

WELLNESS MANAGER 1.0

Project Design Document

Team D

Afzal Ali <ama3589@rit.edu>

Catherine Liu <cl1252@rit.edu>

Seth Landers <sdl1009@rit.edu>

Stephen Morrissey <sjm3524@rit.edu> Steven

Jackling <sj3152@rit.edu>

Project Summary

Wellness manager is a program that allows for users to keep a collection of food as well as a daily log of their food intake.

The program reads in basic foods from a csv file. The user also has the option to add their own basic foods to the collection. The foods added to the collection have basic nutritional information such as calories, fat, carbohydrates, and protein which can be used to keep track of nutritional intake each day. The user can also use basic foods from the collection to create their own recipes and create more complex recipes using other recipes as sub-recipes.

There is also a daily log built into the program. The user can use the daily log to track their food and nutrient intake each day. This also allows the user to set daily goals for weight and calories. The program will be able to indicate to the user whether or not they have met their goals for the day. As well as this, the program will generate graphs for the user based on the nutrient distribution of the day.

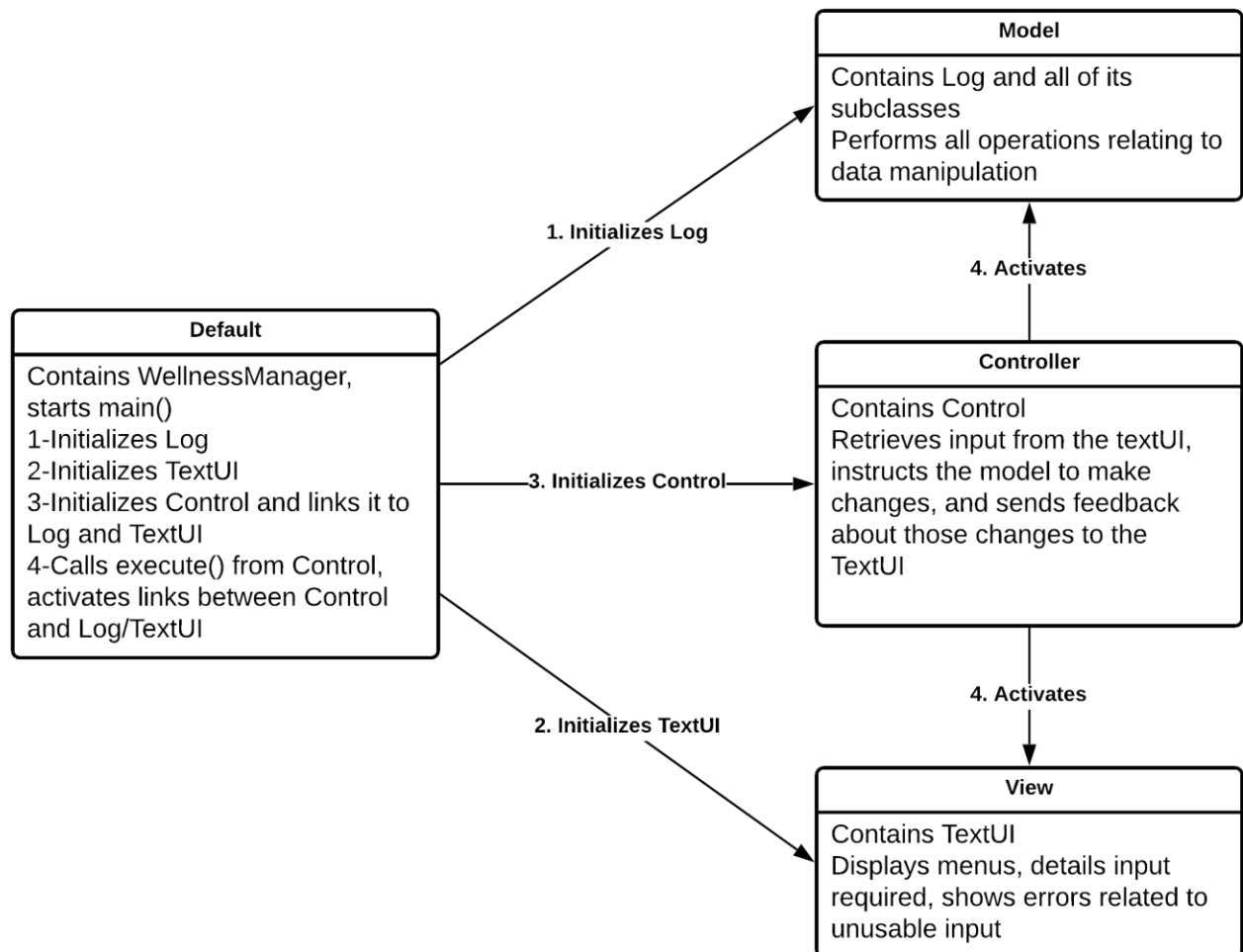
Design Overview

In first designing our system, we started building out the classes for patterns we will be implementing in our design. In doing this, it gave us a solid and organized foundation in which to build the rest of our infrastructure upon. The two patterns being implemented are the Model-View-Controller pattern and the Composite pattern. With “View” and “Controller” focused heavily on UI, and the model being the center in which information is obtained and stored to pass on to the controller. The composite pattern is utilized in creating the food and recipe objects. We