Nham Quoc Hung - Project Portfolio

PROJECT: ResuMe

Overview

ResuMe is a desktop application that helps you craft and manage all your resume versions in a centralised location. Using only Command Line Interface, it aims to revolutionise the way you record your personal data, education, skills, projects and internships and use these components to construct and export a fully formatted resume with ease.

The project builds upon Address Book Level 3 (AB3), a desktop Address Book application written in Java. Our developer team consists of five Year 2 Computer Science students from the National University of Singapore. This project was completed within six weeks to constitute a major component of our CS2103T Software Engineering module.

Summary of contributions

1. Individual contributions

1.1. Contributions to Source Code

You can view my code contributions via [Functional and Test code]

- **Major enhancement**: added a user profile panel to display all user details.
    - What it does: allows user to customise and display all their essential details as a university student which can be automatically inserted into each version of their resumes upon its PDF export. This function serves only a single user of the application.
    - Justification: This feature improves the product significantly because user profile is an essential component in each resume and thus it should be made clearly visible and easily edited by the user. That is why one separate section of the application is reserved for consistent display of this feature and user details are included in every resume PDF version.
    - Highlights: This enhancement involves a key consideration which had to be made with regards to the user. Ultimately, the team decided to manage a single user profile only as our application envisions to support only a single user in managing his/her multiple resumes. At the same time, it is relatively UI intensive which took some time to get used to.
    - Credits: `https://docs.oracle.com/javafx/2/ui_controls/table-view.htm` (#80)
- **Major enhancement**: added a note taking feature for user to record notes or set simple reminders when working on his/her resumes.
    - What it does: allows user to create simple notes which will be displayed on start-up. Each note has a status of `done` or `not done` so user can decide when to remove it.

- Justification: This feature improves the product as it enhances user experience overall. As crafting their resumes and summarising their skills, projects or internship experiences can take time, even requires brainstorming sometimes, the note taking feature allows user to set simple reminders so they can be more organised when pausing or resuming their work. This is in line with our vision of the application through which we want to make the process of managing and creating resumes more of a personal learning journey through an activity called `experience logging`.
- Highlights: This feature, though appearing simple, requires a look through of the entire application code structure as it involves most components including Ui, Logic, Model and Storage.
- Contribution: #159

1.2. Contributions to documentation:

- Updated README (#173)
- Updated the User Guide to reflect my feature enhancements. Sections contributed include:
  - Editing user profile: `me`
  - Finding items by name: `find`
  - Marking a note as done: `done`
  - Exiting the program: `exit`
- Updated the Developer Guide to reflect my feature enhancements. Sections contributed include:
  - Ui Component
  - Me feature
  - Appendix C: Use Cases

2. Contributions to team-based tasks:

2.1. Ui Refactoring

- What it means: Updated Ui for **ResuMe** to match our team's design and reduce potential for Ui breaks (#135, #336)

2.2. Testing

- Wrote additional tests for existing features to increase coverage (#285, #328, #300)

2.2 Reviewing contributions:

- PRs reviewed with non-trivial review comments (#320, #260)
- Reported bugs and suggestions for other teams in class (examples: 1, 2)

Contributions to the User Guide (Extracts)

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: me}

Editing user profile: `me`

Edits and updates user's display profile.

Format: `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]`

NOTE	A user profile contains the following fields: `Display Picture`, `Name`, `Description`, `Phone`, `Email`, `Github`, `University`, `Major`, `From`, `To`, `CAP`.

