Table of Contents

1. Introduction

1.1. Purpose

This document describes the architecture, software design and implementation decisions of BakingHome, a one-stop application for managers of home bakeries to manage their business efficiently.

1.2. Audience

This documentation is intended for the developers, designers, and software testers of BakingHome.

1.3. Using the Guide

This guide follows the following format:

`mark-up` text represents a component or a command.

NOTE	This box provides additional information.

2. Design

This section describes the architecture and components of BakingHome.

2.1. Architecture

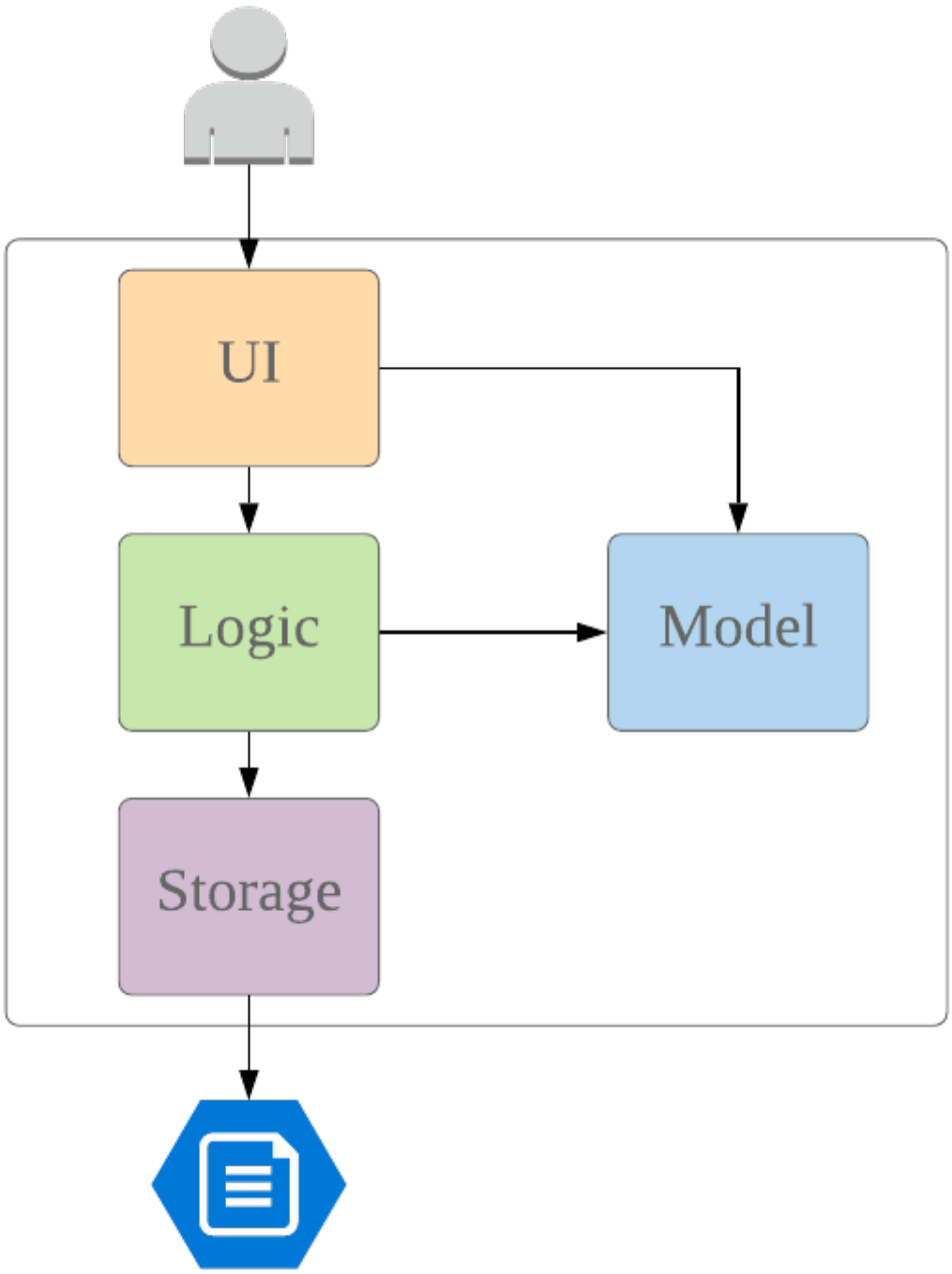

Figure 1. Architecture Diagram

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

BakingHome consists of four components.

- `UI`: The UI of the App.
- `Logic`: The command executor.
- `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 `Manager` class.

