Diet Manager / Version 4

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 sub-system 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.

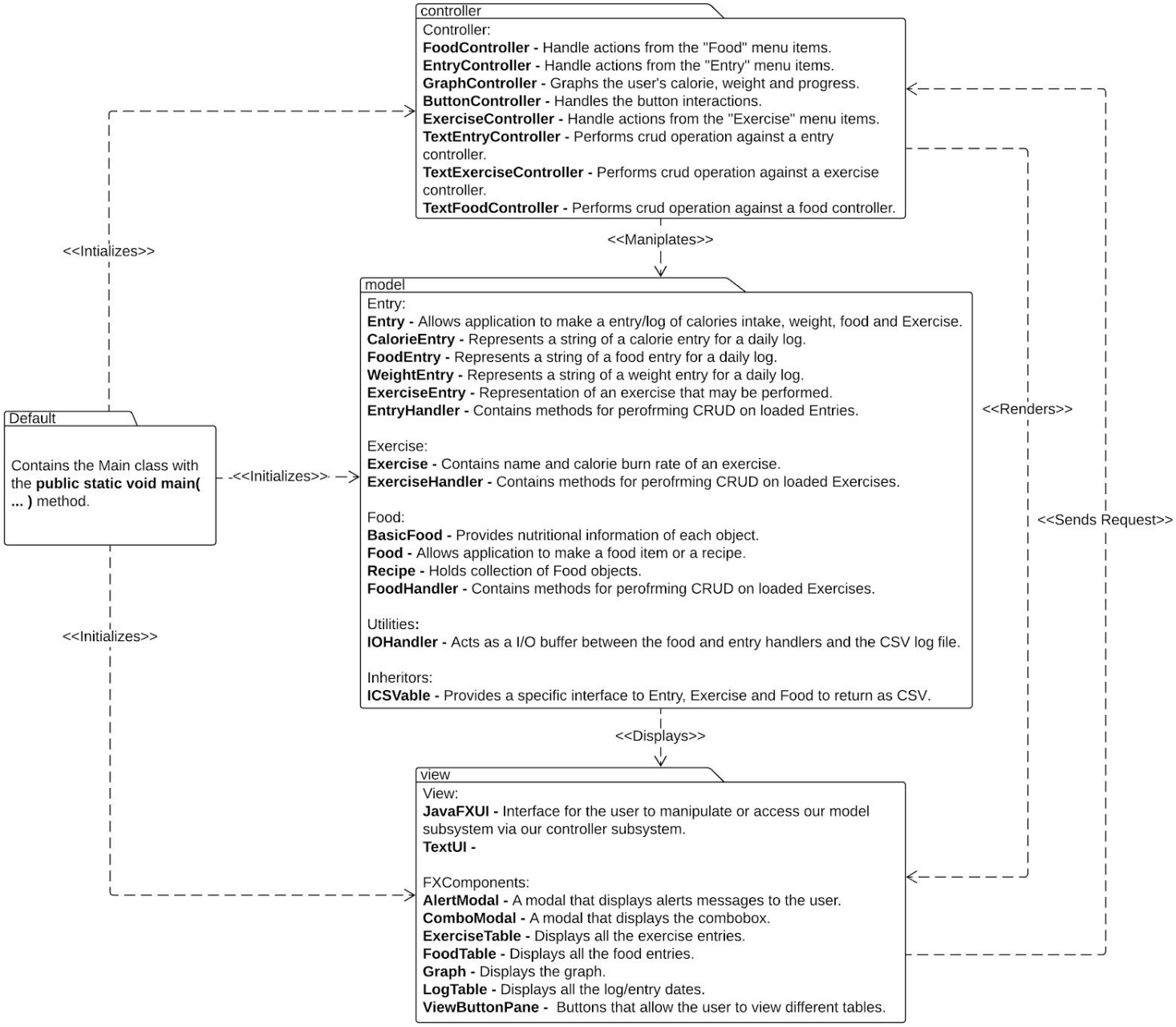
In order to provide the user with their retrospective analysis of their entries, it was necessary to have the Entry class implement a comparable interface by our use of Gregorian Calendar. This way the data can be stored in a data structure that respects their natural ordering. Because data will be stored 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.

