Diet Manager / Version 2

Diet Manager Design Document

Bits and Bytes

Zachary “Bubba” Lichvar <znl2181@rit.edu>
Brennan Jackson <btj9560@rit.edu>
Brandon Connors <bdc5435@rit.edu>
Rosis Sharma <rs9110@rit.edu>
Visalakshi Nutulapati <vxn1520@rit.edu>

Project Summary

The application outlined in this document details a Diet Manager application which will take input from a user to assist with the user's health goals. The application will take the users weight and calorie goals as well the users caloric input in the form of the food that they eat and the recipes that they use. This application will compute whether or not the user is on a path to obtaining their goals.

The user will log the food that they eat as well as their weight. This will allow the user to obtain retrospective analysis towards their progress. This is used to better help the user understand the consequences of their actions and how the food that they eat affects their health. This means that if a user has a high caloric intake and the user is gaining weight they will be able to see the correlation between these two events. The inverse is also true; if the user has a low caloric input and is losing weight they will be able to see the correlation between their intake of low-calorie food and their weight loss.

The goal of this application is not to substitute for professional medical advice. It is only to provide a means to provide a user with a retrospective log which might show the user insights into their activities and how it affects their overall health. It will hopefully motivate users towards a healthy lifestyle by “gaming” the user into achieving their own goals.

Design Overview

It was important for the software engineering team to incorporate best practices when designing the Diet Manager application. That is why the software engineering team insisted on using design patterns that promote high cohesion and low coupling as well as separating the concerns between the different software modules based on the individual aspects of their functionality.

The overall architecture of the software uses the Model-View-Controller (MVC) architectural pattern. This allows different components of the MVC to be independently substituted and/or upgraded in the future without having to refactor other components. This will reduce the technical debt that is associated with extensibility and maintainability of the code base.

It was clear that the handling of data in our Controller class would be of the utmost importance for this type of application. That is why we decided from the beginning to use factory patterns and dependency inversion in order to handle the input and output of the data from the underlying file system. For example, the user will create an Entry into their log containing relevant information. When that event takes place the EntryHandler class containing the values of the user's Entry will call the IOHandler to write the information to the log.csv file. When a user adds data to the file, it will update the data in the respective file. Each of the Entries will use the dependency inversion principle to supply the IOHandler with their information.

Data objects in our Model such as CalorieEntry, FoodEntry, WeightEntry, BasicFood, and Recipe implement a common interface of ICSVable. The implementers of this interface must provide a way for their internal data to be represented as CSV output. Using the dependency inversion principle in this way means that the IOHandler does not need to know how to write the respective data from each entry, it simply gets the data from the class to write. This helps promote separation of concerns.

In the inverse operation of reading data from the file to build our model, we use factory patterns to create data objects that are part of the model component within EntryHandler and FoodHandler classes. Using this pattern in this way means that the IOHandler will be able to read a provided file and then the handler of that type of data will use its factory to instantiate objects that represent that data.

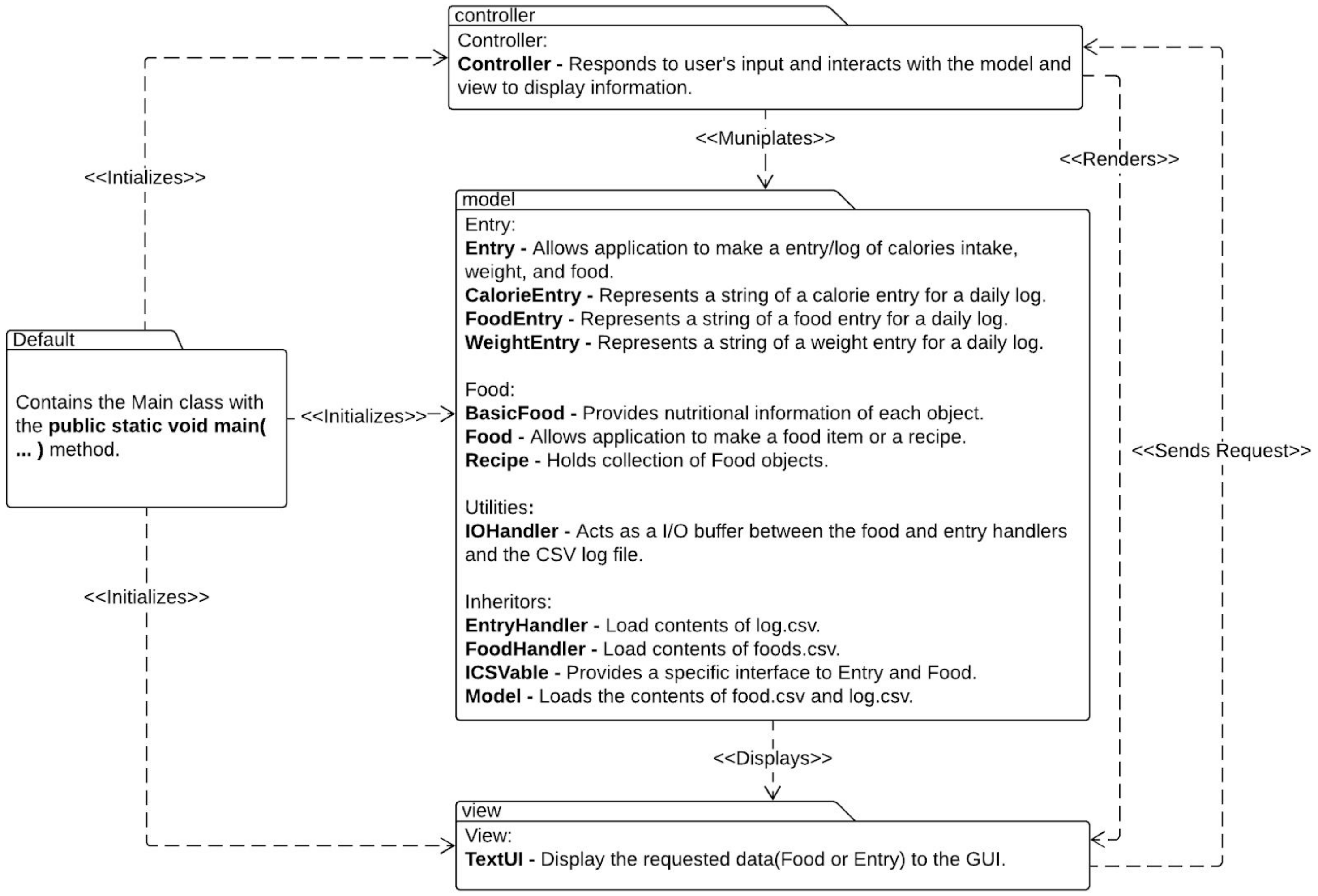
In order to provide the user with their retrospective analysis of their entries, it was necessary to have the Entry class implement a comparable interface by our use of Gregorian