2.2. UI component

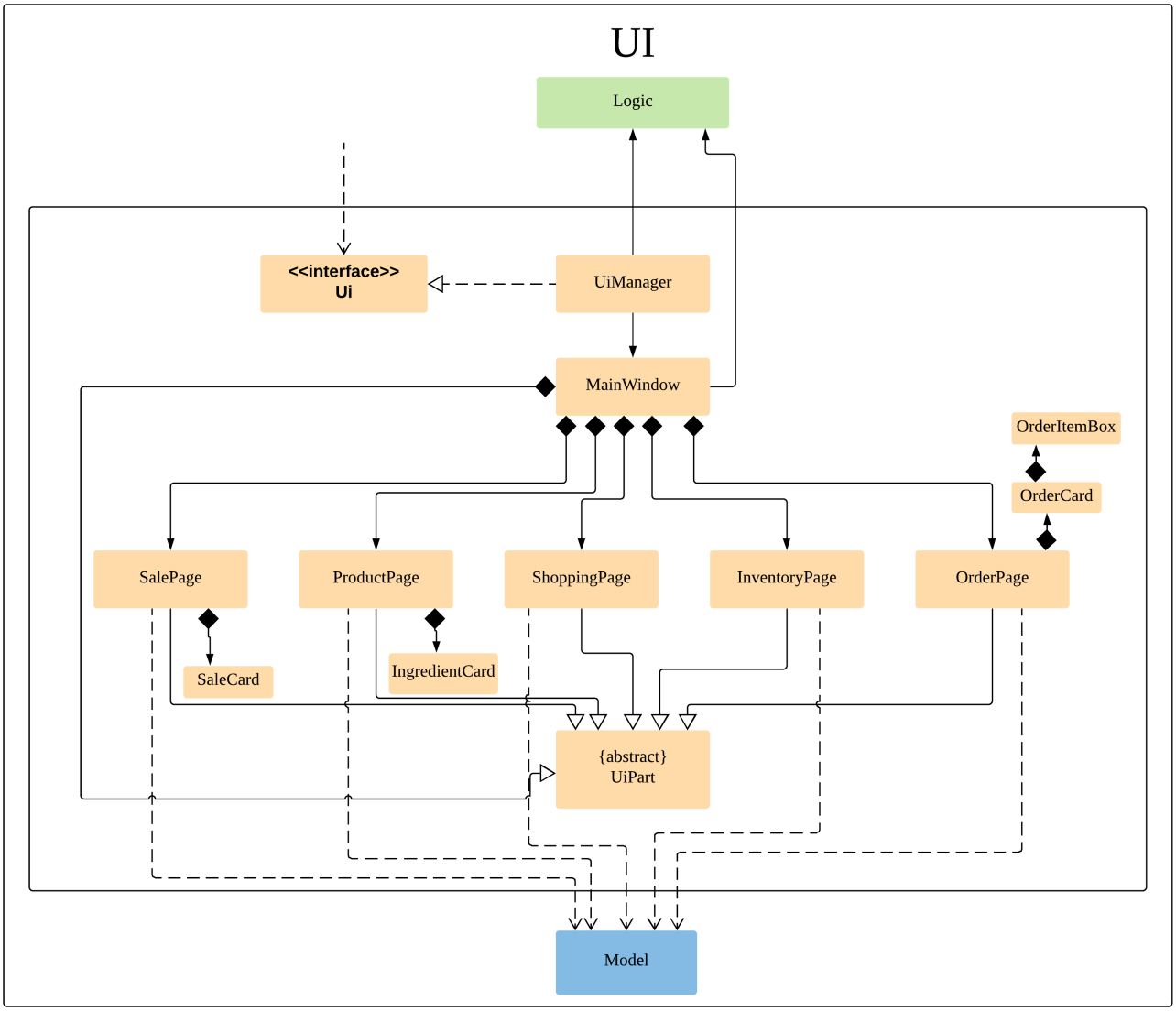

Figure 2. Structure of the UI Component

API : `Ui .java`

The UI consists of a `MainWindow` that contains 5 separate pages, namely the `SalePage`, `ProductPage`, `ShoppingPage`, `InventoryPage` and `OrderPage`. All these, including the `MainWindow`, inherit from the abstract `UiPart` class. Some pages consist of smaller components. For example, `SalePage` has a

`SaleCard`, `ProductPage` has an `IngredientCard`, and `OrderPage` has an `OrderCard` and an `OrderItemBox`.

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

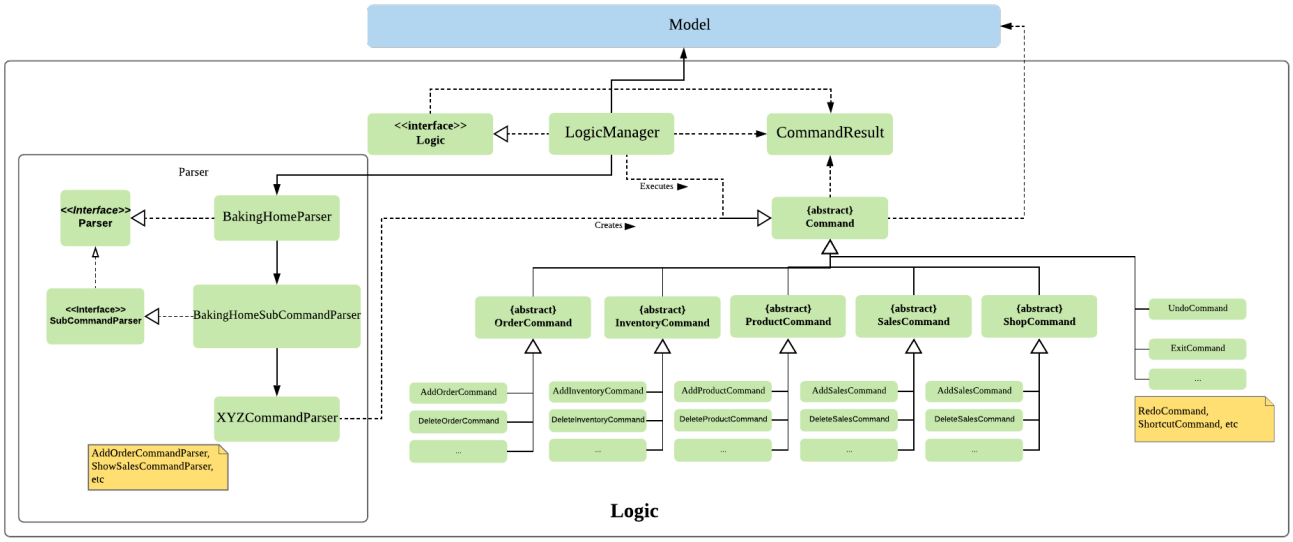

Figure 3. Structure of the Logic Component

API : `Logic.java`

1. `LogicManager` uses the `BakingHomeParser` 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` (for example, deleting an order).
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 a certain page to the user.

