HealthNCare / RELEASE 0

Project Design Document

TEAM 1A Team Dakota

Daniel McKee | dgm6546@rit.edu

Luke Picciano | lmp3797@rit.edu

Kyle Weishaar | kmw2153@rit.edu

Sahand Nowshiravani | sn2198@rit.edu

Leeon Noun | lsn3369@rit.edu

Google Doc link to document with version history:

https://docs.google.com/document/d/1XAAhTr80pEceyChAUEymAxwZlg3ph3z-KA5jYVNuyp8/edit?usp=sharing

1 Project Summary

The HealthNCare App is a program meant to encourage users to monitor and improve their daily lives with efficient and easy-to-use interactions to provide the best user experience.

Users are provided with a basic list of foods and their nutrition for users to pick from, with the opportunity to create their own collections of foods and recipes that they may consume in order to track statistics like carbohydrates, sodium, calories, fat, and protein. They are also able to use recipes as ingredients for other recipes.

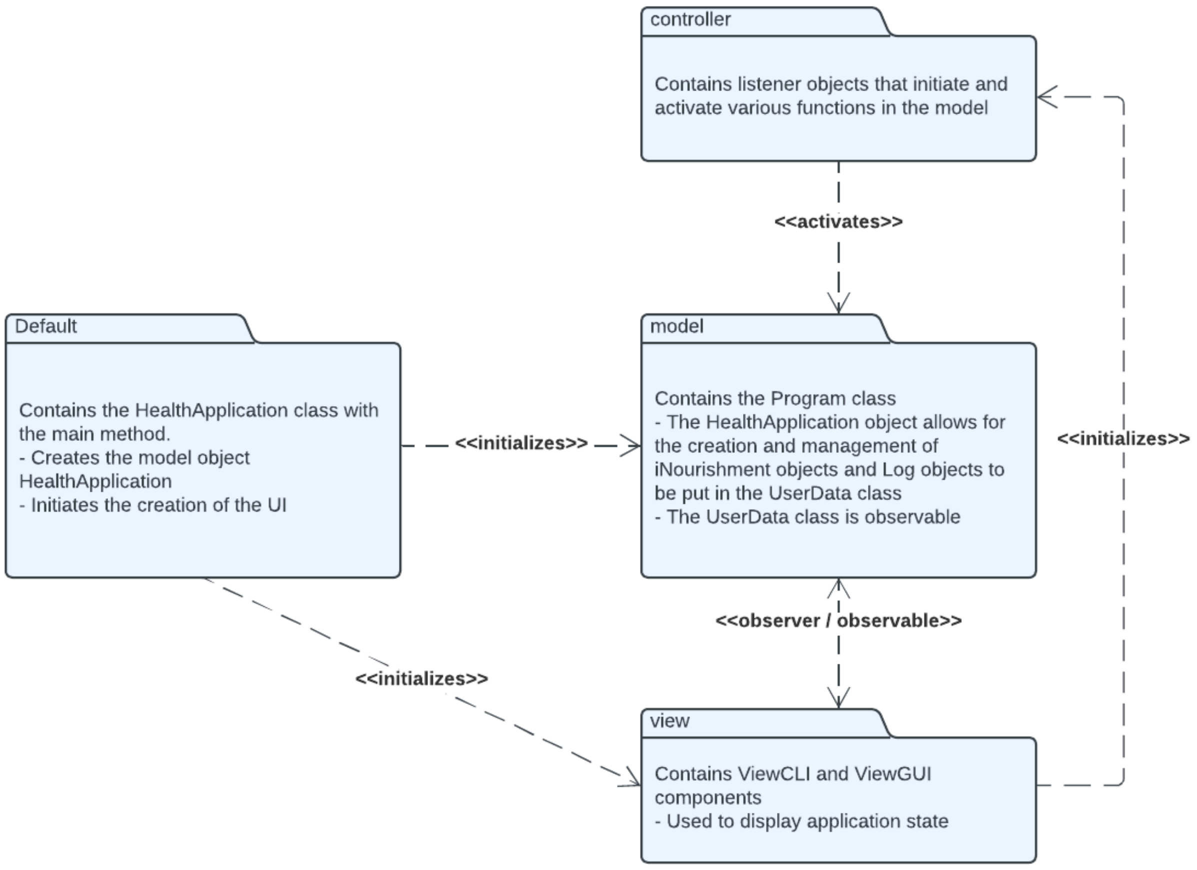
Each day they are able to create a log of the foods they've eaten and view statistics from the data. Users have the ability to see/set fields like daily consumption, calorie targets, and recorded weights. User data will be saved.

