

Cooking Papa - Developer Guide

1. Setting up	2
2. Design	2
2.1. Architecture	2
2.2. UI component	4
2.3. Logic component	5
2.4. Model component	6
2.5. Storage component	8
2.6. Common classes	9
3. Implementation	9
3.1. Add Recipe to the Cookbook	9
3.2. View recipe in the cookbook	13
3.3. Search for recipes based on ingredients in the inventory	16
3.4. Adding ingredients to inventory and cart	20
3.5. Remove ingredients of a recipe from the inventory	22
3.6. Moving ingredients from cart to inventory	24
3.7. Add a Recipe's Ingredients to Cart	26
3.8. Export ingredients in cart to PDF file	28
3.9. Configuration	31
4. Documentation	31
5. Testing	31
6. Dev Ops	31
Appendix A: Product Scope	32
Appendix B: User Stories	32
Appendix C: Use Cases	33
Appendix D: Non Functional Requirements	35
Appendix E: Glossary	35
Appendix F: Product Survey	35
Appendix G: Instructions for Manual Testing	36
G.1. Launch and Shutdown	36
G.2. Adding a recipe to the cookbook	36
G.3. Removing a recipe from the cookbook	37
G.4. Searching for recipes by tags	37
G.5. Exporting the cart to a PDF file	37
Appendix H: Effort	38

By: **AY1920S2-CS2103T-F11-4** Since: **Jun 2016** Licence: **MIT**

1. Setting up

Refer to the guide [here](#).

2. Design

2.1. Architecture

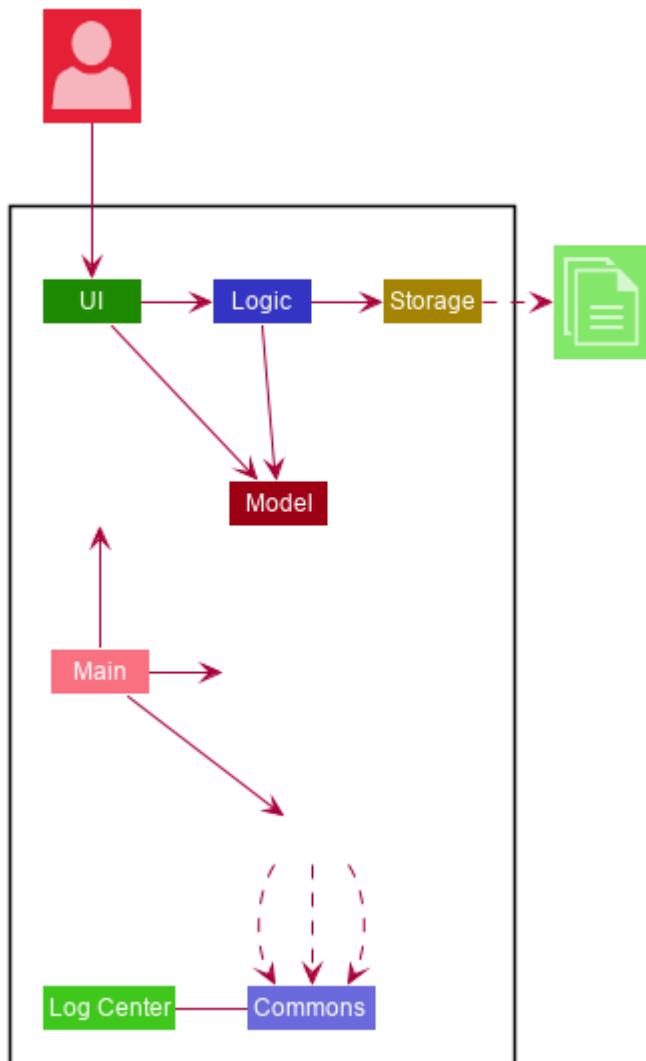


Figure 1. Architecture Diagram

The **Architecture Diagram** given above explains the high-level design of the App. Given below is a quick overview of each component.

Main has two classes called **Main** and **MainApp**. It is responsible for,

- At app launch: Initializes the components in the correct sequence, and connects them up with each other.
- At shut down: Shuts down the components and invokes cleanup method where necessary.

Commons represents a collection of classes used by multiple other components. The following class

plays an important role at the architecture level:

- **LogsCenter** : Used by many classes to write log messages to the App's log file.

The rest of the App consists of four components.

- **UI**: Controls the user interface of the App.
- **Logic**: Executes the commands of the App.
- **Model**: Holds the data of the App in-memory.
- **Storage**: Reads data from, and writes data to, the hard disk.

Each of the four components

- Defines its *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.

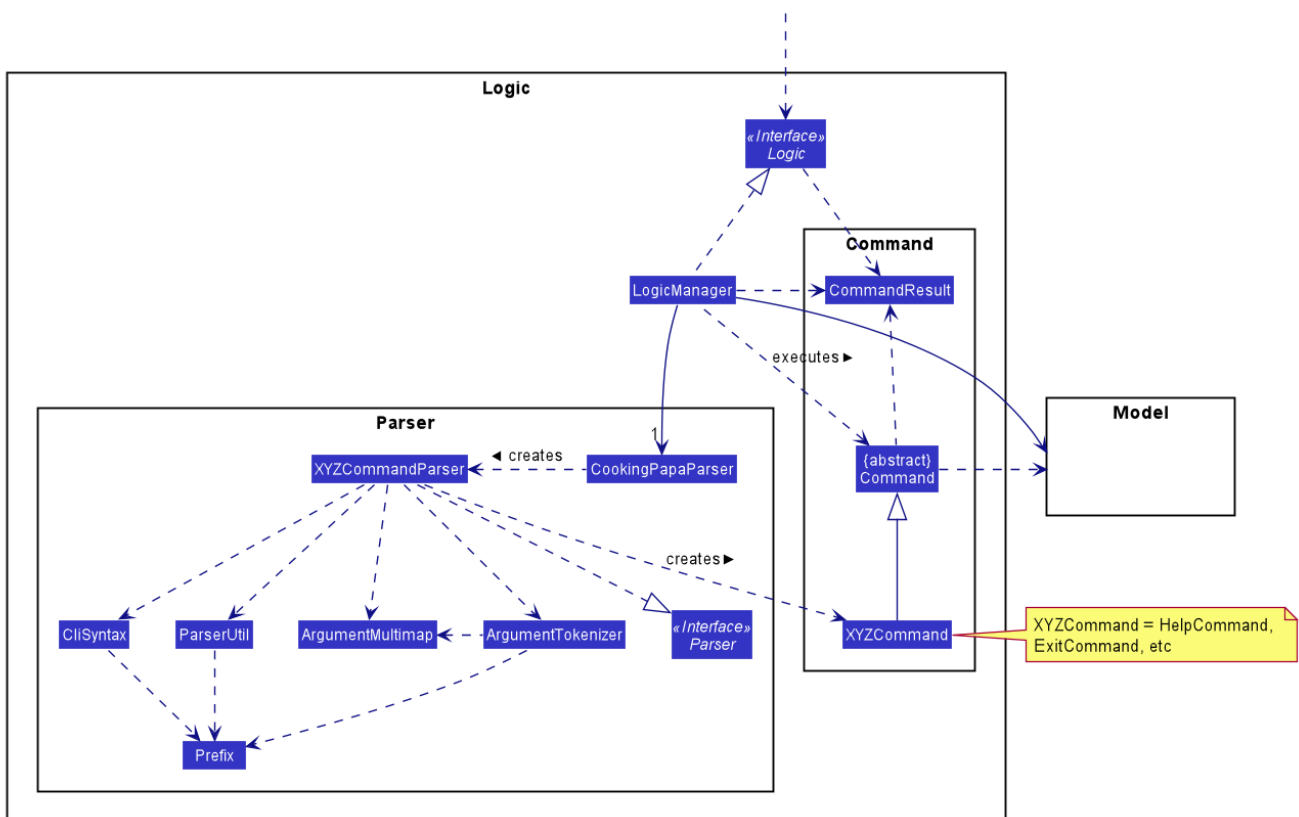


Figure 2. Class Diagram of the Logic Component

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 **cookbook remove recipe 2**.

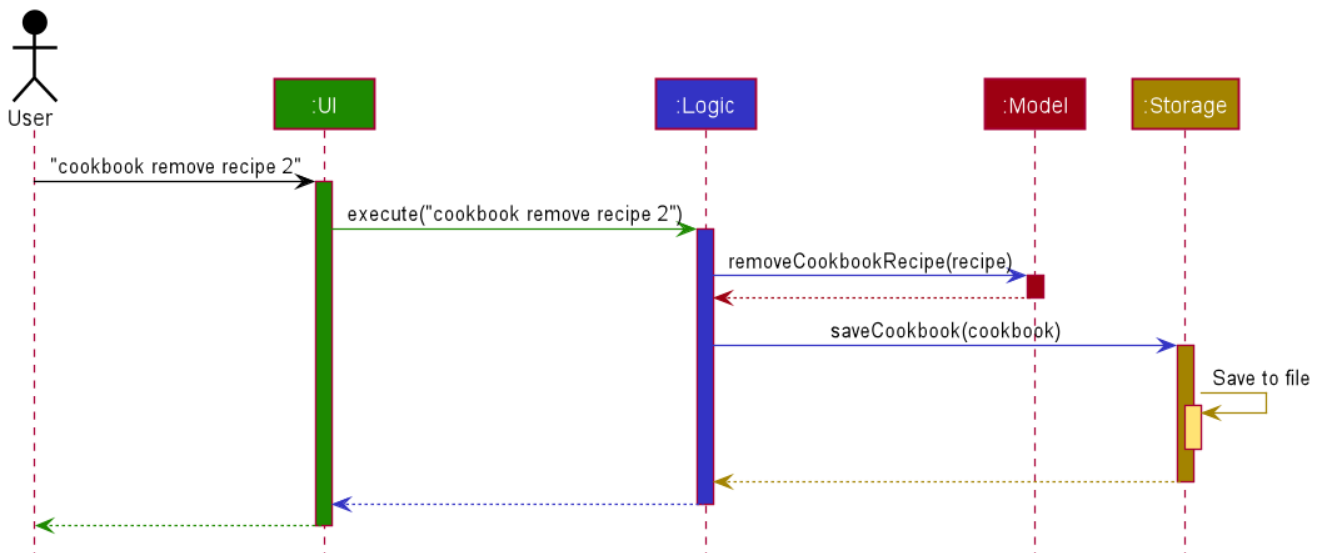


Figure 3. Component interactions for `cookbook remove recipe 2` command

The sections below give more details of each component.

2.2. UI component

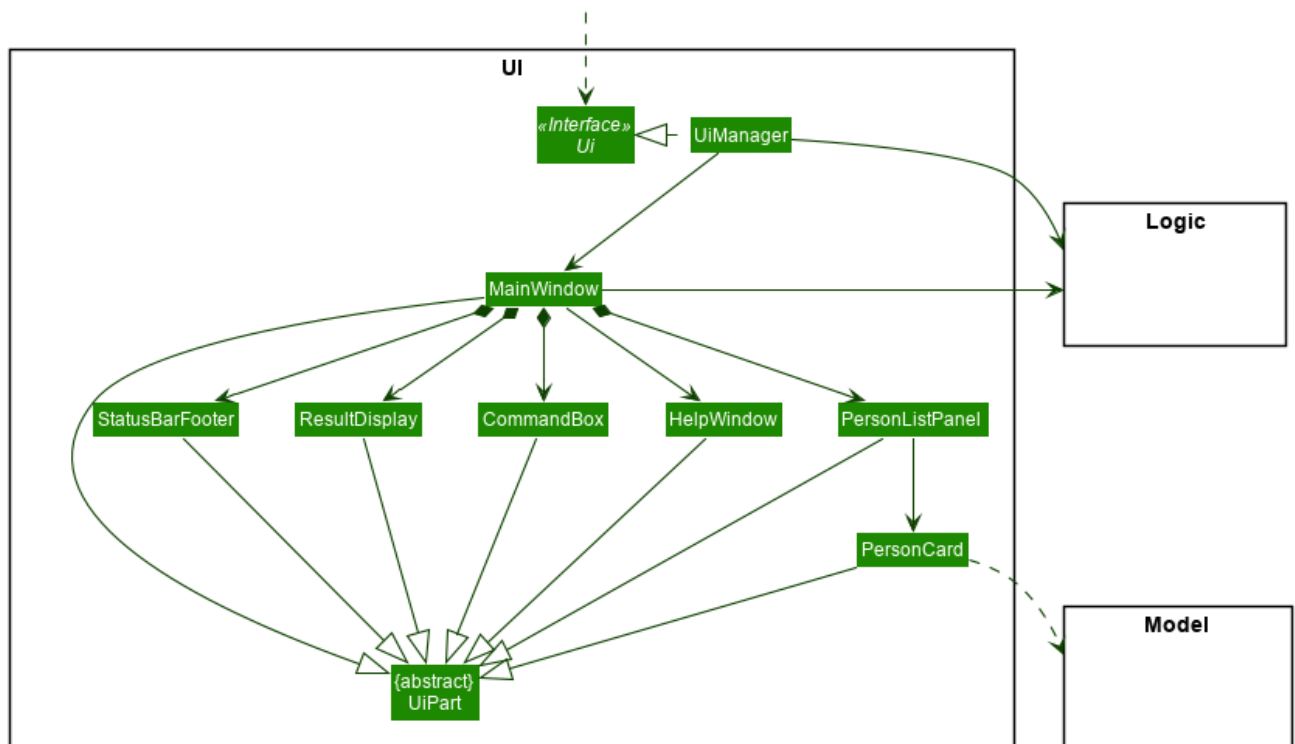


Figure 4. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g. `CommandBox`, `ResultDisplay`, `PersonListPanel`, `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,

