DIET MANAGER 2.0

Project Design Document

TEAM A
Matthew Atwell <ma3505@rit.edu>
David Camacho <dac6892@rit.edu>
Nathan Hinds <nah7111@rit.edu>
Brendon Strowe <Brendon.Strowe@mail.rit.edu>
Jenna Tillotson <jet7469@rit.edu>

Project Summary

The Diet Manager application aims to provide a way for users to manage day-to-day the foods they consume, the exercises they perform, and track metrics over time. Users can enter daily logs to report their daily weight, calorie limit, food intake, and exercises performed. From these logs, users can view metrics in the form of bar graphs and calculated values.

In order for users to create their log entries, users are able to create a collection of various foods and exercises. Foods are created with associated dietary information (i.e. calories, grams of fat, carbohydrates, and protein). Users can then combine these foods into recipes made up of either those basic foods or other, simpler recipes. Exercises are created with the number of calories that would be burned by a 100lb person exercising for 60 minutes.

From the food and exercise collection the user creates, they are able to make daily entries into a log of their weight, their calorie limit, the servings of foods they consumed, and the exercises performed and the time spent on those exercises on a given day. If a user does not enter a weight or calorie limit for a given day, the program will use the most recently entered values.

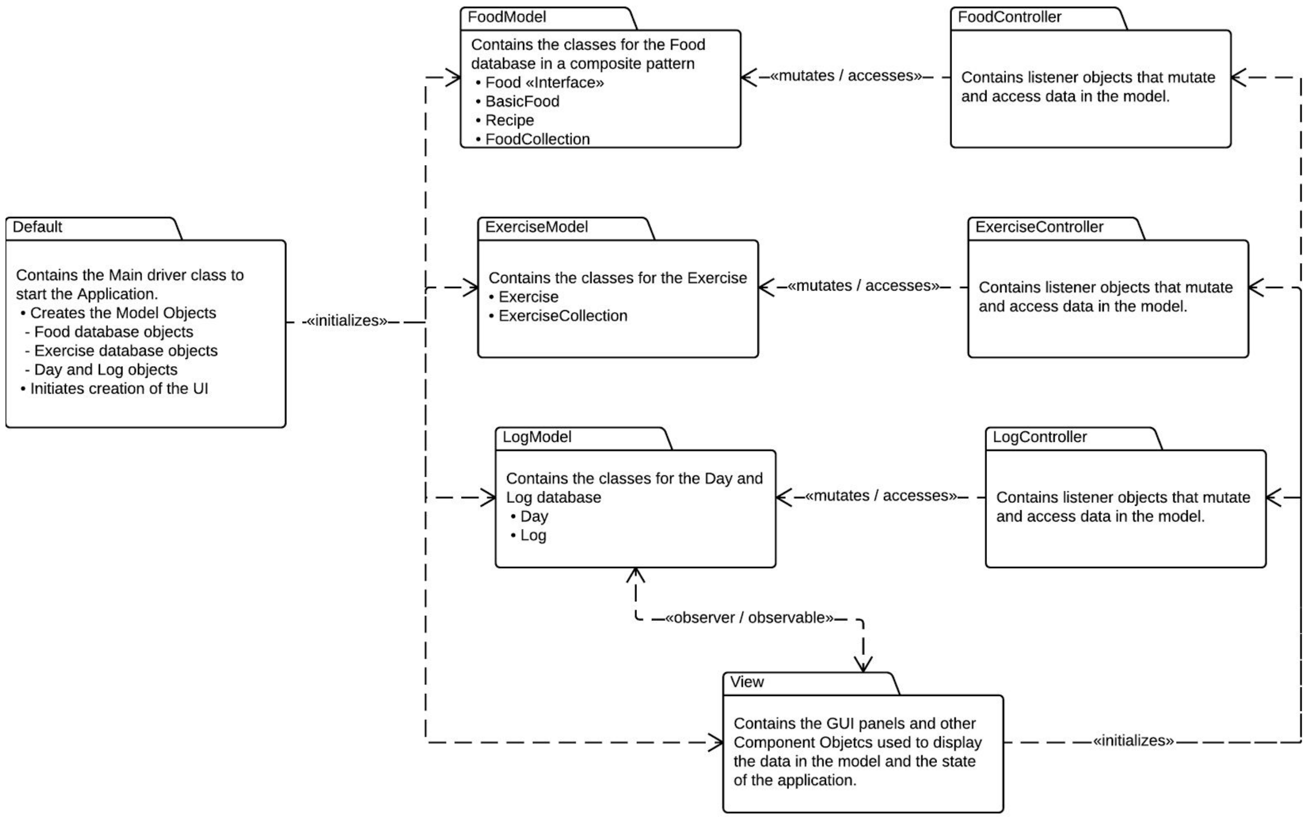
The application will then be able to generate a bar chart which displays the nutrition consumed for a day in grams of fat, carbohydrates, and protein. The application will also calculate the total calories consumed for a day, the number burned during exercise, and the difference. The application will then determine and display if the user is over or under their calorie limit goal.

