there is no more error.
- **Alternative 3:** Create a separate preview box to display the resume preview. Additionally, this box could be implemented such that it automatically updates when the content of the `Resume` item is edited.
  - Pros: User can see the resume preview in the preview box, and the item details in the view box at the same time, hence saving time switching between views.
  - Cons: Too many panels could be confusing for the user to navigate. The space is also often wasted since user does not need to use `rpreview` regularly.

Conclusion: Given that the application already has quite a number of panels (User Box, Command Box, Result Box, View Box and List Box), alternative 1 is chosen to minimise the layout and improve on user experience.

Every time the user makes an edit, the Result Box will display a confirmation message, and the View Box will shows the edited details. Hence, user will not need to call `rpreview` after each edit just to confirm that the command is successfully executed and error-free. As such, `rpreview` will not be a command that is used often, and the benefits of a clean UI outweighs the compromise made when additional action to close the pop-up window is required.

4.6.2. Generating a `.pdf` file from a resume: `rgen`

Implementation

This feature utilises the external **Apache Pdf Box** library. When using `rgen`, the user could specify the desired name of the generated file, which will be saved in the root directory of the project.

The executing of `rgen` is facilitated by an addition `PdfBuilder` class, which implements the following main operations:

- `addPage()` - starts a new page in the `.pdf` file. This method is called dynamically to fit the content of the resume to be generated.
- `addPersonalDetails()` - adds the relevant resume and user details to the `.pdf` file.
- `addInternships()`, `addProjects()` and `addSkills()` - adds the resume content to the `.pdf` file in the same order that they appear in the resume. If a section is empty, it will not be added to the output file.

Given below is an example usage scenario:

Step 1. The user launches **ResuMe**. After loading data from storage to the application memory, the list of `resumes` in the `ResumeBook` could either contain some resumes, or is empty.

Step 2. The user executes `rgen INDEX n/ FILE_NAME`. If the specified resume `INDEX` is out of bound, **ResuMe** throws an error message.

Step 3. The application retrieves the correct `Resume` item, create a new `.pdf` file and populates it with the corresponding items inside the `Resume`.

The following activity diagram summarises what happens when a user executes `rgen` command:

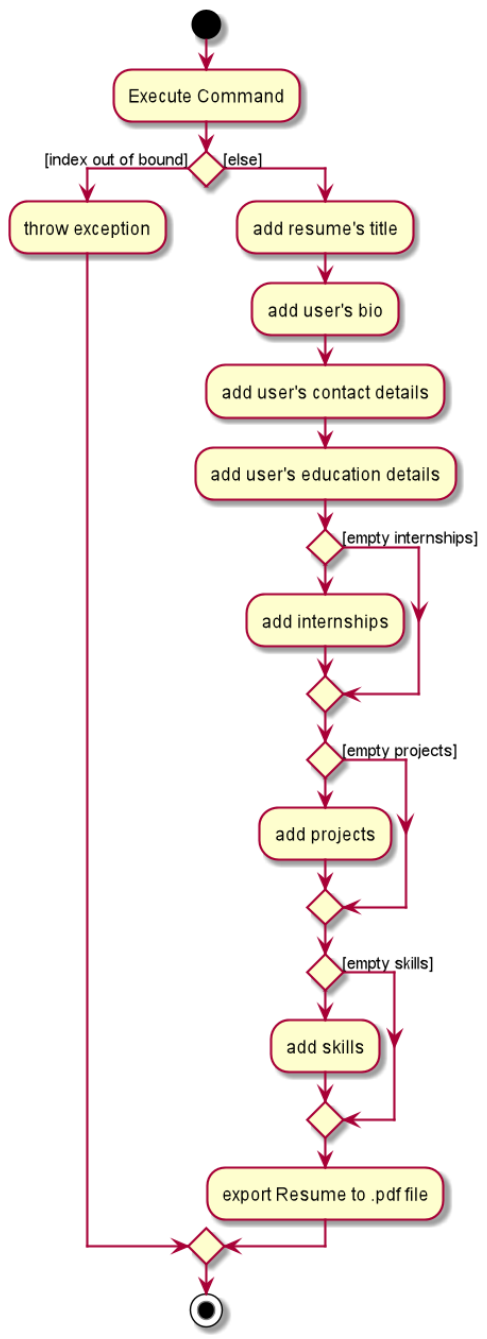


Figure 22. Activity Command for GenerateResumeCommand

Detailed steps are shown in the sequence diagram below:

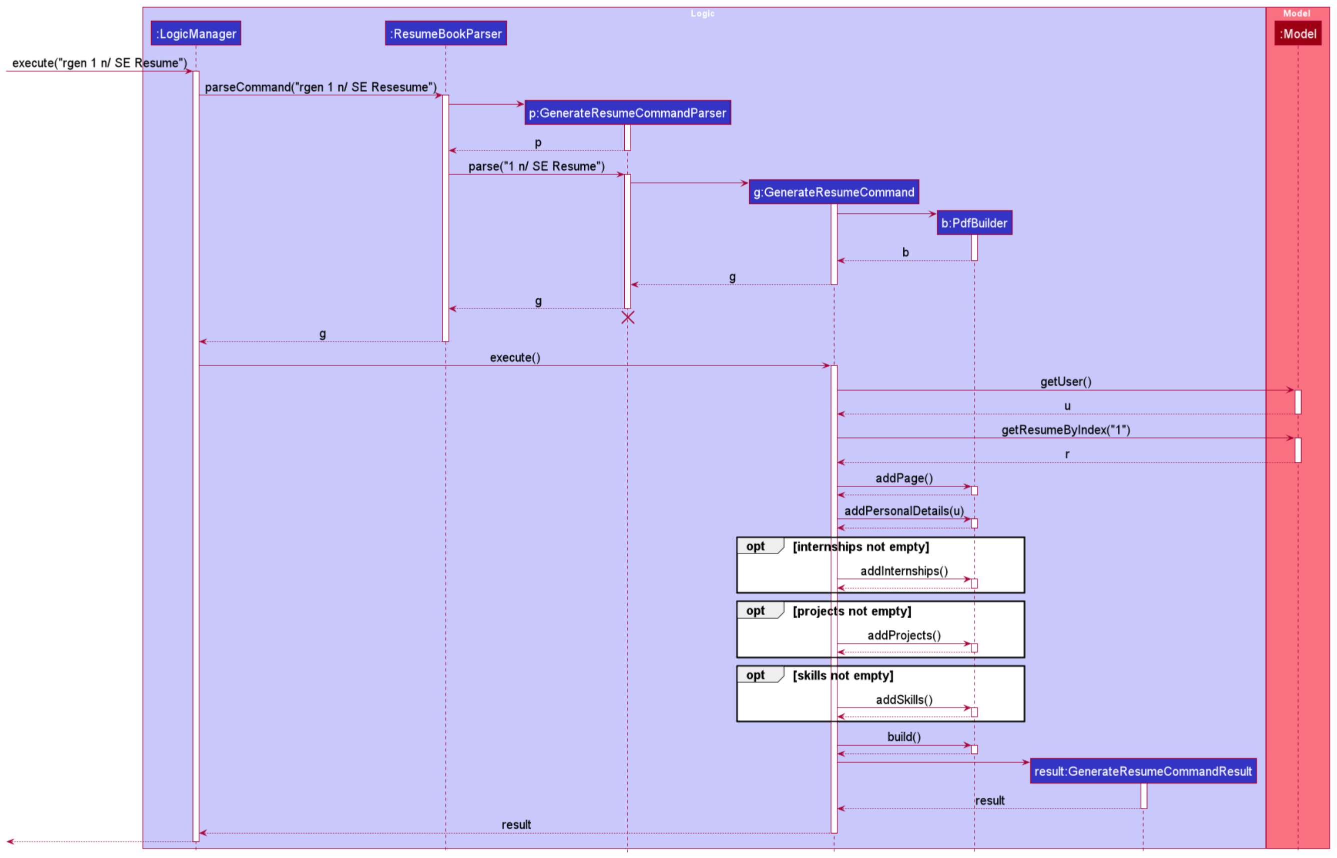


Figure 23. Sequence Diagram for GenerateResumeCommand

4.6.3. Design Considerations

Aspect: How `rgen` executes

- **Alternative 1 (current choice):** Generate `.pdf` file by iteratively adding `items` contained in `resume`.
    - Pros: Better control of the output layout, as the position and formatting of each section could be set individually. Additionaly, `rgen` is dynamic, in the sense that even without calling `rpreview` every time, the generated file will be consistent with any item update.
    - Cons: Coupled with `rpreview`, the content of a `Resume` must be read twice every time the user wishes to export.
- **Alternative 2:** Generate `.pdf` file directly from the previewed text output by `rpreview`
    - Pros: Faster and simpler `rgen`
    - Cons: Limited formatting options (font type, font size and page layout) since the whole document is now input as one long string of text. `rgen` implemented this way is also static, and might not reflect the most updated content if there are item changes after `rpreview` is called.

Conclusion: Alternative 1 is chosen because each resume is not likely to hold more than 20 items, hence the cost of reading its content twice (once during `rpreview` and once during `rgen`) is relatively small. Choosing alternative 1 will further allows the application to have more control when formatting the output file, and leaving room for potential extended features (e.g. allow user to choose from a variety of pre-defined resume templates).

{ end of `export` section written by: Pham Thuy Duong }

{ start of `undo-redo` section written by: Nguyen Minh Hoang }

4.7. Undo/Redo feature

4.7.1. Implementation

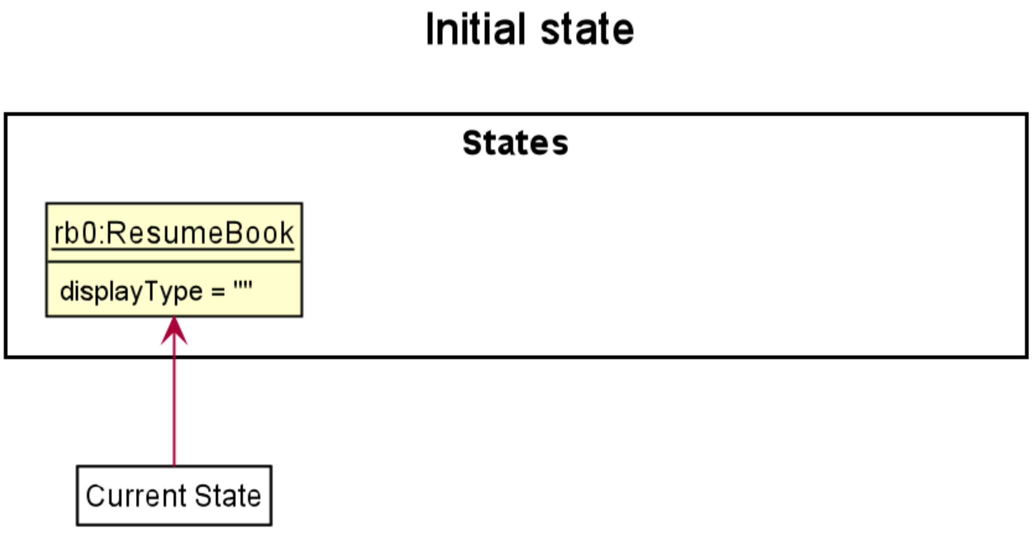
The undo/redo mechanism is facilitated by `VersionedResumeBook`. It extends `ResumeBook` with an undo/redo history, stored internally as an `ResumeBookStateList` and `currentStatePointer`. Additionally, it implements the following operations:

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

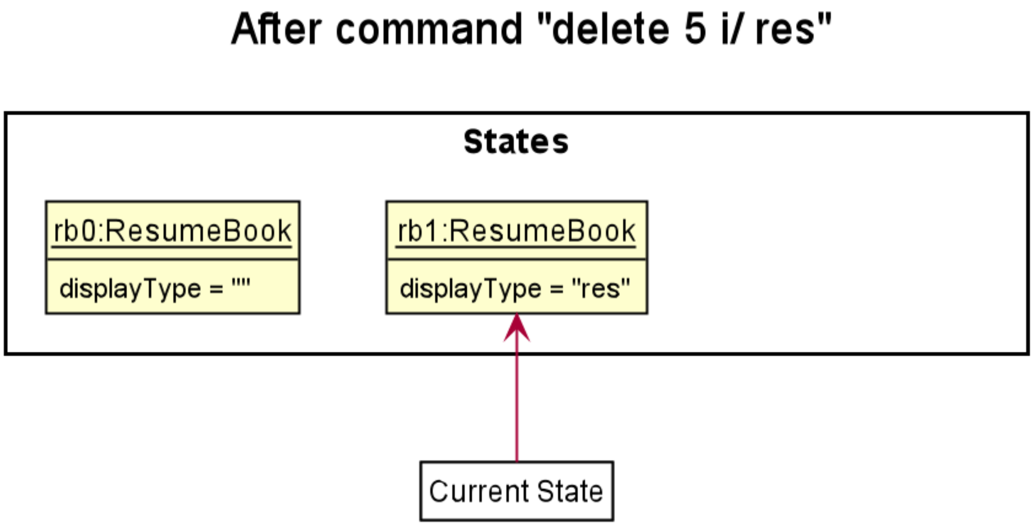
These operations are exposed in the `Model` interface as `Model#commitResumeBook()`, `Model#undoResumeBook()` and `Model#redoResumeBook()` respectively.