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

2.3. Logic component

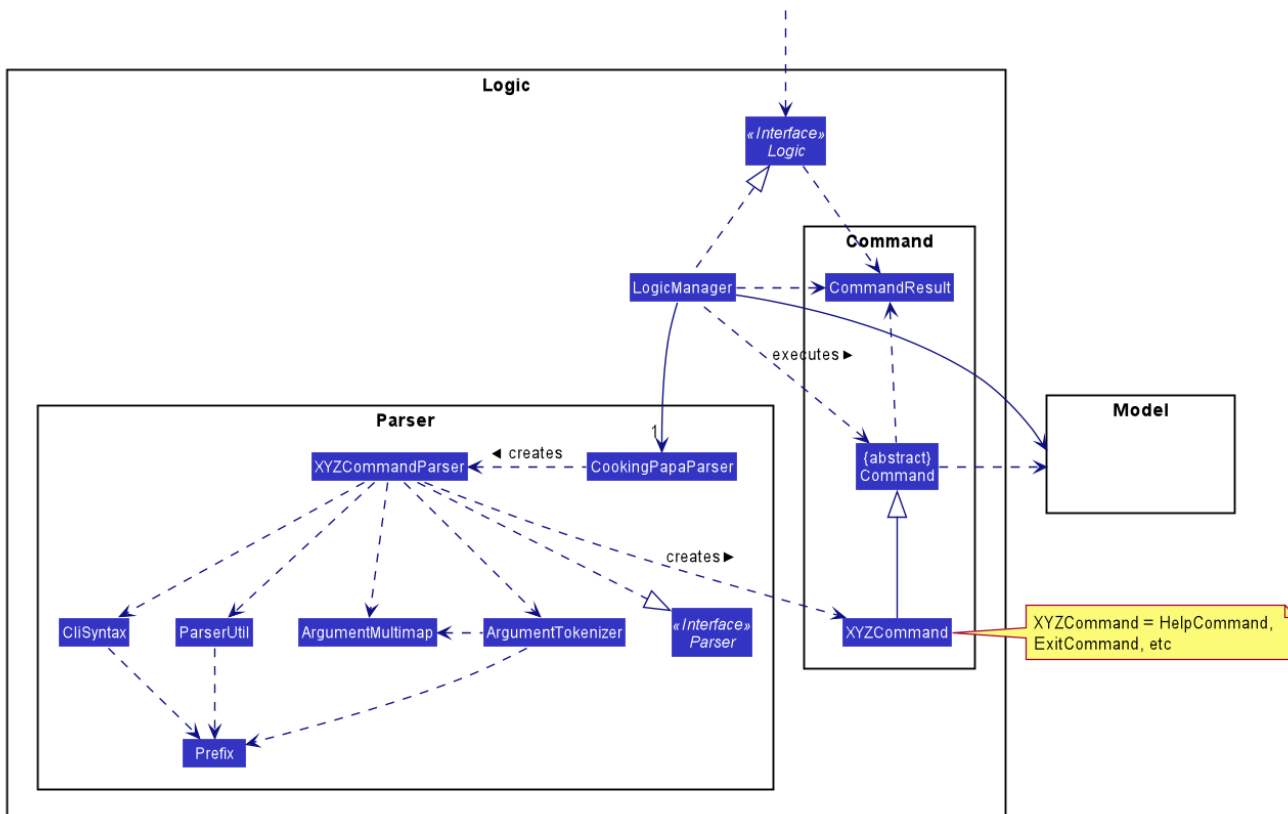


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. **Logic** uses the `CookingPapaParser` 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 recipe).
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("cookbook remove recipe 2")` API call.

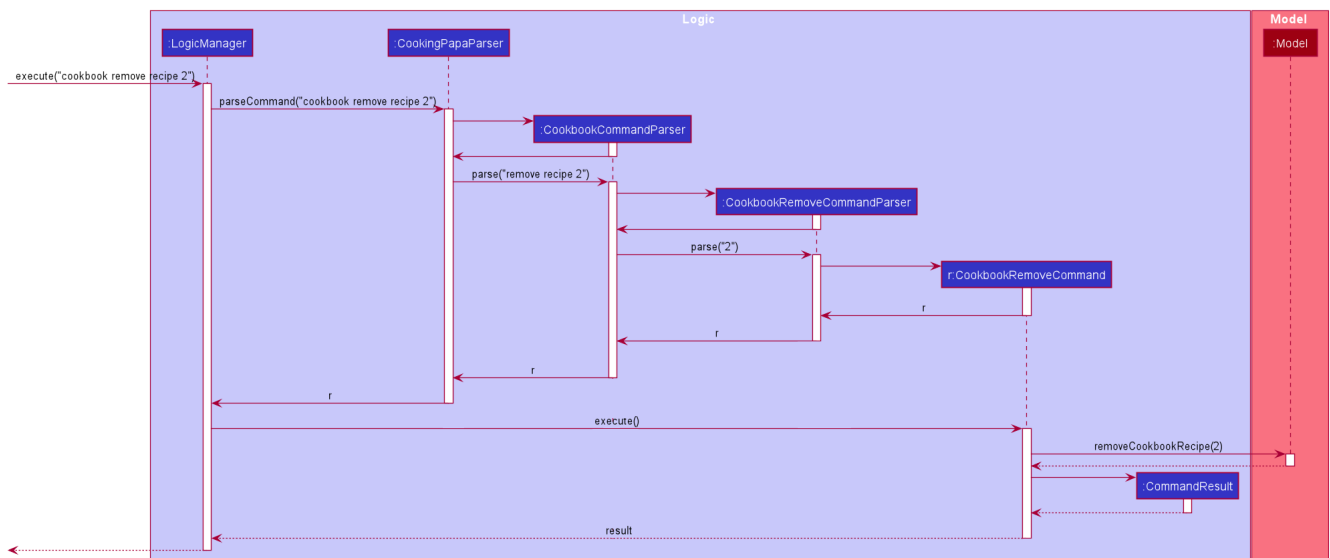


Figure 6. Interactions Inside the Logic Component for the **cookbook remove recipe 2** Command

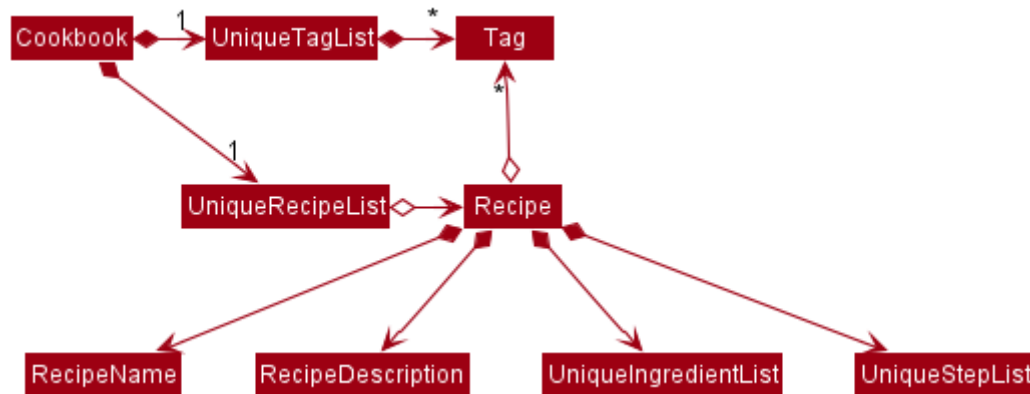
NOTE

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.

2.4. Model component

As a more OOP model, we can store a **Tag** list in **Cookbook**, which **Recipe** can reference. This would allow **Cookbook** to only require one **Tag** object per unique **Tag**, instead of each **Recipe** needing their own **Tag** object. An example of how such a model may look like is given below.

NOTE



2.5. Storage component

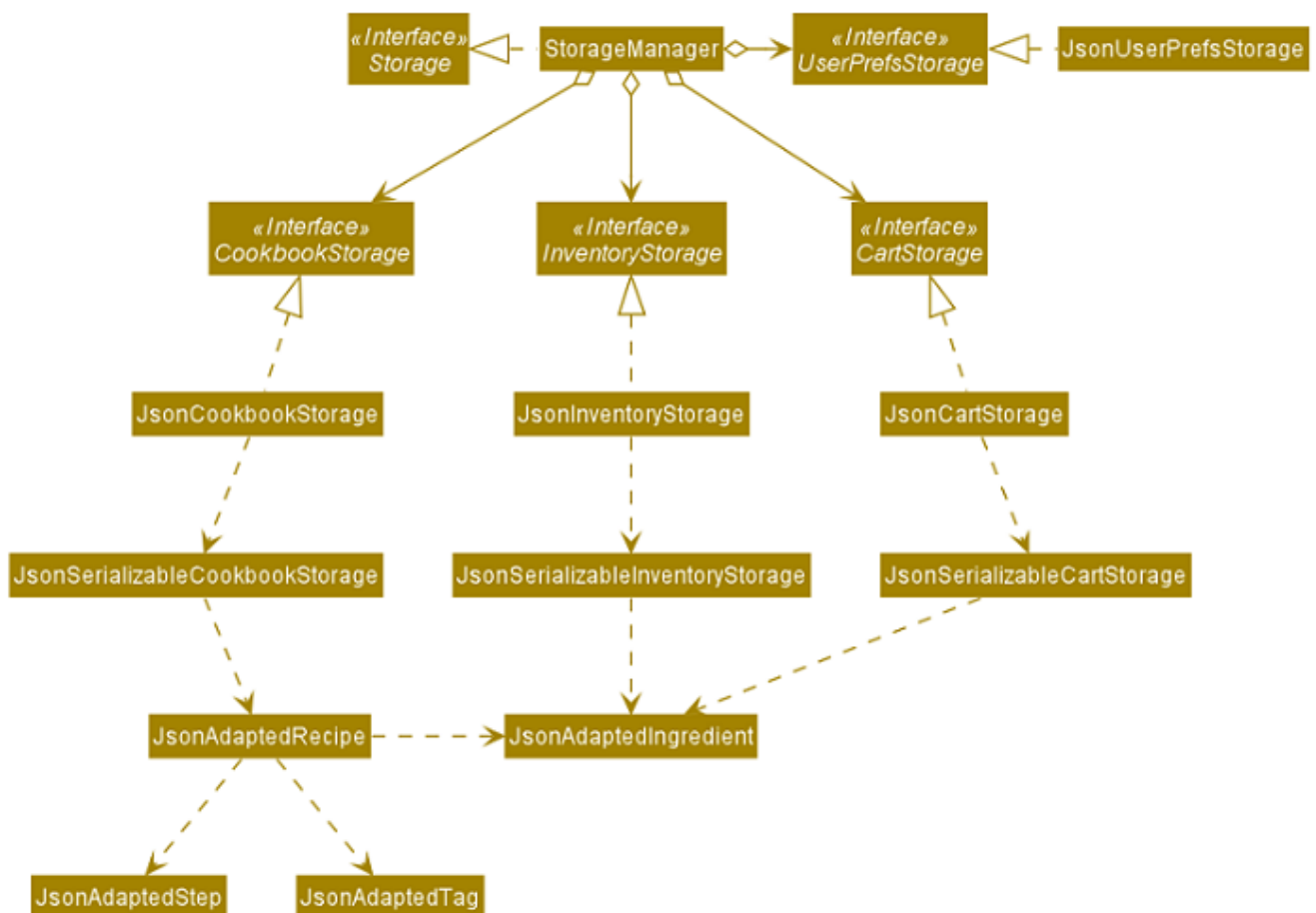


Figure 8. Structure of the Storage Component

API : **Storage.java**

The **Storage** component,

- can save **UserPref** objects in json format and read it back.

- can save `Cookbook` data in json format and read it back.
- can save `Inventory` data in json format and read it back.
- can save `Cart` data in json format and read it back.

2.6. Common classes

Classes used by multiple components are in the `seedu.addressbook.commons` package.

3. Implementation

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

3.1. Add Recipe to the Cookbook

3.1.1. Implementation

The recipe addition mechanism is facilitated by `CookbookAddCommand`, which extends the `Command` abstract class. The format of the command is as follows: `cookbook add recipe n/NAME d/DESCRIPTION [i/INGREDIENT_NAME]... [q/INGREDIENT_QUANTITY]... [t/TAG]...`.

This command is implemented this way to allow a user to add a recipe with optional fields (ingredients, steps, tags) - only the recipe name and recipe description are mandatory fields. This way, a user does not have input all the fields that they may not have at the moment to create a recipe. After creating the skeleton of the recipe, the user can then use the other `Cookbook` commands to add ingredients and steps to the recipe. However, one key point is that should ingredient names be provided, the same number of ingredient quantities have to be provided as well.