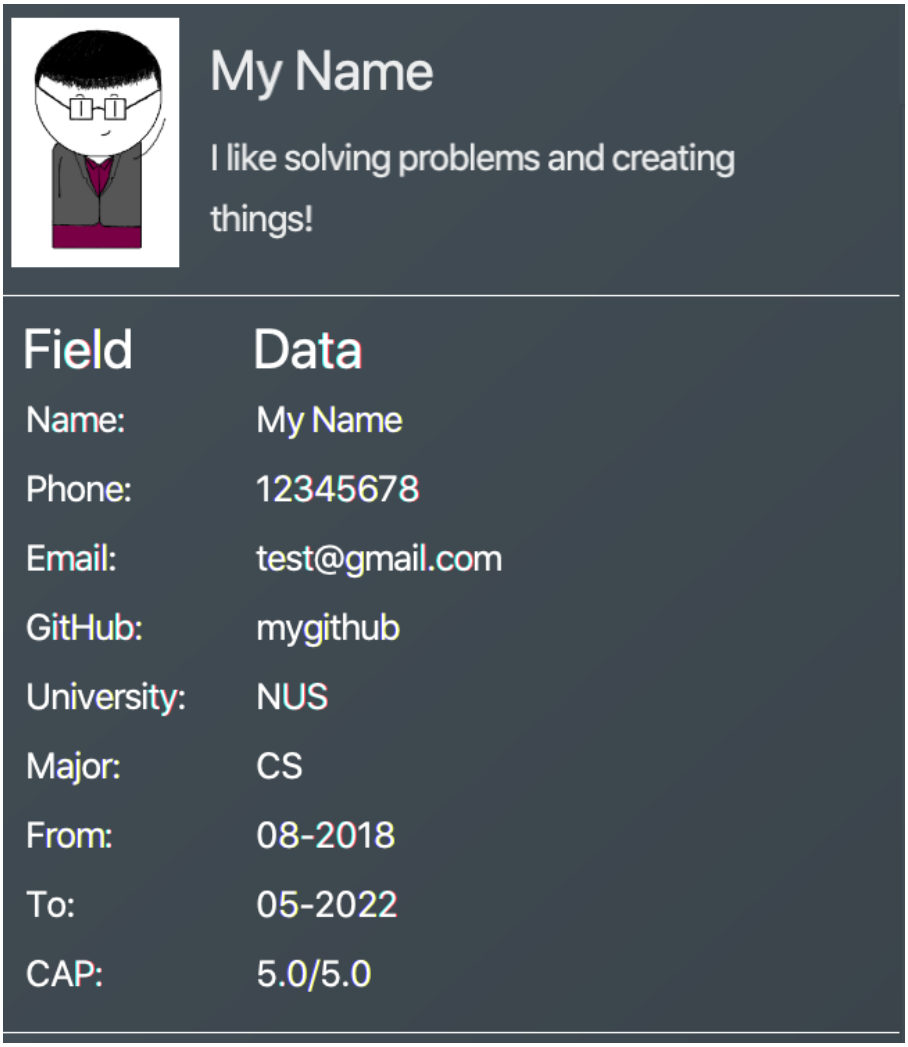
Example 1: Update user profile details

Try typing in the command box this command:

```
me n/ My Name p/ 12345678 e/ test@gmail.com d/ I like solving problems and creating things! g/ mygithub u/ NUS m/ CS f/ 08-2018 t/ 05-2022 c/ 5.0 5.0
```

Outcome:

The user box is updated accordingly as below:



Example 2: Update user profile picture

Follow the steps in one of these two links to copy an absolute file path according to your respective Operating System.

1. Mac: https://osxdaily.com/2013/06/19/copy-file-folder-path-mac-os-x/
2. Windows: https://www.laptopmag.com/articles/show-full-folder-path-file-explorer

Afterwards, try a command similar to the one below:

```
me dp/ /Users/nhamquochung/Desktop/test.png
```

Outcome:

The user profile picture is updated accordingly as below:

{end of extract 1}

{start of extract 2: find}

Finding items by name: `find`

Finds items of a specific `type` in the corresponding list of items whose names contain the specified keyword(s).

Format: `find KEYWORD [MORE_KEYWORDS]… i/ TYPE`

The specific command syntax could be found in the table below:

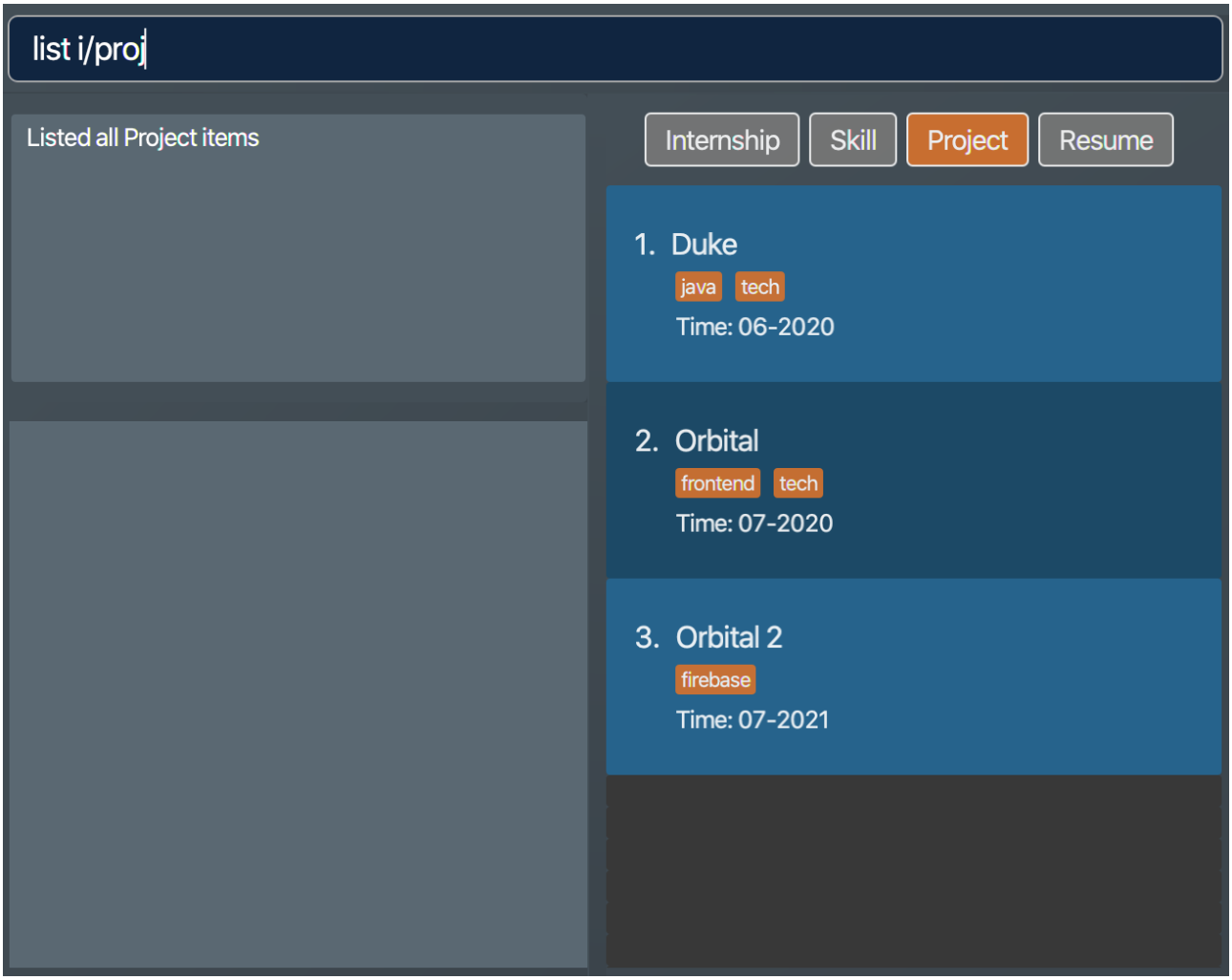
Type	Format
Internship	`find KEYWORD [MORE_KEYWORDS]… i/ int`
Project	`find KEYWORD [MORE_KEYWORDS]… i/ proj`
Skill	`find KEYWORD [MORE_KEYWORDS]… i/ ski`
Resume	`find KEYWORD [MORE_KEYWORDS]… i/ res`
Note	`find KEYWORD [MORE_KEYWORDS]… i/ note`

Example: Try typing in the command box these commands one by one!

1. `list i/ proj`
2. `find Orbital i/ proj`

Outcome

1. All `project` items are listed in the list box. A sample project list is shown below.



2. `Projects` whose names match keywords are listed in the list box.

{end of extract 2}

{start of extract 3: done}

Mark a note as done: done

Mark a `Note` at a specific index from the current note list as `done`.

