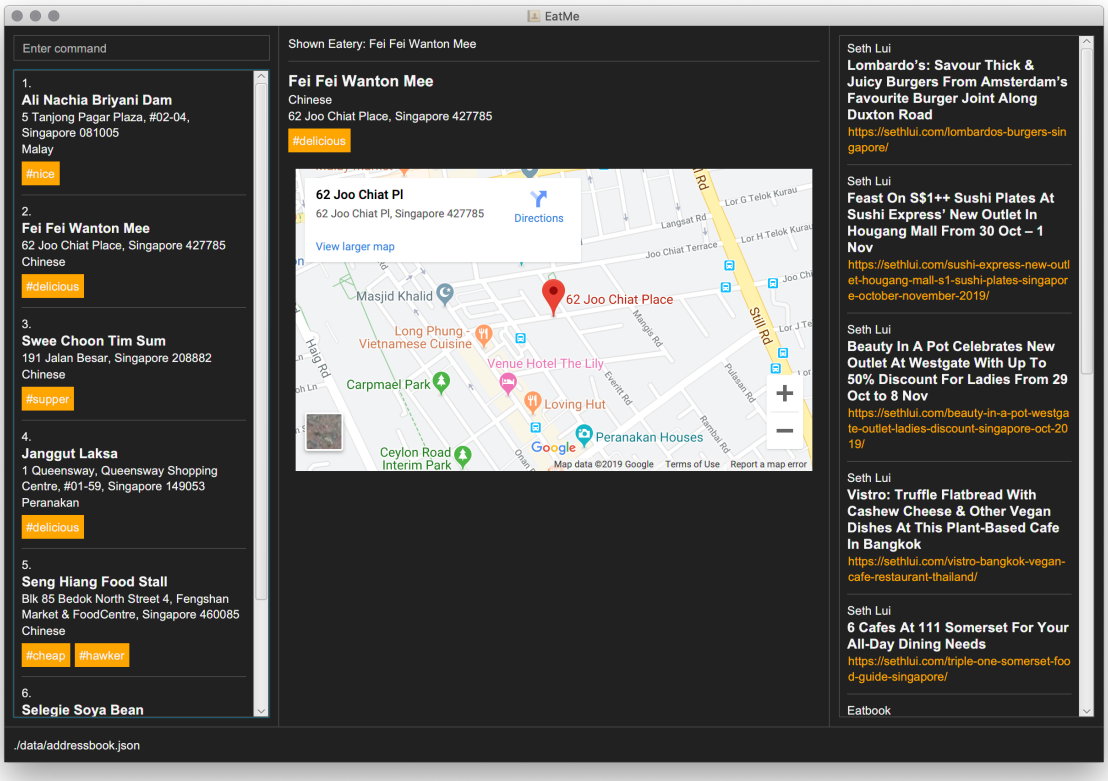
Ang Kai Qi - Project Portfolio for EatMe

PROJECT: EatMe

About the Project

My team of 5 software engineering students and I were tasked with enhancing a basic command line interface desktop addressbook application for our Software Engineering Project. We chose to morph it into an eatery records management called EatMe. This enhanced application enables users to file and recall eatery information; manage places visited and to be visited; and reading up on places visited by my favourite food bloggers. The user interacts with it using a CLI, and it has a GUI created with JavaFX. It is written in Java, and has about 10 kLoC.



Summary of contributions

- **Code Contributed**: Code Contributed
- **Major enhancement**: added **the ability to Load your friends' Eatery List**
  - What it does: Allows the user to view and share their friends' Eatery List.

