Cube Developer Guide

Team: CS2113T-F09-2 | Updated: Nov 2019 | License: MIT

Cube

Table of Contents

Preface

About Project Cube

Cube is a simple Bookkeeping and Inventory Management System targeted for sellers looking to set-up a small online marketplace. This application is targeted at online store owners who wish to easily keep track of the various products and revenue earned from the products they are selling. The early development stages (pre-v2.0) of **Cube** is currently limited to managing food products.

1. Introduction

This section will give you a brief overview of this project and introduce the purpose of this document.

Purpose

This document describes the architecture and system design of the Project Cube developed by the CS2113T Team CS2113T-F09-2 for the Module CS2113T, AY19/20 S1, NUS. It is a living document that evolves throughout the design and implementation for each major release, and the current edition of the document is intended for the release v1.4.

The goal of this document is to cover the high-level system architecture and design. The document is divided into three major parts: system architecture, software design and external interfaces. The system architecture includes views from various perspectives. The software design details the main software components that operate under, and support the system architecture. It also includes the content lifecycle and workflows that support the business operations of the system. External interfaces are documented in a separate section because of their special roles in the system.

The table below details the explanation behind various symbols and formatting that are used in this document:

filename	A grey highlight usually denotes some code snippets, filenames, navigation or interaction with the interface within the user’s local machine.
Parser	A light blue highlight indicates components, class or methods in Java code.
important	A yellow highlight indicates key files or points of the instructions to be noted by the user.

ℹ️	This symbol indicates that there is some additional information to be taken note by the user.

2. User Profile

Cube is developed to cater for a range of small online retail business owners in mind, and a typical user profile that may be interested in using Cube may include:

- Users who are looking to kick-start their first online retail business.
- Users who have an existing small-scale online retail business.
- Users who do not have an existing solution to help manage their retail business.
- Users who prefer to work with a Command Line Interface (CLI) environment.
- Users with fast typing speed of around 60 words per minute (wpm) and over.

3. Value Proposition

Cube aims to integrate key features that online store owners would find it useful to assist in the operation of their store, such as product, profit and schedule management, all within a single package.

3.1 Product Management

- Cube allows users to manage their existing inventory by providing features such as categorizing items based on their type, or setting customised reminders such as notifications on expiring perishable products.

3.2 Profit Management

- Cube has an added functionality that allows users to keep track of their profit and revenue obtained the sales of their products.

3.3 Schedule Management [coming in v2.0]

- Cube allows users to add tasks related to their store such as meeting customers, or scheduling for product delivery within the application.

4. Setting up

This section aims to guide the user on how to set-up the local machine to begin development of the project.

4.1 Pre-requisites

1. **Open JDK 11** or above is required to run to build the program.
2. **IntelliJ IDEA** with **Gradle** plugin enabled is the recommended integrated development environment (IDE) for the development project.

ℹ	IntelliJ IDEA by default has Gradle and JavaFx plugins installed. If the plugins are not enabled, you can go to File > Settings > Plugins to enable them.

3. **Git** or any graphical Git clients for code version control.

4.2 Setting Up on Local Machine

1. Fork this repository using **Git**, and proceed to clone the forked repository into your local machine.
2. Launch **IntelliJ IDEA** (if you are not presented with the welcome screen, click File > Close Project to close the existing project dialog first).
3. Set up the correct JDK version to be used with Gradle.
   i. Click Configure > Structure for New Projects
   ii. Under Project Settings click on Project
   iii. Check if SDK version 11 is being selected, if not, select JDK 11 from the drop down menu or Click on New... and navigate to the directory of the JDK.
   iv. Click OK to save the project configuration.
4. Click Import Project
5. Locate the build.gradle file and select it. Click OK to proceed.
6. Click Open as Project.
7. Click OK to accept the default settings.
8. Open the IntelliJ console/terminal and run the gradle command `gradlew processResource` for Windows, or `./gradlew processResource` on Mac or Linux.
   i. If you have encountered any permission error while running the command on Mac or Linux, add the required permissions by running `chmod 744 gradlew` in your terminal.
   ii. The command should finish with the BUILD SUCCESSFUL message and generate the resources required by the application and accompanying tests.
9. Now you are ready to start on developing or modifying **Cube**!

4.3 Verifying the Set Up

1. Build and run `cube.Cube` and try out a few commands to verify that the set up has been done correctly. (You can refer to the User Guide here for explanation on the available commands).
2. Run the JUnit tests by running `gradlew test` command to verify that all test cases pass and that there are no regressions in the code.

5. Program Architecture

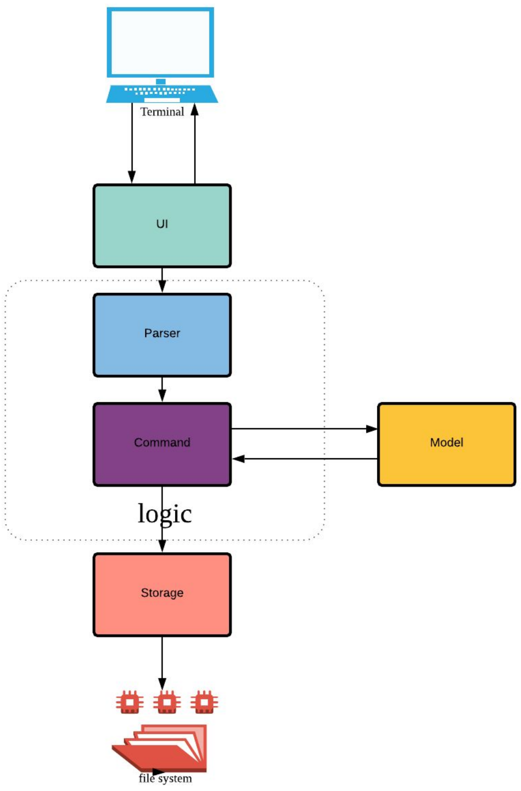


Figure 1: Architecture diagram of Cube.

Cube adopts an n-tier architecture. The highest level is Ui which accept input from and show output to the user. The input are parsed by Parser, which calls a Command. Command interacts with Model and save the changes to Storage. Storage will save data to a Json file.

The usage of each component:
UI : Reading input and showing output.
Logic: Interpreting and executing the user command. Consists of Parser and Command.
Parser: Interpreting the command line arguments from user.
Command: Executing the commands.
Storage: Storing the changes into files.
Model: Representing real-life items.

Each of the components encapsulate in a package of the same name as the component.

6. Design

This section gives the class diagrams for our project design. It is further divided into overall design, logic component design and storage component design.

6.1 Overall Design

The major code breaks down into different components, including ui, parser, command, storage and model. The Main class initializes UI, Parser, Storage and FileUtilJson. The interaction of components are shown in the diagram below:

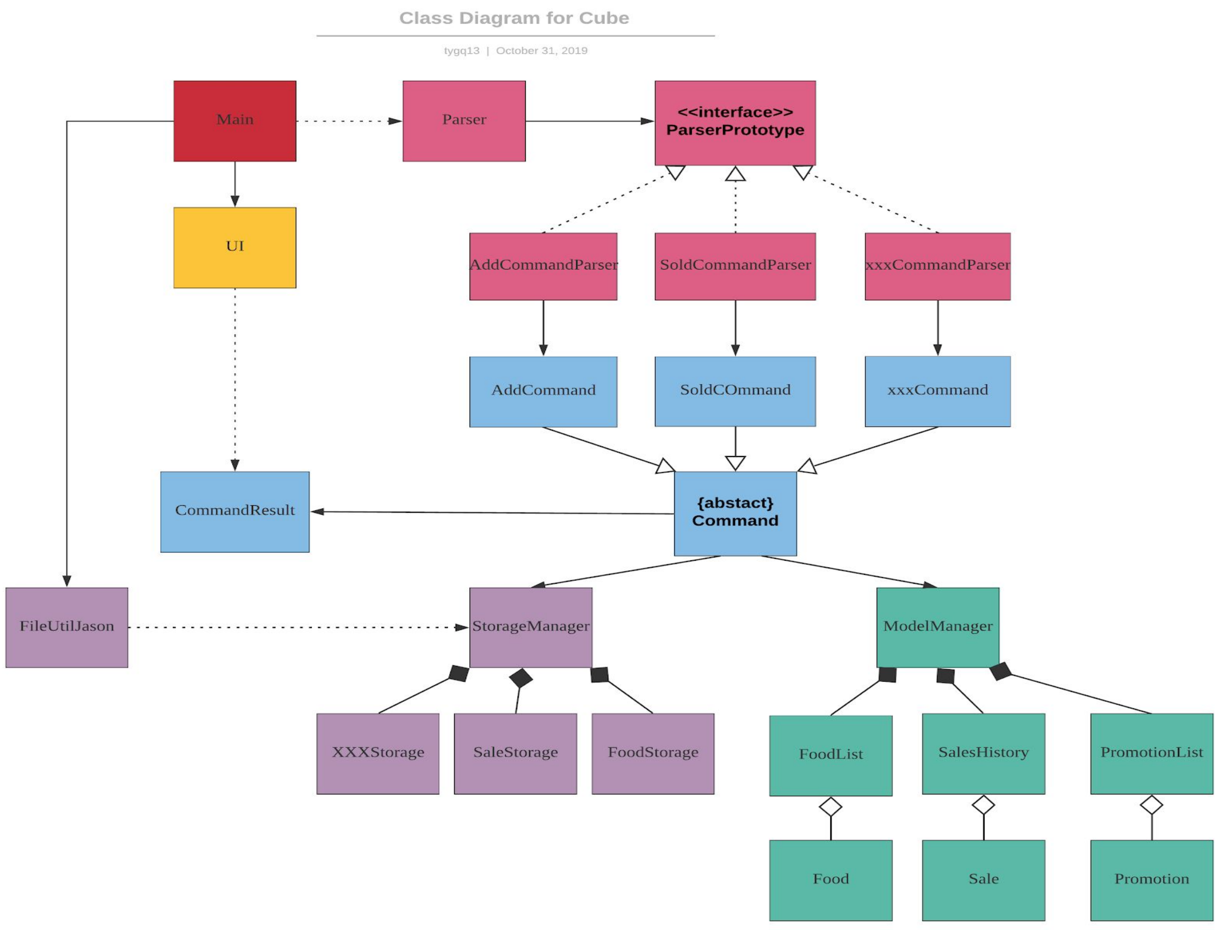


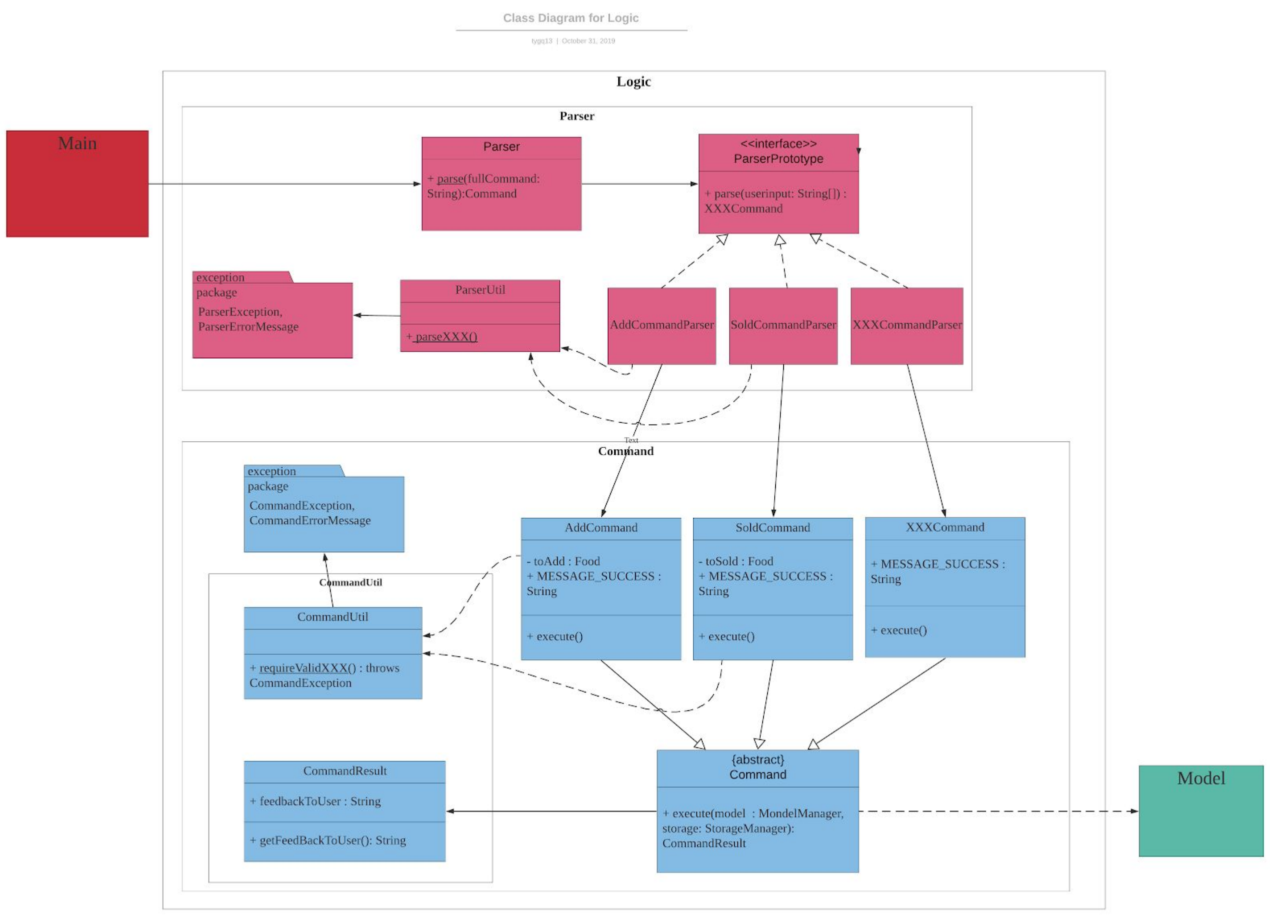
Figure 2: Overview of the essential classes in different components.

Each class of different color represents different component. The break down of details are below:

Main Main component
Ui Ui component
Parser Parser component
Command Command component
Storage Storage component
Model Model component

6.2 Logic Component

Below is the UML class diagram for logic component. Logic component consists of Parser and Command.



Class Diagram for Logic
Logic
Parser
Main
Parser
+ parse(fullCommand: String):Command
<<interface>>
ParserPrototype
+ parse(userinput: String[]) : XXXCommand
exception package
ParserException, ParserErrorMessage
ParserUtil
+ parseXXX()
AddCommandParser
SoldCommandParser
XXXCommandParser
Command
exception package
CommandException, CommandErrorMessage
CommandUtil
CommandUtil
+ requireValidXXX() : throws CommandException
AddCommand
- toAdd : Food
+ MESSAGE_SUCCESS : String
+ execute()
SoldCommand
- toSold : Food
+ MESSAGE_SUCCESS : String
+ execute()
XXXCommand
+ MESSAGE_SUCCESS : String
+ execute()
CommandResult
+ feedbackToUser : String
+ getFeedBackToUser(): String
{abstract}
Command
+ execute(model : MondelManager, storage: StorageManager): CommandResult
Model

Figure 3. UML class diagram for Logic component

Parser breaks down:

1. Parser class is used by Main class through static method parse(String).
2. The parse() method interprets the command type, then pass the user input to a specific parser class through the ParsePrototype interface (eg. AddCommandParser).
3. A typical command parser(e.g AddCommandParser) uses ParserUtil class's static methods and may throw a ParseException.
4. The command parser returns its corresponding command object.

Command breaks down:

1. A typical command initializes with model parameters, such as "toAdd" in AddCommand. It also has a MESSAGE_SUCCESS as the feedback to be passed to user.
2. Command is executed by method execute(). It may use CommandUtil to check the validity of command parameters and throw an CommandException if they are not valid.

3. Command interacts with model, such as adding a Food to FoodList. The change is saved through StorageManager (details are explained in storage component)

Given below is the Sequence Diagram for interactions within the Logic component for the execute("sold food")

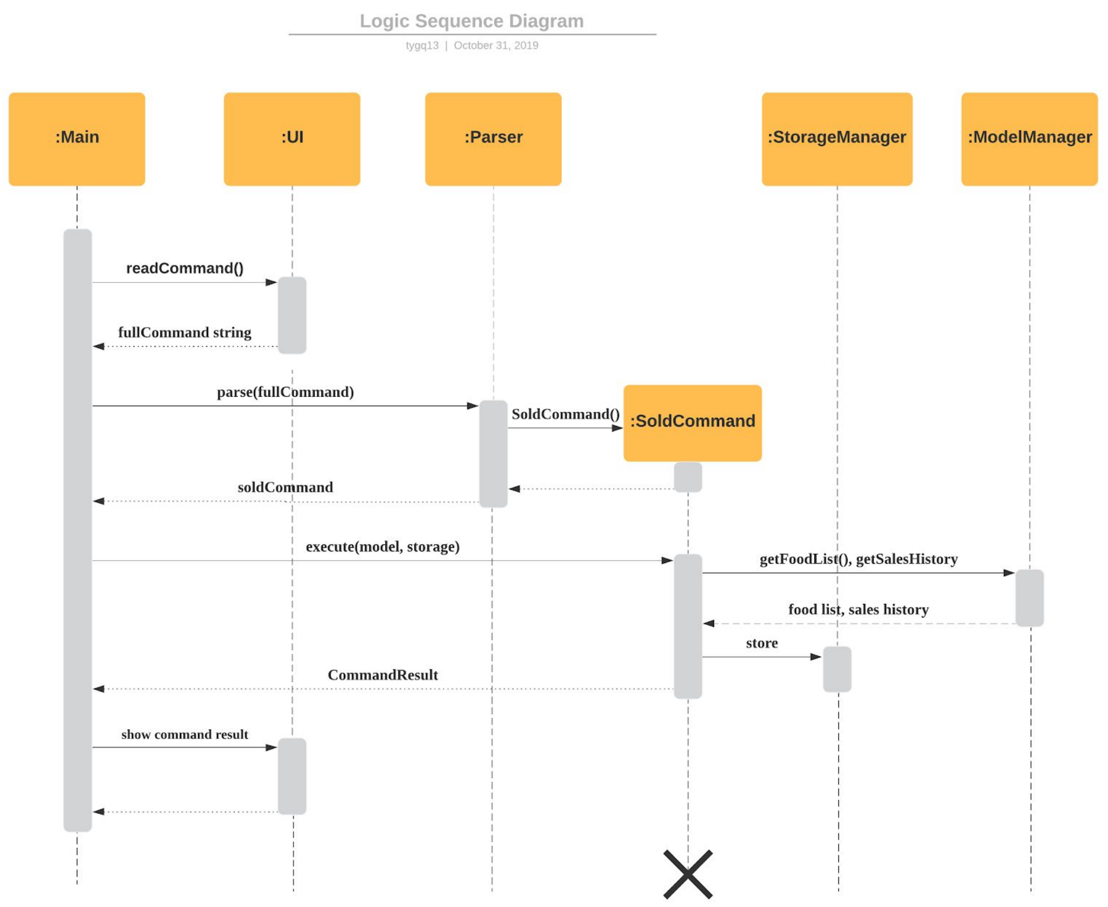


Figure 4. UML sequence diagram for executing a “sold” command.

6.3 Storage Component

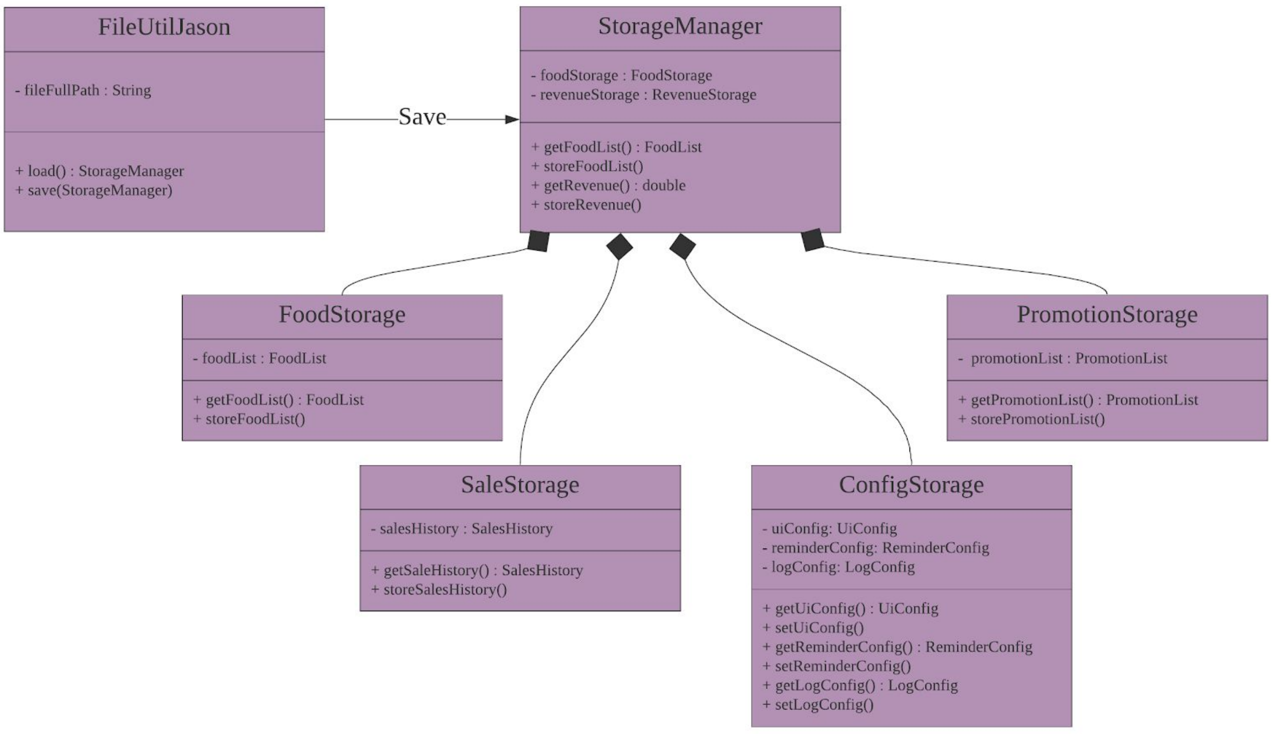

FileUtilJason
- fileFullPath : String
+ load() : StorageManager
+ save(StorageManager)
Save
StorageManager
- foodStorage : FoodStorage
- revenueStorage : RevenueStorage
+ getFoodList() : FoodList
+ storeFoodList()
+ getRevenue() : double
+ storeRevenue()
FoodStorage
- foodList : FoodList
+ getFoodList() : FoodList
+ storeFoodList()
PromotionStorage
- promotionList : PromotionList
+ getPromotionList() : PromotionList
+ storePromotionList()
SaleStorage
- salesHistory : SalesHistory
+ getSaleHistory() : SalesHistory
+ storeSalesHistory()
ConfigStorage
- uiConfig: UiConfig
- reminderConfig: ReminderConfig
- logConfig: LogConfig
+ getUiConfig() : UiConfig
+ setUiConfig()
+ getReminderConfig() : ReminderConfig
+ setReminderConfig()
+ getLogConfig() : LogConfig
+ setLogConfig()

Figure 5: UML class diagram for storage component.

API : Storage.java

1. StorageManager comprises of FoodStorage, SaleStorage, ConfigStorage and PromotionStorage, which have the desired objects saved in respective storage.
2. StorageManager acts as a facade for other storages. Interaction with other components are done through StorageManager.
3. FileUtilJson will save the whole StorageManager.

Given below is the Sequence Diagram for performing a saving of Food.