Users will be able to save out their information at any time to CSV-formatted files as well as read them in.

Design Overview

Our design from the preliminary design sketch all the way to the final implementation did not change much. In the beginning we started out with the composite pattern using Food, BasicFood and Recipes and putting them each of them in a component, leaf and composite, respectively. From there we were able to use that pattern and incorporate it into our MVC pattern. Since we are using the MVC pattern it automatically gives us a good separation of concerns design where we will have classes that have specific function in the Model, View and Controller pattern. Initially, we had the FoodCollection collect BasicFood and Recipe Objects, but later changed it to collect only Food objects as it was easier to implement. This would lower our coupling in the Model subsystem as FoodCollection will only have to depend on Food to get the objects. Our cohesion for our classes are not as high as our coupling is pretty low based on our design pattern. New additions to our design include adding an exercise which includes updating it and showing what exercises done for a specific day. Our design allows us to implement this new addition fairly easy as it just needs its own View, Model and Controller classes.

Subsystem Structure


FoodModel
Contains the classes for the Food database in a composite pattern
• Food «Interface»
• BasicFood
• Recipe
• FoodCollection
FoodController
Contains listener objects that mutate and access data in the model.
«mutates / accesses»
Default
Contains the Main driver class to start the Application.
• Creates the Model Objects
- Food database objects
- Exercise database objects
- Day and Log objects
• Initiates creation of the UI
«initializes»
ExerciseModel
Contains the classes for the Exercise
• Exercise
• ExerciseCollection
ExerciseController
Contains listener objects that mutate and access data in the model.
«mutates / accesses»
LogModel
Contains the classes for the Day and Log database
• Day
• Log
LogController
Contains listener objects that mutate and access data in the model.
«mutates / accesses»
«observer / observable»
View
Contains the GUI panels and other Component Objetcs used to display the data in the model and the state of the application.
«initializes»