also applied the Dependency Inversion Principle with the collection interface as our abstraction level, it separates the log - a high level class- from the two lower level classes: FoodCollection and DailyLog. The reading of a csv file is being handled in those two class as it helps decouple it from the log class.

An advantage of our design is that it is easy for the food and recipes in the csv to be read. It is also easy to make them into objects as the FoodCollection class is injected with the component interface (Food) for the composite design in which we have or basic food and recipes. Another advantage is our low coupling of our system. A disadvantage at the moment is the control class. The control class has a lot of heavy implementation, both in order to validate inputs and do its regular “job” as the controller in the MVC pattern. This makes it the biggest file in the system and lowers cohesion.

Our initial design for our system did not include the layer of abstraction- our collection interface- and reading the csv file was handled by the log class. One of our original ideas to improve coupling was to create a class that would be associated with the log class in order to take some of the functionality load off of it. Another idea was to have a separate class to handle the reading of the csv file and send the information to the log class. This idea was also rejected and then refactored using the collection interface.

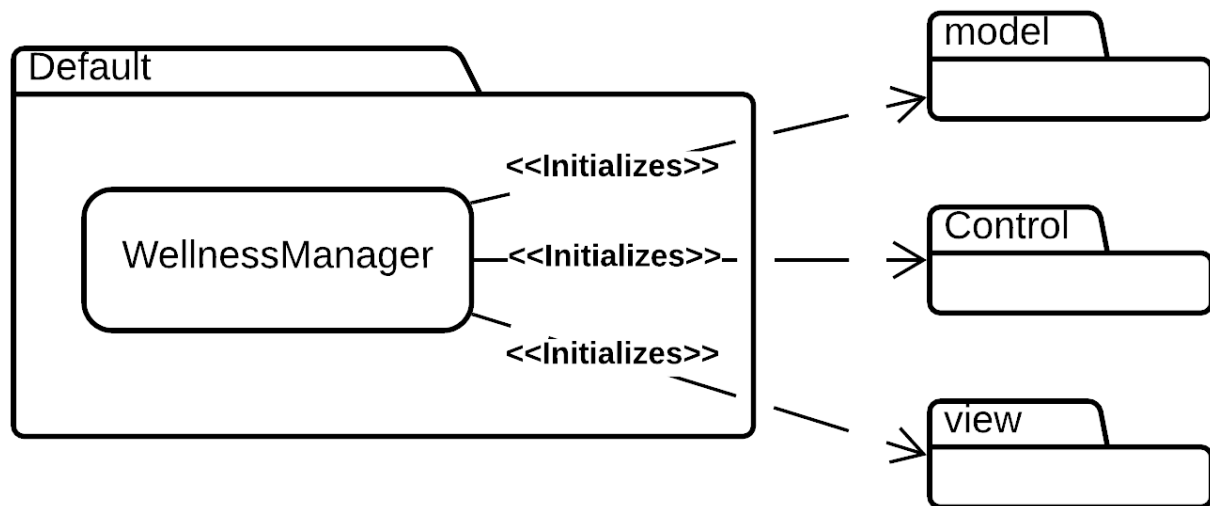
Subsystem Structure



Subsystems

Default Subsystem

Class WellnessManager	
Responsibilities	Create the model objects. Create the text user interface. Display the text user interface so the application can be used.
Collaborators (uses)	model.Log – The primary model class. view.TextUI – The application window. controller.Control – Go between for TextUI and Log