2 Design Overview

The HealthNCare App has separation of concern and modularity at the forefront of the build using the Model-View-Controller (MVC) pattern. The Model handles the foundation data (food items, recipes, exercise, daily logs, etc). The View shows this information to the user, and the Controller takes care of input and updates – coordinating model and view interaction. This layout allows for high coherence within each subsystem and low coupling between subsystems so that when one element of the system changes, other elements don't have to deal with it as much.

The structure is dependent inversive so the structure of the system storage and access via interfaces can be tuned to evolve for future enhancements. The platform is designed for extensibility – so add-ons such as more nutrition tracking or user goals can be added without significant change to the design.

3 Subsystem Structure


controller
Contains listener objects that initiate and activate various functions in the model
<<activates>>
Default
Contains the HealthApplication class with the main method.
- Creates the model object HealthApplication
- Initiates the creation of the UI
<<initializes>>
model
Contains the Program class
- The HealthApplication object allows for the creation and management of iNourishment objects and Log objects to be put in the UserData class
- The UserData class is observable
<<initializes>>
<<observer / observable>>
<<initializes>>
view
Contains ViewCLI and ViewGUI components
- Used to display application state

4 Subsystems

4.1 Subsystem Default

Class HealthApplication	
Responsibilities	Creates the Program object Selects the UI option Contains main method
Collaborators (uses)	**model.Program** - Primary model class. **view.ViewCLI** - Command line interface

4.2 Subsystem Model

Class iNourishment (interface)	
Responsibilities	Provides a generic interface for all food/recipe objects Includes the food name and their statistics Can retrieve if they are a food or a recipe. If they are a recipe, can retrieve the collection of foods that make up the recipe

Class Recipe	
Responsibilities	Composite class for the Composite pattern. Holds a name and other Food leaf objects that make up the recipe
Collaborators (inheritance)	**model.iNourishment** - Interface that defines nourishment objects like Food and Recipe
Collaborators (uses)	**model.iNourishment** - Each ingredient in a recipe is a iNourishment object

Class Food	
Responsibilities	Leaf class for the Composite pattern. Contains data like calories, protein, fat, etc.
Collaborators (inheritance)	**model.iNourishment** - Interface that defines nourishment objects like Food and Recipe

4.3

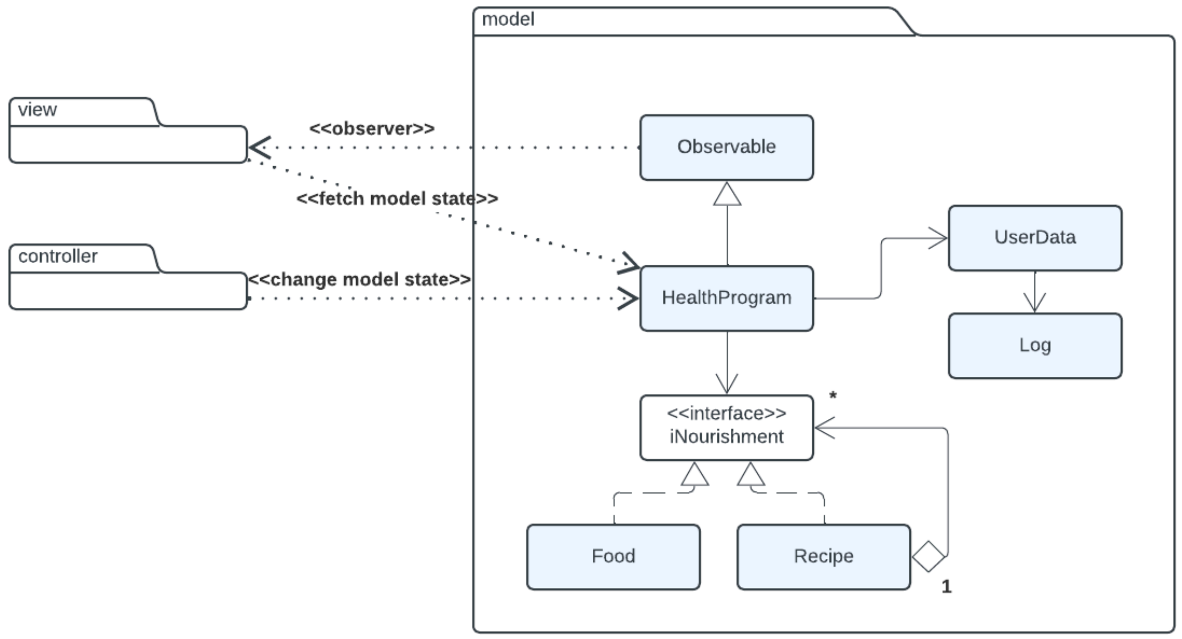
Class iExercise (interface)	
Responsibilities	Provides a generic interface for all exercise objects Includes the exercise unique name and hours of burned calories

