Diet Manager / Version 3

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.

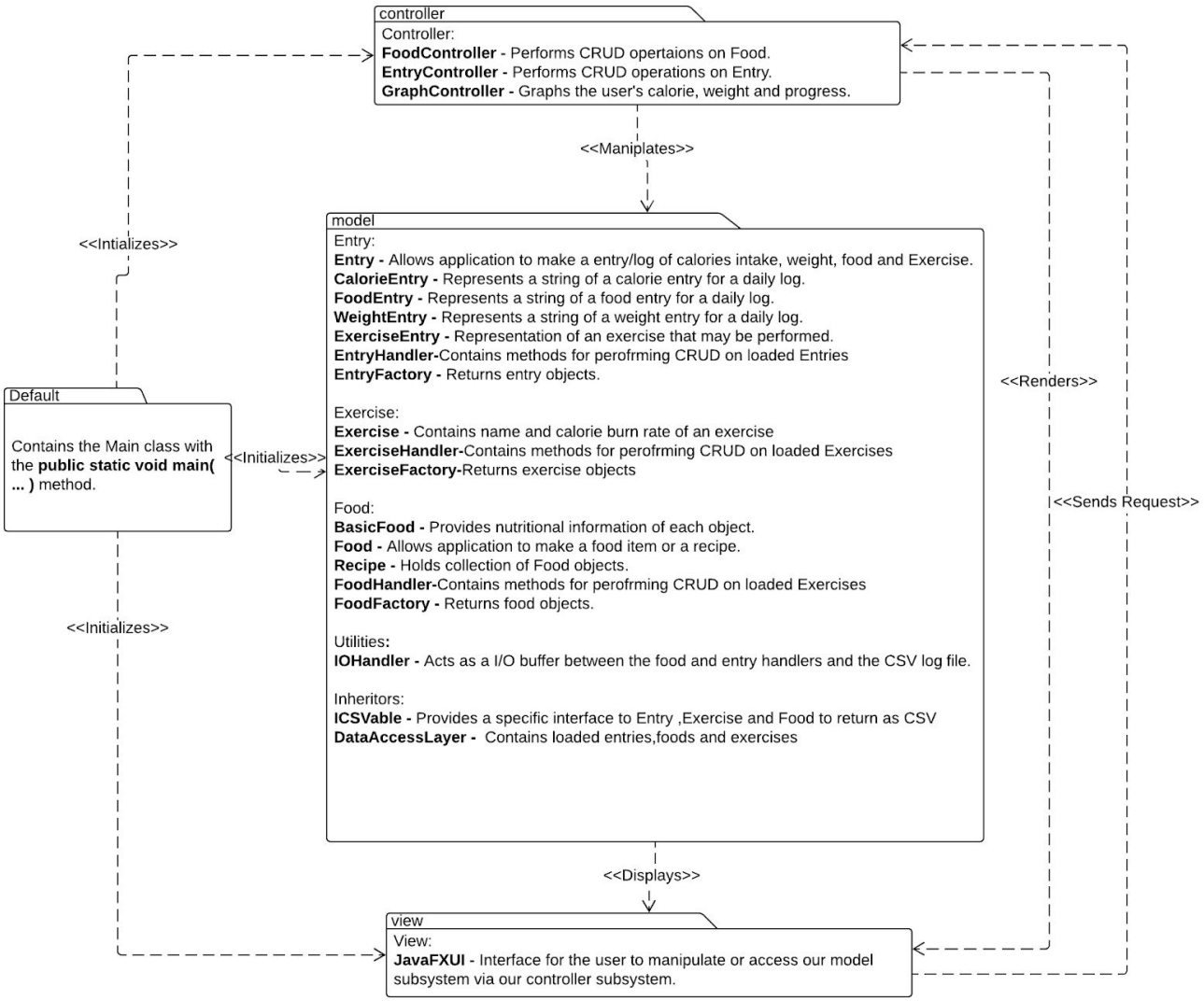
In order to provide the user with their retrospective analysis of their entries, it was necessary to have the Entry class implement a comparable interface by our use of Gregorian Calendar. This way the data can be stored in a data structure that respects their natural

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.