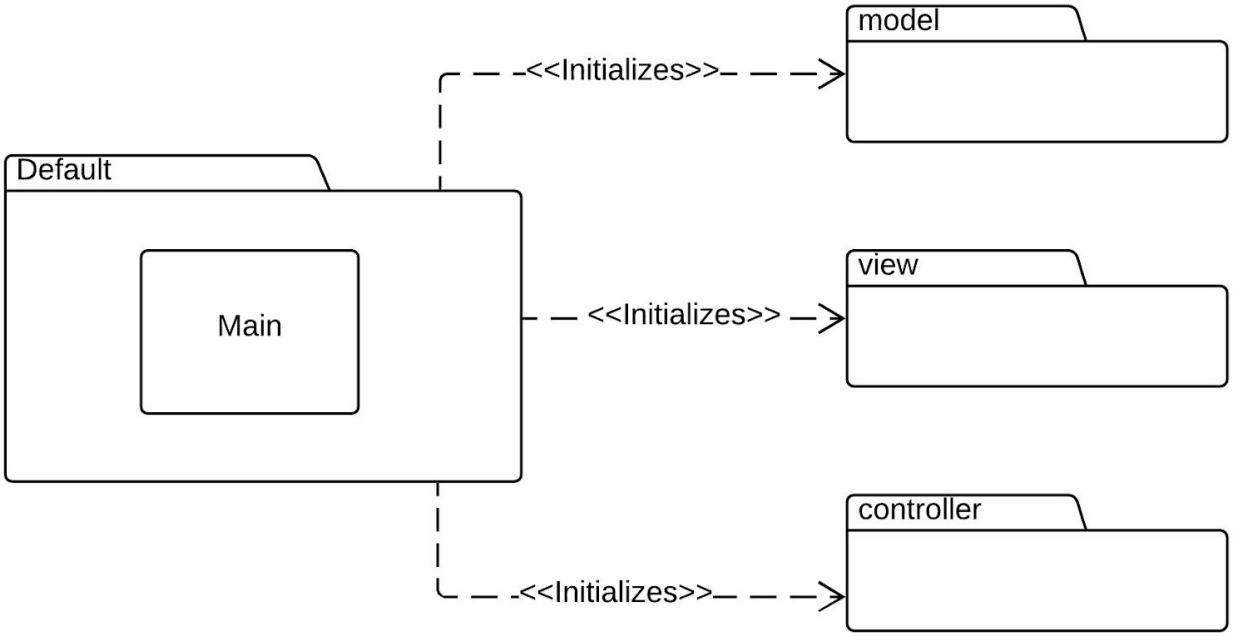
Subsystem Structure



Subsystem CRCs

Default Subsystem

Class Main	
Responsibilities	Instantiates a DataAccessLayer,IOHandler,EntryHandler, FoodHandler,ExerciseHandler,FoodController,EntryController,Exercise Controller and a JavaFXUI
Collaborators	JavaFXUI,IOHandler,DataAccessLayer, FoodHandler,FoodController,EntryHandler,EntryController,Exercise Handler, ExerciseController, TextEntryController, TextExerciseController, TextFoodController



Controller Subsystem

Class FoodController	
Responsibilities	Handle actions from the "Food" menu items. Handles user input and controls the view for crud actions on food objects.
Collaborators	FoodHandler, JavaFXUI, Food, FXComponents

Class EntryController	
Responsibilities	Handle actions from the "Entry" menu items. Handles user input and controls the view for crud actions on entry objects.
Collaborators	EntryHandler, JavaFXUI, Exercise, Food, Entry, FXComponents, JavaFXUI

Class ExerciseController	
Responsibilities	Handle actions from the "Exercise" menu items. Handles user input and controls the view for crud actions on Exercise objects.
Collaborators	Exercise, ExerciseHandler, FXComponents, JavaFXUI