2.4. Model component

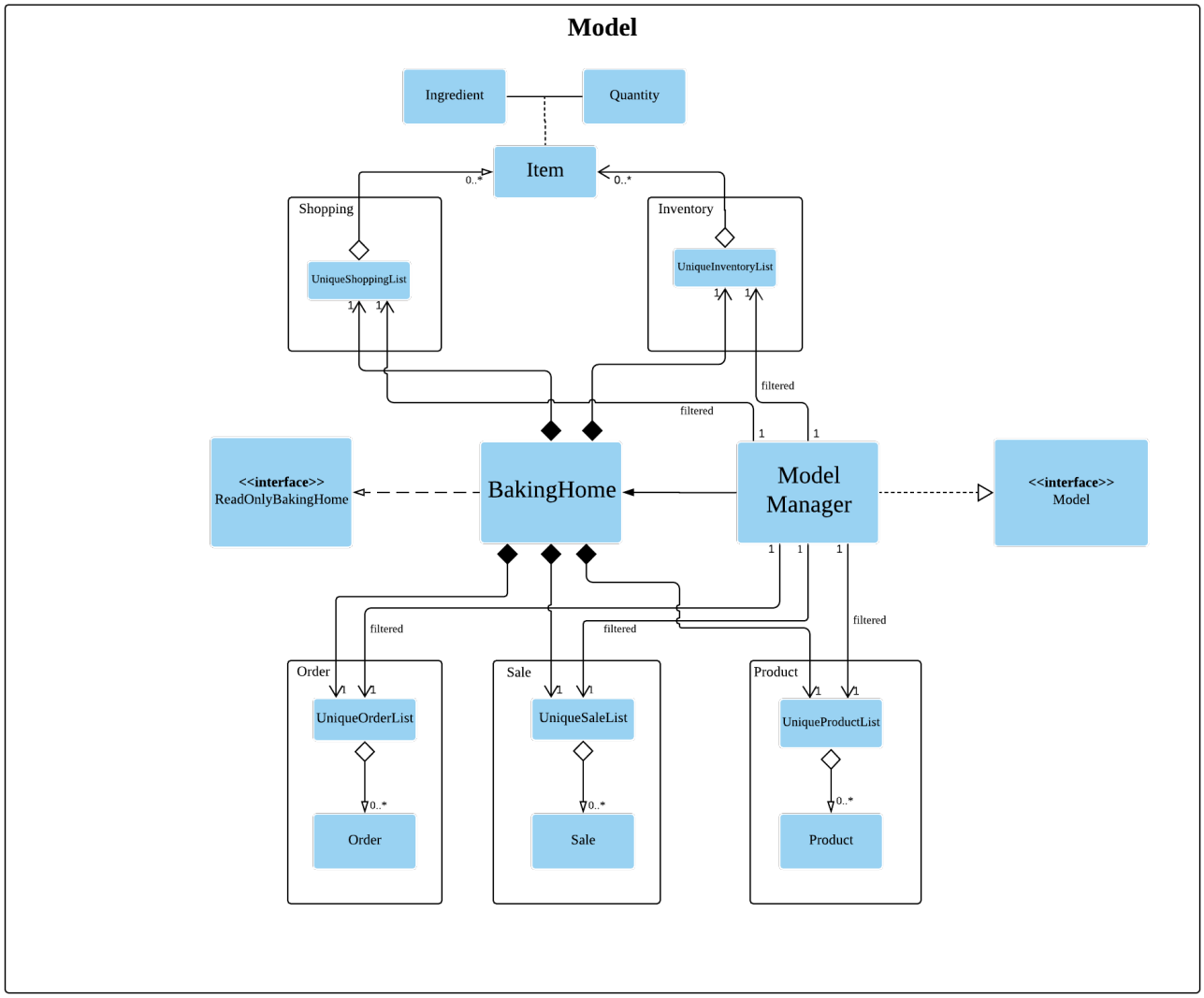

Figure 4. Structure of the Model Component

API : `Model.java`

The `Model`,

- Stores classes that are used for BakingHome (namely Ingredient, Customer, Order, IngredientInfoList, IngredientItemList, Product, ProductIngredient, Sale)
- Exposes unmodifiable `ObservableList<>` of Orders, Products, Sales, Inventory, and ShoppingList that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- Does not depend on any of the other three components.
- Stores the UniqueEntityLists used for BakingHome and its methods.

2.5. Storage component

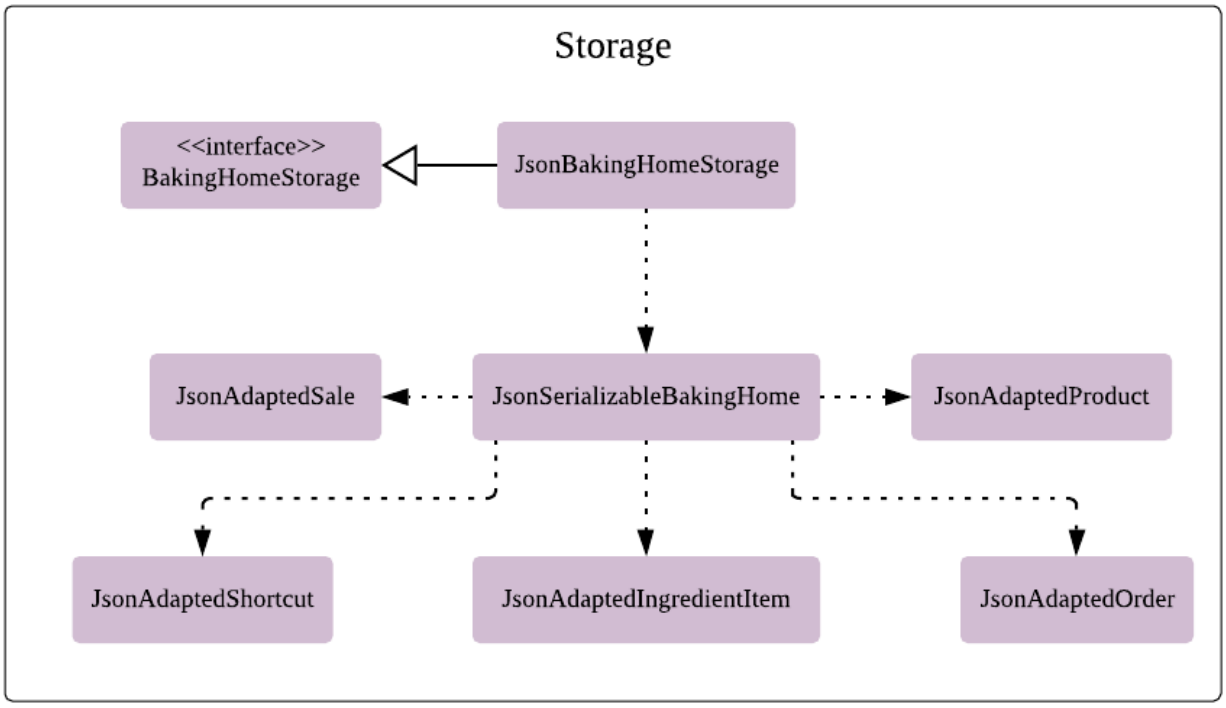

Figure 5. Structure of the Storage Component

API : `BakingHomeStorage.java`

The `Storage` component,

- Can save the Ingredient objects, Ingredient Item objects, Order objects, Product objects, Product Item objects, Sale objects, and Shortcut objects, in json format and read it back.

2.6. Common classes

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

3. Implementation

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

3.1. Auto-Complete Feature

BakingHome comes with an auto-complete feature that predicts the commands or arguments that the user attempts to type based on what has already been entered. Auto-Complete is invoked by pressing the `Tab` key. If there are multiple suggestions available, the user can navigate among the suggestions by repeatedly pressing the `Tab` key.

3.1.1. Implementation

The auto-complete mechanism is facilitated by `AutoCompleter` in `Logic` component.

`AutoCompleter` has a nested class `Input`, which represents the details of user input, including the `text` and the `caretPosition`.

`AutoCompleter` implements the following operations:

- `AutoCompleter#addCommandClass(Class<? extends Command>)` — Adds a command class for `AutoCompleter` to complete.
- `AutoCompleter#isAutoCompletable(Input)` — Returns true if the current user input can be completed by `AutoCompleter`.
- `AutoCompleter#complete()` — Returns an `Input` object that specifies the details of the user input after auto-completion.

The last two operations are exposed in the `Model` interface as `Model#isAutoCompletable(Input)()` and `Model#complete()` respectively.

Workflow

When the user presses a key in the command box, command box checks if the key pressed is `Tab`. If `Tab` is pressed, the command box checks with `AutoCompleter` to verify if the current input is auto-completable. If the state is eligible for auto-completion, the command box will request for a suggestion by calling `Model#complete()` and set its text and caret position accordingly. The workflow is illustrated in the diagram below:

