Nguyen Chi Hai - Project Portfolio

PROJECT: ResuMe

Overview

ResuMe is a resume managing application made specifically for University students by University Students to allow then to add and manage their different experiences, edit personal data, and create the resume of their dreams in just a few commands.

Summary of contributions

Code contributed:

Link to my code [Code]

Contribution to team based tasks:

Morphed Commands in overall architecture

- What it means: Commands that are item dependent are now separated to their own separate class according to the item type that they modify, to be created by parser and calls on changes in the model.
- Justification: This is to make sure that the logic of each excution of each command class is independent from the type that it is linked to. Eg. ListCommand is split into ListResumeCommand and ListSkillCommand... ListXYZCommand will show all XYZ without having to check what Item type is passed into it.

Set up the foundation for tests

Created typical classes, item builders, assert command success and failure to aid the team in the setting up of tests.

Manage releases

managed releases for release v1.3 and v1.4.

Enhancements implemented:

Preview a resume: `rpreview`

- What it does: Allows users to preview their fleshed out resume in the application before generating it.
- Justification: This was done to provide users with a ability to go check their resumes for any typos or make sure that all their relevant experiences have been added to the resume before actually making it into a PDF file. This will save users' time as generating the PDF does take some time, and decrease the clutter as users would not have to generate many files.
- Highlights: This feature wrote the framework to to pass data back to UI and allow the creation of the window with the text data. This feature made enhancing the `help` command much easier due to similar logic and styling.

List all items: `list`

- What it does: Allows users to view all items of a certain type.
- Justification: This is done so that users know what items they have to pick and choose from to add into their resume.
- Highlights: Bigger rework on UI and CommandResult so that CommandResult passes up the type currently showing for UI to update and give visual feedback to users.

Clear all data: `clear`

- What it does: Allows users clear all their data to start fresh.
- Highlights: This feature was morphed from AddressBook so that it can work with our data types, undo, and redo.

Wrote tests for `add`, `clear`, `done`, `list`, `rpreview`, `view`, `JsonAdaptedInternship`, `EditUserCommand`...

Review/mentoring contributions:

- Took part in reviewing of many PRs (Pull requests #234, #159, #327 and more)
- Took part in fixing bugs in application to make sure app runs wells as a whole(Pull requests #252, #331, #228, #148 and more)

Contributions to the User Guide

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

{Start of extract 1: Introduction}

Introduction

Hello fellow University friends! Have you ever found writing resumes for job and internship applications a hassle? Have you ever dreamt about an app that can help you manage different resumes for different companies? Have you ever wished for a more streamlined **Command Line Interface** application that can match your efficiency and typing prowess?

If this sounds like you, **ResuMe** is *the* resume managing application for you! Made by University students for University students with *looovee*, this wonder understands your struggle and strives to cater to your specific resume needs and job application woes.

{End of extract 1: Introduction}

{Start of extract 2: List Command}

Listing all items : `list`

Lists items in the storage.

Format: `list i/ TYPE`

TIP

- Listed items are in short form, only showing their index, `name`, `tags` and a short summary. To view items in full details, use view.
- The type of items listed will light up in orange.

NOTE

`note` list is always showing on the bottom left of the application. `list i/ note` command will show the entire `note` lis, this is to be used in conjunction with `find` for `note` to show all `note` items after you have filtered the list.

Examples:

- `list i/ res`: Lists all `resume` items.

