tp
FridgeFriend is an app for managing food in the fridge that is optimised for use via a Command Line Interface (CLI) and is targeted at new homeowners who cook. If you can type fast, `FridgeFriend` can track your cold or frozen groceries faster and easier than any other apps. It is written in Java, and has more than 6000 lines of code.
View the Project on GitHub at https://github.com/

Developer Guide

Introduction

`FridgeFriend` is an app for managing food in the fridge that is optimised for use via a Command Line Interface (CLI) and is targeted at new homeowners who cook. If you can type fast, `FridgeFriend` can track your cold or frozen groceries faster and easier than any other apps. It is written in Java, and has more than 6000 lines of code.

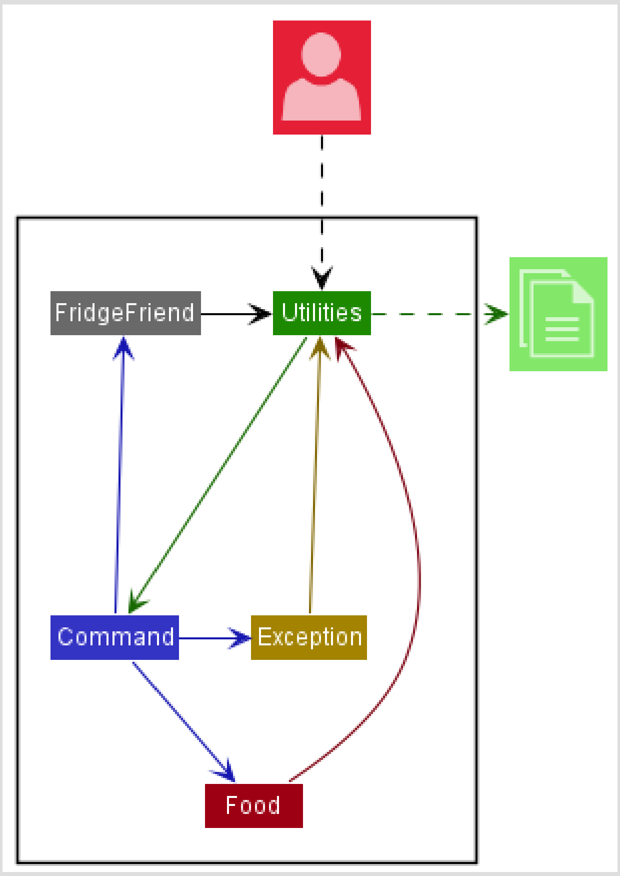
Contents

Design

This section focuses on the general architecture of the application as well as the relationships between the classes involved. The overall architecture would be covered first, before the four individual components.

Architecture

The ***Architecture Diagram*** given below explains the high-level design of the App.



The user interacts with the `Ui` class which is part of the `Utilities` component.

The Main driver class for the FridgeFriend app is named `FridgeFriend`. It is responsible for,

- At app launch: Initialising the components in the correct sequence, and connecting them up with each other.
- At shut down: Shutting down the components and invoking cleanup methods where necessary.

The rest of the App consists of four components.

- `Utilities`: The main component containing the key driver classes in FridgeFriend, such as `Parser`, `UI`, and `Storage`.

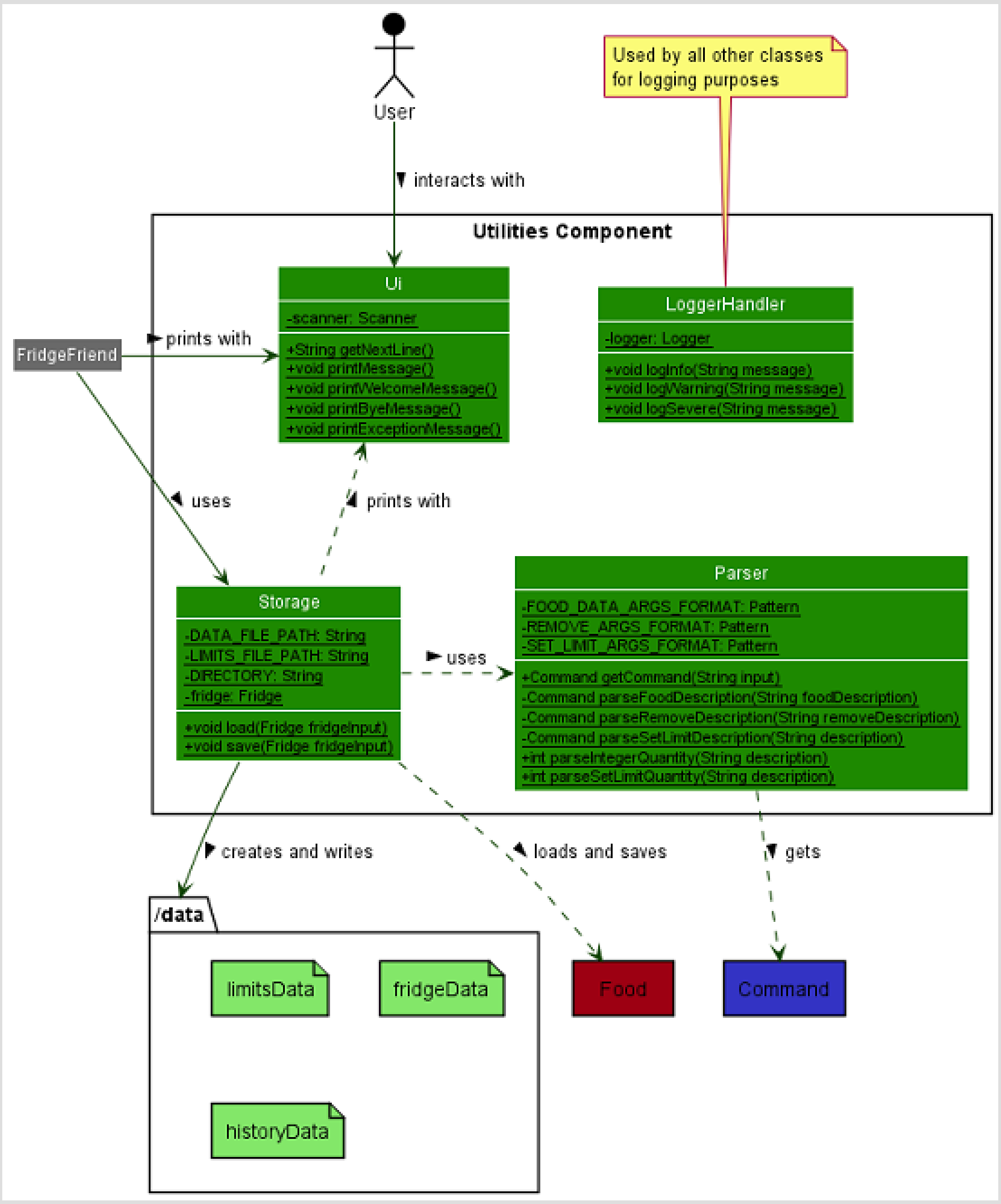
- `Command`: Executes commands based on the input obtained and processed from `Parser` in `Utilities`. The list of executable commands can be found in our User Guide.

- `Food` contains a collection of classes used by the FridgeFriend application to represent food items. Food objects are instantiated by the `Command` Component. Once a `Food` object is created, it will be stored to disk using the `Storage` function in `Utilities`.

- `Exception` contains a collection of classes that represent potential exception events that may occur during the usage of `FridgeFriend`. The `Exception` component facilitates the return of exceptions to the `UI` class in `Utilities`, which will display the error message to the user.

Utilities Component

The Utilities component contains the main classes that run the main functions of FridgeFriend.


User
interacts with
Used by all other classes for logging purposes
Utilities Component
FridgeFriend
prints with
Ui
-scanner: Scanner
+String getNextLine()
+void printMessage()
+void printWelcomeMessage()
+void printByeMessage()
+void printExceptionMessage()
LoggerHandler
-logger: Logger
+void logInfo(String message)
+void logWarning(String message)
+void logSevere(String message)
uses
prints with
Storage
-DATA_FILE_PATH: String
-LIMITS_FILE_PATH: String
-DIRECTORY: String
-fridge: Fridge
+void load(Fridge fridgeInput)
+void save(Fridge fridgeInput)
uses
Parser
-FOOD_DATA_ARGS_FORMAT: Pattern
-REMOVE_ARGS_FORMAT: Pattern
-SET_LIMIT_ARGS_FORMAT: Pattern
+Command getCommand(String input)
-Command parseFoodDescription(String foodDescription)
-Command parseRemoveDescription(String removeDescription)
-Command parseSetLimitDescription(String description)
+int parseIntegerQuantity(String description)
+int parseSetLimitQuantity(String description)
creates and writes
loads and saves
gets
/data
limitsData
fridgeData
historyData
Food
Command

The ***Utilities Class Diagram*** given above shows how the classes in the Utilities component interact with each other and classes from other component.

The Utilities Component consists of 4 classes.

