Wellness Manager

Project Design Document

CTeam

John Mule <jmm1872@rit.edu>
Justin Nauta <jrn2778@rit.edu>
Andrea Pallotta <ap4534@rit.edu>
Miguel Galan <mag3561@rit.edu>
Dev Bhatt <dmb4086@rit.edu>

Project Summary

This project is a calorie tracking software that will provide a collection of basic foods, which can be combined to create recipes in the system, with the ability to add custom foods and recipes by providing macros information.
It also gives you a personalized experience as it gives you data like your caloric distribution based on your height, weight etc**.**

Design Overview

For the design, we chose to go with the Composite and Observer Patterns. We wanted to notify the view (What the user looks at) of any state change from our backend to be more inline with the MVC design approach. Moreover, this does not allow the view to be updated autonomously without user input. We also felt the food and recipe data objects could be processed recursively so we felt a composite pattern for these two objects made a lot of sense.

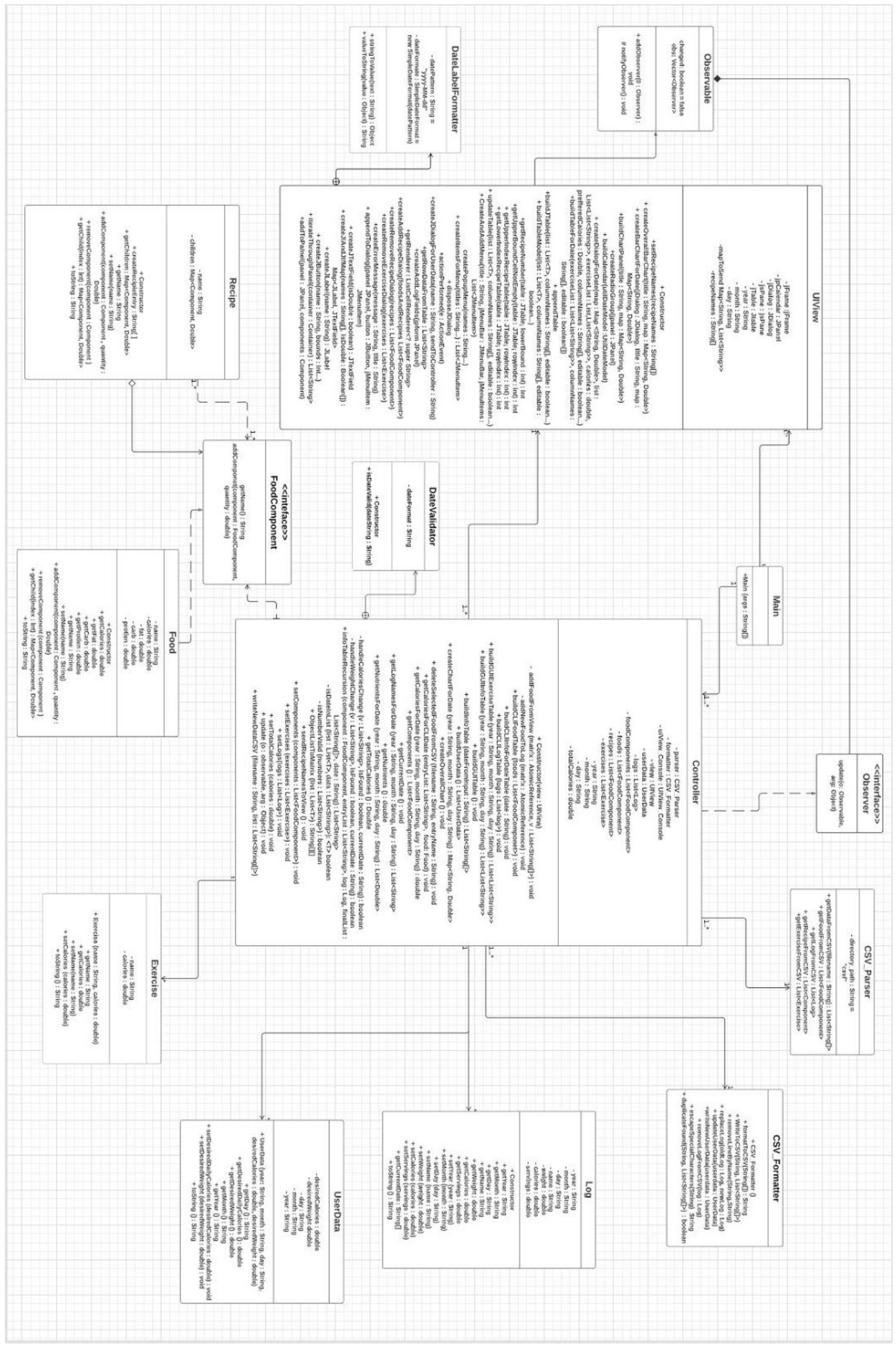
We also debated making additional controllers for each data type to increase cohesion but that would add a lot of additional workload and a lot of refactoring. Ironically too adding these additional managers for each object type would actually wind up increasing the programs coupling so we decided against it.