Calendar. This way the data can be stored in a data structure that respects their natural ordering. Because data will be stored in its natural order the EntryHandler will not need to know how to order the entries based on their dates. This further promotes separation of concern in regards to how the EntryHandler will order the individual Entries.

It was clear from the beginning that relationships between Food objects (Recipe and BasicFood) is a composite pattern where a Recipe and a BasicFood are both Food objects. But Recipe objects are an aggregation of individual BasicFood objects. Since the user will be interested in the total calories, fat, protein, and carbohydrates in each Recipe, using the composite pattern here means that we can total these values from all of the individual BasicFoods that compose the Recipe. This also maintains our ability to add BasicFood types individually which may, or may not be part of a Recipe object.

The software engineering teams goal is to promote extensibility and maintainability for the foreseeable future. In order to achieve this, the principles of high cohesion and low coupling, along with the separation of concerns and dependency inversion of our classes were implemented through various design patterns. The document below details the design patterns used in order to succeed in meeting the goals set forth.

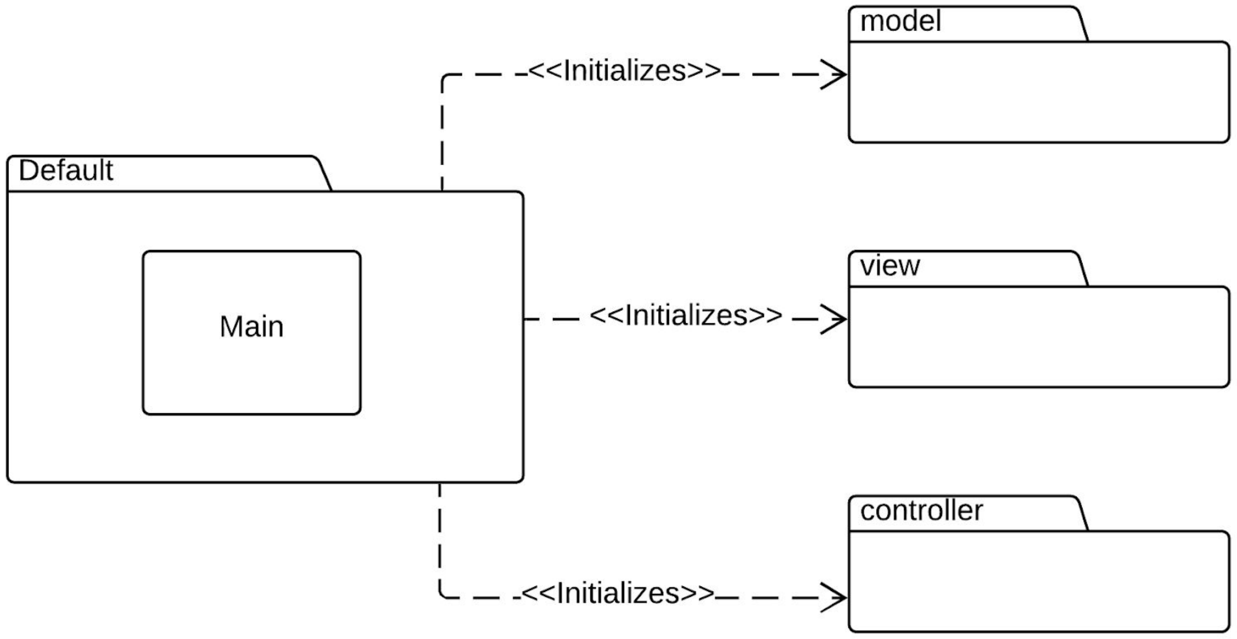
Subsystem Structure


controller
Controller:
Controller - Responds to user's input and interacts with the model and view to display information.
<<Muniplates>>
<<Renders>>
<<Intializes>>
model
Entry:
Entry - Allows application to make a entry/log of calories intake, weight, and food.
CalorieEntry - Represents a string of a calorie entry for a daily log.
FoodEntry - Represents a string of a food entry for a daily log.
WeightEntry - Represents a string of a weight entry for a daily log.
Food:
BasicFood - Provides nutritional information of each object.
Food - Allows application to make a food item or a recipe.
Recipe - Holds collection of Food objects.
Utilities**:**
IOHandler - Acts as a I/O buffer between the food and entry handlers and the CSV log file.
Inheritors:
EntryHandler - Load contents of log.csv.
FoodHandler - Load contents of foods.csv.
ICSVable - Provides a specific interface to Entry and Food.
Model - Loads the contents of food.csv and log.csv.
Default
Contains the Main class with the **public static void main(...)** method.
<<Initializes>>
<<Sends Request>>
<<Initializes>>
<<Displays>>
view
View:
TextUI - Display the requested data(Food or Entry) to the GUI.