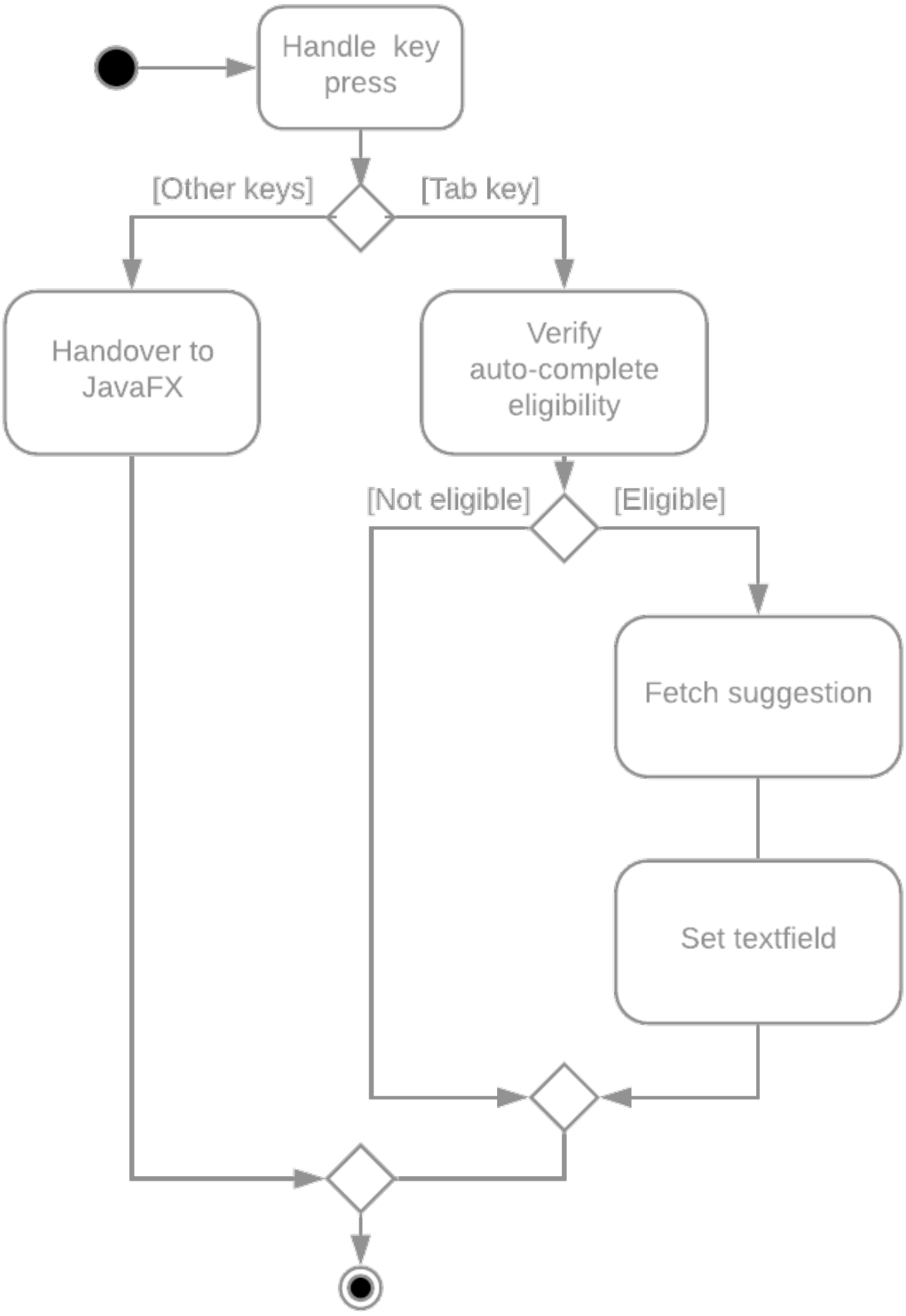

Figure 6. Workflow of AutoCompleter

Navigating among suggestions

The auto-complete feature allows users to navigate among possible suggestions by repeatedly pressing `Tab`.

Internally, `AutoCompleter` maintains a list `suggestionList` containing all possible `Input` suggestions. `suggestionList` is implemented as a cyclic list by maintaining a `suggestionPointer`:

- When `AutoCompleter#complete()` is called, the `Input` object pointed by `suggestionPointer` in `suggestionList` is returned, and `suggestionPointer` is set to `(suggestionPointer + 1) % suggestionList.size()`.
- `suggestionList` is updated if the user input no longer matches any of the suggestions in `suggestionList`.

Extending Auto-Complete to More Commands

Following the Open-Closed Principle, the Auto-Complete feature is highly extensible.

You can add a command to support auto-completion by taking the following steps:

Step 1. Declare `AUTO_COMPLETE_INDICATOR` and `AUTO_COMPLETE_PARAMETERS` fields in your command class.

NOTE	`AUTO_COMPLETE_INDICATOR` is a string specifying when the arguments should be completed. Auto-complete only completes the arguments of a command when the text in command box with that command's `AUTO_COMPLETE_INDICATOR`. `AUTO_COMPLETE_PARAMETERS` is an array of `Prefix` that you want to auto-complete.

An example is shown below:

```
public class AddOrderCommand extends Command {
    public static final String AUTO_COMPLETE_INDICATOR = "order add"; // This tells
AutoCompleter that  if  user input begins with "order add", it should be recognized as
an AddOrderCommand by AutoCompleter.
    public static final Prefix[] AUTO_COMPLETE_PARAMETERS = new Prefix("by"), new
Prefix("name"); //This tells AutoCompleter that  AddOrderCommand has these parameters.
}
```

Step 2. Add the command to `AutoCompleter` by calling `AutoCompleter#addCommandClass(Class<? extends Command>)`

3.1.2. Design considerations

Aspect 1: Extending AutoComplete to more commands

- Alternative 1: Hard-code command words and arguments in AutoCompleter class.
  - Pros: Easy to implement.
  - Cons: Violates the Open-Closed Principle because developers need to modify

AutoCompleter's internal structure to add new commands. Also, it makes code more coupled since if we change parameters of a command, we need to change corresponding fields in AutoCompleter as well.

- Alternative 2 (Current choice): Use Reflection API to obtain command words and arguments from CommandClass at runtime.
  - Pros: Avoids modification to the internal structure of AutoCompleter class
  - Cons: Since Reflection allows code to perform operations that would be illegal in non-reflective code, it could lead to unexpected side-effects if implemented wrongly.

Aspect 2: Displaying multiple suggestions

- Alternative 1: Use a drop-down list to display all possible suggestions.
  - Pros: Intuitive and allows users to see all possible commands in one place.
  - Cons: Hard to implement. May require additional components other than JavaFx's built-in components.
- Alternative 2 (Current choice): Navigate between possible suggestions by repeatedly pressing `Tab` key.
  - Pros: Easier to implement since no additional components are needed
  - Cons: Cannot display all possible commands in one place.

3.2. Filter Product Feature

3.2.1. Implementation

BakingHome's products can have two status: `ACTIVE` or `ARCHIVE`. This feature allows user to view products with a given certain status, i.e. shows only products with an ARCHIVE status.

The filter mechanism in product is facilitated by FilteredList which wraps a ObservableList and filters using the provided Predicate. A `FilteredList<Product> filteredProducts` is stored in the `ModelManager`. In `BakingHome`, there is an `ObservableList<Product> products` which contains all products, regardless of its status. `filteredProducts` in the ModelManager is initialized with this ObservableList.

Since a FilteredList needs a Predicate, which matches the elements in the source list that should be visible, the filter mechanism implements the following operation to support filtering:

- `Model#updateFilteredProductList(Predicate<Product> predicate)` — Sets the value of the property Predicate in the `filteredProducts`.
  - Predicates are declared statically in the `Model` interface, namely `PREDICATE_SHOW_ACTIVE_PRODUCTS`, `PREDICATE_SHOW_ARCHIVE_PRODUCTS`, and `PREDICATE_SHOW_ALL_PRODUCT`. In particular `PREDICATE_SHOW_ARCHIVE_PRODUCTS` is as follows

```
Predicate<Product> PREDICATE_SHOW_ARCHIVE_PRODUCTS = product -> {
    return product.getStatus() == Product.Status.ARCHIVE;
};
```

- The `ListProductCommand` will call this method to change the visibility of products with different status by passing in the corresponding predicate.

An example usage scenario and how the filter mechanism behaves at each step is shown below.

Step 1. The user launches the application for the first time. `UniqueProductList` will be initialized with a list of default products in BakingHome. This list contains a few active products and a few archived products.

Step 2. The user inputs `product filter -scope archive` to list all archived products. `UI` passes the input to `Logic`. Logic then uses a few `Parser` classes to extract layers of information out as seen from steps 3 to 5.