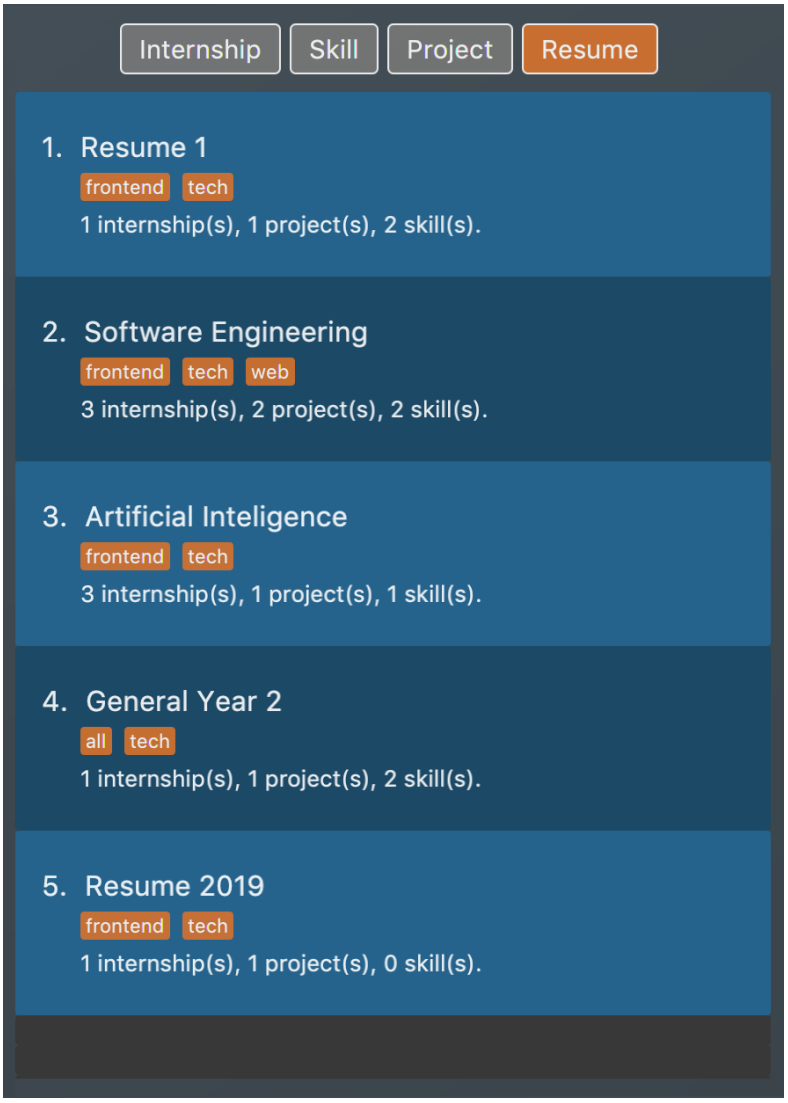

Figure 1. List of all resume items

- `list i/ int`: Lists all `internship` items.

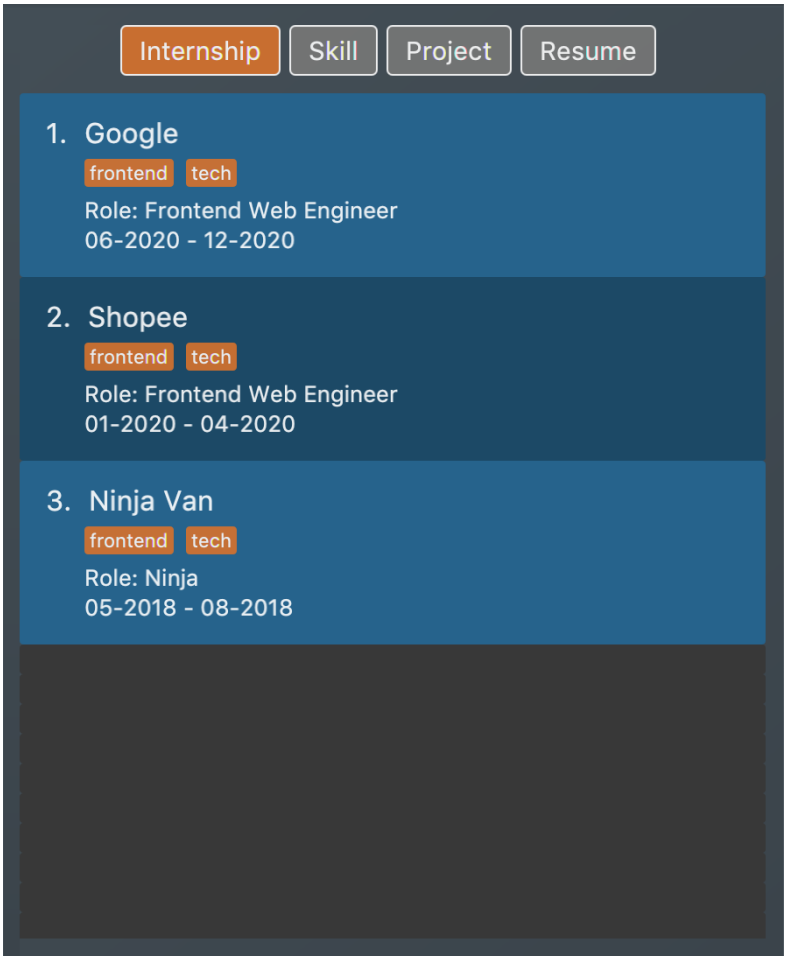

Figure 2. List of all internship items

- `list i/ note`: Lists all `note` items.

