Diet Manager / Version 1

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

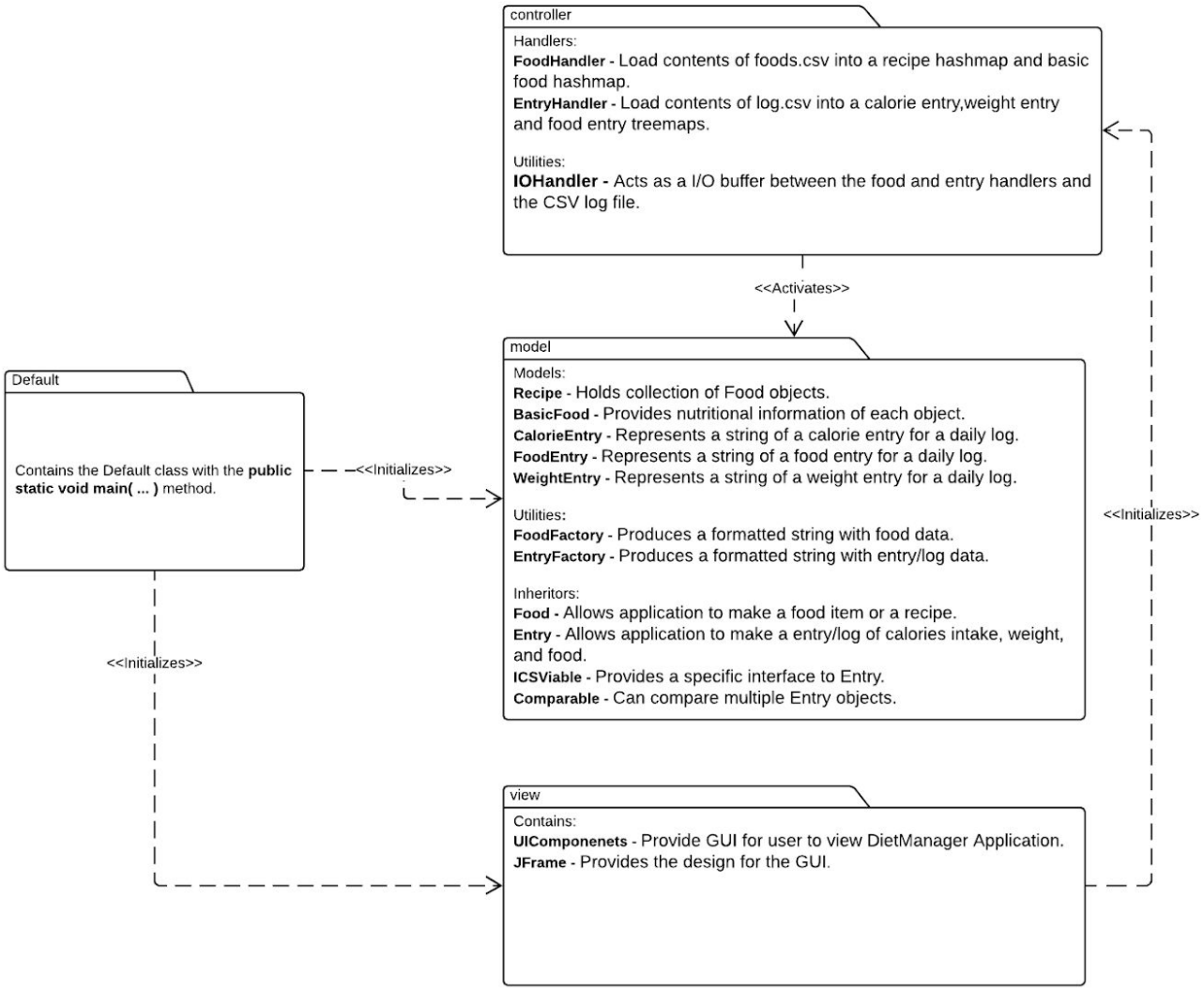
In order to provide the user with their retrospective analysis of their entries, it was necessary to have the Entry class implement a comparable interface. This way the data can be stored in a data structure that respects their natural ordering. Because data will be stored in its natural order the EntryHandler 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 Default	
Responsibilities	Creates handler objects.
Collaborators	FoodHandler - 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. EventHandler - 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.