Subsystem CRCs

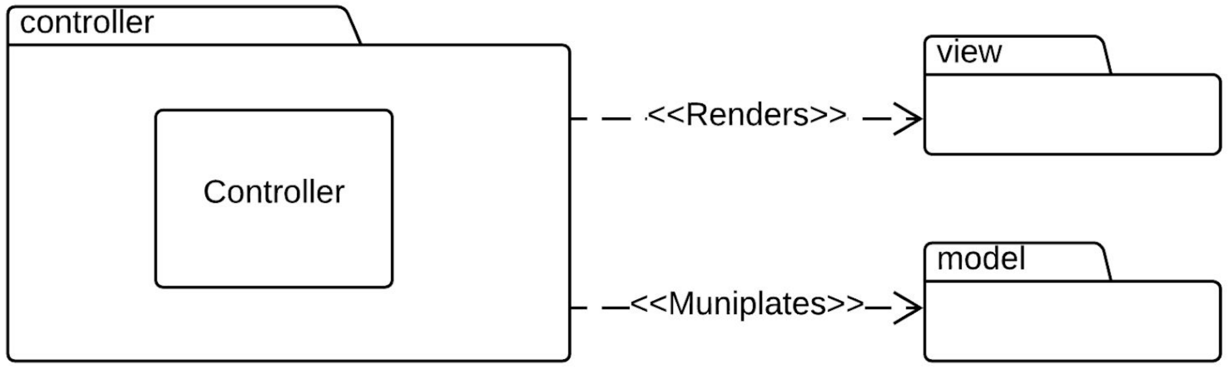
Default Subsystem

Class Main	
Responsibilities	instantiate a TextUI, instantiates a Model, instantiates an EntryHandler(Model, IOHandler), FoodHandler(Model, IOHandler, and a Controller(TextUI, EntryHandler, FoodHandler)
Collaborators	TextUI, FoodHandler, EntryHandler, Controller, IOHandler



Controller Subsystem

Class Controller	
Responsibilities	Responding to user input, retrieving information, interacting with the model and controlling the view to display the proper information
Collaborators	FoodHandler, EntryHandler, IOHandler, TextUI



Model Subsystem

Class FoodHandler	
Responsibilities	Load contents of foods.csv into a recipe hashmap and basic food hashmap. The models can be retrieved from the hashmaps by providing FoodHandler with their unique name.
Collaborators (uses)	FoodFactory - creates and returns food object based on provided unique string. “b” for basic food and “r” for recipe. IOHandler - Acts as a I/O buffer between the food and entry handlers and the CSV log file.

Class EntryHandler	
Responsibilities	Load contents of log.csv into a calorie entry,weight entry and food entry treemaps. The models can be retrieved from the treemaps by providing EntryHandler with their unique Date key.
Collaborators (uses)	EntryFactory - creates and returns entry object based on provided unique string. “w” for weight entry,“c” for calorie entry and “f” for food entry. IOHandler - Acts as a I/O buffer between the food and entry handlers and the CSV log file.

Class IOHandler	
Responsibilities	Acts as a I/O buffer between the food and entry handlers and the CSV log file.
Collaborators	EntryHandler - Calls IOHandler to retrieve csv log FoodHandler - Calls IOHandler to retrieve csv log

Class Recipe	
Responsibilities	Holds collection of Food objects Allows objects to be iterated through Allows objects to be added or removed from collection

Collaborators (uses)	Food - Allows application to make a food item or a recipe. BasicFood - Provides nutritional information of each object.

Class BasicFood	
Responsibilities	Provides nutritional information of each object.
Collaborators (uses)	Food - Allows application to make a food item or a recipe. Recipe - Holds collection of Food objects.

Class FoodFactory	
Responsibilities	Produce a formatted string with food data Creates and returns food object based on provided unique string "b" for basic food and "r" for recipe
Collaborators	Food - Allows application to make a food item or a recipe.

Class EntryFactory	
Responsibilities	Produce a formatted string with entry/log data Creates and returns entry object based on provided unique string "w" for weight entry,"c" for calorie entry and "f" for food entry
Collaborators	Entry - Allows application to make a entry/log of calories intake, weight, and food.

Class CalorieEntry	
Responsibilities	Represents a string of a calorie entry for a daily log.
Collaborators (uses)	Entry - Allows application to make a entry/log of calories intake, weight, and food.

Class FoodEntry	
Responsibilities	Represents a string of a food entry for a daily log.
Collaborators (uses)	Entry - Allows application to make a entry/log of calories intake, weight, and food. Food - Allows application to make a food item or a recipe.

Class WeightEntry	
Responsibilities	Represents a string of a weight entry for a daily log.
Collaborators (uses)	Entry - Allows application to make a entry/log of calories intake, weight, and food.

Class Food	
Responsibilities	Allows application to make a food item or a recipe.
Collaborators (inherits)	Recipe - Holds collection of Food objects. BasicFood - Provides nutritional information of each object.

