HealthNCare

Project Design Document

Team Java Juggernauts

Christian Berko

Nathaniel Pellegrino

Tryder Kulbacki

Riley Basile-Benson

11/8/2023

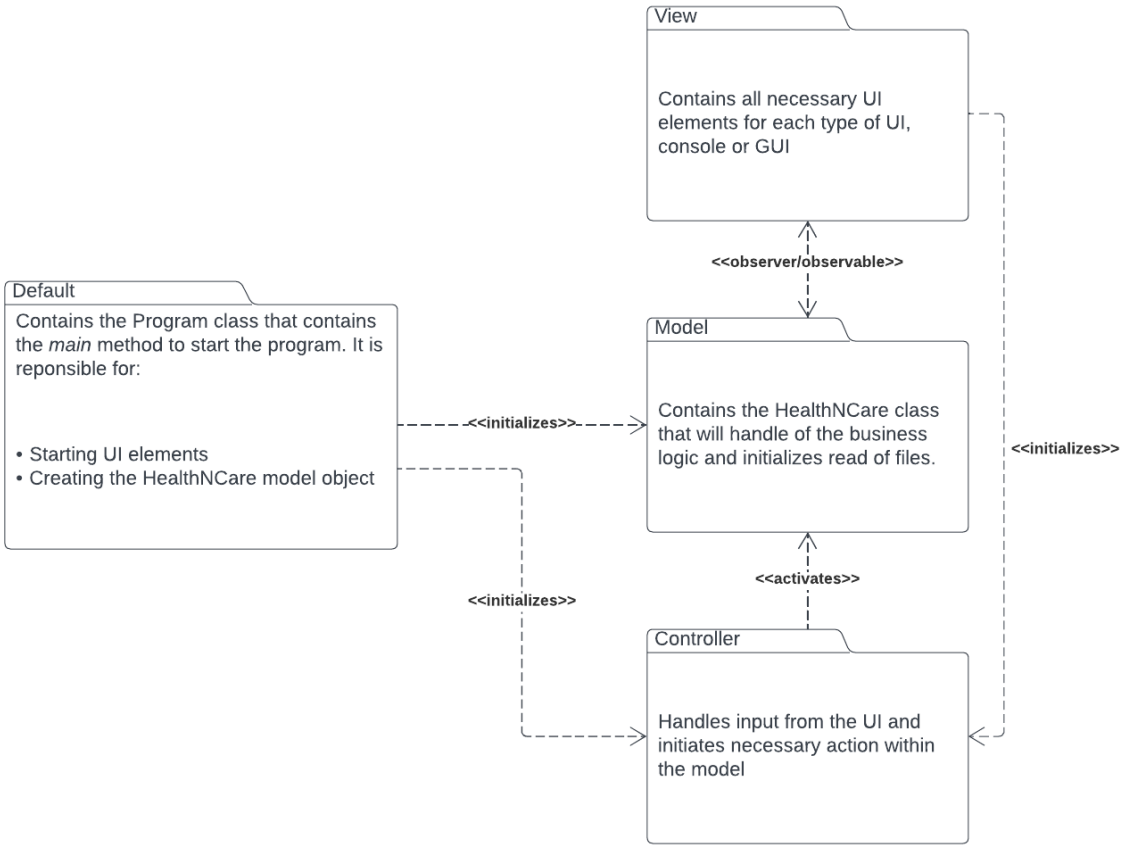
Project Summary

This project will allow users to track their eating habits and health. They can log foods and recipes they've eaten, and the app will calculate the total nutritional values they've consumed that day. The user can also add other information to their entries to help track their health progress, like weight and desired caloric intake.

Design Overview

The Program class is the main class and will contain the main function to start the program. From there it will interact with the File objects to log the weight data and the recipes and then retrieve that data. The File objects will contain a reference to the file on the filesystem and will have methods that read in the CSV and process data to return to HealthNCare. IFood is an interface that acts as the component within a composite pattern that will allow a recipe and a food object to be treated as the same thing. It will have operations such as getCalories or getSodium and will be able to return the values for either an entire recipe or a single food item. The Recipe class will implement the IFood interface and contain a collection of other IFood items. The Recipe class's implementation will have a recursive nature that will then call the getCalories for each item in the collection since they will all be IFood items. The FoodItem class will also implement IFood and will be used for any lowest-level object. It will have attributes such as calories, fat, carbohydrates, protein, and sodium. Our design is broken down into components so that it allows for the separation of concerns. The storage components will allow us to log the data but have the ability to change how the data is logged if necessary. Currently, our system would work well if it needs to be moved to a database since it is dependent on the StorageManager class. This might be something to consider if the program were to expand. We also used an interface for the view so the CLI is decoupled from the rest of the code through the use of listener objects that handle interaction with the model. This allows us to easily swap in a GUI if needed.