Controller 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

Model Subsystem

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 Comparable (interface)	
Responsibilities	Provide a specific interface to Entry Can compare multiple Entry objects

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

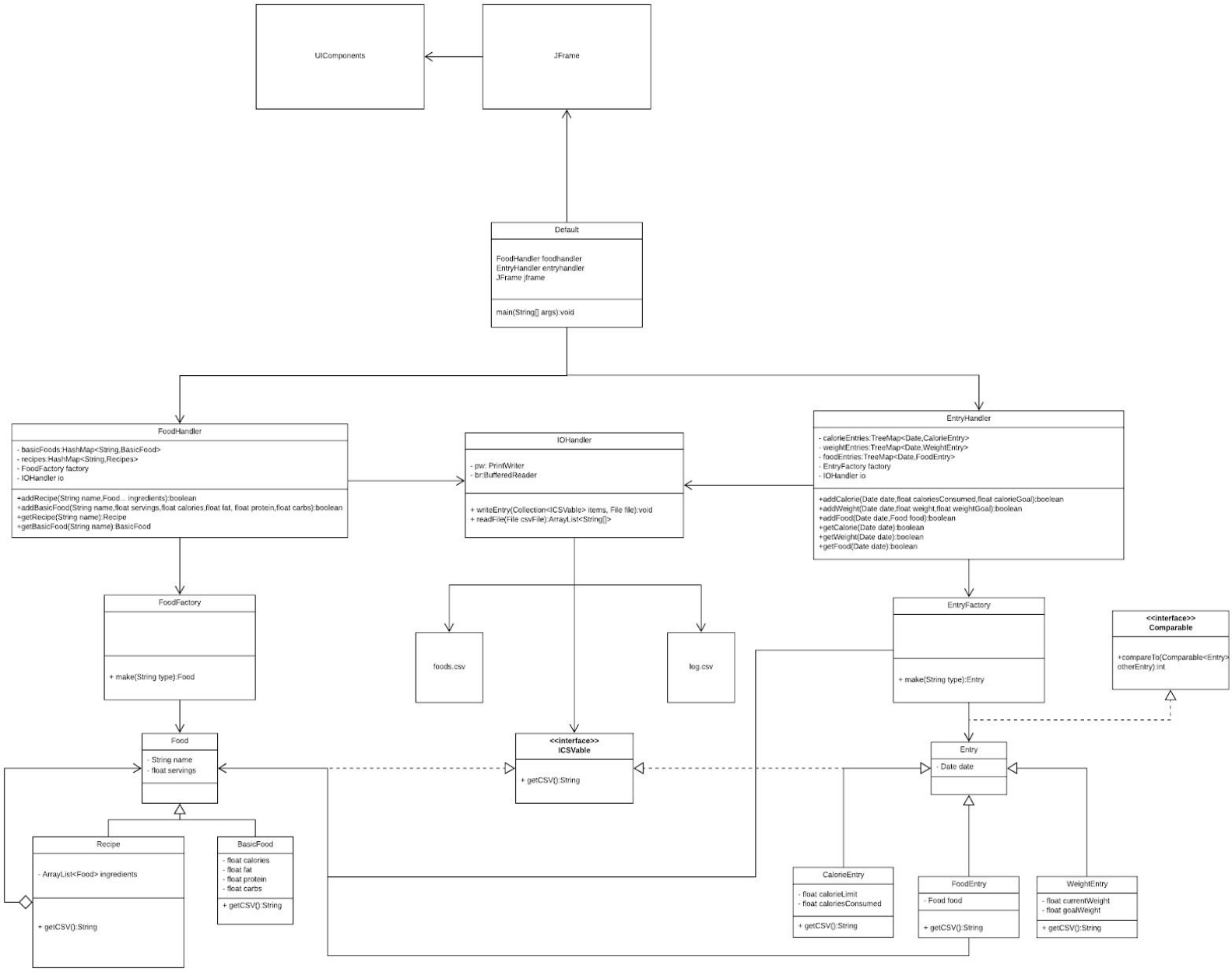
View Subsystem

Class UIComponents	
Responsibilities	Provide GUI for user to view DietManager Application
Collaborators (inherits)	JFrame

Class JFrame	
Responsibilities	Provides the design for the GUI
Collaborators (inherits)	Default