Below is a step by step sequence of what happens when a user enters this command:

1. The user enters a recipe adding command using the command line input `cookbook add recipe n/NAME d/DESCRIPTION [i/INGREDIENT]... [q/QUANTITY]... [s/STEP_DESCRIPTION]... [t/TAG]...`.
2. `CookingPapaParser` parses the user input and checks if it is valid. If it is invalid, i.e. an unknown command category, a `ParseException` will be thrown. If the input is valid, with the command category `cookbook`, a new `CookbookCommandParser` is created.
3. `CookbookCommandParser` then parses `add recipe n/NAME d/DESCRIPTION [i/INGREDIENT]... [q/QUANTITY]... [s/STEP_DESCRIPTION]... [t/TAG]...`. If it is invalid, i.e. an unknown command word, a `ParseException` will be thrown. If the input is valid, with the command category `add`, a new `CookbookAddCommandParser` is created.
4. `CookbookAddCommandParser` parsers `recipe n/NAME d/DESCRIPTION [i/INGREDIENT]... [q/QUANTITY]... [s/STEP_DESCRIPTION]... [t/TAG]...` and checks if `n/NAME` and `d/DESCRIPTION` are provided. It then parses the input into the following fields: recipe name, recipe description, ingredients, steps, and tags.

Note that the ingredient names and ingredient quantities provided must be the same, or a `ParseException` will be thrown:

```

if (names.size() != quantities.size()) {
    throw new ParseException(
        String.format(MESSAGE_DIFFERENT_NUMBER_OF_INPUTS, names.size(), quantities
            .size()));
}

```

5. These fields are then passed as parameters for `Recipe`, which is then passed as the parameter for `CookbookAddCommand` and returned to `LogicManager`.
6. `LogicManager` calls `CookbookAddCommand#execute()` which checks if the cookbook already contains the same recipe with the same name, description, ingredient names, ingredient quantities, and tags using `Model#hasCookbookRecipe()`.

If there is a duplicate, a `CommandException` is thrown, stating that the user is attempting to add a duplicate recipe:

```

if (model.hasCookbookRecipe(toAdd)) {
    throw new CommandException(MESSAGE_DUPLICATE_RECIPE);
}

```

7. If `CommandException` is not thrown, `Model#addCookbookRecipe` will be executed, with the recipe to be added as a parameter.
8. `Model#addCookbookRecipe()` then executes `Cookbook#addRecipe()`, which adds the recipe to the cookbook, and the `FilteredList<Recipe>` representing the recipes in the cookbook are updated with `Model#updateFilteredCookbookRecipeList()`:

```

updateFilteredCookbookRecipeList(PREDICATE_SHOW_ALL_RECIPES)

```

where `PREDICATE_SHOW_ALL_RECIPES = unused → true`.

9. A `CommandResult` with the text to display to the user is then returned to `LogicManager`, which can be passed back to `MainWindow`, which displays it to the user on the CLI and GUI: `resultDisplay.setFeedbackToUser(commandResult.getFeedbackToUser())`. The text displayed will notify the user on whether their addition was successful.

The following `Recipe` object diagram is an overview of the attributes of a `Recipe` object:

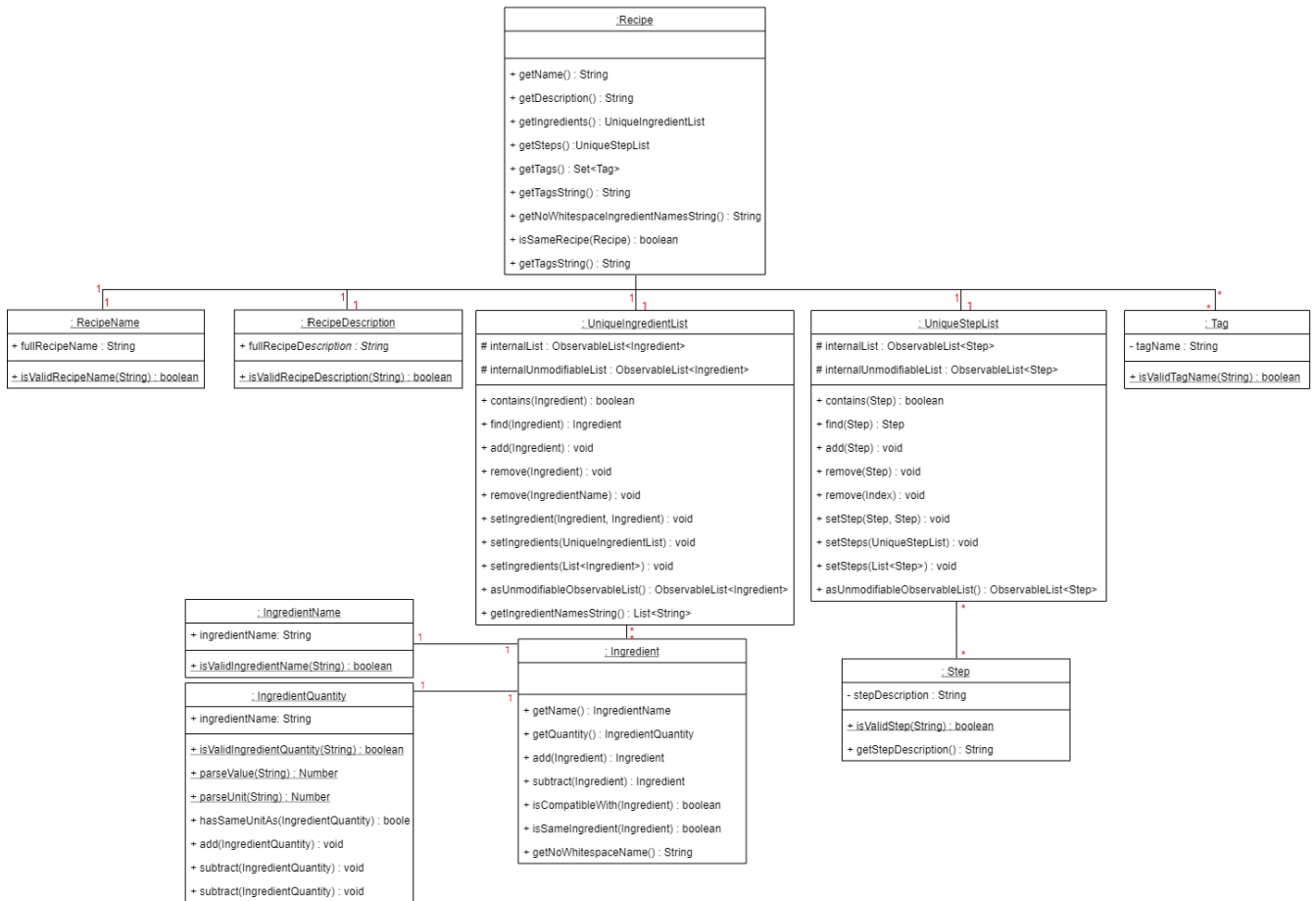


Figure 9. UML object diagram of Recipe providing an overview on how the various objects interact

The following sequence diagram shows how the recipe adding function works (full command [cookbook add recipe n/Recipe name d/Recipe description i/Ingredient 1 q/1 piece i/Ingredient 2 q/20 ml s/Do step 1 s/Do step 2 t/This t/Is t/A t/Tag] omitted from diagram for brevity):

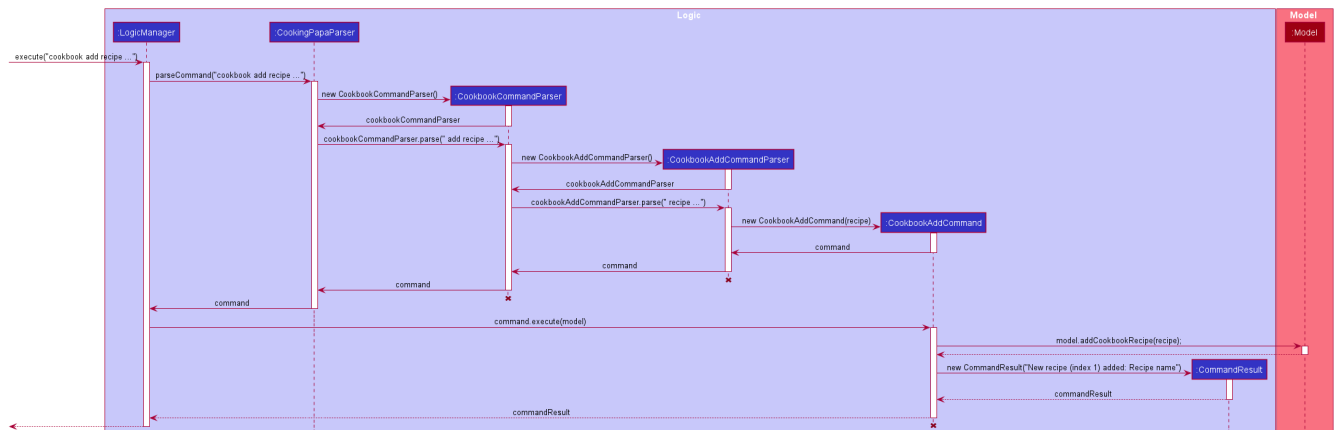


Figure 10. Sequence diagram for CookbookAddCommand

3.1.2. Design considerations

Aspect 1: How to parse optional parameters

Table 1. Design considerations for parsing optional parameters


	Design A: Parse each category separately (current choice)	Design B: Parse all the categories together
--	---	---

Description	Each category (ingredient name, ingredient quantity, step description, tag) are parsed separately and returned as <code>List</code> . If the returned <code>List</code> is empty, then it means that that field was not provided in the input, and will be set to an empty <code>List</code> in the recipe.	Each category will be parsed together in one function in <code>CookbookAddCommandParser</code>
Pros	<ul style="list-style-type: none"> Provides more flexibility for the user and does not make it mandatory to input fields that they may not necessarily have. No need to deal with null values, can simply check if list is empty. 	<ul style="list-style-type: none"> Straightforward No need to create and call multiple methods from other classes
Cons	<ul style="list-style-type: none"> More methods have to be executed which may increase time and NPath complexity. Debugging and tracing becomes more confusing due to the method being defined in the lowest level of abstraction. 	<ul style="list-style-type: none"> Have to deal with null values and include null checks (<code>ifPresent()</code> etc.) Method will be very long and decreases readability

Aspect 2: Result to show user

Table 2. Design considerations for results to show users

	Design A: Show a short result on the success of the command	Design B: Show all the details back to the user
Description	Show a message to a usage which notifies them that the command was successful in adding the recipe to the cookbook.	Shows a message similar to design choice A, and also show all the details of the added recipe.
Pros	<ul style="list-style-type: none"> Short and succinct message, tells the user what they need to know User interface is cleaner and more intuitive, and does not overload users with unnecessary information 	<ul style="list-style-type: none"> Easier to implement

Cons	<ul style="list-style-type: none"> Requires the graphical user interface to be able to toggle and show recipes, without the need for a command, implemented here: 	<ul style="list-style-type: none"> Overloads the user with unnecessary information Requires result display to take up more space than required, to reduce the need for users to scroll down the result display.
------	--	---

3.2. View recipe in the cookbook

The user may use this command to view a recipe in the cookbook. This command is integrated into the Graphical User Interface (GUI) through a button.

3.2.1. Implementation

The recipe viewing mechanism (via the command line input) is facilitated by `CookbookViewCommand`, which extends the `Command` abstract class. The format is as follows: `cookbook view recipe INDEX`, which index has to be a valid integer that is not out of bounds.

The recipe viewing mechanism (via the GUI) is facilitated by `RecipeCard`, which extends the `UiPart` abstract class. It is triggered upon clicking the "view" icon in the recipe panel:



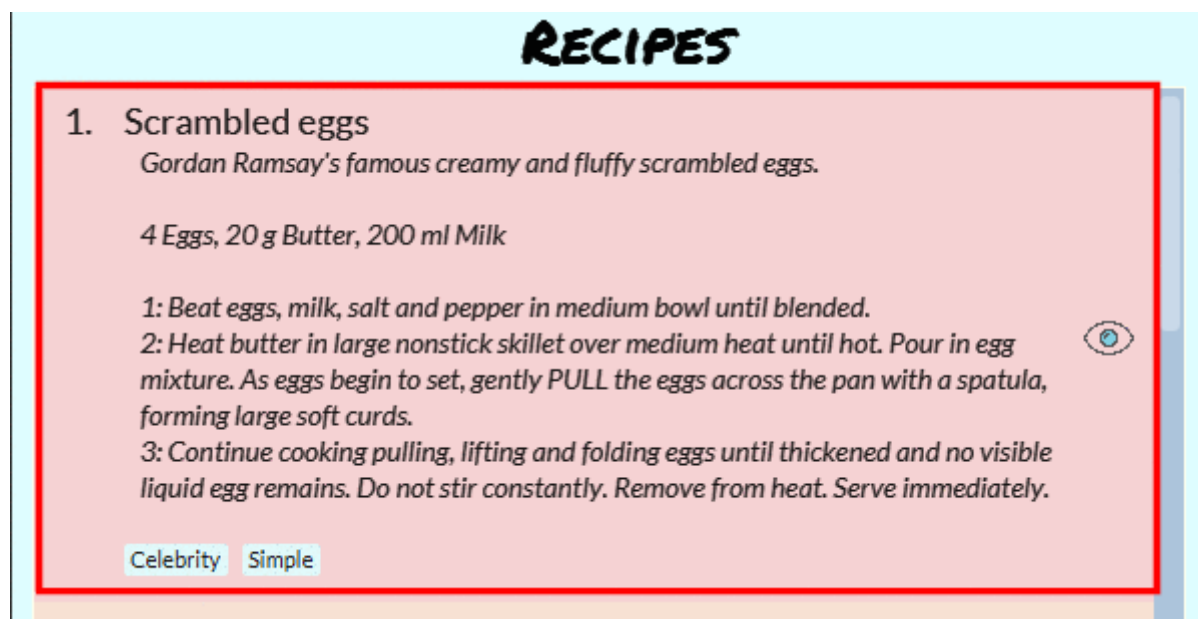
Implementing this function, `cookbook view recipe` through a button in the GUI allows user to view the details of a recipe with a click of a button, greatly increasing convenience and user experience. The button also had to be "activated" without the button, as the command still had to be testable through the command line.