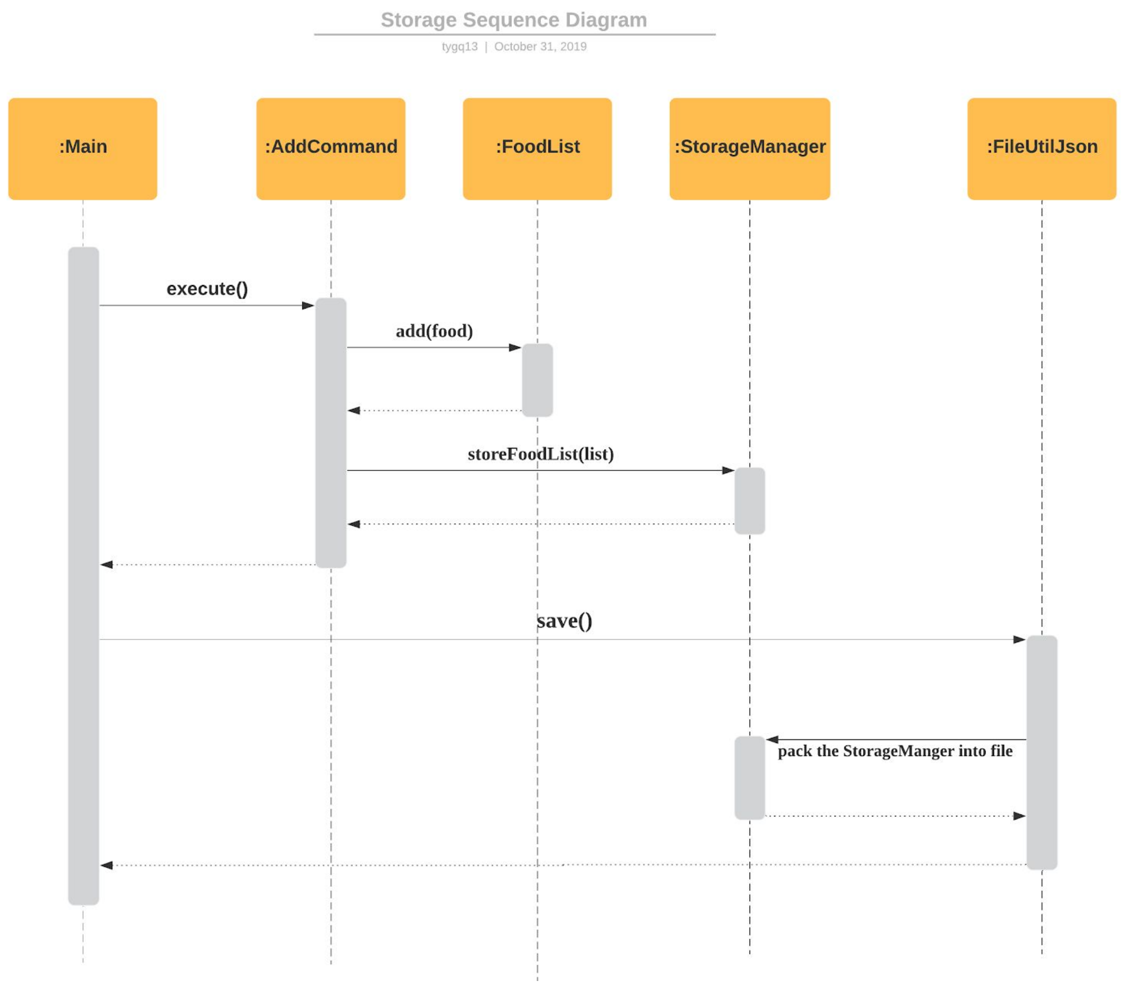


Figure 6: UML sequence diagram for performing a saving of Food.

- During execution of a command, the command interacts with Models(Food) and saves the changes through StorageManager. StorageManager will update the FoodList it contains.
- After execution, the main drive will perform storage through save() method in FileUtilJson which saves the StorageManager into a file.

7. Implementation

This section introduces the specific implementation of different commands in Cube. Major feature commands in cube can be categorized into three parts: manipulation of food list, information retrieval from food list, and features related to profit. This section also includes the underlying assumptions when implementing these commands.

7.1 Implementation Assumptions

The process of parsing user inputs and passing relative parameters to corresponding commands are implemented in Logic component. Therefore, the correct execution of logic component relies on correct implementation of ModelManager component, UI component, and Storage component. More specifically, in ModelManager the correctness of food list, promotion list, and sales history is required.

As shown in Figure 7 below, the correct implementation of Logic component is relied on the three other components of this project.

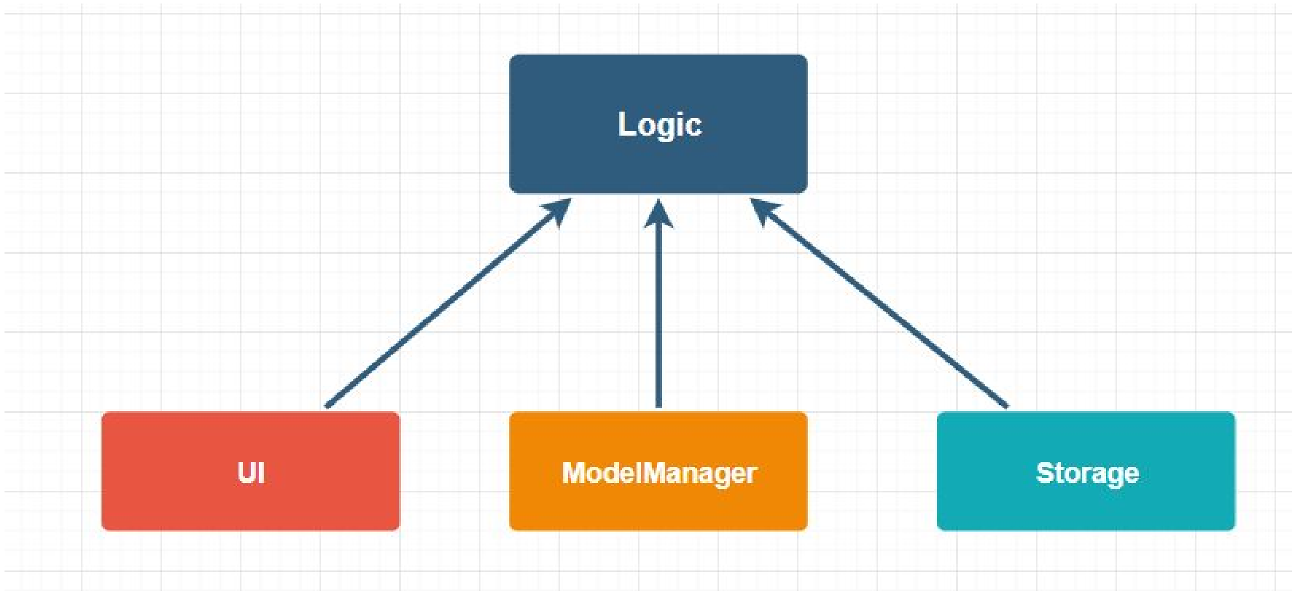


Figure 7. Logic component based on other components

For more information on other components in Cube, please refer to the previous section, [Section 6. Design](), in this Developer Guide.

7.2 Implementation of Manipulation on Food List

Basic stock inventory management commands include create, retrieve, update, and delete (CRUD). Among them, manipulation of food list involves create (`add`), update (`update`), and delete (`delete`).

7.2.1 Implementation Description

The command format for add is add <FOOD_NAME> [-t FOOD_TYPE] [-p PRICE] [-c COST] [-s STOCK] [-e EXPIRY_DATE]. The implementation of add command allow users to execute the following operations:

- add <FOOD_NAME>: add a food item with FOOD_NAME as name and all other attributes empty
- add <FOOD_NAME> [any combination of parameters]: add a food item with FOOD_NAME as name and all other attributes specified by user

The command format for update is update <FOOD_NAME> [-t FOOD_TYPE] [-p PRICE] [-c COST] [-s STOCK] [-e EXPIRY_DATE]. The implementation of update command allow users to execute the following operation:

- update <FOOD_NAME> [any combination of parameters]: update a food item with FOOD_NAME as name and all other attributes specified by user

The command format for delete is delete -i [INDEX] | -n [FOOD_NAME] | -t [FOOD_TYPE] | -all. The implementation of delete command allow users to execute the following operations:

- delete -i [INDEX]: delete a product from food list by index
- delete -n [FOOD_NAME]: delete a product from food list by name
- delete -t [FOOD_TYPE]: delete all products of a given type from food list
- delete -all: delete all products from food list

In this section, delete will be taken as an example to introduce the implementation of food list manipulation in Cube.

Figure 8 below shows the Sequence Diagram when user inputs delete -all.

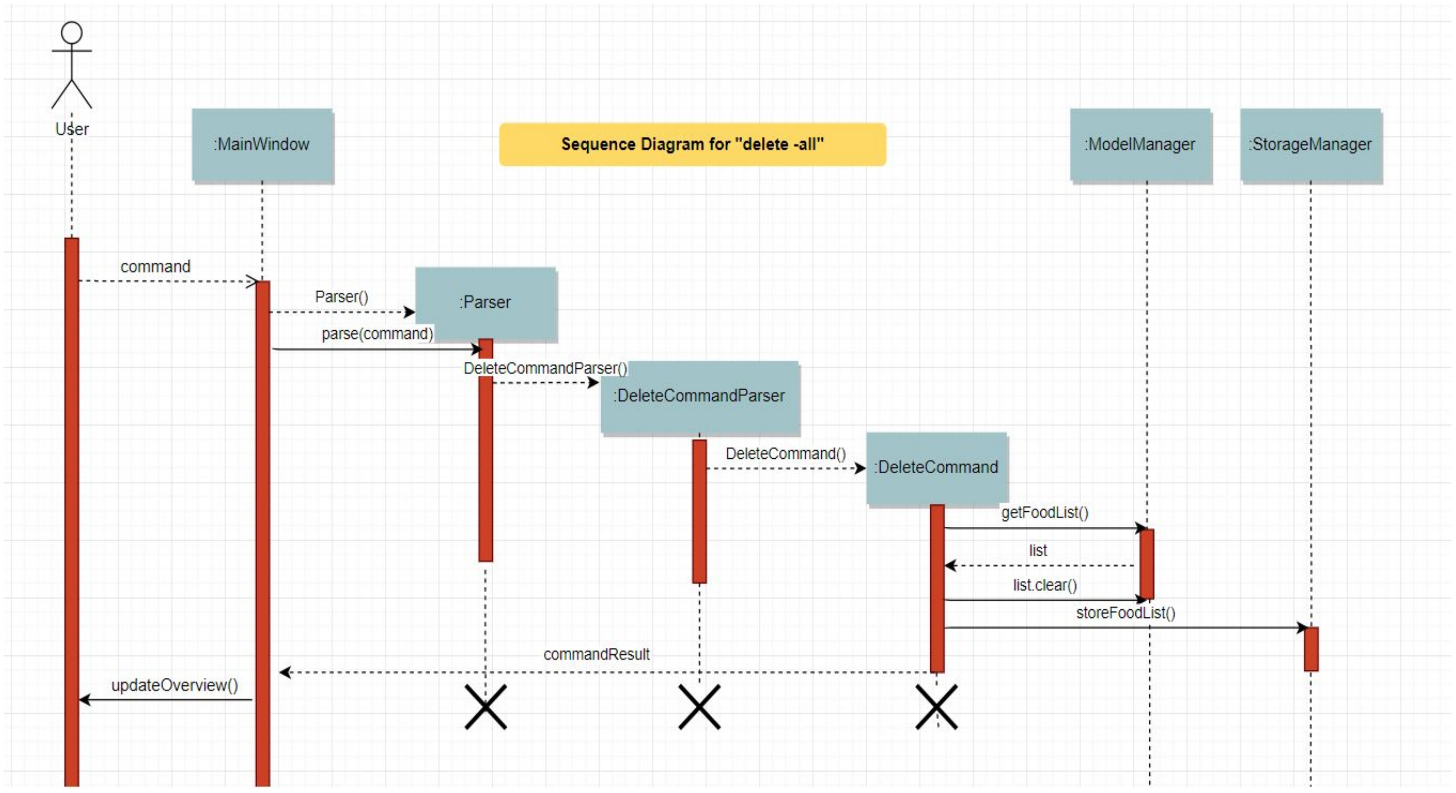


Figure 8. The UML Sequence Diagram for “delete -all”

Step 1. User inputs command “delete -all” through GUI main window.
Step 2. GUI main window creates a new Parser class and calls the parse() method with the command.
Step 3. Parser extracts “delete” from command. According to the command type, it creates a DeleteCommandParser class with the command.
Step 4. DeleteCommandParser parses the command and extracts delete type “-all”. It then creates a DeleteCommand and passes “-all” to it.
Step 5. According to the instruction “-all”, DeleteCommand gets food list from model manager and clears the list using clear() method declared inside FoodList class. It then calls storage manager to store the new list.
Step 6. DeleteCommand will return the command result to GUI main window, and GUI will display the result to the user.