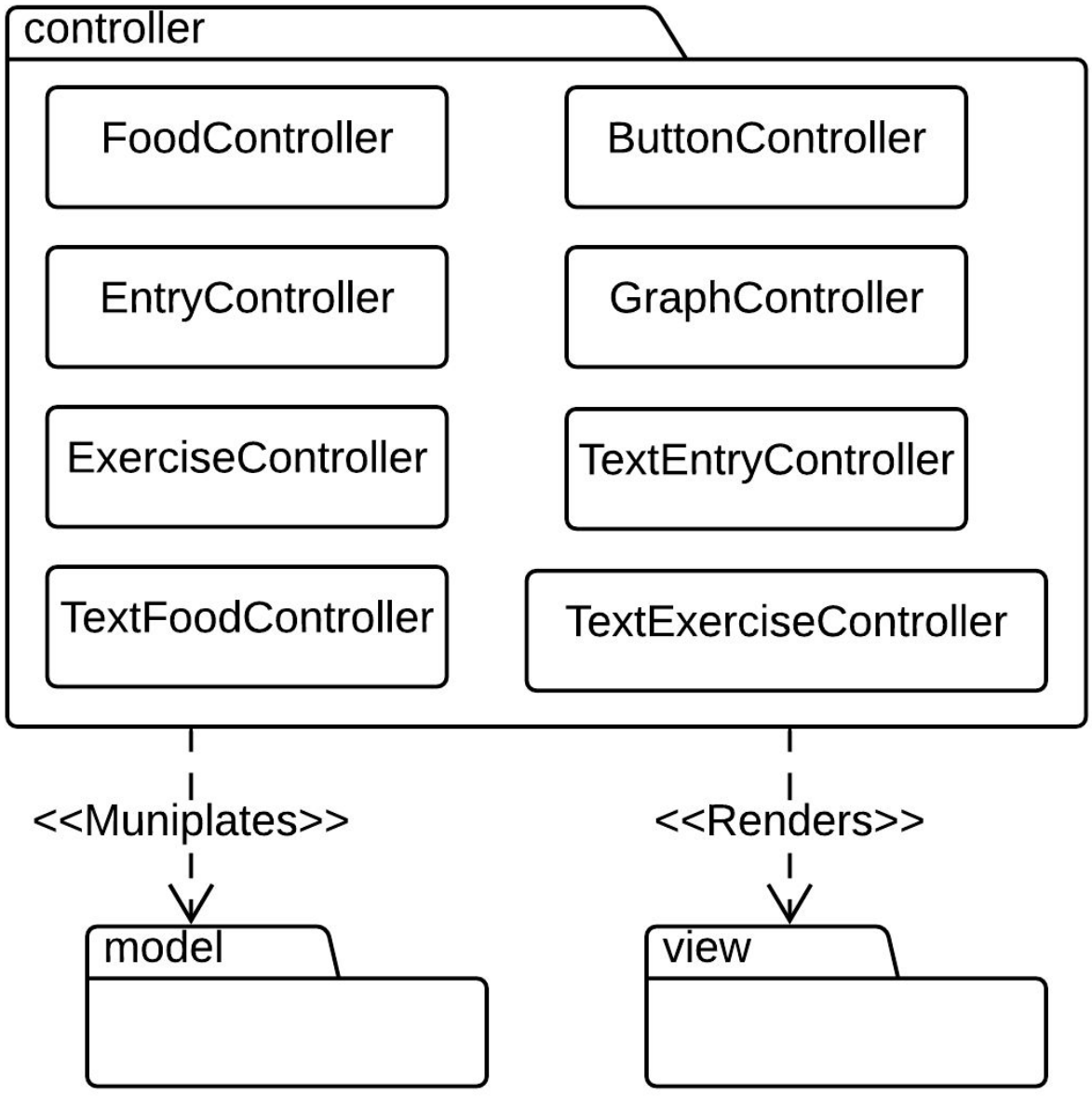
Class ButtonController	
Responsibilities	Handles the button interactions from the viewPane for View exercise, logs, and foods buttons.
Collaborators	JavaFXUI

Class GraphController	
Responsibilities	Graphs the user's calorie, weight and progress.
Collaborators	Entry

Class TextEntryController	
Responsibilities	Performs crud operations against a entry controller.
Collaborators	EntryHandler, Exercise, ExerciseHandler, Food, FoodHandler

Class TextExerciseController	
Responsibilities	Performs crud operations against a exercise controller.
Collaborators	ExerciseHandler, Exercise

Class TextFoodController	
Responsibilities	Perform crud operation against a food controller.
Collaborators	BasicFood, Food, FoodHandler



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.”e” for exercise, “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 ExerciseHandler - 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 “e” for exercise, “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 ExerciseEntry	
Responsibilities	Representation of an exercise that may be performed.
Collaborators (uses)	Entry - Allows application to make a entry/log of calories intake, weight, food, and exercise

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,food and exercise
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. ExerciseEntry - Represents a string of an exercise entry with minutes for a daily log

