Low Zhang Xian - Project Portfolio

PROJECT: CALGO

Overview

This Project Portfolio page aims to showcase my contributions to Calgo - a Software Engineering project developed during my second year of undergraduate studies at the National University of Singapore.

About the Team

We are 5 Year 2 Computer Science undergraduates reading CS2103T: Software Engineering.

About the Project

Calgo was created to help foodies achieve their fitness goals. Our story first began when the team was given the source code of Address Book Level 3 (AB3) project. We were challenged to collaborate as a team to morph this into a new product incrementally using Brownfield software development.

As all of us were food lovers, we decided to embark on this journey to create Calgo, a personal meal tracking assistant. Today, Calgo is well-equipped to help users keep track of their meal consumption and store all their favourite food, along with its nutritional information. On top of this, Calgo has the ability to offer our users insightful reports based on their data. Calgo was created with a strong emphasis on providing the best user experience to our users.

Summary of contributions

- **Major enhancement**: I implemented the `update` command.
  - What it does: This command allows the user to add new `Food` entries or edit existing `Food` entries in the `Food Record`.
  - Justification: Instead of having two separate commands: add and edit to perform the function of updating the Food Record, this smart command decides whether to add or edit depending on whether the Food entered by the user exists in the Food Record. This improves user experience and reduces the necessity of another command.
  - Highlights: Implementing this requires comprehensive understanding of the entire application architecture. Additional thought and effort was required to ensure that the process is seamless for the user.
- **Major enhancement**: I implemented the Real-time Suggestion feature
  - What it does: This feature shows the user existing similar Food items in real time when using any of these three commands: `update`, `delete`, `nom`.
  - Justification: Instead of the need to use additional commands or having to manually scroll

through the `Food Record`, this feature allows user to know whether a particular Food item already exists. This is especially helpful for these three core commands that generally require this knowledge.

    - Highlights: This enhancement shows our emphasis on providing a good user experience to the user. It also requires a good understanding of the all core components used by the application to achieve this.

- **Minor enhancement**: I designed the GUI for the `DailyFoodList`
    - What it does: This enhancement helps the DailyFoodList be displayed in a more minimalistic and inituitive manner.
    - Justification: This helps the user quickly see important aspects of their meal consumption, such as the `Name`, `Index`, `Portion` and `Rating` of each Food consumed. This improves the design and also user experience for the user.
    - Highlights: This enhancement is well-designed with good color scheme. A good mix of understanding of JavaFX components and research into new JavaFX APIs were required to achieve this.
- **Code contributed**: You can view my contributions to Calgo here.
- **Other contributions**:
    - Documentation:
        - Contributed sections for delete and update commands in User Guide: #69, #269, #286
        - Contributed sections for Logic Component, Modifying the Food Record and Real-time Suggestion for existing Food in Food Record for the Developer Guide: #69, #128, #279, #286, #295
    - Project and team management:
        - Morphed test cases for AB3 to new features in Calgo: #136, #139
        - Update team pages/documentation: #69, #128, #130, #205, #251, #269, #279, #286, #295, #296
        - Software Developer: Handled object modelling, designing overall architecture and maintaining good code quality.
        - Product ideation and brainstorming, contributions to GUI design and user testing.
    - Beyond the team:
        - Peer testing and bug reporting: #1, #2, #3, #4, #5, #6, #7, #8, #9, #10, #11, #12, #13, #14

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. Please note that some hyperlinks may not work as the guide is not part of this portfolio.*

`update` : Updating Calgo's `Food Record`

(by Zhang Xian)

Tired of searching for nutritional information online for the same food repeatedly? Frustrated of having no convenient place to note it down?

The `update` command allows you to enter new `Food` entries into Calgo's `Food Record`. Moreover, as a smart feature, Calgo detects if there is an existing `Food` entry with the same name. If so, it will edit that `Food` entry with new information provided by you.

Here are some key pointers:

- All of Calgo's `Food` entries have unique names.
- Calgo automatically formats name inputs to proper case. Therefore it does not matter whether you input a name in upper or lower case.
  - For instance, updating a `Food` with name `chicken nugget spicy` will result in the `Food` being saved as `Chicken Nugget Spicy` in the `Food Record`

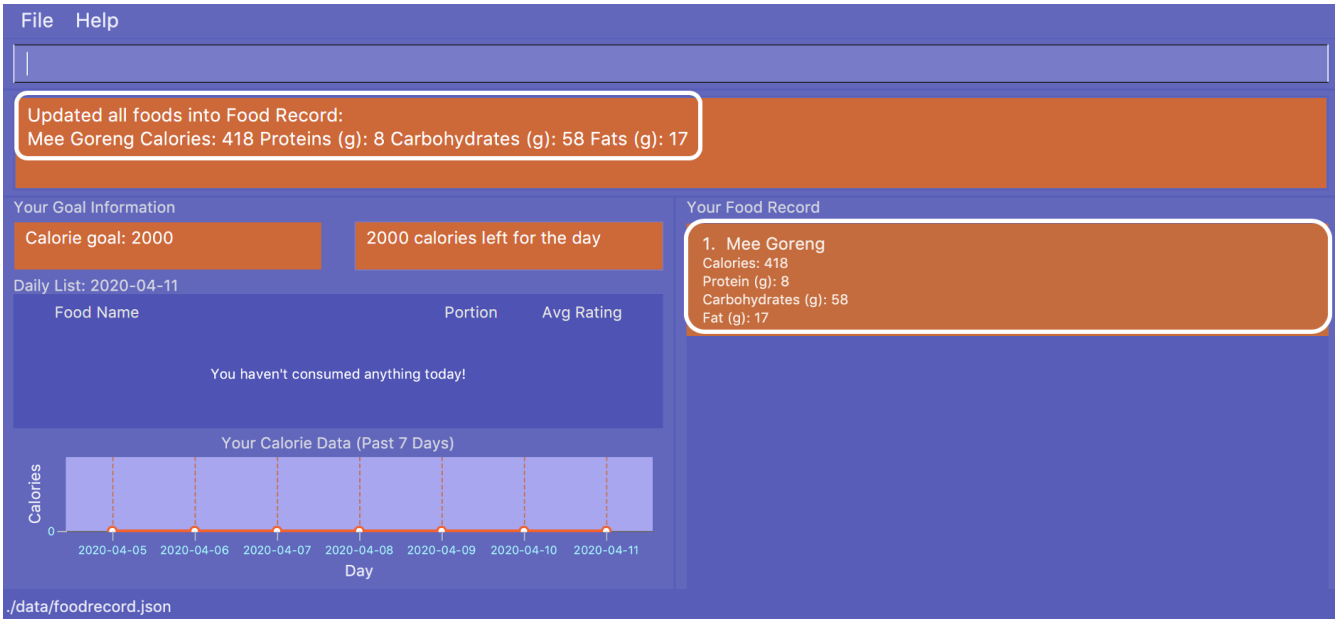
TIP We suggest you to be as specific as possible in naming your `Food`. Instead of naming your `Food` "Chocolate", perhaps "White Chocolate" or "Dark Chocolate" would be a better idea.