Subsystem Structure

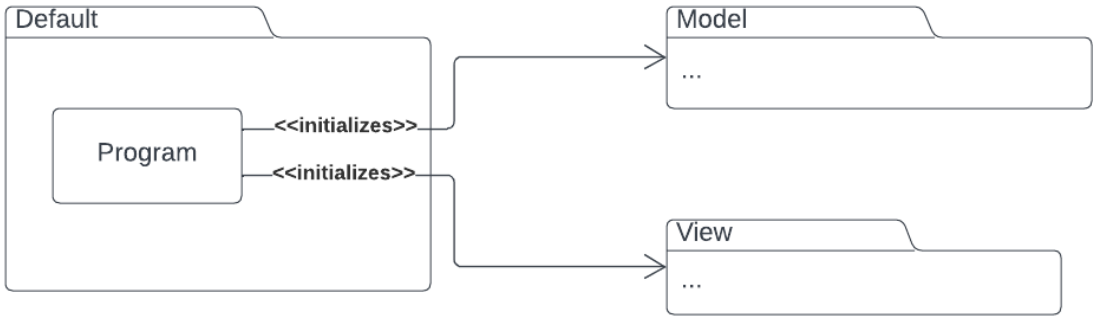


Subsystems

Default Subsystem

Class Program	
Responsibilities	Create and manage the application's core objects and data models. Create the TextUI and, eventually GUI. (User Interface) Display the user interface(s) to enable interaction with the application Handle inputs and execute appropriate application logic.

Collaborators (uses)	**model.HealthNCare** - starts main business logic class **view.UserInterface** - starts one or more user interfaces



Model Subsystem

Class HealthNCare	
Responsibilities	● Notify observers of any changes ● Handle reading and writing to main data structures ● Uses business logic for interacting with the data
Collaborators (inherits)	**java.util.Observable** - so the program changes can be observed by others
Collaborators (uses)	**model.StorageManager** - interacts with the storage media and allows HealthNCare to write out or read the log and foods **java.util.Observer** - for notifications of log inputs to views. **java.util.HashMap** - stores the available foods **java.util.ArrayList** - stores the LogEntries

Class StorageManager	
Responsibilities	Interact with the storage media to read and write the data
Collaborators (uses)	**model.IFood** - Gets changes from the recipes and food selected. **model.ILog** - handles the log data **model.IRecipeLog** - handles the food data **java.util.HashMap** - stores the available foods **java.util.ArrayList** - stores the LogEntries

Class LogCSVFile	
Responsibilities	Writes the current log back to the CSV file. It will completely overwrite the existing file and replaces it with what the given object
Collaborators (inherits)	**ILog** interface for all log storage media
Collaborators (uses)	**org.apache.commons.csv.CSVReader** - reads in and parses a CSV file **org.apache.commons.csv.CSVReader** - writes out a CSV file **model.LogEntry** - contains information about each log entry

Class RecipeCSVFile	
Responsibilities	Concrete implementation of link IRecipeLog for using a CSV file
Collaborators (uses)	**org.apache.commons.csv.CSVReader** - reads in and parses a CSV file **org.apache.commons.csv.CSVReader** - writes out a CSV file **model.IFood** - Gets list of available

	recipes, food, and their corresponding nutrition values. **model.FoodItem** - holds nutritional information about a food **model.Recipe** - holds information about a recipe

Interface IFood	
Responsibilities	Common interface for all food items. Whether the item is a basic food item or a recipe it should be able to get the nutritional information from it.