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

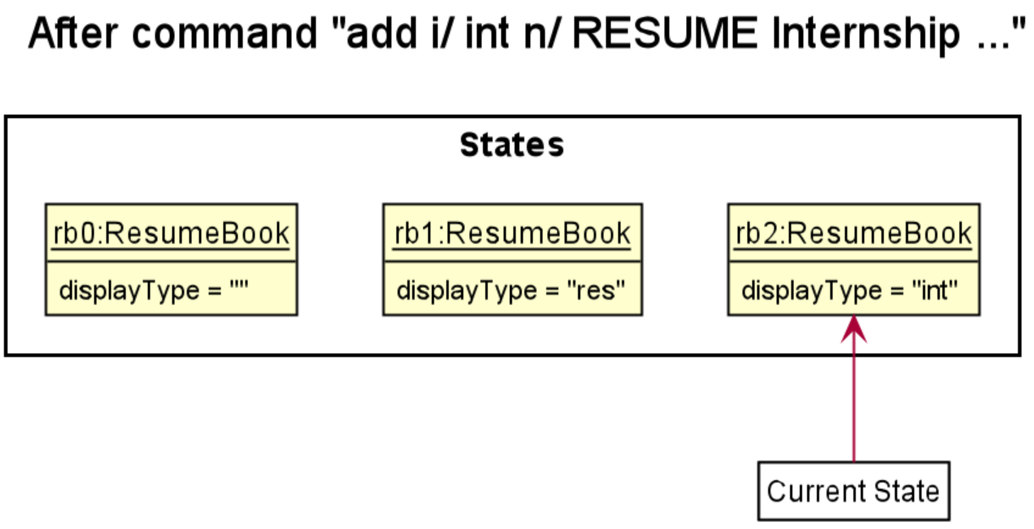
Step 1. The user launches the application for the first time. The `VersionedResumeBook` will be initialized with the initial resume book state, and the `currentStatePointer` pointing to that single resume book state. Do note that all resume books begin with its display type set to an empty string. Display type of a resume book determines which item list will be selected for display (e.g. if the display type is "res" - an alias for resume, then the list of all resumes will be selected).



Step 2. The user executes `delete 5 i/ res` command to delete the 5th resume in the resume book. The `delete` command calls `Model#commitResumeBook()`, causing the modified state of the resume book after the `delete 5 i/ res` command executes to be saved in the `resumeBookStateList`, and the `currentStatePointer` is shifted to the newly inserted resume book state. In addition, the display type of the current resume book will be set to resume.



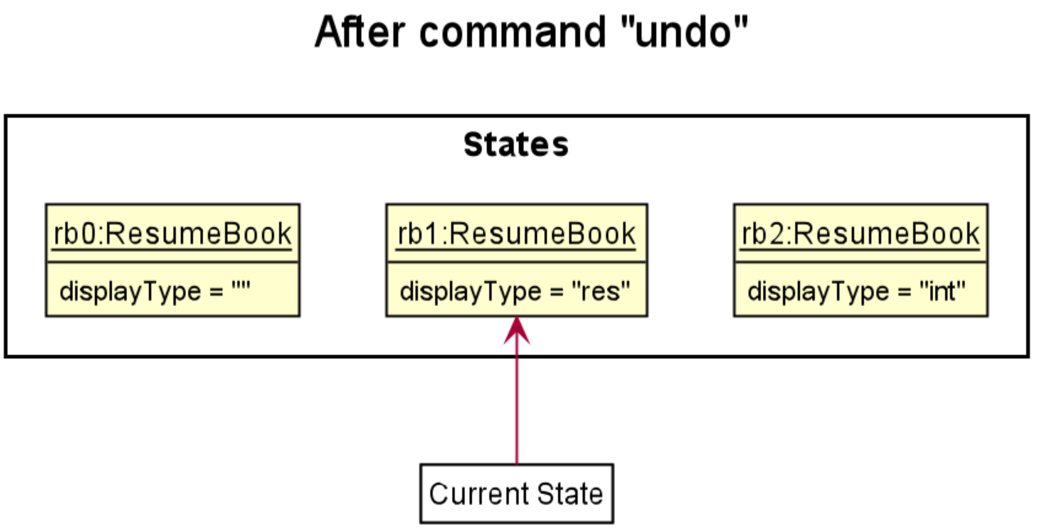
Step 3. The user executes `add i/ int n/ RESUME Internship …` to add a new internship. The `add` command also calls `Model#commitResumeBook()`, causing another modified resume book state to be saved into the `resumeBookStateList`. This resume book has its display type set to `int` - which is an alias for internship.



If a command fails its execution, it will not call `Model#commitResumeBook()`, so the resume book state will not be saved into the `resumeBookStateList`.

Step 4. The user now decides that adding the internship was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoResumeBook()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous resume book state, and restores the resume book to that state. This resume book state, however, does not has the display type that we want. Thus, `VersionedResumeBook#setItemsToDisplay()` will be called to set the current resume book's display type to that of the resume book state one to the right of the `currentStatePointer` - which corresponds to the command that just has been undone.

Note: `resumeBookStateList` stores resume book states as `ReadOnlyResumeBook`, thereby allowing these states to stay unmodified throughout their life cycle. Operations like `Model#commitResumeBook()`, `Model#undoResumeBook()`, and `VersionedResumeBook#setItemsToDisplay()` will not change any of these state at all. Immutable state like this is desirable because it gives rise to fewer bugs and fewer unexpected behaviours.