Class Exercise	
Responsibilities	Leaf class for the Composite pattern. Contains data like the name of exercise and calories burned per hour for a 100-pound person
Collaborators (inheritance)	**model.iexercise** - Interface that manages exercise database like types of Workout and calories burned per hour

Class HealthProgram	
Responsibilities	Notifies observers of changes Creates new nourishment Create new unique exercise Accesses UserData
Collaborators (inheritance)	**Java.util.Observable** - So program changes can be reflected through the view
Collaborators (uses)	**Java.util.Observer -** For notifications of data change for view.

Class UserData	
Responsibilities	Container class for user data Handles modification of user data Handles creation of Log objects / log collection Handles nourishment collection Handles file IO Handles exercise unique name Handles exercise log
Collaborators (uses)	**model.Log -** Creates different log entries for the user to be saved

Class Log	
Responsibilities	Object to hold daily log information that the user inputs.

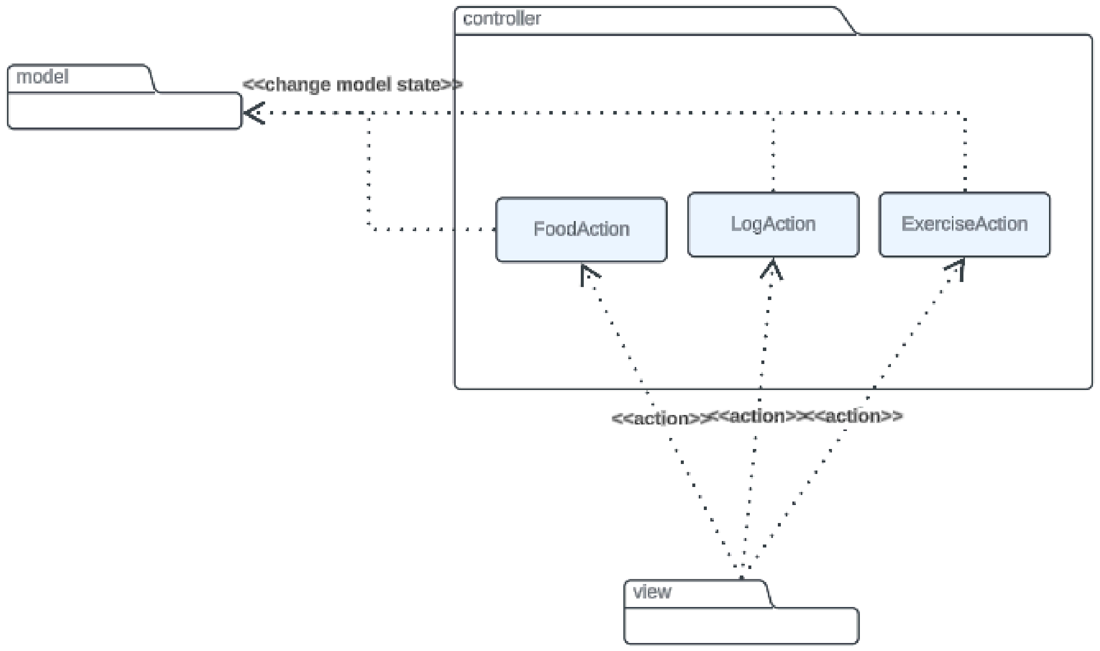


4.4 Subsystem Controller

Class FoodAction	
Responsibilities	Used for conveying user input to the **HealthProgram** to handle iNourishment objects
Collaborators (uses)	**model.HealthProgram** - High level model class that contains all nourishment creation and handling functions