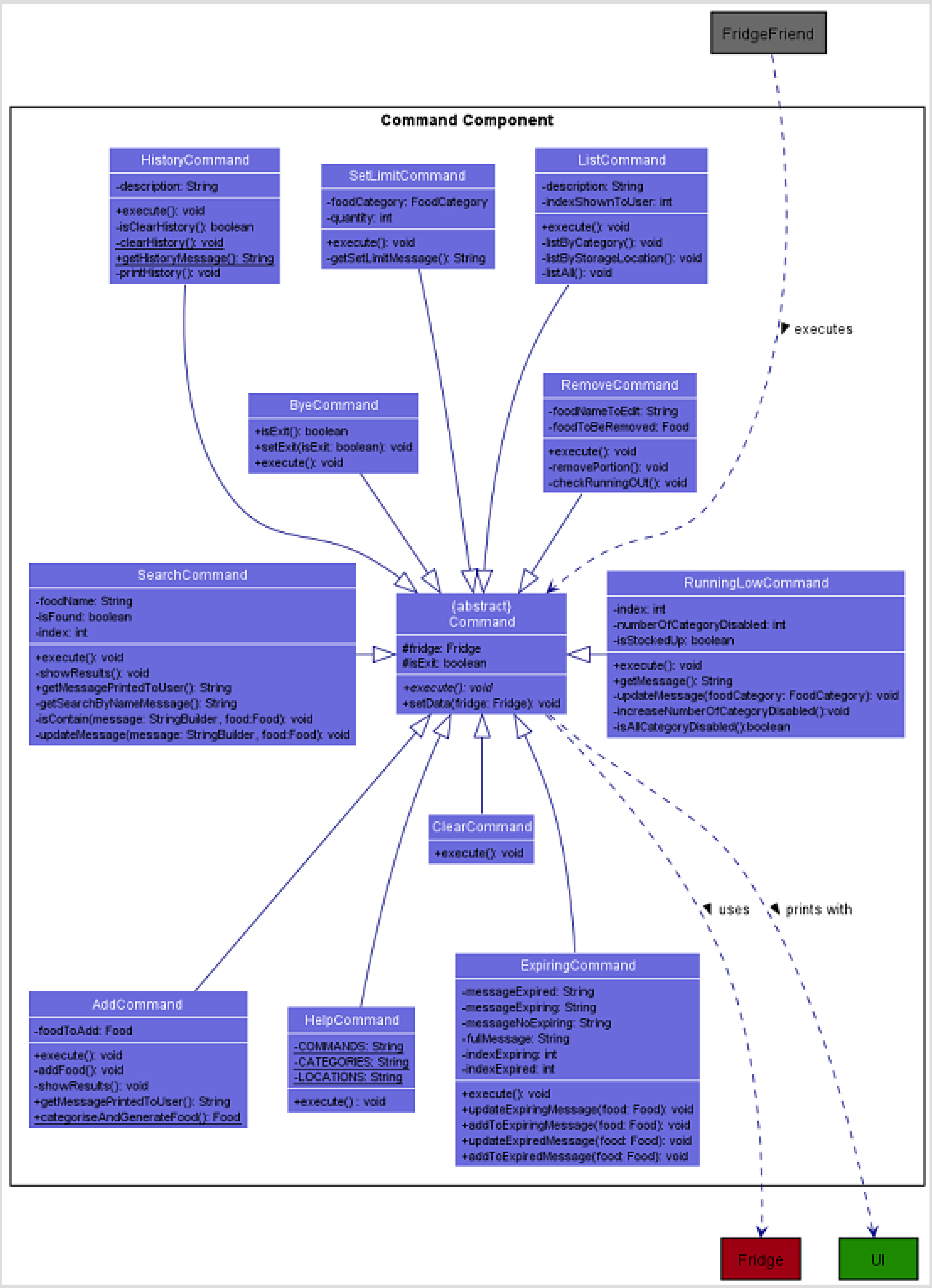
- `LoggingHandler`: Logs information during execution to the console.
- `Parser`: Breaks down user input into relevant objects.
- `Storage`: Reads data from, and writes data to, the local disk.

- `Ui`: Handles the input and output of the application.

Command Component

The Command component contains the sub classes of the features that will be executed.

The command object is executed by the main method in `FridgeFriend`. The execution of the command can affect the fridge. After the execution, the results of the command object is passed to the `Ui` which displays the message to the user.



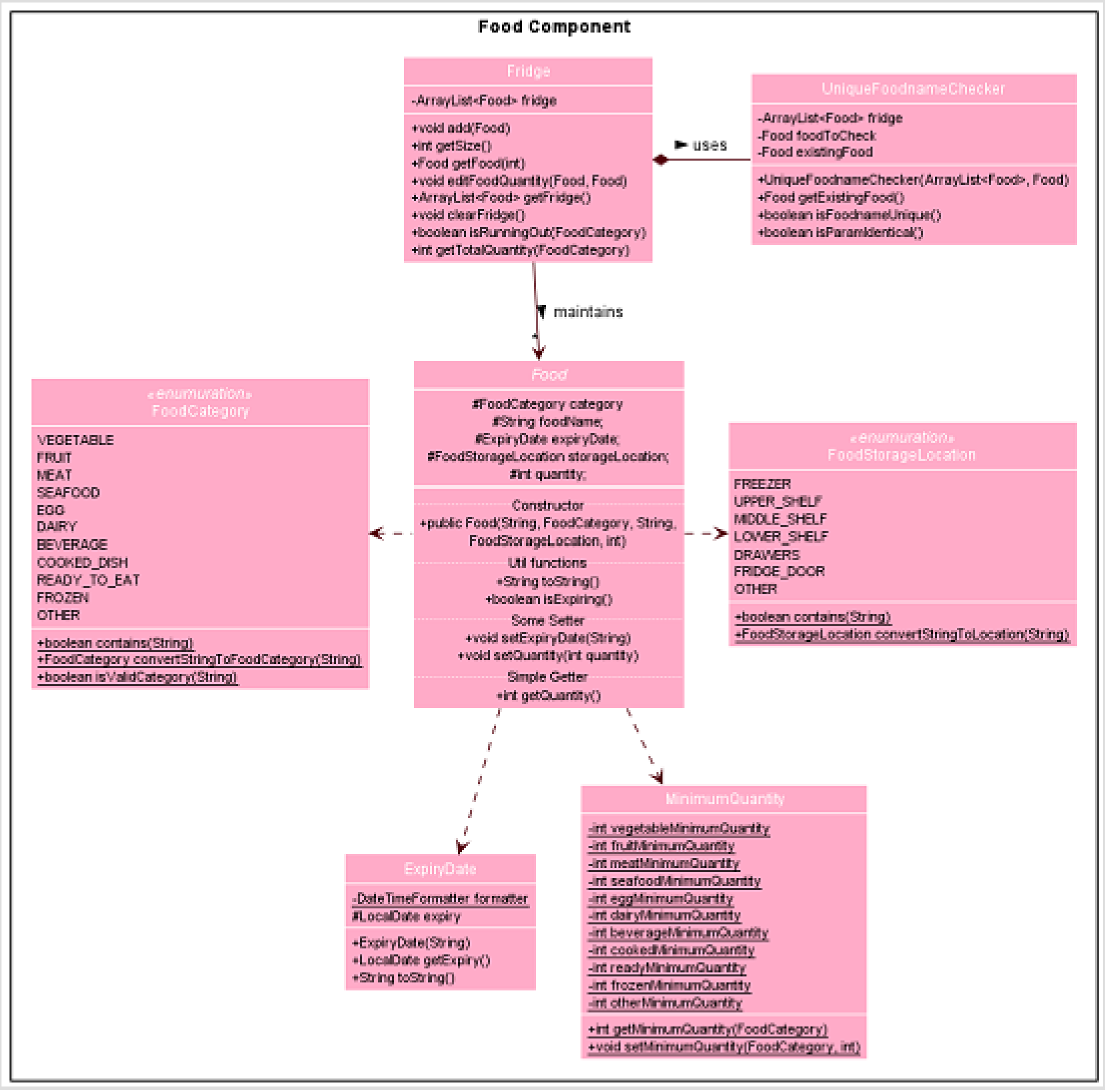
The ***Command Class Diagram*** given above shows how the Command interacts with `Fridge` class.

The Command Component consists of 11 subclasses where each class represents a command feature.

- `AddCommand`: Add a new food object to the fridge or add the quantity of existing food item with same parameters when executed.
- `RemoveCommand`: Remove a portion of food from a particular food in the fridge when executed.
- `ListCommand`: List details of food either by a category, storage location or all off it when executed.
- `SearchCommand`: Search for the food that contains a particular food name.
- `ExpiringCommand`: Provide the list of items that are expiring in a week.
- `ClearCommand`: Clear the list of food items in the fridge object.
- `HelpCommand`: List all the commands in `FridgeFriend`.
- `RunningLowCommand`: Provide the food categories that are running low compared to the limit set for that category.
- `SetLimitCommand`: Change the default quantity limit in that particular Food Category.
- `HistoryCommand`: Display the history of all food item additions in the fridge.
- `ByeCommand`: Exit the application.