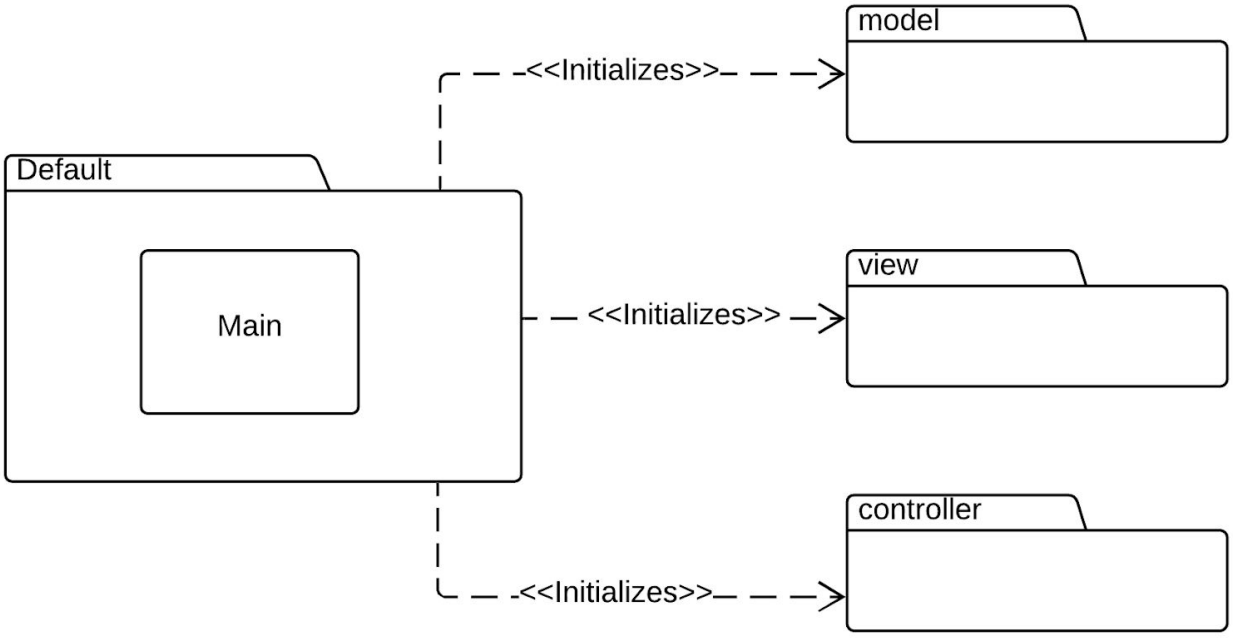
Subsystem Structure



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 and ExerciseController

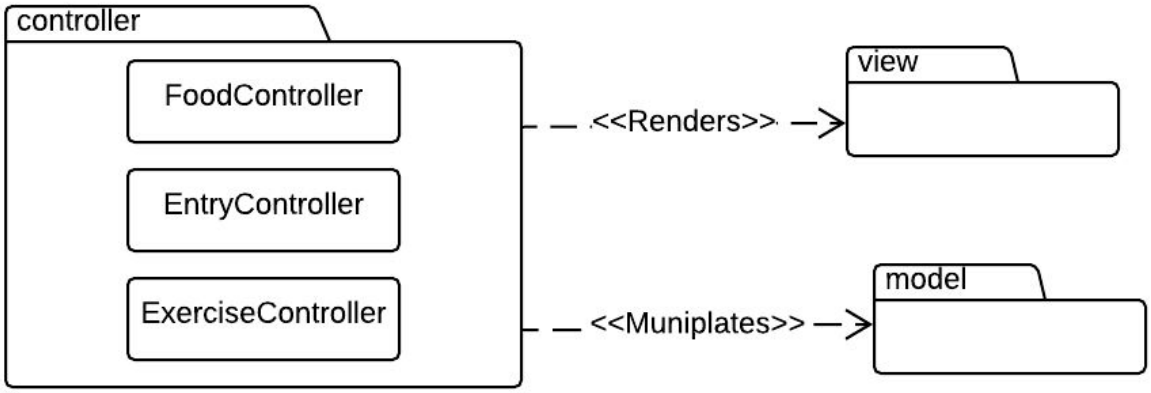


Controller Subsystem

Class FoodController	
Responsibilities	Perform CRUD - add, delete, update, and view foods
Collaborators	FoodHandler ,JavaFXUI

Class EntryController	
Responsibilities	Perform CRUD - add, delete, update, and view entries
Collaborators	EntryHandler, JavaFXUI

Class ExerciseController	
Responsibilities	Perform CRUD - add, delete, update, and view exercises
Collaborators	ExerciseHandler, JavaFXUI