Step 3. Logic passes the user input to `BakingHomeParser`. `BakingHomeParser` identifies that this is a `ProductCommand` through the word "product". It then creates a `ProductCommandParser` to parse the remaining information, i.e. "filter -scope archive".

Step 4. `ProductCommandParser` identifies that this is a `FilterProductCommand` through the word "filter". It then creates a `FilterProductCommandParser` to parse the scope.

Step 5. `FilterProductCommandParser` parse "-scope archive" and get the scope. It then returns a `FilterProductCommand` with the scope information.

Step 6. `Logic` finally gets the `FilterProductCommand` and execute it. The execution firstly calls `Model#updateFilteredProductList(Predicate<Product> predicate)` to update the Predicate in `filteredProducts` in `Model`. This execution then returns a `CommandResult` to `UI`, containing the response to the user.

Step 7. `UI` displays the response in the `CommandResult`. In addition, UI will change to display archived products after model updates `filteredProducts`, since `UI` is constantly listening for the change in `Model`.

The Sequence Diagram below shows how the components interact with each other for the above mentioned scenario.

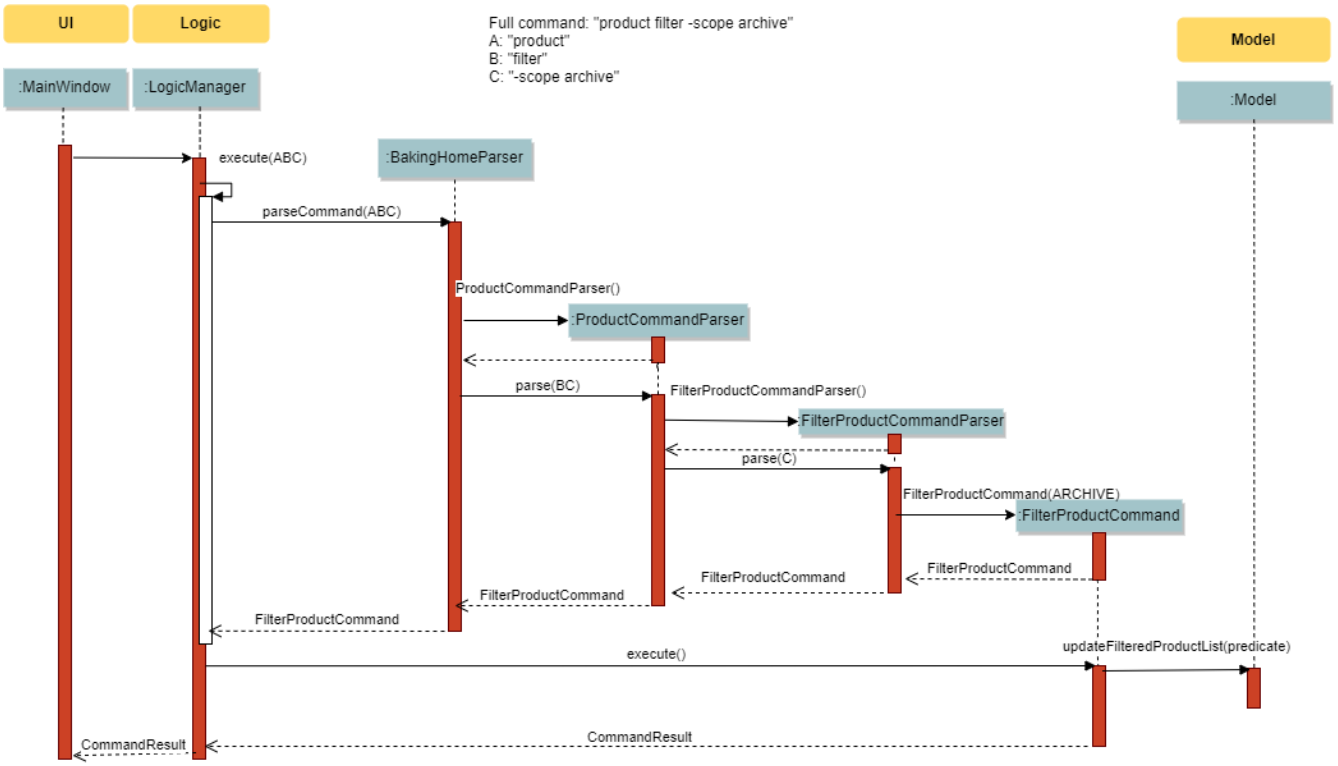

Figure 7. Sequence Diagram for Filter Product Mechanism

Note that almost all other commands follow the same sequence, with different `Command` and `Parser` classes.

3.2.2. Design considerations

- Alternative 1 (current choice): Save all products in an `ObservableList` in `BakingHome`, and keep a `FilteredList` in the `ModelManager`. `ProductCommandParser` parses the user input and gets the Predicate to update the `FilteredList`.
    - Advantages: Implementation is clearer and code is more human-readable.
    - Disadvantages: More difficult to write a Predicate.
- Alternative 2: Keep two separate product lists, one for archived products and one for active products.
    - Advantages: Fast access to products of both status.
    - Disadvantages: Implementation will become complicated. It also makes it very expensive when adding features like sorting all products according to name, price or cost.
- Alternative 2: Keep only one list of products. Loop through the list to get the products with the desired status.
    - Advantages: Simplicity in storing data.
    - Disadvantages: Time complexity is very high, resulting in a slow response of the application when the product list gets long.

3.3. Buy Shopping List Feature

BakingHome comes with a `shop buy` command in its shopping list feature. This command transfers

ingredients and its respective quantity from the shopping list to the inventory list. It will then generate a sales transaction automatically in the Sales page.

3.3.1. Implementation

The `shop buy` feature is facilitated by the `UniqueEntityLists` initialized in BakingHome, which is an implementation of `Iterable` and contains an `ObservableList`. There are 3 `UniqueEntityLists`, `inventory`, `shoppingList` and `sales`, which are involved in this feature and each of them has an `add` and `set` operation.

- `UniqueEntityList<class>#add(toAdd)` - Adds object `toAdd` into the `ObservableList` stored in the `UniqueEntityList`.
- `UniqueEntityList<class>#set(toEdit, edited)` - Replaces object `toEdit` with the new object `edited` in the `ObservableList` stored in the `UniqueEntityList`. Object `edited` will take the index position of object `toEdit` in the `ObservableList`.

These operations are exposed in the `Model` interface as `Model#addInventory`, `Model#setInventory`, `Model#setShoppingList`, and `Model#addSaleFromShopping`. The `UniqueEntityLists` are also exposed in the `Model` as `FilteredLists`, which wraps an `ObservableList` and filters using the provided `Predicate`.

3.3.2. Workflow

Given below is an example usage scenario and how the `shop buy` mechanism works.

Step 1. The user launches the application for the first time. The `UniqueEntityLists inventory`, `shoppingList`, and `sales` are initialized in BakingHome with the initial data stored in the `Storage`.

Step 2. The user inputs `shop buy 1,2` command to buy the first and second ingredient in the shopping list. This command goes through the `Parser` to get the indices of the ingredients to be bought and executes the `BuyShoppingCommand`.

Step 3. The `BuyShoppingCommand` calls the `FilteredLists` stored in the `Model` through `Model#getFilteredInventoryList()`, `Model#getFilteredShoppingList()` and stores them in the `ArrayLists<Item<Ingredient>>` `inventoryList` and `shoppingList` respectively.