    - Justification: This feature improves the product because it creates social interaction among friends and users to share their favourite places easily. This is especially important in deciding where to eat with friends on common places they like to eat. Unfortunately this extra feature will only be available in future releases.
    - Highlights:
        - This feature was challenging to build as tracing the UserPrefs was a tricky process. It was discovered that the UserPref was saved after exiting the application although the user is never allowed to edit the UserPrefs initially. Thus overriding any changes done to the `preferences.json` file.
        - This implementation of this feature's test is challenging as the load command results in the file being named according to the `System.getProperty("user.name)"` in the data path folder created only when the application is ran for the first time, and thus during the test, we had to be conscious of overriding the load command's file Path and create separate data folders for the test.
        - This feature affects other commands as after loading the desired file, the user will be prompted to relaunch the application to switch files. However, failure to do so and on the user's insistence on executing other commands would affect the current file being displayed on the GUI.
    - Credits: *{Our team's techlead Daryl, helped in identifying the test case constraints and fixing the errors found.}*
    - Relevant PRs: #166
- **Major enhancement**: Added a `close` and `reopen` command that allows the user to close or reopen eateries.
    - What it does: Allows the user to close or reopen an eatery that has either closed or reopened for business.
    - Justification: This feature improves the product as it allows the user to record if the eatery is still in business, thereby reducing the possibility of the user suggesting an eatery to his friends that is closed.
    - Highlights:
        - This feature was tedious to implement as now a new boolean field is added to the Eatery class. Hence, all the existing commands and methods that create a new eatery have to take into consideration this new field that needs to be set.
        - This feature allowed us to implement a visual effect on the eatery list panel on the right so that users can tell off hand if the eatery is closed without the `show` command.
    - Credits: *{Our team's techlead Daryl, helped in implementing the visual changes.}*
    - Relevant PRs: #76, #79, #96
- **Other contributions**:
    - Project management:
        - Managed deadlines for documentations
    - Documentation:

        - Updated the Developer Guide with new information (PRs #14, #82, #187)
        - Standardized the Developer Guide for draft 1 (#92)
        - Enforced User Guide standard for draft 1 (#91)
    - Community:
        - PRs reviewed (with non-trivial review comments): #74, #81, #84, #93, #177
        - Reported bugs and suggestions for other teams in the class (examples: 1, 2, 3)
    - Others:
        - Refactored AddressBook to Eatery List (#61)

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.

Load eateries from another user profile: `load`

Loads eateries from another user profile.

EatMe allows you to share eatery data with your friends by simply transferring your user profile. Your user profile is a file that ends with `.json` and can be found in the same place where you saved the EatMe app.

Did a friend pass you their user profile? Simply place their file where you saved the EatMe app, and execute this command with their username to view their data.

Format: `load \u [username]`

> - Loads eatery data from the specified user profile.
> - The user can supply the user profile with or without ".json" extension, however, the file **must** still be a valid user profile generated by the EatMe app.

Examples:

- `load \u john`
- `load \u john.json`

Marking an eatery as closed: `close`

Closes an eatery if the eatery no longer exists. The eatery will still be listed, but will be highlighted in red to inform you that the eatery no longer exists. In the case that you accidentally closed the wrong eatery or the eatery reopens, the `reopen` command does the opposite of `close`.

Format: `close [index]`

Examples:

- `close 2`

Reopening a closed eatery: `reopen`

Reopens a previously closed eatery. Format: `reopen [index]`

Examples:

- `reopen 2`

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.

Close Eatery feature

Implementation

The close mechanism is facilitated by `EateryList`. It implements ReadOnlyEateryList interface with the following operation:

- `EateryList#setEatery()` — Replaces an Eatery with a modified Eatery specified by the user input.

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

Step 1. The user launches the application for the first time. The `EateryList` will be initialized with the initial json data stored.

Step 2. The user executes `close 1` command to close the 1st Eatery in the address book.