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

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

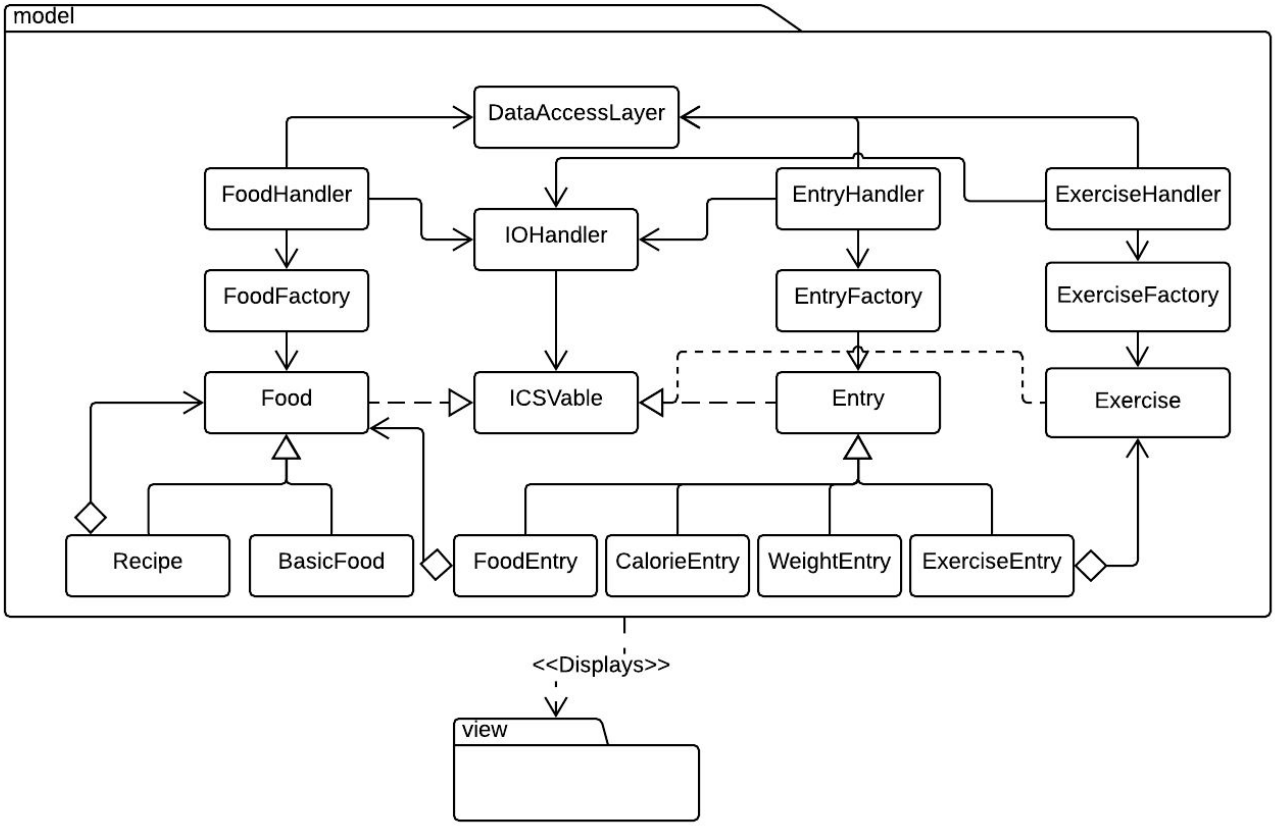
Class FoodHandler	
Responsibilities	Load contents of foods.csv into a recipe hashmap and basic food hashmap. The models can be retrieved from the hashmaps by providing FoodHandler with their unique name.
Collaborators (uses)	FoodFactory - creates and returns food object based on provided unique string. “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.

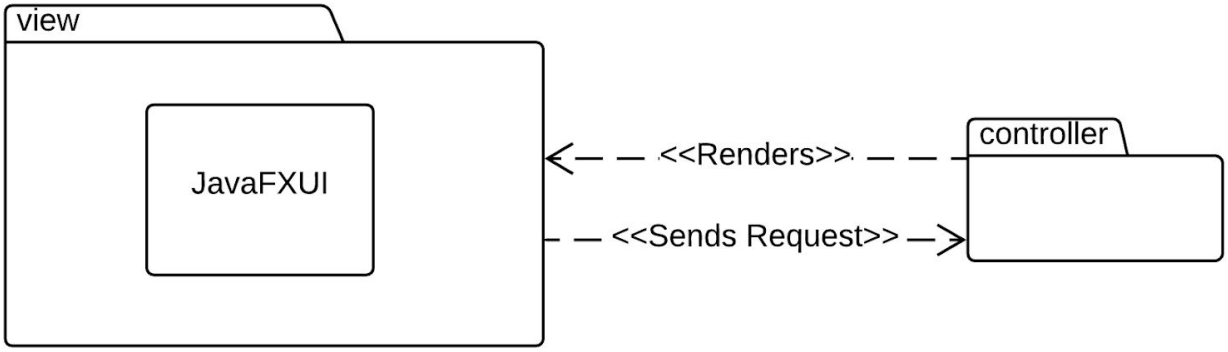
Collaborators (uses)	ExerciseFactory - creates and returns exercise object based on provided CSV string IOHandler - Acts as a I/O buffer between the food and entry handlers and the CSV log file.