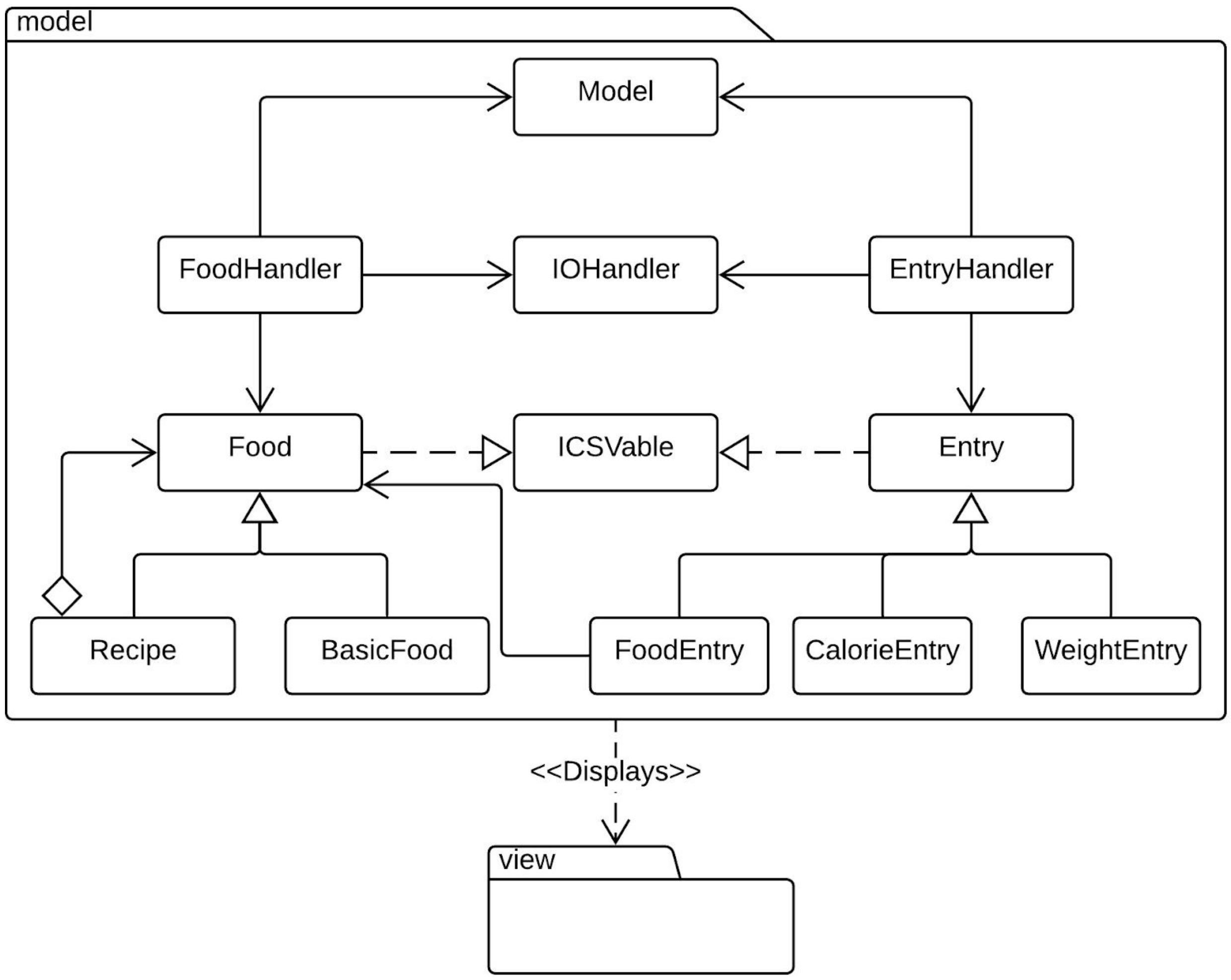
Class Entry	
Responsibilities	Allows application to make a entry/log of calories intake, weight, and food.
Collaborators (inherits)	CalorieEntry - Represents a string of a calorie entry for a daily log. WeightEntry - Represents a string of a weight entry for a daily log. FoodEntry - Represents a string of a food entry for a daily log.

Class ICSVable (interface)	
Responsibilities	Provide a specific interface to Entry and Food Can retrieve a formatted CSV string from Entry and Food Retrieves type of string from Entry and Food

Class FoodHandler	
Responsibilities	Load contents of foods.csv into a recipe hashmap and basic food hashmap. The models can be retrieved from the hashmaps by providing FoodHandler with their unique name.
Collaborators (uses)	FoodFactory - creates and returns food object based on provided unique string. “b” for basic food and “r” for recipe. IOHandler - Acts as a I/O buffer between the food and entry handlers and the CSV log file.

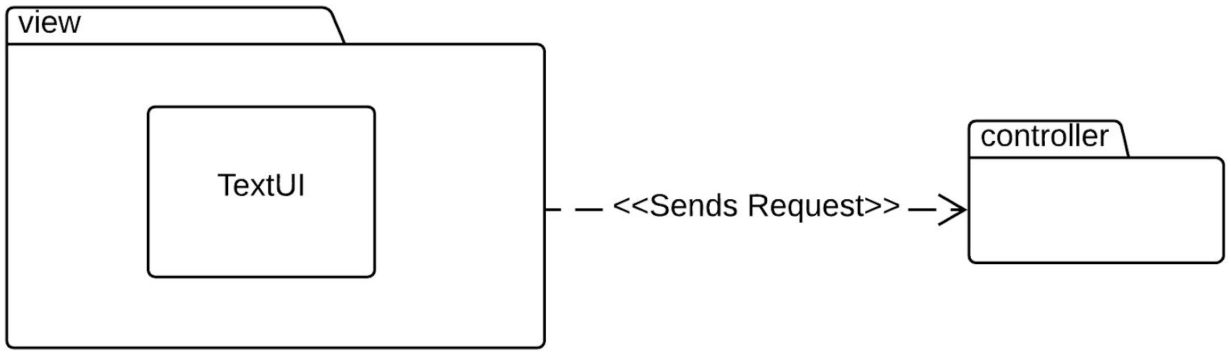
Class EntryHandler	
Responsibilities	Load contents of log.csv into a calorie entry,weight entry and food entry treemaps. The models can be retrieved from the treemaps by providing EntryHandler with their unique Date key.
Collaborators (uses)	EntryFactory - creates and returns entry object based on provided unique string. “w” for weight entry,“c” for calorie entry and “f” for food entry. IOHandler - Acts as a I/O buffer between the food and entry handlers and the CSV log file.