Food Component

The Food component represents the basic `Food` object and related objects in FridgeFriend.



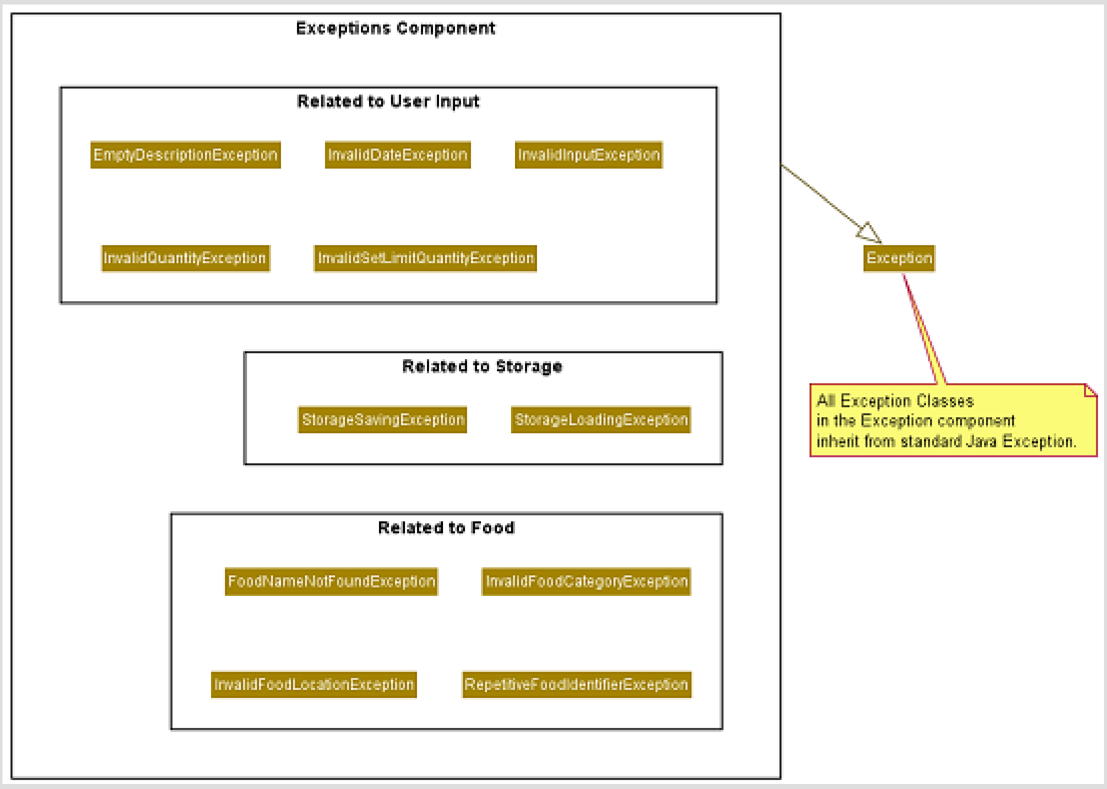
Food Component
Fridge
-ArrayList<Food> fridge
+void add(Food)
+int getSize()
+Food getFood(int)
+void editFoodQuantity(Food, Food)
+ArrayList<Food> getFridge()
+void clearFridge()
+boolean isRunningOut(FoodCategory)
+int getTotalQuantity(FoodCategory)
uses
UniqueFoodnameChecker
-ArrayList<Food> fridge
-Food foodToCheck
-Food existingFood
+UniqueFoodnameChecker(ArrayList<Food>, Food)
+Food getExistingFood()
+boolean isFoodnameUnique()
+boolean isParamIdentical()
maintains
«enumeration»
FoodCategory
VEGETABLE
FRUIT
MEAT
SEAFOOD
EGG
DAIRY
BEVERAGE
COOKED_DISH
READY_TO_EAT
FROZEN
OTHER
+boolean contains(String)
+FoodCategory convertStringToFoodCategory(String)
+boolean isValidCategory(String)
Food
#FoodCategory category
#String foodName;
#ExpiryDate expiryDate;
#FoodStorageLocation storageLocation;
#int quantity;
Constructor
+public Food(String, FoodCategory, String, FoodStorageLocation, int)
Util functions
+String toString()
+boolean isExpiring()
Some Setter
+void setExpiryDate(String)
+void setQuantity(int quantity)
Simple Getter
+int getQuantity()
«enumeration»
FoodStorageLocation
FREEZER
UPPER_SHELF
MIDDLE_SHELF
LOWER_SHELF
DRAWERS
FRIDGE_DOOR
OTHER
+boolean contains(String)
+FoodStorageLocation convertStringToLocation(String)
ExpiryDate
-DateTimeFormatter formatter
#LocalDate expiry
+ExpiryDate(String)
+LocalDate getExpiry()
+String toString()
MinimumQuantity
-int vegetableMinimumQuantity
-int fruitMinimumQuantity
-int meatMinimumQuantity
-int seafoodMinimumQuantity
-int eggMinimumQuantity
-int dairyMinimumQuantity
-int beverageMinimumQuantity
-int cookedMinimumQuantity
-int readyMinimumQuantity
-int frozenMinimumQuantity
-int otherMinimumQuantity
+int getMinimumQuantity(FoodCategory)
+void setMinimumQuantity(FoodCategory, int)

The ***Food Class Diagram*** given above shows all the classes derived from Food or used in Food operations.

- `Food` is an abstract class which has multiple child Food classes, distinguished by category. This is to facilitate different manipulations on different food categories.
- `Fridge` maintains the list of `Food` and has methods to add, remove or modify the foods.
- `Fridge` uses `UniqueFoodnameChecker` to perform add tasks under different user input scenarios.
- Whenever a `Food` object is created or called, exceptions may be thrown.

Exception Component

The Exception component represents a collection of classes that represent potential exception events that may occur during the usage of `FridgeFriend`. The `Exception` component facilitates the return of exceptions to the `UI` class in `Utilities`, which will display the corresponding error message to the user.


Exceptions Component
Related to User Input
EmptyDescriptionException
InvalidDateException
InvalidInputException
InvalidQuantityException
InvalidSetLimitQuantityException
Related to Storage
StorageSavingException
StorageLoadingException
Related to Food
FoodNameNotFoundException
InvalidFoodCategoryException
InvalidFoodLocationException
RepetitiveFoodIdentifierException
Exception
All Exception Classes in the Exception component inherit from standard Java Exception.

The ***Exception Class Diagram*** given above shows the custom `Exceptions` used by `FridgeFriend`, and stored inside the `Exceptions` component. All exceptions extend from the Java default `Exception` class. They only differ with regard to the throwable error message. The `Exceptions` component currently consists of eleven (11) custom Exceptions:

Among these eleven custom exceptions, the `Exceptions` component can generally be classified into three broad categories:

- Invalid User Input
  - Thrown when a raw user input cannot be parsed into a valid value
  - Includes: `EmptyDescriptionException`, `InvalidDateException`, `InvalidInputException`, `InvalidQuantityException`, `InvalidSetLimitQuantityException`
- Food Errors
  - Thrown when a parameter for a food item is invalid

    - Includes: `FoodNameNotFoundException`, `InvalidFoodCategoryException`, `RepetitiveFoodIdentifierException`, `InvalidFoodLocationException`

- File Storage Errors
    - Thrown when there are errors occur during the file loading and saving processes
    - Includes: `StorageLoadingException`, `StorageSavingException`

Implementation

This section covers the application flow and more specific interactions between classes and their methods. A overview of the main application logic would be covered first, before addressing several specific application commands.

ℹ️ For clarity within diagrams, where there are exceptions being thrown, they will be represented in the diagrams as comments. In the actual implementation, exception objects are being initialised and executed.