Subsystems

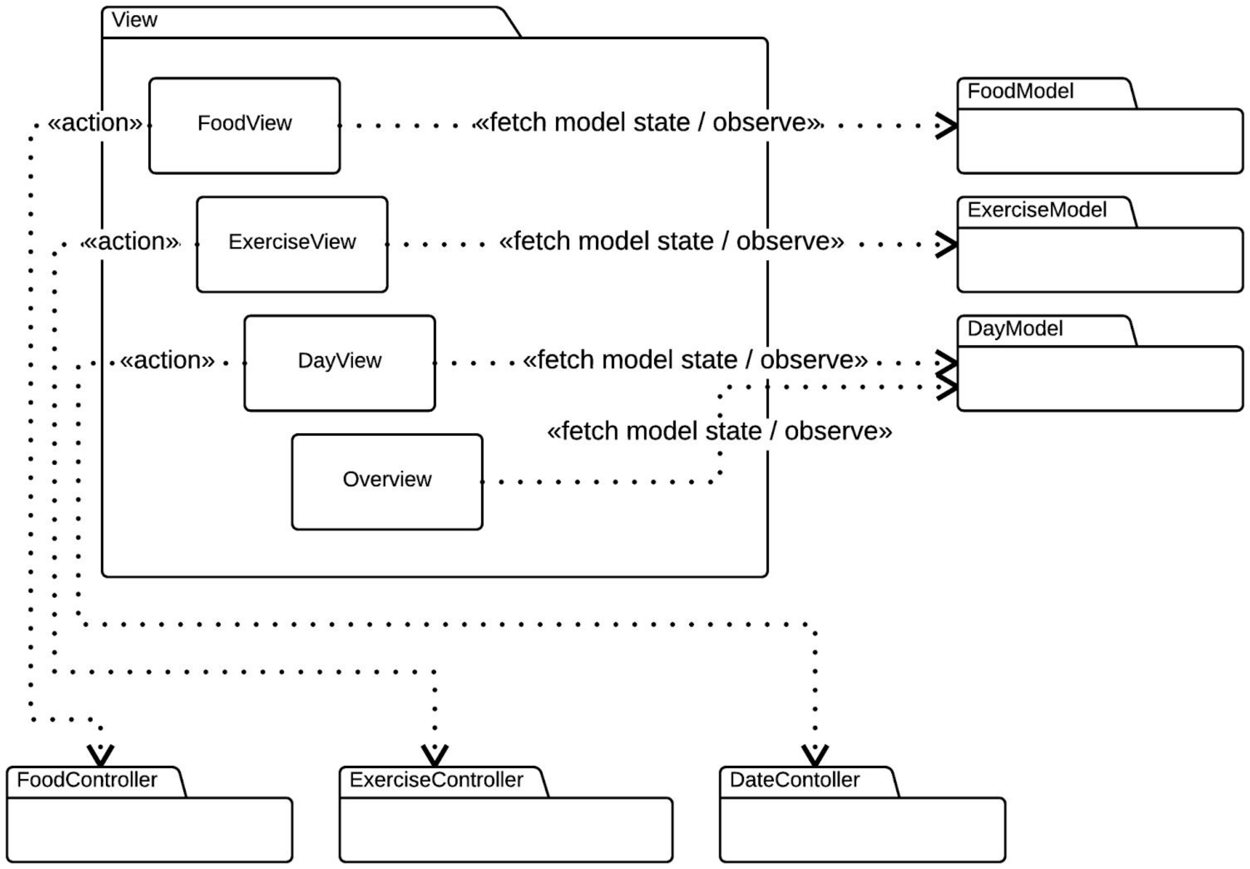
View Subsystem

Class FoodView	
Responsibilities	Displays currently stored BasicFoods and Recipes through a GUI. Gives means to modify Food objects
Collaborators (implements)	**java.util.Observer** - to observe the collection of Food
Collaborators (uses)	**FoodModel.FoodCollection** - The primary model class for accessing Food Objects **FoodController** - How modifications to Food objects are made.

Class ExerciseView	
Responsibilities	Displays currently stored Exercise objects through a GUI. Gives means to modify Exercise objects
Collaborators (implements)	**java.util.Observer** - to observe the collection of Exercises
Collaborators (uses)	**ExerciseModel.ExerciseCollection** - The primary model class for accessing Exercise Objects **ExerciseController** - How modifications to Exercise objects are made.

Class DateView	
Responsibilities	Display and modify information through a GUI about a Day object. Give access to viewing a list of other Day objects collected in the Log object.
Collaborators (implements)	**java.util.Observer** - to observe the Log of Days
Collaborators (uses)	**LogModel.Day** - The primary model class **LogModel.Log** - Where other Day Objects are retrieved from **LogController** - How modifications to the Day objects are made

Class Overview	
Responsibilities	Displays metrics and bar graph for the currently selected Day object
Collaborators (implements)	**java.util.Observer** - to observe the Log of Days
Collaborators (uses)	**LogModel.Day** - The primary model class **java.awt.Color** - control the color of the bar graph. **java.awt.Graphics** - control draw the overall graph. **java.awt.Rectangle** - scale the graph to the actual canvas size.


View
FoodView
ExerciseView
DayView
Overview
«action»
«action»
«action»
«fetch model state / observe»
«fetch model state / observe»
«fetch model state / observe»
«fetch model state / observe»
FoodModel
ExerciseModel
DayModel
FoodController
ExerciseController
DateContoller