Below is a step by step sequence of what happens when a user enters this command:

1. The user enters a view recipe command using the command line input `cookbook view recipe INDEX`.
2. `CookingPapaParser` parses the user input and checks if it is valid. If it is invalid, i.e. an unknown command category, a `ParseException` will be thrown. If the input is valid, with the command category `cookbook`, a new `CookbookCommandParser` is created.
3. `CookbookCommandParser` then parses `view recipe INDEX`. If it is invalid, i.e. an unknown command word, a `ParseException` will be thrown. If the input is valid, with the command category `view`, a new `CookbookViewCommandParser` is created.
4. `CookbookViewCommandParser` then parses `recipe INDEX` and checks if the `String` contains "recipe",

and an index. If either are absent, a `ParseException` will be thrown. If the `String` is valid, a `CookbookView` is created.

5. `CookbookViewCommandParser` then returns a `CookbookViewCommand` to `LogicManager`.
6. `LogicManager` calls `CookbookViewCommand#execute()` which checks if the provided `Index` is within the bounds of the `FilteredCookbookRecipeList()` in `Cookbook`, i.e. `index.getZeroBased() >= list.size()`. If it is not, a `CommandException` will be thrown. If it is valid, a `CommandResult` is created with a boolean value `true`.
7. A `CommandResult` with the text to display to the user will be returned to `LogicManager`. The `CommandResult` is then passed back to `MainWindow`. The boolean value stated in step 6 determines whether a successfully parsed command is a `cookbook view recipe INDEX` command.
8. `MainWindow#handleViewRecipe` is then executed, which creates a new `CookbookPanel` with the same set of data, calling `CookbookPanel#handleViewRecipe`, which creates new `RecipeCard`s for `Cookbook`, and for the `RecipeCard` that has an index equal to the index processed from the user's input, it will create a `RecipeCard` that toggles open the recipe details. More on how the `RecipeCard` manages this will be discussed in the following section on how clicking on a button in the GUI has the same effect as the `cookbook view recipe INDEX` command.
9. Lastly, the user then is shown a `CookbookPanel` with the selected recipe toggled open, which displays the details of that recipe:



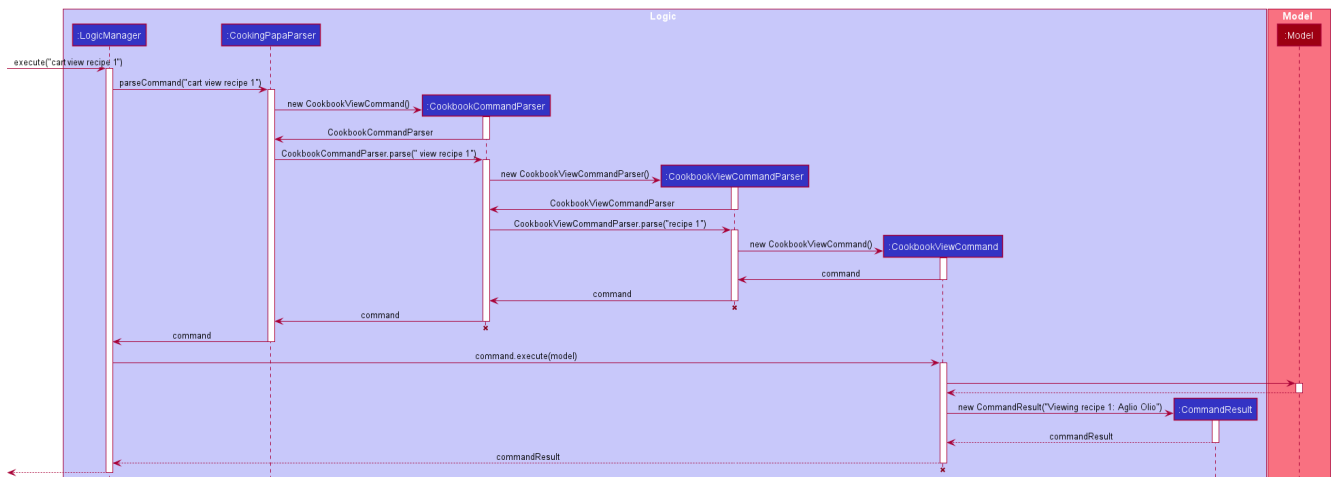
Below is a step by step sequence of what happens when a user clicks the button on the GUI:

1. When the button is pressed, the `onAction` method, `RecipeCard#handleViewButtonAction()` is executed. A `RecipeCard` has a variable `isFullyDisplayed`, which indicates whether it is displaying an overview of the recipe, or fully displaying details of the recipe.
2. If `isFullyDisplayed` is false, i.e. the `RecipeCard` is currently displaying an overview of the recipe, `RecipeCard#displayRecipeComplete()` is executed, which replaces the text displayed by the FXML object, `Label`, with the full details of the recipe.
3. If `isFullyDisplayed` is true, i.e. the `RecipeCard` is currently fully displaying the details of the recipe, `RecipeCard#displayRecipeOverview()` is executed, which replaces the text displayed by the FXML object, `Label` with the overview of the recipe.

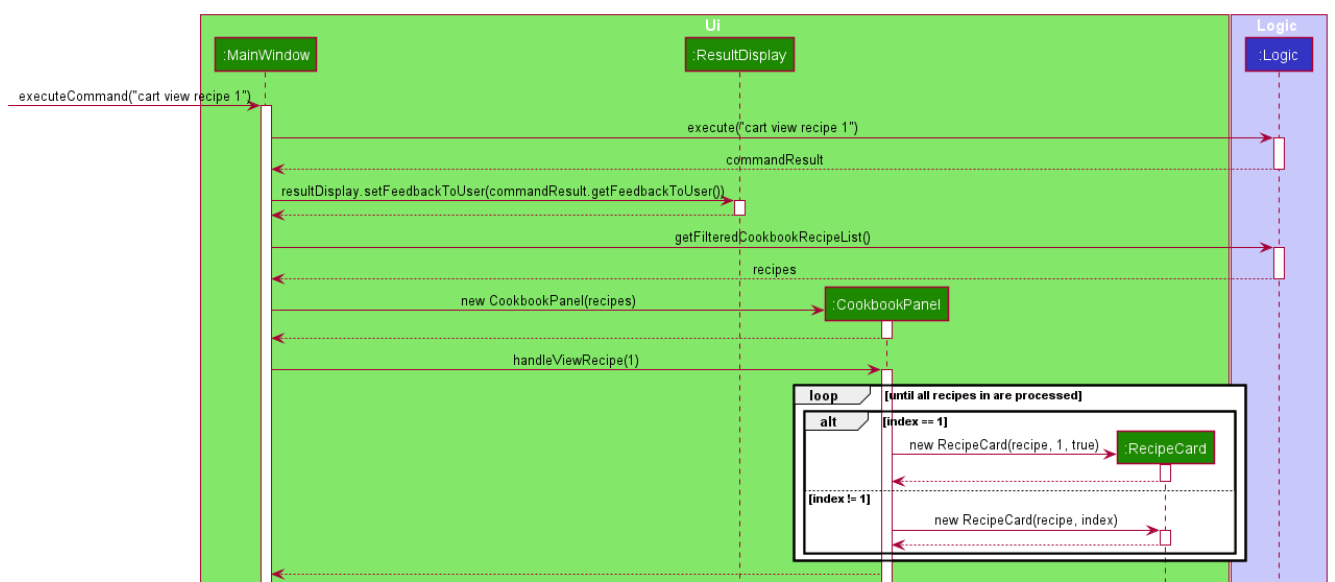
- Both methods executed in step 3 and 4 will flip the boolean value of `isFullyDisplayed`, and this means that the next time the button for the same recipe is clicked, it toggles back. For example, if a recipe with its overview shown has its view button clicked, it will show the full details of the recipe. If the button is clicked again, it toggles, and shows the overview of the recipe.

This feature is not reflected with `cookbook view recipe INDEX` when it is entered again in the command line, because the function of the command is to view a recipe, not to "un-view" it.

The following sequence diagram shows how the recipe viewing function interacts between the classes in **Logic**:



The following sequence diagram shows how the recipe viewing function interacts between the classes in **Ui**:



3.2.2. Design considerations

Aspect: what UI component to display the toggled content

	Design A (current choice): toggles the content in the recipe panel	Design B: add a new UI component that pops up, i.e. overlay
--	--	---

Description	The content in the recipe panel can freely switch from overview to full details of a recipe.	A UI component appears as a small overlay, displaying the details of a recipe. The overlay can then be "exited" by clicking on an area within the application that is outside of the overlay.
Pros	<ul style="list-style-type: none"> • Intuitive that clicking the button once more should return to the previous state • Increases functionality of the GUI, rather than just a "skin" 	No need to interact between various UI components, as much as design A
Cons	There is a need to keep track of the state of a <code>RecipeCard</code> , which means more constructors and conditional statements to implement.	Difficult to implement as it includes creating an entirely new component (overlay) with different features than the existing one. The effort estimated did not seem to be worth, as the use is limited to just this command.

3.3. Search for recipes based on ingredients in the inventory

The user may use this command to search for recipes that they can cook using the ingredients available in their inventory.

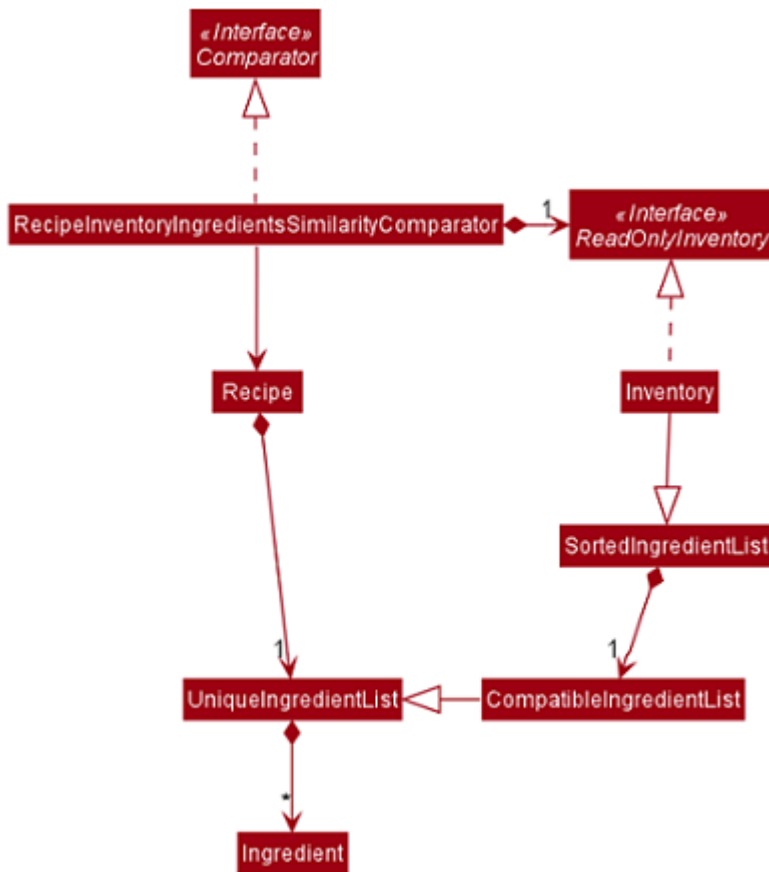
3.3.1. Implementation reasoning

This command was implemented to address users needs of easily finding a recipe based on the ingredients they have. It allows users to whip up a meal without having to go grocery shopping if they are short of time. This feature sorts recipes by how much the inventory fulfils their ingredient requirements, and filters out recipes whose ingredient requirements are not met at all. Users can immediately see at the top of the cookbook the recipes that their ingredients are most suitable for preparing. A user can use this feature by typing the command: `cookbook search inventory`.