With our approach it pretty much perfectly mimics the MVC pattern and its flow of information.

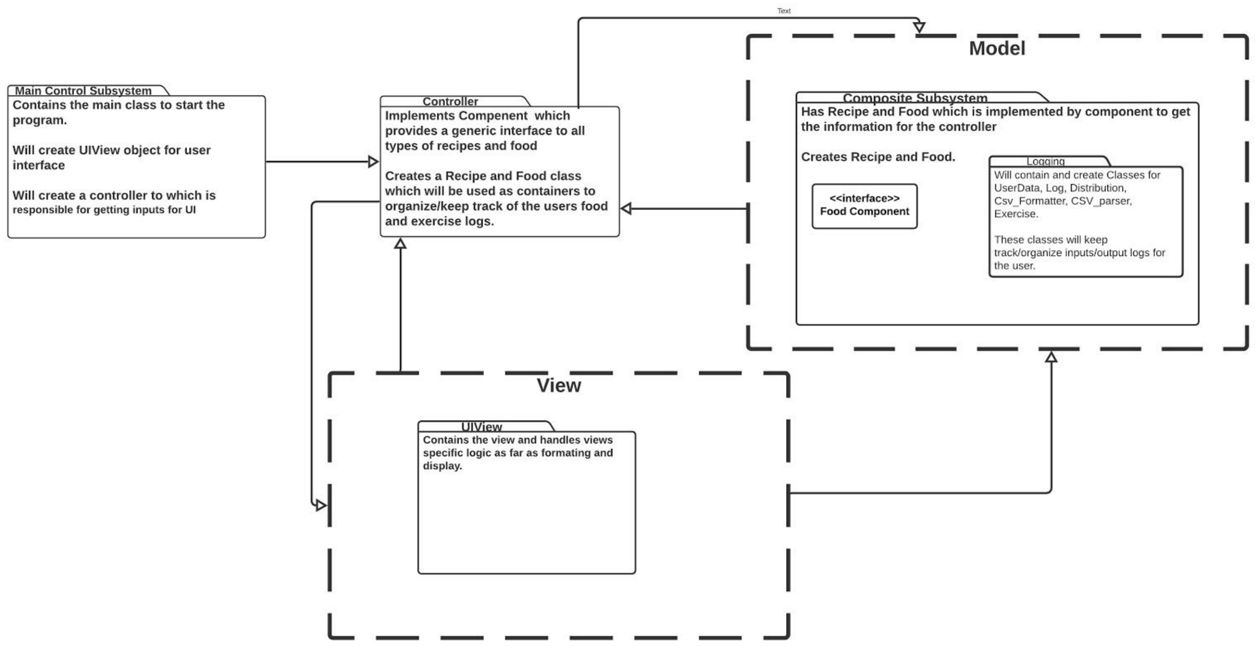
From the beginning, this project incorporated the MVC architectural pattern with the embedded use of the composite pattern which displays a basic user interface that is sufficient to achieve the functionality like adding/logging food items. Moreover, The user data log, and exercise classes are all handled individually in a manner that increases the cohesion and decreases coupling.

UML Class Diagram

The UML diagram was too large to fit an image of it in the design doc properly. Please follow the link and select the “V.2 UML” tab at the bottom left. (As a backup we also included a separate pdf in the src folder of the UML)
https://lucid.app/invitations/accept/8013998a-c51f-4115-83a7-c6083c168e44