Format: `update n/NAME cal/CALORIES p/PROTEINS c/CARBOHYDRATES f/FATS [t/TAGS]`

Examples:

Example 1: Suppose you want to create a `Food` entry for Mee Goreng in Calgo. After searching online for the nutritional values for Mee Goreng, you found that Mee Goreng has 418 calories, 8g of protein, 58g of carbohydrate and 17g of fat. Here's how you update your new `Food`, Mee Goreng, into your `Food Record`:

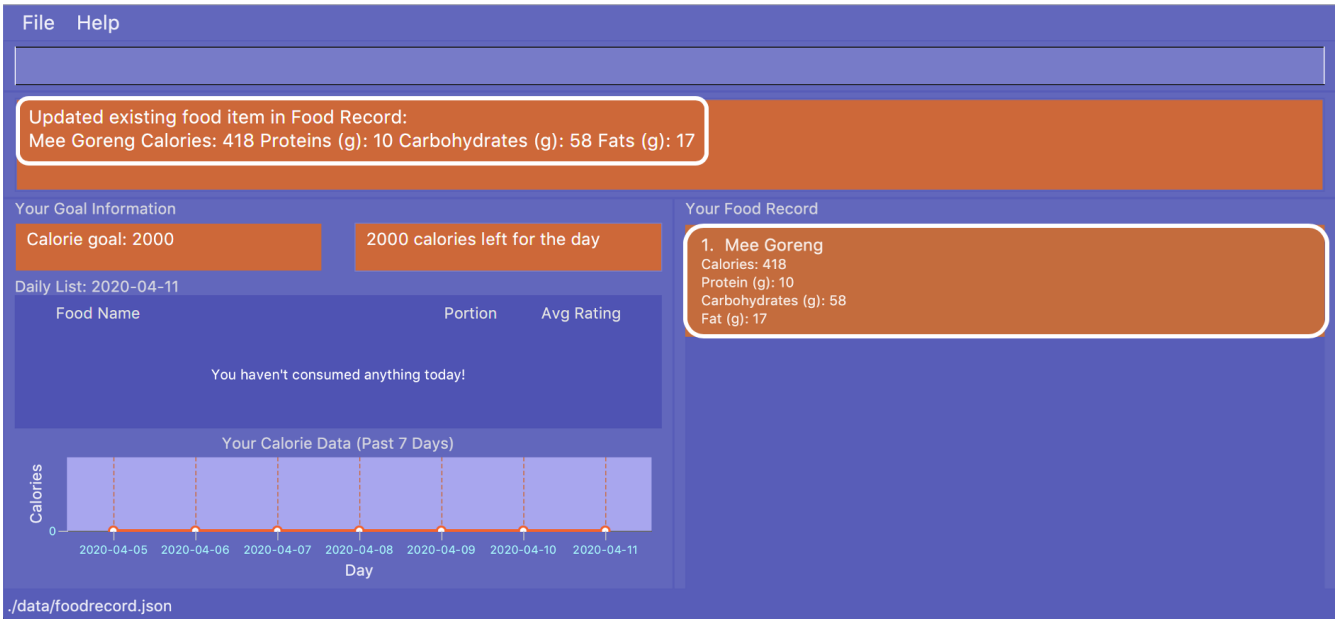
You should type `update n/Mee Goreng cal/418 p/8 c/58 f/17` and press kbd:[enter].



Once the command has been entered, the `Result Display` shows the result of your command and the `Food Record` has been updated with a new `Food`, Mee Goreng.

Example 2: Perhaps you realised that there was an error with the nutritional values keyed in for an existing `Food`, Mee Goreng, inside your `Food Record`. You wish to `update` the protein value for Mee Goreng to a new value of 10g. This is how you can do it:

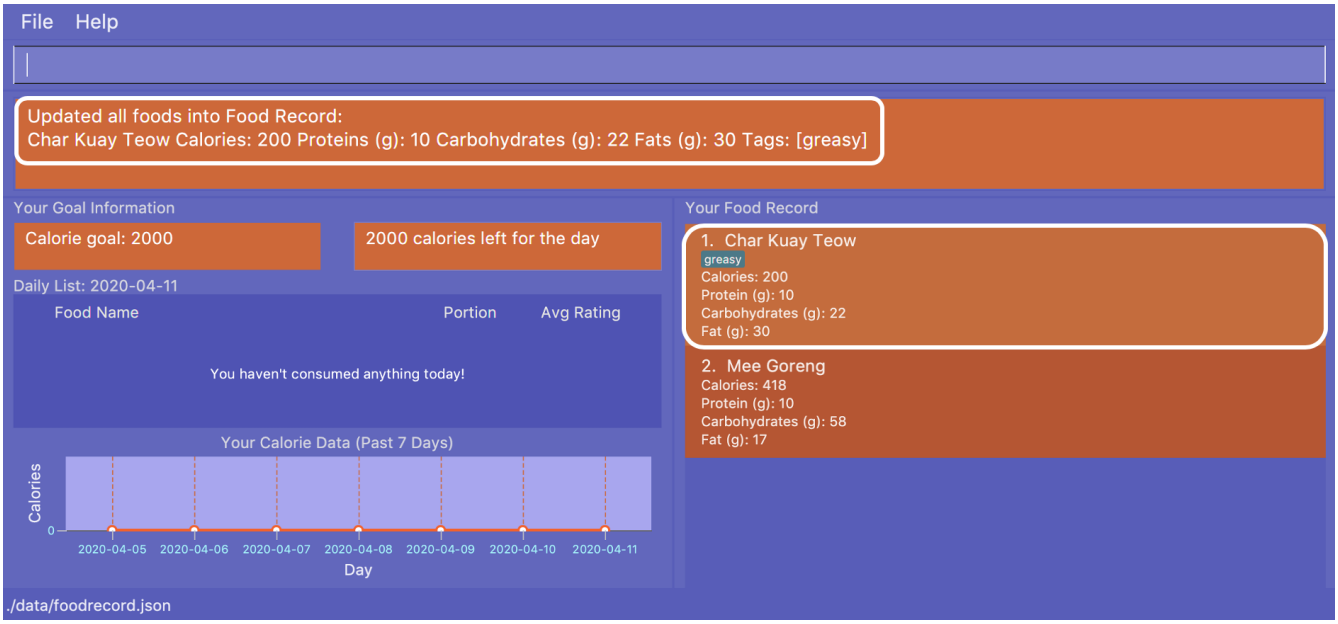
Type `update n/Mee Goreng cal/418 p/10 c/58 f/17` as input and press kbd:[enter].