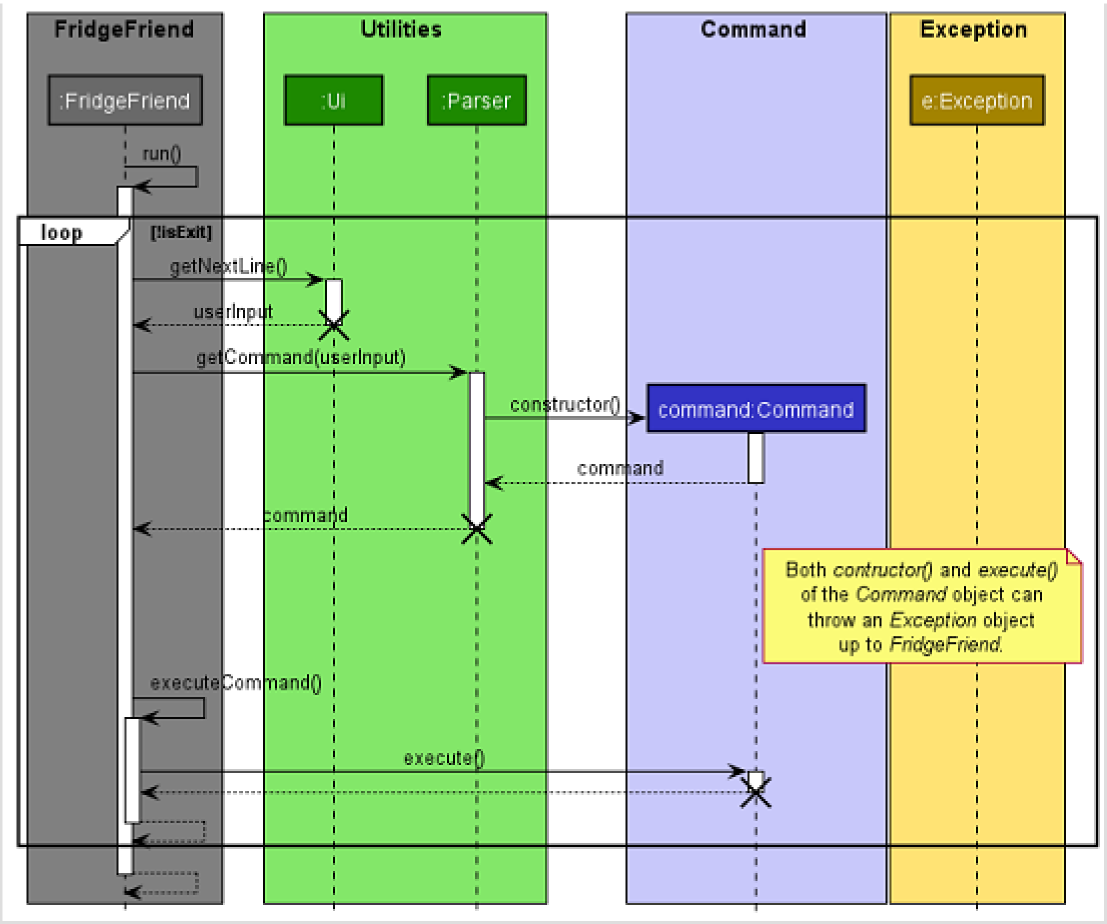
ℹ️ Due to simplicity, some function calls that are not essential to the execution of commmands are not covered in the diagram. Hence, the `Command` objects will not be destroyed at the end of the diagrams.

Main Logic

The main application logic underlines the main executable conditions that make up the bulk of the application. Below is the outline of the logic:

1. `FridgeFriend` uses the `Ui` class to obtain the user input.
2. `FridgeFriend` then uses the `Parser` class to parse the user input.
3. This results in the construction of a `Command` object that is returned to `FridgeFriend`.
4. Following that, `FridgeFriend` then self-invokes a method that executes the `Command` object.
5. In this method, the `Command` object executes its specified action.
6. From steps 2 to 5, should there be an error thrown, the `Command` class would construct an `Exception` object.
7. The `Exception` would be thrown back to `FridgeFriend`, which will then use the `Ui` class to display an error message to the user.

Given below is the sequence diagram for the interactions within the main application logic.



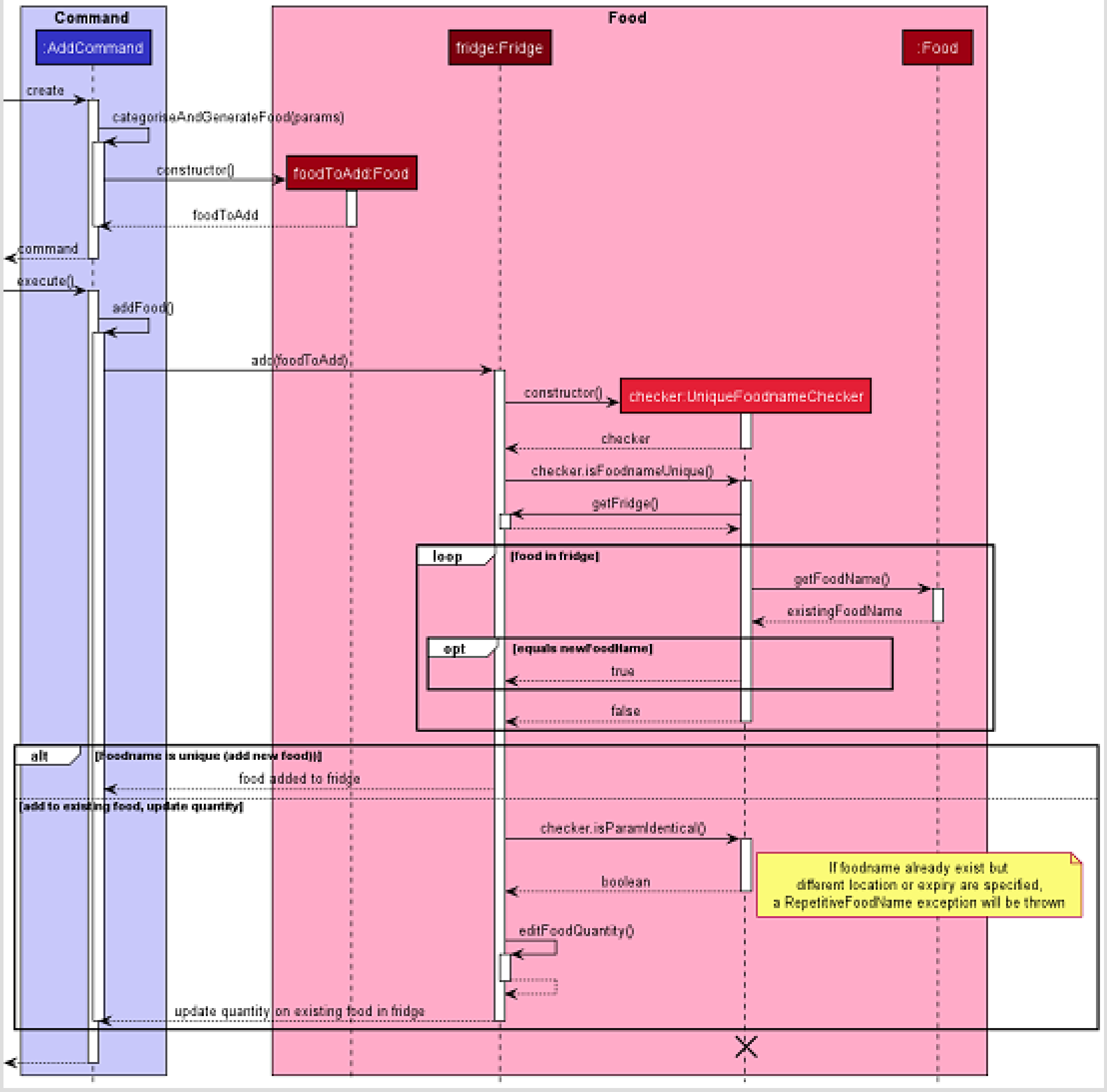
The lifelines for `Ui`, `Parser`, and `Command`, should end at the destroy marker. However, due to a limitation of PlantUML, the three lifelines reach the end of the diagram.

Add Command

AddCommand class is initialised whenever `Parser` recognises the `add` keyword.

1. The constructor should create a new `Food` object according to the parsed user input.
2. The `UniqueFoodnameChecker` will be called from `Fridge` class to decide whether to add a new `Food` or edit on an existing `Food` object.

Given below is the sequence diagram for the AddCommand workflow.


Command
:AddCommand
Food
fridge:Fridge
:Food
create
categoriseAndGenerateFood(params)
constructor()
foodToAdd:Food
foodToAdd
command
execute()
addFood()
add(foodToAdd)
constructor()
checker:UniqueFoodnameChecker
checker
checker.isFoodnameUnique()
getFridge()
loop
[food in fridge]
getFoodName()
existingFoodName
opt
[equals newFoodName]
true
false
alt
[Foodname is unique (add new food)]
food added to fridge
[add to existing food, update quantity]
checker.isParamIdentical()
boolean
If foodname already exist but different location or expiry are specified, a RepetitiveFoodName exception will be thrown
editFoodQuantity()
update quantity on existing food in fridge

Information:

- The `UniqueFoodnameChecker` object will be destroyed after use, but the `Food` object and `Fridge` object will still exist after the command finishes.
- The lifeline for `UniqueFoodnameChecker` should end at the destroy marker. However, due to a limitation of PlantUML, the lifelines reach the end of the diagram.