> **Format:** `done INDEX`

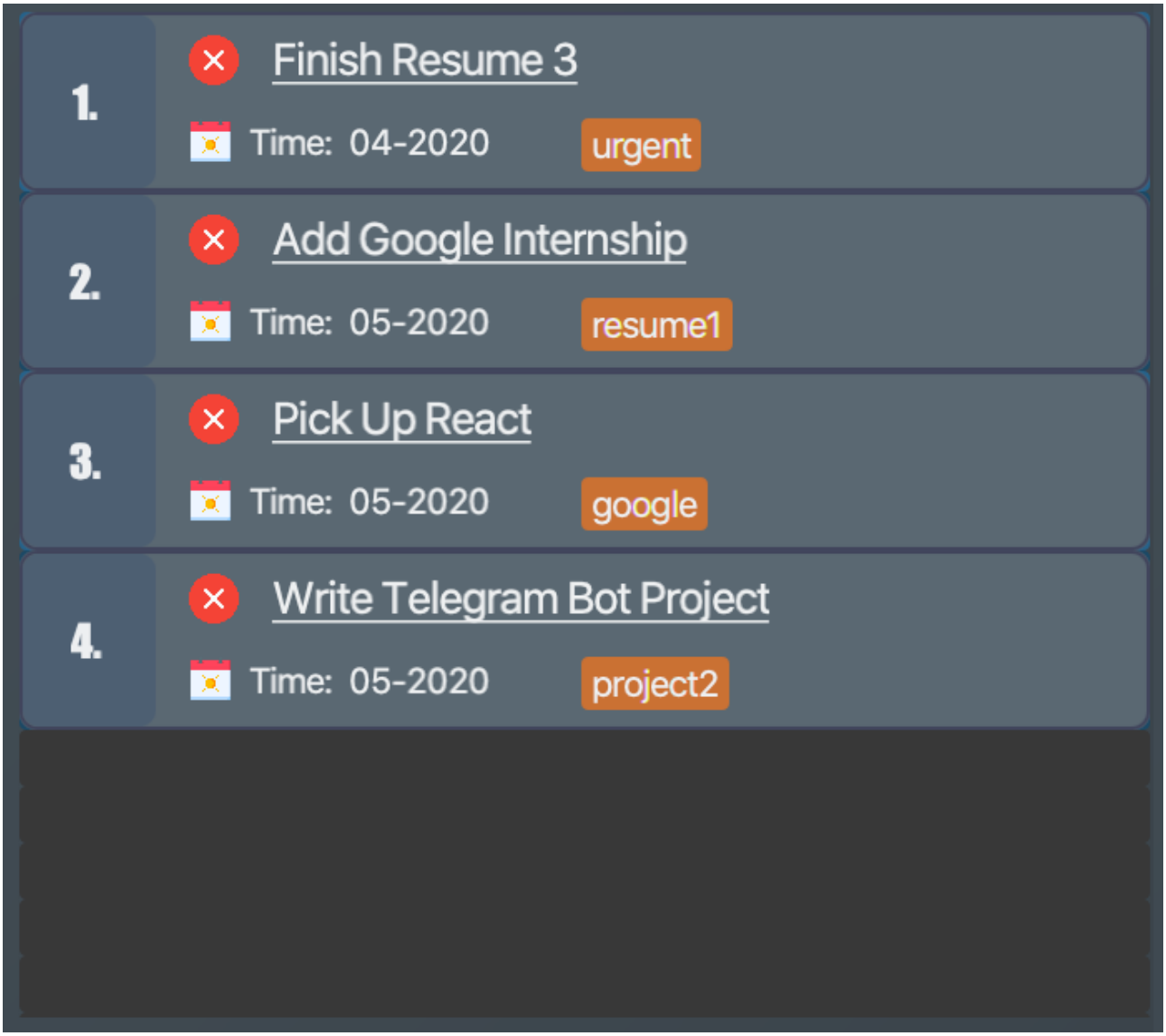
- A valid `INDEX` is a positive integer that identifies an existing `note`.

Example: Let's try out the following commands!