Class Model	
Responsibilities	Model the contents of the foods.csv and log.csv being able to add and get contents into and from the file
Collaborators (uses)	foods.csv, logs.csv, FoodHandler, EntryHandler, Controller


model
Model
FoodHandler
IOHandler
EntryHandler
Food
ICSVable
Entry
Recipe
BasicFood
FoodEntry
CalorieEntry
WeightEntry
<<Displays>>
view

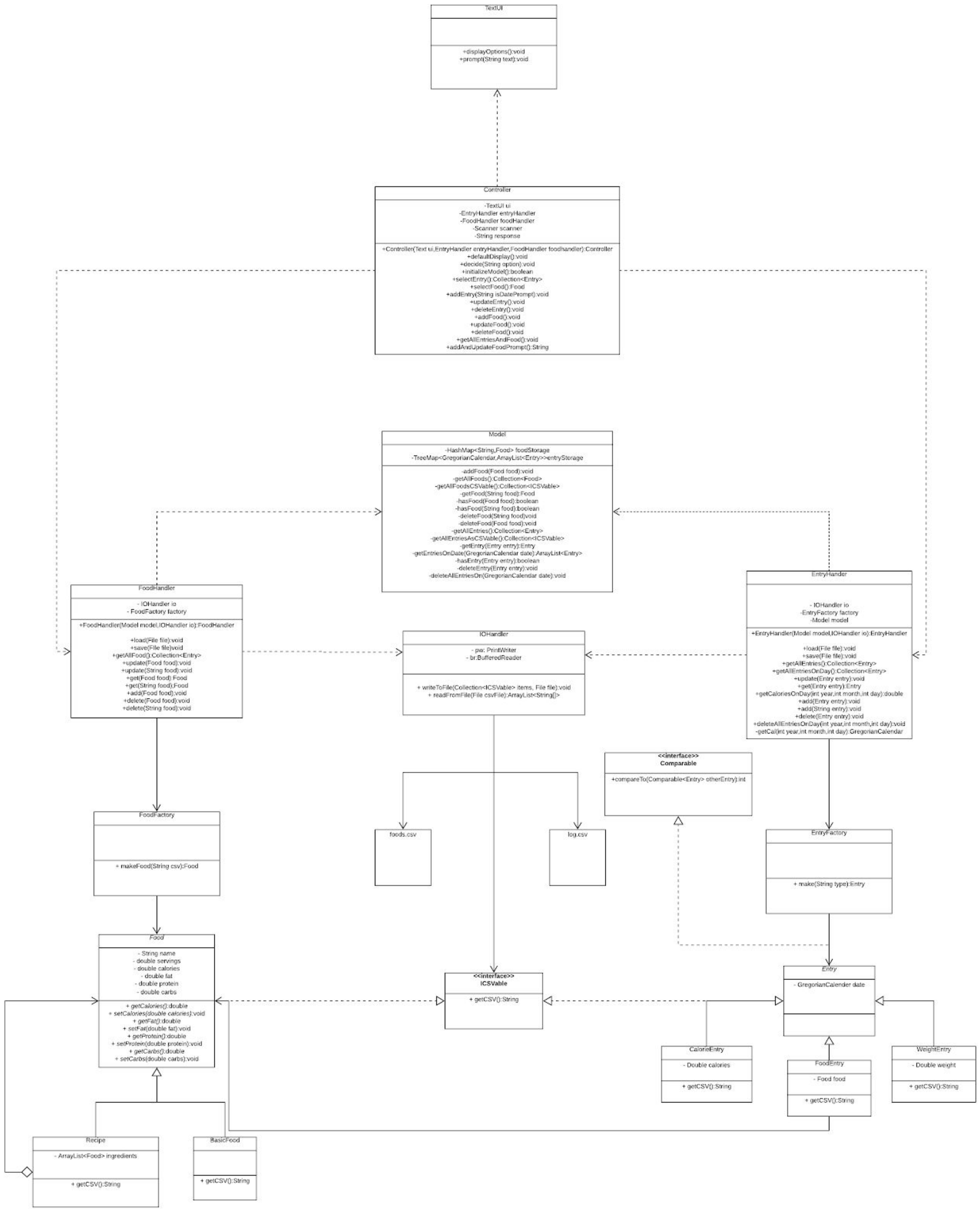
View Subsystem

Class TextUI	
Responsibilities	Displays the requested data(Food or Entries) to the GUI
Collaborators (inherits)	FoodHandler, EntryHandler, and Controller



