NutriApp 4-18-23

Design Documentation
Prepared by TEAM 3:

- Jason Berry jsb9886@rit.edu
- Collin Cleary cjc7891@rit.edu
- Logan Holmes lsh8770@rit.edu
- Nikhil Patil nsp4746@rit.edu
- Aidan Ruiz ajr3293@rit.edu

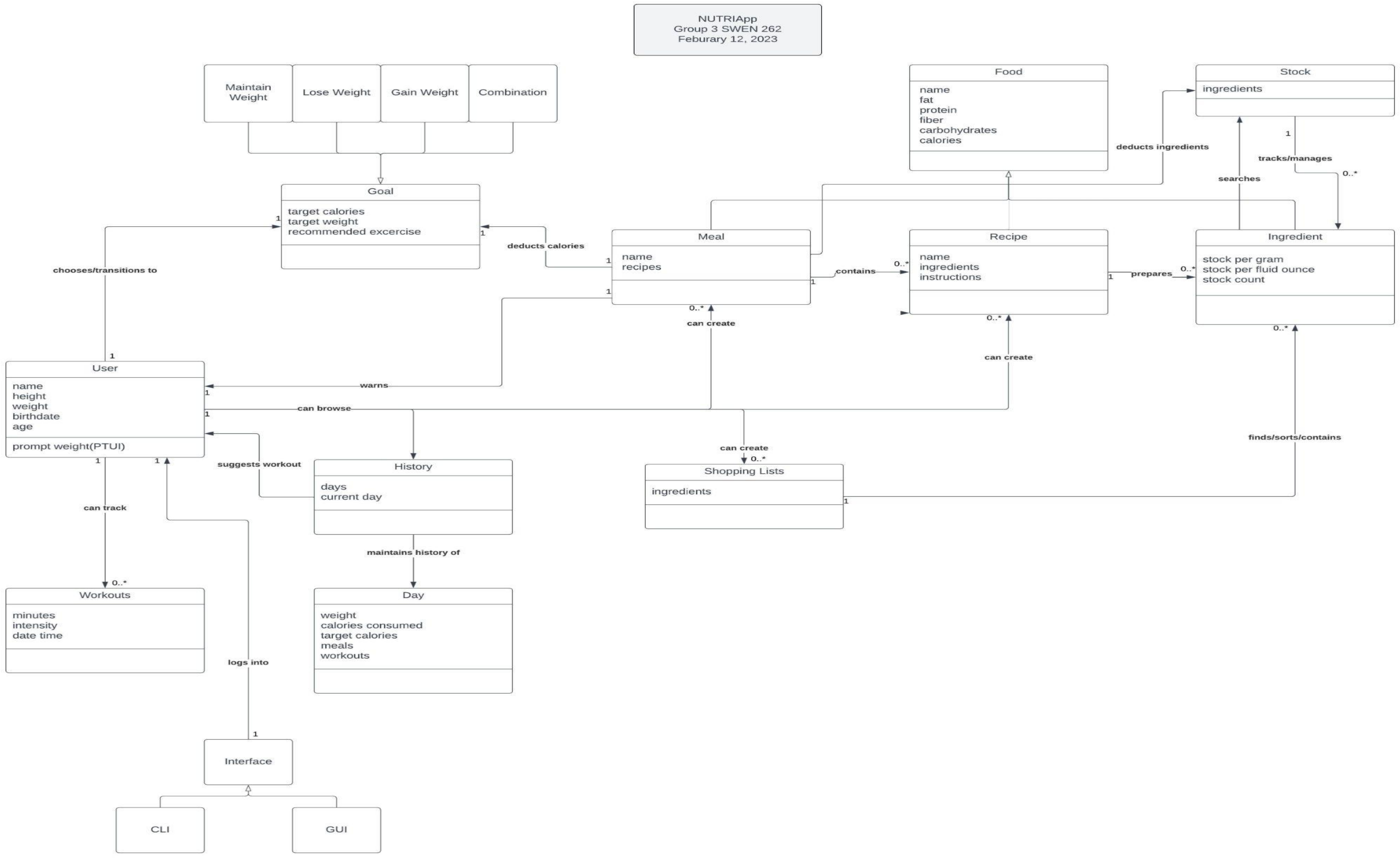
Summary

This section provides a brief overview of the project.

This project is known as NutriApp. It is a health and fitness tracking app. The features of the app will be discussed in brief. A user can enter their name, age (as birthdate), and weight into the app. The user can also select from maintain weight, lose weight, and gain weight as their goals, along with a combination of these. They can also browse their personal history and activity on the app. The app keeps track of various forms of food, which are made up of ingredients. Ingredients can then be used by recipes to define a meal. The app also allows for users to create a shopping list, and for them to track workouts. Additionally, as the project was developed into version 2.0, the app will now start in guest mode. Users can now also form teams as well and invite other users to join their team. The user is also able to undo commands now as well. Last, but not least, the User can now export their history data in a JSON file. The project was developed and designed by Group 3, whose members are as follows: Jason Berry, Collin Cleary, Logan Holmes, Nikhil Patil, and Aidan Ruiz. This document provides a brief description of NutriApp's various architecture, and the approach Group 3 took to developing it.