Class Recipe	
Responsibilities	A collection of food and other recipes.
Collaborators (inherits)	**model.IFood** - component used to structure the collection
Collaborators (uses)	**java.util.ArrayList -** Arraylist to hold the food

Class FoodItem	
Responsibilities	An individual item of food that can be added to a recipe.
Collaborators (inherits)	**model.IFood** - component used to structure the object

Class CalorieLimitEntry	
Responsibilities	A concrete implementation of a LogEntry that tracks a person's calorie limit at a particular point in time
Collaborators (inherits)	LogEntry
Collaborators (uses)	**java.time.LocalDate -** Used to create a date Object

Class ConsumptionEntry	
Responsibilities	Concrete implementation of link LogEntry that logs a food consumed and a specific quantity
Collaborators (inherits)	LogEntry
Collaborators (uses)	**java.time.LocalDate -** date of the consumption entry

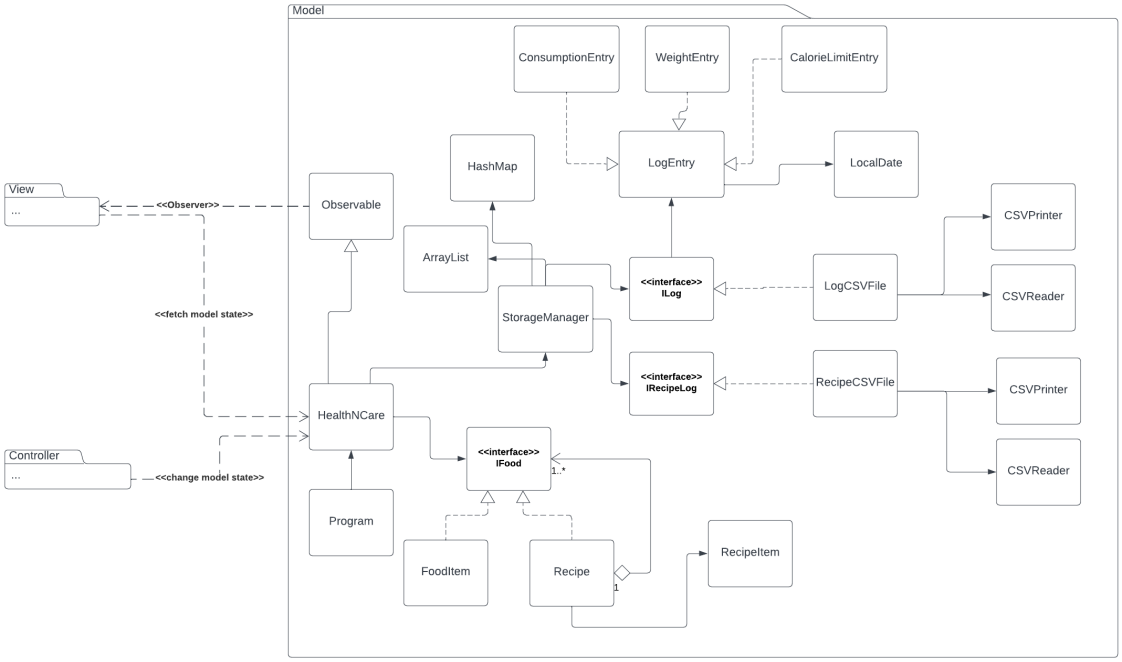
Class ILog	
Responsibilities	Common interface for any log storage medium. It should be able to read all the log entries and write them back out
Collaborators (uses)	**java.util.ArrayList -** stores a list of Log Entry

Class IRecipeLog	
Responsibilities	Interface definition for any Recipe storage object. It should be able to read all the items and write the items back out to the storage medium
Collaborators (inherits)	**java.util.HashMap -** holds recipes

Class StorageManager	
Responsibilities	Manages the storage of the log and recipes
Collaborators (uses)	**model.ILog** - Handles log entries **model.IRecipeLog** - Handles food entries **java.util.ArrayList** - holds log entries **java.util.HashMap** - holds food entries

