Calgo - Developer Guide

By: `Team F11-1` Since: `March 2020` Licence: `MIT`

1. About this guide

This Developer Guide is a document to guide future software developers of the Calgo App by providing a sufficient and comprehensible overview of the project.
While we aim to provide a reasonable amount of depth, the goal of this document is not to serve as

a replacement for reading the actual code.
Welcome on-board the Software Development Team for Calgo!

2. Setting up

Refer to the guide here.

3. Design

3.1. Architecture

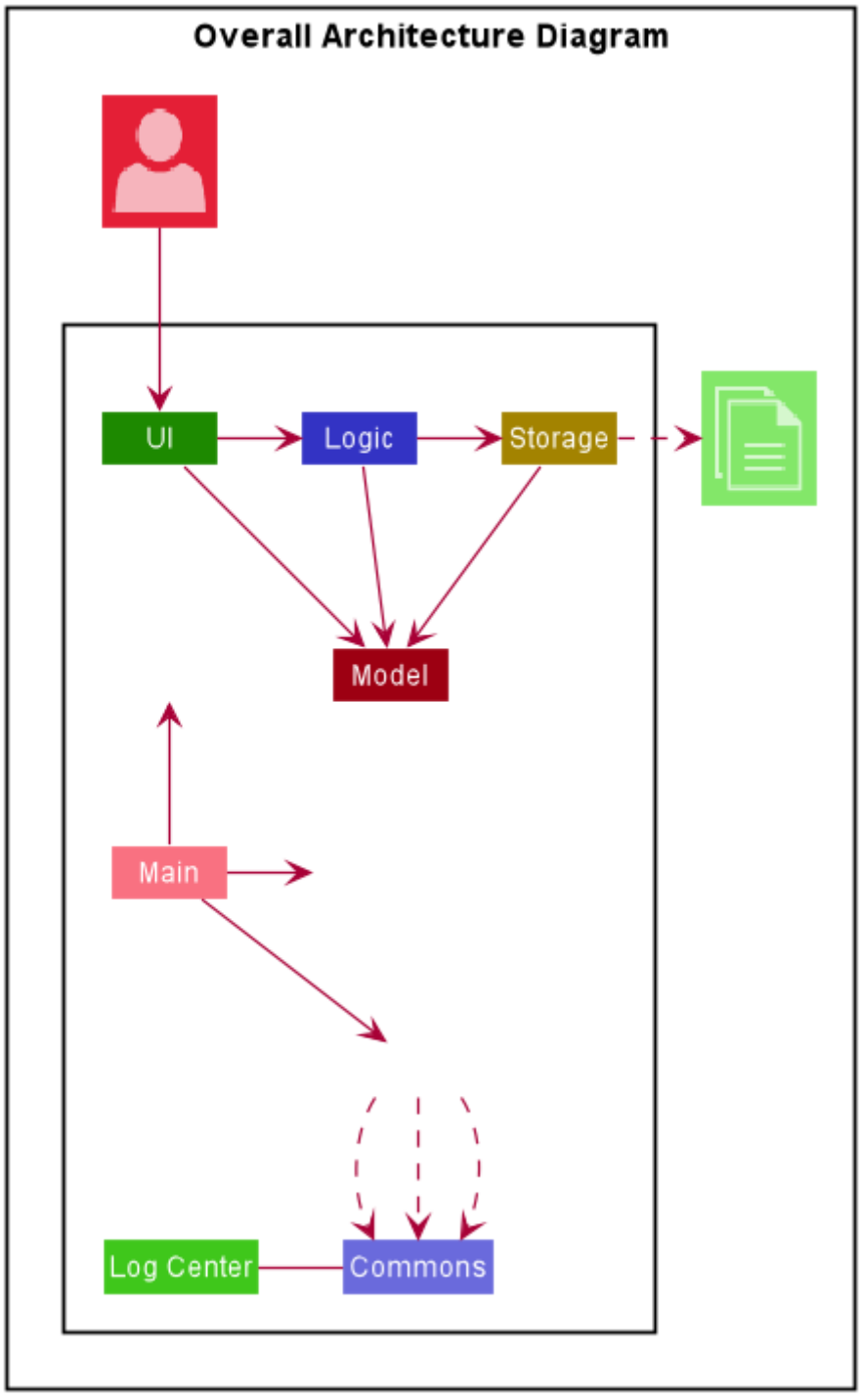


Figure 1. Architecture Diagram

TIP 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.

The ***Architecture Diagram*** given above describes the high-level design of the Calgo Application. From now on, all instances of Calgo Application will be referred to as App. Given below is a quick overview of each component.

The `Main` component comprises of two classes called `Main` and `MainApp`. This component is responsible for:

- Launching App: Initializes the other components in the correct sequence, and connects them up with each other.
- Exiting App: Shuts down the components and invokes cleanup method where necessary.

`Commons` represents a collection of classes used by multiple other components. In particular, the `LogsCenter` class plays an important role at the architecture level:

- `LogsCenter` : Writes log messages to the App's log file, for various classes.

The rest of the App comprises of four components.

- `UI`: The User Interface (UI).
- `Logic`: The command executor.
- `Model`: The in-memory representation of the App data.
- `Storage`: The file manager for reading from and writing to the hard disk.

Each of the four components:

- Defines its *Application Programming Interface (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 its API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

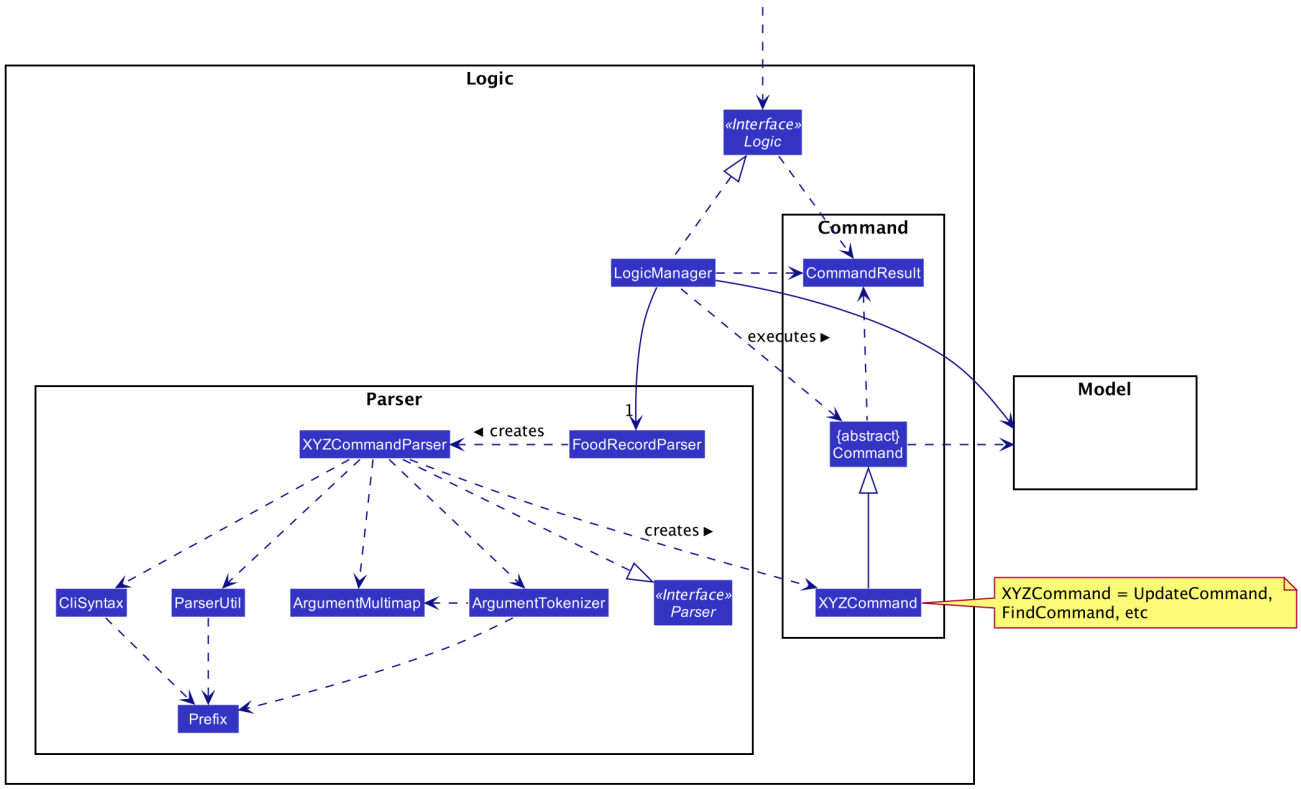


Figure 2. Class Diagram of the Logic Component

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 n/Apple`.

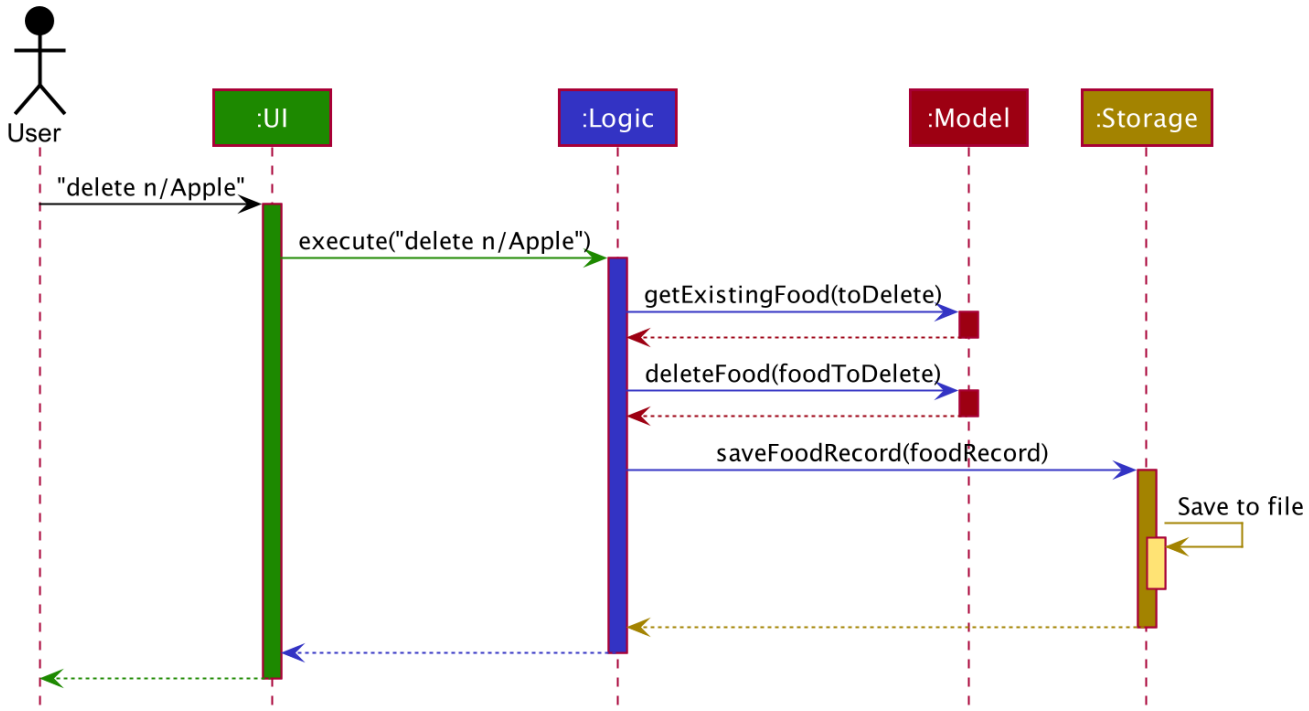


Figure 3. Component interactions for `delete n/Apple` command

The sections below give more details of each component.