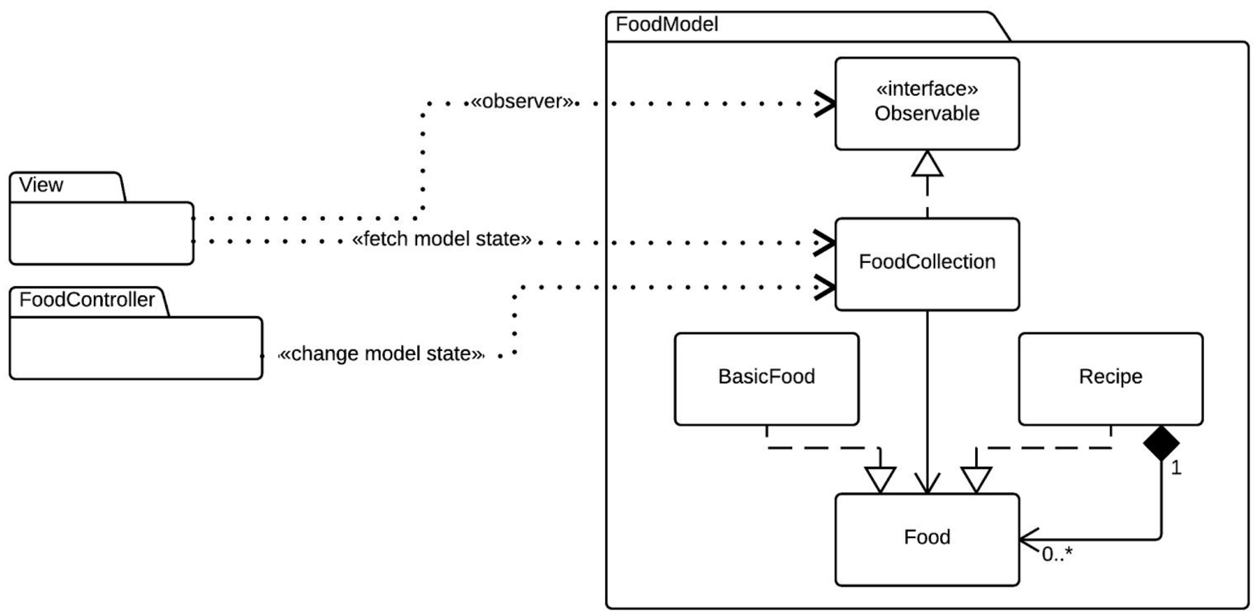
FoodModel Subsystem

Class Food	
Responsibilities	Methods and properties which will be used by all Food Objects, both BasicFood and Recipe Objects.

Class BasicFood	
Responsibilities	Defines Food Objects that the user can use in recipes as well as consume.
Collaborators (inherits)	**FoodModel.Food** - All BasicFood objects are Food objects

Class Recipe	
Responsibilities	A collection of Food Objects that can be made up of BasicFood as well as other Recipes.
Collaborators (inherits)	**FoodModel.Food** - All BasicFood objects are Food objects
Collaborators (uses)	**FoodModel.Recipe** - Recipes can be made up of BasicFood objects

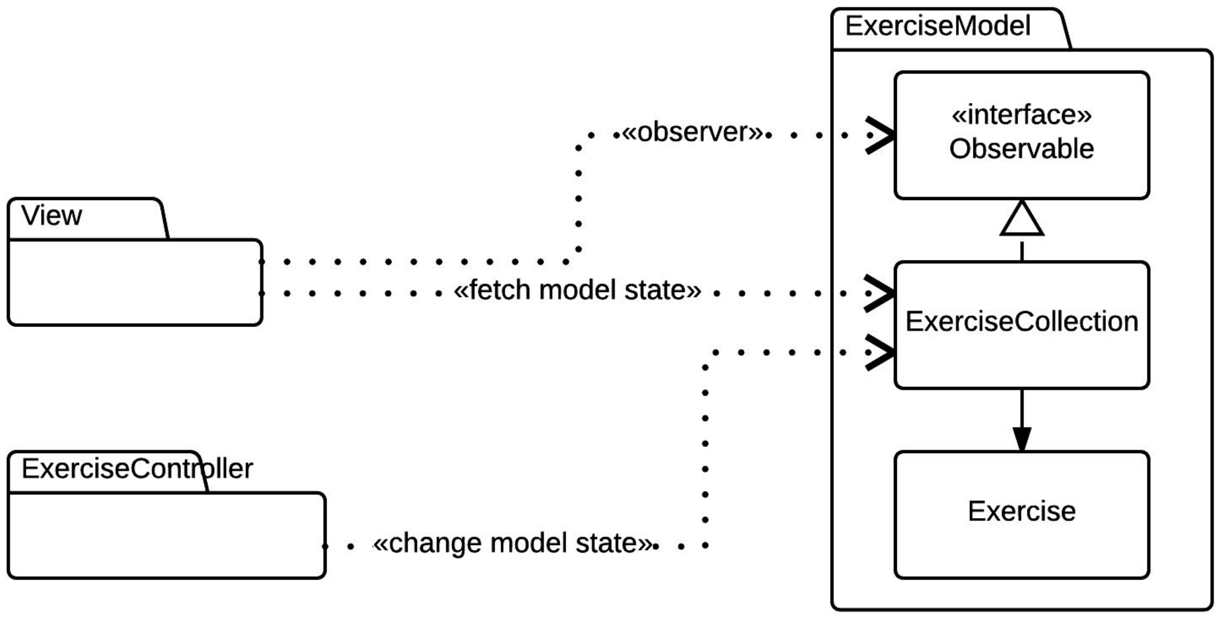
Class FoodCollection	
Responsibilities	Lists a user's BasicFoods and Recipes that they create.
Collaborators (inherits)	**java.util.Observable** - so FoodCollection changes can be observed by others.
Collaborators (uses)	**FoodModel.Food** - FoodCollection contains a collection of Food objects. **java.util.Observer** - for notifications of FoodCollection changes to views.