7.2.2 Implementation Considerations

Aspect: Parsing user input and calling corresponding command

Alternative 1: Creating a whole class for all parsers and another class for all commands

- Pros: It is easy to code and view since all relevant variables and methods are together.
- Cons: Testing will be difficult as the codes are bulky. The principle of separation of concerns is also violated.

Alternative 2: Design patterns: Command pattern

- Pros: It supports polymorphism of different parser and command type as they all inherit from a general parser or command prototype class.
- Cons: Learning a new design pattern takes time and it is more challenging to understand and code.

After considering the two ways of implementation described above, the second way, command pattern design is finally chosen due to its clarity and extendability. By defining a general parser or command prototype class, it is very easy to follow the defined format and add in new implementation of parser and command without worrying about the problem of compatibility.

7.3 Implementation of Information Retrieval from Food List

In Cube, retrieve in CRUD is implemented through two commands: `list` and `find`. Moreover, the reminder command applies data analysis after retrieving information from food list.

7.3.1 Implementation Description

The command format for list is `list [-sort (expiry | name | stock)]`. The implementation of `list` command allow users to execute the following operations:

- `list`: show all products inside the list
- `list -sort (expiry | name | stock)`: sort and show all products by expiry date/name/stock in ascending order

The command format for delete is `find -i [INDEX] | -n [FOOD_NAME] | -t [FOOD_TYPE] [-sort (expiry | name | stock)]`. The implementation of `find` command allow users to execute the following operations:

- `find -i [INDEX]`: find a product from food list by index
- `find -n [FOOD_NAME]`: find a product from food list by name
- `find -t [FOOD_TYPE]`: find all products of a given type from food list
- `find -t [FOOD_TYPE] -sort (expiry | name | stock)`: find all products of a given type from food list and sort the results by expiry date/name/stock in ascending order

ℹ	Index in Cube is not permanent. By sorting the list, the index will also change.

7.3.2 Implementation Considerations

Aspect: Representing the way in which user wants to sort the results

Alternative 1: Using a parameter with a dash and one letter

- Pros: It is easier and faster to type as there are only two keystrokes.
- Cons: It is difficult to check whether the input letter is legal or not. The logical statement may be very long.

Alternative 2: Defining an enumeration containing allowed sort type

- Pros: It is clearer and more understandable. All sort types are defined in full names so that developers know what types are supported at first glance. Adding new sort types in future stage is also easier.
- Cons: Extra attention must be paid to the uppercase or lowercase letter problem because enumeration only accepts strictly the same values.

After considering the two ways of implementation described above, checking sort type is finally implemented using the second way, enumeration. Besides the advantages described above, enumeration also supports many other functionalities, such as checking whether a value is inside the enumeration, thus improving the usability.

7.4 Implementation of Profit Related Features

As a stock inventory management system specifically fine-tuned for online shop owners, Cube implements features related to sales, profits and promotions, with the respective commands being `sold`, `promotion`, and `profit`.

7.4.1 Implementation Description

The command format for `sold` is `sold <FOOD_NAME>  <-q QUANTITY> (<-t DATE_OF_SALE>)`. It is noteworthy that `DATE_OF_SALE` should be of the format of DD/MM/YYYY, where DD is the date, MM is the month, and YYYY is the Year. The implementation of `sold` command allows the users to execute the following operations:

- `sold <FOOD_NAME>  <-q QUANTITY>`: record the quantity of the food sold on the day that the operation is executed. It is noteworthy that in this project, duplicate food names are disallowed. As such, the food name uniquely specifies the food whose profit and revenue the user wishes to generate;
- `sold <FOOD_NAME>  <-q QUANTITY> <-t DATE_OF_SALE>`: record the quantity of the food sold on a particular day. It is noteworthy that in this project, duplicate food names are disallowed. As such, the food name uniquely specifies the food whose profit and revenue the user wishes to generate;

Under the command `promotion`, there are three sub-commands that it supports. The implementation of `promotion` command allows the users to execute the following operations:

- `promotion <FOOD_NAME> -% <DISCOUNT> -s [START_DATE] -e <END_DATE>`: add in a new promotion of food with relative information;
- `promotion -delete (INDEX | -all)`: delete a promotion at specified index in the list (1-indexed), or delete all promotions;

- promotion -list: list all promotions available.

The command format for profit is profit -t1 <START_DATE> -t2 <END_DATE> -all | -i <INDEX> | -n <FOOD_NAME> | -t <FOOD_TYPE>. It is noteworthy that the start date and the end time should be of the format of DD/MM/YYYY, where DD is the date, MM is the month, and YYYY is the Year. The current implementation does not support shifting orders, which means that the command has to stick to the given format above. As such, commands such as profit -t2 <END_DATE> -t1 <START_DATE> -all are not supported. The implementation of profit command allows the users to execute the following operations:

- profit -t1 <START_DATE> -t2 <END_DATE> -all: generate the profit and revenue of all the food stocks within the stipulated time period from the specified start time to the specified end time;
- profit -t1 <START_DATE> -t2 <END_DATE> -i <INDEX>: generate the profit and revenue of a specific food, specified by index in the food list (1-indexed), within the stipulated time period from the specified start time to the specified end time;
- profit -t1 <START_DATE> -t2 <END_DATE> -n <FOOD_NAME>: generate the profit and revenue of a specific food, specified by its name, within the stipulated time period from the specified start time to the specified end time. It is noteworthy that in this project, duplicate food names are disallowed. As such, the food name uniquely specifies the food whose profit and revenue the user wishes to generate;
- profit -t1 <START_DATE> -t2 <END_DATE> -t <FOOD_TYPE>: generate the profit and revenue of all the food belonging to a specific food type, specified by the name of the food type, within the stipulated time period from the specified start time to the specified end time;

In this section, sold will be taken as an example to introduce the implementation of profit related features in Cube.
Figure 8 below shows the Sequence Diagram when user inputs sold xxxx.

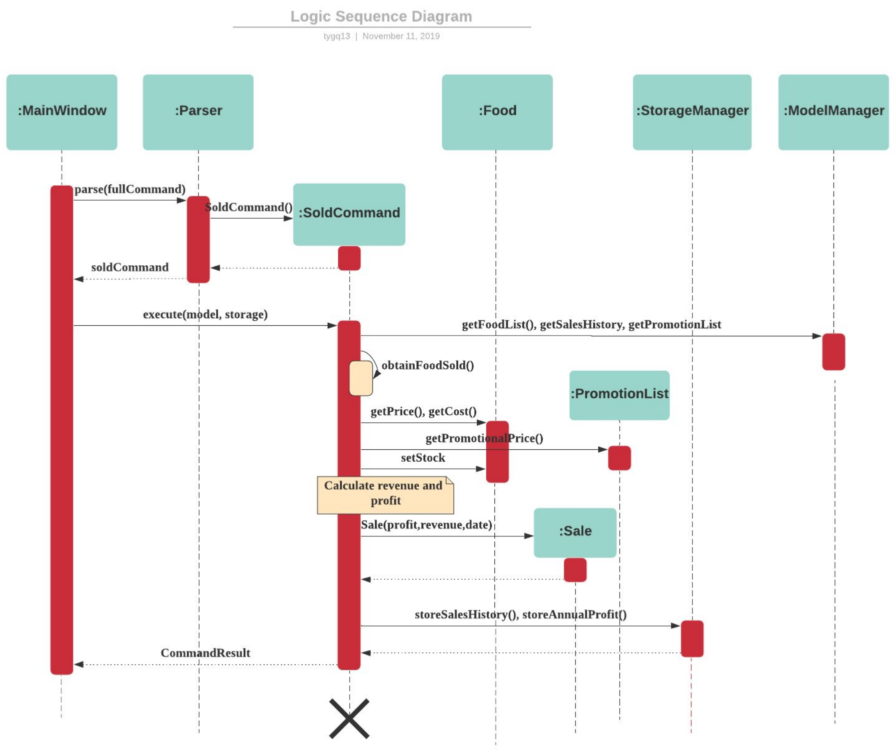


Figure 8: The UML sequence diagram for Sold command

Step 1: MainWindow invokes parser(). A new SoldCommand Object is created in Parser. MainWindow then executes SoldCommand
Step 2: SoldCommand obtains FoodList, PromotionList and SalesHistory from ModelManager.
Step 3: SoldCommand obtains the Food object to be sold and its price and cost.
Step 4: SoldCommand obtains the promotional price from PromotionList.
Step 5: SoldCommand calculate the profit and revenue and create a new Sale object.
Step 6 SoldCommand stores the Sale object in SalesHistory and stores the new stock using StorageManager.

7.4.2 Implementation Considerations

In the process of implementing the features, we have considered multiple ways to implement some features. The most noteworthy ones are profits and promotions. This section hence seeks to explain the rationale and considerations behind our choice of the implementation in the real project.

7.4.2.1 Implementation Considerations in Profit

Aspect: Store and generate the profit and revenue.

Alternative 1: Store the profit and revenue as static variables.

- Pros: This method provides fast access to the profits and revenue, with O(1) Time Complexity if accessing a specific food, or O(k) Time Complexity if accessing a food type, with k being the number of food in the food type. Besides, this method is also easier to implement, compared to Alternative 2.

- Cons: This method can only store the profit and revenue starting from a given point in time. This means that the profit and revenue are two numbers that keep accumulating ever since the very start of all sales entries.

Alternative 2: Generate the profit and revenue dynamically, with a specified time period.

- Pros: This method is very flexible, as it allows for generating profit and revenue with any specified time frame, from any start time to any end time.
- Cons: This method is slow, since it has to iterate through every single entry in the sales record. It hence has an O(n) Time Complexity, where n is the total number of entries in the sales history. Besides, this method is more complicated to implement, compared to Alternative 1.

After considering the two ways of implementation described above, the second way is finally chosen, with consideration of practicality in the real world. The flexibility it provides aside, it is more sensible and applicable in the real world scenario. In the context here, when referring to the profits and revenue, it should not refer to everything under the sun - the online food seller will not care about how much they have made ever since the first day they started the business, but rather, more concerned with how much they have made in a given time frame. For example, they may want to know how much they have made in the last few months, the last quarter, or the last year, per se, but not so much about the total running profits and revenue since Day 1. As such, although Alternative 1 will have a better access time, but it flies in the face of the purpose of the design here. Hence we decided to implement Alternative 2 for profit and revenue generation.

7.4.2.2 Implementation Considerations in Promotion

Aspect: Handle different promotions of the same food.

Alternative 1: Preventing the same product having two different promotions at the same time.

- Pros: It is easy to code and check.
- Cons: It does not support complex promotion rules which are increasingly common in real-life E-commerce platforms.

Alternative 2: Prompting the user to choose which promotion to use when updating sales history.

- Pros: It supports multiple promotions at the same time, thus being capable of handling more real life situations.
- Cons: It is rather complicated to implement the entire process of getting all available promotions, prompting the user, getting user feedback and executing.

After considering the two ways of implementation described above, the first method is chosen. As this is a student project with limited time to code, the developer team finds it more relevant to implement the most fundamental features first, rather than to spend time to improve one specific feature. This objective is also consistent with the very principle behind Amdahl's Law. Unlike the profits consideration, the current implementation of promotion still makes sense in the real world, except that it is less ideal than the other implementation. As such, the team agrees that it is more practical to implement the more complex promotion design paradigm in v2.0, rather than now (v1.4).

7.5 Implementation of File Storage Utilities (FileUtil)

One of the requirements expected from an inventory management system is to have a reliable and fast backend storage utility. This is to cater for shop owners who are managing a large amount of products within their online store, and would require Cube to be able to handle their products without a noticeable drop in performance.