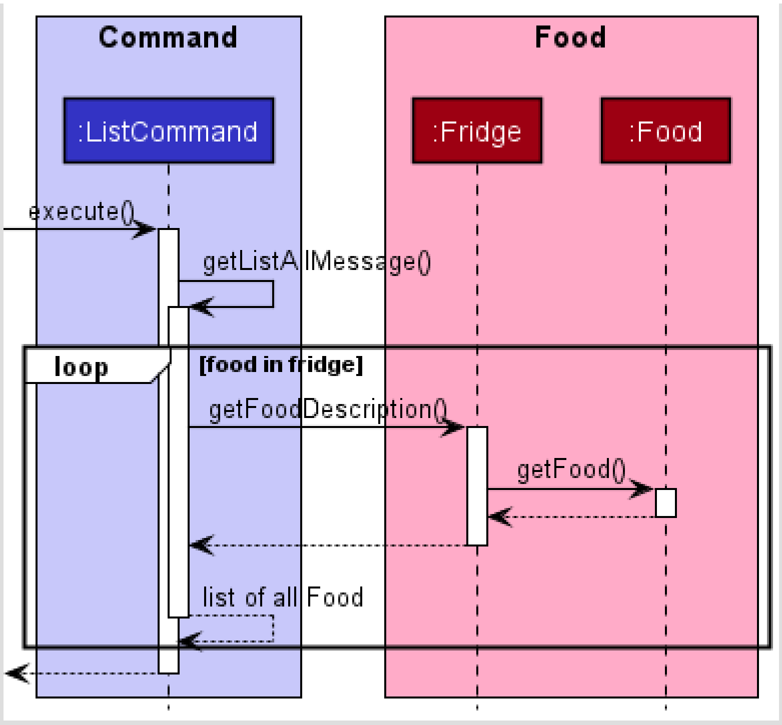
List Command

There are three variations of the `list` command.

1. List all food
2. List by category
3. List by storage location

The first variation is implemented by iterating through the collection of food items in the fridge. The names of all the food are concatenated using a Java `StringBuilder`, and the final result, containing the list of all food items, is returned to the main program.

The sequence diagram below shows how this `list` operation works:



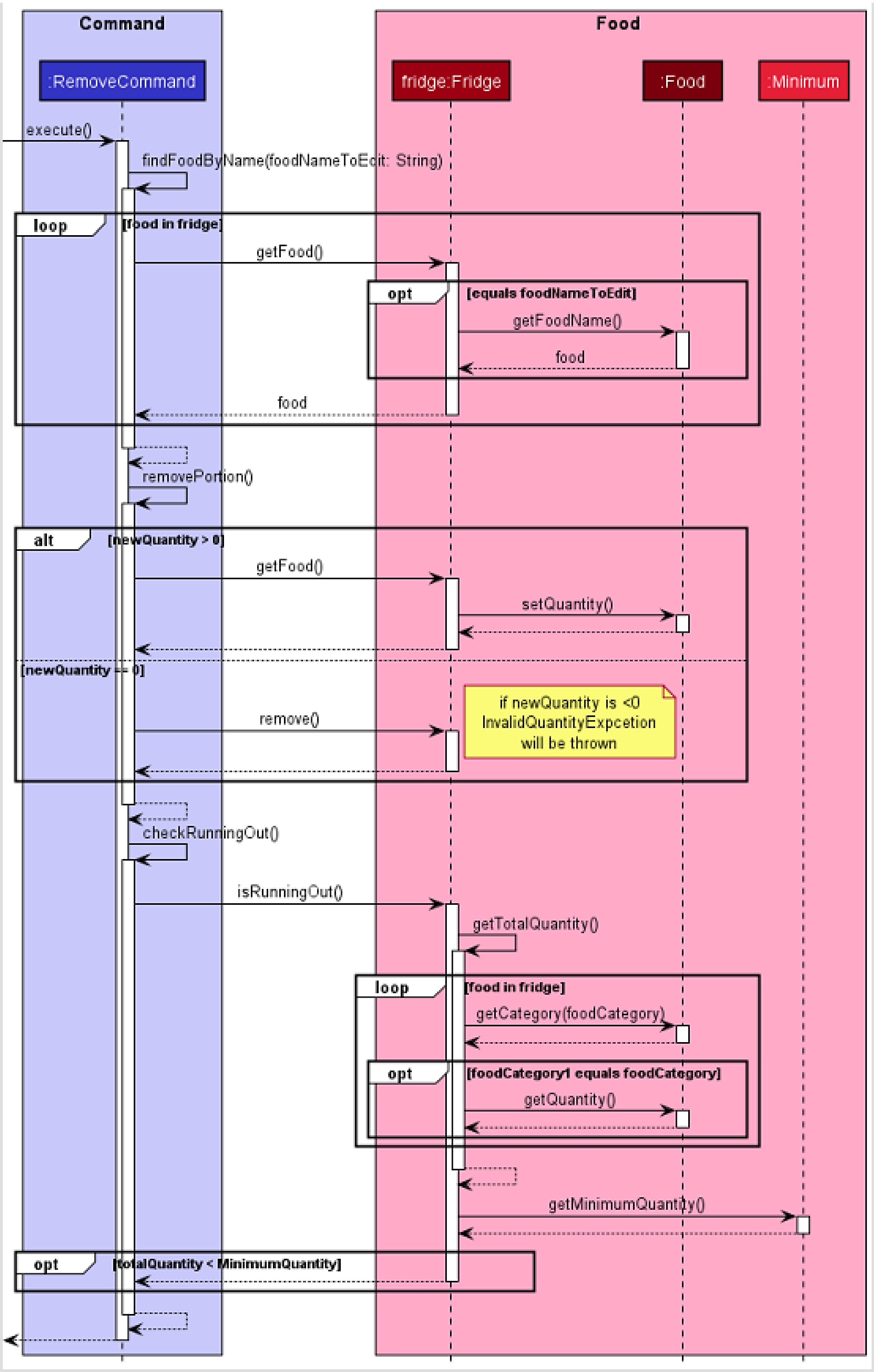
The second and third variations are also implemented by iterating through the collection of food in the fridge. However, there would be an additional check to verify if the given food item contains the correct category or storage location attribute as requested by the user, before being returned to the `ListCommand`. If it does not match the required attribute, then that food item would be excluded from the result. The final result would then be returned to the main program.

Remove Command

When the user requests to remove a portion of a food item in the fridge, the `remove` command will execute the remove operation by:

1. Searching for the food item in the fridge by looping through each food item.
2. If the food item exists, check if the quantity to be removed is greater than the available quantity in the fridge.
3. If it is greater, an exception will be thrown to `FridgeFriend`.
4. Otherwise, check if removing the quantity will reduce the quantity to zero.
5. If it does not reduce to zero, proceed to remove the required quantity.
6. If it does reduce to zero, then remove the food item completely.
7. Afterwards, check that by removing the food, the total quantity of that `FoodCategory` will not be lower than the minimum quantity.
8. Return a warning message if the total quantity is lower than the minimum quantity.

The sequence diagram below shows how the `remove` operation works:



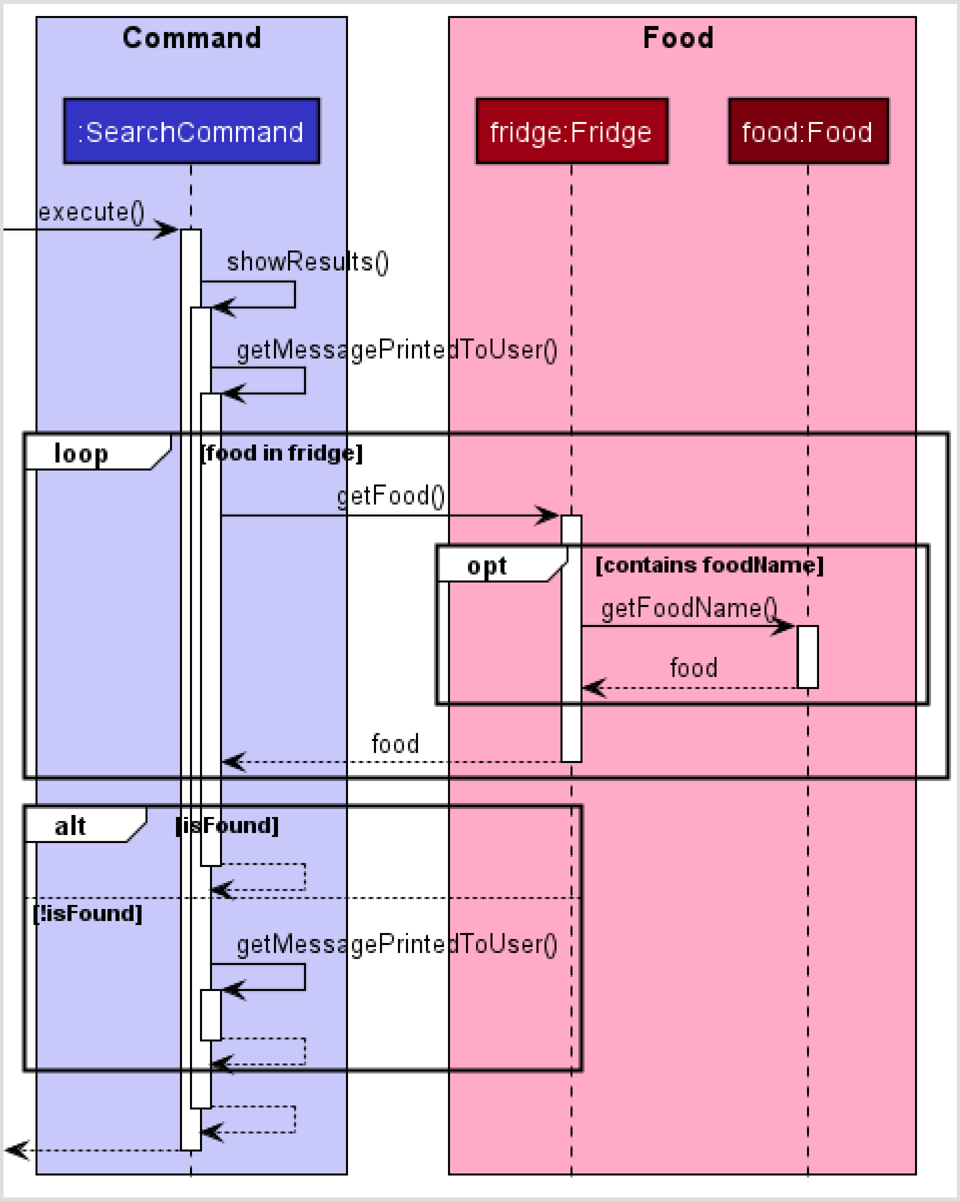
Search Command

When the user requests to search a food item in the fridge, the `search` command will execute search operation by:

1. Searching the list of foods in the fridge by that contain the food name entered by the user through a loop.

2. Afterwards, pass the success message or failure message to the UI.

The sequence diagram shows how the `search` operation works:

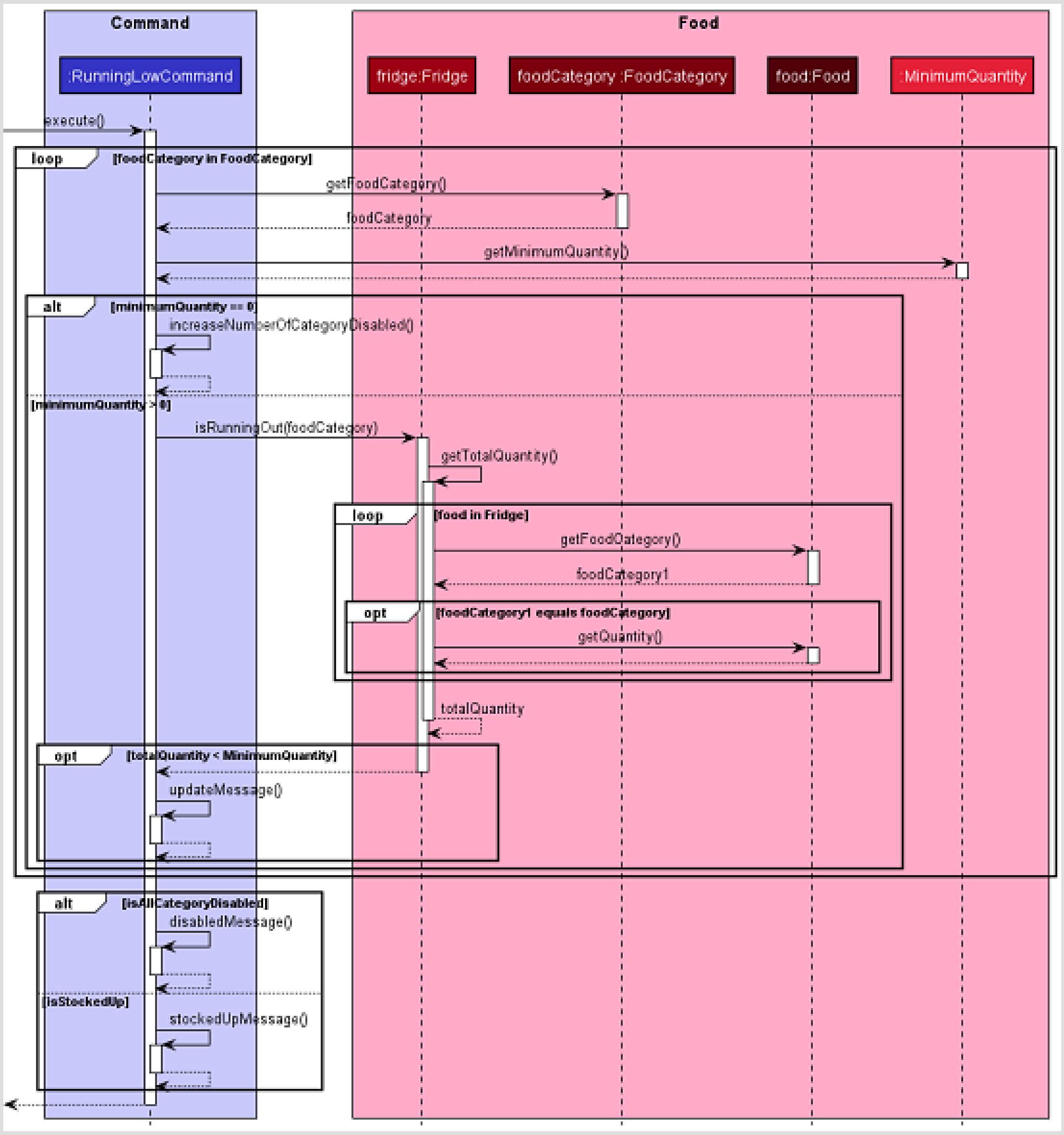


Running Low Command

The `runningLow` command is implemented by:

1. Checking the limit of each category in the `MinimumQuantity`.
2. If the limit is set to 0, the calculation of the total quantity of the category can be omitted.
3. If all the `FoodCategory` are set to 0, return the "disabled message".
4. Otherwise, if any of the `FoodCategory` limits are non-zero, proceed to check the total quantity of that `FoodCategory` in the fridge.
5. Follow up by comparing the total quantity with the default minimum number of quantity in each `FoodCategory`.
6. Then return all the `FoodCategory` that have quantities lower than the default limit in the warning message.
7. Otherwise, return a "stockedUp message".

The sequence diagram shows how the `runninglow` operation works:

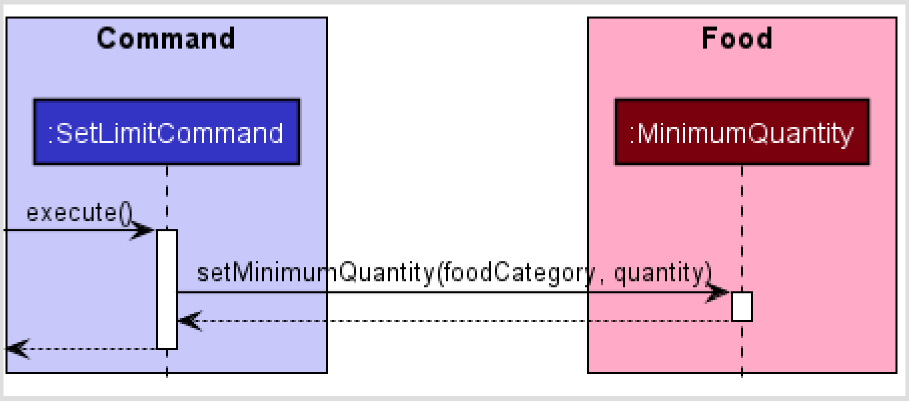


Set Limit Command

The `setlimit` command is implemented by:

1. Calling a setter method in `MinimumQuantity`.
2. The setter method will change the `FoodCategory`'s predefined minimum quantity.

The sequence diagram shows how the `setlimit` operation works:



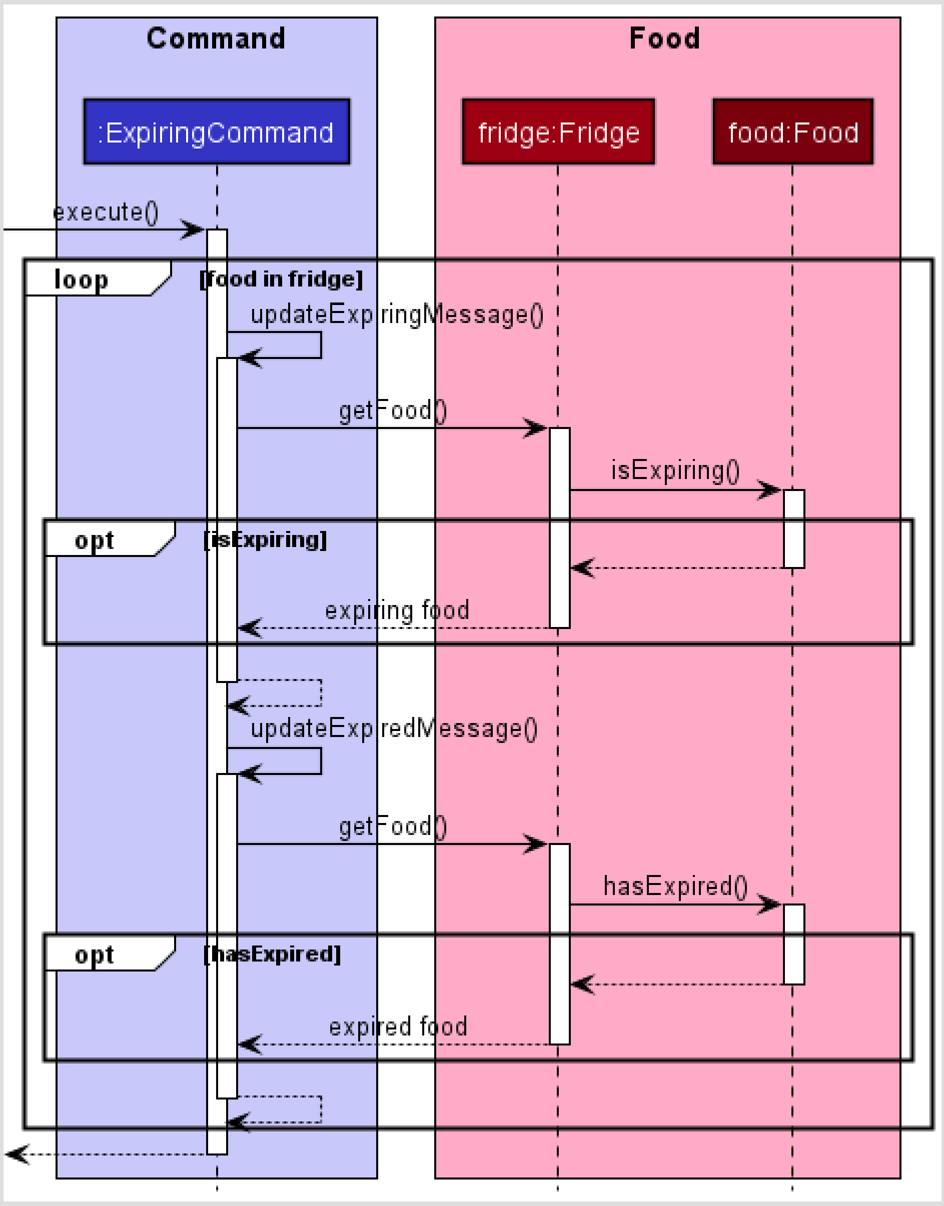
Expiring Command

The `expiring` command is implemented by looping of the food items in the fridge and returning the food items that are expired and expiring.

Information:

- A `Food` item is considered expiring if the expiry date is within one week of calling the command.
- A `Food` item is considered expired if the expiry date is earlier than the day of calling the command. `Food` items will not be considered expired if the command is called on the same day as its expiry.

The sequence diagram shows how the `expiring` operation works:



History Command

The code related to the `history` command is implemented by the `Storage` class in the `Utilities` component.

The implementation of the `history` command is as follows:

1. After every `add` command, a copy of the added food will be appended to `historyData.txt`
2. When the user invokes the `history` command, `historyData.txt` will be read, line by line.
3. The line-by-line contents of `historyData.txt` are concatenated into a single output string.
4. The output string that contains the history data will be displayed to the user.

Additionally, the command `history clear` deletes the contents of `historyData.txt` on the disk.

⚠️ Direct modification of the `historyData.txt` file may lead to corruption and loss of the data.

Product Scope

Target User Profile

- is a new homeowner who cooks
- has a need to manage a significant number of food items in the fridge
- has a tendency to forget expiry date and location of the food stored
- can type fast
- prefers typing to mouse interactions
- is reasonably comfortable using the command line interface (CLI)

Value Proposition

This application allows users to monitor their food in a fridge faster than a typical mouse/GUI driven app. It includes features such as the ability to check for the foods that are expiring in a week, and the food items that require replenishing. It will help new homeowners to keep track of their food into different food categories and storage locations which provides additional ease of searching for their food items. In addition, our users will be abe to keep track of past food items that has been added to the fridge to analyse patterns in their purchasing of food.

User Stories

Version	As a ...	I want to ...	So that I can ...
v1.0	new user	see usage instructions	refer to them when I forget how to use the application
v1.0	user	add items into the fridge	keep track of what is in the fridge
v1.0	user	remove items from the fridge	keep track of what is in the fridge
v1.0	user	list all food items in the fridge	know inside the fridge without having to open the fridge itself
v1.0	forgetful user	search for a specific food item	recall if I have that food item inside the fridge
v1.0	organised user	tag the food’s category when I add into fridge (e.g. veggies, meat, dairy)	track what kinds of food I have in the fridge
v1.0	organised user	list food items by category	check specific categories of food

Version	As a …	I want to …	So that I can …
v1.0	organised user	add a storage location for a specific food item	know which part of the fridge is stored in
v1.0	organised user	add the expiry date to each food item I put into the fridge	keep track of what is going to expire
v1.0	lazy user	search for items that are expiring	see what items are about to expire without having to open the fridge
v1.0	lazy user	automatically save information to a data file	safely exit the system without losing information
v1.0	lazy user	automatically retrieve previously saved information from a data file	know the list of items I already have in my fridge
v2.0	lazy user	list all food stored in a part of the fridge	track how full each location is without having to open the fridge
v2.0	meticulous user	specify the weight or quantity of each food item I add into the fridge	know the exact amount of the food item I have left
v2.0	meticulous user	remove a certain weight of each food package item that I am going to use	know how much weight of the item it has left
v2.0	forgetful user	define the lower limits of each food category that the fridge should contain	receive reminders that the food is depleted
v2.0	forgetful user	receive warnings when certain food items are running low after I remove a food item	not personally check for food quantity
v2.0	lazy user	request for a list of food items running low or expiring	easily create a shopping list
v2.0	health conscious user	view the history of food items that I have bought in the past	better track my food habits
Future	forgetful user	store recipes	retrieve my recipes without having to memorise them
Future	lazy user	request a shopping list based on multiple recipes	buy missing food items I need for a few days
Future	financially conscious user	calculate estimated monthly expenses spent on food	keep track of my monthly budget

Non-Functional Requirements

Type	Requirement	Description	Reason

Type	Requirement	Description	Reason
Performance and Scalability	Single User	The product should be for a single user i.e. (not a multi-user product).	Multi-user systems are hard to test, which is unfair for peer testers who will be graded based on the number of bugs they find.
Performance and Scalability	Typing Preferred	The product should be targeting users who can type fast and prefer typing over other means of input.	To increase comparability of products, and to make feature evaluation easier for peer evaluators.
Portability and Compatibility	Platform Independent	The software should work on the Windows, Linux, and OS-X platforms.	Peer testers should be able to use any of these platforms.
Portability and Compatibility	Java -Version	The software should work on a computer that has version 11 of Java i.e., no other Java version installed.	-
Reliability, Availability, and Maintainability	No Remote Servers	The software should not depend on any remote servers.	As it is a locally-runned command line application, there will be no down time for users. It should be available anytime and anywhere.
Security	Human Editable File	The data should be stored locally and should be in a human editable text file. Thus the data is not protected by nature.	To allow advanced users to manipulate the data by editing the data file.
Usability	Easy to Use	It should be very easy for a user to use the system.	There is a detailed user guide provided as well as a help command. The user guide can be accessed here.

Glossary

- **Human-Editable-File**: files with .txt

Appendix: 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.

Launch and shutdown

1. Ensure that you have Java 11 or above installed.

2. Download the latest version of `FridgeFriend` from here.
3. Copy the file to the folder you want to use as the home folder for your `FridgeFriend`.
4. Open your Command Line Terminal in the folder where the `jar` file is located, and run `FridgeFriend` with `java -jar [CS2113-T10-1][FridgeFriend].jar`.
5. Type the command in the command box and press Enter to execute it. e.g. typing list and pressing Enter will show a list of all current food.
6. To terminate the app, use the `bye` command. It is also acceptable to interrupt the Command Line Terminal with `Ctrl+C` or closing the terminal.

Adding food

When adding food, the `FridgeFriend` application requires a very specific format, so that all attributes of a given food can be added to the Fridge.

Here are some inputs you can try:

- `add chicken /cat meat /exp 30-06-2021 /loc lower_shelf /qty 100`
  - Inputs a `chicken` of category `meat`, with expiry date of `30-06-2021`, location of `lower_shelf`, and quantity of `100`.

- `add milk /cat dairy /exp 31-12-2021 /loc fridge_door /qty 2`
  - Inputs a `milk` of category `dairy`, with expiry date of `31-12-2021`, location of `fridge_door`, and quantity of `2`.

- `add Coke /cat beverage /exp 30-07-2021 /loc upper_shelf /qty 5`
  - Inputs a `Coke` of category `beverage`, with expiry date of `30-07-2021`, location of `upper_shelf`, and quantity of `5`.

- `add squid /cat seafood /exp 15-08-2021 /loc freezer /qty 100`
  - Inputs a `squid` of category `seafood`, with expiry date of `15-08-2021`, location of `freezer`, and quantity of `100`.

For subsequent examples below, this guide assumes that these four foods `chicken`, `milk`, `Coke`, and `squid` have been added to the Fridge.

Listing food

The `list` command in `FridgeFriend` has three (3) variations:

- List all food in the fridge.
- List all food of a certain category in the fridge.
- List all food of a certain storage location in the fridge.

The latter two commands will return an output that is a subset of the first command (*List all food*). No items will be listed if no food in the fridge match the conditions specified in the command.