FoodModel
«interface»
Observable
View
«observer»
«fetch model state»
FoodCollection
FoodController
«change model state»
BasicFood
Recipe
1
Food
0..*

ExerciseModel Subsystem

Class ExerciseCollection	
Responsibilities	Defines Exercise objects that the user can add to Day objects as performed.
Collaborators (uses)	**ExerciseModel.Exercise** - All Exercise objects are stored in the ExerciseCollection **java.util.Observer** - for notifications of ExerciseCollection changes to views.

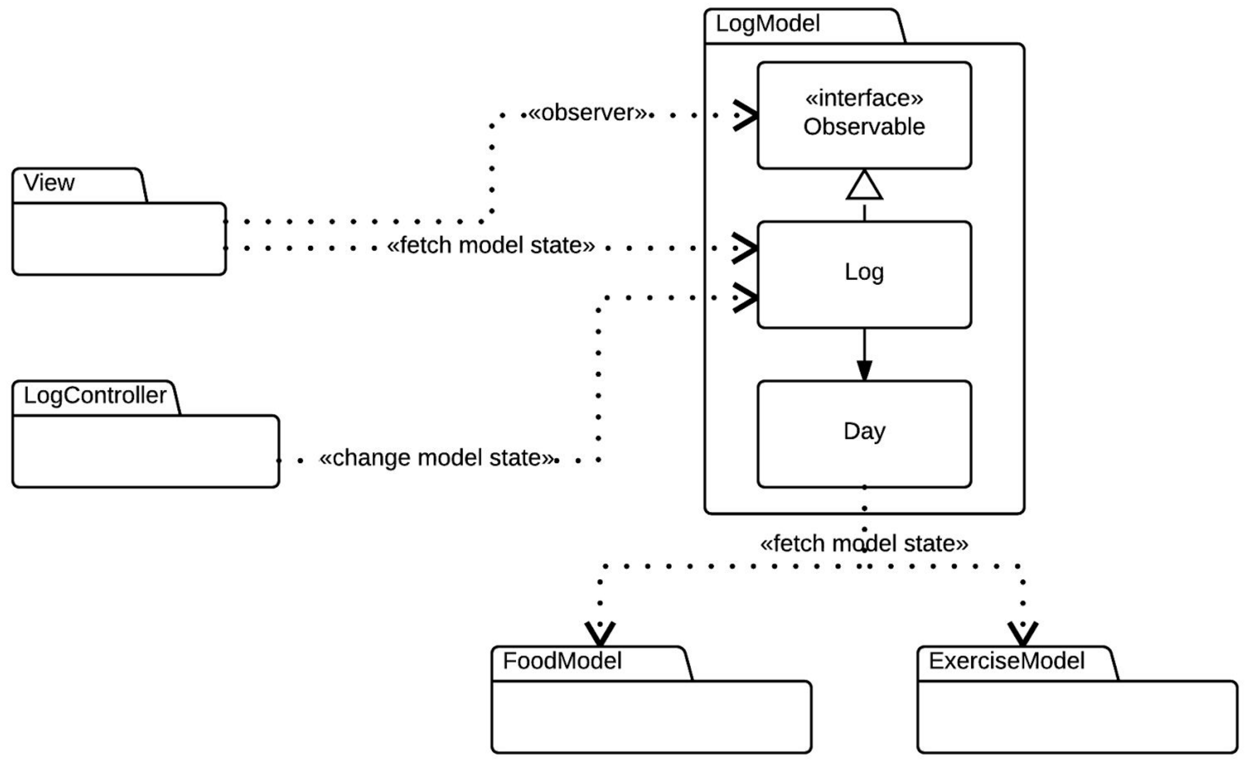
Class Exercise	
Responsibilities	Exercise objects which store a name and a value which is the calories burned by a 100lb person performing the exercise for 60 minutes.
Collaborators (inherits)	**FoodModel.Food** - All BasicFood objects are Food objects