7.5.1 Implementation Description

This section aims to detail how the backend storage utility, FileUtil, is being implemented in Cube. The activity diagram below shows a brief outline of how FileUtil is being utilised throughout the runtime of the application.

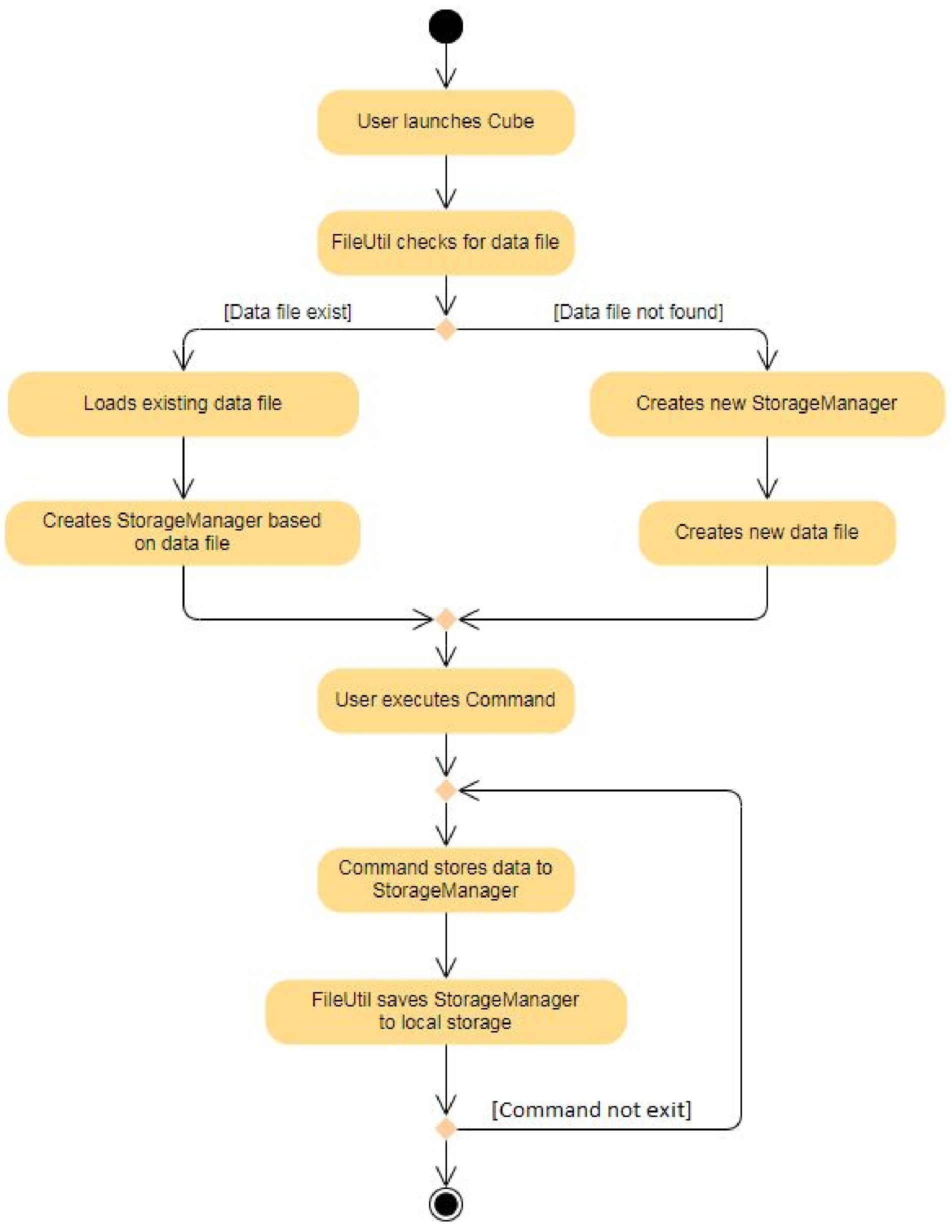

User launches Cube
FileUtil checks for data file
[Data file exist]
[Data file not found]
Loads existing data file
Creates StorageManager based on data file
Creates new StorageManager
Creates new data file
User executes Command
Command stores data to StorageManager
FileUtil saves StorageManager to local storage
[Command not exit]

Figure 9: Activity diagram on how FileUtil is being utilised in Cube

As we can see from the above diagram, FileUtil calls itself after execution of every command, and will save any changes that were made in the StorageManager into the local filesystem. This way, Cube can provide a sort of “auto-saving” feature where it will trigger the save function every time a command has been executed. Thus, in the unexpected event where the application crashes, the data stored within Cube is still present until the previous command that caused the crash, and saves the user of having the trouble to manually save the data file each time.

The `batch` command, which provides batch importing and exporting of food lists from a single CSV file, also utilises FileUtil by manually invoking the save and load capabilities of FileUtil each time whenever the `batch` command has been executed.

7.5.2 Design Considerations

This section shows some of the design considerations taken when implementing the File Storage Utilities.

Alternative 1: Using Serializable from the Java Standard Library to serialise objects into a file.

- Pros: It is easy to code and bugs are less likely to occur.
- Cons: There is a high performance cost when using Serializable, causing the program to slow down significantly over large amounts of data.

Alternative 2 (current design): Using an external library to help serialise data to an external file.

- Pros: It is extremely fast and can be easily integrated within the project.
- Cons: There is a learning curve when it comes to adopting a new library and unexpected bugs are highly like to occur.

After taking into consideration of the two ways of implementation as described above, we have decided to adopt Alternative 2 due to its high performance benefits. This is due to the FileUtil being one of the most commonly used functions in Cube, and we do not want to lose potential users of the product by restricting the number of products that they can manage within our application.

8. Documentation [coming in v2.0]

9. Dev Ops [coming in v2.0]

10. Appendices

Appendix A: User Stories

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

Priority	As a ...	I want to ...	So that I can...
`* * *`	Shop owner	See the revenue of certain or all products	Analyze the sales
`* * *`	Shop owner	Keep track of the quantity of each product sold	Determine the demand for each product

* * *	Shop owner	Know when my products are running low on stock	The product can be stocked up
* * *	Online food shop owner	Have different tabs and labels on different food categories	Easily identify and find the products
* * *	Online food shop owner	View a list of food available in a certain category	Choose a food based on its category
* * *	Online food shop owner	View the stock and storage information of food easily	The necessary actions accordingly
* * *	Shop owner	See the expiry date of all food in my stock	Sell them before they get spoiled
* *	Shop owner	Calculate the profits and losses I have made from my sales	Determine how much I have earned/how my sales are doing
* *	Shop owner	Provide my customers with promotions	Increase sales and ensure that products nearing their expiry date do not go to waste
* *	Online food shop owner concerned about profits	See the weekly profits earned by each specific categories of food sold	Determine what category of food I should promote
* *	Online food shop owner concerned about	Have functionality allowing for feedback to be collected specifically	Take action to improve my services

	customers' feedback	tagged to each specific item sold	
* *	Online food shop owner	Set different reminders for different categories of food sold	Take the necessary actions needed
* *	Online food shop owner concerned about expiry of stock	Have a countdown functionality to the expiry date of each item	Constantly update it
* *	Online food shop owner	Roll out special promotion plans on a weekly basis and set a reminder for that	Improve my sales
* *	Online shop owner	See the ID of every parcel sent out	Know the status of the parcels
* *	Online food shop owner	Be reminded to adjust the price of some food when they are going to expire	So that the food can be sold off and do not go to waste
* *	Online shop owner	Compare sale history of some products	Adjust my selling plan
* *	Online shop owner	Sort my products according to number left in stock	Decide whether to make a promotion or not
* *	Shop owner	Link supplier details to my products	Easily know where to order stocks when they are running low

* *	Lazy online seller	Set templates for my products’ descriptions	Run my shop without having to key in every single time
*	Lazy shop owner	Create alias or script for frequently used commands	Use the commands without needing to type them every time
*	Lazy shop owner	Batch upload information via a file	Upload information without needing to key in one by one manually
*	Customer	Know about how the food I bought should be stored	Ensure that the food does not get spoiled
*	Online food shop owner	See which food is the most popular last month	Increase the stock and/or price of the food
*	Shop owner	Know by when my products can be restocked	Inform customers who are interested in the product and so I don’t lose potential customers
*	Online food shop owner	Be reminded of the stock of certain food is running low	Replenish the stock
*	Sensitive seller	Keep my sales information secretive	Gain the trust of my customers
*	Food retailer	Be aware of any food allergens in my products	Warn my customers

*	Online shop owner	Advertise to all my followers once there is a new promotion	Inform them and find out if any of them are interested
*	Online shop owner	Know the items that are trending	Sell the items and compete with fellow sellers
*	Busy shop owner	Have a planner that help schedule all my meet-ups for the day	Optimize my time and meet more customers
*	Privacy concerned seller	Have the option to password-protected/encry pt	Place data sensitive information without worries
*	Online shop owner	See the list of my followers	I know how popular my store is
*	Shop owner	Have a customizable overview page where I can view critical information I want in just one tab/page	Quickly know the summary of my shop's status
*	User	Option to set user configurable preferences	For better customizability of my program

Appendix B: Use Cases

Use case 1: Add new product to stock
System: Cube
Actor: shop owner
Pre-condition: logged in

MSS:

1. Owner keys in a product name to add

2. System asks for details (number, type, price, colour, expiry date, if applicable)
3. Owner keys in details
4. System applies the addition and update inventory page.

Extensions:

- 2a. System has a list of pre-set templates of products and asks the owner whether he or she wants to choose from the templates.
- 4a. System asks whether the owner wants to apply the addition now or later at a designated time.

Use case 2: Finding food allergens in products
System: Cube
Actor: Shop Owner
Pre-condition: logged in

MSS:

1. Owner searches name of product.
2. Owner searches for 'Food Allergens' under the product.
3. System returns a list of food allergens found in the product.
4. Owner can search for a particular food allergen from the list.
5. System will return result of whether or not the food allergen is found in the product.

Alternate approach:

1. Owner searches the name of a particular food allergen
2. System returns a list of all products that may contain the searched food allergen.
3. Owner can then type in the name of one of the products within the list to get more information regarding the product (for example, other food allergens present in the product)

Use case 3: Enabling low stock reminders
System: Cube
Actor: Shop Owner
Pre-condition: logged in

MSS:

1. Owner navigates to alarms/notifications section.
2. System asks for a minimum quantity and which type of products to trigger notification.
3. Owner keys in details.
4. System applies notification settings on specified products.
5. System saves the settings.

Use case 4: Looking for supplier details
System: Cube
Actor: Shop Owner
Pre-condition: logged in

MSS:

1. Owner navigates to product inventory section.
2. Owner clicks on a specific products.
3. System presents several sections of product information to choose from.
4. Owner selects the supplier section.
5. System present owner with data of supplier linked to the product.

[more user cases coming in v2.0]

Appendix C: Non Functional Requirements

1. Cross platform support on popular operating systems with Java 11 or above installed.
2. Generally able to store 1000 products without a noticeable deterioration in application performance for typical usage.
3. Program should be generally free of bugs.

Appendix D: Glossary

Backend storage utility:
    The internal storage system of the application that is not visible to the user.

CLI:
    Command Line Interface.

Command Pattern:
    A design pattern that encapsulates information as an object.

Encapsulate:
    Packaging data into a single unit (e.g. an object).

Enumeration:
    A set of constant values. It can be considered as a data type.

GUI:
    Graphical User Interface.

Json file:
    A file that stores objects and data structures in 'JavaScript Object Notation' (Json) format.

n-tier architecture:
    A multi-tier architecture in which data management, processing and presentation functions are physically separated.

Parser:
    A software component that uses input data to build a data structure (often in the form of an abstract syntax tree or parse tree.

ParserUtil:
    A library that provides utility functions for parsing.

Polymorphism:
    The ability of an object to take many forms based on similar input messages.

Principle of separation of concerns:
    "To achieve better modularity, separate the code into distinct sections, such that each section addresses a separate concern." — Proposed by Edsger W. Dijkstra

Sequence Diagram:

A diagram that depicts the interaction between multiple objects for a particular event scenario.