3.3.2. Implementation

The comparison between the ingredients a recipe requires and the ingredients in the inventory is facilitated by the `RecipeInventoryIngredientsSimilarityComparator`. It extends `Comparator<Recipe>` and stores the inventory being used for ingredient comparison. Additionally, it implements the method `calculateSimilarity()`, which accepts a `Recipe` and a `ReadOnlyInventory` as parameters, and returns a double value between 0 and 1 (inclusive) that represents the proportion of the recipe's ingredient requirements that are fulfilled.

The following class diagram summarizes how the `RecipeInventoryIngredientsSimilarityComparator` interacts with `Recipe` and `Inventory`:



The `calculateSimilarity()` method first calculates the proportion of ingredient quantity fulfilled by the inventory for each ingredient that the recipe requires. For example, if one of the ingredients required by a recipe is **4 eggs** and the inventory contains **2 eggs**, the proportion fulfilled for this particular ingredient is **0.5**. This is done for all the ingredients in the recipe. If the units of an ingredient in the recipe does not match that of the same ingredient in the inventory, the proportion will be set at **0.5** by default. An example is when the recipe requires **1 cup flour** and the inventory contains **200 g flour**.

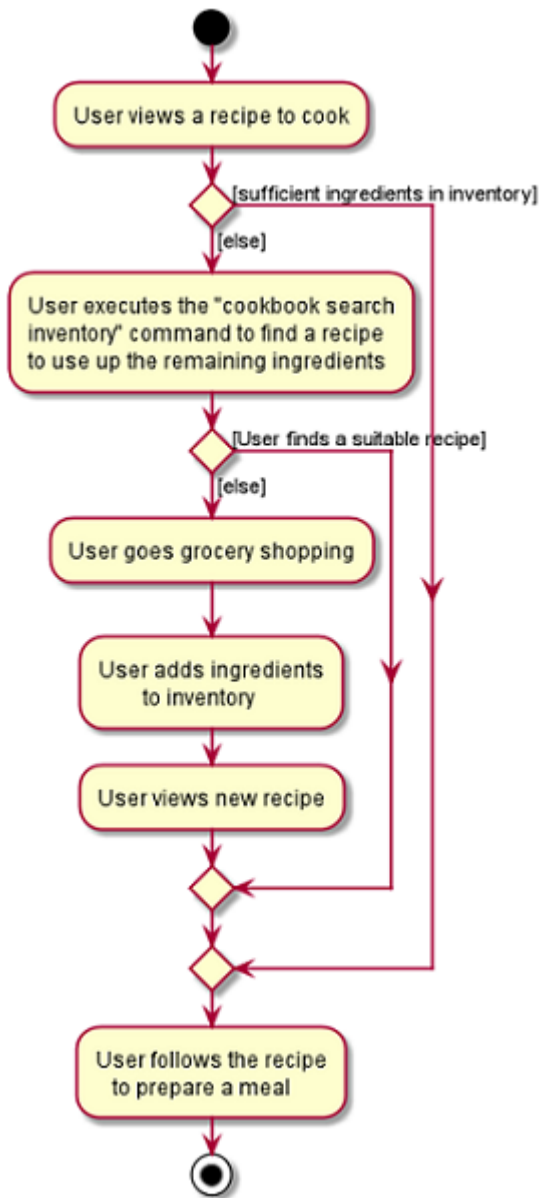
When the proportion fulfilled has been calculated for each ingredient, the values for each ingredient are summed up and divided by the number of ingredients to obtain the average. In the case where the recipe does not have any ingredients added to it yet, the `calculateSimilarity()` method will return **0**, indicating no similarity to the inventory ingredients. This is because it is likely that recipes with no ingredients have just been added by the user, and the ingredients have not been added yet. If the user is using this feature to search for a recipe to cook, they would probably not be interested in seeing a recipe that they have not added ingredients for yet. This is implemented via a guard clause as shown in the following code snippet:

```

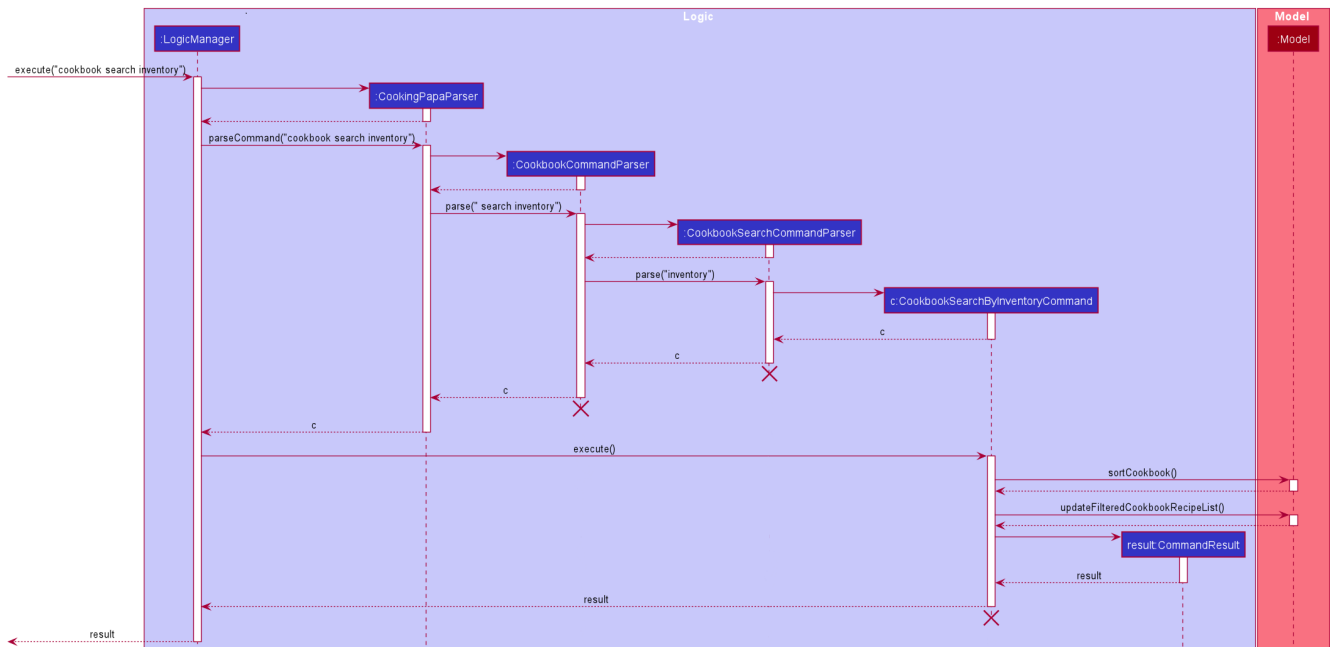
if (recipe.getIngredients().size() == NO_INGREDIENTS) {
    return ZERO_SIMILARITY;
}

```

The following activity digram shows a possible flow of events for a user using this feature:



The following sequence diagram summarizes how objects interact when a user executes the command, with more focus on how the command is parsed in the `Logic` component:



3.3.3. Design considerations

Aspect 1: Weighting of each ingredient

	Design A (Current choice): Every ingredient is weighted equally	Design B: More important ingredients are given a larger weighting
Description	The similarity of a recipe's ingredients to an inventory's ingredients is calculated by taking the mean of the proportions calculated for each ingredient, with equal weighting given to all ingredients.	The similarity of a recipe's ingredients to an inventory's ingredients is calculated by taking the weighted mean of the proportions calculated for each ingredient, with larger weightings given to more important ingredients.
Pros	Gives a good rough estimate of the proportion of ingredient requirements fulfilled for a recipe, and straightforward to implement.	May give a better gauge of the proportion of ingredient requirements fulfilled for a recipe, by accounting for the importance of the ingredient. For example, beef would be an important ingredient for a steak recipe, but garnishes might be considered less important as they can be substituted more easily.
Cons	Does not account for the importance of the ingredient in the recipe	Difficult to judge the importance of the ingredient, and complicated to implement categorisation of the types ingredients and their relative importance.

Design A was chosen as it provided a fair estimate of the similarity between the recipe and inventory ingredients, with a simple implementation. The cons for Design B were deemed to outweigh the pros, especially since the importance of an ingredient in a recipe could be rather subjective.

Aspect 2: Handling ingredients with different units

	Design A (Current choice): Use a default similarity value of 0.5	Design B: Convert the units
Description	The similarity value of an ingredient with different units in the recipe and the inventory is treated as 0.5.	The similarity value of an ingredient with different units in the recipe and the inventory is calculated by converting the units, such that the proportion of the recipe ingredient in the inventory can be determined.
Pros	Simple to implement.	Able to calculate the proportion of the recipe ingredient fulfilled by the inventory, even when dealing with different units.
Cons	Unable to calculate the proportion of the recipe ingredient fulfilled by the inventory when dealing with different units, and can only give a fixed default value of 0.5.	More complicated to implement as it requires CookingPapa to recognise the units in both the recipe and inventory and be able to convert between them. Some units such as cup may also not have a standard conversion factor.

Design A was chosen due to time constraints, as handling the conversion between different units would take time away from developing other parts of the application. Given more time, Design B will be implemented to handle conversion for standard units, such as between **g** and **kg**, but Design A would still have to be used for units with non-standard conversion factors.

3.4. Adding ingredients to inventory and cart

The inventory and cart acts as storage for **Ingredient** classes. They are facilitated by **InventoryCommand** and **CartCommand** respectively, which extends the **Command** abstract class. Since **CartAddCommand** and **InventoryAddCommand** both serve the same purpose in different contexts of **Cart** and **Inventory** respectively, they will be mentioned together in tandem.

This command was implemented to allow the user know to add an ingredient to the cart or inventory respectively. An ingredient only has two main components - its name and quantity. We allow the user to use their own measurement up to their own preferences and do not force any fixed unit of measurement. Although similar, **Cart** and **Ingredients** differ in certain functions from a user's point of view. For a user to immediately sort where they wish to sort the ingredient they are adding, **Cart** and

3.4.1. Implementation

Below is a step-by-step sequence of what happens when the command **cart add ingredient i/INGREDIENT_NAME q/INGREDIENT_QUANTITY** is added.

1. The user adds a ingredient to the cart by entering the command **cart add ingredient i/INGREDIENT_NAME q/INGREDIENT_QUANTITY** in the command line input.

2. `CartAddCommandParser` parses the input to check and verify that the input provided for `i/INGREDIENT_NAME` and `q/INGREDIENT_QUANTITY` are correct. Otherwise a `ParseException` will be thrown.
3. The fields are then passed to `CartAddIngredientCommand` as an `Ingredient` object and is returned to `LogicManager`.
4. `LogicManager` calls `CartAddIngredientCommand#execute()` and checks if the `Ingredient` object given has the same `INGREDIENT_NAME` and `INGREDIENT_QUANTITY` unit. If that `Ingredient` exists, it will simply add on to the quantity of that ingredient. Otherwise, a new instance of that `Ingredient` will be added to the Cart.
5. If `CommandException` is not thrown, `Model#addCartIngredient` will be executed, with the given `Ingredient` as the parameter
6. `Model#addCartIngredient` then executes, adding the `Ingredient` to the local cart storage and updates with `Model#updateFilteredCartIngredientList()`.
7. A `CommandResult` with the successful text message is returned to `LogicManager` and will be displayed to the user via the GUI to feedback to the user that the `Ingredient` has been successfully added.