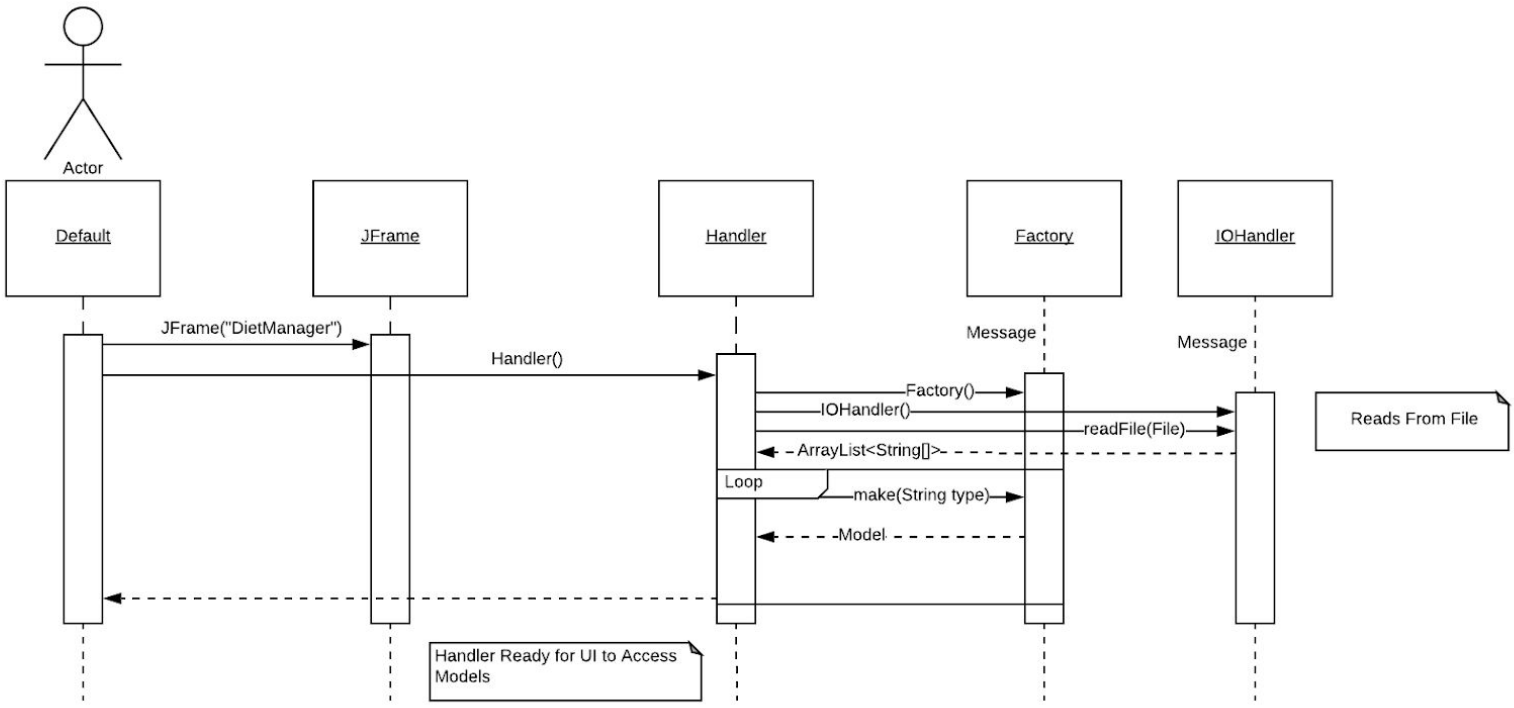
Diagrams

UML Class Diagram



Sequence Diagram

The user begins to interact with the program,the Default class creates a JFrame with a title of “Diet Manager”. The Default class then creates and calls the appropriate handler for the respective object that is being interacted with (food/entry).During its construction, the handler creates an IOHandler and the appropriate factory for the object in use. The IOHandler reads the .csv file associated with the object type and returns an array list to the handler which contains the contents of the file. The Handler then executes a loop which requests an object based on the key found in each line (“b”,”r”,”w”,”c” or “f”). Once the handler receives the appropriate object, it sets the properties of the object and places it into the appropriate Map contained within the Handler (“recipe”,”basic food”,”weight entry”,”calorie entry” or “food entry”). The user now has the capability to retrieve objects from the Maps within the Handler or write the objects properties to a .csv file.



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, Default
Views	JFrame, UIComponents
Controllers	FoodHandler, EntryHandler, IOHandler

The MVC pattern is implemented by the controllers classes 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.