3.2. UI component

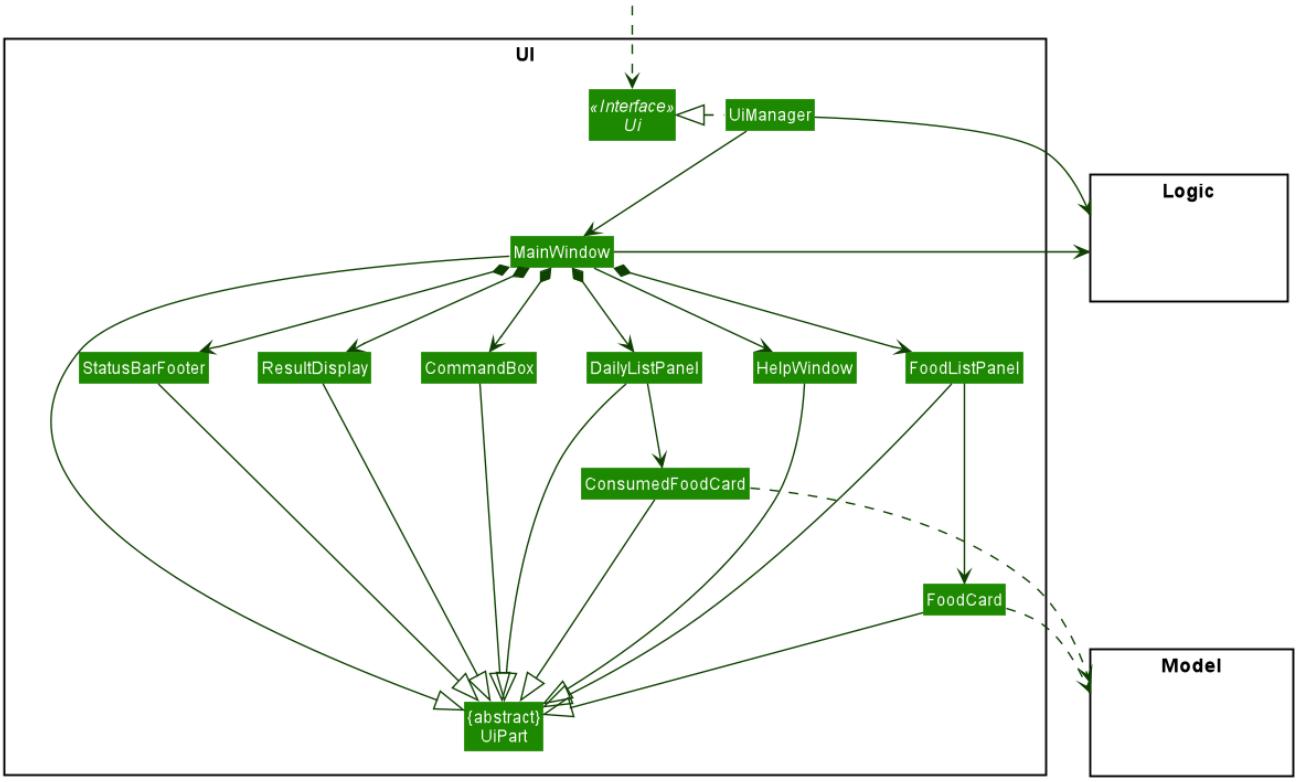


Figure 4. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `FoodListPanel`, `DailyListPanel`, `StatusBarFooter` etc. 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:

1. Executes user commands using the `Logic` component.
2. Listens for changes to `Model` data so that the UI can be updated with the modified data.

3.3. Logic component

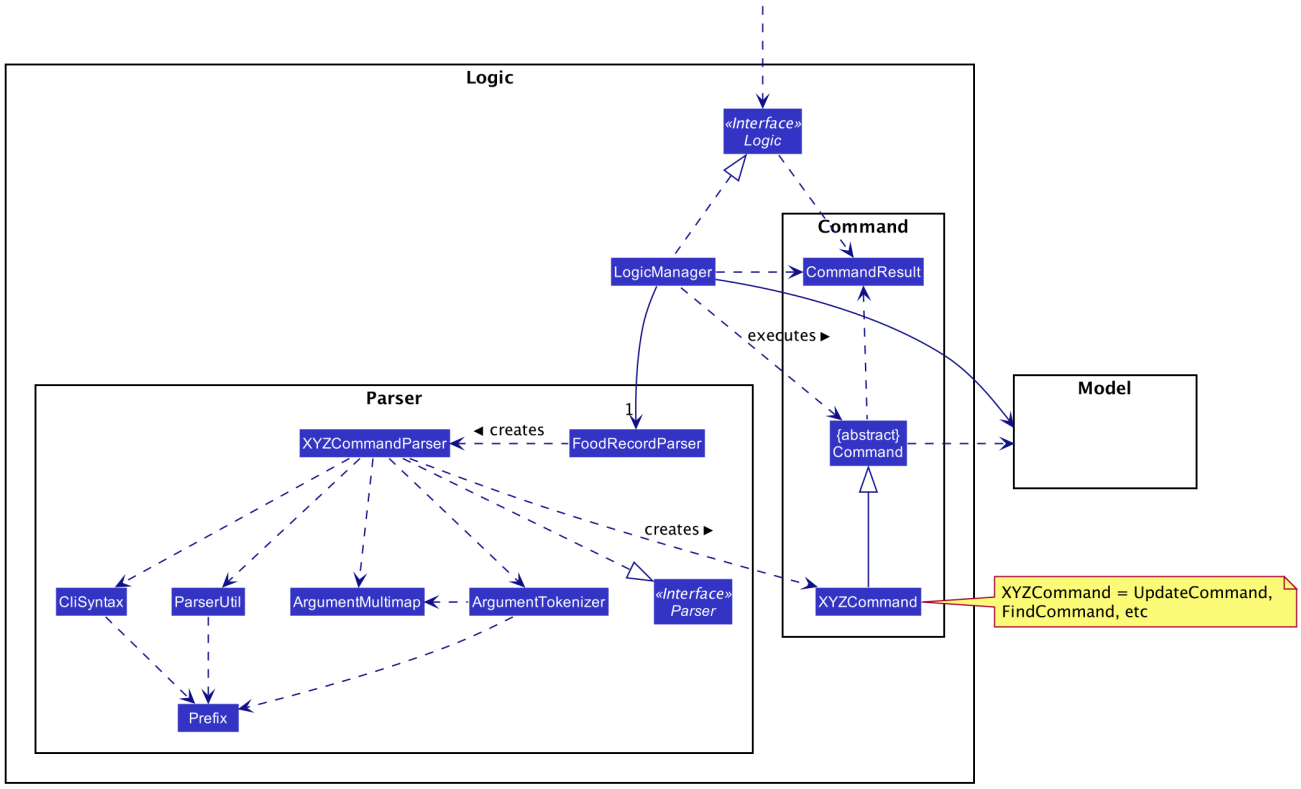


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `FoodRecordParser` 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 food).
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 n/Apple")` API call.

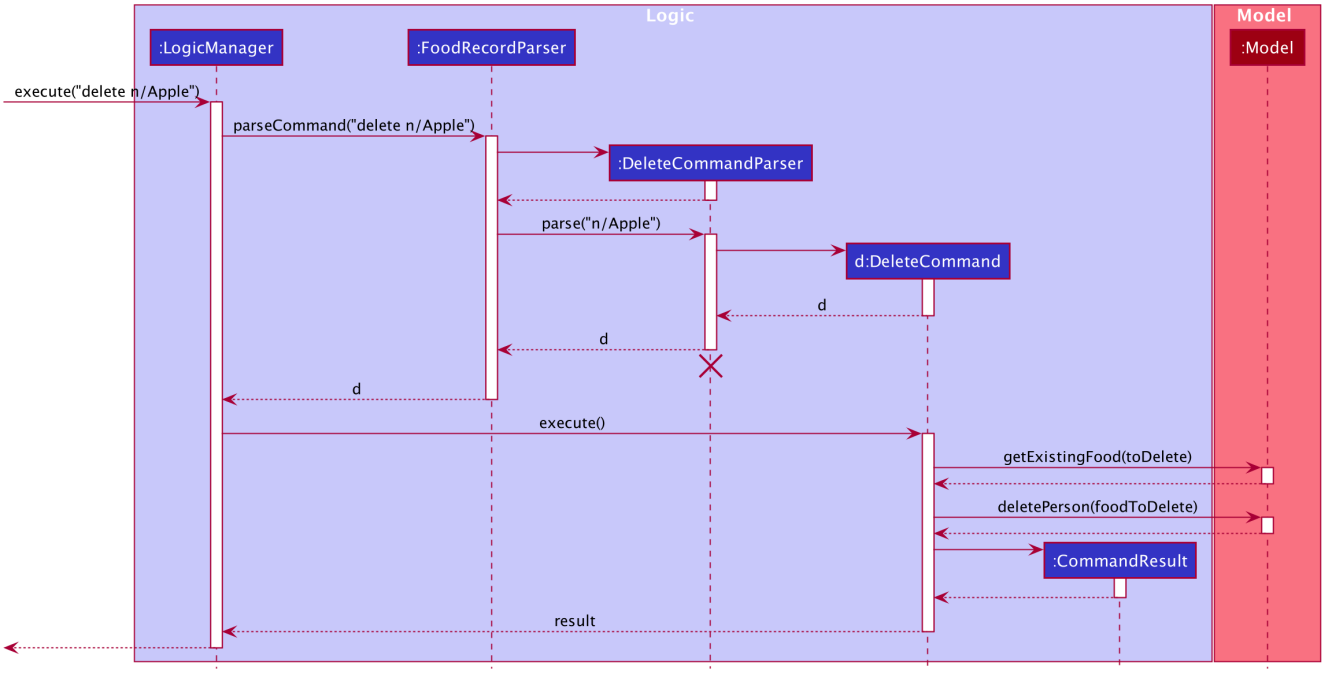


Figure 6. Interactions Inside the Logic Component for the `delete n/Apple` Command

NOTE 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.