Class DataAccessLayer	
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



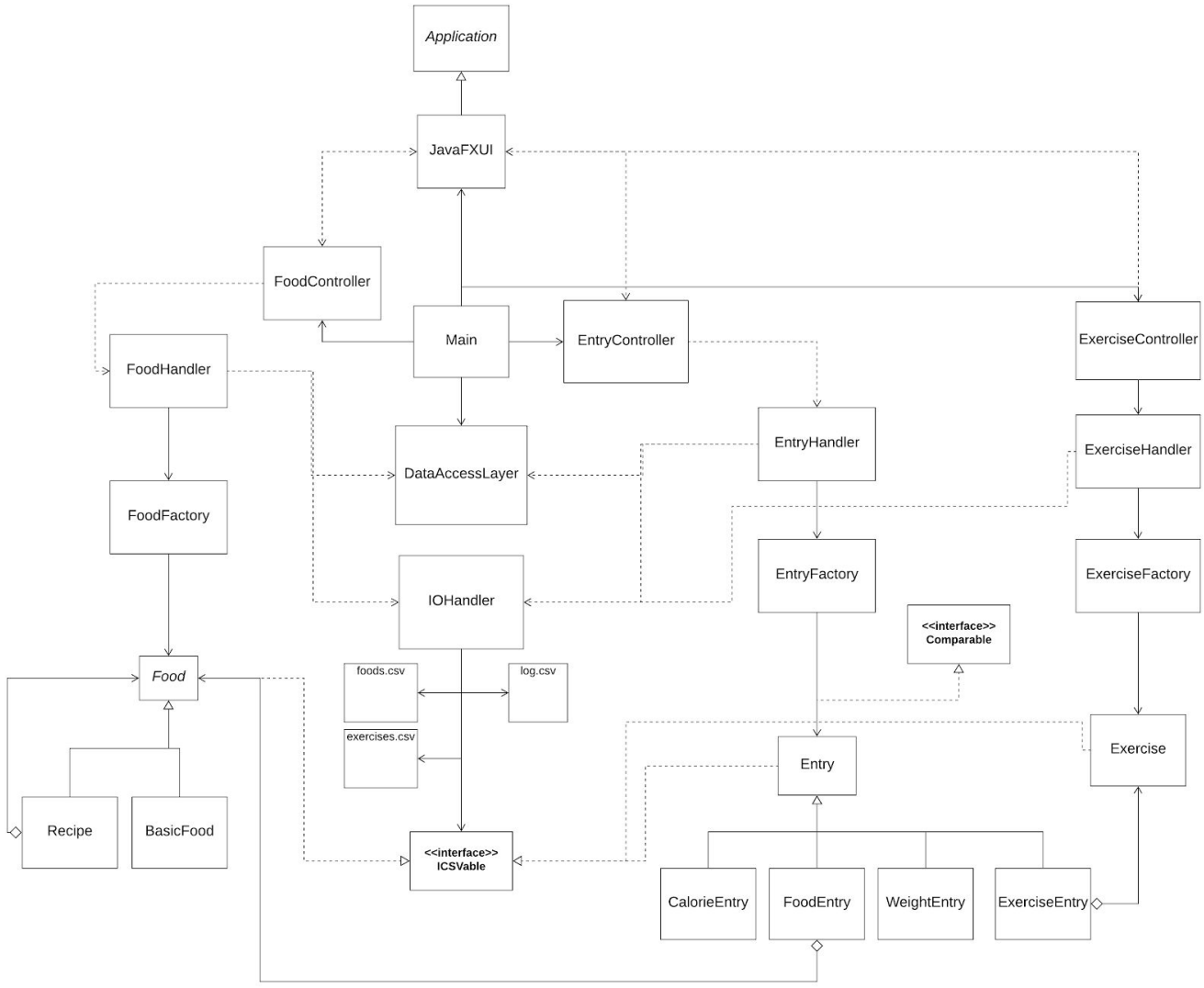
View Subsystem

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 (inherits)	FoodHandler, EntryHandler, and Controller, ExerciseController, FoodController, EntryController