Once the command has been entered, the `Result Display` shows the result of your command and the `Food`, Mee Goreng, in `Food Record` has been updated with a new protein nutritional value of 10g.

Example 3: Suppose you want to `update` a new `Food`, Char Kuay Teow, into the `Food Record`. However, you typed the name of the `Food` in hurry and did not capitalise some letters properly. Instead of "Char Kuay Teow", you accidentally typed "char KUay TeoW" in the name field. You can do this:

You can then type `update n/char KUay TeoW cal/200 p/20 c/22 f/30 t/greasy` and press kbd:[enter].



Calgo automatically formats the name of your `Food` for you to proper case. Hence, you see that instead of a hideous "char KUay TeoW" being updated into the `Food Record`, your new `Food` item is updated as "Char Kuay Teow".

`delete` : Deleting a `Food` from current `Food Record`

(by Zhang Xian)

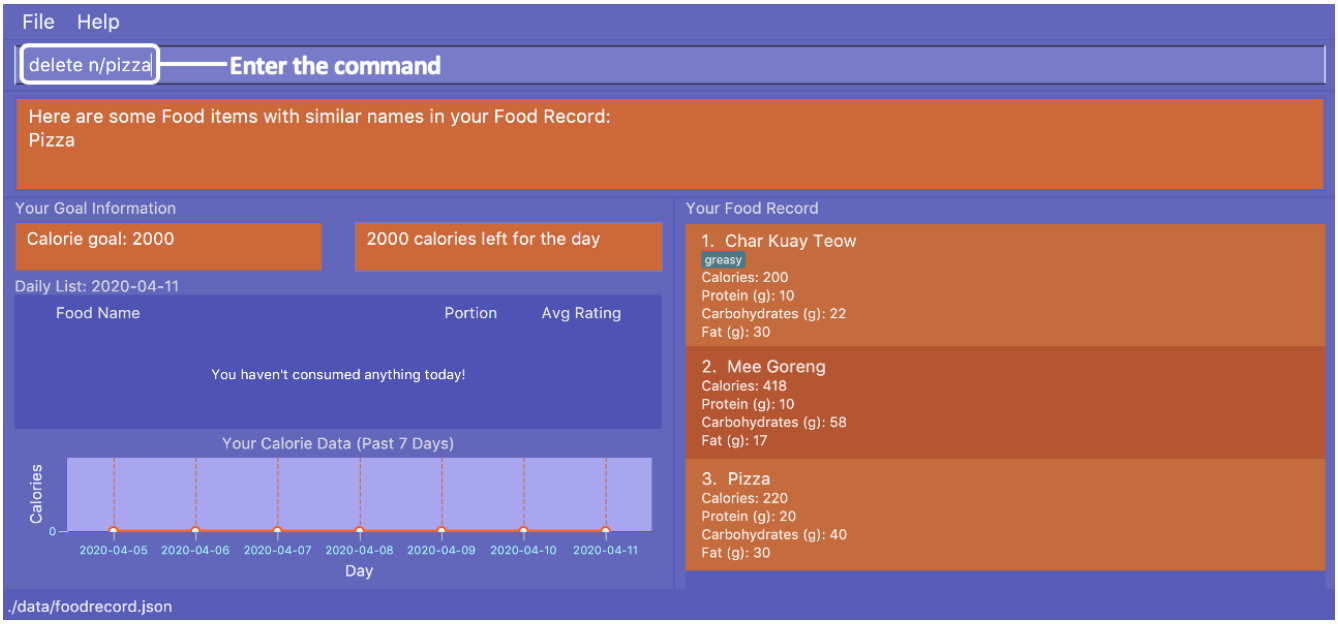
If you no longer require Calgo to store a particular `Food` and its nutritional values for you, you can use the `delete` command to remove the specified `Food` from your `Food Record`.

NOTE	The `Food` that you wish to `delete` must already exist in the current `Food Record`.
TIP	For your convenience, the `NAME` field of your input for the `delete` command is case insensitive. Therefore, `n/Pizza` and `n/pizza` are treated by Calgo as the same `Food`.