3.4. Model component

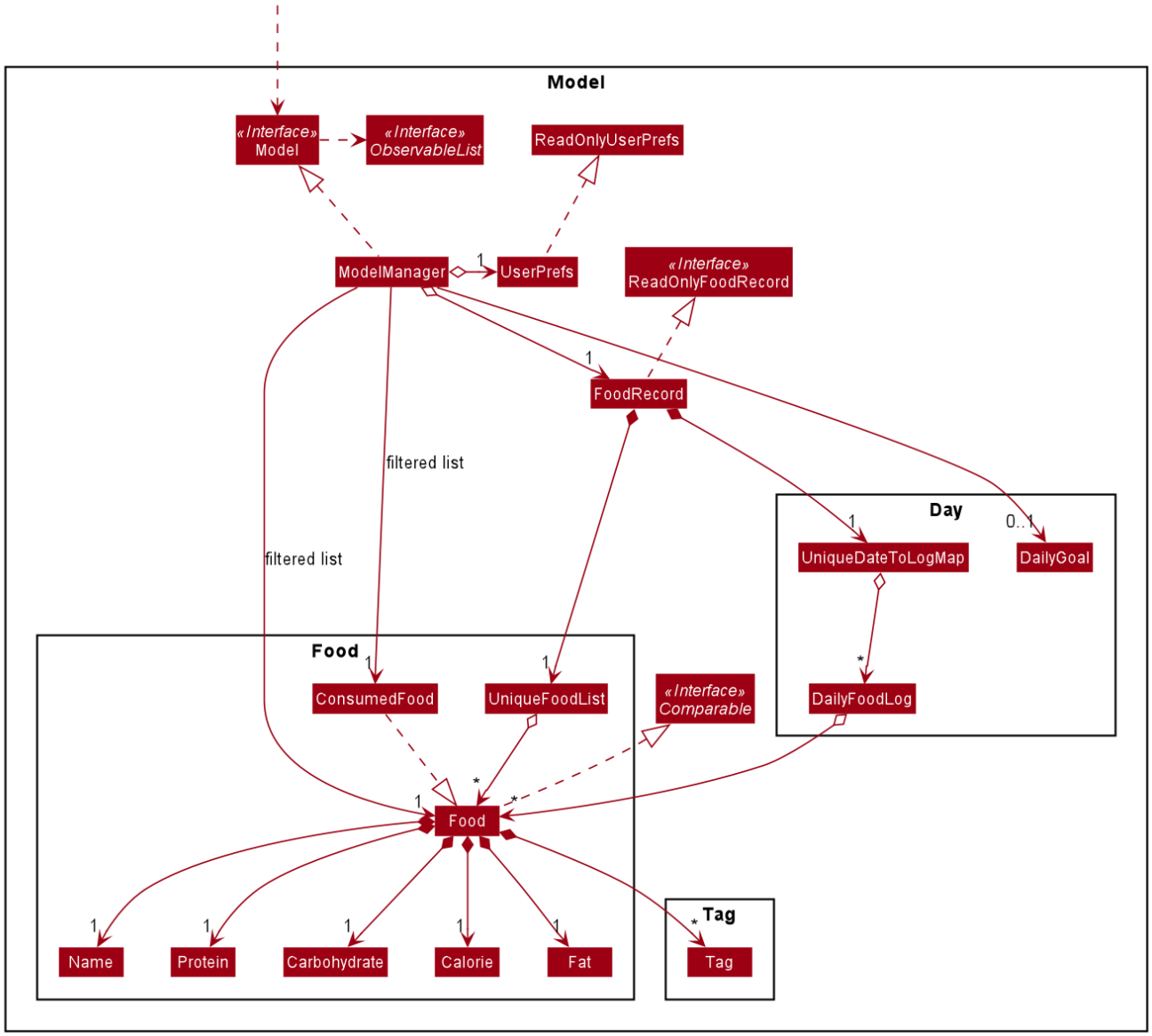


Figure 7. Structure of the Model Component

API : `Model.java`

1. `Model` stores user's preferences in a `UserPref` object.
2. `Model` also stores Food Record data.
3. This component exposes both `ObservableList<Food>` and `ObservableList<ConsumedFood>`. The data stored in these two list objects is reflected in UI. Therefore, any changes made to the data in these lists are shown in the UI in real-time.
4. To update the `Model` (and hence reflect the changes in the UI), `Food` attributes need to satisfy certain `Predicates`, which represent these changes.
5. This component does not depend on any of the other three components.

NOTE

To make `Model` follow the Object Oriented Programming (OOP) Paradigm more closely, we can store a `Tag` list in `Food Record`, which `Food` objects can reference. This would allow `Food Record` to only require one `Tag` object per unique `Tag`, instead of each `Food` needing their own `Tag` object. An example of how such a model may look like is given in the below diagram.

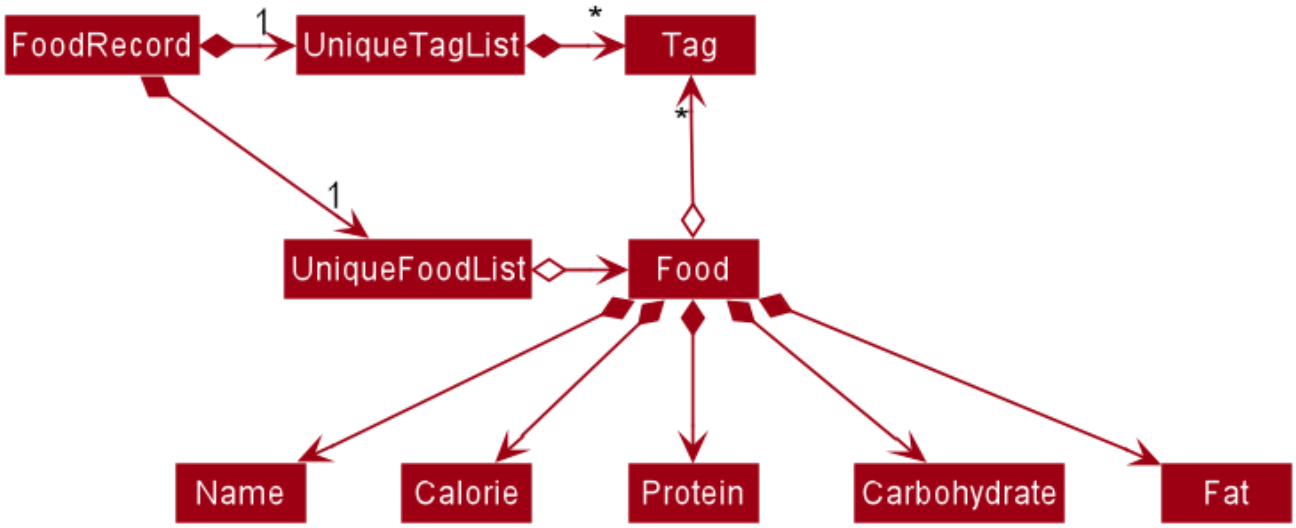


Figure 8. Structure of the Model Component

3.5. Storage component

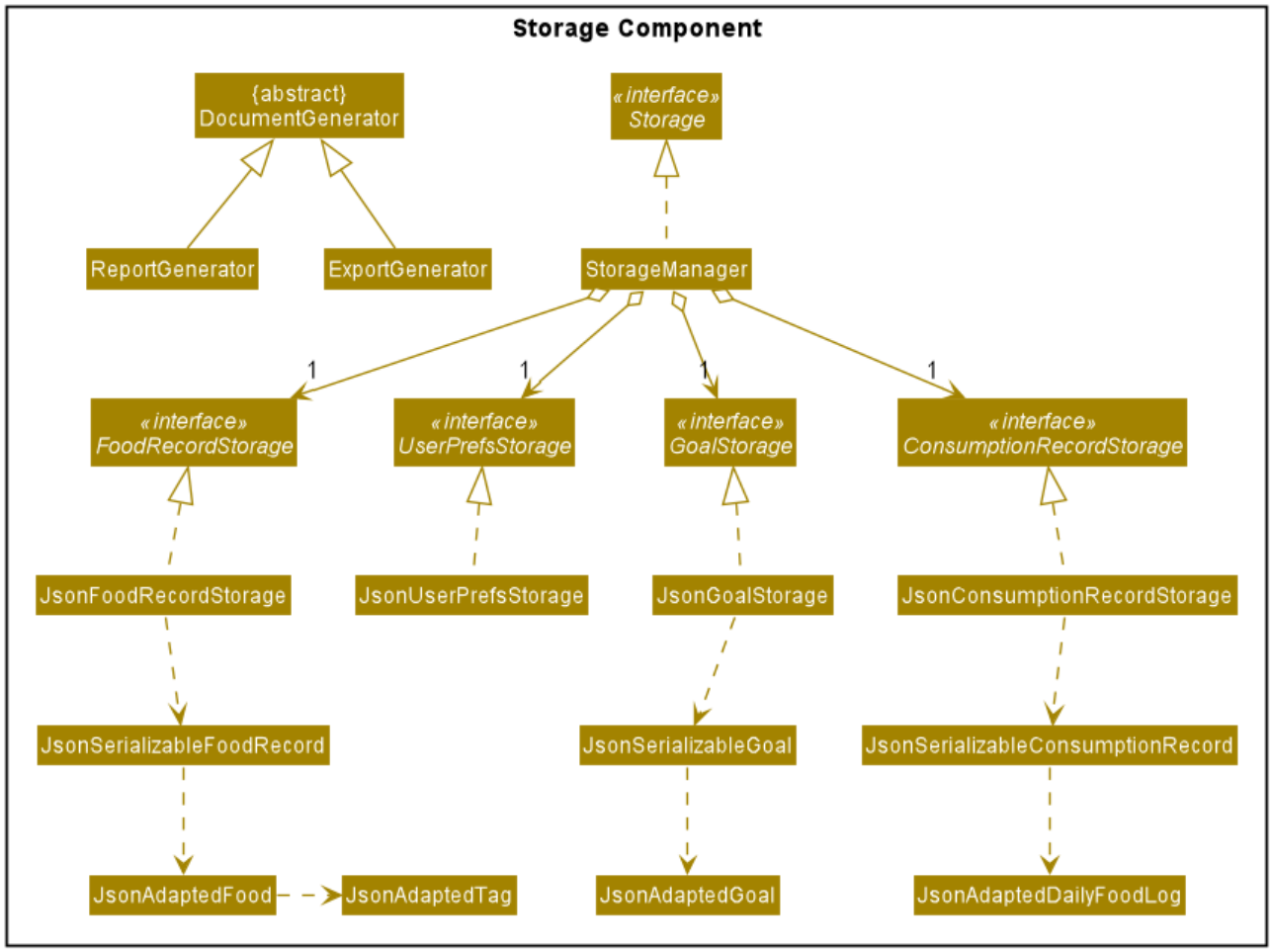


Figure 9. Structure of the Storage Component

API : `Storage.java`

The `Storage` component allows us to save `FoodRecord`, `UserPref`, `Goal`, and `ConsumptionRecord` data in json format onto the disk, and read them back later on during the next session.