view
JavaFXUI
<<Renders>>
<<Sends Request>>
controller

Diagrams

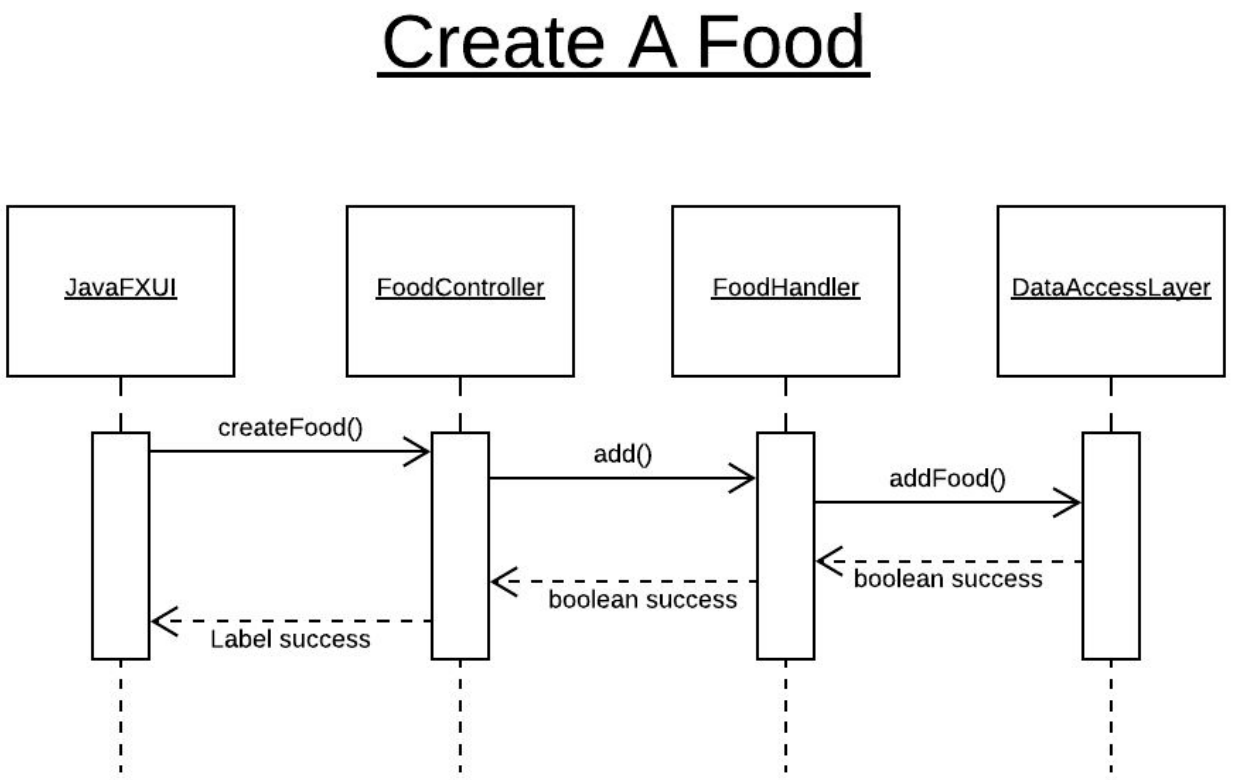
UML Class Diagram



Disclaimer: This is a simplified version of the program's UML. A detailed version by visiting here:

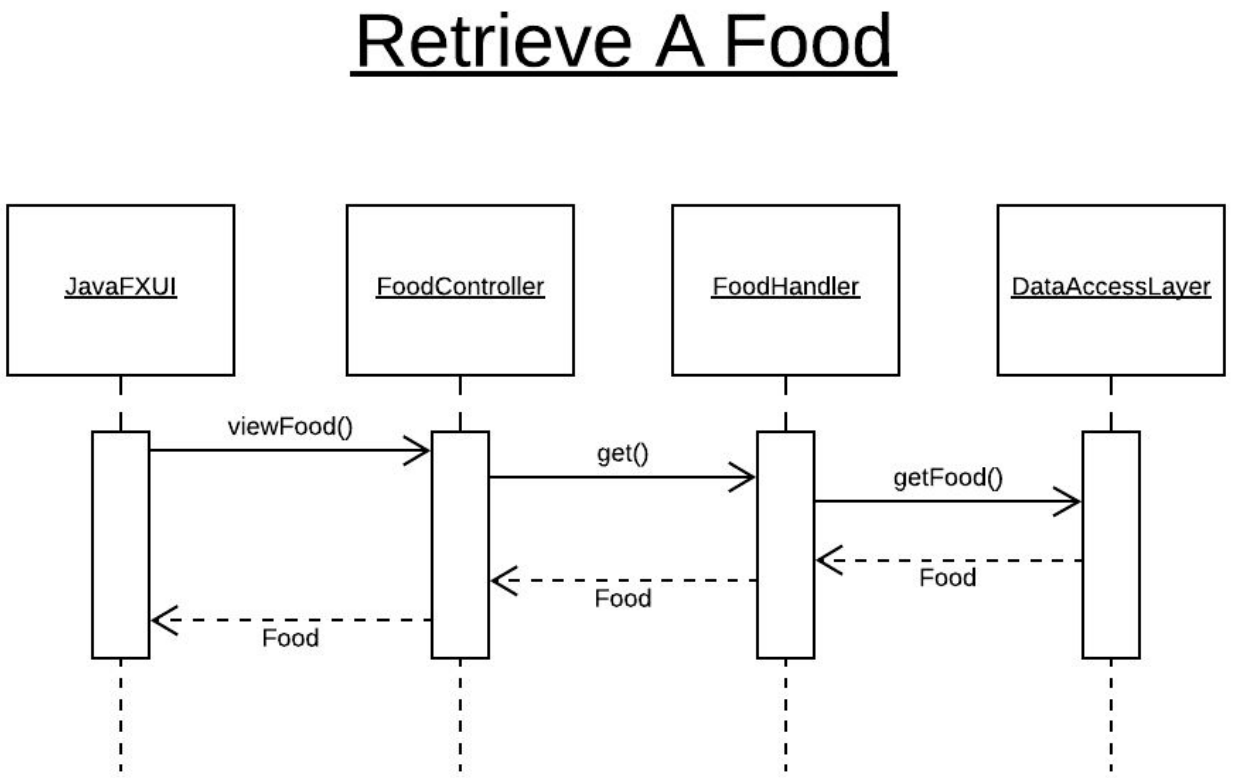
https://drive.google.com/a/g.rit.edu/file/d/1gngi8djTJIP1w9hETtb28LFw2ctRmo7t/view?usp=sharing