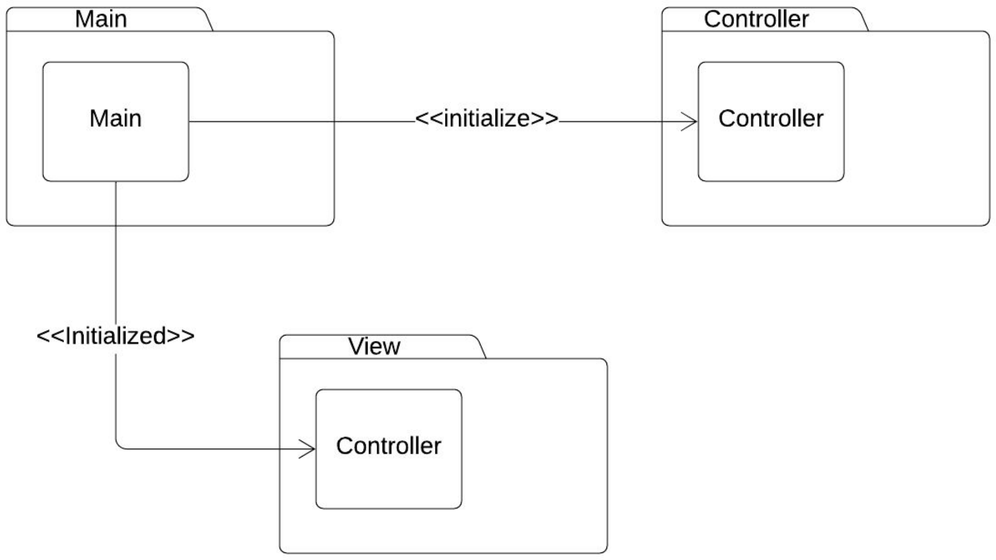
Subsystem Structure


Text
Model
Main Control Subsystem
Contains the main class to start the program.
Will create UIView object for user interface
Will create a controller to which is responsible for getting inputs for UI
Controller
Implements Compenent which provides a generic interface to all types of recipes and food
Creates a Recipe and Food class which will be used as containers to organize/keep track of the users food and exercise logs.
Composite Subsystem
Has Recipe and Food which is implemented by component to get the information for the controller
Creates Recipe and Food.
<<interface>>
Food Component
Logging
Will contain and create Classes for UserData, Log, Distribution, Csv_Formatter, CSV_parser, Exercise.
These classes will keep track/organize inputs/output logs for the user.
View
UIView
Contains the view and handles views specific logic as far as formating and display.

Subsystems

Main Subsystem

Class Main	
Responsibilities	Contains the main method
Collaborators (uses)	Controller UIView

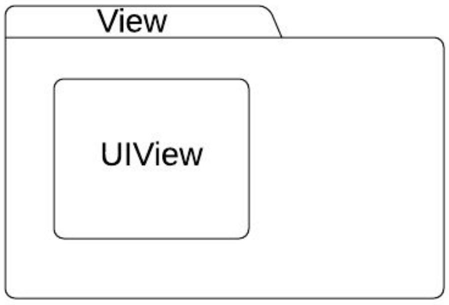


Add observer stuff in the diagram

View Subsystem

Class UIView	
Responsibilities	Contains the view and handles the view’s specific logic
Collaborators (uses)	Observable

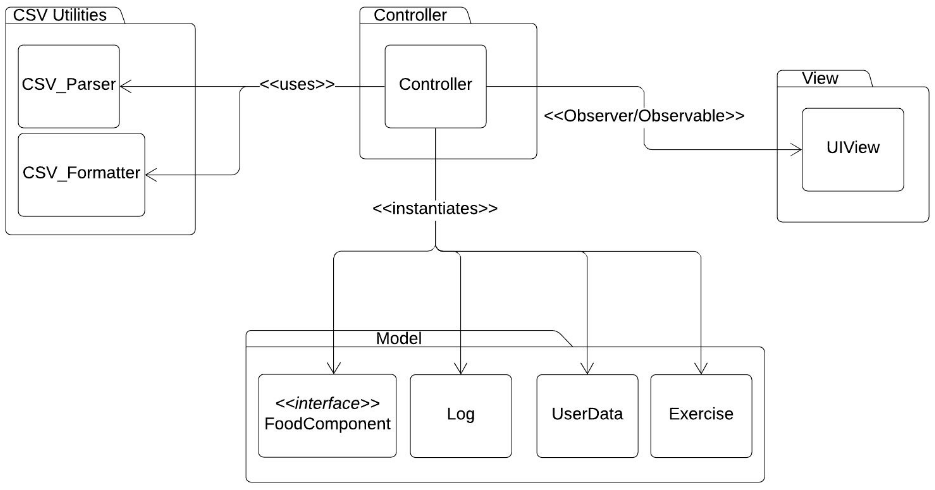
Class Observable	
Responsibilities	Gets informed of changes by the Observer interface
Collaborators (uses)	Controller Observer



Controller Subsystem

Class Controller	
Responsibilities	The controller is responsible for getting inputs from UIView and updating both models and views.
Collaborators (uses)	Main: Starts the program UIView: contains the view Component: interface for recipe and food UserData: contains information about the user (goals, nutritional distribution, etc.) Log: contains information about the user's activity (daily intakes, calories, etc..) CSV_Parser: Parses a CSV file CSV_Formatter: Writes to CSV file

Class *Observer <<interface>>*	
Responsibilities	Informs of changes in observable objects
Collaborators (uses)	Controller Observable