- Test case: `list`

- Expected:

```
Here are the items in your fridge:
1. Food name: chicken, category: MEAT, expiry: 30-06-2021, stored in: LOWER_SHELF,
quantity: 100
2. Food name: milk, category: DAIRY, expiry: 31-12-2021, stored in: FRIDGE_DOOR,
quantity: 2
3. Food name: Coke, category: BEVERAGE, expiry: 30-07-2021, stored in: UPPER_SHELF,
quantity: 5
4. Food name: squid, category: SEAFOOD, expiry: 15-08-2021, stored in: FREEZER,
quantity: 100
```

- Test case: `list meat`
  - Expected:

```
These are the MEAT in your fridge:
1. Food name: chicken, category: MEAT, expiry: 30-06-2021, stored in: LOWER_SHELF,
quantity: 100
```

- Test case: `list freezer`
  - Expected:

```
These are the food stored in FREEZER:
1. Food name: squid, category: SEAFOOD, expiry: 15-08-2021, stored in: FREEZER,
quantity: 100
```

Removing a food

Removing a food while all foods are being shown.

1. Prerequisites: List all foods using the `list` command. Multiple food in the list.
2. Test case: `remove squid /qty 100`
   - Expected: `squid` is deleted from the list. Details of the deleted food shown in the status message. Additional warning that `seafood` is running low may appear.

3. Test case: `remove chicken /qty 50`
   - Expected: Quantity of `chicken` is reduced by 50. Details of the removed food shown in the status message. Additional warning that `meat` is running low may appear.

4. Test case: `remove coke /qty 1`
   - Expected: `Food specified not found.` Note that the search for food name in this command is case sensitive; note that when `Coke` was added, it was with a capital ‘C’. No food is removed.

5. Test case: `remove Coke /qty 69`
   - Expected: `Not enough in fridge to remove!` User attempted to remove a greater quantity than what was available in the fridge. Error message was thrown. No food is removed.

6. Test case: `remove chicken`
   - Expected: `Sorry my friend, you have entered an invalid input. Enter 'help' for more information about the correct input format.` It is required to specify the quantity to be removed with `/qty`.

Searching for a food

Checks if a food is in the fridge, and if it is found, outputs all the food in a list that match the search query. If it is not found, outputs `You do not have FOOD_INPUT in your fridge.`

- Test case: `search chicken`

  - Expected:

```
These are the chicken in your fridge:
    1. Food name: chicken, category: MEAT, expiry: 30-06-2021, stored in: LOWER_SHELF,
    quantity: 100
```

- Test case: `search pear`
  - Expected: `You do not have pear in your fridge.`

- Test case: `search chick`

  - Expected:

```
These are the chick in your fridge:
    1. Food name: chicken, category: MEAT, expiry: 30-06-2021, stored in: LOWER_SHELF,
    quantity: 100
```

Expiring food

This feature might be slightly more challenging to test, since the user tester has to input a food that has an expiry date within 7 days of their **local system time**, as of the date of testing.

1. Prerequisites: Add a food with an expiry date within 7 days of the user's system date.
   - Modify the input in the above add command in order to fulfil this requirement.
   - Example: The current date on my system time is `29-03-2021`.
     - Prior to testing, I perform the command `add duck /cat meat /exp 30-03-2021 /loc lower_shelf /qty 100`.
     - Thus, this food should expire in 1 day, within the 7 days required to trigger the `expiring` command.

2. Test case: `expiring`
    - Expected:

```
These are the food expiring in the next week:
    1. Food name: duck, category: MEAT, expiry: 30-03-2021, stored in: LOWER_SHELF,
    quantity: 100
```

Runninglow and Setlimit

By default, the `limits` for all food is set at `500`. Use `setlimit` to modify this number for a certain food category.

1. Preparation: Change the limits for `meat`, `dairy`, `beverage`, `seafood`, since they are the categories of food we have added to the fridge so far.
    - `setlimit meat /qty 0`
    - `setlimit dairy /qty 5`
    - `setlimit beverage /qty 3`
    - `setlimit seafood /qty 50`

2. Test case: `runninglow`
    - Expected:

```
You are running low on food in these categories:
1. VEGETABLE quantity: 0 out of 500
2. FRUIT quantity: 0 out of 500
3. EGG quantity: 0 out of 500
4. DAIRY quantity: 2 out of 5
5. COOKED_DISH quantity: 0 out of 500
6. READY_TO_EAT quantity: 0 out of 500
7. FROZEN quantity: 0 out of 500
8. OTHER quantity: 0 out of 500
```

Note that `MEAT`, `SEAFOOD` and `BEVERAGE` category are omitted from the above list, as they are not "running low". There is sufficient food from the given category in the fridge, because it is above the quantity we have specified by `setlimit`.

History

While the history command may be simple to test, the expected output may vary depending on the food added by the user during testing.

Therefore, user testers should manually compare the results of the `history` command with their actual input during testing, and see if they correspond.

1. Prerequisites: Only the input in the above add command was added to the list. No other food was added to the list.

   - Note: To reset the history, use `history clear`.
   - `add chicken /cat meat /exp 30-06-2021 /loc lower_shelf /qty 100`
   - `add milk /cat dairy /exp 31-12-2021 /loc fridge_door /qty 2`
   - `add Coke /cat beverage /exp 30-07-2021 /loc upper_shelf /qty 5`
   - `add squid /cat seafood /exp 15-08-2021 /loc freezer /qty 100`

2. Test case: `history`
   - Expected:

```
This is the full history of items you've added in the fridge:
    1. Food name: chicken, category: MEAT, expiry: 30-06-2021, stored in: LOWER_SHELF,
quantity: 100
    2. Food name: milk, category: DAIRY, expiry: 31-12-2021, stored in: FRIDGE_DOOR,
quantity: 2
    3. Food name: Coke, category: BEVERAGE, expiry: 30-07-2021, stored in: UPPER_SHELF,
quantity: 5
    4. Food name: squid, category: SEAFOOD, expiry: 15-08-2021, stored in: FREEZER,
quantity: 100
```

Saving data

Dealing with missing/corrupted data files.

All data is stored in the `/data` folder in the same folder as `FridgeFriend.jar`. Three (3) text files will be generated in the folder during usual execution of `FridgeFriend`. These files are `fridgeData.txt`, `limitsData.txt`, and `historyData.txt`, and are used to store data in the disk for various commands during the usual operation of `FridgeFriend`.

Details of Files:

- Are automatically loaded when `FridgeFriend` starts, with the exception of `historyData.txt` which is loaded only when `history` command is invoked.
- Are updated when the `FridgeFriend` application is terminated using the `bye` command, with the exception of `historyData.txt` which is only updated whenever a successful add command is executed.
- Both `fridgeData.txt` and `limitsData.txt` will also be updated after every command is successfully executed.
- *Missing data file*: A new, blank file will automatically be created upon launching `FridgeFriend`. The fridge at program launch will be empty, and the limits of all food categories will be reset to the default of 500. No further action needed.
- *Corrupted data file*: The table below outlines the different behaviours according to different cases.

Corruption Type	Error Message	Behaviour

Corruption Type	Error Message	Behaviour
Corrupted `fridgeData.txt` file	`There was an error loading the data for FridgeFriend! Index 1 out of bounds for length 1`	`FridgeFriend` will load the contents of the text file until the point in the file where corrupted/invalid data is encountered. User can recover the contents of the file by manually inspecting the text file and removing invalid content.
Corrupted readable categories in `limitsData.txt`	No error message	User can only identify that data has been corrupted when using the `runninglow` command. The quantity of the invalid categories due to corruption will be reset to `500`.
Corrupted unreadable categories in `limitsData.txt`	`There was an error loading the data for FridgeFriend! Index 1 out of bounds for length 1`	The quantity limits will be parsed up until the corrupted unreadable category. Subsequent quantity limits in the file would not be parsed, and will be reset to the default of `500`.
Corrupted integer quantities in `limitsData.txt`	No error message	User can only identify that data has been corrupted when using the `runninglow` command. The limit will be updated to the corrupted value.
Corrupted non-integer quantities in `limitsData.txt`	`There was an error loading the data for FridgeFriend! Sorry my friend, the quantity must be a number.`	The quantity limits will be parsed up until the corrupted non-integer value. Subsequent quantity limits in the file would not be parsed, and will be reset to the default of `500`.
Corrupted readable categories in `historyData.txt`	No error message	User can only identify that data has been corrupted when using the `history` command. The `history` command will continue to print out the contents of the file, including the corrupted data. While this corrupted data would not affect program flow, it may create unexpected output. If necessary, users can manually inspect the file and remove unwanted data at their own discretion.

Corruption Type	Error Message	Behaviour
Corrupted unreadable categories in `historyData.txt`	No error message	User can only identify that data has been corrupted when using the `history` command. The contents of the entire file will not be parsed, even if there is valid content in some parts of the file. The contents of the file will not be affected. If necessary, users can manually inspect the file and remove corrupted data, in order to salvage any valid data.

Attribution

The format of this User Guide was adapted from AddressBook Level 3(AB3) Developer Guide.