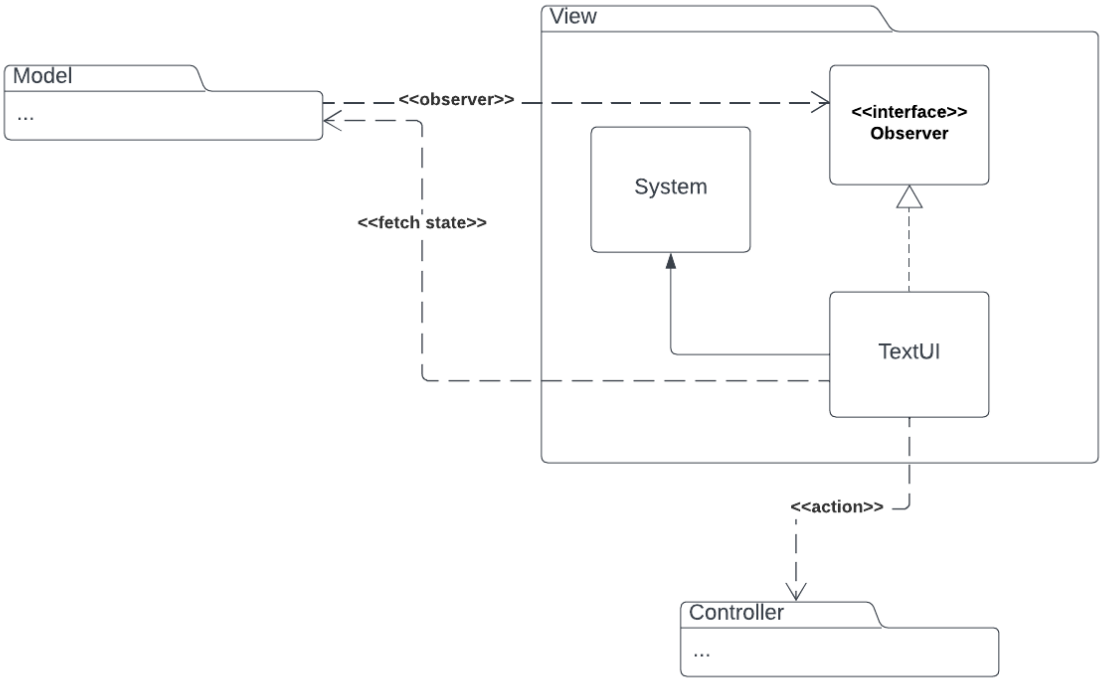
Class WeightEntry	
Responsibilities	Manages the storage of the log and recipes
Collaborators (uses)	**LogEntry** - interface for all log entries
Collaborators (inherits)	**java.time.LocalDate -** date of the consumption entry

Interface LogEntry	
Responsibilities	Common interface for each type of log entry
Collaborators (uses)	**java.time.LocalDate -** date of the consumption entry



View Subsystem

Class TextUI	
Responsibilities	Handles the command line interface which takes in the user input and displays the data
Collaborators(inherits)	**java.util.Observable -** so that changes in the UI can be observed by other classes **java.util.Scanner** - accepts input from the terminal **java.lang.System** - prints output to a terminal window



Controller Subsystem

Class AddFoodListener	
Responsibilities	Record the food (name, cal, fat, carbs) to be added to a recipe. On change will add the food to a recipe.
Collaborators(uses)	**model.HealthNCare**

Class ConsumeFoodListener	
Responsibilities	Records the food to be consumed. When consumed will remove the food that was consumed.
Collaborators (uses)	**model.HealthNCare**

Class AddCalorieLimitListener	
Responsibilities	Responsible for adding new CalorieLimit
Collaborators (uses)	**model.HealthNCare**

Class AddRecipeListener	
Responsibilities	Responsible for responding to requests for daily statistics
Collaborators (uses)	**model.HealthNCare**

Class AddWeightEntry	
Responsibilities	Responsible for handling the addition of a weight entry.
Collaborators (uses)	**model.HealthNCare**

Class CheckFoodListener	
Responsibilities	Responsible for checking if a food exists in the database.
Collaborators (uses)	**model.HealthNCare**

Class ConsumeFoodListener	
Responsibilities	Responsible for handling the consumption of a food item.
Collaborators (uses)	**model.HealthNCare**

Class DailyStatsResponder	
Responsibilities	Responds to requests for daily statistics
Collaborators (uses)	**model.HealthNCare**