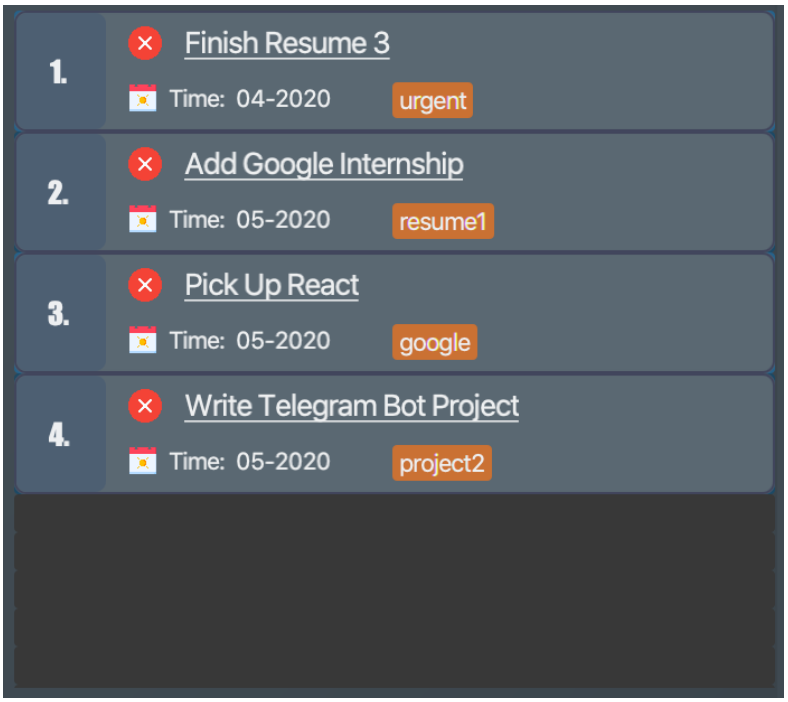

Figure 3. List of all note items

{End of extract 2: List Command}

{Start of extract 3: Resume Preview Command}

Previewing a resume: `rpreview`

Previews a `resume` in text format in a pop-up window.

Format: `rpreview RESUME_INDEX`

NOTE	• `RESUME_INDEX` is the index of the `resume` seen when `list i/ res` is called.
WARNING	• `description` of `internship` and `project` is automatically separated into bullet points when the program detects full sentences marked by a full stop.

Example: Let's try out the following commands!

1. `list i/ res`
2. `rpreview 2`

Outcome:

1. The first command lists out all `resume` items. Assuming that you want to preview the second `resume` in the list box.