LogModel Subsystem

Class Log	
Responsibilities	Contains a list of all Day objects that have been created.
Collaborators (inherits)	**java.util.Observable** - so Log changes can be observed by others.
Collaborators (uses)	**LogModel.Day** - Day objects that are stored in the Log. **java.util.Observer** - for notifications of Log changes to views.

Class Day	
Responsibilities	Handles a set weight and calorie limit for a given Day. Tracks the list of Food a user has consumed on a given Day. Tracks the list of Exercises a user has performed on a given Day.
Collaborators (uses)	**FoodModel.Food** - Used to reference Food objects stored. **ExerciseModel.ExericeCollection** - Used to reference Exercise objects stored.



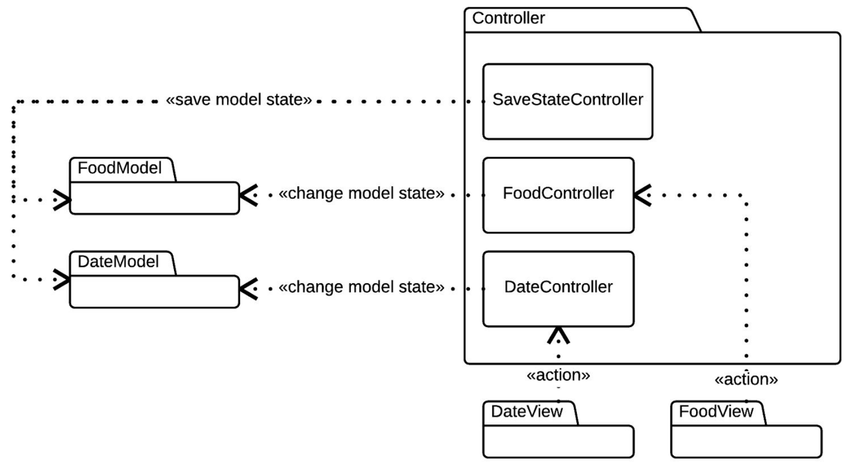
Controller Subsystem

Class LogController	
Responsibilities	Gets the Day objects from the log CSV. Passes that information to the DateView. Populates the Log of Day objects Controls the creation of new Day objects Controls the modification of Day objects Based on the date it updates or creates the exercise in the Log.
Collaborators (uses)	**LogModel.Log** - place for storing Day objects while the application runs **LogModel.Day** - Objects to be manipulated and read **View.DateView** - Where the read Day objects are shown

Class FoodController	
Responsibilities	Allow creation of new BasicFood or Recipe objects. Gets the BasicFood Objects and Recipe objects that are in the food collection class and passes them to the FoodView.
Collaborators (uses)	**FoodModel.FoodCollection** - place for storing Food objects while the application runs **FoodModel.Food** - Objects to be manipulated and read **view.FoodView** - Where the read Food objects are shown

Class ExerciseController	
Responsibilities	Gets the information from the ExerciseView and passes it to the ExerciseModel. Shows the information that was in the ExerciseCollection to the ExerciseView.
Collaborators (uses)	**ExerciseModel.ExerciseCollection -** stores the Exercise objects during runtime **ExerciseModel.Exercise -** the objects used to store, show and change **view.ExerciseView -** where we get the information to make Exercise objects