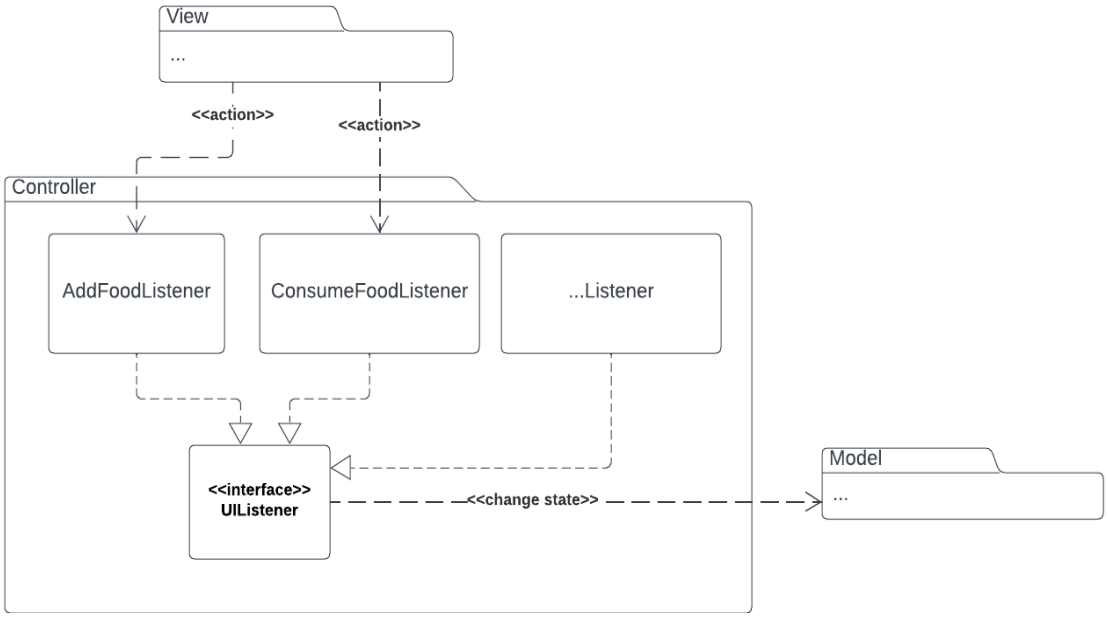
Class DeleteFoodEntryListener	
Responsibilities	responsible for deleting a food entry from the log.
Collaborators (uses)	**model.HealthNCare**

Class DeleteFoodListener	
Responsibilities	Responsible for handling the deletion of a food item from the database.
Collaborators (uses)	**model.HealthNCare**

Class FoodListResponder	
Responsibilities	Responsible for responding to requests for the list of foods.

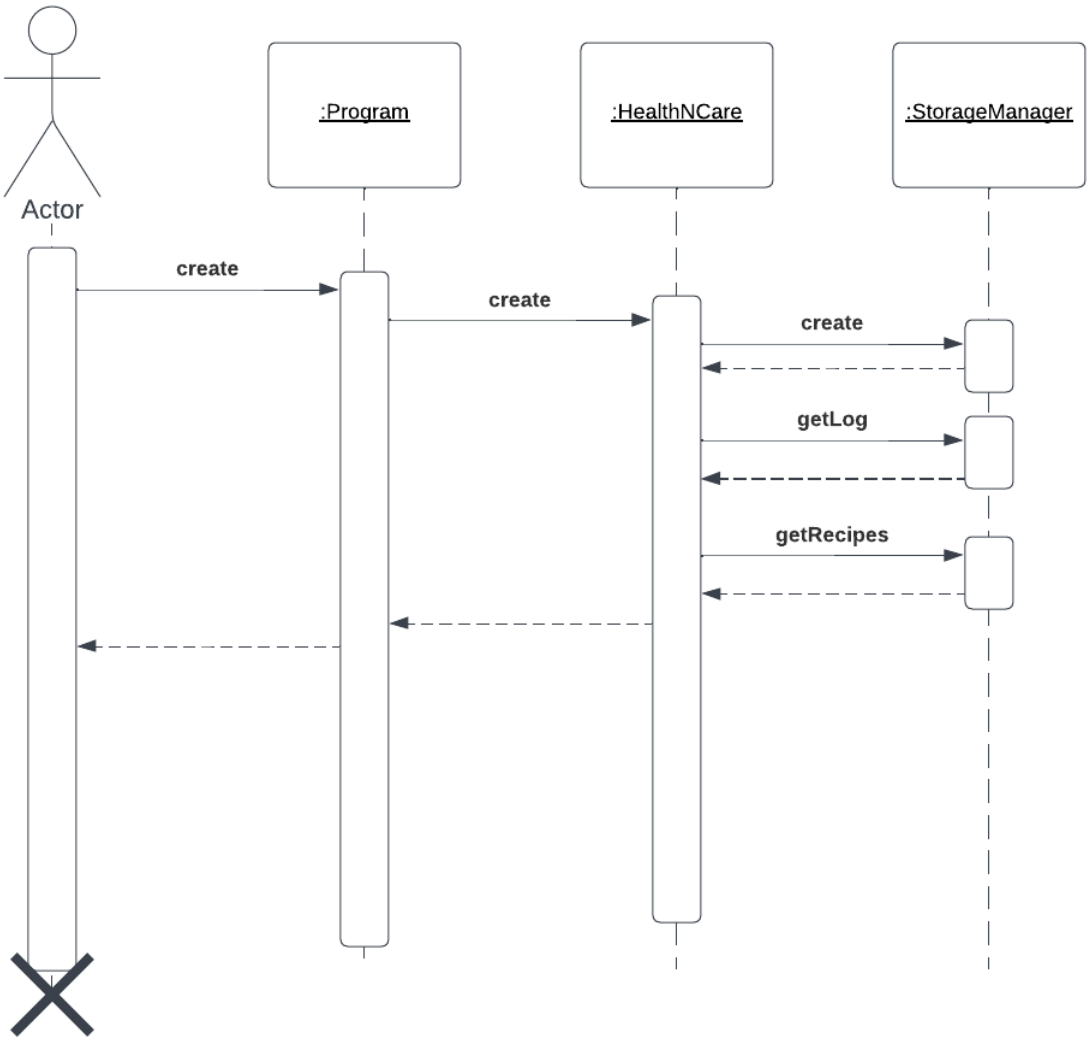
Collaborators (uses)	**model.HealthNCare**