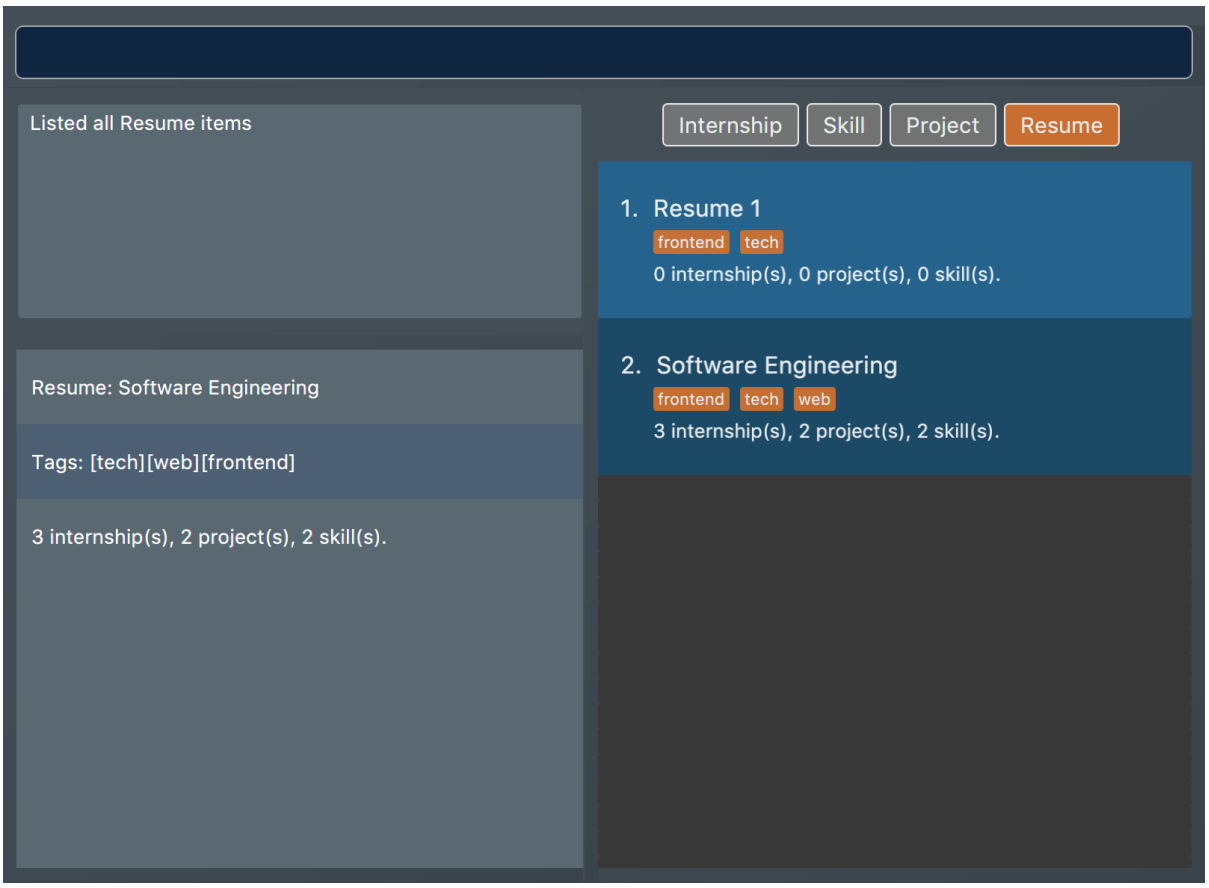

Figure 4. List of all resume items

2. The second command will preview the `resume` at index 2 named "Software Engineering". A pop-up window will be opened, featuring a text-based view of the content of "Software Engineering". The screenshots of the results are as shown below:

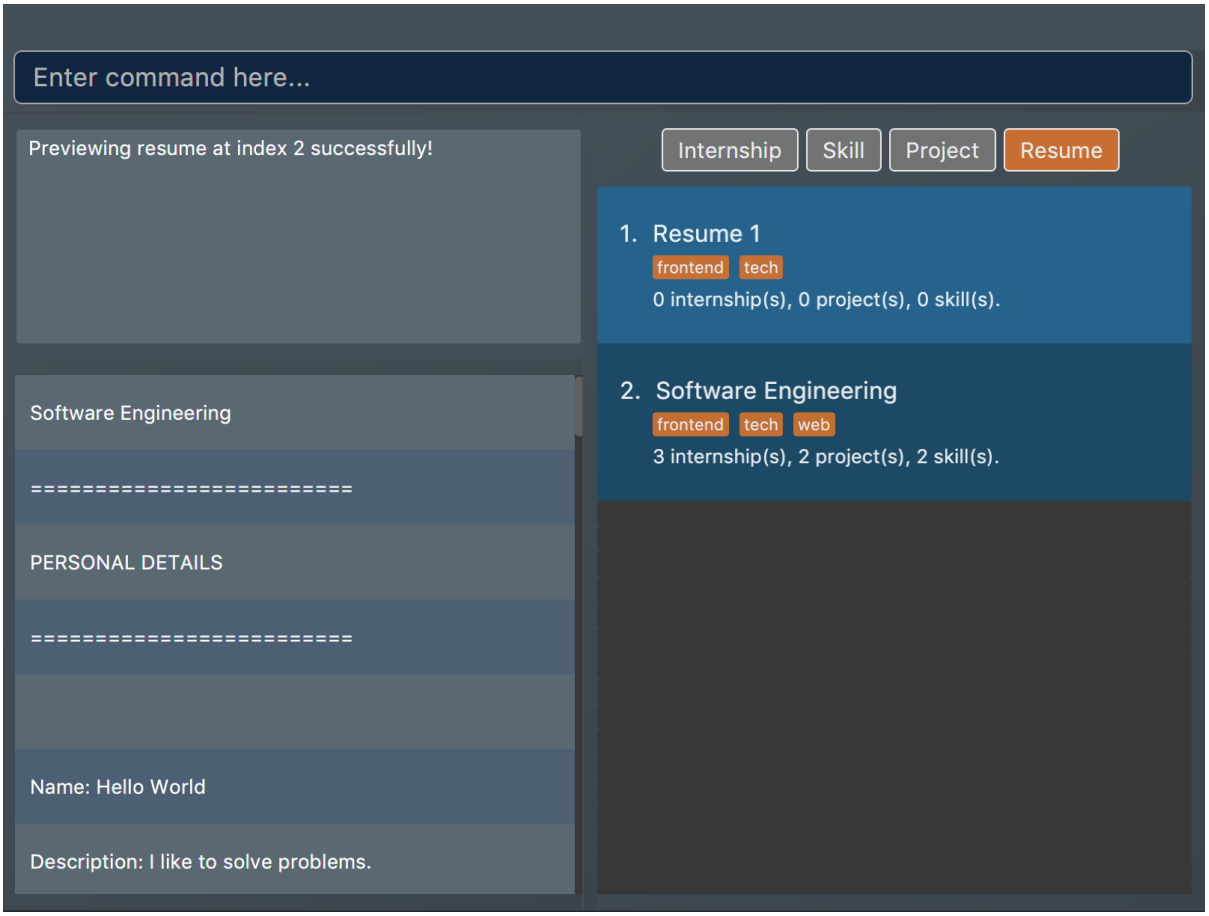

Figure 5. Application view after rpreview is executed

Figure 6. Preview pop-up window

{End of extract 3: Resume Preview Command}

{Start of extract 4: Clear Command}

Clearing all data: `clear`

Clears all data from **ResuMe**. Empties all data in the resume book, user information is replaced with default user information.