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

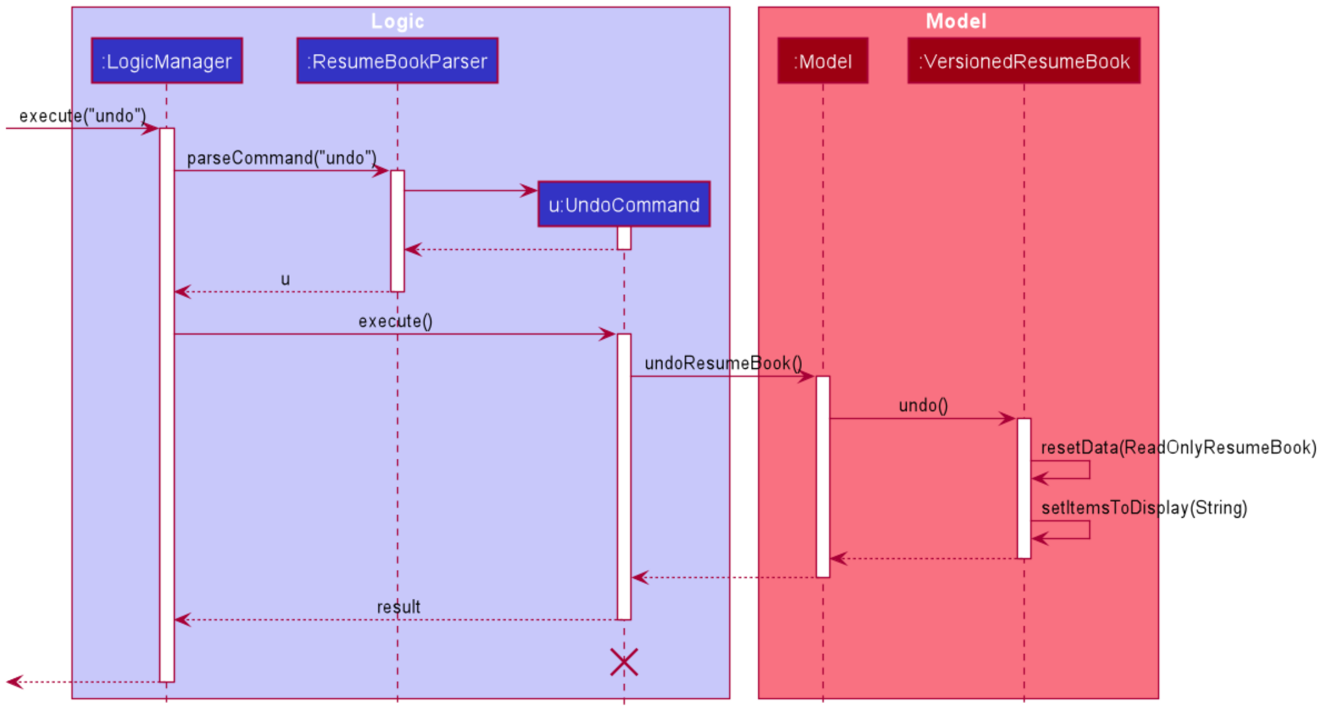


Figure 24. Sequence Diagram for UndoCommand

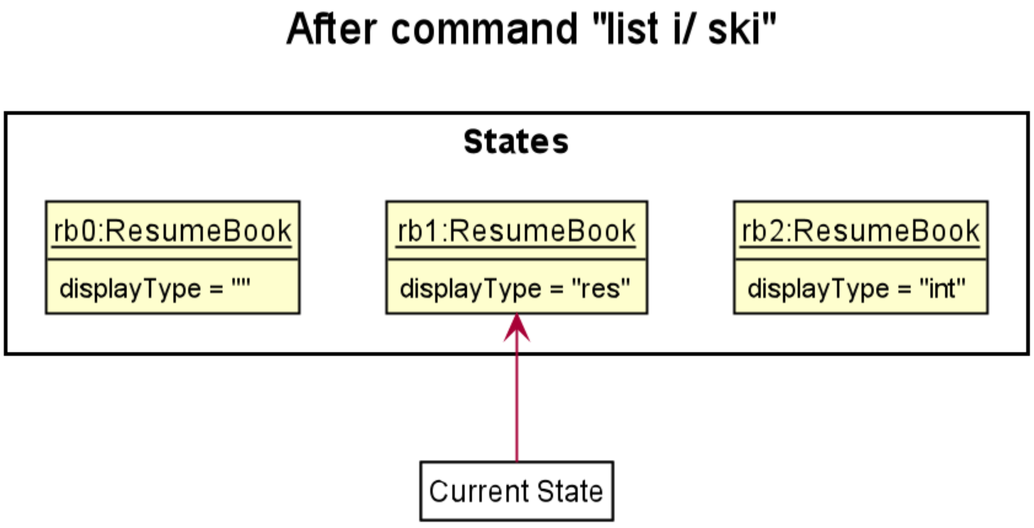
The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

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

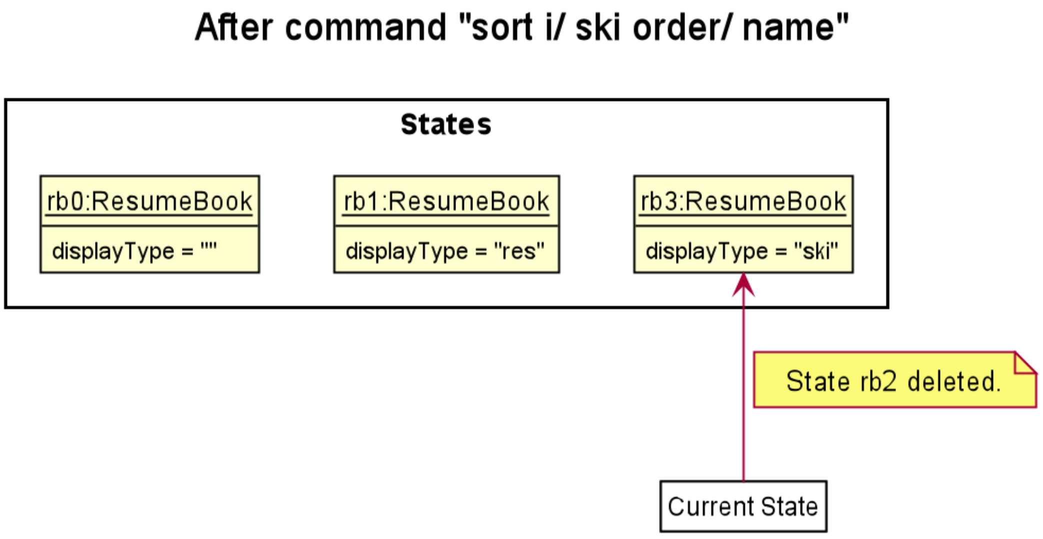
If the `currentStatePointer` is at index `resumeBookStateList.size() - 1`, pointing to the latest resume book state, then there are no undone resume book states to restore. The `redo` command uses `Model#canRedoResumeBook()` to check if this is the case. If so, it will return an error to the user rather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