This would facilitate the following functions:

1. Load past user App data and preferences.
2. Generate and save insights reports based on previously and currently recorded user consumption.
3. Generate and save a user-friendly version of the accumulated `FoodRecord`.

3.6. Common classes

Classes used by multiple components are in the `life.calgo.commons` package.

4. Implementation

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

4.1. Configuration

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

4.2. Command guide `help` command

4.2.1. Implementation

As with any application with a plethora of commands, it is useful to have an in-app and offline method by which users can view the purpose and usage format of each command.

This help feature is a functionality that is carried out by the FoodRecordParser to guide users on how to utilise the App's commands. The guide is displayed in a separate window, as handled by HelpWindow.

With this, a top-level idea of the execution of the help command is given in the sequence diagram below:

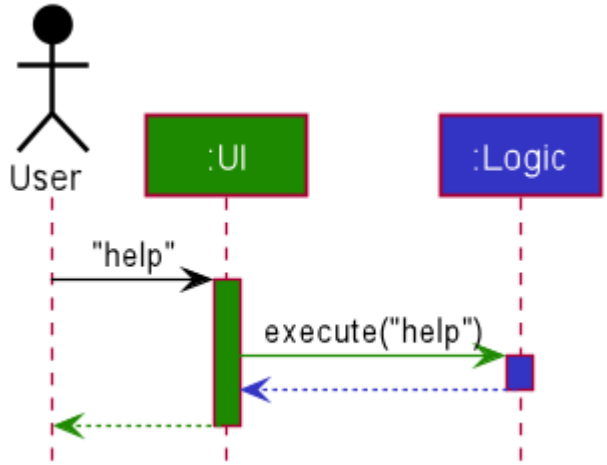


Figure 10. Sequence Diagram for Help Command

Step 1: `LogicManager` takes in the user input of "help".
Step 2: `FoodRecordParser` is passed the String input and is parsed using `parseCommand`.
Step 3: This results in a `HelpCommand` object which is executed by the `LogicManager`.
Step 4: The `LogicManager` encapsulates the result as a `CommandResult` object which is passed back to the `MainWindow`.
Step 5: The `MainWindow` executes the handleHelp() method, displaying the `HelpWindow` if it is not already being displayed.
Step 6: `HelpWindow` is displayed as a separate popup.

4.2.2. Design considerations

Aspect: How Help is displayed

- **Alternative 1 (current choice):** `HelpWindow` is displayed as a self-contained popup.
  - Pros: User can refer to the command guide in a window separately from the main app. Additionally, no internet access is required as all information on commands are stored offline.
  - Cons: As `help` does not redirect to a url containing the most up-to-date User Guide, any changes to command functionality or addition of new commands must be updated for local display.
- **Alternative 2:** `HelpWindow` is not used, and instead content is displayed as part of `ResultDisplay`.
  - Pros: No possibility of a popup blocking the main app, and all information is contained within a single window.
  - Cons: User must use the `help` command every time they require a guide, as `ResultDisplay` will be overwritten after every command.

4.2.3. Summary

`help` will produce a popup, displaying a guide on the App's available commands' purposes and usage format.

4.3. `Food` consumption management

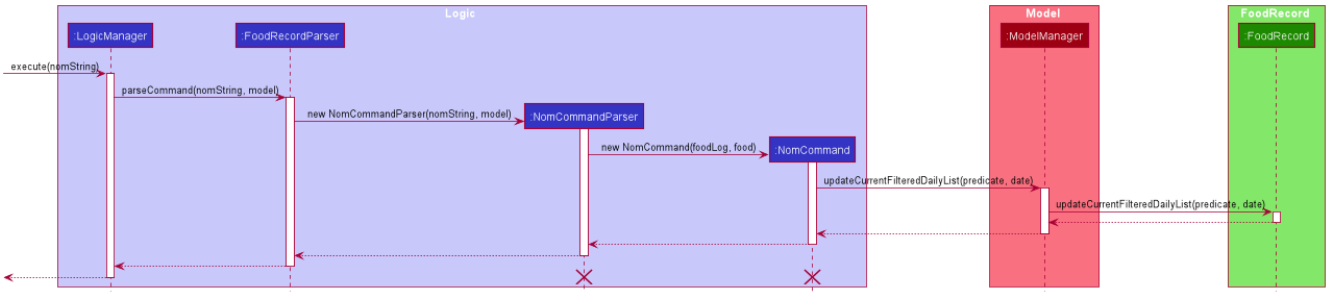
In Calgo, you will find that there is a date associated with each list of `ConsumedFood`. When adding food to be consumed, removing food, or displaying food consumed on certain days, a `FilteredList` will be populated with relevant `ConsumedFood`.

4.3.1. Implementation

The `nom`, `vomit` and `stomach` commands are facilitated by the `FoodRecord`.
`FoodRecord` contains a `UniqueDateToLogMap`, which maps a `LocalDate` to a `DailyFoodLog`.
`DailyFoodLog` contains a `LinkedHashMap` storing `Food` in the sequence that they were consumed and maps those `Food` to a `Double` portion.

This section covers how the `nom` command is implemented. The `vomit` and `stomach` commands work in very similar way, hence their implementation is omitted for brevity.

A top-level illustration of the execution of a `nom` command is given in the sequence diagram below:



Step 1: User enters a command, which is saved as a `String` and passed into the `LogicManager`.
Step 2: The `String` cascades down the layers of abstraction until `NomCommandParser` handles it and creates a `DailyFoodLog` which reflects the consumption.
Step 3: A `NomCommand` is created and executed, updating both `ModelManager` and `FoodRecord` about the consumed food.
Step 4: A `FilteredList` in `ModelManger` will then check with `FoodRecord` to create `ConsumedFood` items to display in the Graphical User Interface (GUI).
Step 5: The GUI automatically detects changes in `FilteredList` and refreshes to display updated content.

4.3.2. Design considerations

Aspect: How `nom` executes

- **Alternative 1 (current choice):** Create a new `DailyFoodLog` to pass into `ModelManager` and `FoodRecord`.
  - Pros: Maintain comprehensive layers of abstraction and allows code to be easily testable.
  - Cons: Difficult for newcomers or even existing users to trace because of long execution path.
- **Alternative 2:** Bypass `ModelManager` or even not use `FoodRecord` for storage of data during runtime by allowing everything to be done from parser.
  - Pros: Reduce dependencies on `ModelManager` and `FoodRecord`, and make code contained in a single class file easier to navigate.
  - Cons: Violates layers of abstraction set in place by previous structure of AddressBook3. Violates Single Responsibility Principle and reduce cohesiveness of code.

Aspect: Data structure to support the consumption commands

- **Alternative 1 (current choice):** Use a single `FilteredList` to store food for any day by repopulating it each time a consumption related command is used.
  - Pros: Only uses a single `FilteredList`, so it is clear which list you are using for display.
  - Cons: May have performance issue in terms of speed when there are too many entries.
- **Alternative 2:** Use a `FilteredList` for each date, to store food consumed on that date.
  - Pros: Faster retrieval for display of `ConsumedFood` items. However, under practical circumstances, the difference is negligible.
  - Cons: May have performance issue in terms of storage because it requires many lists to be stored in addition to `LinkedHashMap` in `DailyFoodLog` for each `LocalDate`.

4.3.3. Summary

The `nom` command adds a `Food` item consumed by the user into the `stomach`. The following activity diagram summarizes what happens when the user executes a `nom` command.

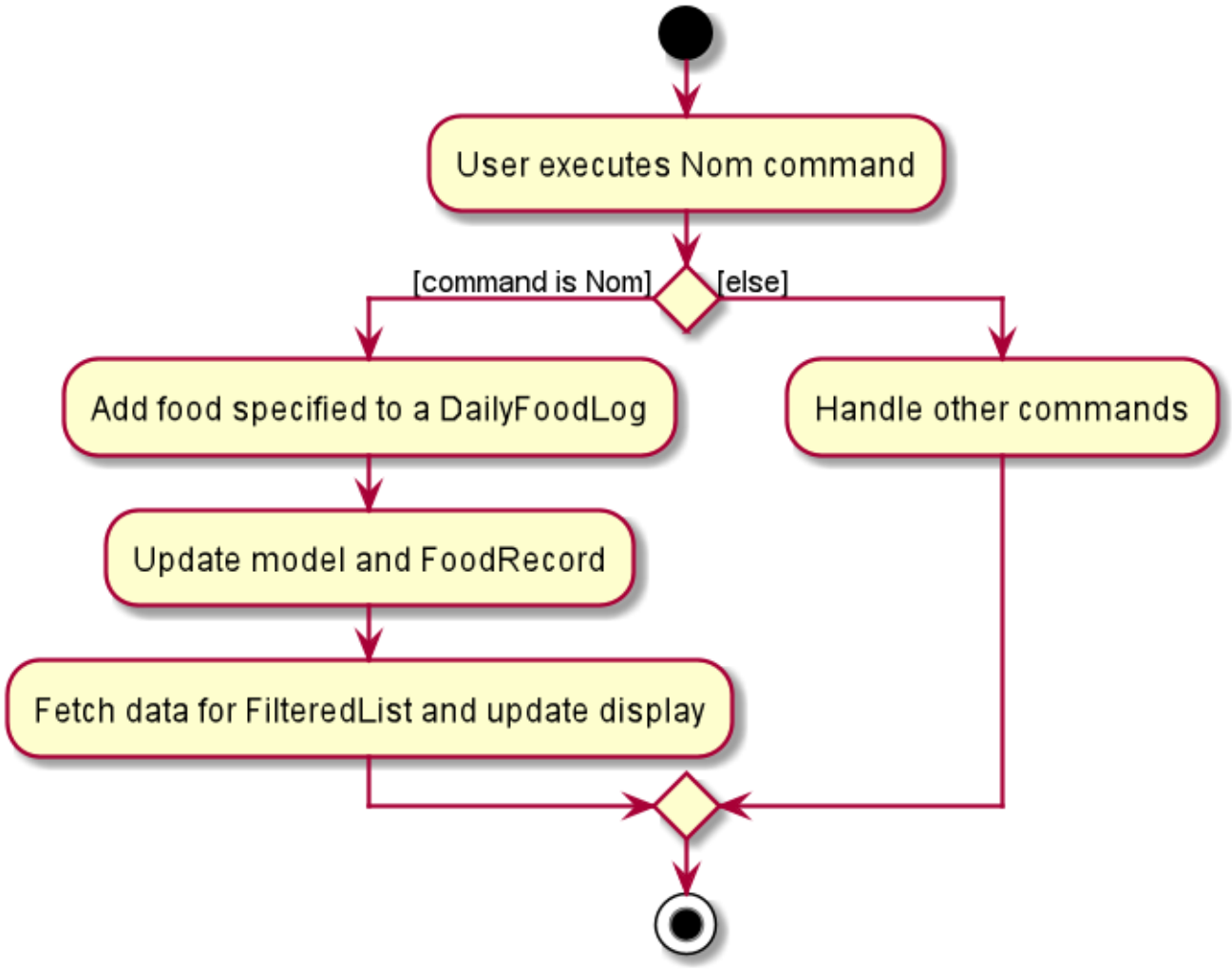


Figure 11. Activity Diagram for Nom

4.4. Generate insights report

This feature allows a user to generate a report that contains statistics and deliverable insights based on personal food consumption patterns.