Format: `clear`

NOTE This command can be undone.

{Start of extract 4: Clear Command}

{Start of extract 5: Command Summary}

Command Summary

This is a summary of all available commands for your reference.

General commands

These are commands that have consistent format regardless of item type.

Command	Format
Clear	`clear`
Delete	`delete INDEX i/ TYPE`
Done	`done INDEX`
Exit	`exit`
Find	`find KEYWORD [MORE_KEYWORDS]… i/ TYPE`
Help	`help OPTION`
List	`list i/ TYPE`
Me	`me [dp/ FILE_PATH] [n/ NAME] [d/ DESCRIPTION] [p/ PHONE_NUMBER] [e/ EMAIL] [g/ GITHUB] [u/ UNIVERSITY] [m/ MAJOR] [f/ FROM] [t/ TO] [c/ CURRENT_CAP MAX_CAP]`
Redo	`redo`
Sort	`sort i/ TYPE order/ SORT_WORD [reverse/ TRUE_OR_FALSE]`
Undo	`undo`
View	`view INDEX i/ TYPE`

Item-specific commands

These are commands whose format varies depending on item type.

Command	Type	Format
Add	Internship	`add i/ int n/ COMPANY_NAME r/ ROLE f/ FROM t/ TO d/ DESCRIPTION [#/ TAG]…`
	Project	`add i/ proj n/ PROJECT_NAME t/ TIME w/ WEBSITE d/ DESCRIPTION [#/ TAG]…`
	Skill	`add i/ ski n/ SKILL_NAME l/ LEVEL [#/ TAG]…`
	Resume	`add i/ res n/ RESUME_NAME [#/ TAG]…`
	Note	`add i/ note n/ NOTE_NAME t/ TIME [#/ TAG]…`

Command	Type	Format
Edit	Internship	`edit i/ int [n/ COMPANY_NAME] [r/ ROLE] [f/ FROM] [t/ TO] [d/ DESCRIPTION] [#/ TAG]…`
	Project	`edit i/ proj [n/ PROJECT_NAME] [t/ TIME] [w/ WEBSITE] [d/ DESCRIPTION] [#/ TAG]…`
	Skill	`edit i/ ski [n/ SKILL_NAME] [l/ LEVEL] [#/ TAG]…`
	Resume	`edit i/ res [n/ RESUME_NAME] [#/ TAG]…`
	Note	`add i/ note [n/ NOTE_NAME] [t/ TIME] [#/ TAG]…`

Resume commands

These are commands specific to `resume` items.

Command	Format
Edit Resume	`redit RESUME_INDEX TYPE/ [ITEM_ID…] [MORE_TYPE/ [ITEM_ID…]]…`
Generate Resume	`rgen RESUME_INDEX n/ FILE_NAME`
Preview Resume	`rpreview RESUME_INDEX`
Tag Pull	`tagpull RESUME_INDEX [#/ TAG]`

{End of extract 5: Command Summary}

Contributions to the Developer Guide

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

{Start of extract 1: Overall Architecture}

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.