Class SaveDataListener	
Responsibilities	Responsible for saving the data in the program.
Collaborators (uses)	**model.HealthNCare**

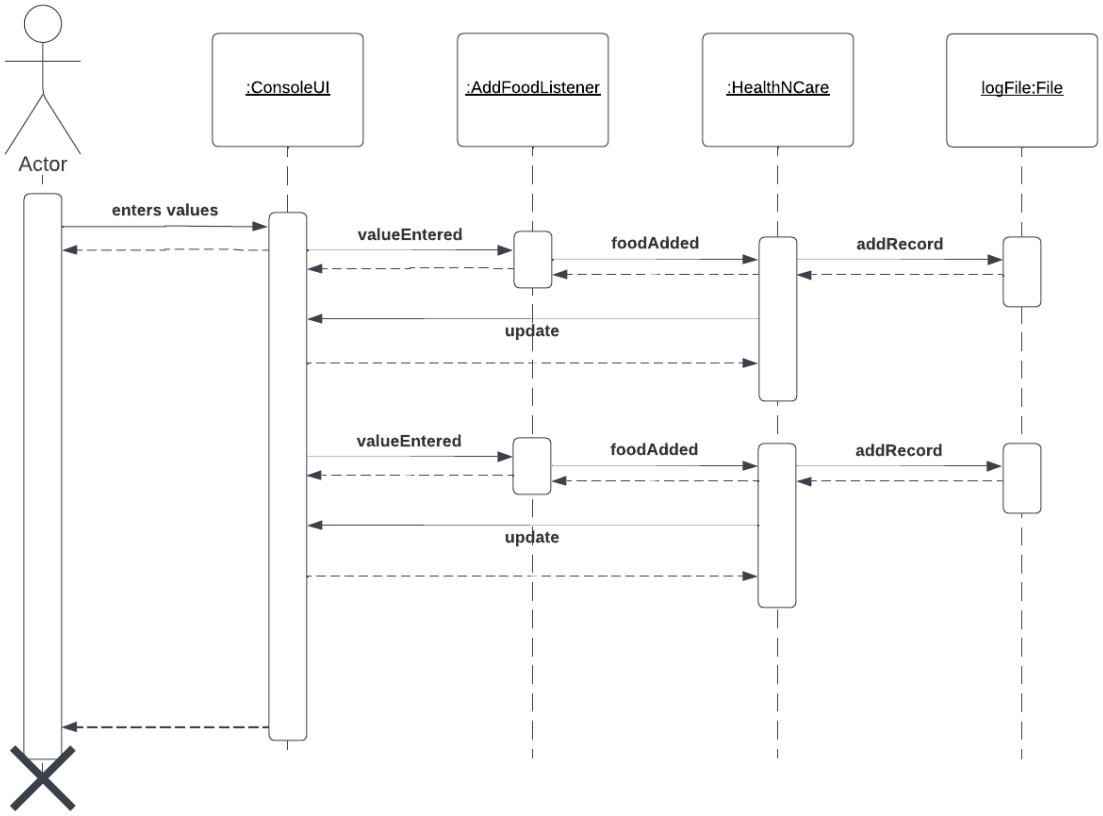


Sequence Diagrams