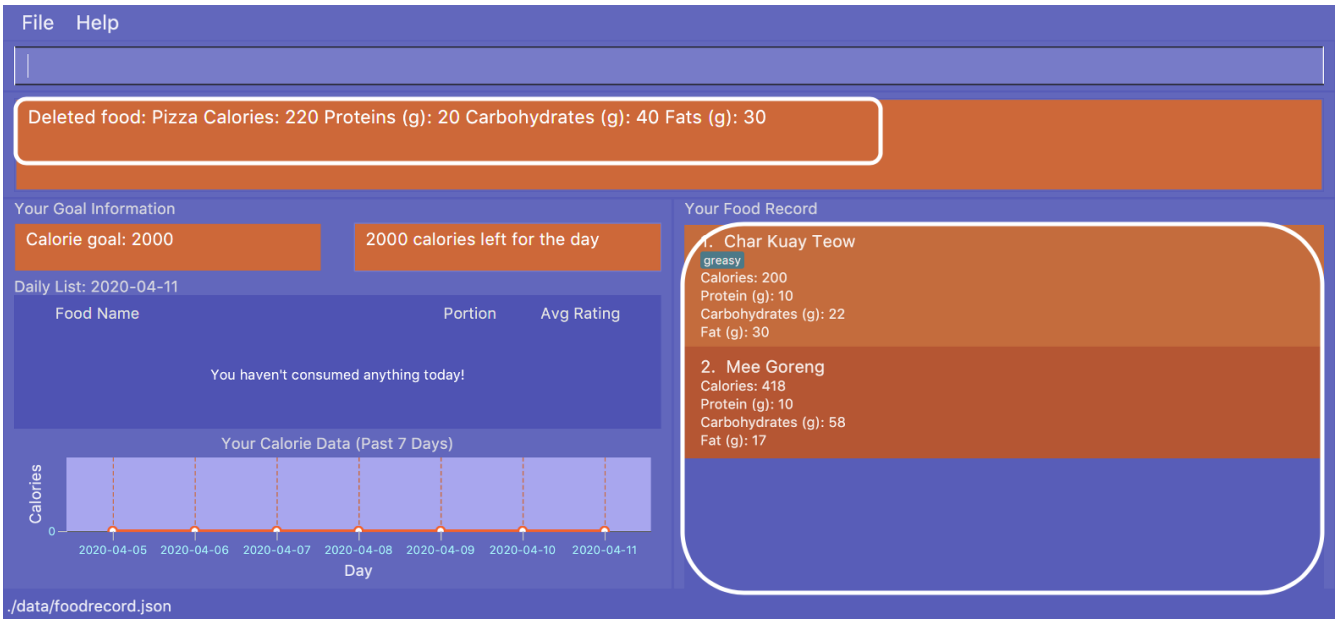
Format: `delete n/NAME`

Example:

Say you want to remove a `Food`, `Pizza`, that already exists in your `Food Record`. This is how you can do it:



You should first enter `delete n/pizza` as input, and press kbd:[enter]. Note that `delete n/Pizza` also has the same effect.



Once the command has been entered, the `Result Display` shows the results of your command and the `Food Record` no longer shows a `Food` entry for `Pizza`.

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. Please note that some hyperlinks may not work as the guide is not part of this portfolio.

Real-time Suggestions for existing `Food` in `FoodRecord`

(By Zhang Xian)

This section addresses how the GUI `Result Display` suggests `Food` with similar `Name` to the user for the commands `update`, `delete` and `nom`.

When the user have many `Food` entries in the `FoodRecord`, they may have difficulties finding out if a particular `Food` exists in the `FoodRecord`. For better user experience, this feature listens to the input of the user for these three commands and suggests similar existing `Food` entries in real time in the GUI's `Result Display`.

This feature listens to the input of the user after the `Prefix n/` and checks if there is a `Food` entry in the `FoodRecord` with a similar `Name`.

NOTE	The `Name` parameter is case-insensitive and searches the `Food` entries in the `FoodRecord` by whether they start with the user input so far after the `Prefix n/`.

Implementation

To be able to process user's input in real-time, we set a `listener` in the `CommandBox` to listen for the input of any of the three commands: `update`, `delete` or `nom` This feature is then facilitated by different objects, mainly `MainWindow` and `UniqueFoodList`. `MainWindow` interacts with `LogicManager` 's method of `getSimilarFood` which exposes the `FoodRecord`, allowing a filtered list of similar `Food` entries in the `UniqueFoodList` to be returned back to the user.

A predicate, `FoodRecordContainsFoodNamePredicate` is also essential in this implementation in ensuring that the correct similar `Food` items can be filtered from the `UniqueFoodList` back to the `LogicManager` to be displayed by the GUI. The `test` method of this predicate which is responsible for the above is shown:

```
public boolean test(Food food) {
    boolean foodStartsWithInputFoodName = food.getName().fullName.toLowerCase()
        .startsWith(foodName.toLowerCase().trim());
    boolean inputFoodNameStartsWithFood = foodName.toLowerCase().trim()
        .startsWith(food.getName().fullName.toLowerCase());

    return foodStartsWithInputFoodName || inputFoodNameStartsWithFood;
}
```

Both of the `boolean` used for this predicate is essential. For instance, if "Laksa is already present" in the `FoodRecord`:

- If the user keys in "Lak", the first `boolean foodStartsWithInputFoodName` ensures that "Laksa" will be suggested to the user.
- If the user keys in "Laksa Spicy", the second `boolean inputFoodNameStartsWithFood` ensures that "Laksa" will be suggested to the user.

The following sequence diagram will explain how the different objects interact to achieve the Real-time Suggestion Feature.

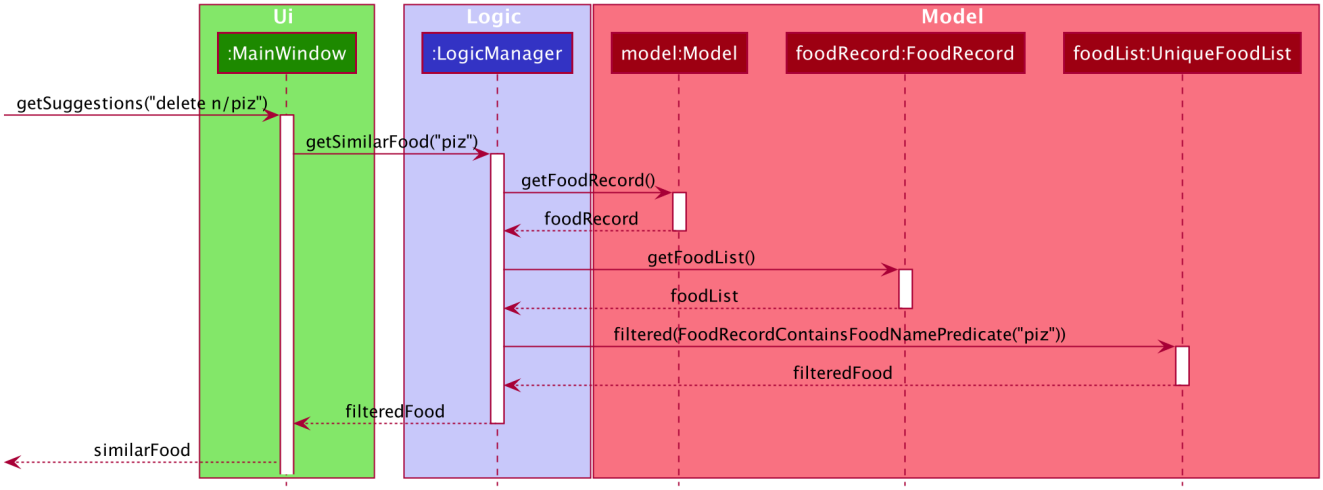


Figure 1. Real-time Suggestion Feature Sequence Diagram

Based on the above diagram, when a user has already entered any of the `CommandWord`: `update`, `delete` or `nom`, and also the Prefix `n/`:

Step 1: `CommandBox` calls the `MainWindow` method of `getSuggestions` with the parameter as the entire `String` of user input in the `CommandBox`.