The functionality can be invoked by entering the `report d/DATE` command. This command generates a report that is based on the food consumed by the user on the specified date.

4.4.1. Implementation

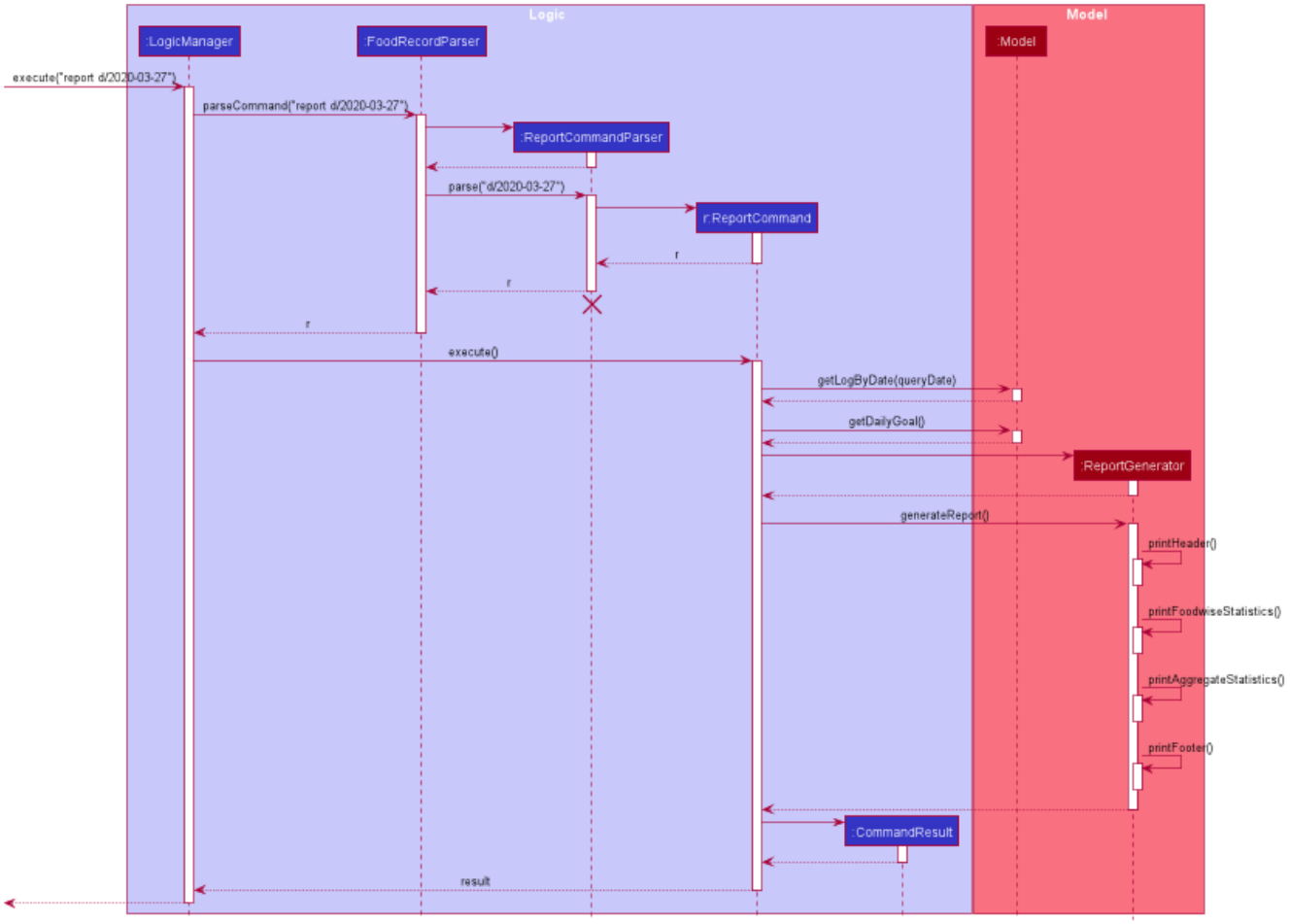
The specified feature is facilitated by `ReportGenerator` class in the `Storage` component. In this section, the implementation features of the `ReportGenerator` class will be further explained.

`ReportGenerator` class implements the following operation:

- `ReportGenerator#generateReport()` - Creates report containing an analysis of all food consumed by user on the given date when inputting the `report` command.

Whenever the `report d/DATE` command is given by the user, the `ReportGenerate#generateReport()` operation is called.

The following sequence diagram illustrates the top-level execution of the `generateReport()` operation:



Step 1: User inputs `report d/2020-03-27` to generate the insights report based on food consumption of 27 March 2020.

Step 2: This input is saved as a `String` and passed into the `LogicManager`.

Step 3: The `String` input is parsed by `FoodRecordParser`, which removes the "d/" prefix tag and sends the date input to `ReportCommandParser`.

Step 4: Once the `ReportCommandParser` checks that the given date is valid, it creates a `ReportCommand` object and returns it to `LogicManager`.

Step 5: `LogicManager` then executes the `ReportCommand`.

Step 6: From `Model`, `ReportCommand` retrieves the `DailyFoodLog` object that stores all `Food` consumed on the input date.

Step 7: From `Model`, `ReportCommand` also retrieves `DailyGoal` object, which stores the daily number of calories the user wants to consume.

Step 8: With the relevant objects retrieved from Steps 6 and 7, `ReportCommand` constructs a `ReportGenerator` object.

Step 9: Using the `ReportGenerator` object, `ReportCommand` calls `#generateInsights()`, which prints

metainformation , food-wise statistics, aggregate statistics and insights based on the `DailyFoodLog` of the input date.

Step 10: This newly generated report is saved in the `/reports` folder. If the report is successfully generated, the `CommandResult` is true. Otherwise, it is false. This `CommandResult` object is finally returned to `LogicManager`, to signify the end of the command.

4.4.2. Design considerations

Aspect: How generate report executes

- **Alternative 1 (current choice):** Print insights into a .txt file.
  - Pros: The implementation allows users to easily edit the contents of the report should they have realised they did not log in certain food items on that day.
  - Cons: Users could cheat by modifying values in the report. This defeats the purpose of the report to improve their self-awareness of their food consumption patterns.
- **Alternative 2:** Print insights into a pdf file.
  - Pros: The insights appear more legitimate and neatly formatted.
  - Cons: Requires use of external libraries, which occupy memory of the App. PDF files generally require more memory than .txt files as well.

4.4.3. Summary

The following activity diagram summarizes what happens when user executes a `report d/DATE` command:

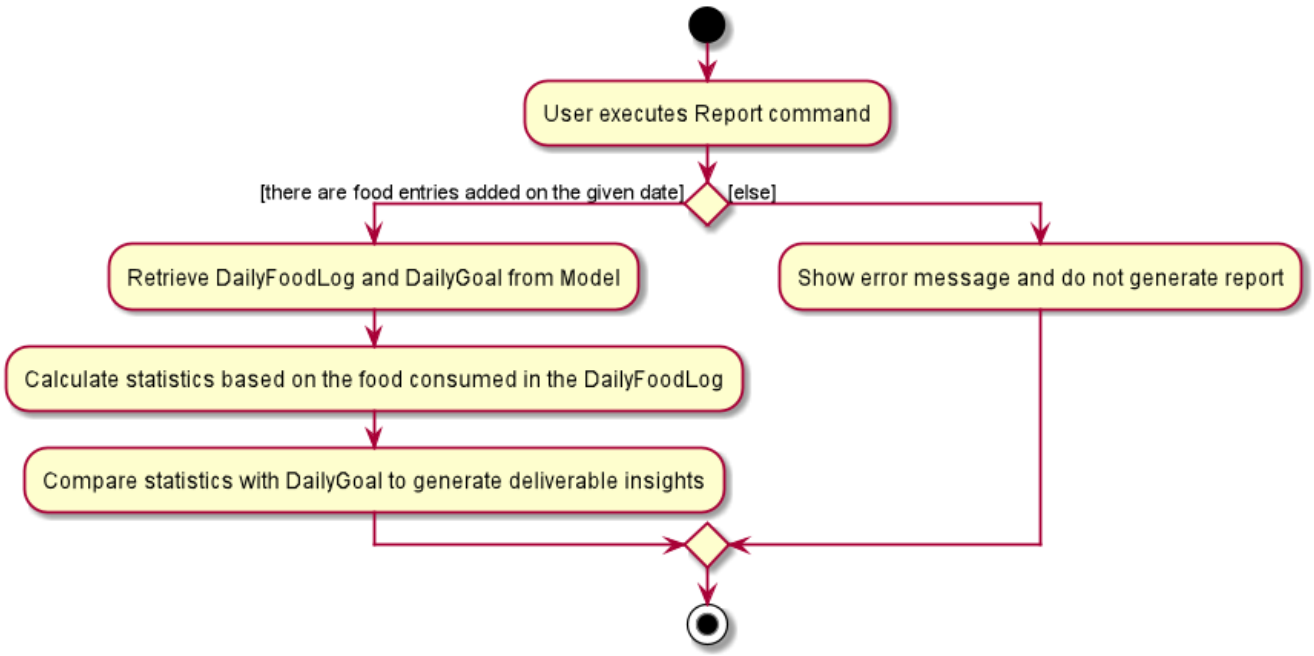


Figure 12. Activity Diagram for Report command

4.5. Lexicographical `Food` order

(by Eugene)

This section addresses how the GUI `Food Record` entries appear in lexicographical order, which is an effect of sorting `Food` objects in the `FoodRecord`.