Time complexity:

The amount of time taken for an algorithm to run.

Ui:

User interface.

Appendix E: Instructions for Manual Testing

1. Setting up

Test Case:

a. Download jar file and copy it into an empty folder.
b. Double click the jar file.

Expected:

a. Cube starts up correctly and shows the GUI window.
b. Cube automatically creates a data folder and a cube.log file under the same folder as the jar file.

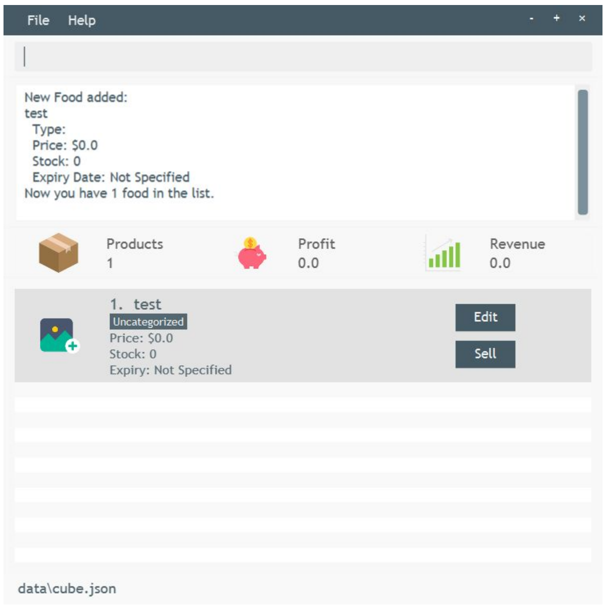
※Please refer to Section 4. Setting up for more detailed instructions.

2. Testing for add

2.1 Correct input

Test Case: `add test`

Expected:



```
New Food added:
test
  Type:
  Price: $0.0
  Stock: 0
  Expiry Date: Not Specified
Now you have 1 food in the list.
```

1. test
Uncategorized
Price: $0.0
Stock: 0
Expiry: Not Specified

data\cube.json

2.2 Repeated food name

Test Case: `add test`

※Assuming that the tester has already added food ‘test’ in 2.1

Expected:

Cube will give an error message “OOPS!!! The food already exists”

2.3 Invalid parameter

Test Case: `add invalid -x test`

Expected:

Cube will give an error message “OOPS!!! Your input contains invalid parameter.”

2.4 Invalid integer

Test Case: `add invalid -s 1.1`

Expected:

Cube will give an error message “OOPS!!! The number inside input should only be non-negative integer and less than 10000.00.”

2.5 Invalid number

Test Case: `add invalid -p 1000000000`

Expected:

Cube will give an error message “OOPS!!! The number inside input should only be non-negative numerical and less than 10000.00.”

2.6 Date before today

Test Case: `add invalid -e 1/1/2019`

Expected:

Cube will give an error message “OOPS!!! The food expiry date cannot be before today”

2.7 Invalid date format

Test Case: `add invalid -e 1-12-2019`

Expected:

Cube will give an error message “OOPS!!! The date is invalid. Please specify an existent date in 'dd/mm/yyyy'"”

2.8 Empty field value

Test Case: `add invalid -e`

Expected:

Cube will give an error message “OOPS!!! Your input after a parameter is empty.”

2.9 Not enough parameter

Test Case: `add`

Expected:

Cube will give an error message “OOPS!!! The parameter you input is not enough”

2.10 Repeated parameter

Test Case: `add invalid -p 1 -p 2`

Expected:

Cube will give an error message “OOPS!!! Your input contains repetitive parameter”

3. Testing for find

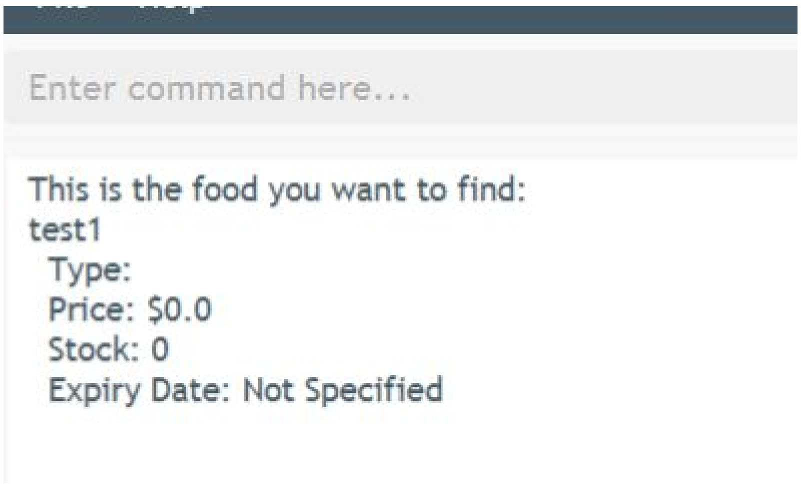
※Pre-requisites: copy and paste these inputs line by line first

```
delete -all
add test1
add test2
add test4 -t test
add test3 -t test
```

3.1 Finding food with index

Test Case: `find -i 1`

Expected:

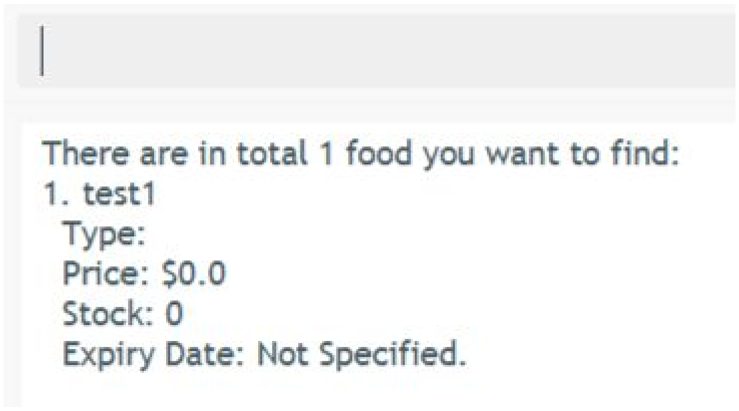


```
This is the food you want to find:
test1
  Type:
  Price: $0.0
  Stock: 0
  Expiry Date: Not Specified
```

3.2 Finding food with name

Test Case: `find -n test1`

Expected:

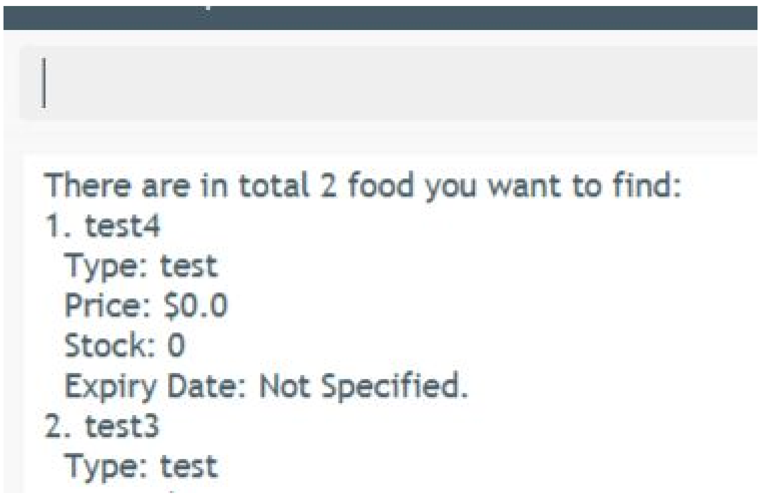


```
There are in total 1 food you want to find:
1. test1
  Type:
  Price: $0.0
  Stock: 0
  Expiry Date: Not Specified.
```

3.3 Finding food with type

Test Case: `find -t test`

Expected:

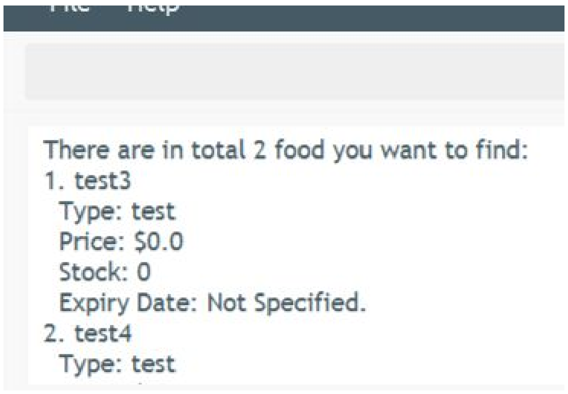


```
There are in total 2 food you want to find:
1. test4
  Type: test
  Price: $0.0
  Stock: 0
  Expiry Date: Not Specified.
2. test3
  Type: test
```

3.4 Finding food with type and sort by name

Test Case: find -t test -sort name

Expected:



```
There are in total 2 food you want to find:
1. test3
  Type: test
  Price: $0.0
  Stock: 0
  Expiry Date: Not Specified.
2. test4
  Type: test
```

3.5 Invalid index

Test Case: find -i 5

Expected:

Cube will give an error message “OOPS!!! The index is out of the range of food list”

3.6 Invalid name

Test Case: find -n test5

Expected:

Cube will give an error message “OOPS!!! The food does not exists”

3.7 Invalid type

Test Case: `find -t invalid`

Expected:
Cube will give an error message “OOPS!!! The food type does not exist”

3.8 Invalid sort type

Test Case: `find -t test -sort invalid`

Expected:
Cube will give an error message “OOPS!!! The sort type can only be expiry/name/stock.”

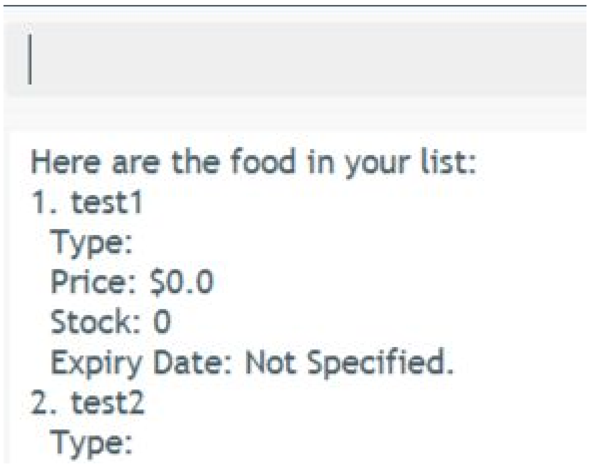
4. Testing for list

※Pre-requisites:
The tester has already copied and pasted the inputs in 3. Testing for find into Cube.

4.1 List all products

Test Case: `list`

Expected:

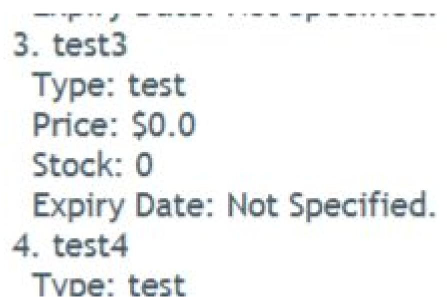


```
Here are the food in your list:
1. test1
  Type:
  Price: $0.0
  Stock: 0
  Expiry Date: Not Specified.
2. test2
  Type:
```

4.2 List products and sort by name

Test Case: `list -sort name`

Expected:
Note that test3 and test4 will change place.



```
3. test3
  Type: test
  Price: $0.0
  Stock: 0
  Expiry Date: Not Specified.
4. test4
  Type: test
```

4.3 Invalid sort type

Test Case: `list -sort invalid`

Expected:

Cube will give an error message "OOPS!!! The sort type can only be expiry/name/stock."

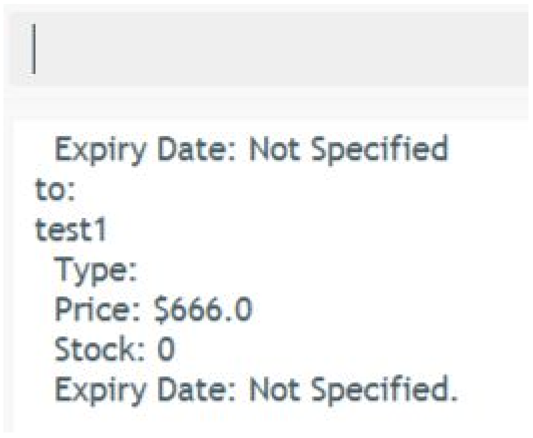
5. Testing for update

※Pre-requisites:
The tester has already copied and pasted the inputs in 3. Testing for find into Cube.