Step 4. For every index, the `Item<Ingredient>` object is called from `shoppingList`. Each ingredient is checked whether `inventoryList` already contains it using `inventoryList#contain(Item<Ingredient> toBuy)`.

- If `inventoryList` contains it, a new `Item<Ingredient>` constructor is created with the added quantities of both lists. The new constructor then replaces the current one in `inventoryList` using the `inventoryList#set()` method.
- Else, the `Item<Ingredient>` object in `shoppingList` is just added to `inventoryList` using the `inventoryList#add()` method.

Step 6. For every ingredient that is bought in the shopping list, a new `Item<Ingredient>` constructor is created using the original ingredient's data but with quantity = 0. This new constructor then replaces the current one in `shoppingList` using the `shoppingList#set()` method.

Step 7. `BuyShoppingCommand` will calculate the total cost of the ingredients bought and pass it as parameters to `AddSaleFromShopping` method in `Model`, along with an `ArrayList` of the bought ingredients. `AddSaleFromShopping` will then create a `Sale` constructor with these values and add it to `sales`.

Step 8. These will be updated in the `UI` automatically as these objects are stored in `ObservableLists`.

The following sequence diagram shows how the `shop buy` mechanism works in showing the correct UI to the user after a `shop buy` command is inputted.

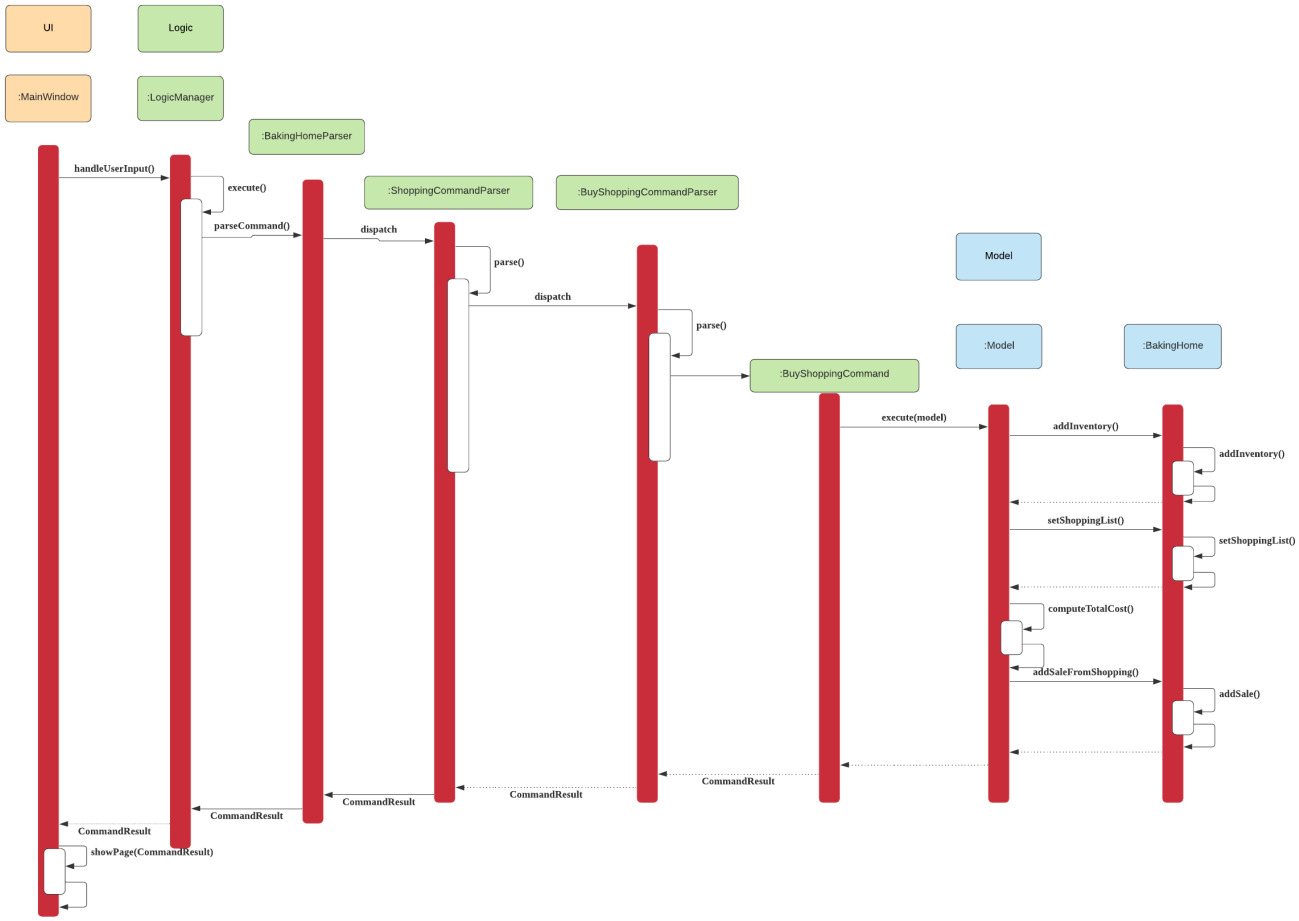

Figure 8. Sequence Diagram for Shop Buy Mechanism

3.3.3. Design considerations

- Alternative 1: Removing the ingredients from the shopping list after they are bought.
  - Pros: The shopping list is clearer and more readable for the user as redundant ingredients that he/she has already bought will not be shown on the list.
  - Cons: The costs and remarks that the user had made will be lost and he has to input them again the next time he wants to buy the same ingredients.
- Alternative 2 (current choice): Set the bought ingredients' quantity to 0 in the shopping list.
  - Pros: There is a saved template of the shopping list with past costs and remarks of the ingredients, making it convenient for the user to just edit the quantity to the quantity he needs to buy.
  - Cons: The shopping list may become very cluttered with too many ingredients. Hence, a `shop`

`list` command can be executed in the command line to filter out ingredients that have 0 quantity.

3.4. Add Sale Feature

For every purchase made under the shopping tab and every order delivered successfully under the orders tab, a sale entry is automatically added into BakingHome. Apart from that, users can manually add their own sale entries by using the `sale add` command. These entries can then be accessed under the Sales tab in the application.

3.4.1. Implementation

The `sale add` feature involves adding a `Sale` object to a `UniqueEntityList` named `sales` which is initialized inside BakingHome. A filtered copy of this list is also found inside `ModelManager` named `filteredSales`. Having two separate lists allow us to store all `Sale` objects that have been added while at the same time displaying only the desired entries using predicates.

Similar to the other 3 features, the operations to edit the two lists are exposed in `ModelManager` as the following public methods:

- `ModelManager#AddSale(sale)` – Adds a `Sale` object to the `UniqueEntityList sales`. The `filteredList` is also updated to show the new entry and filter predicate is reset to show all entries.
- `ModelManager#AddSaleFromOrder(order)` – Same as the above. Except the argument being passed in is an `Order` object, from which the necessary fields are being copied over to a new `Sale` object being created and added to the two lists.
- `ModelManager#AddSaleFromProduct(totalCost, toBuyList)` – Same as the above. Except the arguments being passed in are a `double` value (total cost) and an `ArrayList` (ingredients). A new `Sale` object is created with a fixed description denoting its origin from Products. The value and remarks are populated with the given value and ingredient lists respectively.