Step 2: `MainWindow` then parses the user inputted `String` and calls `LogicManager` method of `getSimilarFood` with the parameter `foodName` which is the entire `String` after the `Prefix n/`

Step 3: The `Model` then does the necessary work by calling methods `getFoodRecord` and `getFoodList`. This results in the current `UniqueFoodList` being returned

Step 4: The `UniqueFoodList` is then filtered with the `Predicate<Food>`, `FoodRecordContainsFoodNamePredicate` which returns a `List<Food>` of `Food` objects that have similar `Name` fields to the user input.

Step 5: Finally, the filtered `List<Food>` is then parsed into a `String` for the user by the `MainWindow` and then displayed in the GUI's `Result Display`.

Design Considerations

Aspect: How the suggestions is shown to the user.

- **Alternative 1: (current choice):** `ResultDisplay` displays the names of similar `Food` entries in `Food Record`.
  - Pros:
    - Improved user experience, allowing user to still view the unfiltered `FoodRecord` in the GUI.
    - User can have access to the raw `String` of the `Name` similar `Food` entries for copying and pasting.
  - Cons:

        - Additional interacting with `UI` components required, instead of just filtering `UniqueFoodList`
        - Cannot reusing existing lexicographical sorting feature of `FoodRecord`.
- **Alternative 2:** Filter the GUI's `Food Record` to show similar Food entries.
    - Pros:
        - Feature is limited to minimal interactions with `UI`, making use of existing `UI-Model` abstractions.
        - Compatible with existing code relating to the `FoodRecord`, allowing code to be reused.
    - Cons:
        - Takes away most of the need for `find` and `list` features since they achieve mostly the same purpose.

Aspect: Commands that utilise Real-time Suggestions

- **Alternative 1: (current choice):** Only three commands: `update`, `delete`, `nom`
    - Pros:
        - Improves computational performance, since real-time features for every command will be computationally expensive.
        - Keeps the desired outcomes of other features such as `find` and `list` intact
    - Cons:
        - Decrease in user experience, as they might expect this feature to be universal for all commands
- **Alternative 2** All the commands
    - Pros:
        - Better standardisation of feature across all commands.
    - Cons:
        - Additional computational overhead.
        - Not all commands have a `Name` field.
        - Additional implementation or significant change in how this feature works is necessary to make it universal.

Summary

`CommandBox` listens for any of the three commands as mentioned, allowing `LogicManager` and `FoodRecord` to facilitate the suggestions of similar `Food` entries from the `UniqueFoodList` to display in the GUI's `Result Display`. This can be summarised in the activity diagram below:

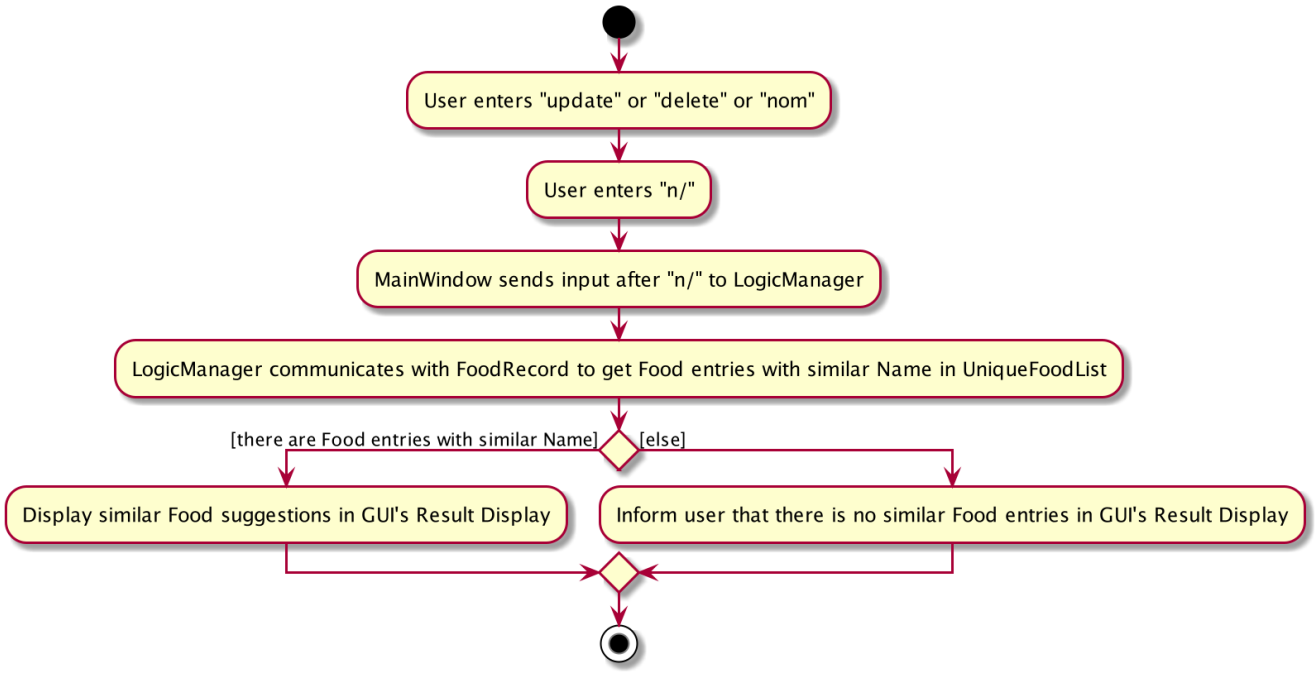


Figure 2. Real-time Suggestion Feature Activity Diagram

Modifying the `FoodRecord`

(By Zhang Xian)

This section addresses how the `FoodRecord` can be modified by the `update` and `delete` commands.

The `update` command allows the user to modify the `FoodRecord` by either adding a new `Food` into the `FoodRecord` or editing the nutritional values of an existing `Food` in the `FoodRecord`.

From the user's perspective, the `update` command does either of the adding and editing functions. This implementation of `update` decides whether to override an existing `Food` in the `FoodRecord` with new values, or create a new `Food` in the `FoodRecord` for them.