5.1 Correct input

Test Case: `update test1 -p 666`

Expected:



```
  Expiry Date: Not Specified
to:
test1
  Type:
  Price: $666.0
  Stock: 0
  Expiry Date: Not Specified.
```

5.2 Invalid food name

Test Case: `update test5 -p 666`

Expected:

Cube will give an error message "OOPS!!! The food does not exists"

5.3 Not enough parameter

Test Case: `update test1`

Expected:

Cube will give an error message “OOPS!!! The parameter you input is not enough”

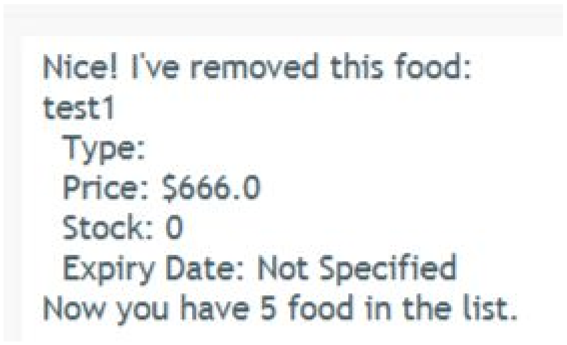
6. Testing for delete

※Pre-requisites: In addition to the inputs in 3. Testing for find
add test5
add test6

6.1 Delete by index

Test Case: `delete -i 1`

Expected:

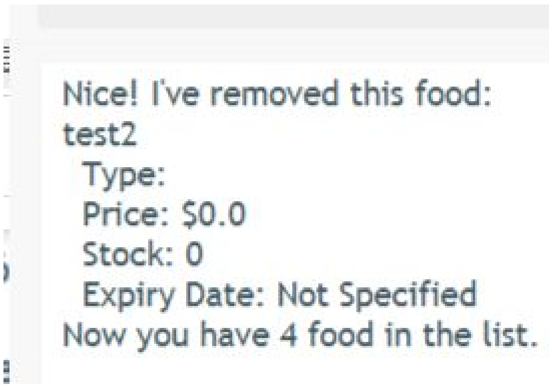


```
Nice! I've removed this food:
test1
  Type:
  Price: $666.0
  Stock: 0
  Expiry Date: Not Specified
Now you have 5 food in the list.
```

6.2 Delete by name

Test Case: `delete -n test2`

Expected:

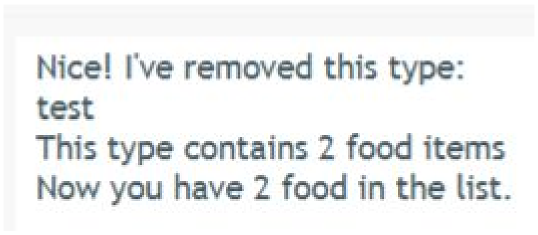


```
Nice! I've removed this food:
test2
  Type:
  Price: $0.0
  Stock: 0
  Expiry Date: Not Specified
Now you have 4 food in the list.
```

6.3 Delete by type

Test Case: `delete -t test`

Expected:

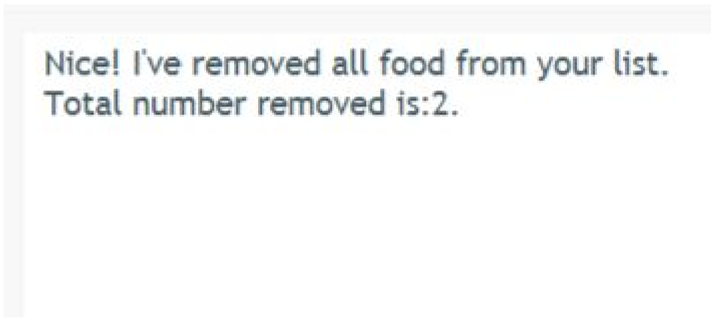


```
Nice! I've removed this type:
test
This type contains 2 food items
Now you have 2 food in the list.
```

6.4 Delete all

Test Case: `delete -all`

Expected:



```
Nice! I've removed all food from your list.
Total number removed is:2.
```

6.5 Invalid index

Test Case: `delete -i 1`

Expected:
Cube will give an error message “OOPS!!! The index is out of the range of food list”

6.6 Invalid name

Test Case: `delete -n test1`

Expected:
Cube will give an error message “OOPS!!! The food does not exists”

6.7 Invalide deletion type

Test Case: `delete -x invalid`

Expected:
Cube will give an error message “OOPS!!! Your input contains invalid parameter.”

7. Testing for Reminder

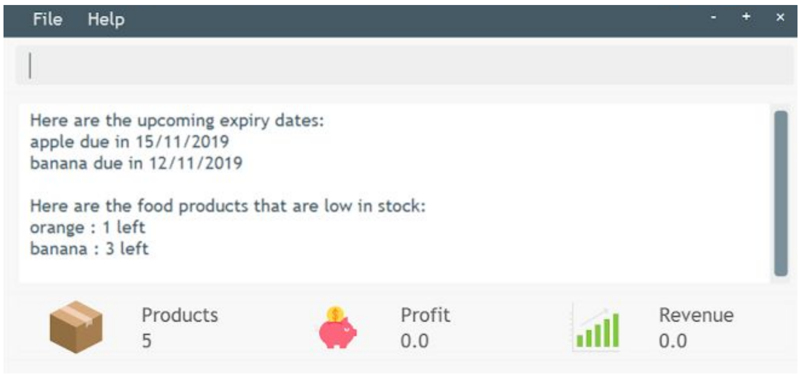
※Pre-requisites: copy and paste these inputs line by line first

```
add apple -s 25 -e 15/11/2019
add orange -s 1 -e 25/12/2019
add banana -s 3 -e 12/11/2019
add strawberry -s 150 -e 03/03/2020
add pear -s 15 -e 22/11/2019
```

7.1 Reminder with default values

Test Case: `reminder`

Expected:



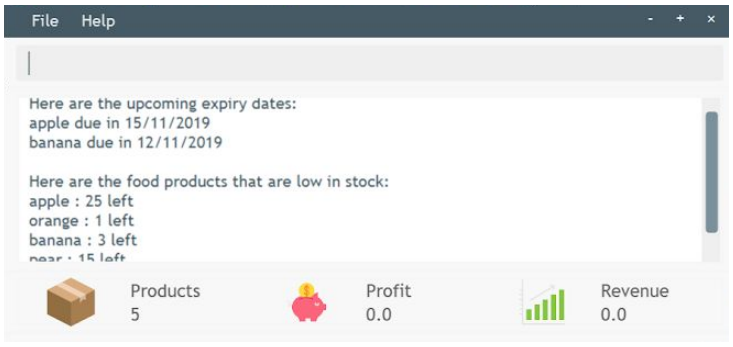
```
Here are the upcoming expiry dates:
apple due in 15/11/2019
banana due in 12/11/2019

Here are the food products that are low in stock:
orange : 1 left
banana : 3 left
```

7.2 Reminder with customized stock

Test Case: `reminder -s 50`

Expected:



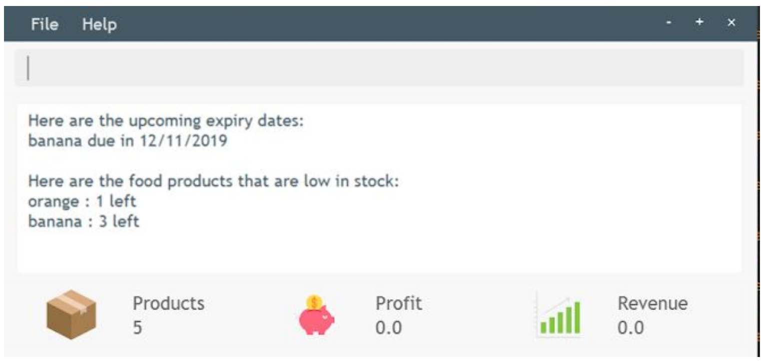
```
Here are the upcoming expiry dates:
apple due in 15/11/2019
banana due in 12/11/2019

Here are the food products that are low in stock:
apple : 25 left
orange : 1 left
banana : 3 left
pear : 15 left
```

7.3 Reminder with customized remaining day

Test Case: `reminder -d 3`

Expected:

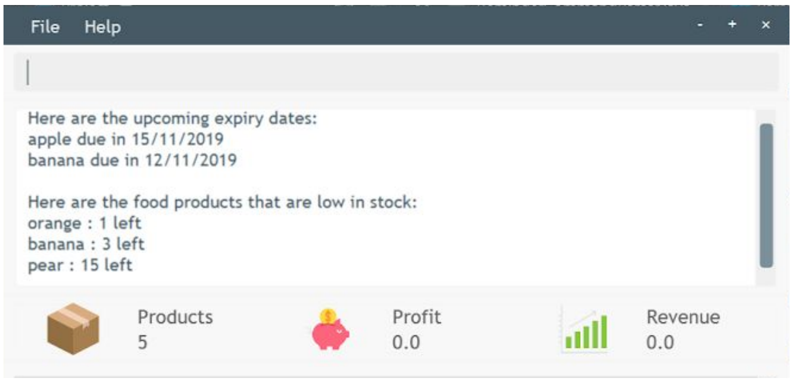


```
Here are the upcoming expiry dates:
banana due in 12/11/2019

Here are the food products that are low in stock:
orange : 1 left
banana : 3 left
```

7.4 Reminder with customized stock and remaining day

Test Case: `reminder -d 10 -s 20`
Expected:



```
Here are the upcoming expiry dates:
apple due in 15/11/2019
banana due in 12/11/2019

Here are the food products that are low in stock:
orange : 1 left
banana : 3 left
pear : 15 left
```

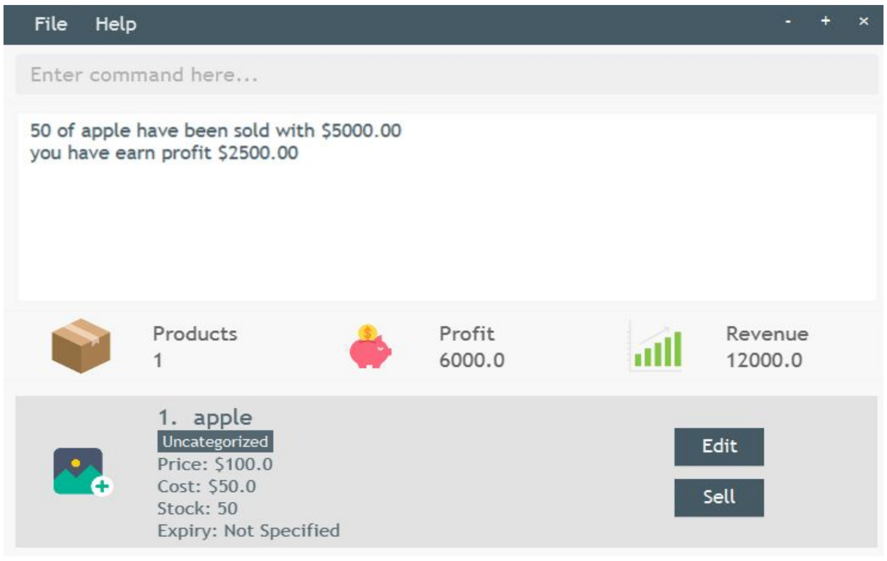
8. Testing for Sold

※Pre-requisites: copy and paste these inputs line by line first
delete -all
add apple -p 100 -c 50 -c 100
add banana -q 0.2 -c 0.1 -s 2

8.1 Correct input

Test Case: `sold apple -q 50 -t 25/12/2019`

Expected:



```
50 of apple have been sold with $5000.00
you have earn profit $2500.00
```

```
1. apple
Uncategorized
Price: $100.0
Cost: $50.0
Stock: 50
Expiry: Not Specified
```

8.2 Not enough parameters

Test Case: `sold`
Test Case: `sold apple`

Expected:
Cube will give an error message "OOPS!!! The parameter you input is not enough"

8.3 Invalid foodname

Test Case: `sold anything -q 1`

Expected:
Cube will give an error message "OOPS!!! The food does not exists"

8.4 Invalid quantity

Test Case: `sold apple -q 1000`

Expected:
Cube will give an error message "OOPS!!! The quantity sold is negative or too large"

Test Case: `sold apple -q -3`

Expected:
Cube will give an error message "OOPS!!! OOPS!!! Your input contains invalid parameter."