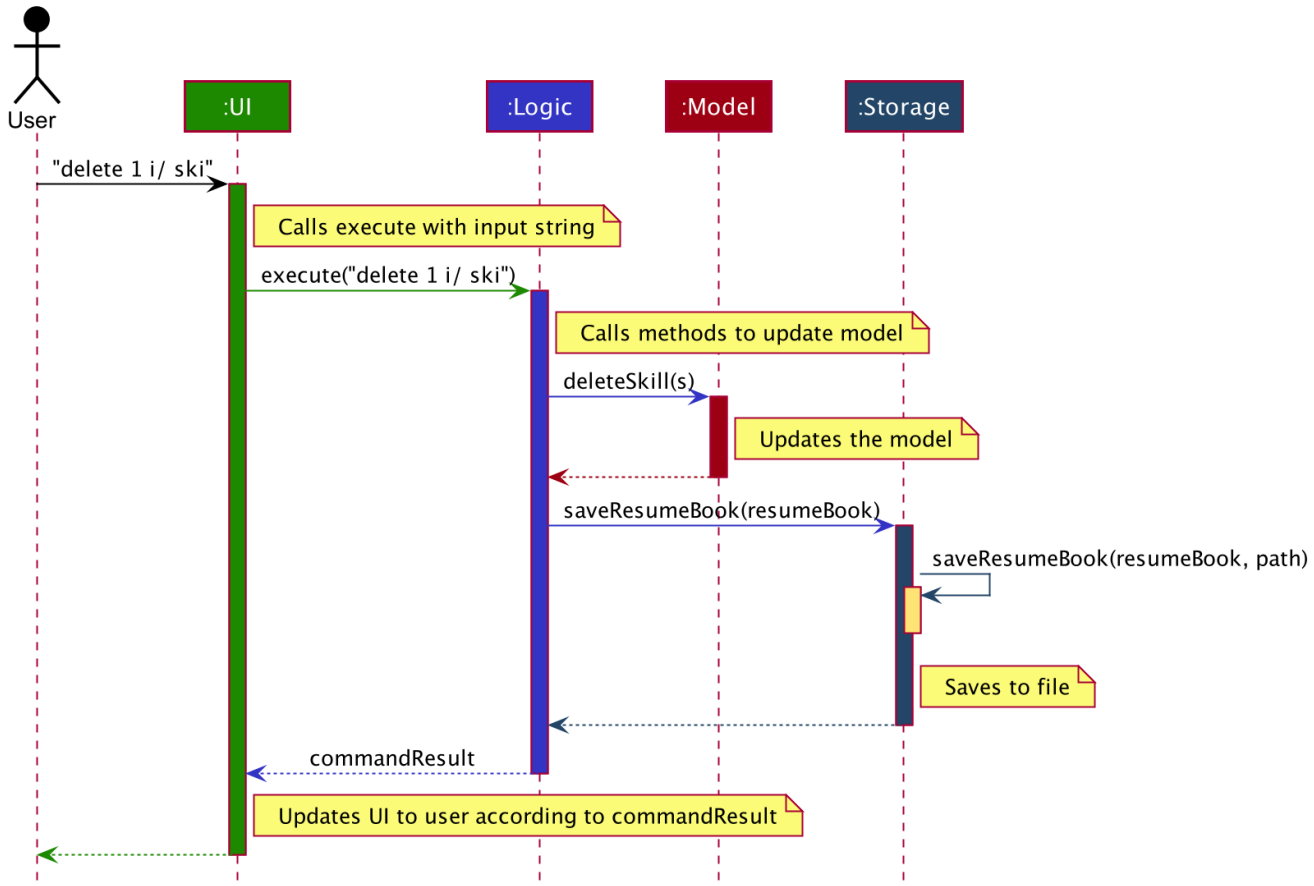

Figure 7. Component interactions for `delete 1 i/ ski` command

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 [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.