Step 5. The user then decides to execute the command `list i/ ski`. Commands that do not modify the resume book, such as `list`, will usually not call `Model#commitResumeBook()`, `Model#undoResumeBook()` or `Model#redoResumeBook()`. Thus, the `resumeBookStateList` remains unchanged.

Here we also see how the stored resume book states are immutable. `list i/ ski` causes the display type of the current resume book to switch to skill (whose alias is "ski"), while the resume book state pointed to by the `currentStatePointer` still maintains its display type (which is resume).



Step 6. The user executes `sort i/ ski order/ name`, which calls `Model#commitResumeBook()`. Since the `currentStatePointer` is not pointing at the end of the `resumeBookStateList`, all resume book states after the `currentStatePointer` will be purged. We designed it this way because it no longer makes sense to redo the `add i/ int …` 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:

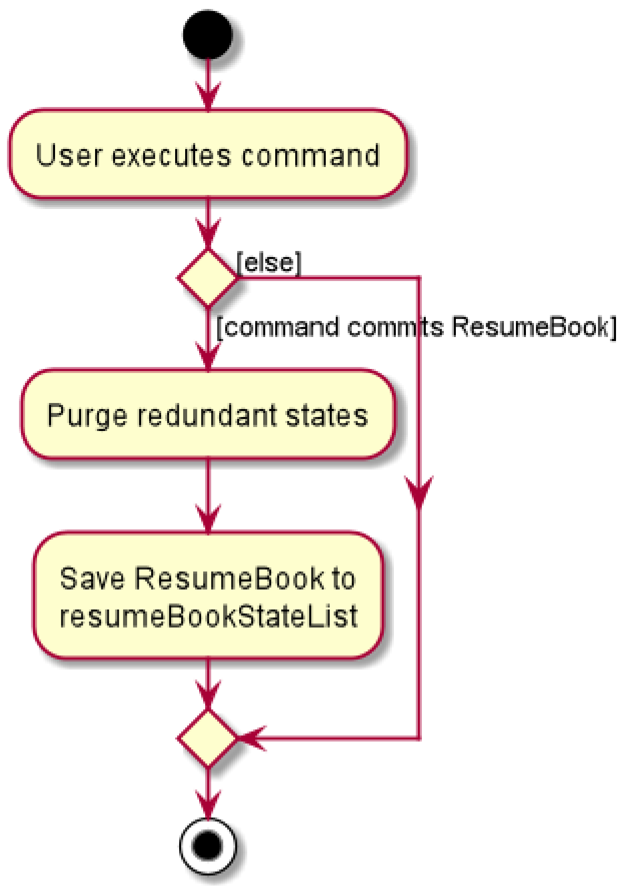

User executes command
[else]
[command commits ResumeBook]
Purge redundant states
Save ResumeBook to resumeBookStateList

Figure 25. Activity Diagram when a user executes a new command

4.7.2. Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire resume book.
  - Pros: Easy to implement. Easy to understand.
  - 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 item being deleted).
  - Cons: We must ensure that the implementation of each individual command is correct. It is further complicated by the fact that there is an `add`, `delete`, `edit`, and `sort` command for each item type. Also, Separation of Concerns Principle is violated as in essence, the `undo()` method of a command is doing more than what the command is responsible for, e.g. undoing a `delete` command is essentially performing an `add` command.

Aspect: Data structure to support the undo/redo commands

- **Alternative 1 (current choice):** Use a list to store the history of resume book states.
  - Pros: Very straightforward. Developers, even the novice ones, can easily understand and pick up if they wish to improve upon our application.
  - Cons: We need handle the `currentStatePointer` accurately; this is further complicated as we also need to reach to the undone resume book state to retrieve the correct display type for

the current resume book.

- **Alternative 2:** Use `HistoryManager` for undo/redo. `HistoryManager` will contain two stacks: `UndoStack` and `RedoStack`. We push a command into the former stack when it is executed; when an `undo` is performed, we pop the top of the `UndoStack` and store the command in the `RedoStack`.
    - Pros: We do not need to maintain a separate list, and just reuse what is already in the codebase. We also just need to store the history of commands as opposed to the entire resume book.
    - Cons: Handling of the stacks can be confusing, especially since there are commands that make no change to the model and thus are not (and should not be) stored. `edit` and `delete` require the old-versioned item to be stored as well so that it can be restored while `add` does not, thereby affecting consistency.

4.7.3. [Proposed] Improvements for Undo/Redo

A new function can be added that allows users to navigate resume book states. This is very possible because we can already store all these states and can map them to keys easily. This new function can let user traverse between resumes without needing to undo or redo too many times.

A better data structure - like a search tree - can be used in place of a list. If the above proposed change is taken into consideration, then it is desirable to improve search time for a resume book state, from worst case linear to worst case logarithmic.

{ end of `undo-redo` section written by: Nguyen Minh Hoang }

{ start of `data-encryption` section written by: Nguyen Minh Hoang }

4.8. [Proposed] Data Encryption

Data encryption can be implemented in the future versions.

Aspect: How data encryption will be implemented

- **Alternative 1**: Encrypt and store data locally.
    - Pros: Easy to implement. There are many existing encryption libraries that can be used.
    - Cons: Performance issues may arise because every call to save to storage requires encrypting a lot of data. Security threat is still present if we store the encryption key in the same machine.
- **Alternative 2**: Store data outside the user’s machine and issue them access token.
    - Pros: Data is not bound to one machine and can be restored if the user forgets their credentials. If a DBMS is used then there are additional benefits to storage functionality as well.
    - Cons: We need to generate access tokens and check them. We also need to change the current implementation to work with external storage.