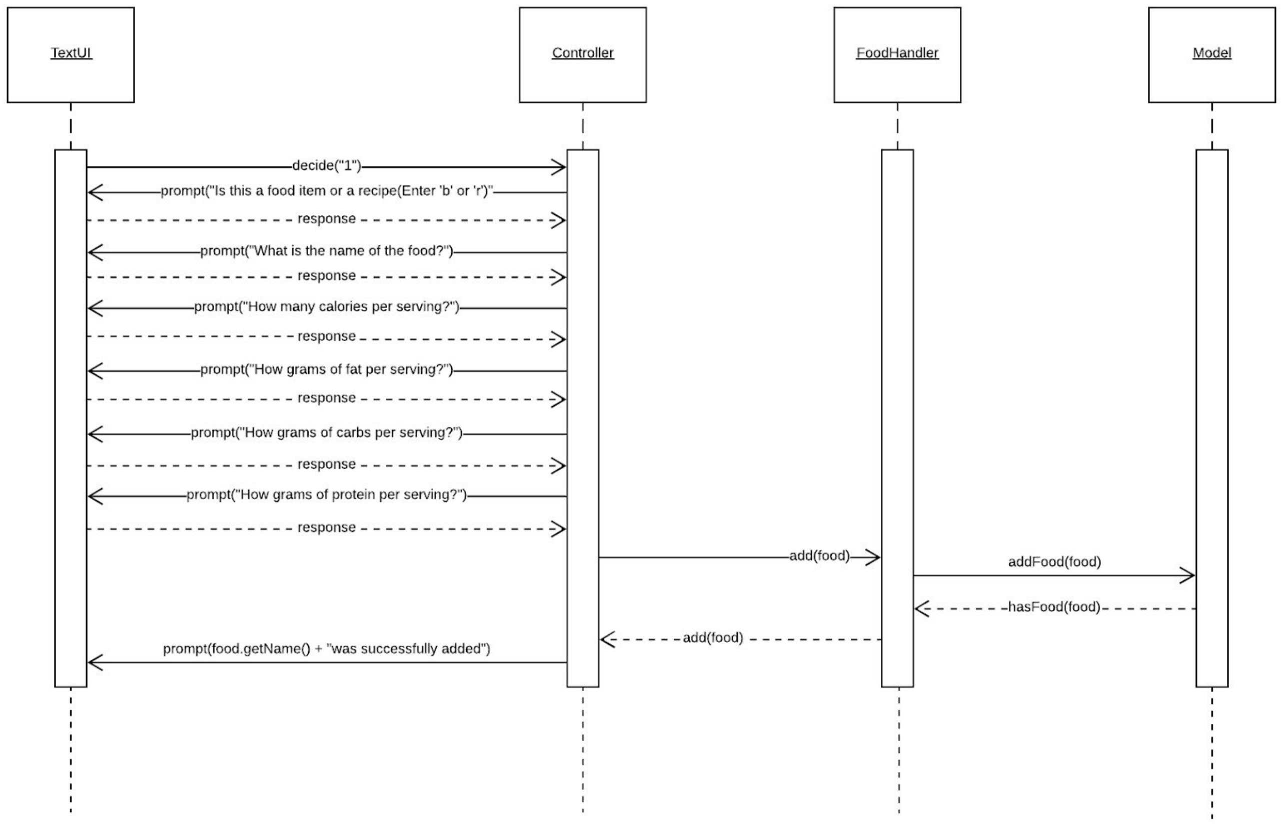
Diagrams

UML Class Diagram


TextUI
+displayOptions():void
+prompt(String text):void
Controller
-TextUI ui
-EntryHandler entryHandler
-FoodHandler foodHandler
-Scanner scanner
-String response
+Controller(Text ui,EntryHandler entryHandler,FoodHandler foodhandler):Controller
+defaultDisplay():void
+decide(String option):void
+initializeModel():boolean
+selectEntry():Collection<Entry>
+selectFood():Food
+addEntry(String isDatePrompt):void
+updateEntry():void
+deleteEntry():void
+addFood():void
+updateFood():void
+deleteFood():void
+getAllEntriesAndFood():void
+addAndUpdateFoodPrompt():String
Model
-HashMap<String,Food> foodStorage
-TreeMap<GregorianCalendar,ArrayList<Entry>>entryStorage
-addFood(Food food):void
-getAllFoods():Collection<Food>
-getAllFoodsCSVable():Collection<ICSVable>
-getFood(String food):Food
-hasFood(Food food):boolean
-hasFood(String food):boolean
-deleteFood(String food)void
-deleteFood(Food food):void
-getAllEntries():Collection<Entry>
-getAllEntriesAsCSVable():Collection<ICSVable>
-getEntry(Entry entry):Entry
-getEntriesOnDate(GregorianCalendar date):ArrayList<Entry>
-hasEntry(Entry entry):boolean
-deleteEntry(Entry entry):void
-deleteAllEntriesOn(GregorianCalendar date):void
FoodHandler
- IOHandler io
- FoodFactory factory
+FoodHandler(Model model,IOHandler io):FoodHandler
+load(File file):void
+save(File file)void
+getAllFood():Collection<Entry>
+update(Food food):void
+update(String food):void
+get(Food food):Food
+get(String food);Food
+add(Food food):void
+delete(Food food):void
+delete(String food):void
IOHandler
- pw: PrintWriter
- br:BufferedReader
+ writeToFile(Collection<ICSVable> items, File file):void
+ readFromFile(File csvFile):ArrayList<String[]>
EntryHander
- IOHandler io
-EntryFactory factory
-Model model
+EntryHandler(Model model,IOHandler io):EntryHandler
+load(File file):void
+save(File file):void
+getAllEntries():Collection<Entry>
+getAllEntriesOnDay():Collection<Entry>
+update(Entry entry):void
+get(Entry entry):Entry
+getCaloriesOnDay(int year,int month,int day):double
+add(Entry entry):void
+add(String entry):void
+delete(Entry entry):void
+deleteAllEntriesOnDay(int year,int month,int day):void
-getCal(int year,int month,int day):GregorianCalendar
<<interface>>
Comparable
+compareTo(Comparable<Entry> otherEntry):int
FoodFactory
+ makeFood(String csv):Food
foods.csv
log.csv
EntryFactory
+ make(String type):Entry
Food
- String name
- double servings
- double calories
- double fat
- double protein
- double carbs
+ getCalories():double
+ setCalories(double calories):void
+ getFat():double
+ setFat(double fat):void
+ getProtein():double
+ setProtein(double protein):void
+ getCarbs():double
+ setCarbs(double carbs):void
<<interface>>
ICSVable
+ getCSV():String
Entry
- GregorianCalender date
CalorieEntry
- Double calories
+ getCSV():String
FoodEntry
- Food food
+ getCSV():String
WeightEntry
- Double weight
+ getCSV():String
Recipe
- ArrayList<Food> ingredients
+ getCSV():String
BasicFood
+ getCSV():String