The above implementation is the same for `Inventory` with the command `inventory add ingredient i/INGREDIENT_NAME q/INGREDIENT_QUANTITY`

The following sequence diagram shows how the function of adding ingredients to cart work (full command omitted for brevity):

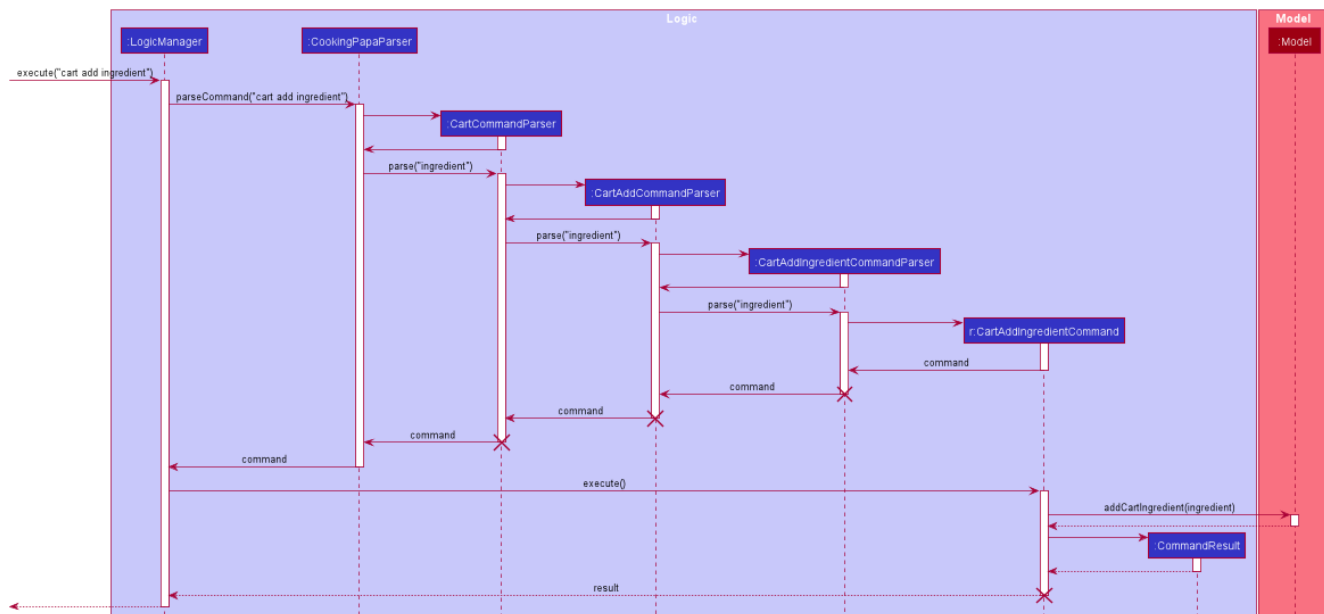


Figure 11. Sequence diagram for `CartAddIngredientCommand`

3.4.2. Design Considerations

Aspect: The need for many parsers for this command

Table 3. Design considerations for the need for many parsers for this command

	Design A (Current choice): Create parsers for every individual action	Design B: Create parsers for each specific action
Description	The command will go through the parsers in the following order: <code>CookingPapaParser</code> → <code>CartCommandParser</code> → <code>CartAddCommandParser</code> → <code>CartAddIngredientParser</code> before finally returning <code>CartAddIngredientCommand</code> . We eventually went with this as we wanted the add functionality to be expanded, namely to be able to add all the ingredients of cookbook recipes into the cart.	<code>CartAddCommand</code> will not be created to facilitate <code>CartAddIngredientCommand</code> and <code>CartAddRecipeIngredientCommand</code> .
Pros	More organised and can do more with <code>cart add</code> as the prefix.	The classes can be more specific and atomic in their actions.
Cons	Many parser classes to make and keep track of.	Might lead to disorganisation during troubleshooting with so many classes to keep track.

3.5. Remove ingredients of a recipe from the inventory

3.5.1. Implementation

The mechanism is facilitated by `InventoryCookCommand`, which extends the `Command` abstract class. The format of the command is as follows: `inventory cook recipe INDEX`.

This command was implemented to allow users to remove multiple ingredients and their quantities found in a recipe from their inventory. If the inventory contains an ingredient that has a higher quantity than specified in the selected recipe, its quantity will be subtracted accordingly. If the ingredient has a lower quantity than specified in the selected recipe or if there is a missing ingredient in the inventory, the feature will not be executed and an error will be thrown. Without this command, users can only remove ingredients through the `inventory remove ingredient` command one at a time. Moreover, they have to constantly cross-check the ingredient quantities in the recipe for accuracy. Therefore, this command provides convenience after users have prepared a recipe and wish to update their inventory ingredients through a single step.

Below is a step-by-step sequence of what happens when a user enters this command:

1. The user enters an inventory cook command `inventory cook recipe INDEX` using the command line input.
2. `InventoryCookCommandParser` parses the input to check and verify the input provided by the user. If the input provided is invalid, a `ParseException` will be thrown.
3. The valid index is then passed to `InventoryCookCommand` as an `Index` object.
4. `LogicManager` calls `InventoryCookCommand#execute()` and checks if the `Index` provided is within bounds and if the specified `Recipe` contains ingredients. Otherwise, a `CommandException` is

thrown.

- Subsequently, two checks are performed to check if the inventory contains all of the ingredients specified and whether those quantities are sufficient to be subtracted.
- If all the checks passed, `model#removeInventoryIngredient` is called through a `stream()` to remove the ingredients of a selected recipe from the inventory.

```
selectedRecipe.getIngredients().stream().forEach(model::removeInventoryIngredient);
```

- A `CommandResult` with a success message is returned to `LogicManager` and passed back to `MainWindow` which displays the text to the user through the GUI.

The following sequence diagram shows how the command `inventory cook recipe 1` works:

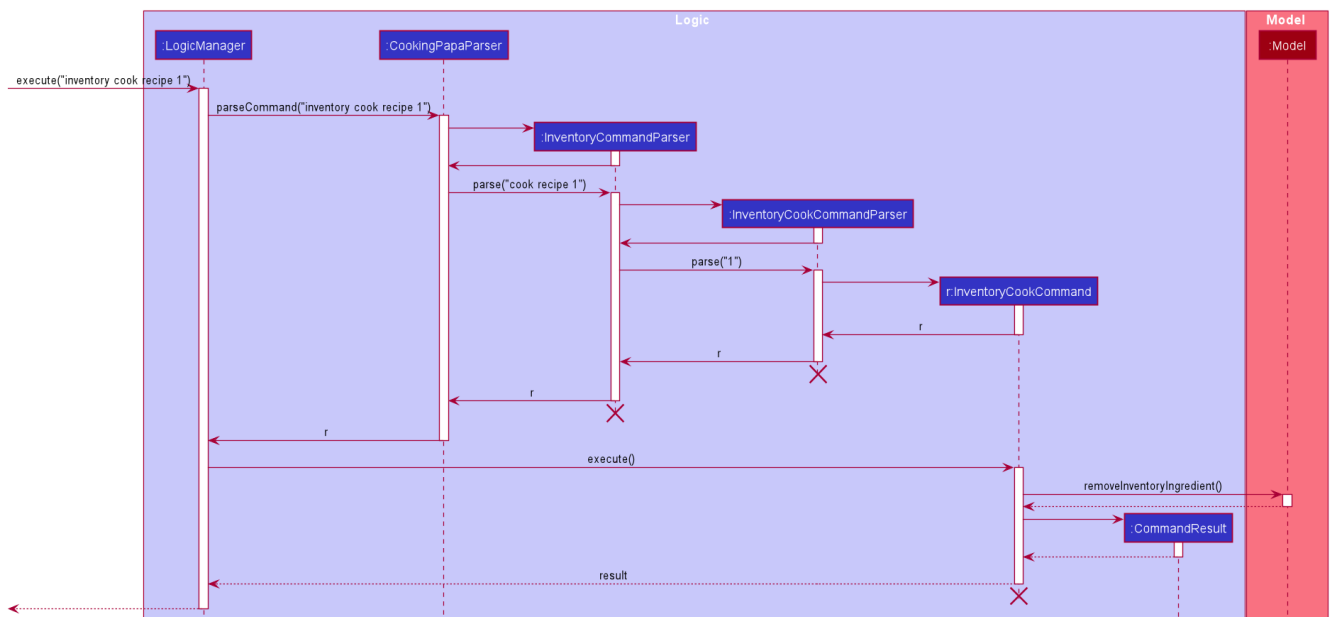


Figure 12. Sequence Diagram for `InventoryCookCommand`

3.5.2. Design considerations

Aspect: Allowing users to execute the `inventory cook recipe` command when there are missing or insufficient ingredients in the inventory.

Table 4. Design considerations for the `inventory cook recipe` command

	Design A: Allow the execution of <code>inventory cook recipe</code> command regardless of missing or insufficient ingredients in the inventory	Design B (Current choice): Do not allow execution of <code>inventory cook recipe</code> command when there are missing or insufficient ingredients in the inventory
Description	Allow the users to execute the command regardless of missing or insufficient ingredients in the inventory. Missing ingredients will be ignored and ingredients with insufficient quantities will be entirely removed.	When there are missing or insufficient ingredients in the inventory, the execution of the command will throw an error to warn users whether they have missing ingredients or insufficient ingredients in their inventory.

	Design A: Allow the execution of <code>inventory cook recipe</code> command regardless of missing or insufficient ingredients in the inventory	Design B (Current choice): Do not allow execution of <code>inventory cook recipe</code> command when there are missing or insufficient ingredients in the inventory
Pros	Straightforward for users to use the command as they do not have to check whether they have all the ingredients in sufficient quantities.	Enhances user experience. The application can notify users that they have missing or insufficient ingredients when they attempt to prepare a recipe through this command.
Cons	Reduces code readability as more methods and steps are needed to check and isolate a list of missing and insufficient ingredients. This list of ingredients are also to be treated differently from the other ingredients when removing from the inventory.	A potential hassle for users as they have to ensure that all ingredients are present and are sufficient in their inventory to use the command.

3.6. Moving ingredients from cart to inventory

The user may use this command after their shopping trip. With this one command, all ingredients will be shifted from the cart to the inventory.

This command is implemented to ease the process of having the user adding every single ingredient to their inventory after they have bought ingredients from their cart and eventually deleting the cart after that tedious process. These gives a convenience to users that frequently use our application and we foresee that such an action will be used very often by these users. As this command only performs an atomic action, no extra arguments are needed to further supplement the use of this command.

3.6.1. Implementation

This command is facilitated by `CartMoveCommand`, which extends the `Command` class. The format of the command is as follows: `cart move`.

Below is a step by step sequence of what happens when the user executes this command.

1. The user enters the command `cart move` in to the command line input.
2. `CartMoveCommandParser` then ensures that the user does not enter any other commands after `cart clear`.
3. `CartMoveCommandParser` then returns a `CartMoveCommand` and returns it to `LogicManager`
4. `LogicManager` calls `CartMoveCommand#execute()`. If there are other commands after `cart clear`, a `CommandException` will be thrown.
5. If `CommandException` is not thrown, `Model#cartMoveIngredients()` will be executed.
6. `Model#cartMoveIngredients()` will move every ingredient from the `cart` and add it into the `inventory`

7. A **CommandResult** with the success message text will be returned to **LogicManager**, which will then be passed to **MainWindow** and will then feedback to the user.

The following sequence diagram shows how this function works (full command omitted for brevity):

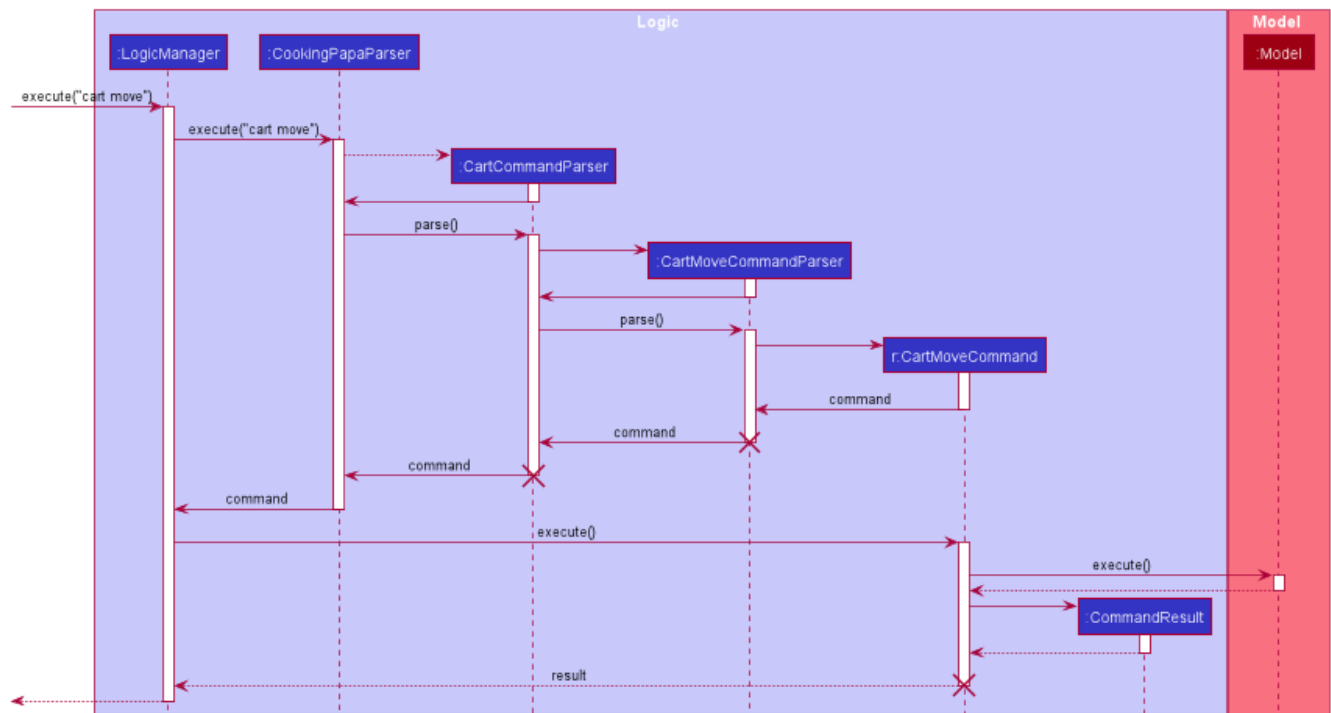


Figure 13. Sequence Diagram for CartMoveCommand

3.6.2. Design considerations

Aspect: Allowing users to move some or all ingredients from cart to inventory

Table 5. Design considerations for allowing users to move some or all ingredients from cart to inventory

	Design A (Current choice): Move all ingredients	Design B: Allow users to move individually or exclude some ingredients when moving
Description	There was a consideration to allow the user to move the ingredients by individual ingredients. Eventually the options was not given as we know that typical users will want to move all the ingredients except for individual ingredients.	The use cases of such an action happening is very little and the user can simply manually remove the few ingredients they do not wish to add to the inventory after using the cart move command. The user can also manually add back the ingredients to the cart after it is cleared if they wish to.
Pros	Straightforward to implement	Lesser implementations, more time to focus on other parts of the project
Cons	Lesser functionality to users that really want to only move certain ingredients	Poorer user experience for users that do not want to move all ingredients from the cart to inventory on a regular basis,

3.7. Add a Recipe's Ingredients to Cart

The user may want to buy the required ingredients to cook a certain recipe in the cookbook. This feature allows the user to add a certain recipe's required ingredients into the cart.