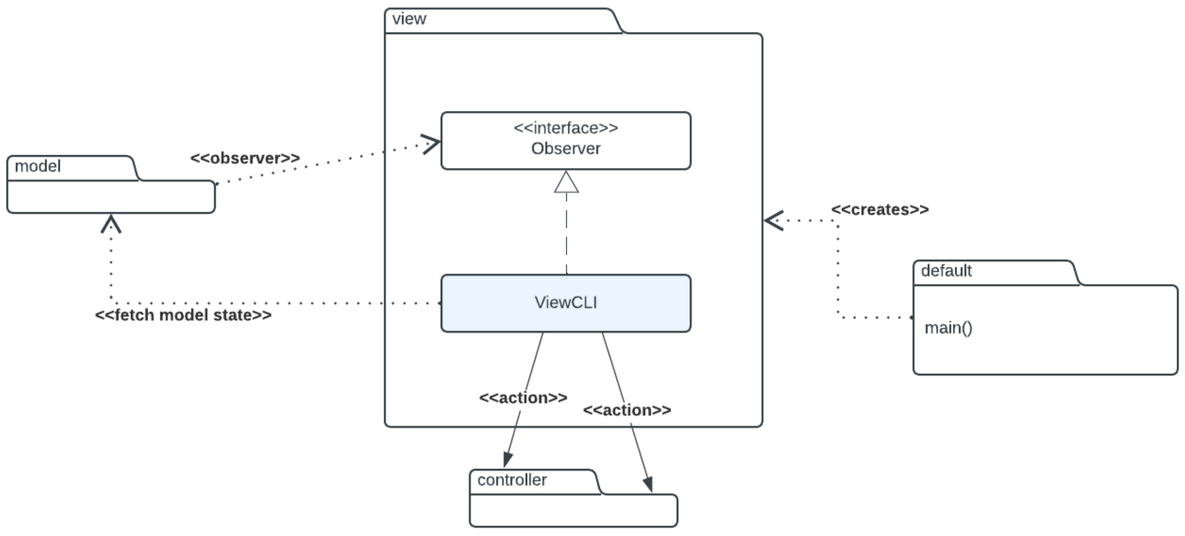
Class LogAction	
Responsibilities	Used for conveying user input to the **HealthProgram** class to handle Log objects
Collaborators (uses)	**model.HealthProgram** - High level model class that contains all log creation and handling functions

Class ExerciseAction	
Responsibilities	Used for conveying user input to the **HealthProgram** class to handle Exercise objects
Collaborators (uses)	**model.HealthProgram** - High-level model class that contains all exercise creation and handling functions, ensures all log entires reflect the data accurately



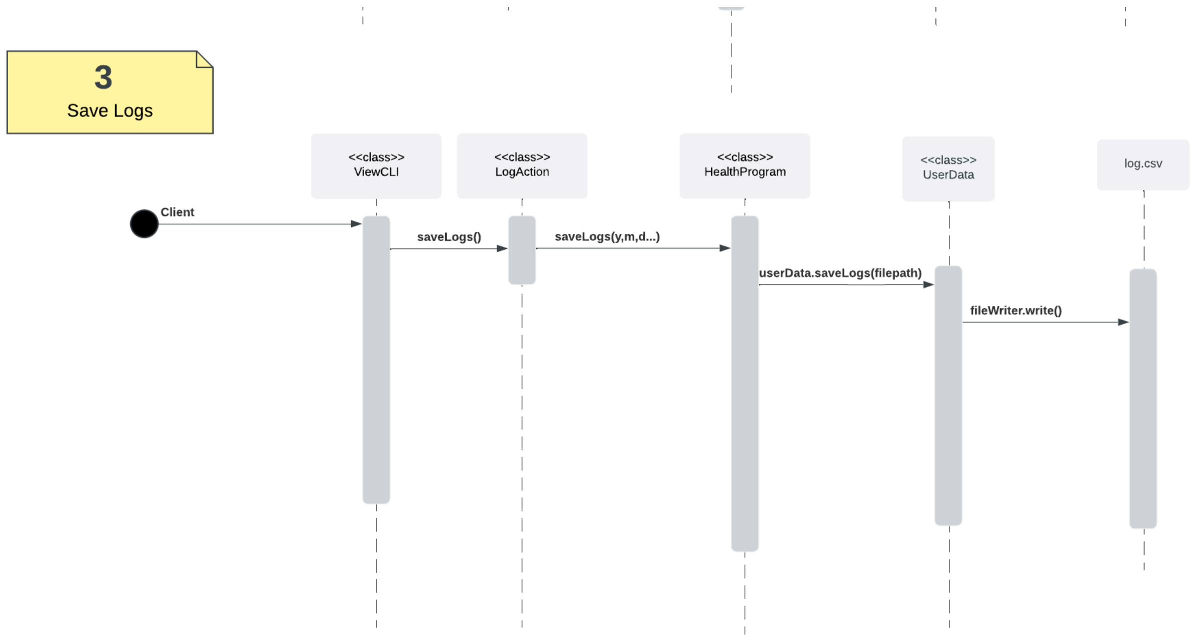
4.5 Subsystem View

Class ViewCLI	
Responsibilities	Section for where the user can create/modify iNourishment and exercise objects
Collaborators (inherits)	**view.UserInterface** - Abstract for UI **java.util.Observer** - Used to observe HealthProgram
Collaborators (uses)	**controller.FoodAction** - Controller class to pass through Nourishment commands **controllerLogAction** - Controller class to pass through Log commands **controller.ExerciseAction** - Controller class to pass through exercise commands