Explanation:
"-3" is parsed as a parameter.

Test Case: `sold apple -q 3.3`

Expected:
Cube will give an error message "OOPS!!! The number inside input should only be non-negative integer and less than 10000.00."

Explanation:
"3.3" is parsed as a string and subsequently treated as invalid quantity.

8.5 Invalid date

※ Future date is considered as valid since it can be a pre-ordered sale.

Test Case: `sold apple -q 1 -t ok`

Expected:
Cube will give an error message "OOPS!!! The date is invalid. Please specify an existent date in 'dd/mm/yyyy'"

Test Case: `sold apple -q 1 -t 29/2/2019`

Expected:
Cube will give an error message "OOPS!!! The date is invalid. Please specify an existent date in 'dd/mm/yyyy'"

Explanation: The date is non-existent since there is no 29th Feb in 2019.

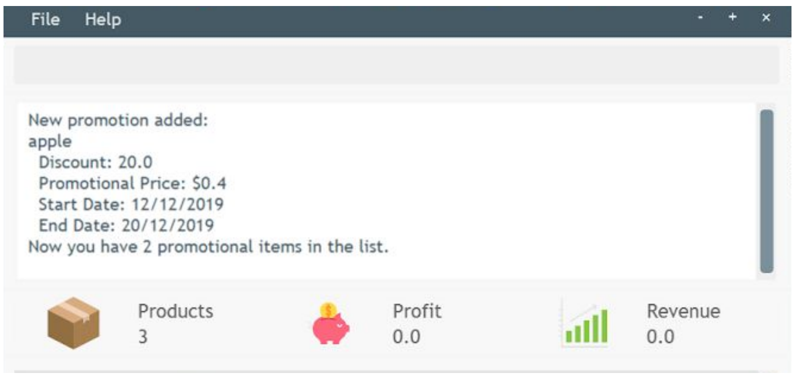
9. Testing for Promotion

※Pre-requisites: copy and paste these inputs line by line first

```
delete -all
add apple -p 2 -s 100
add pear -p 0 -s 100
add banana -p 4 -s 100
promotion banana -% 50 -s 12/12/2019 -e 20/12/2019
```

9.1 Correct input

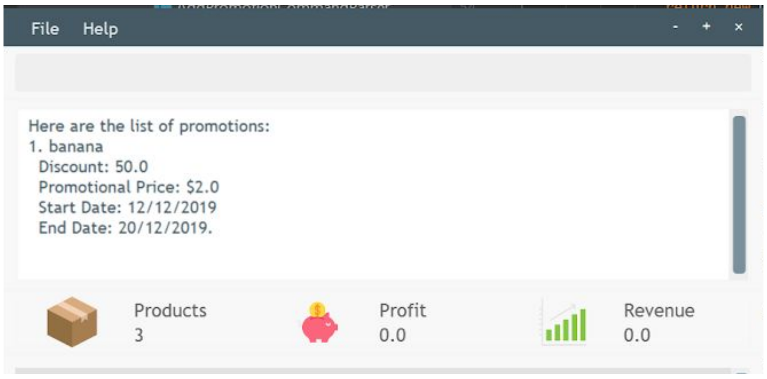
Test Case: `promotion apple -% 20 -s 12/12/2019 -e 20/12/2019`
Expected:



```
New promotion added:
apple
  Discount: 20.0
  Promotional Price: $0.4
  Start Date: 12/12/2019
  End Date: 20/12/2019
Now you have 2 promotional items in the list.
```

9.2 View promotion list

Test Case: `promotion -list`
Expected:



```
Here are the list of promotions:
1. banana
  Discount: 50.0
  Promotional Price: $2.0
  Start Date: 12/12/2019
  End Date: 20/12/2019.
```

9.3 Invalid foodname

Test Case: `promotion orange -% 20 -s 11/12/2019 -e 30/12/2019`
Expected:
Cube will give an error message “OOPS!!! The food does not exist.”

9.4 Invalid date

Test Case: `promotion apple -% 20 -s 11/11/2018 -e 11/11/2019`
Expected:
Cube will give an error message “OOPS!!! The dates cannot be before today.”

Test Case: `promotion apple -% 20 -s 22/11/2019 -e 15/11/2019`
Expected:
Cube will give an error message “OOPS!!! The end date cannot be before the start date.”

9.5 Overlapping promotion period

Test Case: `promotion banana -% 10 -s 13/12/2019 -e 15/12/2019`
Expected:

Cube will give an error message “OOPS!!! There is already a promotion for the same food in this period.”

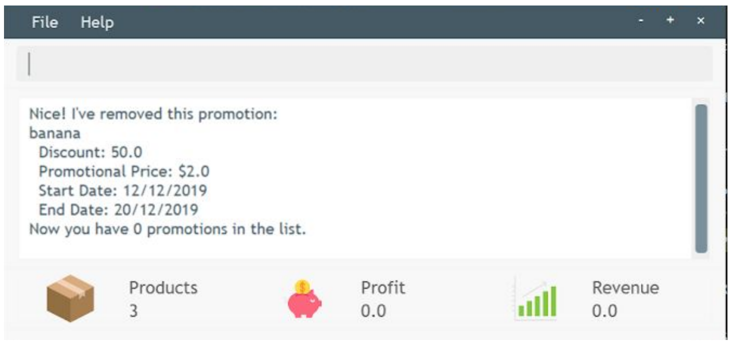
9.6 Free item

Test Case: `promotion pear -% 30 -s 13/12/2019 -e 15/12/2019`
Expected:

Cube will give an error message “OOPS!!! The item you are going to promote is already free.”

9.7 Delete promotion

Test Case: `promotion -delete 1`
Expected:



```
Nice! I've removed this promotion:
banana
  Discount: 50.0
  Promotional Price: $2.0
  Start Date: 12/12/2019
  End Date: 20/12/2019
Now you have 0 promotions in the list.
```

10. Testing for Profit

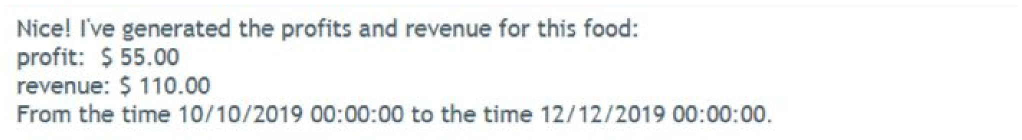
※Pre-requisites: copy and paste these inputs line by line first

```
delete -all
add apple -p 2.2 -c 1.1 -s 100
Add banana -p 100 -c 50 -s 100
sold apple -q 50
sold apple -q 30 -t 3/3/2019
sold banana -q 50
```

10.1 Correct Input

Test Case: `profit -t1 10/10/2019 -t2 12/12/2019 -n apple`

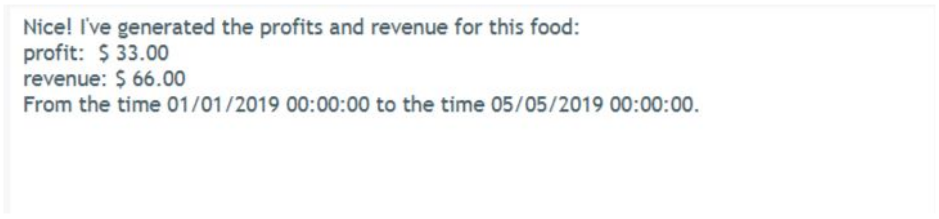
Expected:



```
Nice! I've generated the profits and revenue for this food:
profit:  $ 55.00
revenue: $ 110.00
From the time 10/10/2019 00:00:00 to the time 12/12/2019 00:00:00.
```

Test Case: `profit -t1 1/1/2019 -t2 5/5/2019 -n apple`

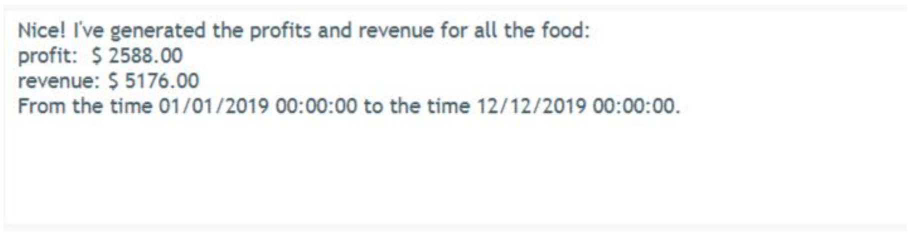
Expected:



```
Nice! I've generated the profits and revenue for this food:
profit:  $ 33.00
revenue: $ 66.00
From the time 01/01/2019 00:00:00 to the time 05/05/2019 00:00:00.
```

Test Case: `profit -t1 1/1/2019 -t2 12/12/2019 -all`

Expected:



```
Nice! I've generated the profits and revenue for all the food:
profit:  $ 2588.00
revenue: $ 5176.00
From the time 01/01/2019 00:00:00 to the time 12/12/2019 00:00:00.
```

11. Testing for Batch

※Pre-requisites: copy and paste these inputs line by line first
delete -all
add test

11.1 Batch export

Test Case: `batch -o test`

Expected:

a. Cube will show a success message "The product list has been successfully exported as file".
b. Go to date folder under the same directory as the jar file. You should be able to see a file named "test" there.

11.2 Batch import

Test Case:

a. `delete -all`
b. `batch -i test`

※Assuming that the file is not moved after 11.1

Expected:

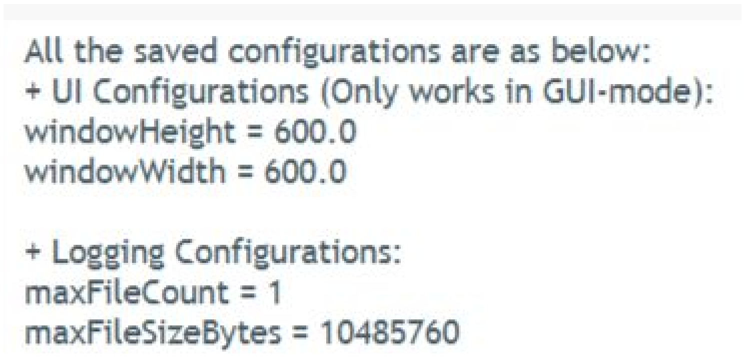
a. Cube will show a success message "The product list has been successfully imported as file".
b. Your list now should show the food item "test".

12. Testing for Config

12.1 See config list

Test Case: config

Expected:



```
All the saved configurations are as below:
+ UI Configurations (Only works in GUI-mode):
windowHeight = 600.0
windowWidth = 600.0

+ Logging Configurations:
maxFileCount = 1
maxFileSizeBytes = 10485760
```

12.2 Config UI

Test Case: config UI -h 700 -w 1000

Expected:

a. Cube will show a success message “The UI settings has been configured successfully”.
b. Your GUI window should become taller and wider now.

13. Testing for Help

13.1 Using command line

Test Case: help
Expected:
The GUI window will show a list of commands and corresponding formats.

13.2 Using GUI

Test Case:
Click the ‘help’ button at the top of the window.
Expected:
‘help’ command will automatically appear in the command box. Press enter to see the list of commands.

14. Testing for Exit

14.1 Using command line

Test Case: `exit` / `quit` / `bye`
Expected:
The GUI window automatically shuts down.

14.2 Using GUI

Test Case:
Click the ‘X’ button at the top right most of the window.
Expected:
The GUI window automatically shuts down.

15. Testing for Invalid Command

Test Case: `test`
Expected:
Cube will give an error message “OOPS!!! The command is invalid. Enter 'help' to view the list of command”