Model Subsystem

Class Log	
Responsibilities	Instantiate a food collection and a daily log collection. Support access to the food collection and daily log collection.
Collaborators (uses)	model.FoodCollection – Collection containing all foods usable in the daily log. model.DailyLog – Collection containing all days recorded in the wellness manager. model.day – Object pulled from the daily log containing data for a given day. model.Food – Object pulled from the food collection containing data about a specific food. java.time.LocalDate – Used for comparing the date variable of Day.

Class SimpleCollection (interface)	
Responsibilities	Provide a generic interface for collections. Defines a means to add, remove, and access objects in the collection. Defines a means to read and write for file access. Defines a mean to show the contents of the collection.

Class DailyLog	
Responsibilities	Support access to the days in the daily log collection. Add, find, modify an existing day in the collection. Reads in a csv log file.
Collaborators (inheritance)	model.SimpleCollection – Generic interface that collections adhere to.
Collaborators (uses)	model.FoodCollection – The collection of foods that can be added to the foods consumed on a day of the daily log. model.Day - The basic type for days in the collection. java.util.ArrayList – Used internally to store the days in the collection java.util.Comparator – Used to sort the array list and keep the daily organized by date. java.util.Scanner, java.io.File, java.io.FileWriter, java.io.BufferedWriter – Used to implement the file read and write functionality of the simple collection interface

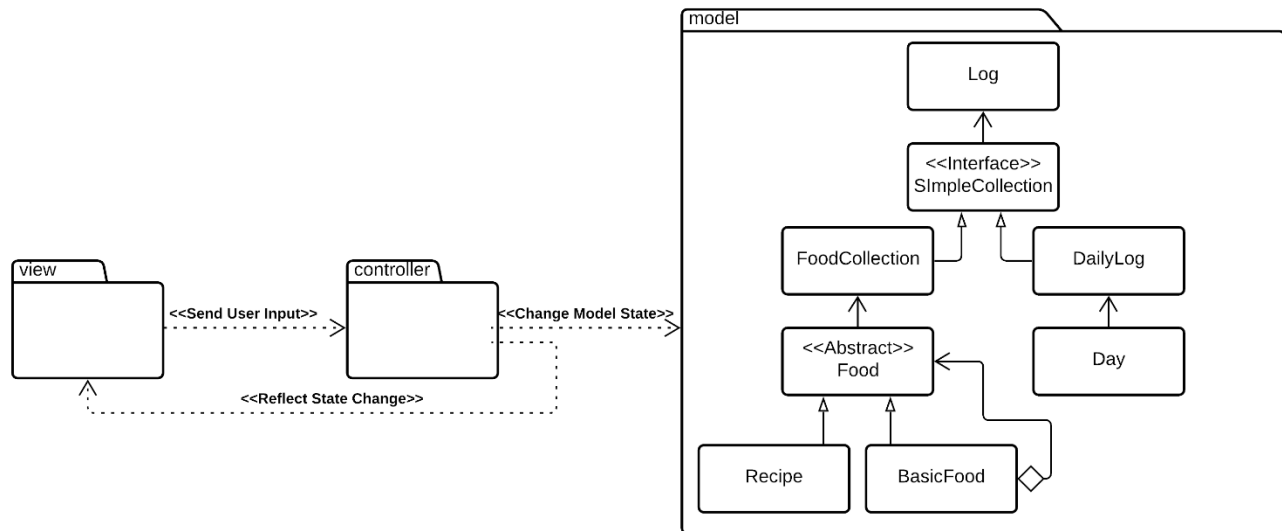
Class Day	
Responsibilities	Represents a day in the daily log collection. Provides access to the date. Provides access to the weight, calorie limit, and list of foods consumed for the given date.
Collaborators (uses)	model.Food – One of the objects being stored in the array list of foods consumed on a given day. java.time.LocalDate – Used as a variable of day. Identifies a specific date. java.util.ArrayList – The internal structure to store foods consumed on a given day. java.util.Map – Used to store map entries of foods and their corresponding servings in the array list. java.util.AbstractMap – Used to create a new entry for the array list.