Sequence Diagrams



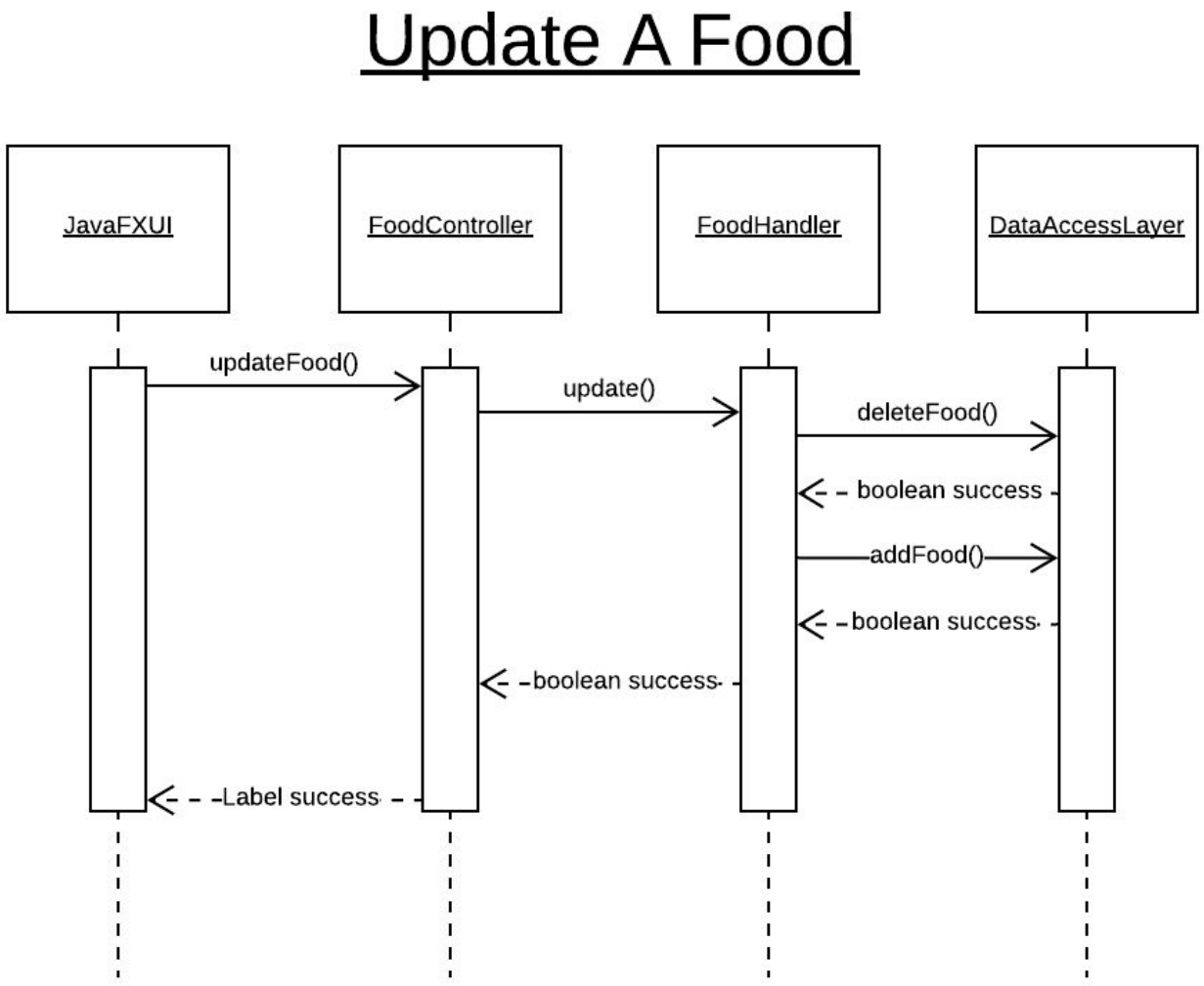
Create A Food Sequence Diagram

The User clicks a “Create Food” button and enters the information into a form. Then the user clicks the “Submit” button, which fires an event handler that calls the FoodController’s createFood() method. The FoodController calls the FoodHandler’s add() method, which creates a new Food object. The FoodHandler calls the DataAccessLayer’s addFood() method, which adds the newly created Food object to its internal HashMap. The DataAccessLayer’s addFood() method checks to see if the Food is present within its HashMap, and returns a boolean indicating success or failure to the FoodHandler. The FoodHandler passes the boolean to the