3.4.2. Workflow

Given below is an example usage scenario and how the add sale command works.

Step 1. The user launches the application for the first time. The `UniqueEntityList sales` is initialized in BakingHome with the initial data stored in the storage.

Step 2. The user inputs `sale add -desc TestDescription -val 12.34` to add a `Sale` entry with description “TestDescription” and positive value of 12.34 in revenue to sales.

Step 3. The command is parsed by SaleParserUtil to collect the relevant `String` and `Double` data which are then copied over to the new `Sale` object. The fields that are not given are initialized to their default value.

Step 4. `ModelManager#AddSale()` is called which in turn calls `BakingHome#AddSale()` and passes the created `Sale` object into the `UniqueEntityList sales`. At the same time, the `FilteredSaleList` is updated with the predicate `PREDICATE_SHOW_ALL_SALES` which as the name suggests, is used to display every sale entry.

3.4.3. Design Considerations

- Alternative 1 (current implementation): Store all sale entries in an `ObservableList` in BakingHome and keep a `FilteredList` in `ModelManager`.
    - Pros: Implementation is clearer and code is more readable.
    - Cons: Memory is wasted in keeping two sets of the list.
- Alternative 2: Store only one sale list. Loop through the list to get the desired entries for display.
    - Pros: Structure is simpler.
    - Cons: Time complexity is higher, which may result in slower queries especially as the number of entries increase.
- Alternative 3: Store a new list for each month / year.
    - Pros: Fast access times and easy transfer of data should new features require sale data for a specific month.
    - Cons: Potentially messier code.

3.5. Logging

We are using `java.util.logging` package for logging. The `LogsCenter` class is used to manage the logging levels and logging destinations.

- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

- `SEVERE` : Critical problem detected which may possibly cause the termination of the application.
- `WARNING` : Can continue, but with caution.
- `INFO` : Information showing the noteworthy actions by the App.
- `FINE` : Details that is not usually noteworthy but may be useful in debugging. e.g. print the actual list instead of just its size

Appendix A: Product Scope

Target user profile:

Bakery managers of home bakeries, who prefer typing and is willing to use a Desktop application to manage his business.

Such a manager needs to take care of every single aspect of his bakery business, from allocating responsibilities and keeping track of revenue, to taking the customer's order. He might even need to do the baking, since there is limited manpower.

Though currently there are many well-developed applications for the work he needs to do, there isn't one that integrates all the features he needs. Thus, it is hard for him to switch between different apps to manage his bakery business.

BakingHome is a one-stop desktop application that has all the important features for such a manager to eliminate the trouble of changing between different apps.

Value proposition:

A one-stop bakery management system for home bakeries.

Appendix B: User Stories

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

Priority	As a …	I want to …	So that I can…
* * *	user	Add products with details	Track what products my bakery has
* * *	user	Edit a product’s details	Keep my products updated to new improvements
* * *	user	Delete a product	Remove irrelevant products that have been phased out
* *	user	Archive a product	In case my business has evolved but I do not want to lose an older product
* *	user	Have a default ingredient cost calculated for me even if I don’t enter the cost	Have something to refer to when deciding the retail price
* *	user	Search for a product through keywords	find a product easily when the list gets long.
* *	user	Sort products through name, cost, price, profit	find a product easily when the list gets long.
* * *	user	Add a new order without specifying any details	Add orders more quickly
* * *	user	Edit an order’s details	Adjust the order if my customer’s preferences change
* * *	user	Delete multiple orders in one go	Save myself from the trouble of deleting them one by one
* *	user	Sort the orders by date created, deadline, and total price	Look for orders more easily
* *	careless user	Undo deleting an order	save myself from the trouble of typing out the whole order again
* *	user	Mark orders as done, canceled or completed	Track the status of my orders more easily
* * *	user	Add, edit and delete my ingredients in the shopping list easily	Manage the bakery more easily
* * *	user	Transfer my ingredients from the shopping list to inventory list in a single step	Save the trouble of having to manually re-key every single ingredient
* *	user	Clear all list items in one go	saved myself from the trouble of deleting one by one

Priority	As a …	I want to …	So that I can…
* *	user	Be able to know the price or estimated prices of the ingredients in my shopping list	I can budget myself and cut costs if necessary
* *	user	Be able to track the expiry dates of ingredients in my inventory	Keep stock without having to physically check it myself
* *	user	Be reminded of ingredients that are going to expire soon in the inventory	Reduce wastage of ingredients
* *	user	Have saved templates of my shopping lists	I do not have to input main ingredients that I usually buy every time
*	user	Be able to input the places of the ingredients sold in my shopping list and sort those ingredients according to those places	I do not miss out an ingredient when going shopping at a certain location
*	user	Input where I store my ingredients in my inventory	I can find them easily in real life
* * *	user	Store my transactions	Reference them easily in the future
* * *	user	Edit older transaction details	Change individual records which may have been logged wrongly
* * *	user	Delete older transaction	Remove older and irrelevant data from my sales calculations
* *	user	Automatically log expenditures and sales	There is no need to retype information from completing an order or shopping buy
* *	user	Calculate revenue, cost and hence profit instantly	Check the bakery’s finances with a quick glance

Appendix C: Use Cases

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

Use case 1: Deleting an Order

MSS

1. User requests to list all orders.
2. BakingHome shows a list of orders.
3. User requests to delete a specific order or multiple orders in the list.
4. BakingHome deletes the order(s).

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index(indices) is(are) invalid.

3a1. BakingHome shows an error message.

Use case resumes at step 2.

Use case 2: Buying an ingredient in the shopping list

MSS

1. User requests to list all ingredients in the shopping list.
2. BakingHome shows a list of ingredients.
3. User requests to buy a specific ingredient or multiple ingredients in the shopping list.
4. BakingHome transfers these ingredients from the shopping list to the inventory list and adds a sales transaction with the total cost of the bought ingredients to the Sales page.

   Use case ends.

Extensions

2a.The list is empty.

Use case ends.

3a. The given index(indices) is(are) invalid.

3a1. BakingHome shows an error message.

Use case resumes at step 2.

Use case 3: Completing an Order

MSS

1. User requests to list all orders.
2. BakingHome shows a list of orders.
3. User requests to complete a specific order or multiple orders in the list.
4. BakingHome checks inventory for the required ingredients by the order and deducts the necessary amount from inventory (if insufficient it deducts to zero).
5. BakingHome marks the order(s) as complete.

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index(indices) is(are) invalid.

3a1. BakingHome shows an error message.

Use case resumes at step 2.

Use case 4: Deleting a Sale Entry

MSS

1. User requests to list all sales.
2. BakingHome shows a list of sales.
3. User requests to delete a specific sale or multiple sales in the list.
4. BakingHome deletes the sale(s).

   Use case ends.

Use case 5: Viewing Active Orders

MSS

1. User requests to list all active orders.

2. BakingHome shows a list of active orders.

   Use case ends.

Use case 6: Viewing Product's ingredients

Precondition: BakingHome has at least 1 product in the product list.

MSS

1. User request to list products
2. BakingHome lists products
3. User request to view a specific product's ingredients
4. BakingHome shows the ingredients of that product

   Use case ends.

   - 3a. The given index(indices) is(are) invalid.