Model Subsystem

Class *FoodComponent <<interface>>*	
Responsibilities	Provide a generic interface to all types of recipes and foods
Collaborators (uses)	Food Recipe

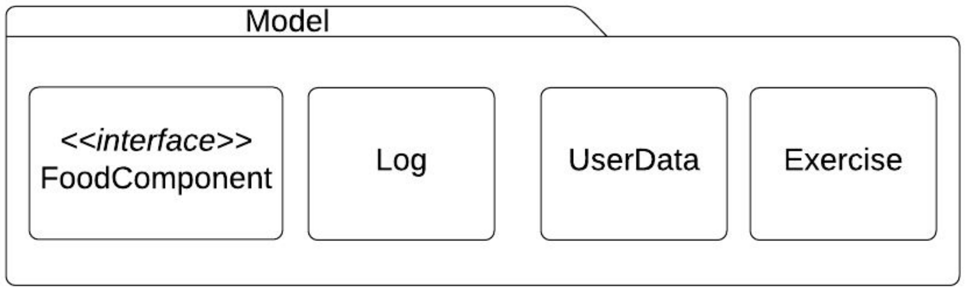
Class Recipe	
Responsibilities	Container for foods as organized into a recipe. Acts as a composite object
Collaborators (uses)	FoodComponent

Class Food	
Responsibilities	Keeps track of the name and nutritional info of a specific food. Acts as a leaf object.
Collaborators (uses)	FoodComponent

Class Log	
Responsibilities	Specifically handles logging of entries
Collaborators (uses)	

Class UserData	
Responsibilities	Handles the storing and editing of user data
Collaborators (uses)	

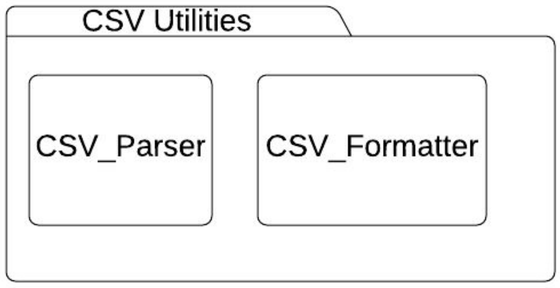
Class Exercise	
Responsibilities	Handles the storing and editing of exercises
Collaborators (uses)	



CSV Utility Subsystem

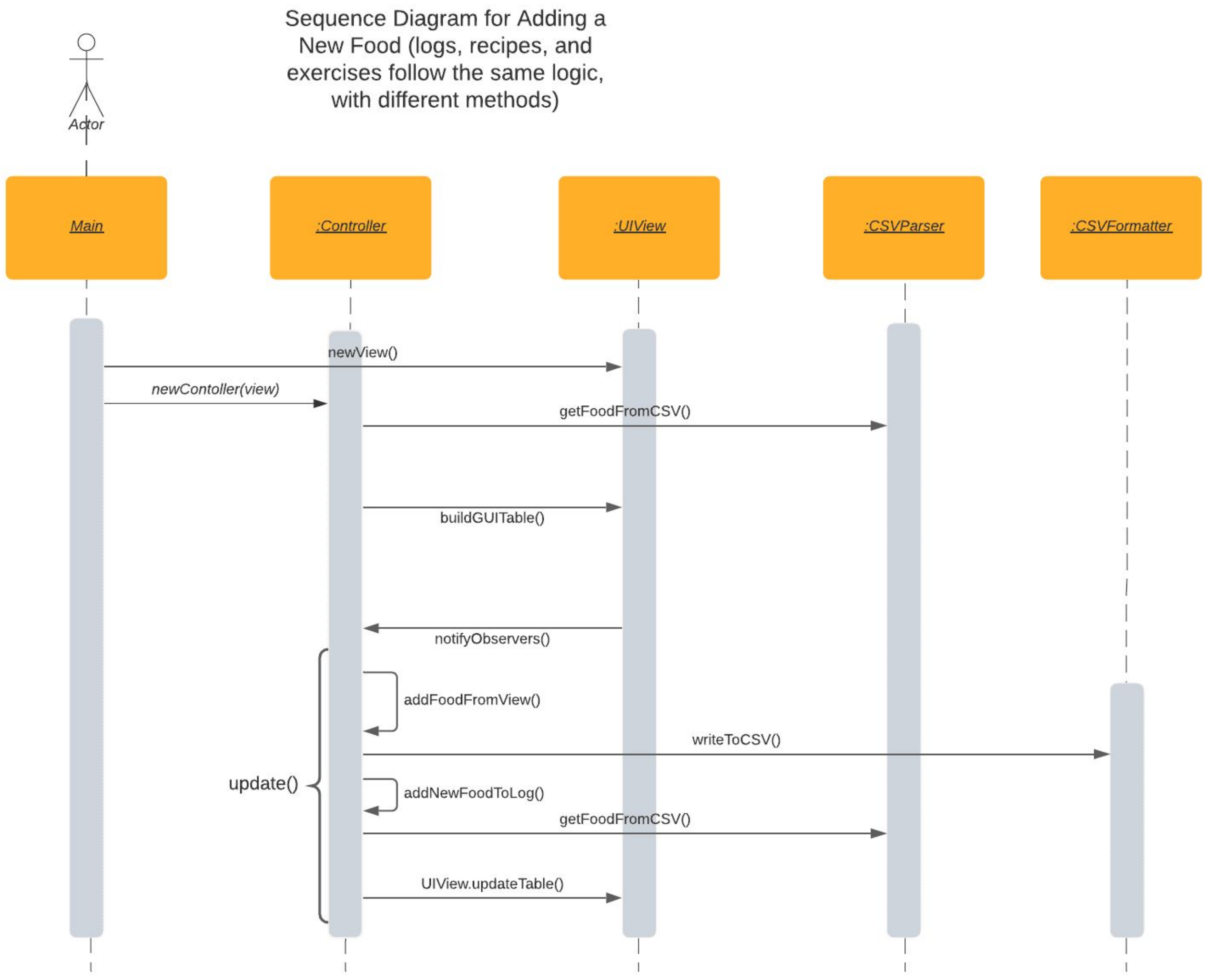
Class CSV_Formatter	
Responsibilities	Holds the method to format a string in order to be added to a .csv file
Collaborators (uses)	