FoodController, which creates a Label with a message indicating the success or failure of the operation. The Label is then returned to the JavaFXUI and displayed to the User.



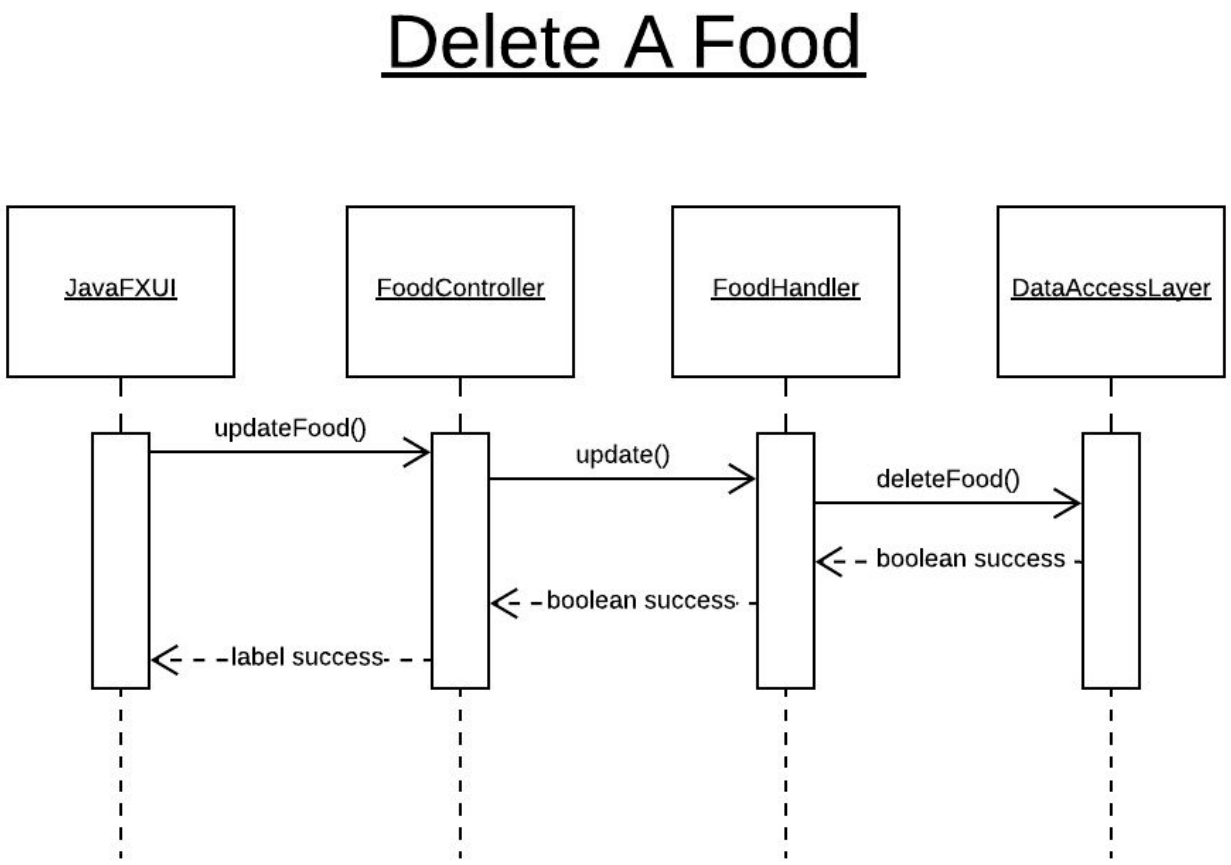
Retrieve A Food Sequence Diagram

The User selects a Food from a list displayed by JavaFXUI, which then calls the FoodController's viewFood() method. The FoodController calls the FoodHandler's get() method, which then calls the DataAccessLayer's getFood() method. The DataAccessLayer's getFood() method retrieves the Food from its internal HashMap and returns it to the FoodHandler. The FoodHandler passes the returned Food to the FoodController, which formats the information for display and applies it to the JavaFXUI.