{ end of `data-encryption` section written by: Nguyen Minh Hoang }

4.9. 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 4.10, “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

4.10. Configuration

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

5. Documentation

Refer to the guide here.

6. Testing

Refer to the guide here.

7. Dev Ops

Refer to the guide here.

{ start of `product-scope` section written by: Pham Thuy Duong }

Appendix A: Product Scope

Target user profile:

- is a University student looking for internship or job

- has a need to manage a significant number of personal experiences and resumes for different purposes
- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: build and manage resumes faster than a typical mouse/GUI driven app

{ end of `product-scope` section written by: Pham Thuy Duong }

{ start of User Stories section written by: Nguyen Chi Hai }

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…
`* * *`	user	manage and customise different resumes	use different resumes for different companies
`* * *`	user	create a resume file	print it
`* * *`	user	add to and remove 'items' from a certain resume	
`* * *`	user	add, edit, and remove 'items'	
`* * *`	user	preview my resume	check for mistakes in a resume before generating it
`* * *`	user	search for items containing certain keywords	find items that are relevant to my current need

Priority	As a …	I want to …	So that I can…
* *	user	see all the 'items' I've added	
* *	user	see all my resumes	manage them in a centralised location
* *	user	categorise the 'items' I've added	to ensure my resume will have 'items' of different types
* *	new user	view more information about various commands	learn to use new commands
*	careless user	undo my previous command	skip the step of manually editing or deleting them.
*	busy user	auto-format my resumes	

'items' refers to a resume field of type `project`, `skill`, `internship`, `note` *and* `resume`.

{ end of User Stories section written by: Nguyen Chi Hai }

Appendix C: Use Cases

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

Use case: Edit an item

MSS

1. User requests to list all items or only items of a specific `TYPE`
2. ResuMe shows a list of corresponding items
3. User checks for the `ID` of a specific item in the list to edit
4. User requests to edit a specific item in the list
5. ResuMe updates that item and displays edited item to user

   Use case ends.

Extensions

3a. The `ID` given by user does not match any item of type

3a1. ResuMe shows an error message

Use case: Find item(s)

MSS

1. User enters `find KEYWORD` without specifying a `TYPE`
2. ResuMe displays all items whose names contain the `KEYWORD`
3. If user enters `find -TYPE KEYWORD`
4. ResuMe displays only items of the `TYPE` whose names contain the `KEYWORD`

   Use case ends.

Extensions

1a. None of the items contain the `KEYWORD`.

1a1. ResuMe shows an error message.

Use case: Delete an item

MSS

1. User requests to list all items or only items of a specific `TYPE`
2. ResuMe shows a list of corresponding items
3. User checks for the `ID` of a specific item in the list to delete
4. User requests to edit a specific item in the list
5. ResuMe deletes that item from the list and displays deleted item to user

   Use case ends.

Extensions

- 2a. The list is empty.

  Use case ends.
- 3a. The given `ID` is invalid.
  - 3a1. ResuMe shows an error message.
  - Use case resumes at step 2.

{More to be added}

{ start of `nfr` section written by: Pham Thuy Duong }

Appendix D: Non Functional Requirements

1. The system should work on any mainstream OS as long as it has Java `11` or above installed.
2. The system should work on both 32-bit and 64-bit environments.
3. The system should be able to support at least 10 resumes per user, 20 items per resume, and 200 resume items in total.
4. Data should be stored locally such that users can access them (either through the application or by directly accessing the data file) without any network connection.
5. The system should response within two seconds for a normal item query, and five seconds for a pdf generation request.
6. The system should be usable by a novice who has never used a resume management application before.
7. A user with some familiarity with Command Line Interface (CLI) should be able to accomplish most of the tasks faster using commands than using the mouse to navigate the Graphic User Interface (GUI).
8. The product is not required to handle the printing of resumes.

{ end of `nfr` section written by: Pham Thuy Duong }

{ start of `glossary` section written by: Pham Thuy Duong }

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Content items

`Internship`, `Project`, `Skill`

{ end of `glossary` section written by: Pham Thuy Duong }

Appendix F: Product Survey

Product Name

Author: …

Pros:

- …
- …

Cons:

- …
- …

{ start of Manual Testing section written by: Nguyen Chi Hai }

Appendix G: Instructions for Manual Testing

Given below are instructions to test the app manually.

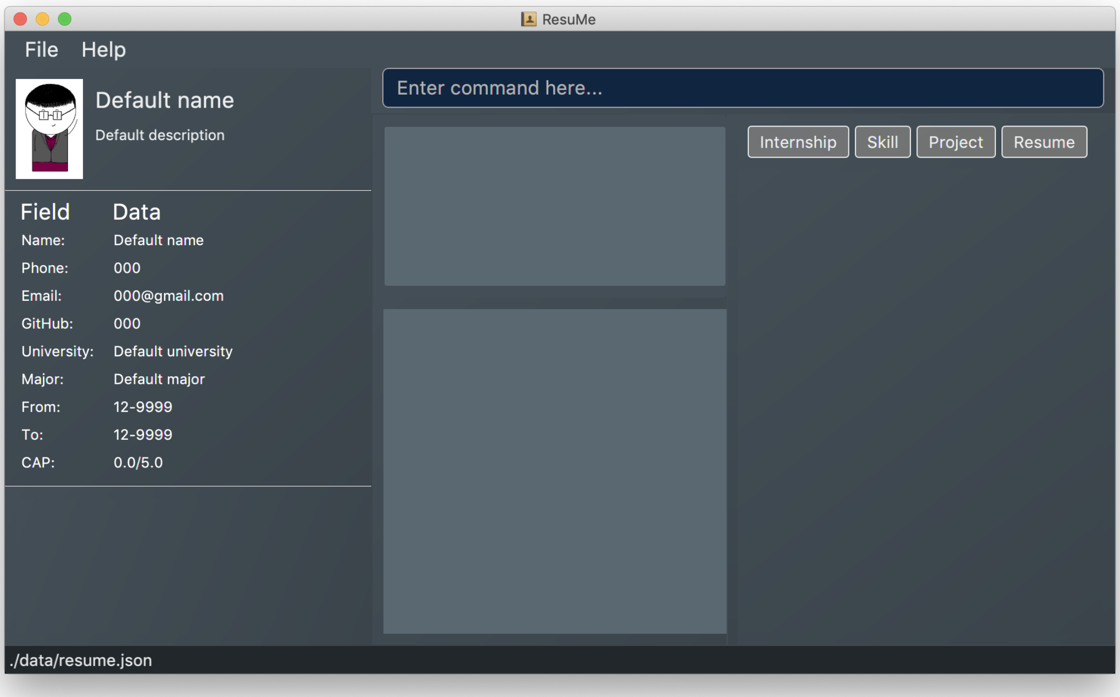
> These instructions only provide a starting point for testers to work on; testers are expected to do more *exploratory* testing.