3.7.1. Implementation

The action of adding a recipe's ingredients to cart mechanism is facilitated by `CartAddRecipeIngredientCommand`, which extends the `CartAddCommand` abstract class. The format is as follows: `cart add recipe INDEX`.

This command is implemented to ease the tedious process of having the user adding every single ingredient to their cart when they want to purchase ingredients to cook a certain recipe. This provides convenience to users that frequently use our application and such process like shopping for a certain recipe's ingredient is intuitive to users. Furthermore, this command creates interaction between the `Cookbook` and `Cart` which helps to further integrate the application as an all-in-one application.

Below is a step by step sequence of what happens when a user enters this command:

1. The user enters the command `cart add recipe INDEX` in the command line input.
2. `CartAddRecipeIngredientParser` parses the user input and checks if the index provided is an integer. Note that the parser will throw a `ParseException` if the given index is not an integer.

```
try {
    recipeIndex = ParserUtil.parseIndex(argMultimap.getPreamble());
} catch (ParseException pe) {
    throw new ParseException(String.format(MESSAGE_INVALID_RECIPE_DISPLAYED_INDEX,
    CartAddCommand.MESSAGE_USAGE), pe);
}
```

3. The index is passed as a parameter for `CartAddRecipeIngredientCommand` which is returned to `LogicManager`.
4. `LogicManager` calls `CartAddRecipeIngredientCommand#execute()` which checks if the given index is a valid index of a recipe. Note that the command will throw a `CommandException` if the given index is not valid.

```
if (recipeIndex.getZeroBased() >= model.getCookbook().getRecipeList().size()) {
    throw new CommandException(String.format(
    MESSAGE_INVALID_RECIPE_DISPLAYED_INDEX, MESSAGE_USAGE));
}
```

5. If the index is valid, the selected recipe's ingredients will be added accordingly. This is done through calling `Model#addCartIngredient()`, with each ingredient as the parameter.
6. `Model#addCartIngredient` calls `Cart#addIngredient()` which then adds the ingredient to the cart. If a certain ingredient exists in the cart, adding a ingredient to a cart will increase the quantity

instead. Otherwise, a new instance of that ingredient will be added to the cart. After adding an ingredient, the cart will be updated with `Model#updateFilteredCartItemIngredientList()`.

7. A `CommandResult` with the successful text message is returned to `LogicManager` and will be displayed to the user via the GUI to feedback to the user that the selected recipe's ingredients has been successfully added to the cart.

The following sequence diagram shows how the function of adding recipe's ingredients to cart works:

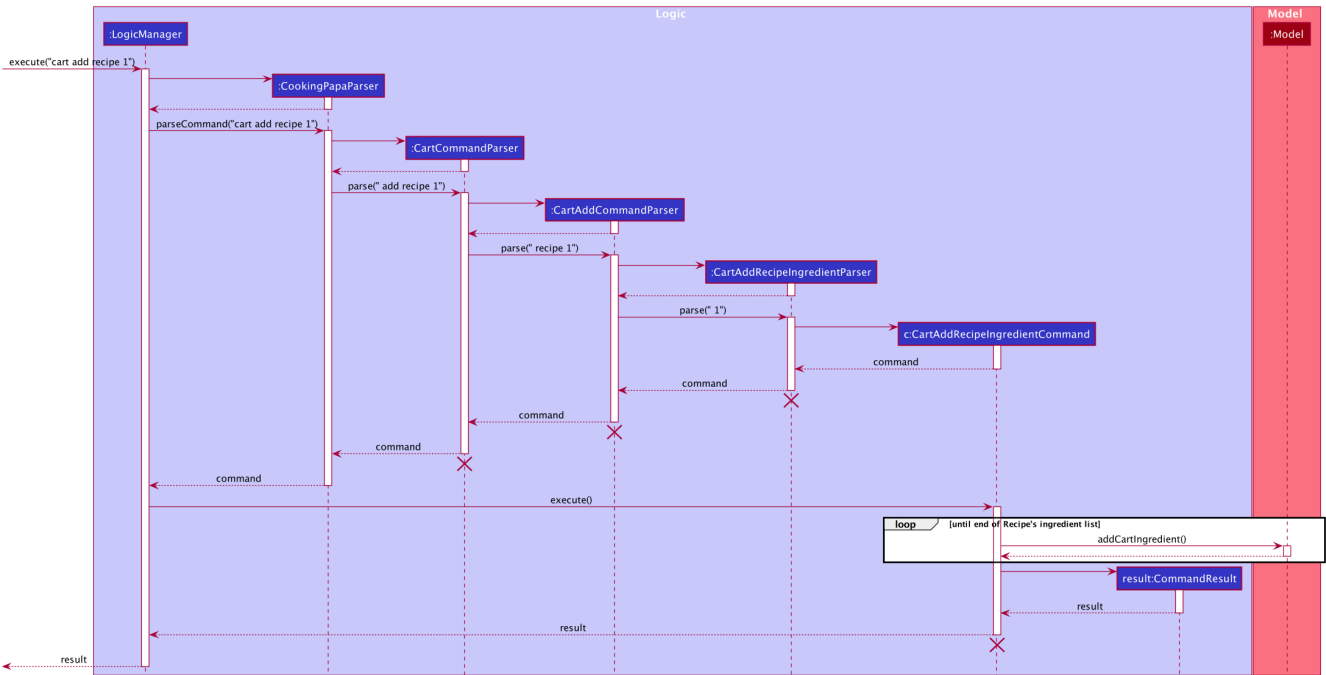


Figure 14. Sequence diagram for `CartAddRecipeIngredientCommand`

3.7.2. Design considerations

Aspect: Allowing users to add all or some recipe's ingredients

Table 6. Design considerations for allowing users to add only recipe's ingredients that are not present in the inventory to the cart

	Design A (current choice): Adding all recipe's ingredients to the cart	Design B: Adding only recipe's ingredients that are missing in the inventory to the cart
Description	Allows user to add a recipe's ingredients to the cart for shopping. This design is currently chosen due to ease of implementation and it works for all situations.	Allows user to add a recipe's ingredients base on the inventory status. However, there are some situations where this design not does work. One example would be like planning to cook at outside where the inventory status is unknown.

Pros	<ul style="list-style-type: none"> • Easier to implement as it does not need to check if the ingredients are already present in the inventory • Works for all situations as it ensures that the user is able to cook this recipe after buying the ingredients in the cart 	<ul style="list-style-type: none"> • Provide a more intuitive experience of the application as user only need to buy ingredients that are missing in the inventory
Cons	<ul style="list-style-type: none"> • Less flexible as users have to manually remove some of the recipe's ingredients that they do not want to buy 	<ul style="list-style-type: none"> • Harder to implement as additional checking is required to filter a recipe's ingredients that are missing in the inventory • Users have to manually add some of the recipe's ingredients that they want to buy although it is present in the inventory

3.8. Export ingredients in cart to PDF file

The user may use this command to export the ingredients in their cart to a [PDF](#) file, which they can then use as their shopping list at the supermarkets.

3.8.1. Implementation

The cart exporting mechanism is facilitated by `CookbookExportCommand`, which extends the `Command` abstract class. The format is as follows: `cart export`.

This command was implemented to bridge the (current, v1.4) inadequacy of Cooking Papa, which is that it is not portable (yet). It was still not convenient *enough* to be able to organize cart ingredients. Eventually, users had to go outside to the supermarket, and Cooking Papa is a desktop-only application. By allowing users to export the ingredients in their cart to a PDF file, they can then print it out, or transfer it to their mobile devices, and bring them along as shopping lists. Additionally, the layout and content of the generated PDF file is simple, informational, and easy for users to extend, allowing them to add (handwritten or annotated) remarks.

Below is a step by step sequence of what happens when a user enters this command:

1. The user enters a cart export command using the command line input `cart export`.
2. `CookingPapaParser` parses the user input and checks if it is valid. If it is invalid, i.e. an unknown command category, a `ParseException` will be thrown. If the input is valid, with the command category `cart`, a new `CartCommandParser` is created.
3. `CartCommandParser` then parses `export`. If it is invalid, i.e. an unknown command word, a `ParseException` will be thrown. If the input is valid, with the command category `export`, a new `CookbookExportCommandParser` is created.
4. `CartExportCommandParser` parses the user input and checks if the argument passed to it is an empty String, as the command takes in no extra parameters.

Note that if the String is not empty, a `ParseException` will be thrown:

```
if (userInput.isEmpty()) {  
    return new CartExportCommand();  
} else {  
    throw new ParseException(String.format(MESSAGE_INVALID_COMMAND_FORMAT,  
    CartExportCommand.MESSAGE_USAGE));  
}
```

This means that `cart export ingredient` will not work.

5. `CartExportCommandParser` then returns a `CartExportCommand` to `LogicManager`.
6. `LogicManager` calls `CartExportCommand#execute()` calls the static method of `PdfExporter`, `PdfExporter#exportCart()`, which takes in the `ObservableList<Ingredient>` stored in `Cart`
7. Step 4 is executed within a try-catch block. If a previously generated pdf (saved as `cart.pdf` by default) is opened in another program, or there is an issue writing to the PDF file, a `CommandResult` with an error message will returned to `LogicManager` (skipping step 7 and 8):

```
try {  
    PdfExporter.exportCart(model.getCart().getIngredientList());  
} catch (IOException e) {  
    return new CommandResult(MESSAGE_FILE_NOT_FOUND);  
}
```

8. The ingredients in the `Cart` is passed to the static method `PdfExporter#exportCart()`, which then makes use of the library, `PDFbox`, to parse the data.
9. Within `PdfExporter`, `PdfExporter#getTextFromCart` parse the data and splits them manually, in order to wrap the text (this has to be done due to the inadequacy of `PDFbox`). The method returns a `List<String>`, where each string represents a new line on the PDF file.
10. Subsequently, `PdfExporter` checks if the number of `String`s in the list in step 7 is greater than the number of lines a single page of the PDF can accomodate. If it is, it adds a new page, and adds lines to the PDF until the limit is hit. This repeats until all the lines are added to the PDF.
11. A `CommandResult` with the text to display to the user will be returned to `LogicManager`. The `CommandResult` is then passed back to `MainWindow`, which displays it to the user on the CLI and GUI: `resultDisplay .setFeedbackToUser(commandResult.getFeedbackToUser())`. The text displayed will notify the user on whether their addition was successful.

The following sequence diagram shows how the function of exporting ingredients in the cart to a PDF file works:

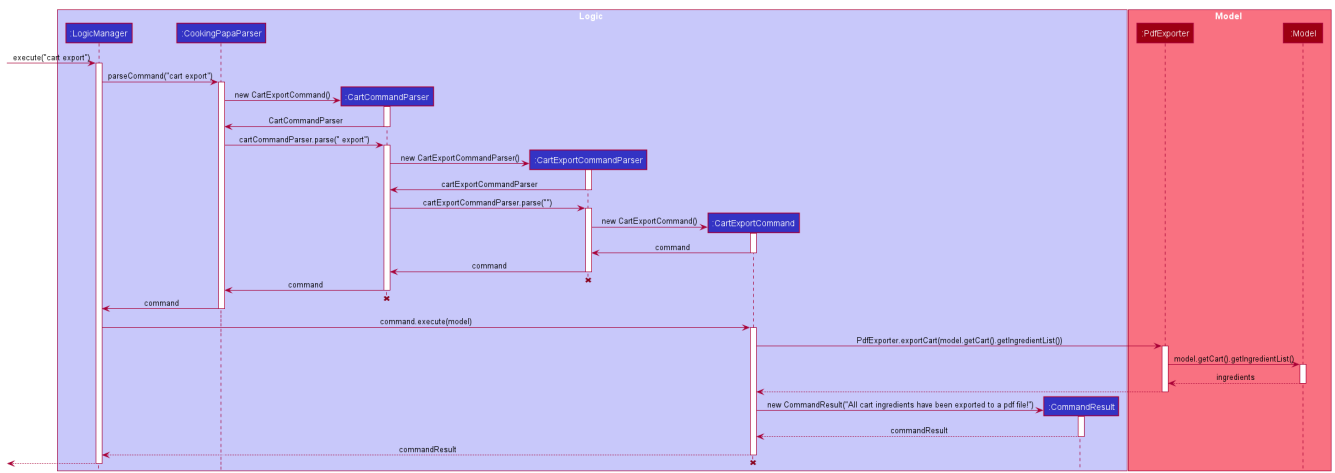


Figure 15. Sequence diagram for CartExportCommand

3.8.2. Design considerations

Aspect 1: File format to export ingredients in cart to

	Design A (current choice): .pdf	Design B: .txt
Description	Exports it to a flexible pdf file	Exports it to a txt file
Pros	<ul style="list-style-type: none"> Easier to format with Apache PDFbox's vast library and API More versatile in that images can be added if the function was to be extended to include images of the ingredients 	<ul style="list-style-type: none"> Simple to implement
Cons	<ul style="list-style-type: none"> More formatting code required May take slightly longer to export as compared to design B 	<ul style="list-style-type: none"> Lack of design/formatting flexibility