For better user experience, for all new `Food` being updated into the `FoodRecord` with the `update` command, the `Name` attribute will be formatted to proper case. This means that if the user updates a new `Food` into the `FoodRecord` with the `Name` as "char kuay teow", the `Food` that is stored in the `FoodRecord` will be of `Name` "Char Kuay Teow".

NOTE	When a new Food is updated into the FoodRecord, the FoodRecord is sorted in lexicographical order. For more information on how this is implemented, please refer to its relevant section here.

The `delete` command allows the user to modify the FoodRecord by deleting a specified `Food` entry from the FoodRecord. This command takes in the `Name` of the `Food` entry to be deleted.

For both `delete` and `update` commands, the `Name` parameter is implemented to be case-insensitive. This means that `n/APPLE` and `n/apple` refers to the same `Food` entry with `Name` stored as `Apple`.

Implementation

The modification of the FoodRecord is facilitated by UniqueFoodList, which is responsible for storing all the Food entries in the FoodRecord. Additional abstractions were used by Model and Logic for any operations that results in a modification of the UniqueFoodList.

Both commands require an additional operation, hasFood, in FoodRecord to be implemented. hasFood checks if there is an existing Food in FoodRecord by checking if there is any Food in the FoodRecord with the same Name. Two Food entries is deemed to be of the same Name if their lowercase variant is the same.

This operation was exposed in the Model interface as hasFood, allowing UpdateCommand and DeleteCommand this functionality.

Implementation of update command:

For the update command, the hasFood operation decides whether UpdateCommand adds a new Food into UniqueFoodList or edits the nutritional values of an existing Food in the UniqueFoodList.

The following sequence diagram shows how the update operation works in both cases:

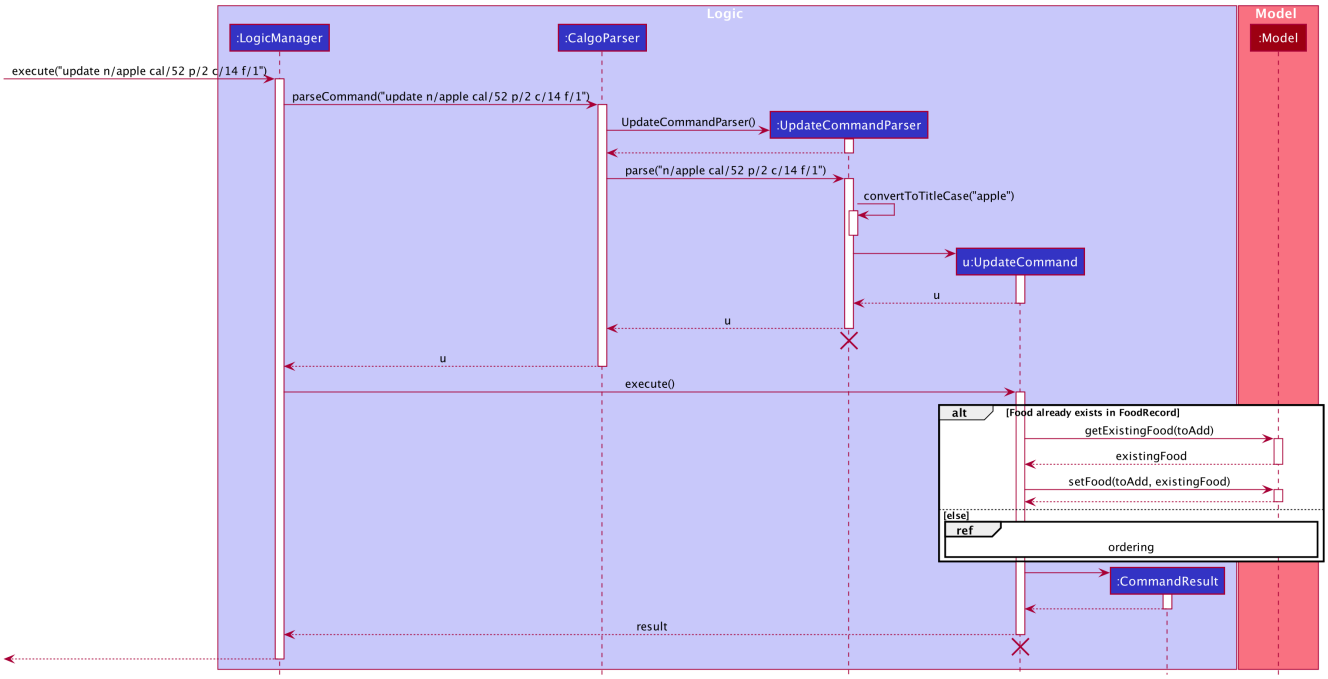


Figure 3. Sequence Diagram for update command

NOTE	The lifeline for UpdateCommandParser and UpdateCommand should end at their destroy markers (X) but due to a limitation of PlantUML, the lifelines reach the end of diagram.

How the update command works:

Step 1: LogicManager executes the user input of update n/apple cal/52 p/2 c/14 f/1", using `CalgoParser to realise this is an update command and creates a new UpdateCommandParser object.

Step 2: UpdateCommandParser then parses the arguments provided by CalgoParser with the parse

method. During this parsing process, `UpdateCommandParser` calls the `covertToTitleCase` method on the `Name` argument, converting it to proper case.

Step 3: `UpdateCommandParser` then creates a new `UpdateCommand` object, which `LogicManager` calls the `execute` method with this object as an argument.

Step 4: `UpdateCommand` now checks if there exists an existing `Food` in the `FoodRecord` by calling `Model` 's `hasFood` method.

Step 5:

- Scenario 1: If Food already exists in the `FoodRecord`:
  - Model calls the `getExistingFood` method with the user inputted `Food` as a parameter to get the existing `Food`, `existingFood` in the `UniqueFoodList`. It thens call the `setFood` method to replace the existing `Food` in the `UniqueFoodList` with the new `Food` which contains new nutritional values.
- Scenario 2: If `Food` does not exist in `FoodRecord`:
  - This scenario is handled by the Lexicographical Ordering feature. Please refer to its relevant section here.
  - Model calls the `addFood` method with the user inputted `Food` as a parameter to add the new `Food` into the `UniqueFoodList` in `FoodRecord`
  - After the `Food` is added into the `UniqueFoodList`, the `UniqueFoodList` is also sorted in lexicographical order.