Class CSV_Parser	
Responsibilities	Handles the logic for parsing a CSV file
Collaborators (uses)	

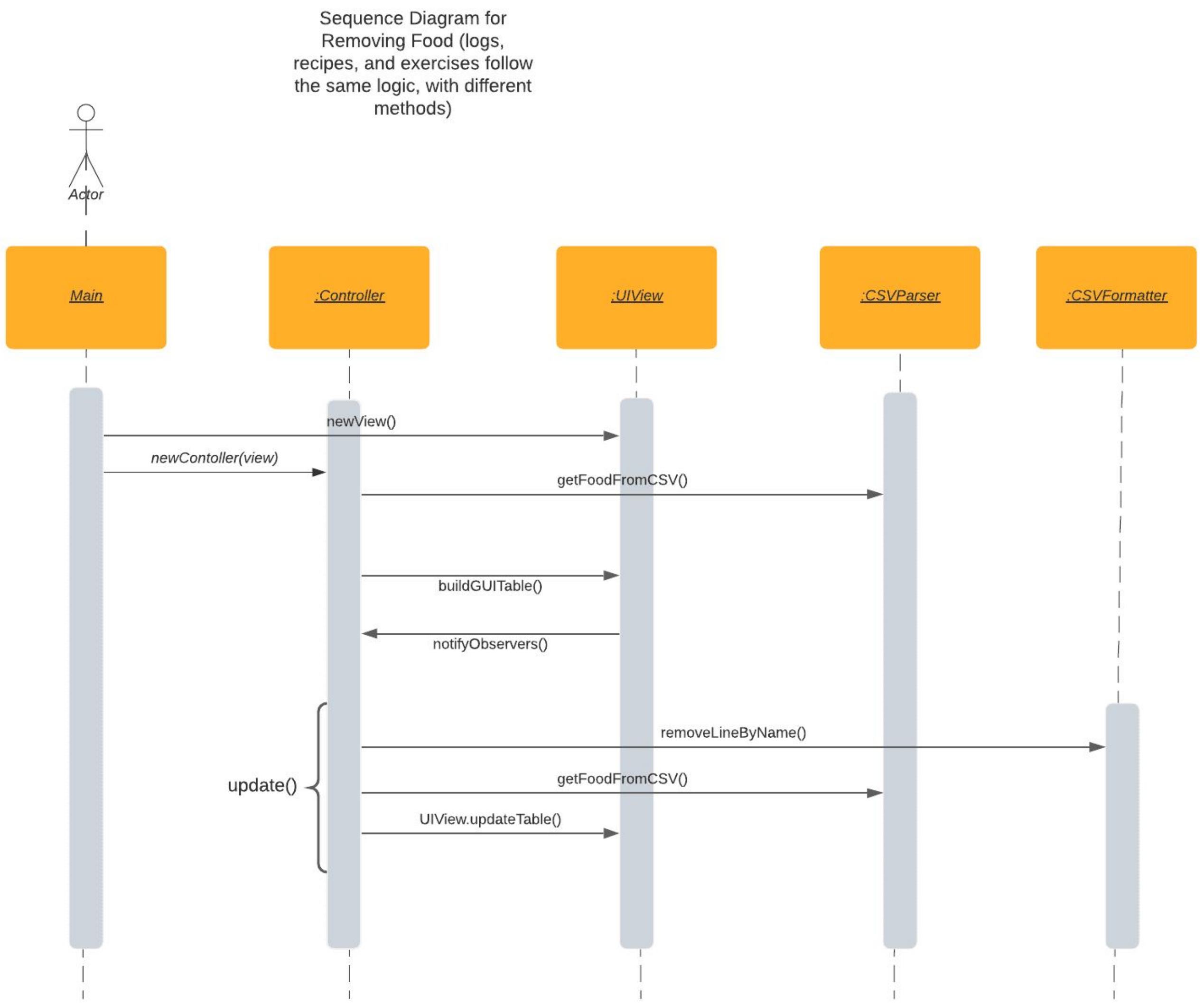


Sequence Diagrams