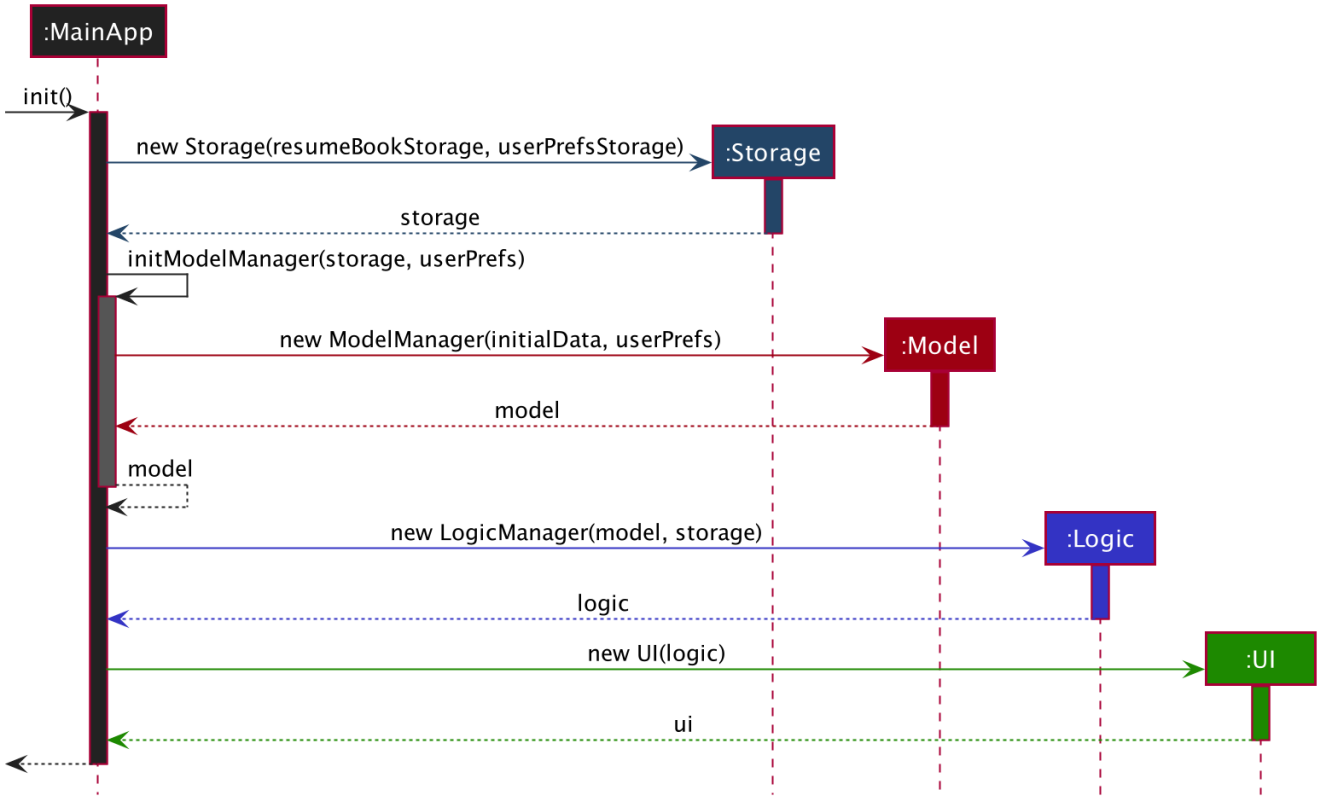

Figure 8. Component interactions for initialisation

The sections below give more details of each component.

{End of extract 1: Overall Architecture}

{Start of extract 2: Implementation of Command Classes}

Implementation of `Command` classes

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.

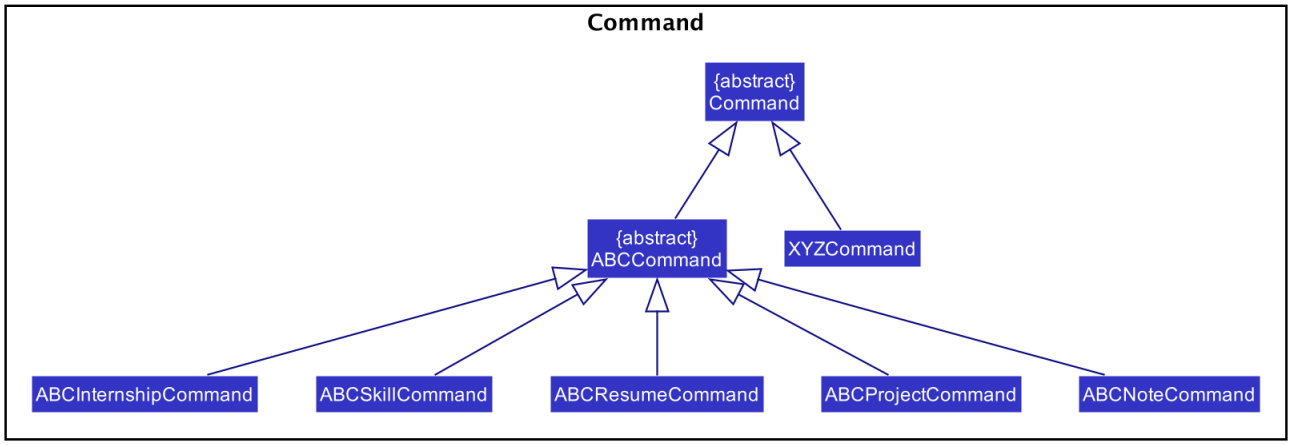

Figure 9. Component `XYZCommand` is independent of `Type` whereas `ABCCommand` is dependent on `Type`.

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.