Over time, users will eventually have many `Food` entries — these should be sorted for a better experience. Intuitively, the lexicographical order is the most suitable here.

In essence, `Food` objects are sorted by the `UniqueFoodList` (which is inside `FoodRecord`). Sorting is performed each time `Food` object(s) are newly added to the `UniqueFoodList`, edited by the user, or when the `UniqueFoodList` is initialised during App start-up. There is no need to re-sort during deletion as the order is maintained.

NOTE	For a better understanding of adding and editing `Food` objects using the `update` command, please refer to its relevant section here.
NOTE	Although the the `list` command changes the GUI `Food Record` display, it does not actually perform sorting. It simply resets the GUI `Food Record` to show all `Food` entries, and is usually used after a `find` command.

4.5.1. Implementation

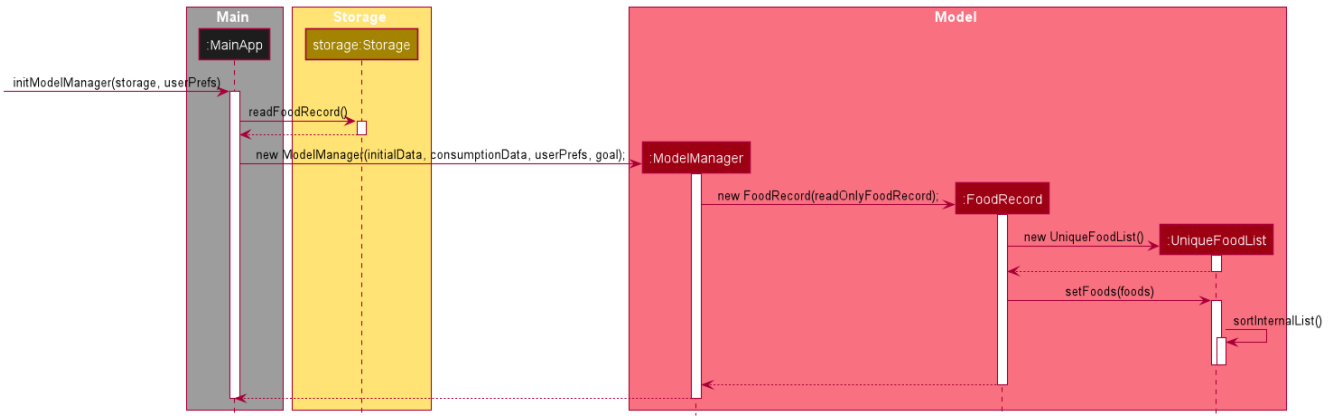
The `UniqueFoodList` is able to sort `Food` objects because the `Food` class implements the `Comparable<Food>` interface. This allows us to specify the lexicographical order for sorting `Food` objects via their `Name`, using the following compareTo method in the `Food` class:

```
public int compareTo(Food other) {
    String currentName = this.getName().toString();
    String otherName = other.getName().toString();
    return currentName.compareTo(otherName);
}
```

How the sorting process works:

- When the App starts up, a new `UniqueFoodList` is created from the source json file (if available) or otherwise the default entries, and the created `Food` objects are sorted as they are added to it.
- Existing `Food` objects are therefore arranged in lexicographic order by `Name`.
- Thereafter, `UniqueFoodList` sorts the `Food` whenever they are added or edited in the `Model`.

The sequence diagram below shows how the lexicographical ordering is performed when Calgo starts up:

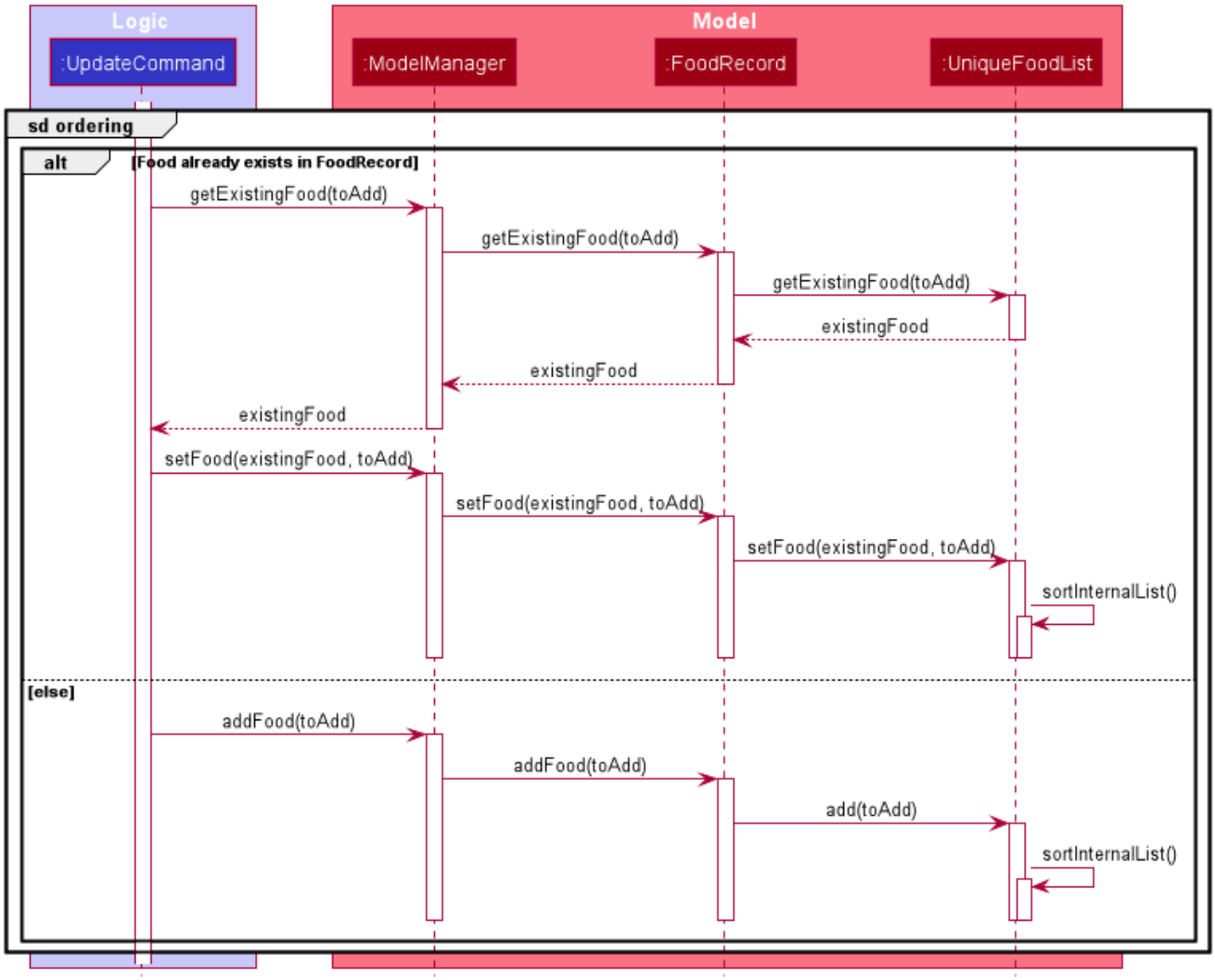


Lexicographical Ordering Sequence Diagram for App Start-up

Based on the above diagram, when Calgo starts:

Step 1: We initialise the `ModelManager`. For this, we use previously stored user data if available. Otherwise, we use the default Calgo `Food` entries.
Step 2: In creating a `ModelManager`, we require the creation of a `FoodRecord` which requires the creation of a `UniqueFoodList`.
Step 3: The next step in the creation of the `FoodRecord` is the introduction of the initialising data, into the `UniqueFoodList`. Here, the `sortInternalList` method sorts the `ObservableList<Food>` contained in `UniqueFoodList` according to the specified lexicographical order defined in the `Food` class.

The sequence diagram below (a reference frame omitting irrelevant `update` command details) describes the main sorting process when `Food` objects are added or edited using the `update` command:



Lexicographical Ordering Sequence Diagram for Updating (Note: this is in a reference frame as it is reused in the `update` section here)

Based on the above diagram, after parsing the user input and creating an `UpdateCommand` object:

- If the user-entered `Food` already exists in `UniqueFoodList`:
    - Step 1: `UpdateCommand` calls `getExistingFood` method of `ModelManager` for the user-entered `Food`, which then calls that of `FoodRecord`, and subsequently that of `UniqueFoodList` to eventually obtain an existing `Food` object with an equivalent `Name`.
    - Step 2: Using the same sequence of classes, we call the respective `setFood` methods,

eventually setting the desired `Food` object and arriving at the `sortInternalList` method of `UniqueFoodList`.

    - Step 3: The `sortInternalList` method then sorts the `ObservableList<Food>` contained in `UniqueFoodList` according to the specified lexicographical order defined in the `Food` class.

- Otherwise, the user-entered `Food` is an entirely new `Food` object:
    - Step 1: Using the same sequence of classes as the former case, we call the respective `addFood` and `add` methods of the classes, eventually adding the `Food` object and arriving at the `sortInternalList` method of `UniqueFoodList`.
    - Step 2: The `sortInternalList` method then sorts the `ObservableList<Food>` contained in `UniqueFoodList` according to the specified lexicographical order defined in the `Food` class.

Any re-ordering will eventually be reflected in the GUI using the following (or its similar):

```
model.updateFilteredFoodRecord(Model.PREDICATE_SHOW_ALL_FOODS);
```

This allows for the GUI `Food Record` to be updated in real-time, once the user makes the changes to the `Model`.

4.5.2. Design considerations

Aspect: Frequency of sorting operation

- **Alternative 1 (current choice):** Sort whenever a new `Food` is added or edited.
  - Pros:
    - Guarantees correctness of sorting.
    - Computational cost is not too expensive since the introduced `Food` objects usually come individually rather than as a collection (except during App start-up).
    - We save computational cost by not sorting during deletion as the order is maintained.
  - Cons:
    - Need to ensure implementations of various commands changing the `Model` are correct and do not interfere with the sorting process.
    - May be computationally expensive if there are many unsorted `Food` objects at once, which is possible when Calgo starts up.
- **Alternative 2:** Sort only when calling the `list` command.
  - Pros:
    - Easier to implement with fewer existing dependencies.
    - Uses less computational resources since sorting is only done when `list` command is called.
  - Cons:
    - User experience is diminished.
    - May lead to bugs in overall product involving order of `Food` objects.
    - May be incompatible with certain `Storage` functionalities.