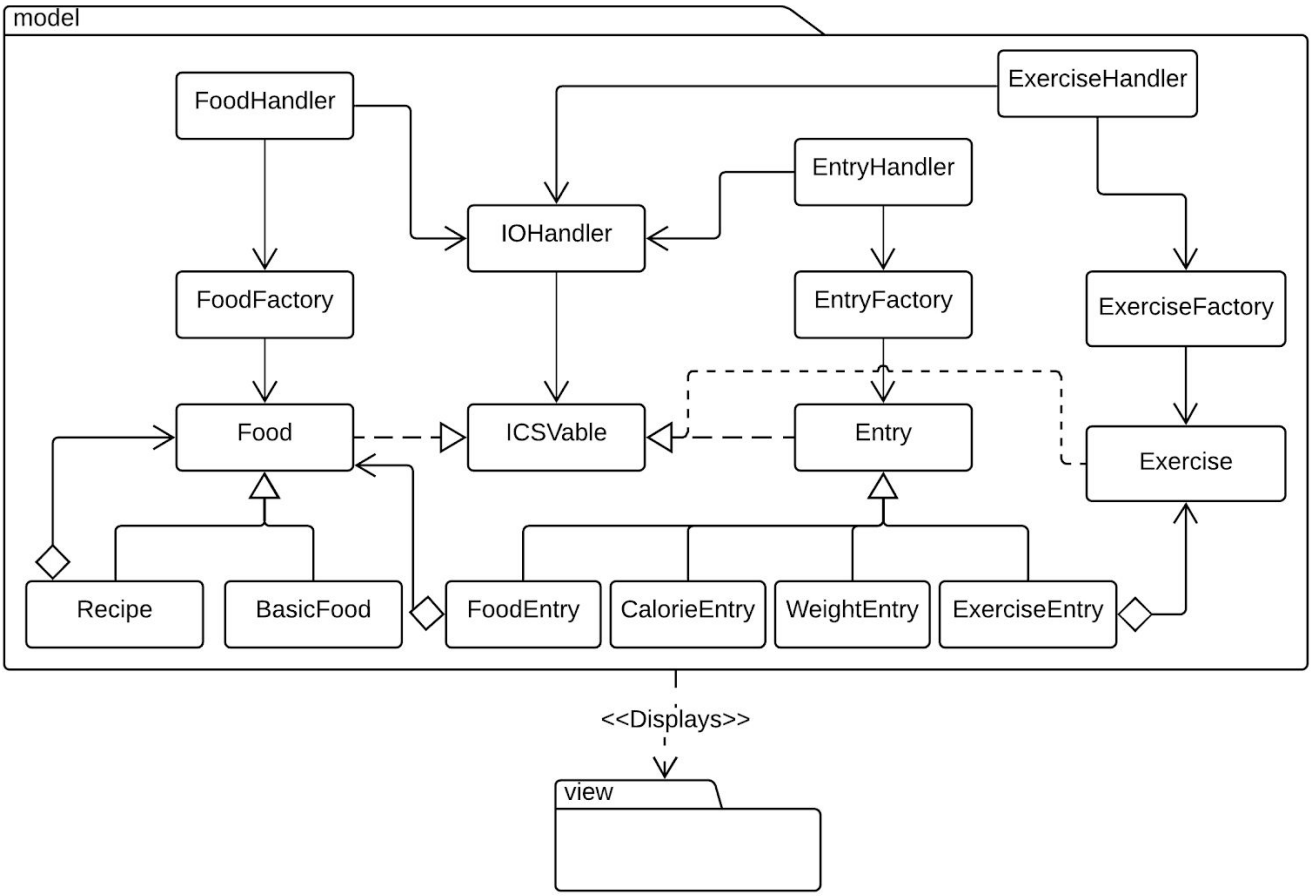
Class Exercise	
Responsibilities	Allows application to make an Exercise containing name of exercise and calorie burn
Collaborators	ICSVable

Class ICSVable	
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 ExerciseHandler	
Responsibilities	Load contents of exercise.csv into a Exercise objects, then adds them to DataAccessLayer’s exerciseStorage HashMap. The models can be retrieved from the HashMaps by providing ExerciseHandler with their unique name key.
Collaborators (uses)	ExerciseFactory - creates and returns exercise object based on provided CSV string IOHandler - Acts as a I/O buffer between the food and entry handlers and the CSV log file.


model
FoodHandler
ExerciseHandler
EntryHandler
IOHandler
FoodFactory
EntryFactory
ExerciseFactory
Food
ICSVable
Entry
Exercise
Recipe
BasicFood
FoodEntry
CalorieEntry
WeightEntry
ExerciseEntry
<<Displays>>
view

View Subsystem

Class JavaFXUI	
Responsibilities	Displays the requested data(Food, Entries or Exercise) to the GUI Buttons will call controllers which will manipulate/access the Model subsystem and display that data
Collaborators (uses)	FoodHandler, EntryHandler, ExerciseHandler, EntryController, ExerciseController, FoodController, FXComponents

Class TextUI	
Responsibilities	Displays the requested data(Food or Entries) to the GUI
Collaborators (uses)	TextEntryController, TextExerciseController, TextFoodController

Class AlertModal	
Responsibilities	A modal used to display alerts (warnings, prompts, notices, etc.) to users. Contains a label for text and an OK button which closes the modal.

Class ComboModal	
Responsibilities	A modal used to display a combobox with options to the user. Eliminates the need to create a new modal for many options.