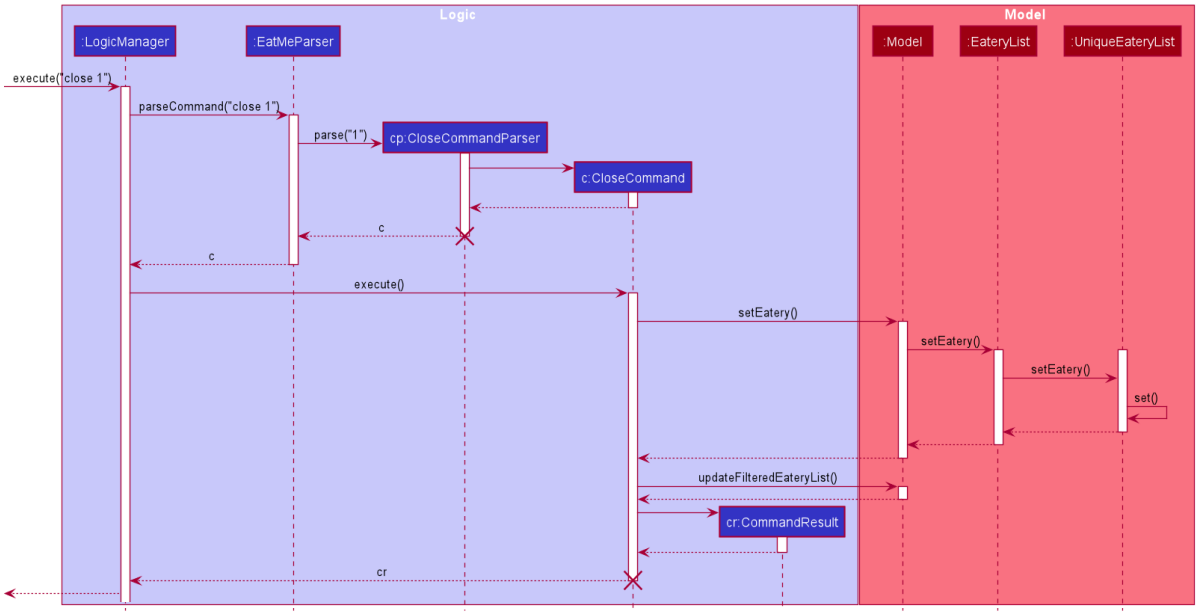
NOTE	If the index given is not a valid index (ie, out of bounds or negative), CloseCommand will throw a CommandException to the user rather than attempting to close the Eatery.

Step 3. The address book now returns a success message upon successfully closing the Eatery, and the Eatery will be highlighted in red.

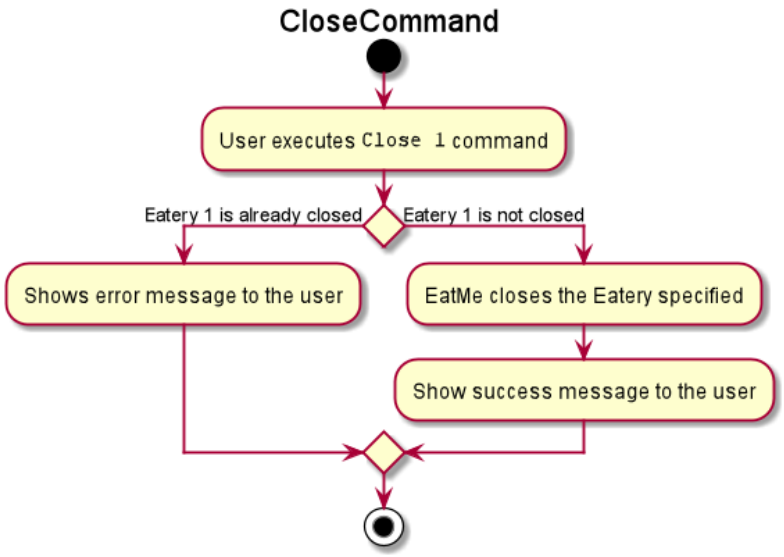
Step 4. The user then decides that he wants to close another Eatery.

NOTE	If the index given points to an Eatery already closed (ie, 1 in this situation), CloseCommand will throw a CommandException to the user rather than attempting to close the Eatery.

The following sequence diagram shows how the `close` command works:



The following activity diagram summarizes what happens when a user executes a new `close` command:



Design Considerations

Aspect: How Close executes

- **Alternative 1 (current choice):** Returns a new Eatery with identical fields except for the isOpen field.
    - Pros:
        - Follows pre-exisiting EditCommand implementation.
        - No need for setter methods.
    - Cons: Have to return a new object each time a change is made.
- **Alternative 2:** Setter method for isOpen field of Eatery.
    - Pros: No need for extra methods in the flow to change the object.

    - Cons:
        - Breaks pre-existing EditCommand implementation.
        - Need for setter methods.