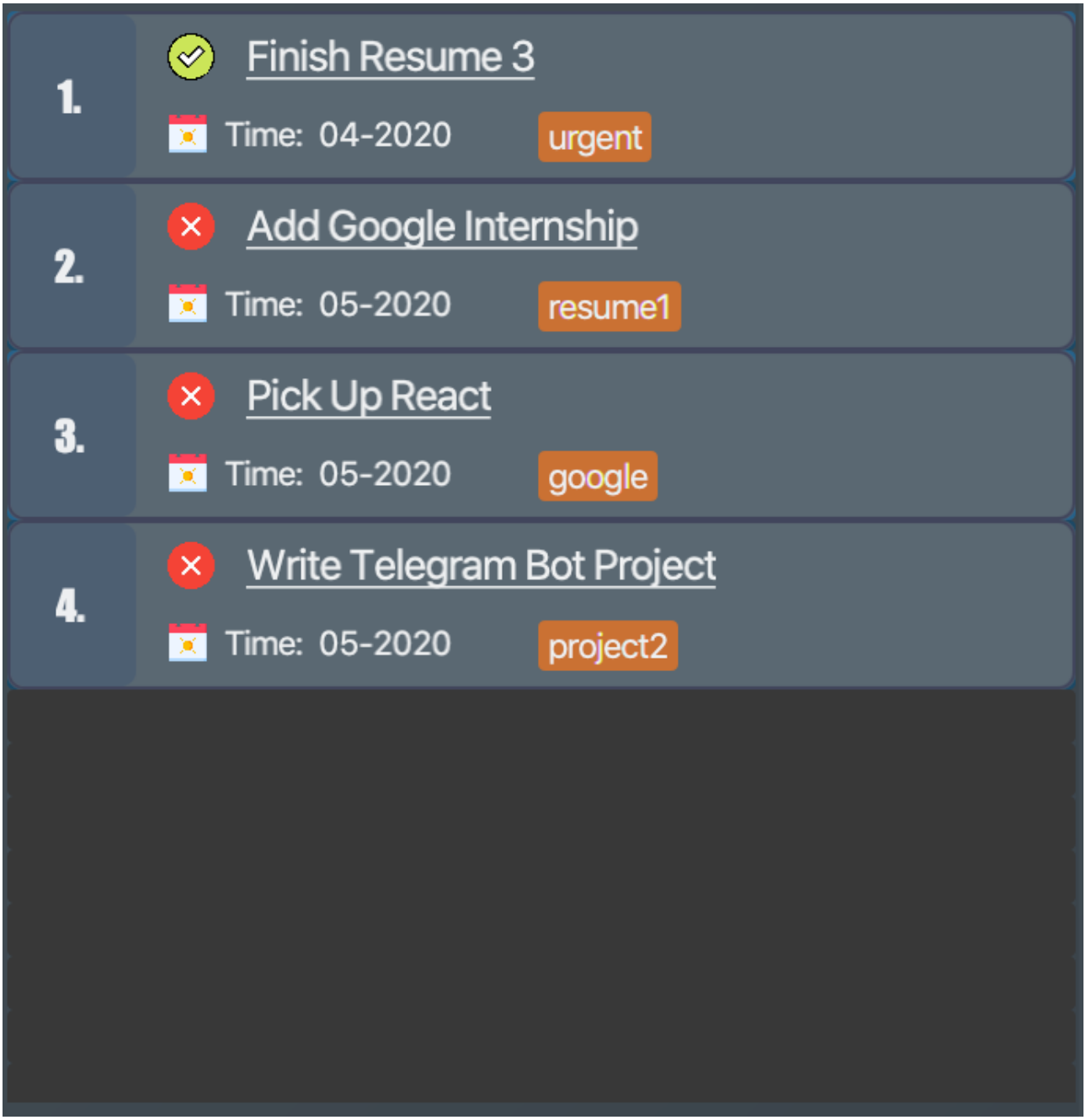
1. `list i/ note`
2. `done 1`

Outcome:

1. The first command lists out all `notes`. Assuming that you want to mark the first `note` in the list as `done`.



2. The second command marks this `note` as done, by updating the tick box.



{end of extract 3}

{start of extract 4: exit}

Exiting the program : `exit`

Exits from **ResuMe**.

> **Format:** `exit`

{end of extract 4}

Contributions to the Developer Guide (Extracts)