Class ExerciseTable	
Responsibilities	Displays all exercise entries to the user via the JavaFXUI.
Collaborators (uses)	EntryHandler, Exercise, ExerciseHandler, Food, FoodHandler

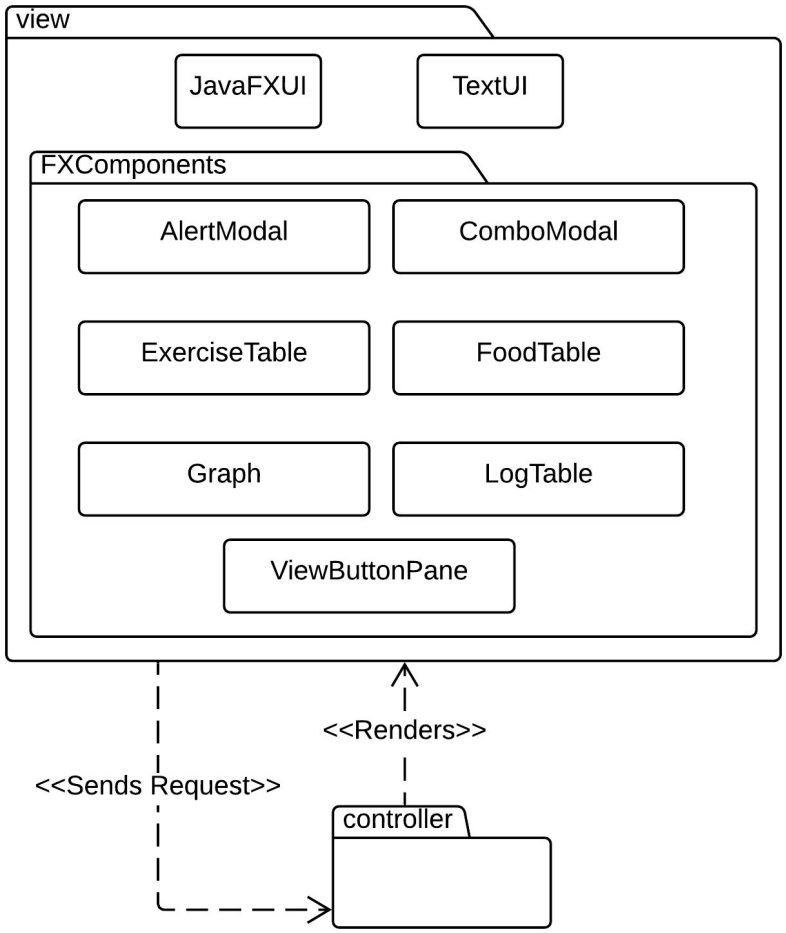
Class FoodTable	
Responsibilities	Displays all food entries to the user via the JavaFXUI.
Collaborators (uses)	EntryHandler, ExerciseHandler, Food, FoodHandler

Class Graph

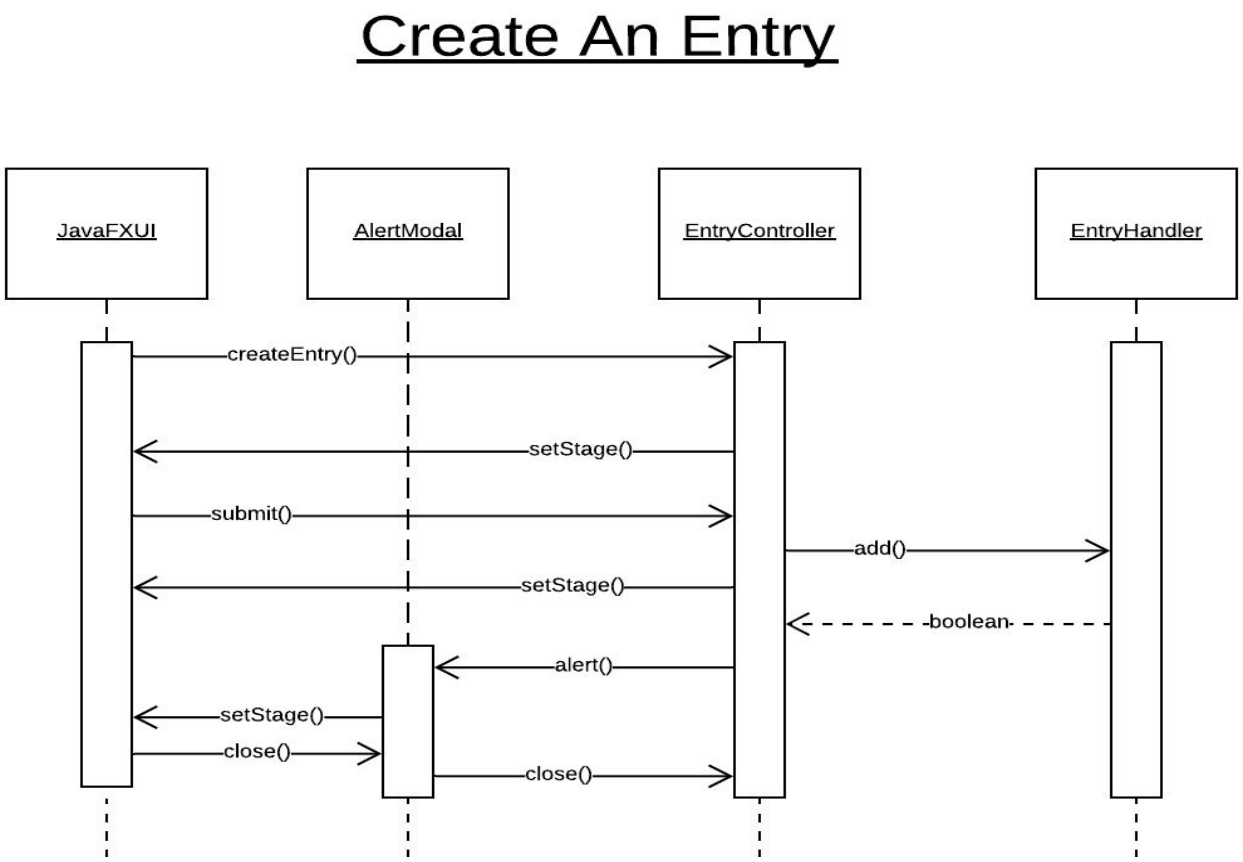
Responsibilities	Displays the bar graph to the user via the JavaFXUI.
Collaborators (uses)	EntryHandler, ExerciseHandler, FoodHandler