Aspect: Data structure to support the Close command

- **Alternative 1 (current choice):** Uses a boolean value to keep track if the Eatery is reopened or closed.
    - Pros: Easily implemented.
    - Cons: An additional variable to check when executing other commands. Possibility of incorrect manipulation of an Eatery object
- **Alternative 2:** Maintain two separate lists of Eateries for Reopened and Closed.
    - Pros: Closed Eateries stored apart from Reopened Eateries. Commands executed will only affect Eateries stored in a particular list.
    - Cons: Requires proper handling of individual data structures to ensure each list is maintained and updated correctly.

Reopen Eatery feature

Implementation

The reopen mechanism is facilitated by `AddressBook`. It implements the `ReadOnlyAddressBook` interface with the following operation:

- `AddressBook#setEatery()` — Replaces an Eatery with a modified Eatery specified by the user input.

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

Step 1. The user launches the application for the first time. The `AddressBook` will be initialized with the initial json data stored.

Step 2. The user executes `reopen 1` command to close the 1st Eatery in the address book.

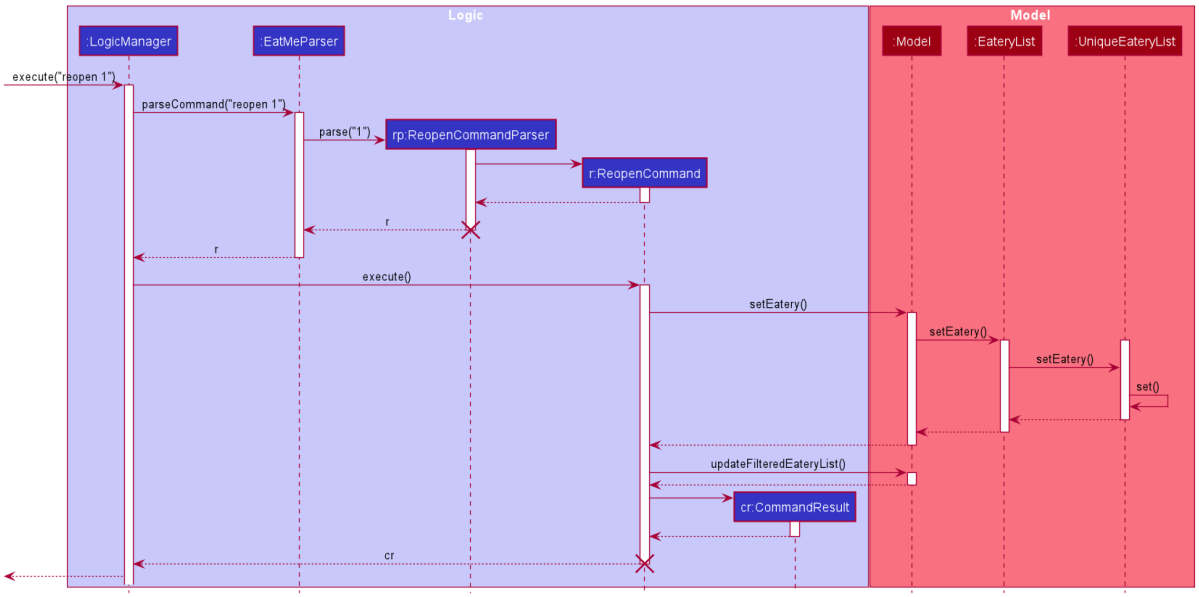
NOTE	If the index given is not a valid index (ie, out of bounds or negative), ReopenCommand will throw a CommandException to the user rather than attempting to reopen the Eatery.

Step 3. The address book now returns a success message upon successfully reopening the Eatery, and the Eatery will not longer be highlighted in red.