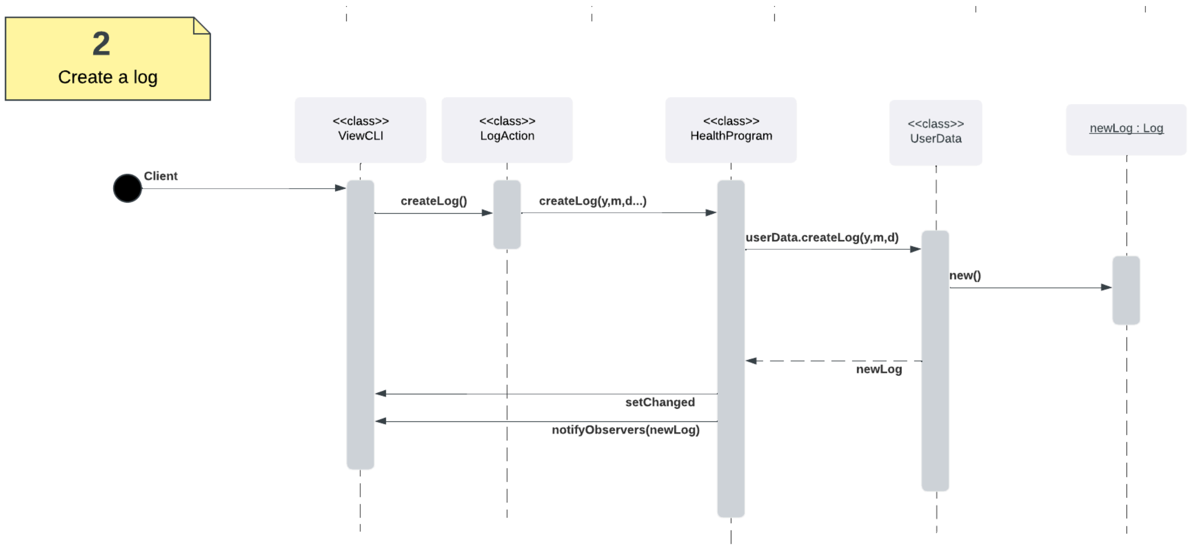

view
<<interface>>
Observer
ViewCLI
model
<<observer>>
<<fetch model state>>
<<creates>>
default
main()
<<action>>
<<action>>
controller