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: Ui}

UI component

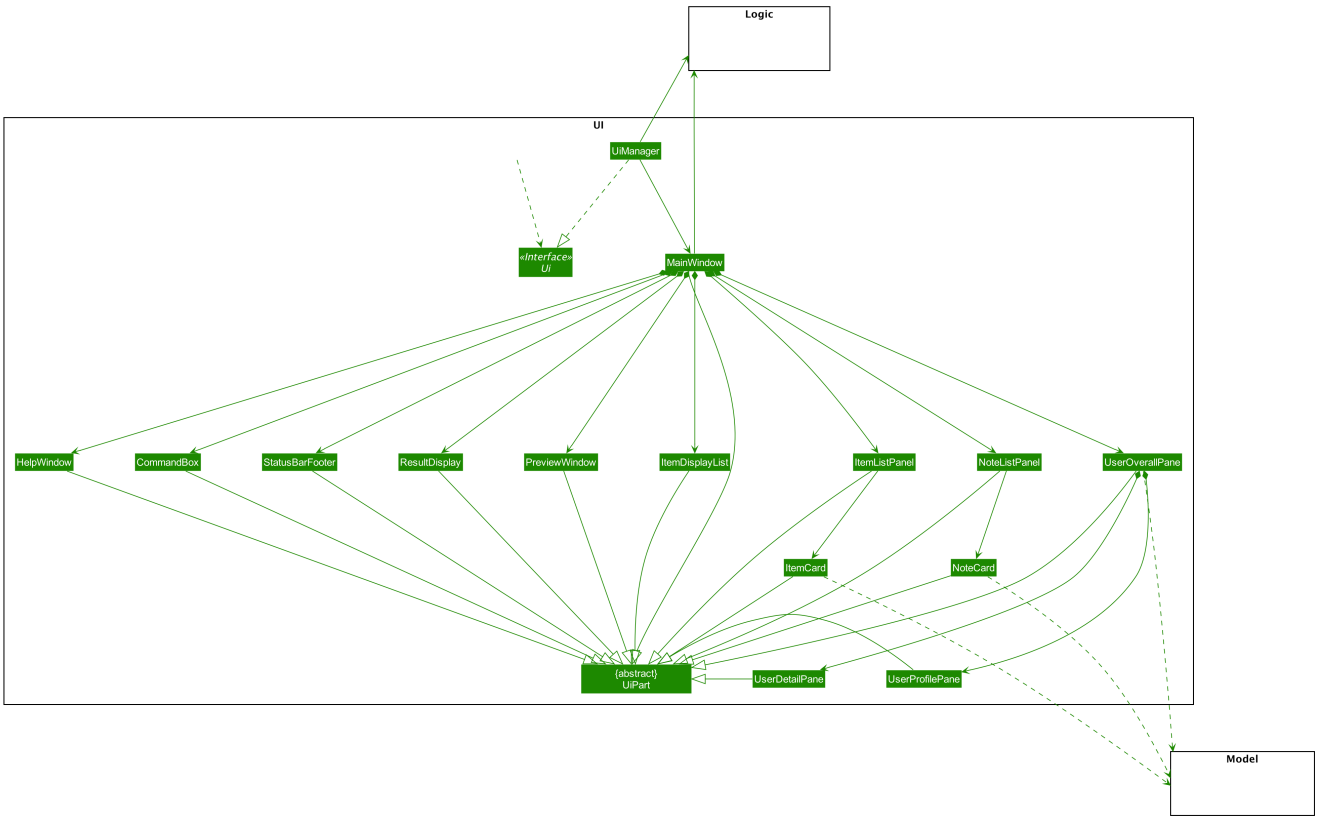


Figure 1. Structure of the UI Component

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 extract 1}

{start of extract 2: Me}

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.

`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.

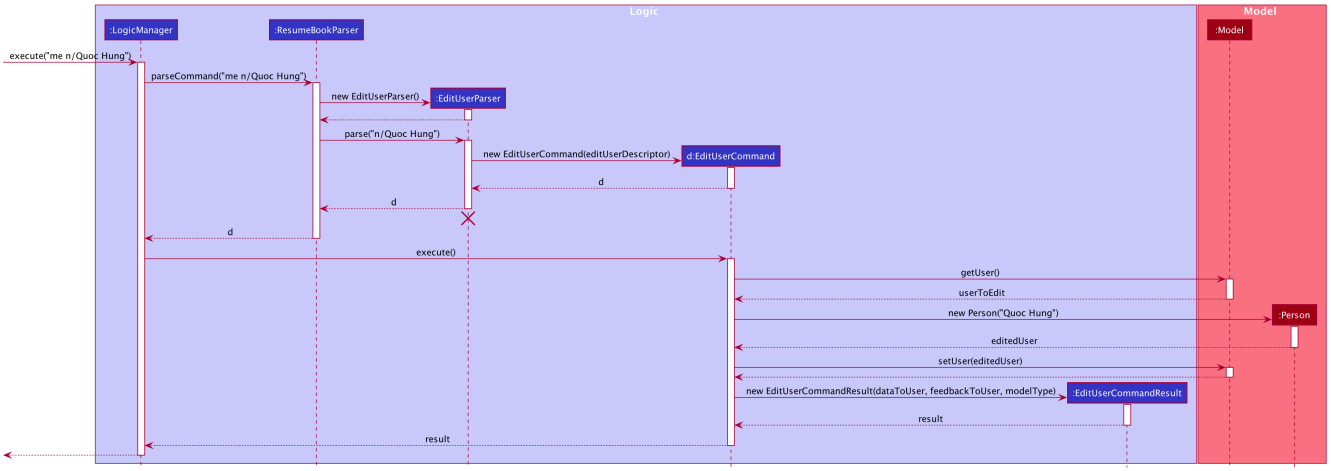
Note: To set customised user picture, the file path of your display picture has to be absolute and from the root directory of your computer.