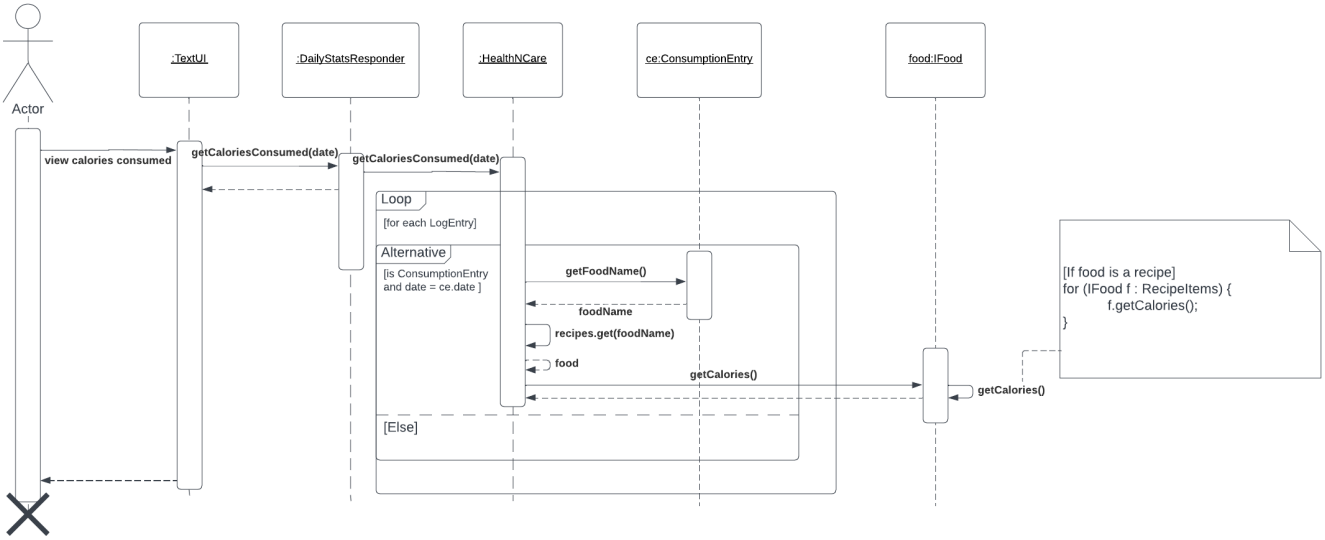
Sequence 1 - Reading in food data



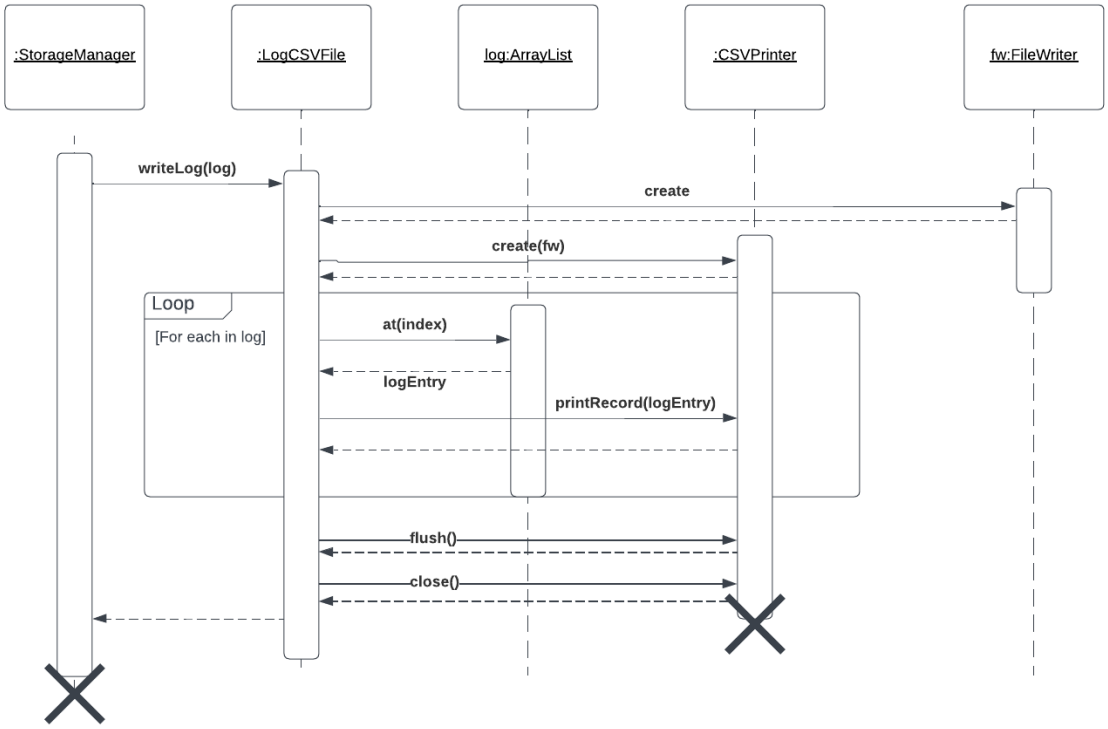
Sequence 2 - Add servings of food to the log and saving