5 Sequence Diagrams

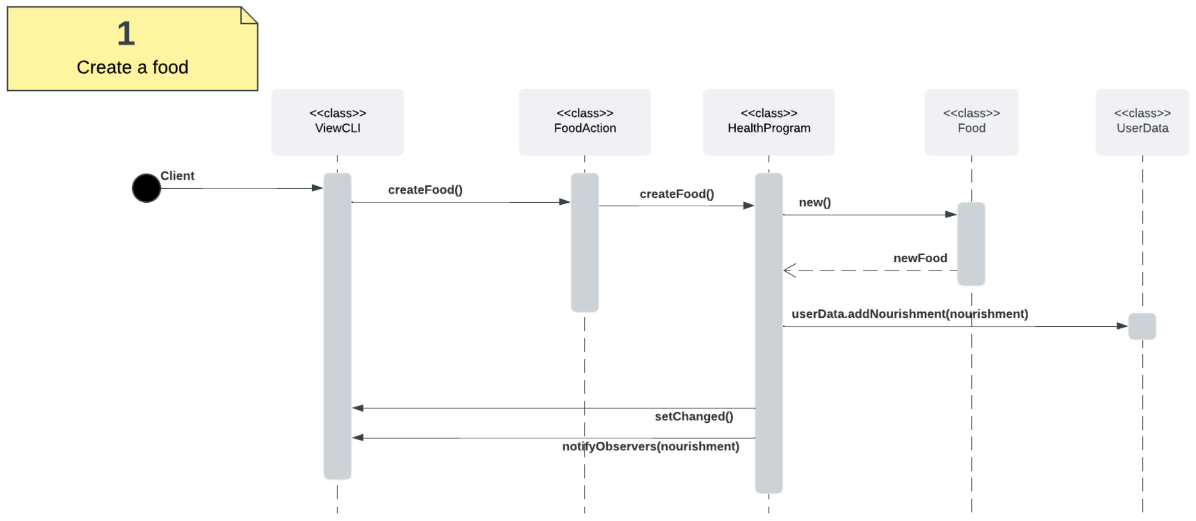
5.1 Save a log



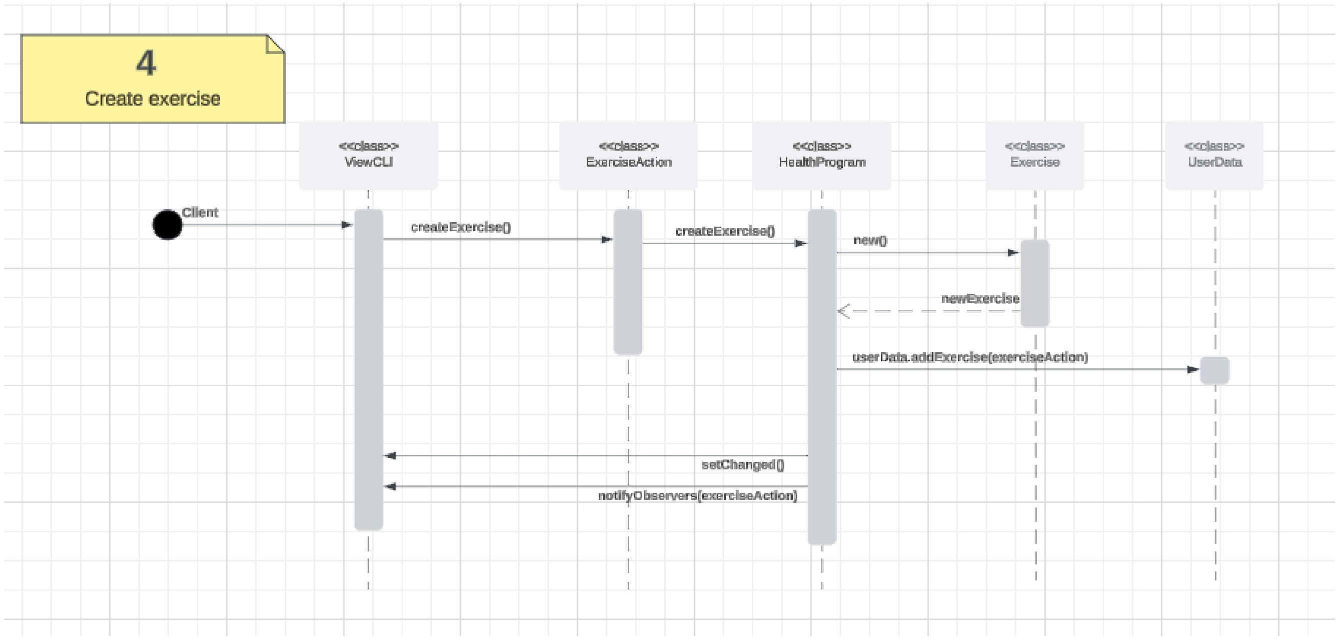
5.2 Create a log



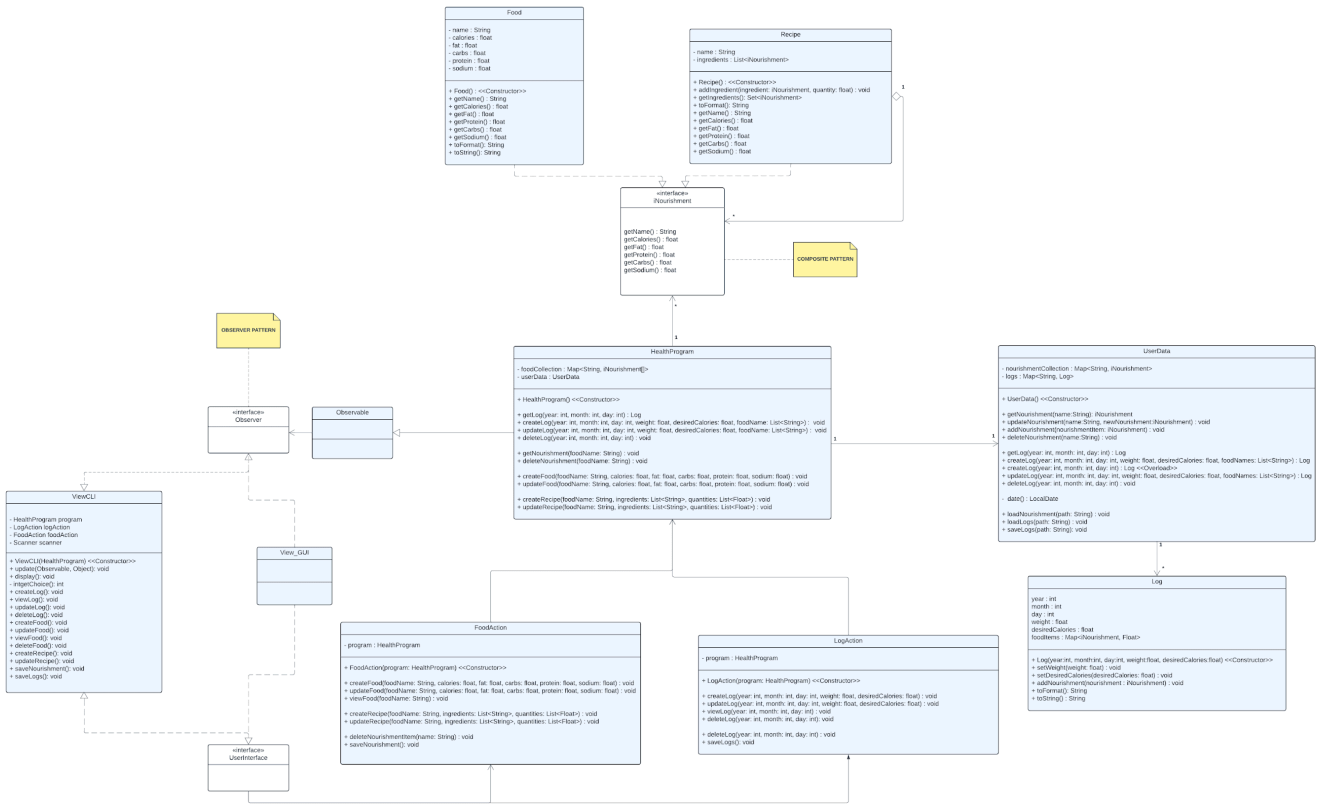
5.3 Create a food



5.4 Create exercise



CLASS DIAGRAM


Food
- name : String
- calories : float
- fat : float
- carbs : float
- protein : float
- sodium : float
+ Food() : <<Constructor>>
+ getName() : String
+ getCalories() : float
+ getFat() : float
+ getProtein() : float
+ getCarbs() : float
+ getSodium() : float
+ toFormat(): String
+ toString(): String
Recipe
- name : String
- ingredients : List<iNourishment>
+ Recipe() : <<Constructor>>
+ addIngredient(ingredient: iNourishment, quantity: float) : void
+ getIngredients(): Set<iNourishment>
+ toFormat(): String
+ getName() : String
+ getCalories() : float
+ getFat() : float
+ getProtein() : float
+ getCarbs() : float
+ getSodium() : float
1
*
«interface»
iNourishment
getName() : String
getCalories() : float
getFat() : float
getProtein() : float
getCarbs() : float
getSodium() : float
COMPOSITE PATTERN
*
1
HealthProgram
- foodCollection : Map<String, iNourishment[]>
- userData : UserData
+ HealthProgram() <<Constructor>>
+ getLog(year: int, month: int, day: int) : Log
+ createLog(year: int, month: int, day: int, weight: float, desiredCalories: float, foodName: List<String>) : void
+ updateLog(year: int, month: int, day: int, weight: float, desiredCalories: float, foodName: List<String>) : void
+ deleteLog(year: int, month: int, day: int) : void
+ getNourishment(foodName: String) : void
+ deleteNourishment(foodName: String) : void
+ createFood(foodName: String, calories: float, fat: float, carbs: float, protein: float, sodium: float) : void
+ updateFood(foodName: String, calories: float, fat: float, carbs: float, protein: float, sodium: float) : void
+ createRecipe(foodName: String, ingredients: List<String>, quantities: List<Float>) : void
+ updateRecipe(foodName: String, ingredients: List<String>, quantities: List<Float>) : void
1
1
UserData
- nourishmentCollection : Map<String, iNourishment>
- logs : Map<String, Log>
+ UserData() <<Constructor>>
+ getNourishment(name:String): iNourishment
+ updateNourishment(name:String, newNourishment:iNourishment) : void
+ addNourishment(nourishmentItem: iNourishment) : void
+ deleteNourishment(name:String) : void
+ getLog(year: int, month: int, day: int) : Log