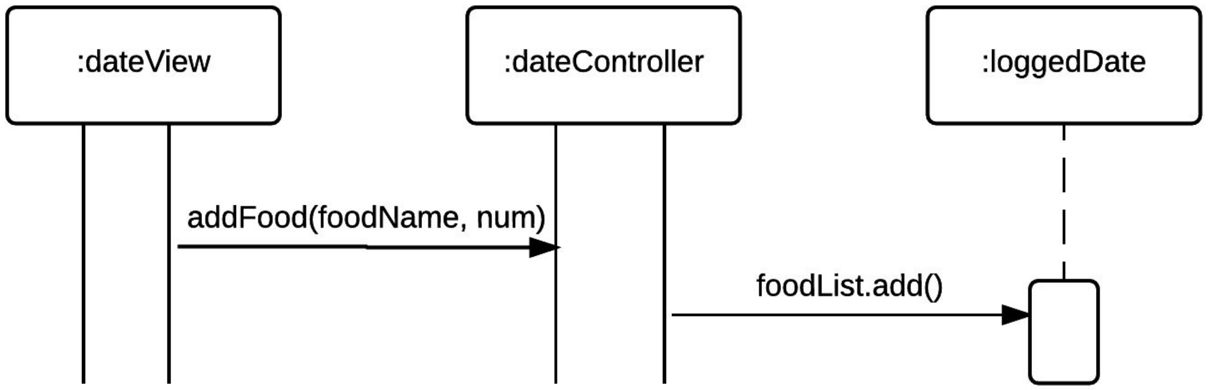
Class SaveStateController	
Responsibilities	Gets the information from the Log and FoodCollection lists and saves them out to CSV files upon invocation from any UI View.
Collaborators (uses)	**LogModel.Log** - where all of the Day data is gathered from to save. **FoodModel.FoodCollection** - where all of the Food data is gathered from.


Controller
SaveStateController
FoodController
DateController
«save model state»
FoodModel
«change model state»
DateModel
«change model state»
«action»
«action»
DateView
FoodView