Sequence Diagrams

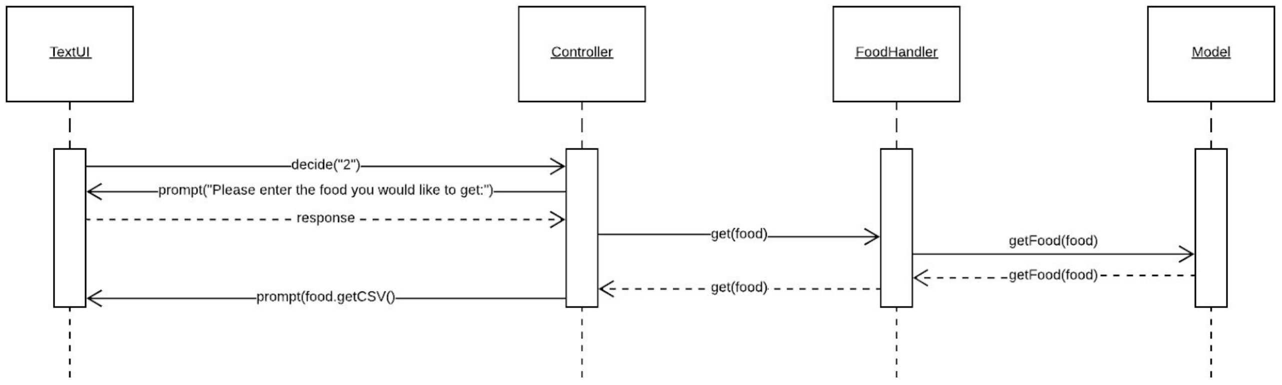
Create a Food


TextUI
Controller
FoodHandler
Model
decide("1")
prompt("Is this a food item or a recipe(Enter 'b' or 'r')"
response
prompt("What is the name of the food?")
response
prompt("How many calories per serving?")
response
prompt("How grams of fat per serving?")
response
prompt("How grams of carbs per serving?")
response
prompt("How grams of protein per serving?")
response
add(food)
addFood(food)
hasFood(food)
add(food)
prompt(food.getName() + "was successfully added")

Description

The user chooses the update option from the TextUI with a keystroke of 1. The controller is sent a String containing the keystroke and responds by calling TextUI's prompt() method to display the "Is this a food item or a recipe?" instructions. The user inputs a 'b' and the TextUI delivers the response as a String back to the controller. The Controller receives the response and prompts the user for name,calories,fat,carbs and protein. The controller creates a CSV string and sends it to the FoodHandler as a parameter using FoodHandler's add() method.The FoodHandler receives the String response and creates a new Food object with its internal FoodFactory class and sends it to the Model by calling the Model's addFood() method. The Food object is then added to the Model's internal HashMap and checks that the Food object is present with the hasFood() method. The Model returns a boolean to the FoodHandler which is then sent back to the Controller. The Controller calls the TextUI's prompt() method to display "Food was successfully added!" to the user.

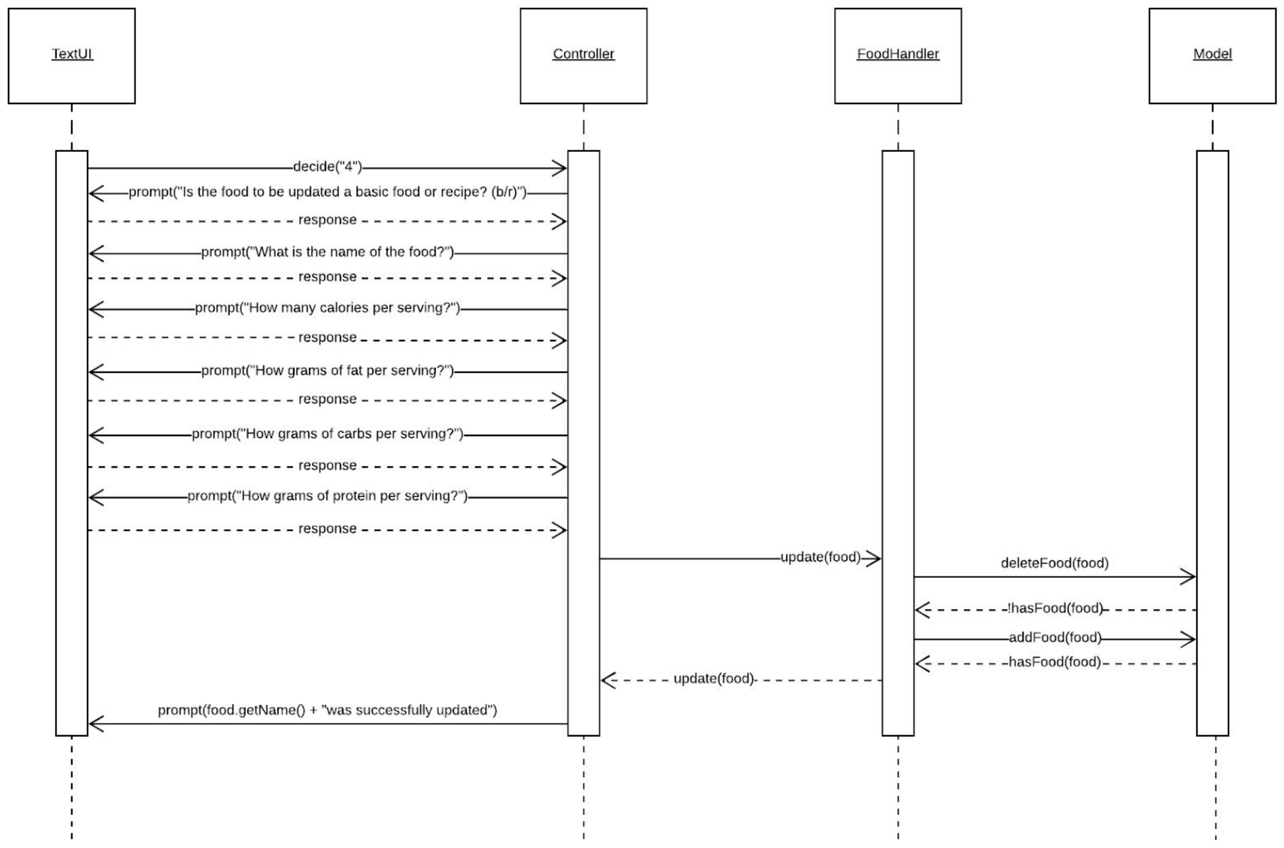
Retrieve a Food



Description

The user chooses the get option from the TextUI with a keystroke of 2. The controller is sent a String containing the keystroke and responds by calling TextUI's prompt() method to display the "Please enter the name of the food you would like to get" instructions. The response is sent to the controller which calls the FoodHandlers get() method using the response String as a parameter. The FoodHandler sends the String to the Model by calling its getFood() method. The Model uses the String to retrieve the Food object from its internal HashMap which is then returned the FoodHandler. The FoodHandler returns the Food object to the controller which then displays the Food information to the user by calling the TextUI's prompt() method.