+ createLog(year: int, month: int, day: int, weight: float, desiredCalories: float, foodNames: List<String>) : Log
+ createLog(year: int, month: int, day: int) : Log <<Overload>>
+ updateLog(year: int, month: int, day: int, weight: float, desiredCalories: float, foodNames: List<String>) : Log
+ deleteLog(year: int, month: int, day: int) : void
- date() : LocalDate
+ loadNourishment(path: String) : void
+ loadLogs(path: String) : void
+ saveLogs(path: String): void
1
*
Log
year : int
month : int
day : int
weight : float
desiredCalories : float
foodItems : Map<iNourishment, Float>
+ Log(year:int, month:int, day:int, weight:float, desiredCalories:float) <<Constructor>>
+ setWeight(weight: float) : void
+ setDesiredCalories(desiredCalories: float) : void
+ addNourishment(nourishment : iNourishment) : void
+ toFormat(): String
+ toString() : String
OBSERVER PATTERN
«interface»
Observer
Observable
ViewCLI
- HealthProgram program
- LogAction logAction
- FoodAction foodAction
- Scanner scanner
+ ViewCLI(HealthProgram) <<Constructor>>
+ update(Observable, Object): void
+ display(): void
- intgetChoice(): int
+ createLog(): void
+ viewLog(): void
+ updateLog(): void
+ deleteLog(): void
+ createFood(): void
+ updateFood(): void
+ viewFood(): void
+ deleteFood(): void
+ createRecipe(): void
+ updateRecipe(): void
+ saveNourishment(): void
+ saveLogs(): void
View_GUI
FoodAction
- program : HealthProgram
+ FoodAction(program: HealthProgram) <<Constructor>>
+ createFood(foodName: String, calories: float, fat: float, carbs: float, protein: float, sodium: float) : void
+ updateFood(foodName: String, calories: float, fat: float, carbs: float, protein: float, sodium: float) : void
+ viewFood(foodName: String) : void
+ createRecipe(foodName: String, ingredients: List<String>, quantities: List<Float>) : void
+ updateRecipe(foodName: String, ingredients: List<String>, quantities: List<Float>) : void
+ deleteNourishmentItem(name: String) : void
+ saveNourishment(): void
LogAction
- program : HealthProgram
+ LogAction(program: HealthProgram) <<Constructor>>
+ createLog(year: int, month: int, day: int, weight: float, desiredCalories: float) : void
+ updateLog(year: int, month: int, day: int, weight: float, desiredCalories: float) : void
+ viewLog(year: int, month: int, day: int) : void
+ deleteLog(year: int, month: int, day: int): void
+ deleteLog(year: int, month: int, day: int) : void
+ saveLogs(): void
«interface»
UserInterface