Class Food (abstract)	
Responsibilities	Provide a generic interface to all food types in the food collection. Includes the food name as a unique ID. Can retrieve nutrition information regardless of underlying food type.

Class FoodCollection	
Responsibilities	Support access to the foods in the food collection. Add, find, delete an existing food in the collection. Reads in a csv food file.
Collaborators (inheritance)	model.Collection – Generic interface that collections adhere to.
Collaborators (uses)	model.BasicFood – One of the food types. Used during file reading to build the collection. model.Recipe - One of the food types. Used during file reading to build the collection. model.Food - The basic type for all different foods in the collection. java.util.LinkedHashMap – Internal storage structure for food collection java.util.Scanner, java.io.File, java.io.FileWriter, java.io.BufferedWriter – Used to implement the file read and write functionality of the simple collection interface

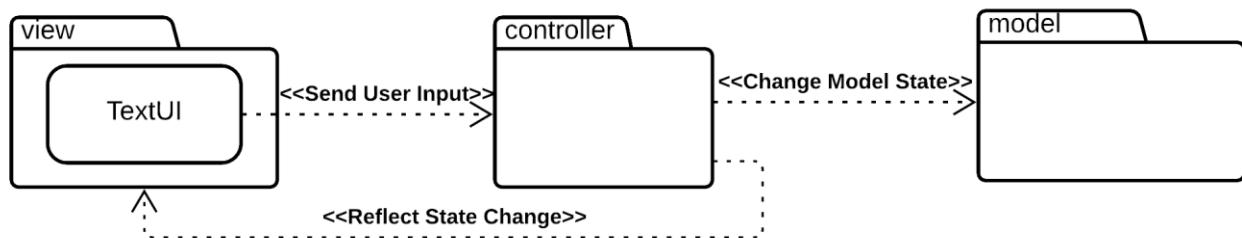
Class BasicFood	
Responsibilities	Represents a basic food in the food collection. Provides access to the name of the food. Provides access to the number of calories, grams of fat, grams of carbs, and grams of protein per serving.
Collaborators (inheritance)	model.Food – Abstract class that all food items adhere to.

Class Recipe	
Responsibilities	Represents a recipe in the food collection. Provides access to the name of the recipe. Provides access to a list with n entries containing name/count pairs where name is the name of the basic food or sub-recipe and count is the number of servings.
Collaborators (inheritance)	model.Food – Abstract class that all food items adhere to.
Collaborators (uses)	java.util.HashMap – Internal storage structure for foods that make up this recipe. java.util.Set – Used to grab the keys of the internal storage structure



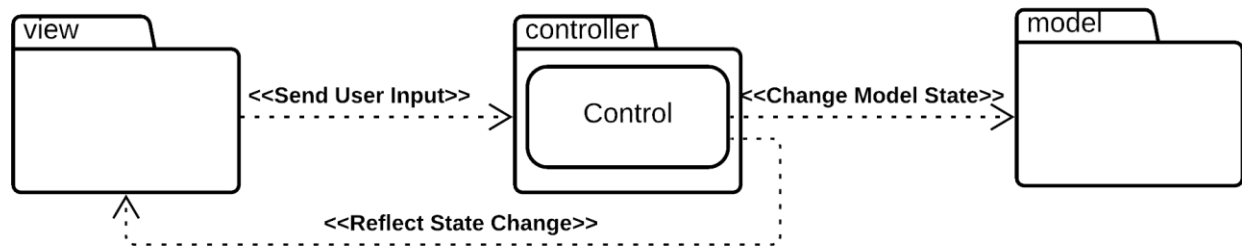
View Subsystem

Class TextUI	
Responsibilities	Display the menu. Display information requested from the model.
Collaborators (uses)	model.Log – The model the TextUI refers to