Step 6: A new `CommandResult` object is then created and returned back to `LogicManager`.

Implementation of `delete` command:

For the `delete` command, the `hasFood` operation allows `UpdateCommand` to check whether the Food that the user requests to be deleted exists in the `UniqueFoodList`.

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

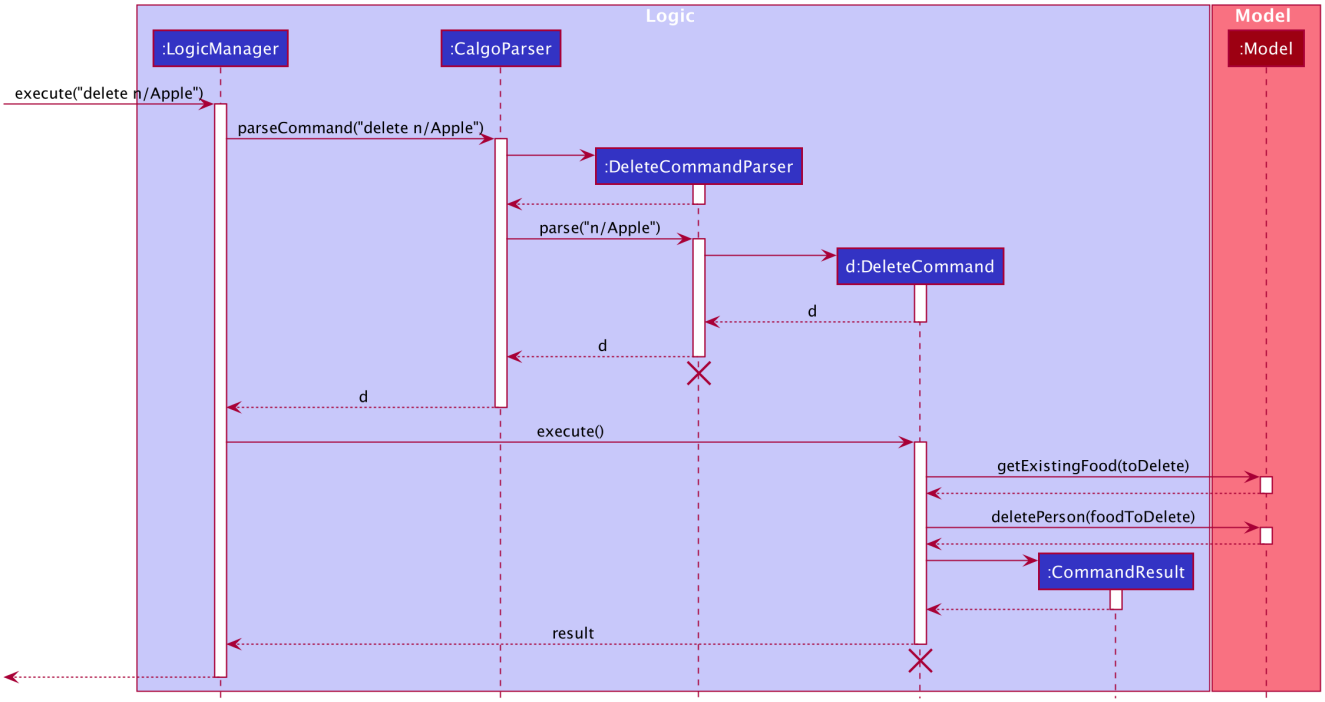


Figure 4. Sequence Diagram for `delete` command

NOTE	The lifeline for `DeleteCommandParser` and `DeleteCommand` should end at their destroy markers (X) but due to a limitation of PlantUML, the lifelines reach the end of diagram.

How the `delete` command works:

Step 1: `LogicManager` executes the user input of "delete n/Apple", using `CalgoParser` to realise this is an `delete` command and creates a new `DeleteCommandParser` object.

Step 2: `DeleteCommandParser` then parses the arguments provided by `CalgoParser` with the `parse` method, before creating a new `DeleteCommand` object that is returned back to the `LogicManager` which calls the `execute` method with this as an argument.

Step 3: `DeleteCommand` now checks if there exists an existing `Food` in the `FoodRecord` by calling `Model`'s `hasFood` method, which checks if there is such `Food` in the `UniqueFoodList`.

Step 4: `Model` then calls the `getExistingFood` method to return the `Food` object to be removed from the `UniqueFoodList`. Thereafter, `Model` calls the `deleteFood` method with this `Food` object as an argument to remove this `Food` from the `UniqueFoodList`.

Step 5: A new `CommandResult` object is then created and returned back to the `LogicManager`.

Design considerations

Aspect: Updating the `FoodRecord` when there is an existing `Food` item in `FoodRecord`

- **Alternative 1 (current choice):** Overrides the existing `Food` item with the new `Food` item
  - Pros:
    - No need for an additional command of `edit` just for the user to edit an existing `Food` item in the `FoodRecord`.
  - Cons:
    - Might not be intuitive for the user since the word "update" is generally assumed to be for editing something only and not necessarily adding something.
    - May result in additional performance overhead.
- **Alternative 2:** Informs the user that there is already an existing `Food` item, and direct him to use another command `edit` to edit the existing `Food` instead.
  - Pros:
    - More intuitive for user, since he might not know that he is overriding an existing `Food` item
  - Cons:
    - Additional command has to be created just to handle editing
    - More tedious for user since more steps are required to achieve the same result.

Summary

In summary, this section explains how commands related to modifying the `FoodRecord` is implemented.