6 Pattern Usage

6.1 MVC

MVC Pattern	
Model	Takes care of all the data like food, exercise and log management.
View	Shows the user some data, like the amount of calories consumed, amount of burned calories or their changes in weight over time.
Controller	Connects the two and handles user input (food and exercise in the daily log) and refreshes the model or view accordingly.

6.2 Composite

Composite Pattern	
Component	iNourishment
Leaf	Food
Composite	Recipe

6.3 Observer

Observer Pattern	
Observer(s)	ViewCLI
Observable(s)	HealthProgram

7 RATIONALE

One of the decisions we made was to use a composite pattern with the food in our application. In the structure of a composite pattern here, the food objects serve as components. The recipe can serve as the composite since it is made up of food, and can also contain other recipes. Additionally, the ingredients can be a leaf to the food. These all come together to form a hierarchy for our food, ingredients and recipes. By using the composite pattern here, it allows for flexibility with the food, and uniform handling of anything food-related.

In our application, the observer pattern keeps the user's data synchronized with any changes to logs and data. The LogEntry and UserData classes serve as our observables, which will notify the observers whenever there are updates to the users food logs and other information. The UserInterface and LogDisplay are our observers, which automatically update to reflect any changes in the data such as new food being added. By using the observer pattern, we can ensure that the User Interface will remain consistently up to date with any changes made without having to manually refresh it.

Our decision to use the MVC pattern helps with the maintainability of our application. The Model has the responsibility of managing the core of the application, such as the user’s food entries, recipes and logs. It handles all of our logic related to tracking of nutrients and calories as well. The View presents our user with their information, such as their calories, food entries and recipes. This ensures that the user has a clear look at their data. The Controller is the middle man for the two, it processes all of the user input such as adding food to the log. It updates the Model with any new data, and the View then reflects the new information. This makes our application easier to maintain and upgrade.