Sequence 3 - Compute the total calories for today



Sequence 4 - Writing the log to the CSV file



Pattern Usage

Pattern 1 - Composite

The composite pattern allows the application to handle a food item and a recipe as if they are the same. This allows recipes to have sub-recipes recursively.

Composite Pattern	
Leaf	FoodItem
Component	IFood
Composite	Recipe

Pattern 2 - Observer

The observer pattern is used to allow the model to update the view elements whenever its state changes.

Observer Pattern	
Observer	TextUI
Observable	HealthNCare

Pattern 3 - MVC

This application will use the MVC pattern so that commands are passed to the controller which then send it to the model to adjust its state.

MVC Pattern	
Model	HealthNCare
Views	ConsoleUI
Controllers	AddFoodListener ConsumeFoodListener

Rationale

The application is built using the MVC pattern as an architectural design. This separates the UI elements from the business logic and will make it easier to swap UI types in the future without needing to change anything in the model. The observer pattern helps us to decouple the UI elements from the model while allowing for the UI to automatically be updated whenever the state changes. The composite pattern allows us to create food items that satisfy the need of being able to contain many sub recipes and still having the calculations not be very complex. Each section of the application has a distinct purpose which allows for good separation of concerns. For example, the storage techniques are completely decoupled from the actual implementation of the storage medium. Whether the data is stored in CSV, or JSON, or a database, as long as it satisfies the requirements set in the appropriate interface, it will work.

User Documentation

Basic Startup:

1. Upon startup of the application you will be prompted to enter a date. You can leave these fields blank to use the current date
2. You can then enter a number corresponding to the function you would like to perform as shown on the menu options.
3. To exit the application, enter 0 and the Enter key. This will close the application and save any changes that you made.
4. If you want to save but not exit, you can select option number 7 to ensure that any changes will not be lost if the application unexpectedly closes.

Setting calorie limits and tracking weight:

1. To set your ideal calorie limit select option 1 and enter the number of calories that you would like to eat in a day and press Enter.
2. You will then enter your weight in pounds on this day
3. Both of these changes will now be reflected in any further interaction with the app such as showing the daily stats.

Adding a new food:

1. Select option 2 to create a new basic food item to use in recipes and logs
2. Follow each prompt and enter the values requested such as the name and the nutritional information

Logging a food you've eaten

1. When you consume an item you will want to log that in the app and you can easily do that
2. Select option 5 from the main menu and type the name of the food to be logged and then the amount of servings you consumed

Seeing your daily stats

1. You can see where you stand as far as calorie consumption by looking at your stats for the day
2. Select option 4 from the main menu, it will tell you whether you are above or below the calorie limit and will show you the breakdown of nutrients consumed
3. At the top you will see a list of foods eaten as well as how many servings were consumed

Adding a new recipe:

1. A recipe in this application is defined as a food that can be consumed and is composed of other food items which may or may not include other recipes.
2. To create a recipe select option 3 in the main menu and give the recipe a name
3. The system will now prompt for the number of ingredients required, after you enter the number it will prompt for each food item required and how many are needed. (Note: a

recipe’s sub items need to exist in the food list before they can be used in a recipe) After you complete filling out the recipe information you can then log it like any normal food and the calculations will be done automatically from the base food that comprise the recipe.