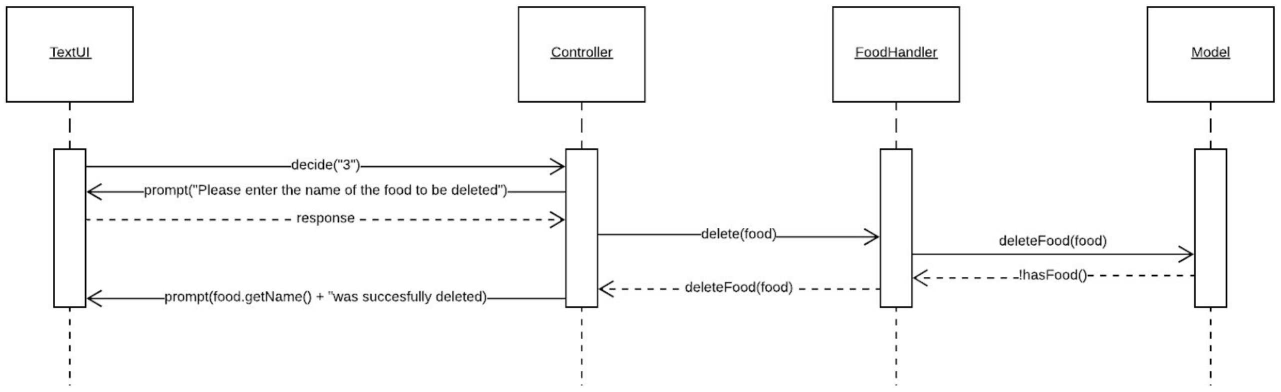
Update a Food


TextUI
Controller
FoodHandler
Model
decide("4")
prompt("Is the food to be updated a basic food or recipe? (b/r)")
response
prompt("What is the name of the food?")
response
prompt("How many calories per serving?")
response
prompt("How grams of fat per serving?")
response
prompt("How grams of carbs per serving?")
response
prompt("How grams of protein per serving?")
response
update(food)
deleteFood(food)
!hasFood(food)
addFood(food)
hasFood(food)
update(food)
prompt(food.getName() + "was successfully updated")

Description

The user chooses the update option from the TextUI with a keystroke of 4. The controller is sent a String containing the keystroke and responds by calling TextUI's prompt() method to display the "Is the food to be updated a basic food or recipe?" instructions. The user inputs a 'b' and the TextUI delivers the response as a String back to the controller. The Controller receives the response and prompts the user for name,calories,fat,carbs and protein. The controller creates a CSV string and sends it to the FoodHandler as a parameter using FoodHandler's update method.The FoodHandler receives the String response and sends it to the Model as parameter using the Model's deleteFood() method. The Model removes the specified Food by using its internal HashMap's remove() method, with the response String representing the Key of a Food object within the HashMap. The Model checks if the Food object is still present by using a !hasEntry() method. The Model returns a boolean to FoodHandler from it's deleteFood() method to indicate a successful removal. If the removal is successful, the FoodHandler creates a Food object from the CSV string using its internal FoodFactory class. The newly created Food object is sent to the Model via its addFood() method as the parameter. The Food object is then added to the Model's internal HashMap and checks that the Food object is present with its hasFood() method. If both the deleteFood() and addFood() return true to FoodHandler, the FoodHandler returns true back to the controller via the update() method. The Controller calls the TextUI's prompt() method to display "Food was successfully updated!" to the user.

Delete a Food


TextUI
Controller
FoodHandler
Model
decide("3")
prompt("Please enter the name of the food to be deleted")
response
delete(food)
deleteFood(food)
!hasFood()
deleteFood(food)
prompt(food.getName() + "was succesfully deleted)

Description

The user chooses the delete option from the TextUI with a keystroke of 3. The controller is sent a String containing the keystroke and responds by calling TextUI’s prompt() method to display the “Please enter the name of the food to be deleted” instructions. The user inputs the name of a food and the TextUI delivers the response as a String back to the controller. The controller receives the String response and sends it to the FoodHandler as a parameter using FoodHandler’s delete() method.. The FoodHandler receives the String response and sends it to the Model as parameter using the Model’s deleteFood() method. The Model removes the specified Food by using its internal HashMap’s remove() method, with the response String representing the Key of a Food object within the HashMap. The Model checks if the Food object is still present by using a !hasEntry() method. The Model returns a boolean to FoodHandler from it's deleteFood() method to indicate a successful removal. The FoodHandler returns a boolean to the Controller from its delete() method, reporting whether the removal was achieved. The Controller calls the TextUI’s prompt() method to display the “Food was successfully deleted message” to the user.

Pattern Usage

Composite Pattern

Component	Food
Leaf	BasicFood
Composite	Recipe

The composite pattern is fulfilled by the Food abstraction, along with the BasicFood and Recipe classes. Both the Basicfood and Recipe classes extend the Food abstraction and as such may be treated uniformly. The Recipe class is a recursive composite of BasicFoods, as well as other Recipes, following the principle of the Composite Pattern.

MVC Pattern

Models	Food, Entry, EntryHandler & FoodHandler
Views	TextUI
Controllers	Controller

The MVC pattern is implemented by the controller class reacting to events in the view classes, making and getting changes from the model classes, and finally returning the updated data back to the view classes.

Factory Pattern

Factories	FoodFactory, EntryFactory

Both of the Factory classes will make an instance of an object of their respective types. Based on the argument passed to their constructors, the factories will create a different type of their respective objects. The FoodFactory will make either basic foods or recipes, the EntryFactory will make calorie, food, or weight entries.