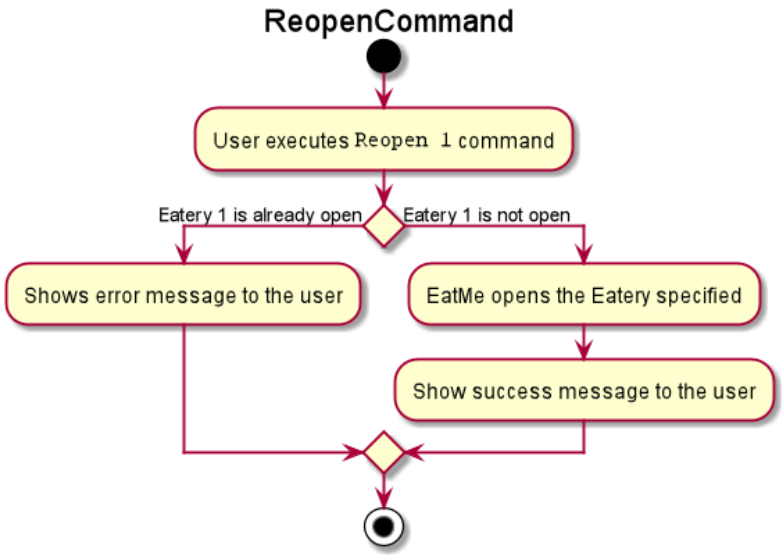
Step 4. The user then decides that he wants to reopen another Eatery.

NOTE	If the index given points to an Eatery already reopened (ie, 1 in this situation), ReopenCommand will throw a CommandException to the user rather than attempting to reopen the Eatery.

The following sequence diagram shows how the `reopen` command works:



The following activity diagram summarizes what happens when a user executes a new `reopen` command:



Design Considerations

Aspect: How Reopen executes

- **Alternative 1 (current choice):** Returns a new Eatery with identical fields except for the isOpen field.
    - Pros:
        - Follows pre-exisiting EditCommand implementation.
        - No need for setter methods.
    - Cons: Have to return a new object each time a change is made.
- **Alternative 2:** Setter method for isOpen field of Eatery.

    - Pros: No need for extra methods in the flow to change the object.
    - Cons:
        - Breaks pre-existing EditCommand implementation.
        - Need for setter methods.

Aspect: Data structure to support the Reopen command

- **Alternative 1 (current choice):** Uses a boolean value to keep track if the Eatery is reopened or closed.
    - Pros: Easily implemented.
    - Cons: An additional variable to check when executing other commands. Possibility of incorrect manipulation of an Eatery object
- **Alternative 2:** Maintain two separate lists of Eateries for Reopened and Closed.
    - Pros: Reopened Eateries stored apart from Closed Eateries. Commands executed will only affect Eateries stored in a particular list.
    - Cons: Requires proper handling of individual data structures to ensure each list is maintained and updated correctly.

Load friend's EateryList feature

Implementation

The load mechanism is facilitated by `ModelManager`. It implements the `Model` interface with the following operation:

- `Model#setUserPrefs()` — Replaces the existing UserPrefs with a modified UserPrefs specified by the user input.

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

Step 1. The user launches the application for the first time. The `EateryList` will be initialized with the initial json data stored.

Step 2. The user executes `load \u Alice` command to change the `UserPrefs` of the `ModelManger`.

NOTE	If the command is missing the username (i.e. `load \u`), LoadCommandParser will throw an ParserException to the user with an error message specifying that the command parameters are incorrect, and an example usage of the command.

NOTE	If the command is missing the username and the prefix (i.e. `load`), this will be interpreted as loading the owner's own Eatery List.

Step 3. The Eatery List now returns a success message upon successfully changing the `UserPrefs`, and prompts the user to relaunch the application.