G.1. Launch and Shutdown

1. Initial launch

   a. Download the jar file and copy into an empty folder

   b. Double-click the jar file
   Expected: You are supposed to see this GUI, with default values for user details, and two empty resumes, enter `list i/ res` to check it out. The window size may not be optimum.

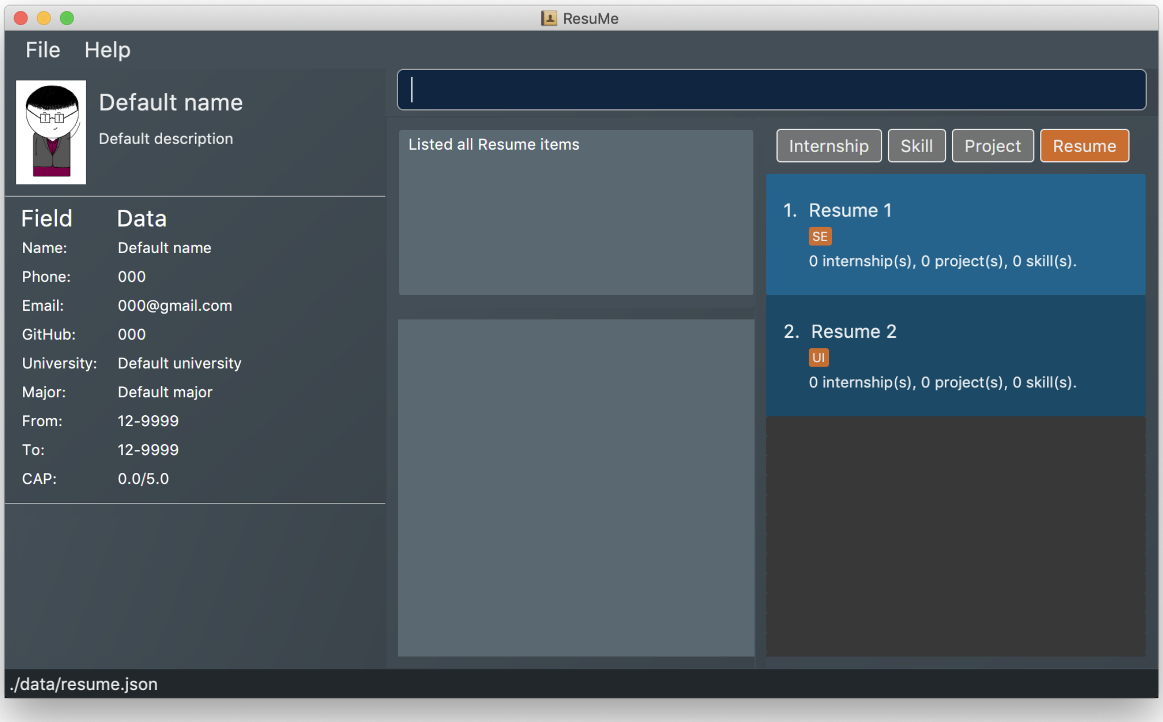


Default name
Default description

Field	Data
Name:	Default name
Phone:	000
Email:	000@gmail.com
GitHub:	000
University:	Default university
Major:	Default major
From:	12-9999
To:	12-9999
CAP:	0.0/5.0

./data/resume.json

Figure 26. Initial view when jar file runs



Default name
Default description

Field	Data
Name:	Default name
Phone:	000
Email:	000@gmail.com
GitHub:	000
University:	Default university
Major:	Default major
From:	12-9999
To:	12-9999
CAP:	0.0/5.0

Listed all Resume items

1. Resume 1
   SE
   0 internship(s), 0 project(s), 0 skill(s).
2. Resume 2
   UI
   0 internship(s), 0 project(s), 0 skill(s).

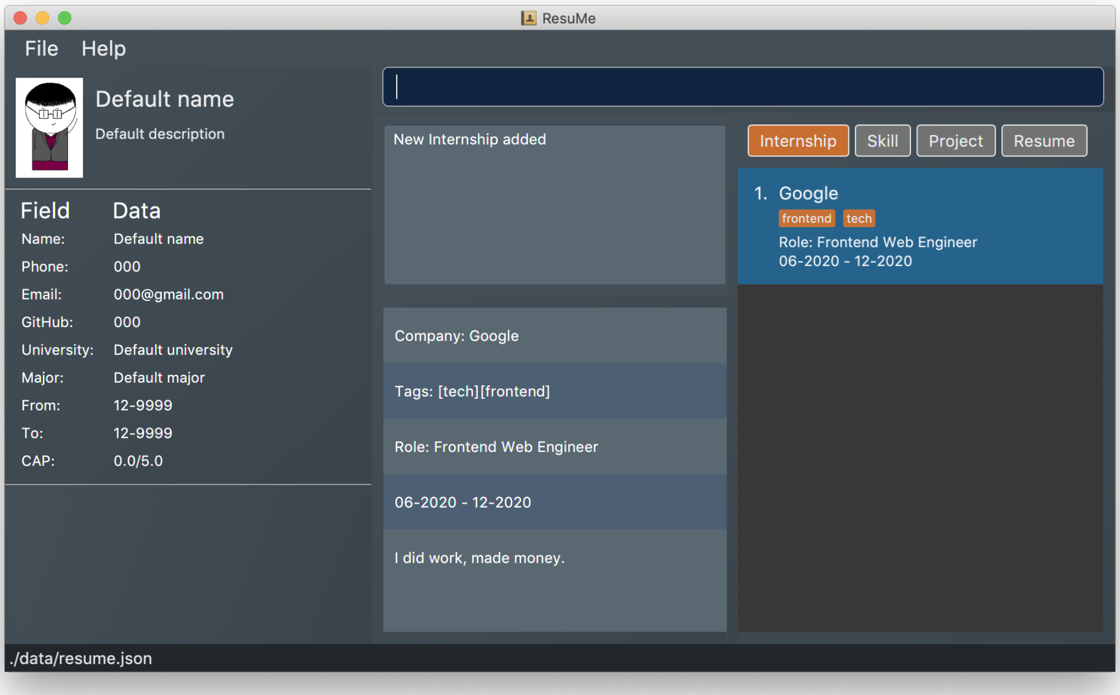
./data/resume.json

Figure 27. Expected view after running the command `list i/ res` *to show all resumes*

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.
3. Shutting down the program
   a. To exit the program, you can enter the command `exit` or click the red close button, don't worry all your data is already saved.