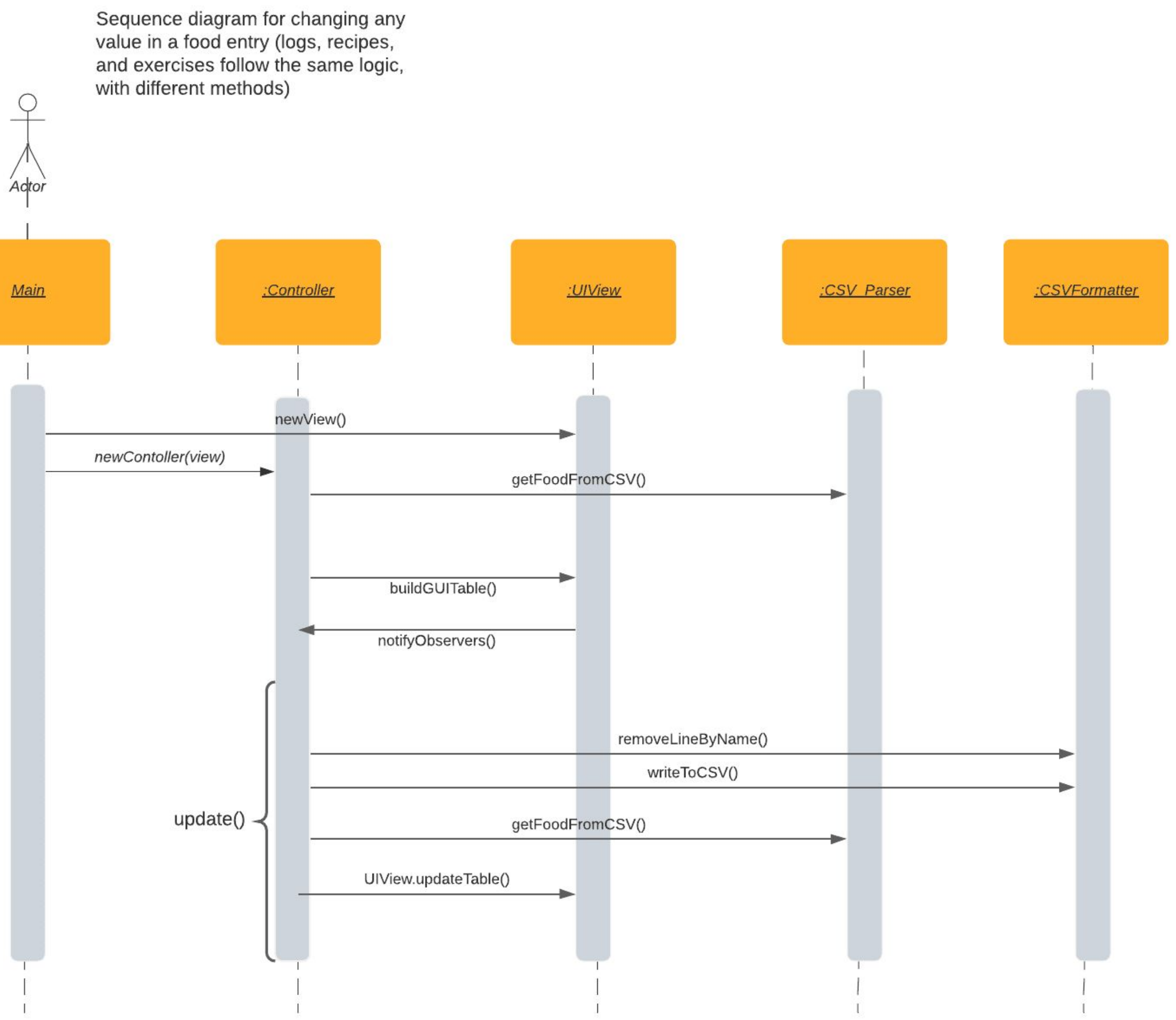
Sequence Diagram for Adding a New Food (logs, recipes, and exercises follow the same logic, with different methods)