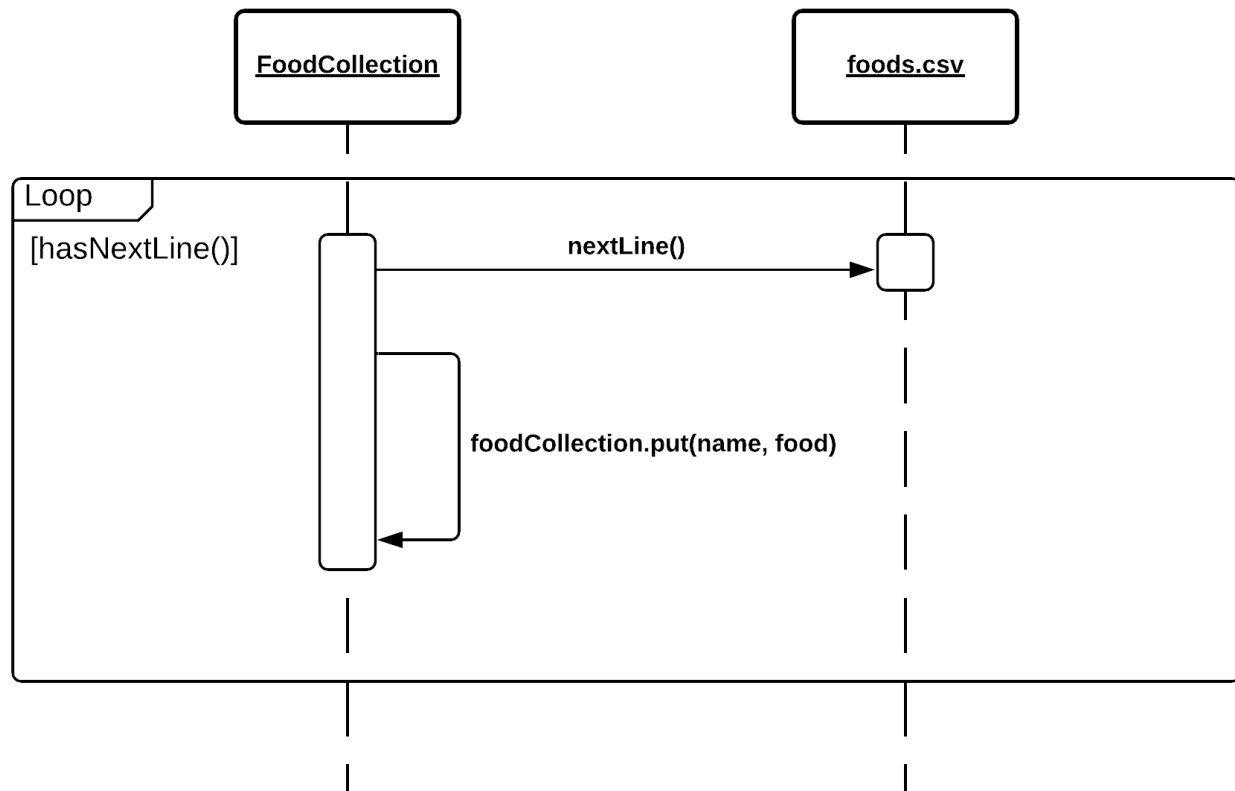
Controller Subsystem

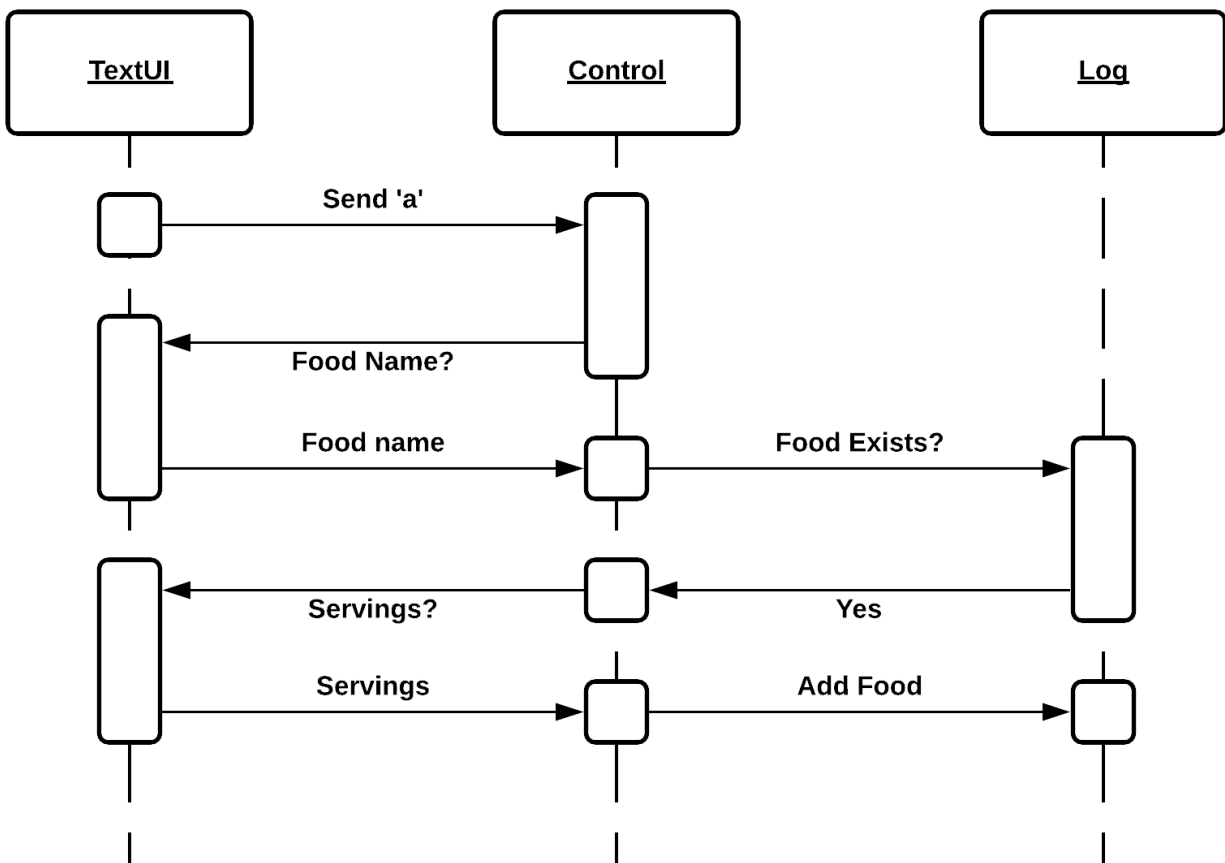
Class Control	
Responsibilities	Retrieve user input. Update the model and view.
Collaborators (uses)	view.TextUI – Displays changes processed by the controller model.Log – Manipulated by the controller model.Food, model.Recipe, model.BasicFood, java.time.LocalDate – Used for data comparison/verification. java.util.Scanner – Retrieve user input from the text interface. java.util.regex.Pattern – Used for input verification.