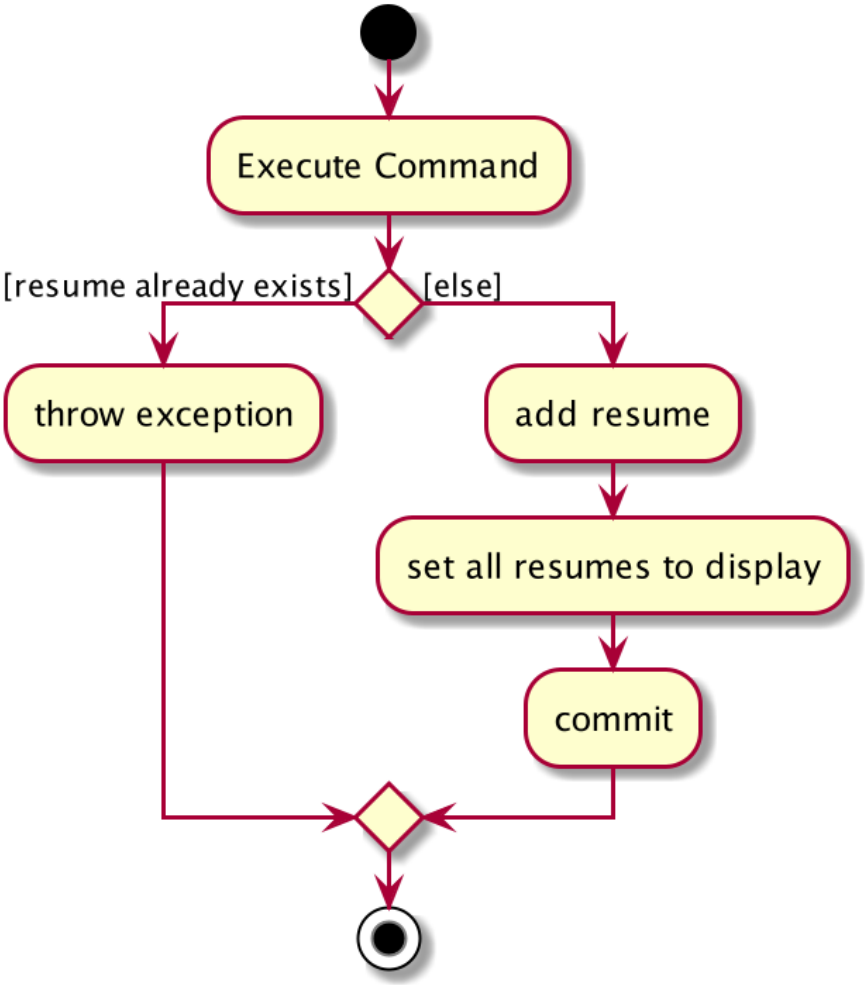

Figure 10. Activity diagram for execution of `AddResumeCommand`

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.

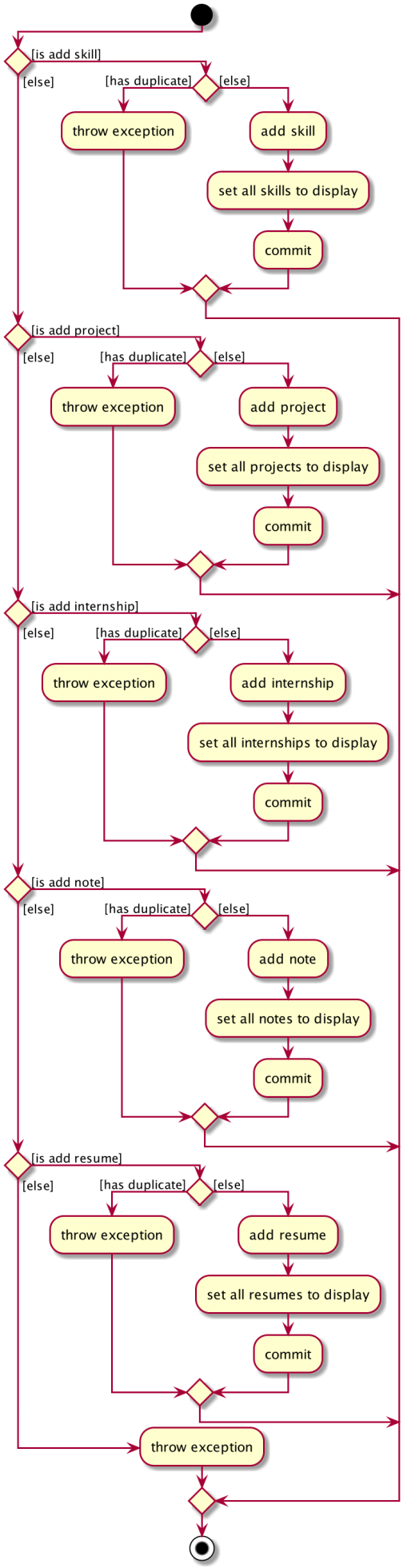
[is add skill]
[else]
[has duplicate]
[else]
throw exception
add skill
set all skills to display
commit
[is add project]
[else]
[has duplicate]
[else]
throw exception
add project
set all projects to display
commit
[is add internship]
[else]
[has duplicate]
[else]
throw exception
add internship
set all 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 11. 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`.

{End of extract 2: Implementation of Command Classes}

More contributions (not rendered for brevity):

- User Stories
- Manual Testing