Class LogTable	
Responsibilities	Displays all log/entry dates to the user via the JavaFXUI.
Collaborators (uses)	EntryController, Entry, EntryHandler

Class ViewButtonPane	
Responsibilities	Setup and add the buttons used to view different tables in the JavaFXUI.
Collaborators (uses)	ButtonController, JavaFXUI

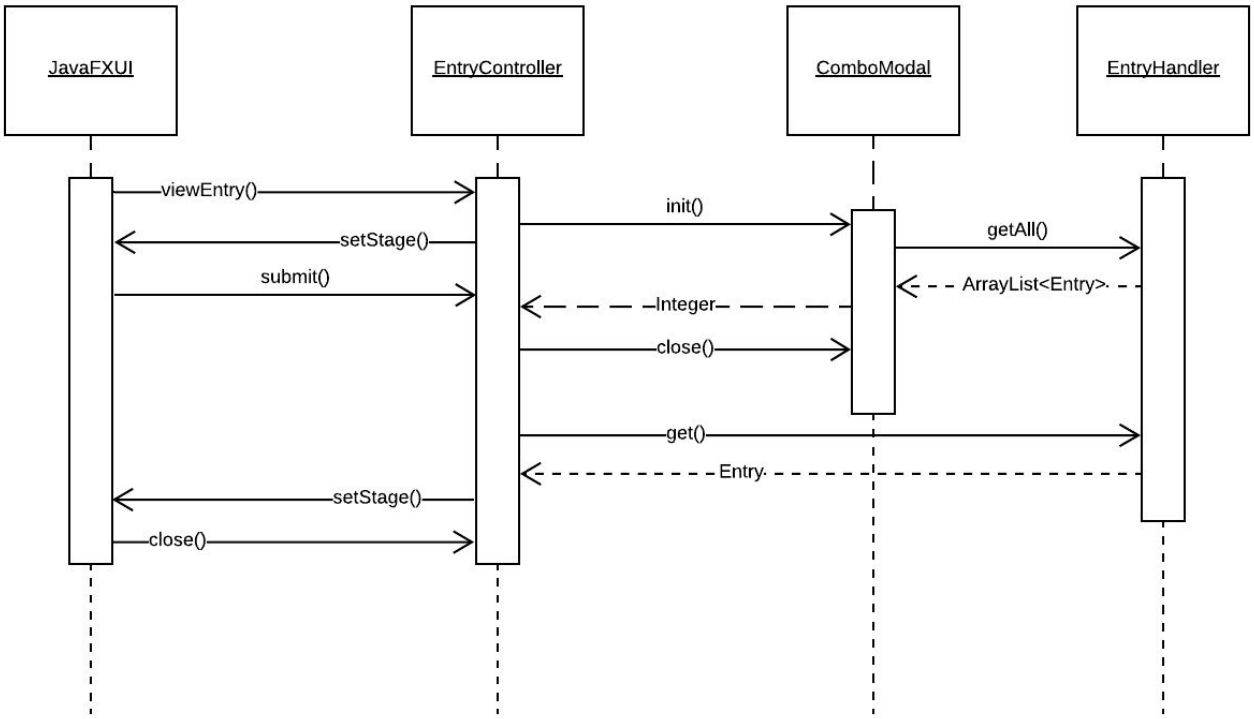


Sequence Diagrams



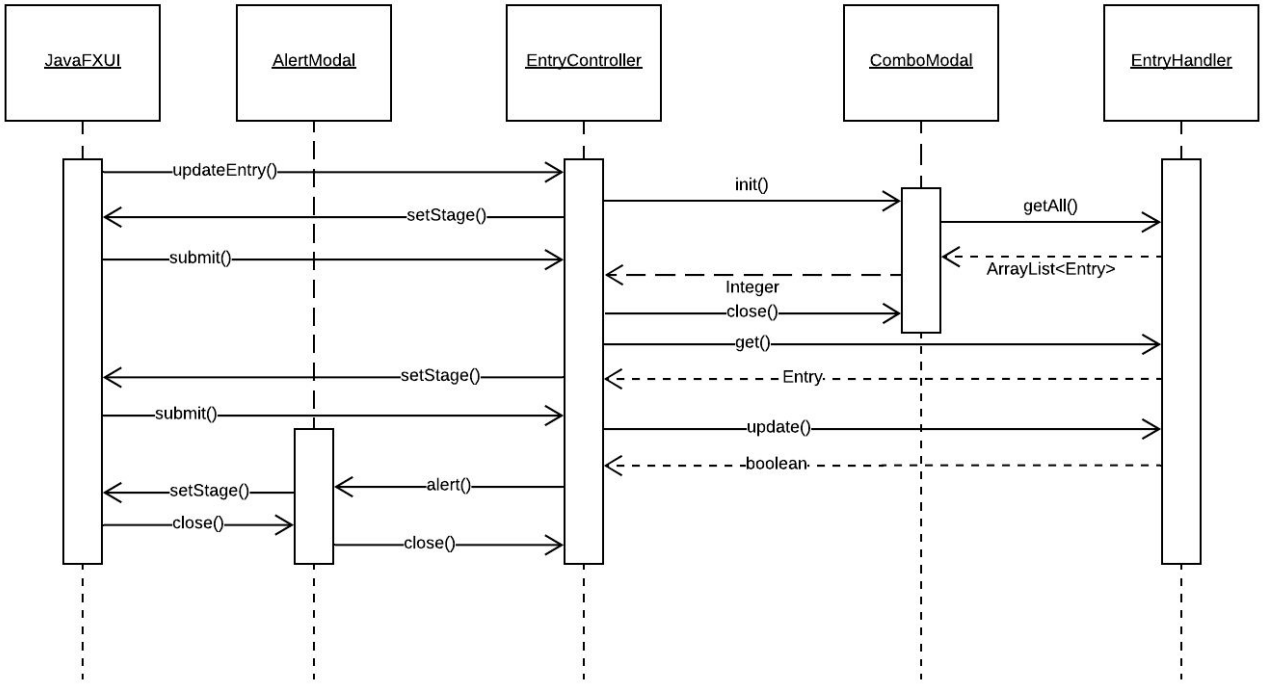
Description:
JavaFXUI calls createEntry() from the entry controller, the EntryController responds by calling setStage() displaying a window for user input. JavaFXUI calls EntryController's submit(), which takes in the user input and creates a new Entry. EntryController calls EntryHandler's add() sending in the newly created Entry, which is added to the EntryHandler's internal HashMap, the EntryHandler checks that the Entry was successfully added and returns a boolean indicating success or failure. The EntryController makes a AlertModal and calls it to alert(), and send setStage displays the AlertModal to the user with an appropriate message based on success or failure. JavaFXUI calls AlertModal's close() which then calls the EntryController's close(), closing all windows involved in the operation.

View An Entry



Description:
JavaFXUI calls EntryController's viewEntry(), which responds by creating a new ComboModal and calling its init(). The ComboModal calls EntryHandler's getAll(), which responds with an ArrayList containing all entries. The ComboModal adds all the Entry dates to a ComboBox and the EntryController uses setStage() to display the ComboModal. A date is selected and JavaFXUI calls EntryController's submit(), ComboModal responds with an integer before being closed by EntryController with close(). The EntryController uses the integer, which corresponds to an index and is used to call get() from EntryHandler,which responds with the selected Entry. The EntryController calls setStage(), displaying the Entry information to the user.