Aspect: Data structure to store `Food` objects

- **Alternative 1 (current choice):** Use `UniqueFoodList` to store all `Food` objects.
  - Pros:
    - Any changes to the `Model` are automatically reflected in the GUI. This is very useful for testing and debugging manually.
    - Do not need to maintain a separate list, simply reusing what is already in the codebase.
  - Cons:
    - Many of the underlying `ObservableList` methods are built-in and cannot be edited. They are also difficult to understand for those unfamiliar. This can make development slightly trickier, especially in following certain software engineering principles.
- **Alternative 2:** Use a simpler data structure like an `ArrayList`.
  - Pros:
    - Easy for new Computer Science student undergraduates to understand, who are likely to

be the new incoming developers of our project.

- Cons:
  - More troublesome as we require self-defined methods, abstracted over the existing ones. If not careful, these self-defined methods can possibly contain violations of certain software engineering principles, which may introduce regression in the future.

4.5.3. Summary

The `UniqueFoodList` facilitates the lexicographical ordering of `Food` objects and hence their appearance in the GUI `Food Record`. This can be summarised in an activity diagram below:

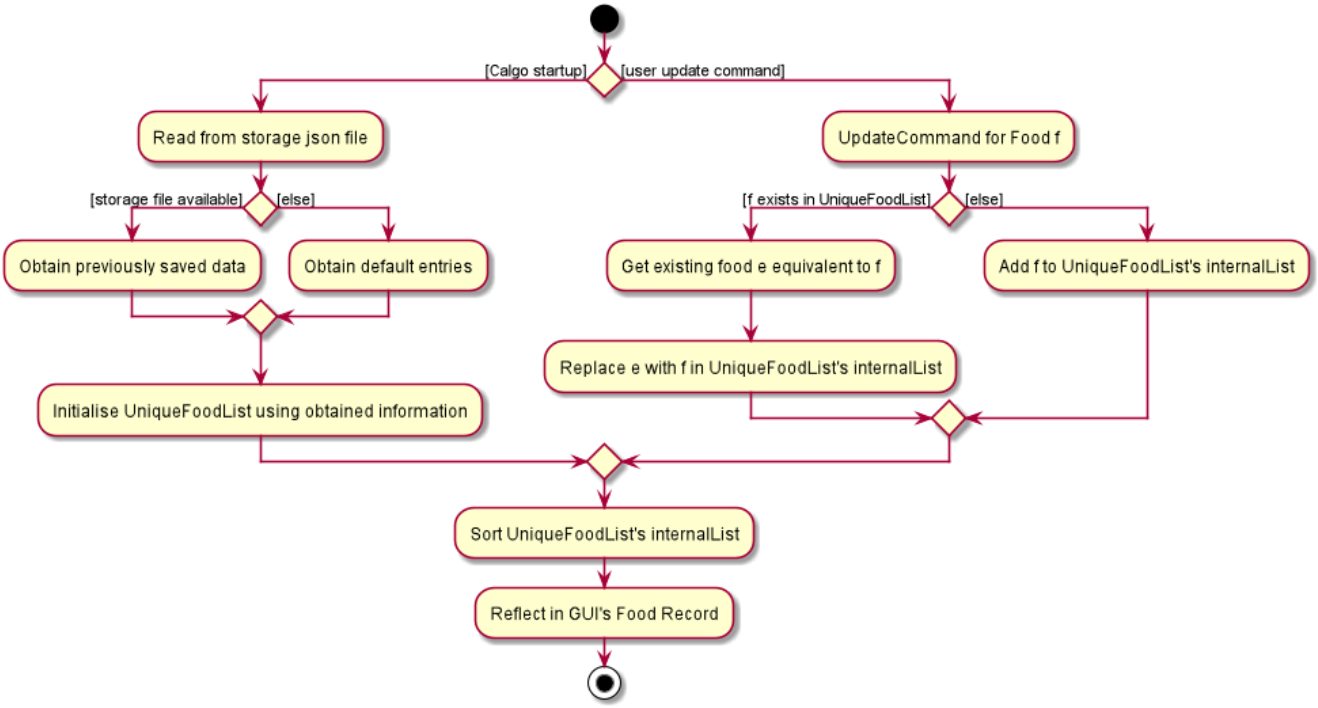


Figure 13. Activity Diagram for Lexicographical Ordering

4.6. 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 4.1, “Configuration” below)
- 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 App
- `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

4.7. Updating `FoodRecord`

This feature allows you to add a food preset with all its nutritional details into the `FoodRecord`. This

makes it convenient for you to keep track of your `Food` consumed in the day without having to manually key in the nutritional details every time you do so.

4.7.1. Implementation

The update mechanism is facilitated by `FoodRecord` and `UpdateCommand`. An additional operation was implemented into `FoodRecord`:

- `FoodRecord#hasExistingFood()` - Checks if there is an existing `Food` in `FoodRecord` based on its name only

This operation was exposed in the `Model` interface as `Model#hasExistingFood()`.

The update feature first checks if there is already an existing `Food` item with the same name inside `FoodRecord` using the method `FoodRecord#hasExistingFood()`.

If there is already an existing `Food` with the same name, the existing `Food` item will override the `Food` item inside `FoodRecord` with the new nutritional information provided by the user.
Otherwise, the new `Food` item will be added into the `FoodRecord`.

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

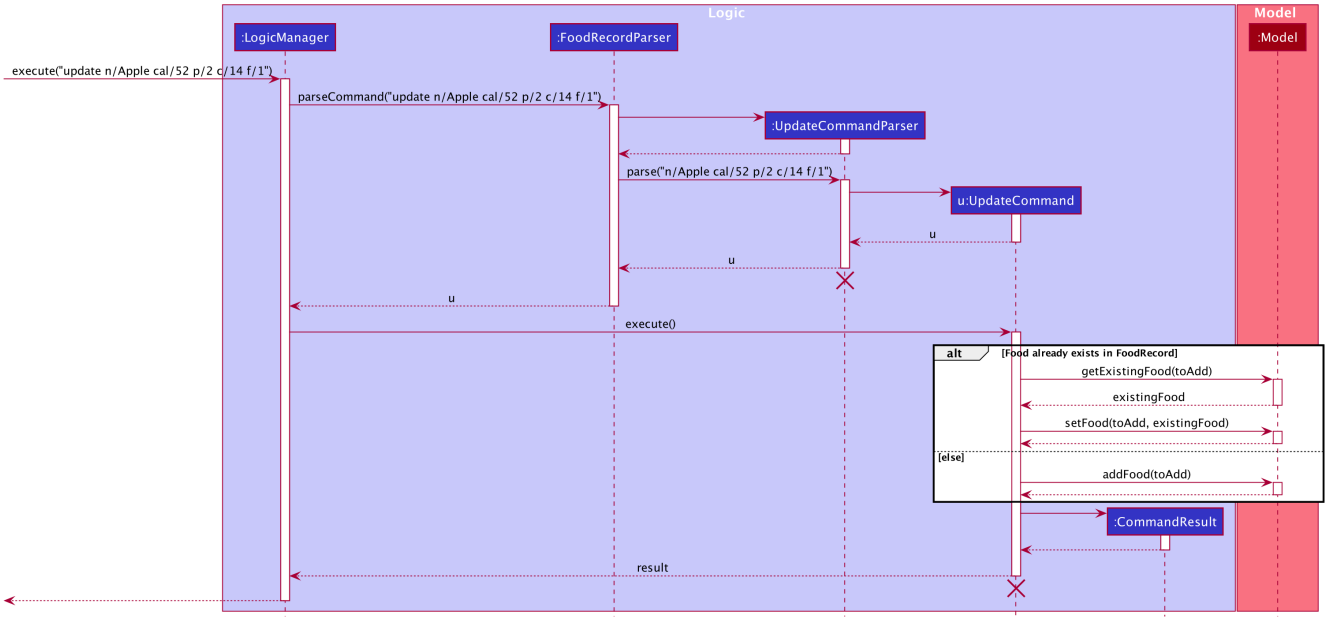


Figure 14. Sequence Diagram for Update command

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

4.7.2. 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 a separate command of `edit` to deal with existing `Food` item apart from `add` to add new `Food` item into the `FoodRecord`. Instead, a smarter command of `update` is used to deal with both scenarios.
  - Cons: This 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.
- **Alternative 2:** Informs the user that there is already an existing `Food` item, and direct him to use another function `edit` to edit the existing `Food` instead.
  - Pros: In the event where the user is unaware that there is already an existing `Food` item, this two step process will be clearer to him that he is in fact editing a `Food` item and not adding a new one in.
  - Cons: This is more tedious for the user since more steps is required to change an existing `Food` item. On top of that, an additional command of `edit` will be required and `update` should be replaced with `add` for clearer user experience.

4.7.3. Summary

The `update` command is a smart command that either updates an existing `Food` item 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:

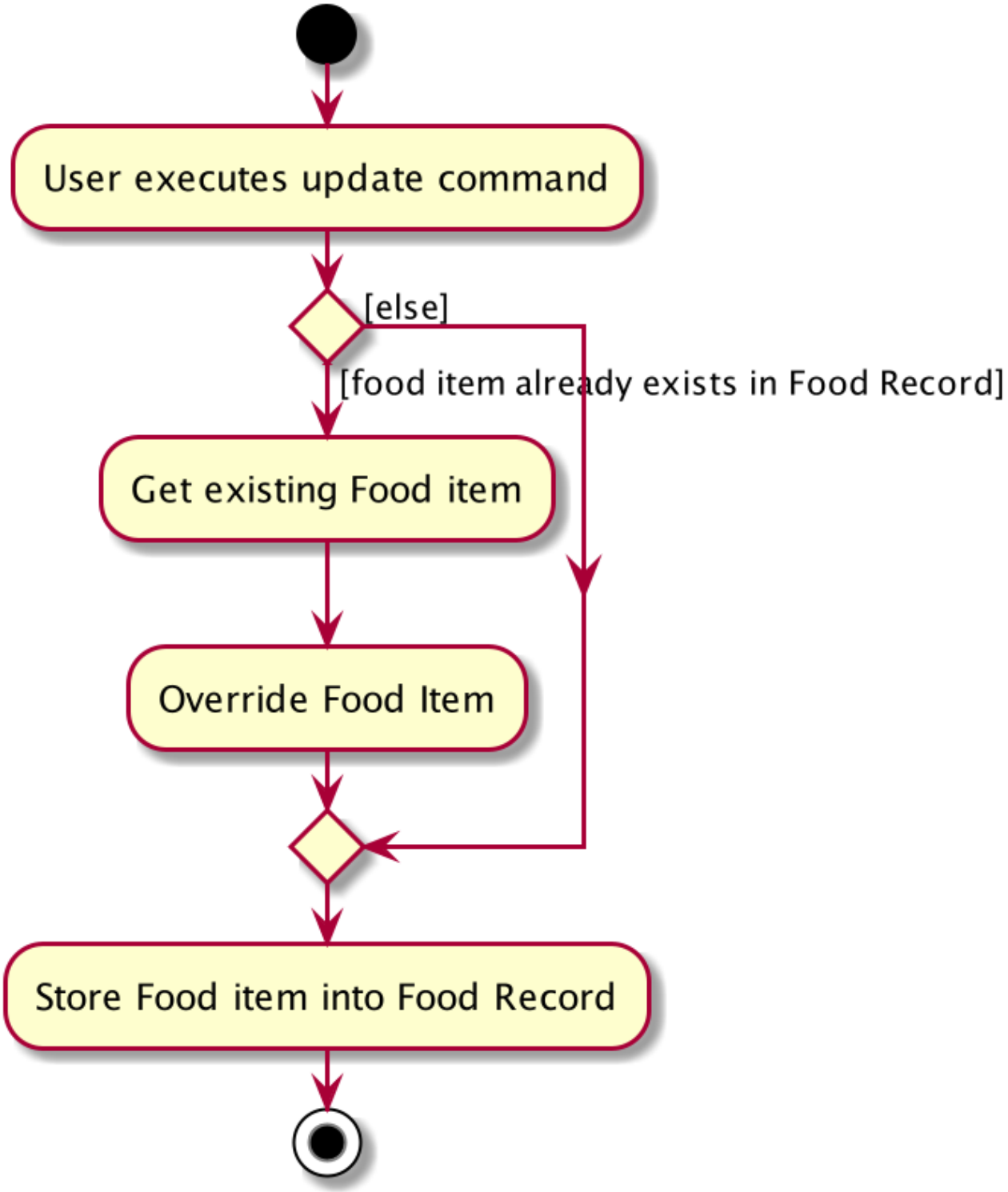


Figure 15. Activity Diagram for Update command

5. Documentation

Refer to the guide here.

6. Testing

Refer to the guide here.

7. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile:

- wants to have, or already has, a lifestyle of eating healthy
- manages a significant number of Food items (finding information about each Food item, tracking consumption, etc)
- prefers desktop Apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI Apps

Value proposition:

- **Insights:** set goals, generate consumption reports and view progress and statistics
- **Hassle-Free Convenience:** conveniently handles entry conflicts, tolerates incomplete search inputs and produces fast responses
- **Flexibility:** generate Food records as a portable file, tracking wherever, whenever, without a device
- **Efficiency:** manage caloric tracking faster than a typical mouse/GUI driven App

Appendix B: User Stories

Priorities: High (must have) - * * *, Medium (nice to have) - * *, Low (possible future development) - *

Priority	As a …	I want to …	So that I can…
* * *	user who does not know what my food is made of	find out the nutritional composition of a particular food by name	locate details of the Food item without having to go through the entire Food record
* * *	new user	see usage instructions	refer to instructions when I forget how to use Calgo

Priority	As a …	I want to …	So that I can…
* * *	user	have a portable and readable file to store the relevant values for each Food item	backup, share or export my personal Food records list
* * *	user who may not be able to access his laptop at some time	have a copy of my past Food records	use it for physical reference
* * *	user who wants to save a copy of my current Food records	save my file at a convenient location	easily access it
* * *	user who dislikes sieving through information and prefers to have only the relevant information presented	have a way to easily find what Food items I want in the records	save time and effort and not get annoyed
* * *	lazy user who does not like typing too many tedious characters	find entries using incomplete words or phrases	obtain the same intended results for a search through the Food records as in the case of typing fully and correctly
* * *	user who dislikes memorising things	have an option to see the entire Food record	know what Food items have their data currently in the records

Priority	As a …	I want to …	So that I can…
* * *	user who has many entries	view entire food record in lexicographic order	easily navigate to the entry in the record
* * *	user who is forgetful	be able to edit the nutritional value of a previously saved Food item in the Food record	I can edit the Food Item if I remembered a nutrition value of the Food Item wrongly previously
* * *	user who is busy	be able to create a list of Food records with preset nutritional values	so that I can quickly choose a Food Item with preset values and add it to my calorie tracker
* * *	user who doesn't like redundant things	see and be warned if a Food item that I am about to update my Food Record with already exists	so that I can save time and effort and not create a duplicate item in the Food record.
* * *	user who gets bored of food easily	delete a Food item that I no longer want to eat in future from my Food records	so that I do not have so many Food items in the Food records that I no longer eat.
* * *	user who is a foodie	find out the number of times I have eaten a specific food item each day	systematically cut down on overeaten food and monitor progress.

Priority	As a …	I want to …	So that I can…
* * *	user who cannot decide on what to eat	obtain a list of personalised food recommendations that still align with my dietary goals	do not waste time deciding what to eat nor will I give in to impulse and eat junk food.
* * *	user who is interested to lose weight	find out the number of calories I have consumed each day	can check which days I have exceed my desired number of daily calorie and exercise more to compensate.
* * *	user who is busy	obtain an easy-to-understand consumption report	quickly understand my food consumption patterns and make plans to rectify them accordingly.
* * *	user who remembers the big picture but not the specifics	search for a particular part of a guide	not be bothered by unnecessary information.
* *	forgetful user	be able to lookup exact command formats	so that I won't need to go through the trouble of memorising commands
* * *	user who values visuals	curated information expressed in a well organised graph	intuitively understand information

Priority	As a …	I want to …	So that I can…
* * *	user who values opinions	have some suggestions based on my goals and consumption patterns	know my options when I am indecisive on what to eat
* *	user who cannot fully remember the Food name	access a Food item's information by any one of its nutritional values I happen to remember	obtain a list of possible Food items that are relevant
* *	fitness influencer	get a screenshot and share my daily food consumption	can conveniently continue to inspire my followers.
*	user who cannot fully remember the food name	have some form of autocomplete or input correction measure for incomplete keywords	obtain the possible results for a search through the Food records as in the case of typing fully and correctly

{More to be added as development proceeds and is always ongoing}

Appendix C: Use Cases

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

Use case: obtain reference for app's commands

MSS

1. User requests for a guide on the app's commands
2. Calgo shows a list of all available commands and their corresponding purpose and usage.

   Use case ends.

Use case: `find` `Food` item by keyword (which can be an incomplete word)

MSS

1. User requests to find a `Food` item by the keyword.
2. `Calgo` shows a list of `Food` items which contains `name` in any part of the name of the `Food` item.

   Use case ends.

Extensions

The `FoodRecord` is empty
A message indicating that zero matching `Food` items exist is shown.
Use case ends.

Use case: `find` `Food` item by nutritional value

MSS

1. User requests to `find` a `Food` item by a single nutritional value of `Protein`, `Carbohydrate`, or `Fat` (indicated by the prefix).
2. `Calgo` shows a list of `Food` items in the `FoodRecord` which has the same nutritional values.

   Use case ends.

Extensions

The `FoodRecord` is empty.
`Calgo` shows a message indicating that 0 matching `Food` items exist.
Use case ends.

Use case: `export` current `FoodRecord`

MSS

1. User requests to `export` the current `FoodRecord`.
2. Calgo creates a user-friendly text file `FoodRecord.txt` containing all `Food` item details in the `data/exports` folder.

   Use case ends.

Use case: `list` all current `Food` entries

MSS

1. User requests to `list` all current `FoodRecord` entries.
2. `Calgo` shows a list of all `Food` items in the current `FoodRecord`.

   Use case ends.

Extensions

The `FoodRecord` is empty.
`Calgo` shows a message indicating that the `FoodRecord` is currently empty.
Use case ends.

Use case: `update` current `FoodRecord` with a new `Food` item

MSS

1. User requests to add a new `Food` item in the `FoodRecord`.
2. `Calgo` creates and saves a new `Food` item in the `FoodRecord` with nutritional information specified by user.

   Use case ends.

Use case: `update` an existing `Food` item in current `FoodRecord`

MSS

1. User requests to edit an existing `Food` item in the `FoodRecord`.
2. `Calgo` replaces the existing `Food` item's nutritional values with the new information.

   Use case ends.

Use case: `delete` an existing `Food` item in current `FoodRecord`

MSS

1. User requests to delete an existing `Food` item from the `FoodRecord`
2. `Calgo` deletes the existing `Food` item in the `FoodRecord`.

   Use case ends.

Use case: set a dietary `goal`

MSS

1. User uses `goal` command to set a dietary `DailyGoal` for the daily number of `Calorie` s to be consumed.
2. `Calgo` stores this `DailyGoal` in user preferences and analyses `Food` consumption with respect to this `goal`.

   Use case ends.

Use case: generate a `report` on a specific day.

MSS

1. User requests to obtain a `report` on his or her `Food` consumption patterns on a particular day.
2. Calgo analyses the `Food` consumed on that day and generates a document with actionable insights for the user.

   Use case ends.

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be able to hold up to 1000 `Food` items without a noticeable sluggishness in performance for typical usage.
3. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.
4. Calgo should work on both 32-bit and 64-bit environments.
5. The product expects users to initially find out about `Food` items and their respective nutritional values for creating `Food` item entries for the first time.

Appendix E: Glossary

Food

`Food` items entered by the user to represent a real life Food. This contains nutritional values of each of their `Calorie` s, number of grams of `Protein` s, `Carbohydrate` s and `Fat` s. They can also contains a series of `Tag` s.

Food Records

The accumulated list of all `Food` items entered by the user, containing nutritional values of each of their `Calorie` s, number of grams of `Protein` s, `Carbohydrate` s and `Fat` s.

Mainstream OS

Windows, Linux, Unix, OS-X

Food records

The accumulated list of all Food items entered by the user, containing nutritional values of calorie, number of grams of protein, carbohydrates and fats.

Appendix F: Product Survey

Product Name

Author: …

Pros:

- …
- …

Cons:

- …
- …

Appendix G: Instructions for Manual Testing

Given below are instructions to test the App manually.

NOTE	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: Shows the GUI with a set of sample contacts. The window size may not be optimum.
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.

G.2. Deleting a `Food`

1. Deleting a `Food` item from the `FoodRecord`

a. Prerequisites: Launch `Calgo` succesfully and a `Food` item Apple already exists in `FoodRecord`

b. Test case: `delete n\Apple`
   Expected: `Food` item Apple is deleted from `FoodRecord`. Details of the deleted `Food` shown in the status message.

c. Test case: `delete 0`
   Expected: No food is deleted. Error details shown in the status message. Status bar remains the same.

d. Other incorrect delete commands to try: `delete`, `delete n/Banana` (where `Food` banana does not exists in `FoodRecord`)
   Expected: Similar to previous.

G.3. Listing all `Food` entries

1. Listing down all entries, regardless of previous commands

   a. Prerequisites: Launch `Calgo` successfully.

   b. Test case: `list`
      Expected: The GUI will show all `Food` entries existing in the `FoodRecord`.

G.4. Saving data

1. Dealing with missing/corrupted data files

   a. *{explain how to simulate a missing/corrupted file and the expected behavior}*

{ more test cases … }