Aspect 2: What information to export

	Design A (current choice): Export the ingredient names and quantities in the cart	Design B: Export the entirety of Cooking Papa (cookbook, inventory, cart)
Description	Allow exporting of just the cart	Allow exporting of the cart, inventory, and cookbook

Pros	<ul style="list-style-type: none"> • Easier to implement as there will be less information to parse • Keeps the exported PDF short and sweet • Ingredients in the inventory and recipes in the cookbook generally correlates (and are consequential of) with the ingredients in the cart, so exporting the inventory as well as the cookbook is likely to be redundant. <div> <p>My Shopping List</p> <ul style="list-style-type: none"> ○ 10 pieces Bacon ○ 300 g Black pepper ○ 3 sticks Butter ○ 50 g Dried chili flakes ○ 10 Eggs ○ 5 cloves Garlic ○ 250 g Linguine ○ 500 g Minced beef ○ 600 ml Olive oil ○ 1 kg Parmesan ○ 500 g Salt ○ 10 Tomatoes </div> <p><i>Figure 16. A sample shopping list generated by the command</i></p>	<ul style="list-style-type: none"> • Provides users an all-in-one file containing all the information they entered into Cooking Papa • Allows user to reproduce hard-copy recipe books using Cooking Papa
Cons	<ul style="list-style-type: none"> • Certain information may be needed and not exportable by the user, i.e. recipes 	<ul style="list-style-type: none"> • Slightly more difficult to implement • May be providing users with unnecessary information

3.9. Configuration

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

4. Documentation

Refer to the guide [here](#).

5. Testing

Refer to the guide [here](#).

6. Dev Ops

Refer to the guide [here](#).

Appendix A: Product Scope

Target user profile:

- has a need to manage a significant number of recipes
- has a need to manage food resources efficiently
- prefer desktop apps over other types of apps
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: manage recipes and food resources faster than a typical mouse/GUI driven app

Appendix B: User Stories

Priorities: High (must have) - * * *, Medium (nice to have) - * *, Low (unlikely to have) - *

Priority	As a ...	I want to ...	So that ...
* * *	beginner cook	find new recipes easily	I don't waste time searching through recipes from different sources
* * *	regular cook	record my own recipes	I can refer to them easily in future
* * *	forgetful person	add ingredients for my planned meals to a grocery list easily	I know what I need to get when shopping
* * *	disorganized person	keep track of the ingredients I have at home	I can plan my meals better
* * *	busy student	cook a meal with the ingredients I already have	I don't waste time on grocery shopping
* * *	low-income individual	cook a meal with the ingredients I already have	I can save money
* * *	person with food allergies	cook meals that I am not allergic to	I do not have an allergic reaction
* *	regular cook	edit recipes	I can tweak a recipe to my liking

Priority	As a ...	I want to ...	So that ...
* *	regular cook	set a timer during meal preparation	I can control the quality of my meal
* *	CS student	cook a quick meal	I can spend more doing CS2103T
* *	vegetarian	find recipes that don't contain meat	I can keep to my diet constraints
* *	picky eater	choose recipes that only contain the food I like	I can enjoy the meals I cook
* *	working adult	plan meals for the next week	I can buy all the ingredients I need in one trip
* *	person with health issues	record the meals I eat	I can share the information with my doctor easily
*	health-conscious person	keep track of the nutritional value of the food I eat	I can meet my nutritional goals
*	regular gym-goer	keep track of my dietary intake	I can meet my fitness goals
*	obesity fighter	keep track of my calorie and fat intake	I can lose weight
*	stay-at-home parent	plan a variety of meals for the week	I can make sure that my family eats healthily
*	kiasu parent	know how much ingredients I need for 2 weeks	ensure my family never runs out of food
*	party host	scale recipe ingredients by the number of servings	I can prepare meals for large groups
*	cafe manager	keep track of the expiry dates of my ingredients	I know what ingredients I need to stock up on

Appendix C: Use Cases

(For all use cases below, the **System** is **Cooking Papa** and the **Actor** is the **user**, unless specified otherwise)

Use case: UC01 - Create a recipe

MSS:

1. User chooses to create a recipe.
2. Cooking Papa requests for details of the recipe.
3. User enters the requested details.
4. Cooking Papa creates the recipe and stores it in the cookbook, and displays the newly created recipe.

Use case ends.

Extensions:

- 3a. Cooking Papa detects an error in the entered data.
 - 3a1. Cooking Papa shows an error message.
 - 3a2. Cooking Papa requests for the correct data.
 - 3a3. User enters new data.Steps 3a1 to 3a3 are repeated until the data entered is correct.
Use case resumes from step 4.
- *a. At any time, User chooses to end the creation of a recipe.
 - *a1. Cooking Papa cancels creation of a recipe.

Use Case: UC02 - Search for recipes

MSS:

1. User chooses to search recipes.
 2. Cooking Papa requests for the tag to be searched.
 3. User enters the tag.
 4. Cooking Papa displays recipes with the corresponding tag.
- Use case ends.

Use Case: UC03 - View a recipe

MSS:

1. User chooses to view recipes.
 2. Cooking Papa requests for the index of the recipe.
 3. User enters the requested index.
 4. Cooking Papa displays the entire recipe with the corresponding index.
- Use case ends.

Extensions:

- 3a. The given index is invalid.
 - 3a1. Cooking Papa shows an error message.
 - 3a2. Cooking Papa requests for the correct index.
 - 3a3. User enters the new index.Steps 3a1-3a3 are repeated until the index entered is valid.
Use case resumes from step 4.

Use case: UC04 - Add a recipe's ingredients to the cart

MSS:

1. User chooses to add a recipe's ingredients to the cart.
2. Cooking Papa requests for the index of the recipe.
3. User enters the requested index.
4. Cooking Papa add the ingredients to the cart.

Use case ends.

Extensions:

- 3a. The given index is invalid.
 - 3a1. Cooking Papa shows an error message.
 - 3a2. Cooking Papa requests for the correct index.
 - 3a3. User enters the new index.Steps 3a1-3a3 are repeated until the index entered is valid.
- Use case resumes from step 4.

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 500 recipes 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.

{More to be added}

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

PDF

A file format for capturing and sending electronic documents in exactly the intended format.

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, and are in no way exhaustive.

Below are some test inputs for manual testing, please note that these test inputs are **only** valid for the sample cookbook, cart, and inventory data, i.e. the data that is present when Cooking Papa is opened for the first time. If the data has been modified prior to using these commands, please delete the `.json` files in `/data` (`cookbook.json`, `inventory.json`, `cart.json`).

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 cookbook, inventory, and cart.

G.2. Adding a recipe to the cookbook

Please note that these cases are to be tested individually, i.e. should test case a be executed, executing test case e will not be valid as there is already an existing recipe with the recipe name "Name". In such cases, please remove the existing recipe in the cookbook using `cookbook remove recipe INDEX`.

- a. Prerequisites: List all recipes in the cookbook using the `cookbook list` command, and **using the sample cookbook**.
- b. Test case: `cookbook add recipe n/Name d/Description i/Ingredient q/1 s/Step 1 t/Tag`
Expected: a new recipe is added to the cookbook, and displayed as the index 3 (one-based index) in the cookbook panel.
- c. Test case: `cookbook add recipe n/Name d/Description i/Ingredient q/1 s/Step 1 t/Tag` (a duplicate recipe)
Expected: no recipe will be added, and an error message indicating that there is already an existing recipe with the same name in the cookbook will be displayed.
- d. Test case: `cookbook add recipe n/Name d/Description i/Ingredient q/1 s/Step 1 s/Step 1 t/Tag` (a recipe with duplicated steps)
Expected: no recipe will be added, and an error message indicating that there is a duplicate step

in the command will be displayed.

- e. Test case: `cookbook add recipe n/Name d/Description i/Ingredient q/1 i/Ingredient q/1 s/Step 1t/Tag` (a recipe with duplicated ingredients)

Expected: a new recipe is added to the cookbook, with the duplicate ingredients being added to one another. The new recipe will be displayed as the index 3 (one-based index) in the cookbook panel.

G.3. Removing a recipe from the cookbook

Please note that these test cases are to be tested individually, i.e. should test case a be executed, executing test case a again will remove a different recipe from the cookbook. In this case, after executing test case a once, to execute it again, please add back the removed recipe using `cookbook add recipe ...` or by deleting the `.json` files.

- a. Prerequisites: List all recipes in the cookbook using the `cookbook list` command, and **using the sample cookbook**.
- b. Test case: `cookbook remove recipe 1`
Expected: a recipe (Aglio Olio) will be removed from the cookbook.
- c. Test case: `cookbook remove recipe 0` and `cookbook remove recipe 5`
Expected: since the indices in the recipe panel are one-based, i.e. starting from 1, the former command is out-of-bounds; the latter command is out-of-bounds because there are only 4 recipes in the cookbook. Both commands will show an error message reflecting the invalid recipe indices provided.

G.4. Searching for recipes by tags

Please note for this search command, with more tags being included, the number of results returned will be greater, i.e. if there are three tags included, the recipes returned do not have to be tagged with all three tags.

- a. Prerequisites: List all recipes in the cookbook using the `cookbook list` command, and **using the sample cookbook**.
- b. Test case: `cookbook search tag t/Simple`
Expected: the recipe panel will be updated to show only two recipes, both which are tagged with "Simple".
- c. Test case: `'cookbook search tag t/Simple t/Celebrity'` Expected: the recipe panel will be updated to show only three recipes, of these three recipes, they are either tagged with "Simple" or "Celebrity".

G.5. Exporting the cart to a PDF file

Please note that for the export command, the result is based on the sample cart. a. Prerequisite: have the sample cart data in `cart.json`, if the file has been modified, please exit Cooking Papa, and delete it in `/data`, and run Cooking Papa again.

- a. Test case: `cart export` Expected: a PDF file will be created in the same folder as Cooking Papa,

and the content should look like:

My Shopping List

- ☐ 10 pieces Bacon
- ☐ 300 g Black pepper
- ☐ 3 sticks Butter
- ☐ 50 g Dried chili flakes
- ☐ 10 Eggs
- ☐ 5 cloves Garlic
- ☐ 250 g Linguine
- ☐ 500 g Minced beef
- ☐ 600 ml Olive oil
- ☐ 1 kg Parmesan
- ☐ 500 g Salt
- ☐ 10 Tomatoes

Figure 17. Content of `cart.pdf` created from sample cart data

- b. Test case: `cart export`, with a previously created `cart.pdf` open in a program Expected: an error will be thrown, as `PdfExporter` is unable to modify a file that is currently open in another program. Closing the file and executing the command will return the same result (assuming the cart data is the same as the sample cart data) as test case a.

Appendix H: Effort

Achievements/ challenges	Effort required	Difficulty level (out of ***)
Greater number of entities than AB3	As AB3 only had one overarching entity (<code>Person</code>), it was a challenge to extract the implementation for <code>Person</code> and apply it to three overarching entities (<code>Cart</code> , <code>Cookbook</code> , <code>Inventory</code>). Much time was spent refactoring to our needs, but was not too tough given the great documentation and clarity in AB3's code.	**

Development of the GUI	<p>As the team had not much experience with regards to CSS and JavaFX, it took awhile to get rolling and adapt the aesthetics to Cooking Papa's needs. Moreover, one challenge faced was ensuring that the GUI ran as expected on Windows, MacOS, and Linux.</p> <p>Additionally, the use of SceneBuilder was encouraged, however, it led to many unintended changes and extra variables which made troubleshooting a lot more complex (especially to a novice).</p>	**
Integrating cookbook view command with a button on the GUI	<p>We wanted to make the command more of a toggle instead of something users had to type, as it was not intuitive. While implementing the button was rather trivial, one requirement of the app was that it had to be testable via the command line. Connecting the command from the command line (Logic) to the UI was a big challenge, especially while trying to maintain the abstraction between the two.</p> <p>In hindsight, perhaps greater experience with GUIs would have made this process easier, but our team were all novices in that aspect, and being able to pull this off, especially when we could have simply left it as the status quo, is a huge achievement.</p>	***
Refactoring cart export code	<p>As the original PDF library used (iTextPDF) has not permitted due to its license, the whole code had to be refactored to use the current PDF library (Apache PDFbox). The challenge was the lack of features in PDFbox, i.e. tables were not a feature, and had to be drawn using lines. This was a huge hinder in achieving the intended output PDF file. Eventually, it was decided to simply create a list in the PDF instead of a table due to the lack of time, and the payoff for tinkering with PDFbox was not worth the effort.</p>	**