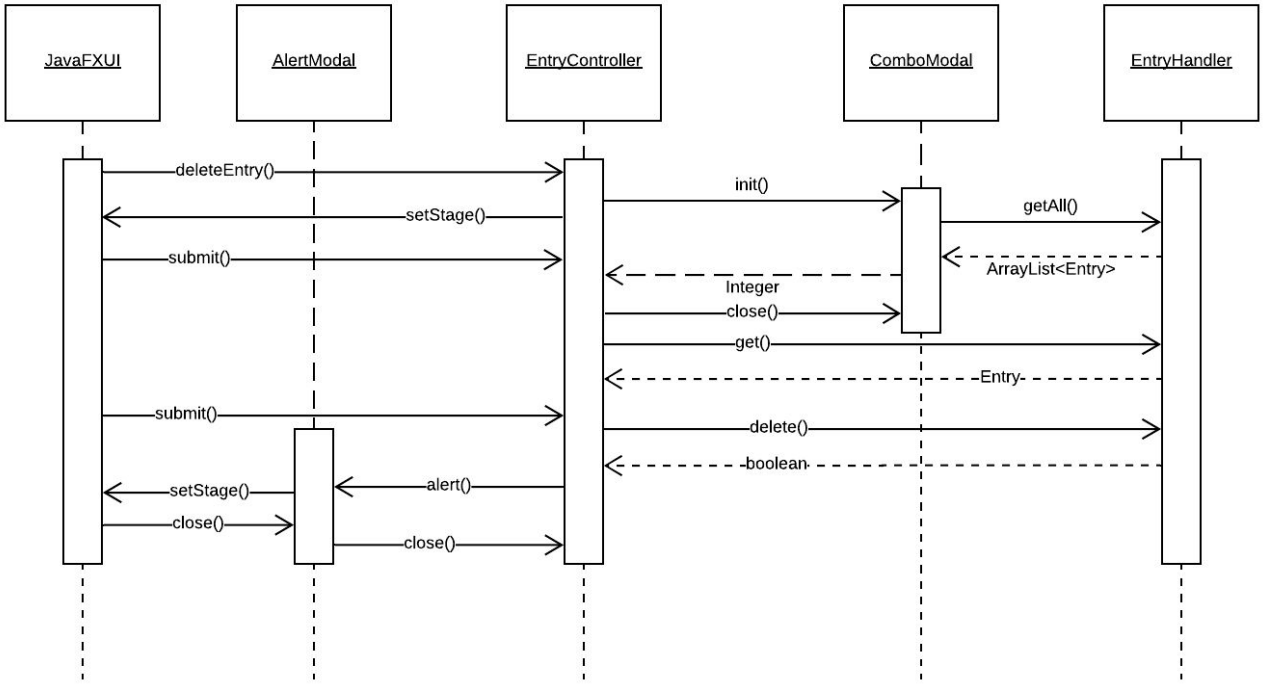
Update An Entry



Description:

JavaFXUI calls EntryController's updateEntry(), which responds by creating a new ComboModal and calling its init(). The ComboModal calls EntryHandler's getAll(), which responds with an ArrayList containing all entries. The ComboModal adds all the Entry dates to a ComboBox and the EntryController uses setStage() to display the ComboModal. A date is selected and JavaFXUI calls EntryController's submit(), ComboModal responds with an integer before being closed by EntryController with close(). The EntryController uses the integer, which corresponds to an index and is used to call get() from EntryHandler which responds with the selected Entry. The EntryController calls setStage(), displaying a window where the user can edit the entry properties. JavaFXUI calls EntryController's submit () which prompt's EntryController to add the new properties to the previously retrieve Entry object. The EntryController then calls EntryHandler's update() sending in the updated Entry object. EntryHandler adds the updated Entry to its internal HashMap,the EntryHandler checks that the Entry was successfully added and returns a boolean indicating success or failure. The EntryController makes an AlertModal and calls it to alert(), and setStage() displays the AlertModal to the user with an appropriate message based on success or failure. JavaFXUI calls AlertModal's close() which then calls the EntryController's close(), closing all windows involved in the operation.

Delete An Entry



Description:

JavaFXUI calls EntryController's deleteEntry(), which responds by creating a new ComboModal and calling its init(). The ComboModal calls EntryHandler's getAll(), which responds with an ArrayList containing all entries. The ComboModal adds all the Entry dates to a ComboBox and the EntryController uses setStage() to display the ComboModal. A date is selected and JavaFXUI calls EntryController's submit(), ComboModal responds with an integer before being closed by EntryController with close(). The EntryController uses the integer, which corresponds to an index and is used to call get() from EntryHandler,which responds with the selected Entry.The EntryController then calls EntryHandler's delete() sending in the Entry object. EntryHandler removes the updated Entry to its internal HashMap,the EntryHandler checks that the Entry was successfully removed and returns a boolean indicating success or failure. The EntryController makes an AlertModal and calls it to alert(), and setStage() displays the AlertModal to the user with an appropriate message based on success or failure. JavaFXUI calls AlertModal's close() which then calls the EntryController's close(), closing all windows involved in the operation.

Pattern Usage

Composite Pattern

Component	Food,Exercise
Leaf	BasicFood,BasicExercise
Composite	Recipe,Routine

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	JavaFXUI
Controllers	FoodController,EntryController,ExerciseController,TextController

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.

Observer Pattern

Updates	ExerciseTable, FoodTable, LogTable, Graph

Observable Pattern

Notifies	FoodHandler, ExerciseHandler, EntryHandler

The tables and graph located in the JavaFXUI observe their respective handlers in order to stay up to date. When the model is affected, the handlers notify their observers of changes. The observers, the tables and graphs, update their displays with the updated data.