The `update` command is a smart command that either updates an existing `Food` entry in the `FoodRecord` with new nutritional information, or updates a new `Food` item into the `FoodRecord` The following activity diagram summarises what happens when a user enters a valid `update` command:

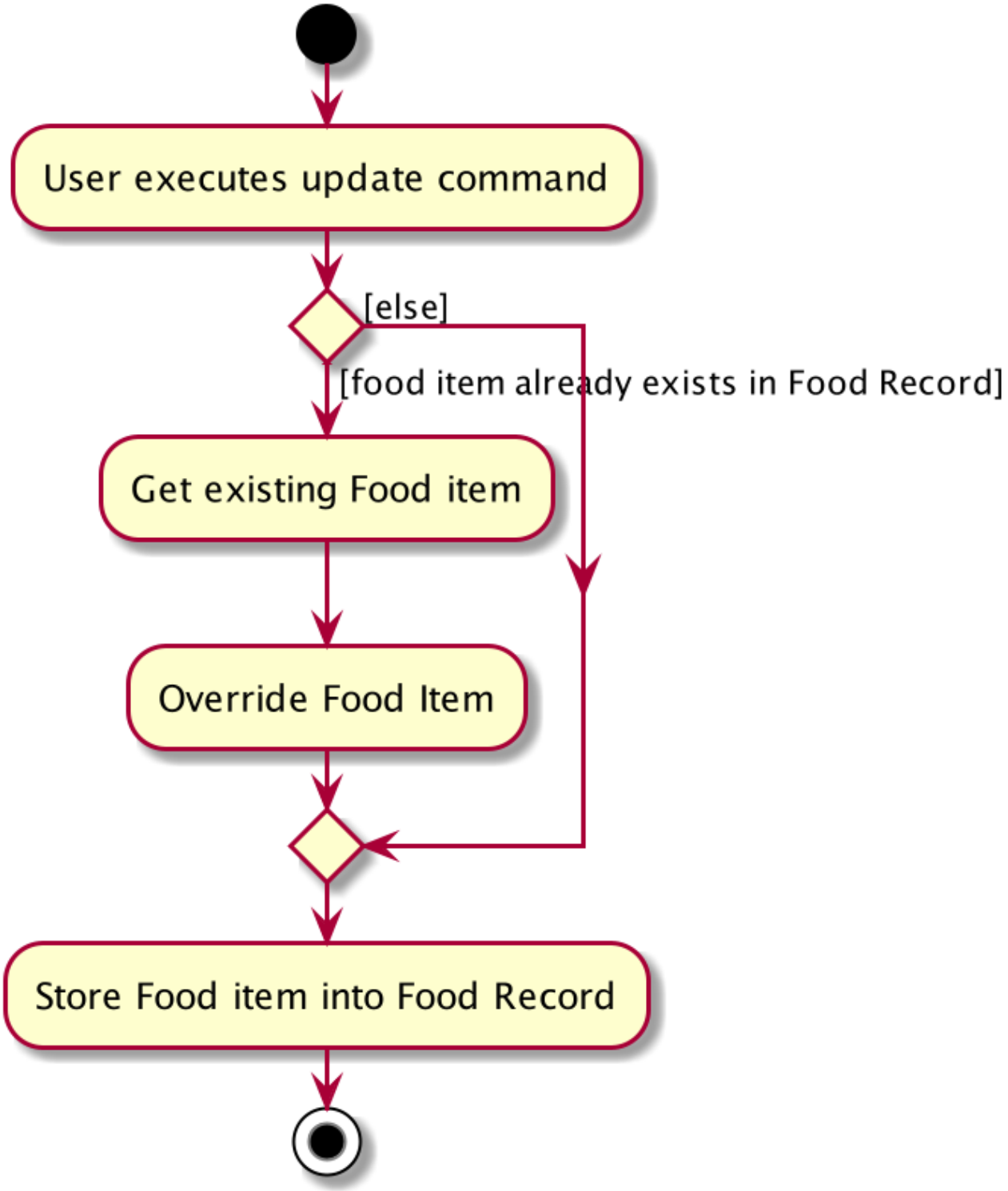


Figure 5. Activity Diagram for `update` command

The `delete` command allows the user to remove a `Food` entry from the `FoodRecord` by specifying it's `Name` as an parameter. The following activity diagram summarises what happens when a user enters a valid 'delete' command:

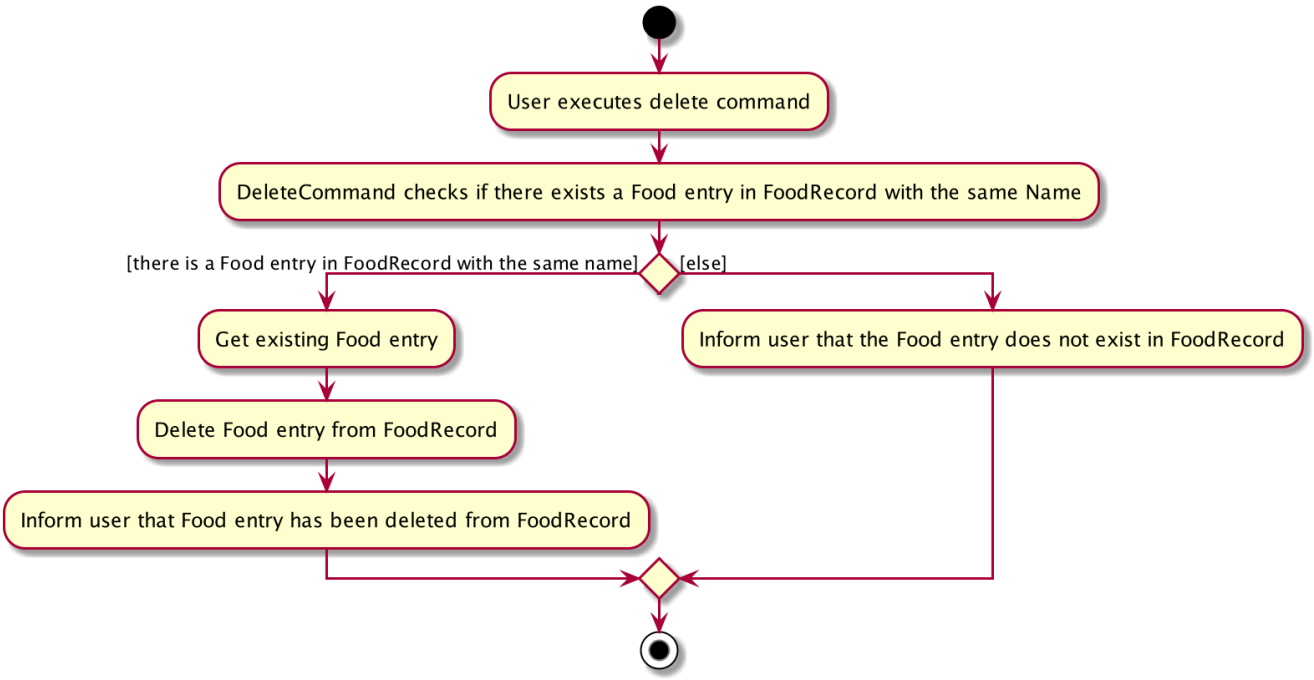


Figure 6. Activity Diagram for `delete` command