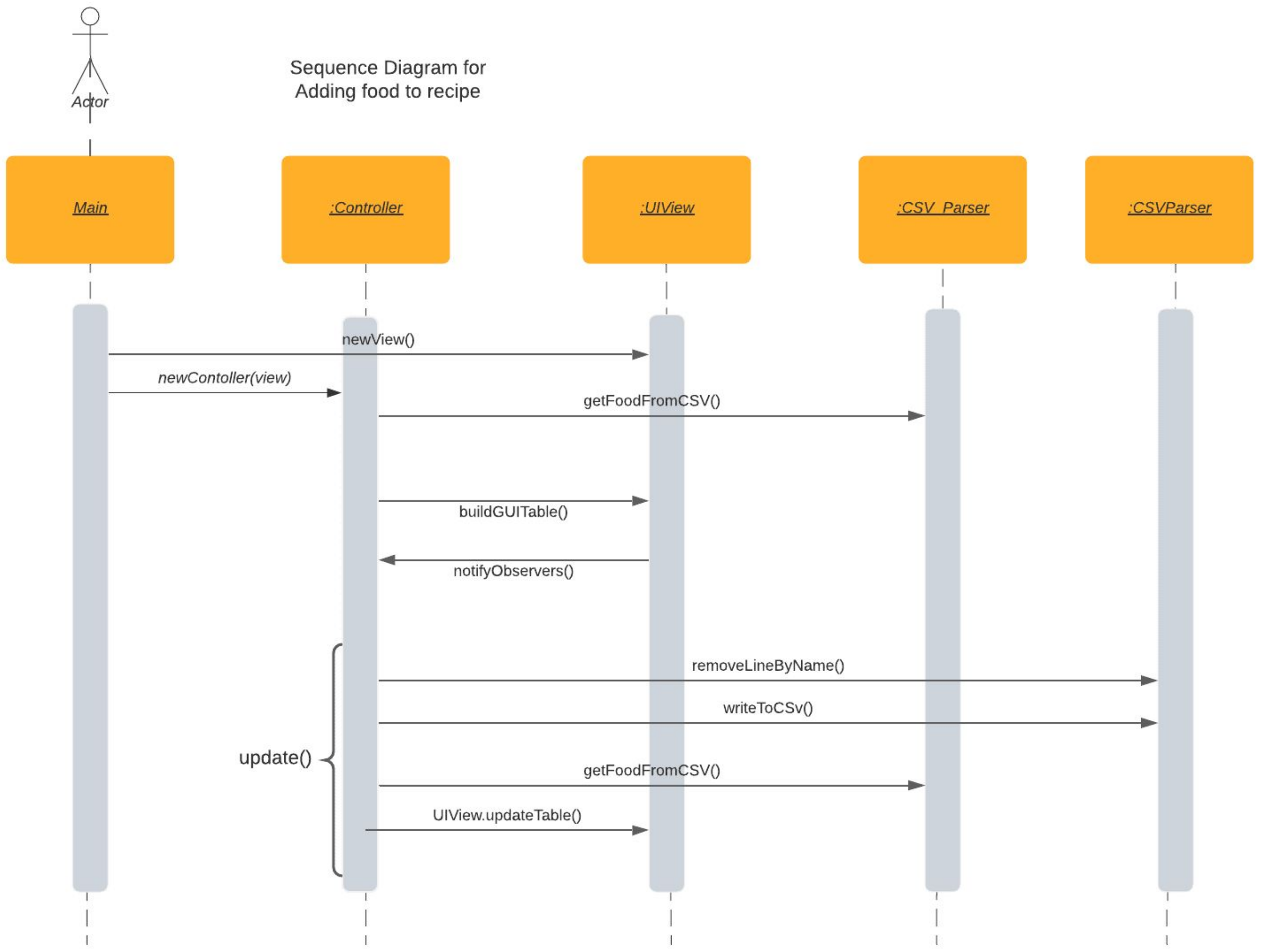
Sequence Diagram for Removing Food (logs, recipes, and exercises follow the same logic, with different methods)