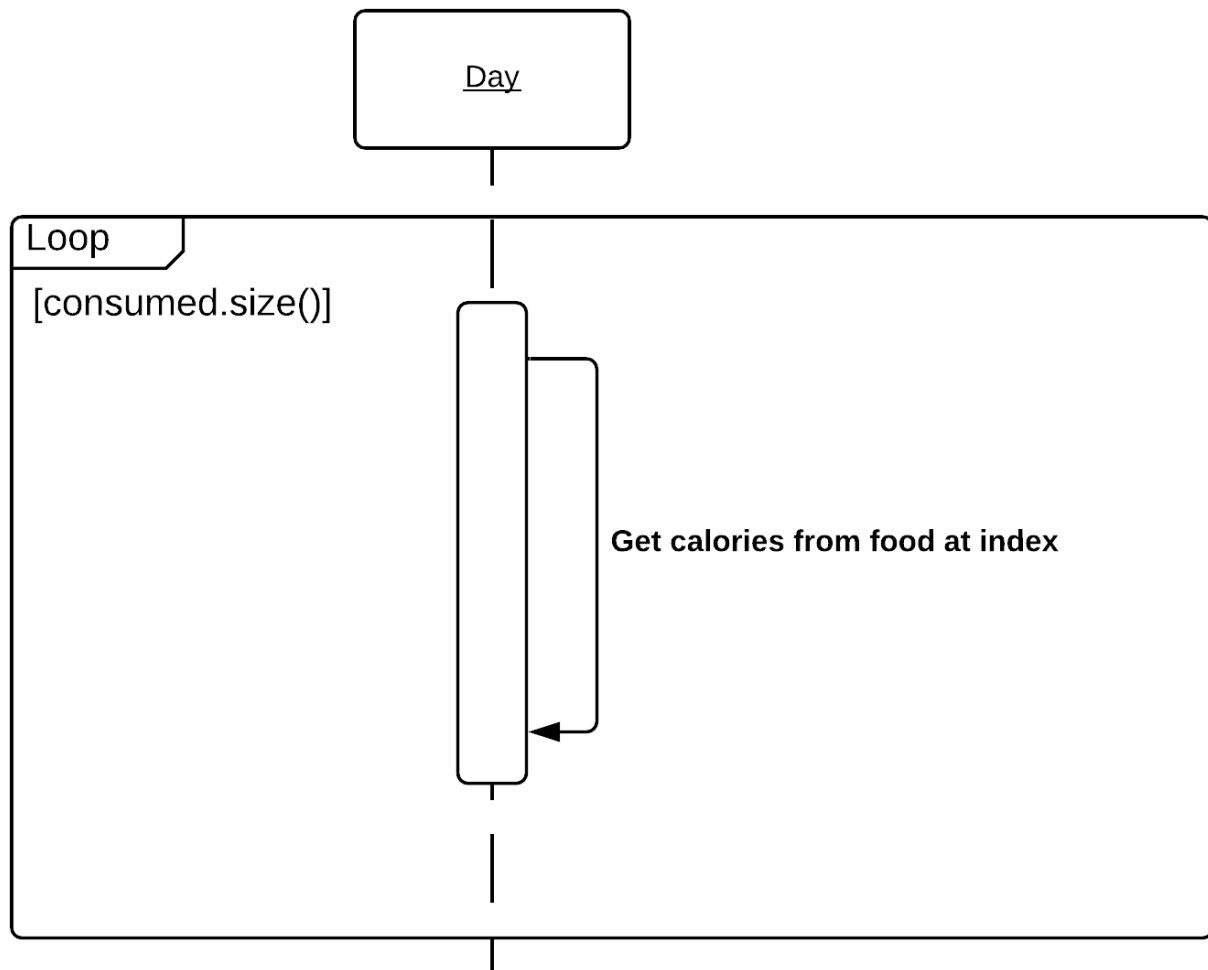
Sequence Diagrams

Sequence Diagram #1: Reads in a food database of three basic foods, a recipe that contains two of the basic foods, and a recipe that contains the first recipe and the remaining food.



Sequence Diagram #2: Add two servings of a food to the log entry for the current date.

Sequence Diagram #3: Compute the total number of calories for the current date assuming the log consists of a basic food and a recipe consisting of two basic foods.



Pattern Usage

Pattern #1 Composite

Basic foods and recipes inherit from the abstract food object so that they can be stored in one collection for use by higher level logic in the application.

Composite Pattern	
Component	Food
Leaf	BasicFood
Composite	Recipe

Pattern #2 MVC

The application is organized using the model view controller pattern to recognize commands to the controllers, manipulate the thermometer as the model, and reflect changes via the views.

MVC Pattern	
Model	Log SimpleCollection DailyLog FoodCollection Day Food BasicFood Recipe
View	TextUI
Controller	Control