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:



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 extract 2}

{start of extract 3: Note}

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.

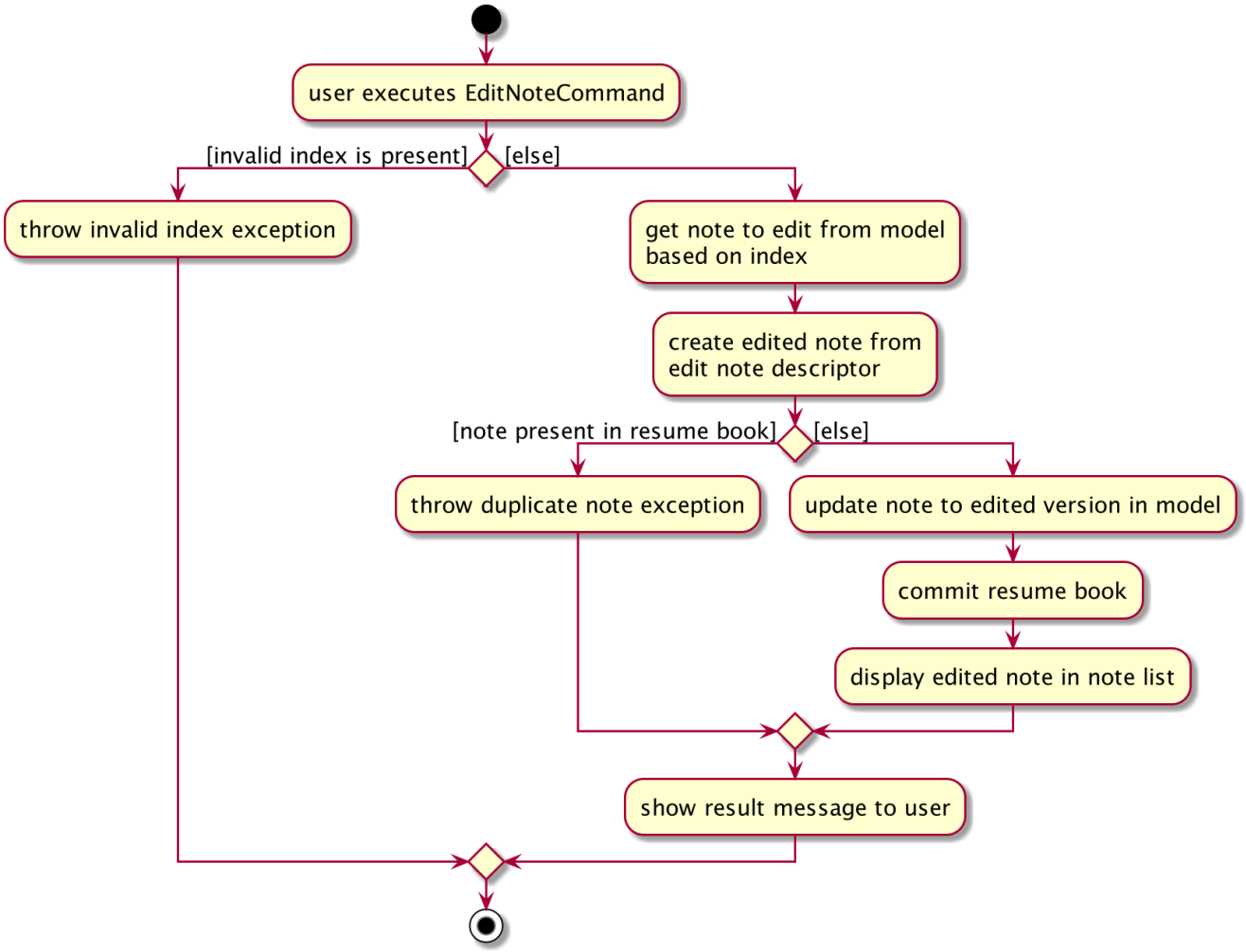
Step 2. The user executes `edit INDEX i/ Note [n/ NAME] [t/ TIME] [#/ TAG]`. If the specified note index is invalid or out of bound, ResuMe will throw an invalid index error message.

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:



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 extract 3}