     3a1. BakingHome shows an error message.

     Use case resumes at step 2.

Use case 7: Show a Product

- **Precondition:** User has at least 1 product in the product list.
- **MSS**
  1. User can be viewing any pages.
  2. User enters a ShowProductCommand indicating the index of the product to be shown, e.g. `product show 1`.
  3. BakingHome parses the command.
  4. BakingHome executes the command.
  5. BakingHome displays the details of the product.
- **Extensions**

  3a. BakingHome detects a invalid command.

  3a1. BakingHome shows an error message

  3a2. Use case ends.

Appendix D: Non Functional Requirements

1. Should be an open-source project.
2. Should be portable (i.e. it does not require installation to run).
3. Should a single user application.
4. Should work on any mainstream OS as long as it has Java 11 installed.
5. Should be able to hold up to 150 entities without a noticeable sluggishness in performance for typical usage.
6. Should have a low response time of not more than 2 seconds.
7. Changes are saved automatically and no manual saving is needed.
8. 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.
9. A user would be able to execute every operation with typing only, and without the assist of a mouse

Appendix E: 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.

E.1. Launch and Shutdown

1. Initial launch
   a. Download the jar file and copy into an empty folder
   b. Run the jar file in console using `java -jar`
      Expected: Shows the GUI. The window size may not be optimum.
2. Showdown
   a. Enter `exit` in command box.
      Expected: The application quits.

E.2. Saving data

1. Dealing with missing/corrupted data files
   a. Test case (missing data file): Delete the folder `data` in BakingHome's directory and restart BakingHome
      Expected: BakingHome loads demo data
   b. Test case (corrupted data file): Change the content of data file at `data/baking.json` to "This file is damaged" and restart BakingHome.
      Expected: BakingHome loads demo data

BakingHome provides basic Add, Edit, Delete operations for all sections.

E.3. Adding entries

Table 1. Add Operation

Section	Test Case	Expected Output
Products	Product with name `Bread` and `Cheese cake` are not in the product list.	
	`product add -name Bread`	A new product named Bread is added. Ingredient cost and Retail price are both $0.0. Status is active.
	`product add -name Cheese cake -ingt [Cream cheese, 3] [Sugar, 5] -cost 3.0 -price 5.9` to check the ingredients	A new product named `Cheese cake` is added. Ingredient cost are $3 and Retail price is $5.9. Status is active. Use `product show INDEX_OF_CHEESE_CAKE` to check ingredients
Orders	`order add -name Jiajun`	A new order with no item is added. The order's customer name field is Jiajun
	`order add -name abcdabcdabcdabcdabcdabcd`	An error message pops up. The error message is "Name should be no more than 20 characters"
	`order add -rmk abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcd`	An error message pops up. The error message is "Remarks should be no more than 50 characters"
	Pre-requisite for below `Order`: Cake product should be in Product List; Fish product should not be in Product List.	
	`order add -item Cake, 1`	A new order with one item Cake is added.
	`order add -item Fish, 1`	A error message pops up.

Section	Test Case	Expected Output
Sales	Adding an empty sale	
	`sale add`	A new sale with default value 0.0 is added. The sale's description and remarks are "N/A" and date is set to current date and time.
	`sale add -desc Refund abcdeabcdeabcdeabcdeabcdeabcde abcde has been completed`	An error message pops up. The error message is "Description should be no more than 50 characters"
	`sale add -rmk Uvuvwevwevwe Onyetenyevwe Ugwemuhwem Osas Uvuvwevwevwe Onyetenyevwe Ugwemuhwem Osas`	An error message pops up. The error message is "Remarks should be no more than 50 characters"
	Adding a sale with date	
	`sale add -at 31/12/2019 23:00`	A new sale with date Tue, Dec 31, 2019 23:00 is added
	`sale add -at 32/13/2019 23:00`	An error message pops up

E.4. Editing entries

Prerequisite: At least one entry in each list.

Table 2. Edit Operation

Section	Test Case	Expected Output
Shopping	`shop edit 1 -qty 10 -cost 10`	In the shopping list, the first ingredient's quantity is changed to 10, and its unit cost is changed to 10. All other data that is not inputted as parameters will be unchanged.
Products	`product edit 1 -name _name -cost 5 -ingt [Cream cheese, 1.0]`	The name of the first product is changed to _name, and cost is changed to $5. It has a ingredient Cream cheese associated. You can check the ingredient using command `product show 1`

Section	Test Case	Expected Output
Orders	`order edit 1 -name Eugene -rmk Birthday`	In orders, the first order's customer name is changed to Eugene, and the remarks is changed to Birthday All other data that is not inputted as parameters will not be changed. This is provided the order status is not completed. If the order status is completed, you cannot edit the order.

E.5. Deleting entries

Delete operation have the same syntax for all five sections. The table below use order as an example. `order` can be replaced with `inv`, `shop`, `product` and `sale`

Table 3. Delete Operation

Section	Test Case	Expected Output
Orders	Deleting a single order Prerequisites: At least 1 order in Order List	
	`order remove 1`	The first order is deleted from the list
	`order remove 0`	No order is deleted. Error details shown in the pop-up bar
	Deleting multiple orders Prerequisites: At least 2 orders in Order List	
	`order remove 2, 1`	The first and second orders are deleted
	`order remove 1~2`	The first and second orders are deleted
	1. `order remove 1~x` 2. `order remove 1,x` (where x is larger than the list size) 3. `order remove 2~1`	Error details shown in the pop-up bar.

E.6. Sales operations

Sales can be filtered by time.

Table 4. Filter Operation

Section	Test Case	Expected Output \
Sales	Showing only sales between two dates not inclusive of date itself.	
	`sale filter -from 01/01/2019 06:00 -to 10/01/2019 06:00`	Sale entries starting from 02/01/2019 00:00 to 09/01/2019 23:59 are shown
	`sale -from 01/01/2019 15:00 -to 32/01/2019 15:00`	Error is shown. Nothing changes to sale. Error details shown in the pop-up bar.

E.7. Shopping List operations

Table 5. Buying Operation

Feature	Test Case	Expected
Buying a single ingredient. Prerequisites: At least 1 ingredient in Shopping List.	`shop buy 1`	The first ingredient is bought and transferred to Inventory List.
	`shop buy 0`	No ingredients are bought. Error details shown in the pop-up bar.
	`shop buy Cheese`	No ingredients are bought. Error details shown in the pop-up bar.
Buying multiple ingredients. Prerequisites: At least 2 ingredients in Shopping List.	`shop buy 1, 2`	The first and second ingredients are bought and transferred to Inventory List.
	`shop buy 0, 2`	No ingredients are bought. Error details shown in the pop-up bar.

E.8. Product operations

Table 6. Filter Operation

Feature	Test Case	Expected
Searching for products whose name contains the given keyword	`product search -include cake`	All products whose name include `cake` are listed. List will be empty if no products' names contain `cake`.

Feature	Test Case	Expected
Filtering Products by Categories, namely `active`, `archive` and `all`.	`product filter -scope active`	Only active products are listed.
	`product filter -scope all`	Both active and archived products are listed. Note that no archived products will be shown if no products are archived.
Sorting Products.	`product sort -by cost -scope active -re`	Active products are sorted by cost in ascending order.
	`product sort -by price`	Active products are sorted by price in descending order.

Appendix F: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X