Sequence Diagrams

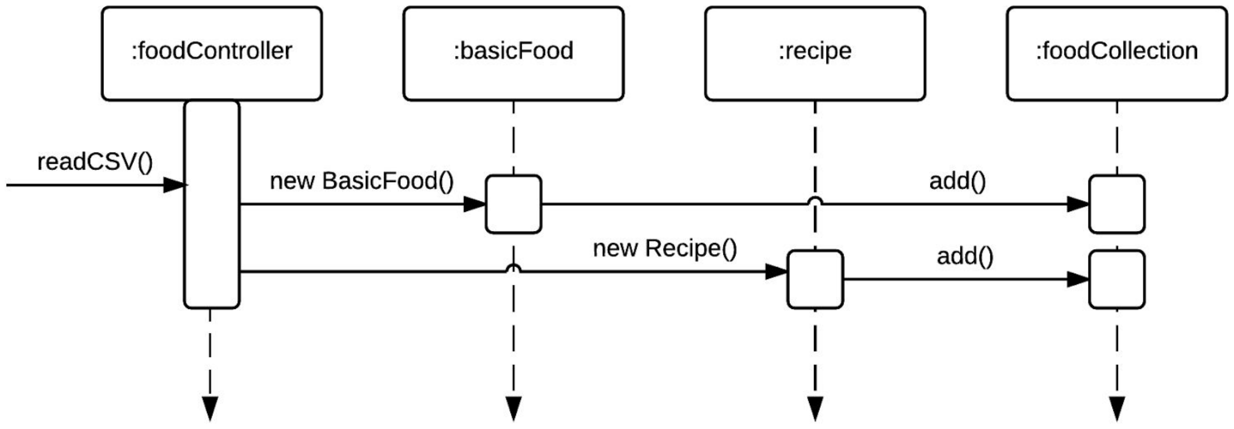
Sequence 1 - Adding a Food Entry

User adds a new food entry on the current date



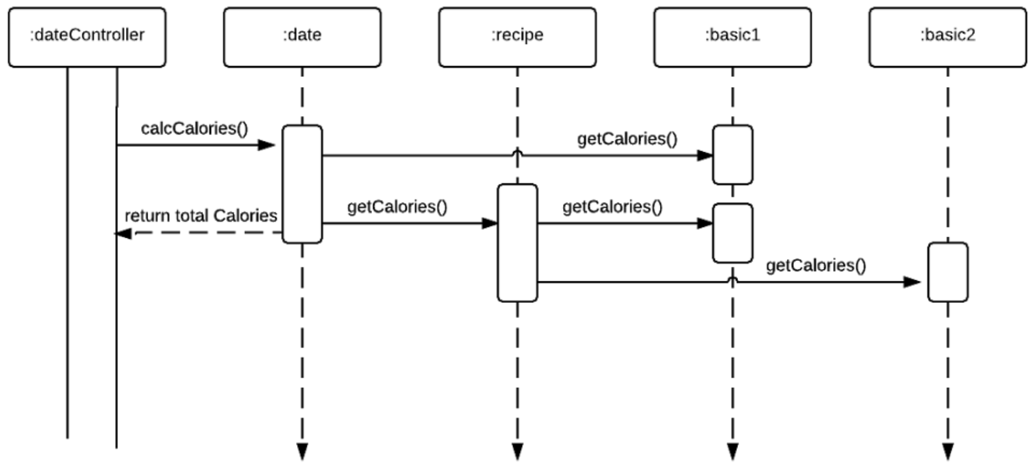
Sequence 2 - Creating BasicFood or Recipe Objects

User creates a new food or recipe object to be entered into the database



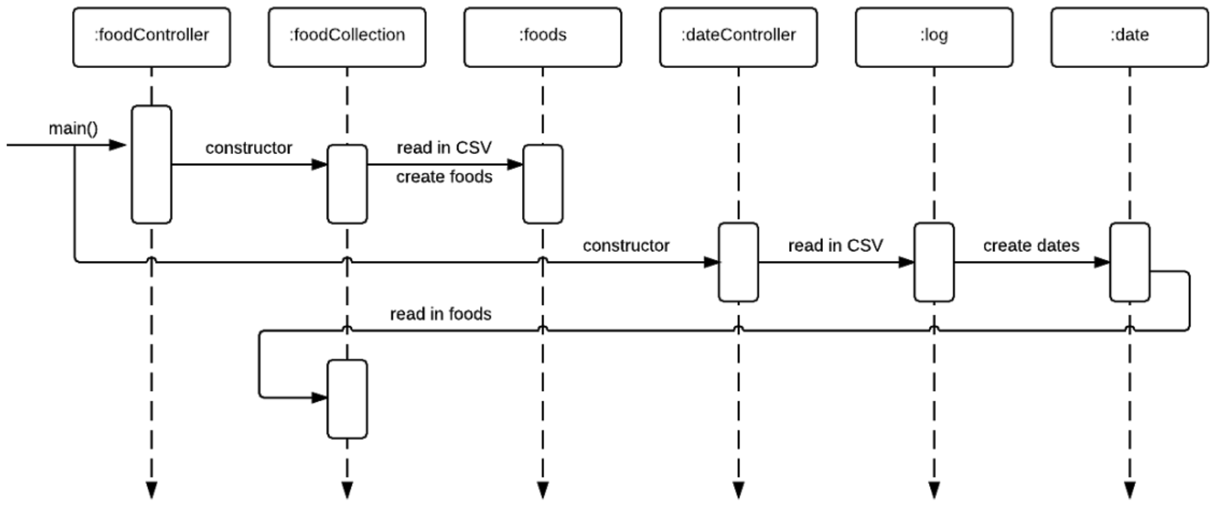
Sequence 3 - Calculating Calories by Date

Calculates the total calories consumed from the food entries for a given date



Sequence 4 - App Initialization

Loads in the current iteration of the data file and loads in the past date entries from the log



Pattern Usage

Pattern #1 MVC

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

MVC Pattern	
Model	Food BasicFood Recipe FoodCollection Exersice ExersiceCollection Day Log
View	FoodView ExerciseView LogView Overview
Controller	LogController FoodController ExerciseController SaveStateController

Pattern #2 Composite

The FoodModel subsystem is organized using the composite pattern. Food is the interface in which the other classes inherit from. BasicFood objects are the leaves as they cannot contain any other objects. Recipes can (and will) consist of either BasicFood objects or other Recipes, which in turn are also made up of other BaiscFood or recipe objects and soforth.

Composite Pattern	
Composite	Recipe
Component	Food
Leaf	BasicFood