Update A Food Sequence Diagram

The User selects a Food from a list displayed by JavaFXUI and chooses to update its properties. The JavaFXUI calls the FoodController's updateFood() method, which calls the FoodHandler's update() method. The FoodHandler calls the DataAccessLayer's deleteFood() method, which removes the Food from its internal HashMap. The deleteFood() method then checks to see that the Food was removed and returns a boolean indicating success or failure to the FoodHandler. If the Food was successfully removed, the FoodHandler then calls the DataAccessLayer's addFood() method, which adds the updated Food to its internal HashMap. The DataAccessLayer's addFood() method checks to see if the Food is present within its HashMap, and returns a boolean indicating success or failure to the FoodHandler. The FoodHandler passes the boolean to the FoodController, which creates a Label with a message indicating the success or failure of the operation. The Label is then returned to the JavaFXUI and displayed to the User.



Delete A Food Sequence Diagram

The User selects a Food from a list displayed by JavaFXUI and chooses to delete it. The JavaFXUI calls the FoodController's deleteFood() method, which calls the FoodHandler's update() method. The FoodHandler calls the DataAccessLayer's deleteFood() method, which removes the Food from its internal HashMap. The deleteFood() method then checks to see that the Food was removed and returns a boolean indicating success or failure to the FoodHandler. The FoodHandler passes the boolean to the FoodController, which creates a Label with a message indicating the success or failure of the operation. The Label is then returned to the JavaFXUI and displayed to the User.

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.

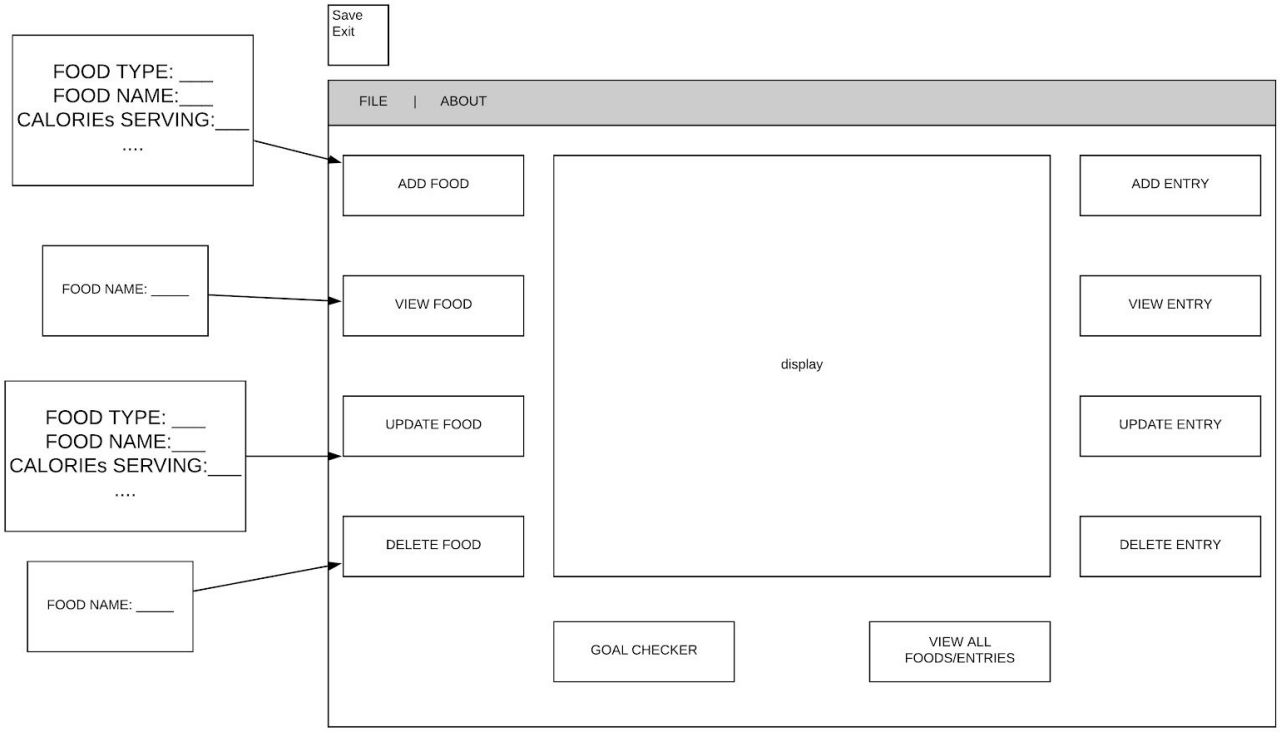
Factory Pattern

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.

Appendix

JavaFXUI



This is a basic wireframe we have created as a possibility for our JavaFX user interface. This wireframe allowed us to ensure that all functionality would be supported for the user. This also allowed us to plan the users interaction to better understand what other functionality might be needed. Revisions to this wireframe will occur as we continue to develop and refine the DietManager application.