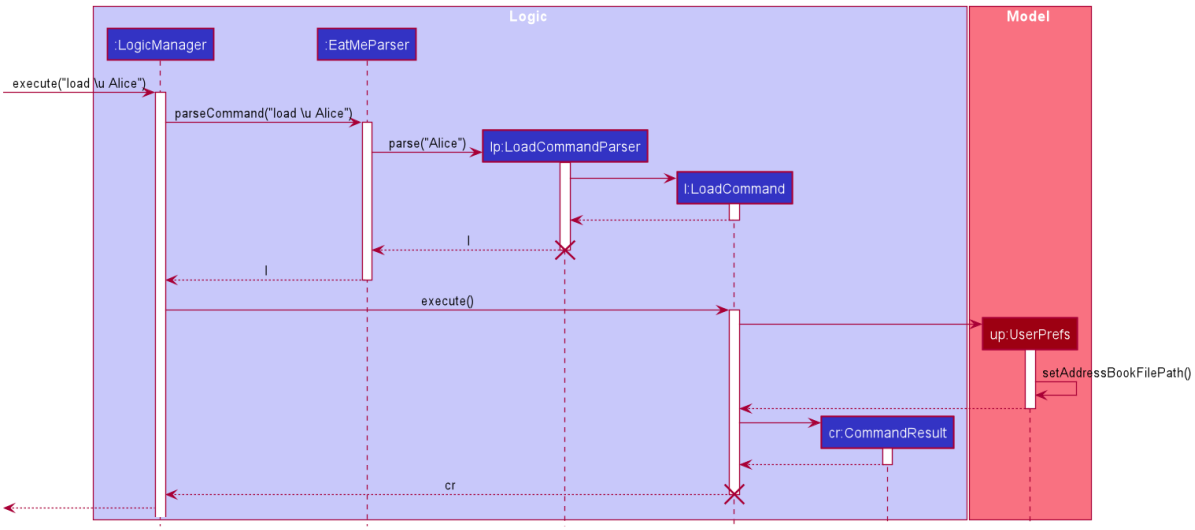
Step 4. The user then decides that he wants to change his `UserPrefs` to another friend's one. Before exiting the application, the user can still enter `load \u Bob` command to change the `UserPrefs`.

NOTE	If the user attempts to load a file that has already been set in the `UserPrefs`, the LoadCommand will throw a CommandException to the user with an error message specifying that the `UserPrefs` has already been changed.

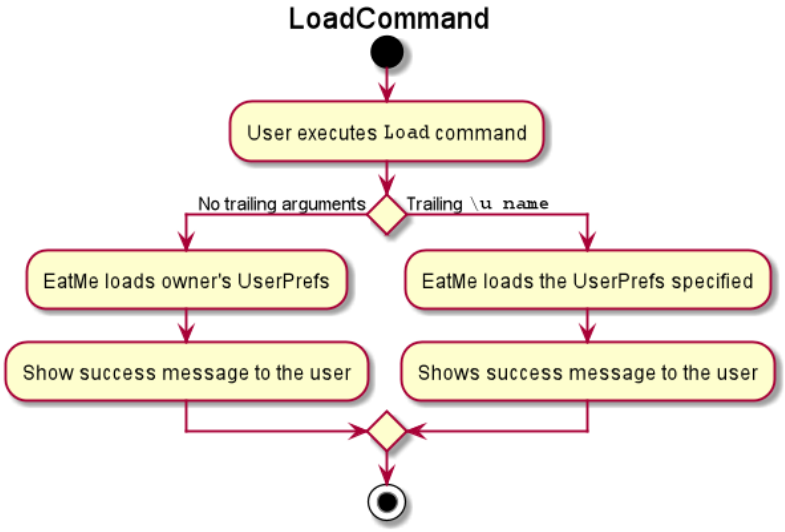
NOTE	If the user attempts to load a file that does not exist in the default filepath, the LoadCommand will throw a CommandException to the user with an error message to prompt him to check that the file exists.

Step 5. The user then decides that he wants to change his `UserPrefs` back to his own instead. Before exiting the application, the user can still enter `load` command to reset the `UserPrefs` of the `ModelManage` back to his own Eatery List.

The following sequence diagram shows how the `load` command works:



The following activity diagram summarizes what happens when a user executes a new `load` command:



Design Considerations

Aspect: How Load executes

- **Alternative 1 (current choice):** Modifies the existing model in the Eatery List.
    - Pros: Easily implemented without needing to relaunch the GUI.
    - Cons: Require the application to be terminated for the storage to save UserPrefs.
- **Alternative 2:** Creates a new model to replace the current one.
    - Pros: Easier to implement the loading of file without relaunching the application.
    - Cons: Need to relaunch the GUI so that the new file can be seen.

Aspect: User Experience to execute Load

- **Alternative 1 (current choice):** The user has to relaunch the application.
    - Pros:
        - Easily implemented.
        - Clear to the user that the UserPaths has been changed.
    - Cons: Additional work from the user
- **Alternative 2:** Relaunch the GUI
    - Pros: The user does not need to reload the application.
    - Cons:
        - Harder to implement.
        - Might confuse the user if the UserPaths has been changed. Especially if the friend's EateryList is similar to the user (i.e. same school).