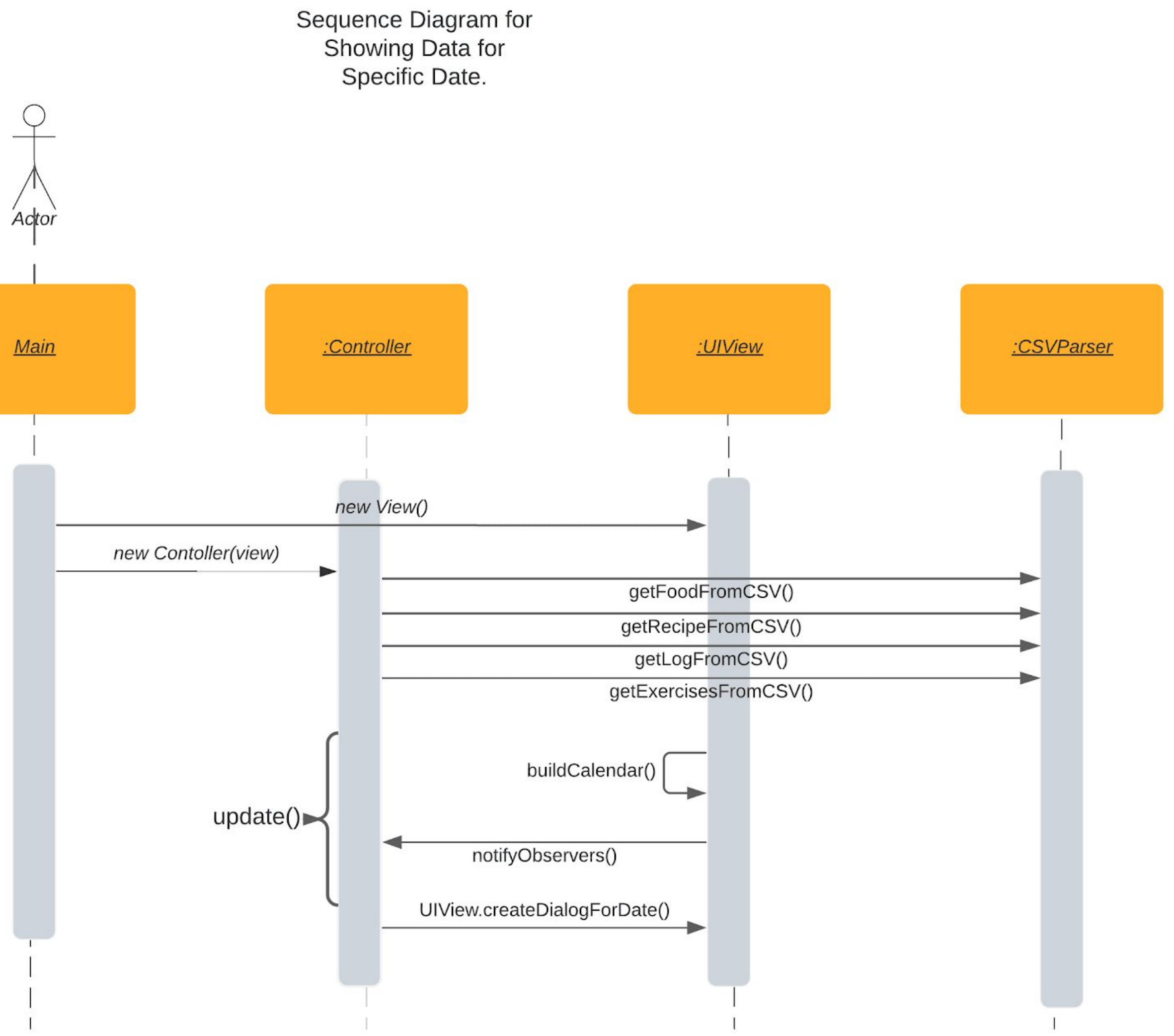
Sequence diagram for changing any value in a food entry (logs, recipes, and exercises follow the same logic, with different methods)



Sequence Diagram for Adding food to recipe



Sequence Diagram for Showing Data for Specific Date



Pattern Usage

Pattern #1 MVC Pattern

The application is organized using the model view controller pattern to recognize commands to the controllers, manipulate the Recipe, Food, Log, and UserData as the model, and reflect changes via the UIView.

MVC Pattern	
Model	Recipe Food Log UserData
View	UIView (will be separated

	into three different Views)
Controller	Controller

Pattern #2 Composite Pattern

The application is organized so that it selectively treats a group of objects (Recipe, Food) as “the same”, by using an interface to represent the group of objects..

Composite Pattern	
Component	*<<interface>> Component*
Composite	Recipe
Leaf	Food

Pattern #2 Observer Pattern

The application is organized so that the Controller is notified of state changes in the View.

Observer Pattern	
Subject	Observable (java.util)
Observer	*<<interface>> Observer*
Concrete Subject	UIView
Concrete Observer	Controller