G.2. Adding items

1. Adding an internship.
   a. This test case assumes that the internship list is empty. It does not have to be, but there must not be a duplicate internship with the internship we're about to add.
   b. Test case **adding an internship**: `add i/ int n/ Google r/ Frontend Web Engineer f/ 06-2020 t/ 12-2020 d/ I did work, made money. #/ frontend #/ tech`
      Expected: An internship item is added to the list of internships and the list view changes to list of internships. Data box is filled with the data of the internship that you just added.



Default name
Default description

Field	Data
Name:	Default name
Phone:	000
Email:	000@gmail.com
GitHub:	000
University:	Default university
Major:	Default major
From:	12-9999
To:	12-9999
CAP:	0.0/5.0

New Internship added

Company: Google

Tags: [tech][frontend]

Role: Frontend Web Engineer

06-2020 - 12-2020

I did work, made money.

1. Google
frontend tech
Role: Frontend Web Engineer
06-2020 - 12-2020

./data/resume.json

c. Test case **adding an duplicate internship** (do this after part b): `add i/ int n/ Google r/ Frontend Web Engineer f/ 06-2020 t/ 12-2020 d/ I did work, made money. #/ frontend #/ tech`
Expected: No internship is added is added. Error details shown in the status message. Status bar remains the same.

d. Test case **adding an internship with compulsory parameters missing**: `add i/ int n/ Shopee f/ 06-2020 t/ 12-2020 d/ I did work, made money. #/ frontend #/tech`
Expected: No internship is added is added. Error details shown in the status message. Status bar remains the same.

e. Other incorrect add internship commands to try: commands where other compulsory parameters such as `to`, `from`, and `description` are missing.
Expected: Similar to previous.

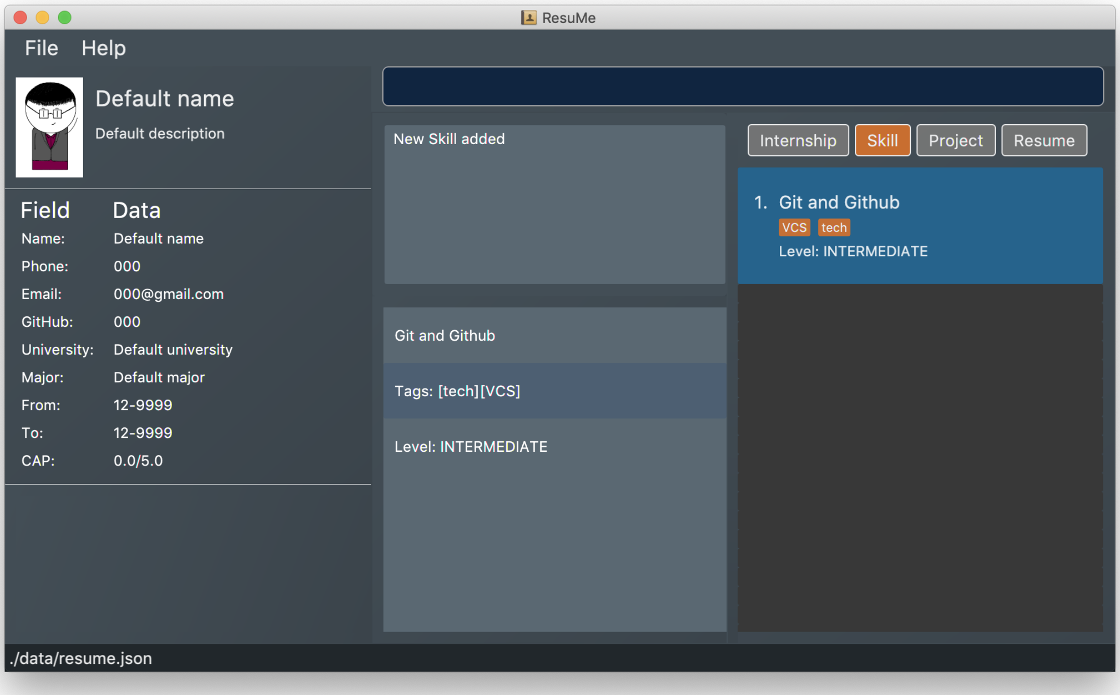
2. Adding items of other types

    a. Simlar to adding internship above, please make sure that you follow our User Guide for the correct formatting of the commands if you are expecting a positive test case.

G.3. Deleting items

1. Deleting a skill.

    a. This test case assumes that there are skills already in the skill list. Run `list i/ ski` to check. If there aren't any skills (shown below), follow G.2 above to add some skills.



Default name
Default description

Field	Data
Name:	Default name
Phone:	000
Email:	000@gmail.com
GitHub:	000
University:	Default university
Major:	Default major
From:	12-9999
To:	12-9999
CAP:	0.0/5.0

New Skill added

Git and Github
Tags: [tech][VCS]
Level: INTERMEDIATE

1. Git and Github
VCS tech
Level: INTERMEDIATE

./data/resume.json

   b. Test case **deleting an internship**: `delete 1 i/ ski`
      Expected: An skill item is deleted from the skills and the data box is filled with the data of the skill that you just deleted.

   c. Test case **delete a skill with wrong index**: `delete 0 i/ ski`
      Expected: No skill is deleted. Error details shown in the status message. Status bar remains the same.

   d. Other incorrect add skill commands to try: `delete 1 i/skill`, `delete -1 i/ski`, `delete abc i/ski` Expected: Similar to previous.

2. Deleting items of other types

   a. Simlar to deleting skills above, please make sure that you follow our User Guide for the correct formatting of the commands if you are expecting a positive test case.

{ end of Manual Testing section written by: Nguyen Chi Hai }