Domain Model

This section provides a domain model for the project. It should follow the guidelines discussed in class and the design project activity sheets. For it to be readable, you may need to turn this page into landscape mode.

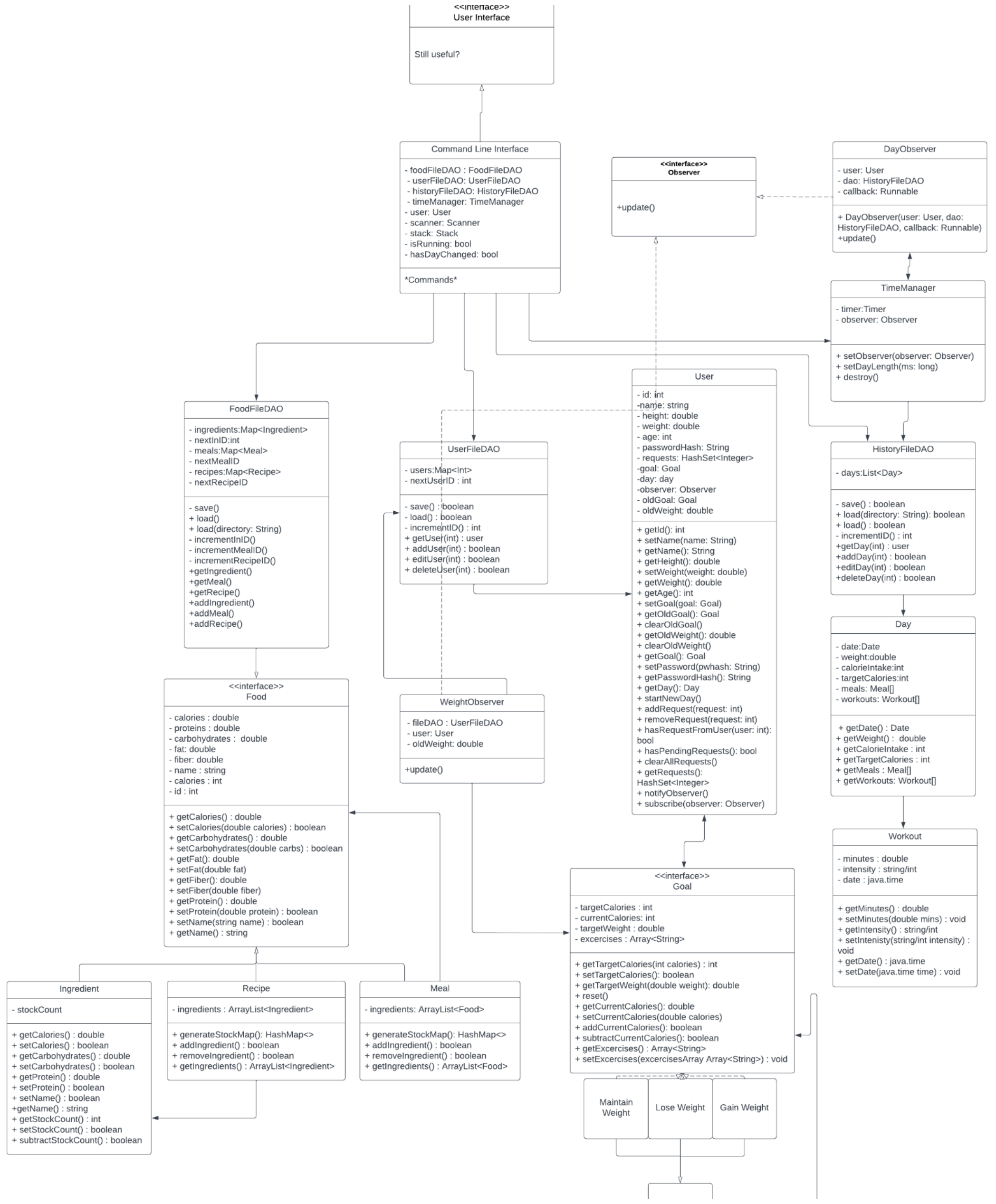


System Architecture

This section provides a model of the subsystem components that make up the overall software architecture for the project. Draw the subsystems as simple boxes with relationships between them. Provide a narrative that describes the responsibilities of each component and the interfaces that are provided between subsystems.

Class Diagram

One of the many subsystems



Subsystems

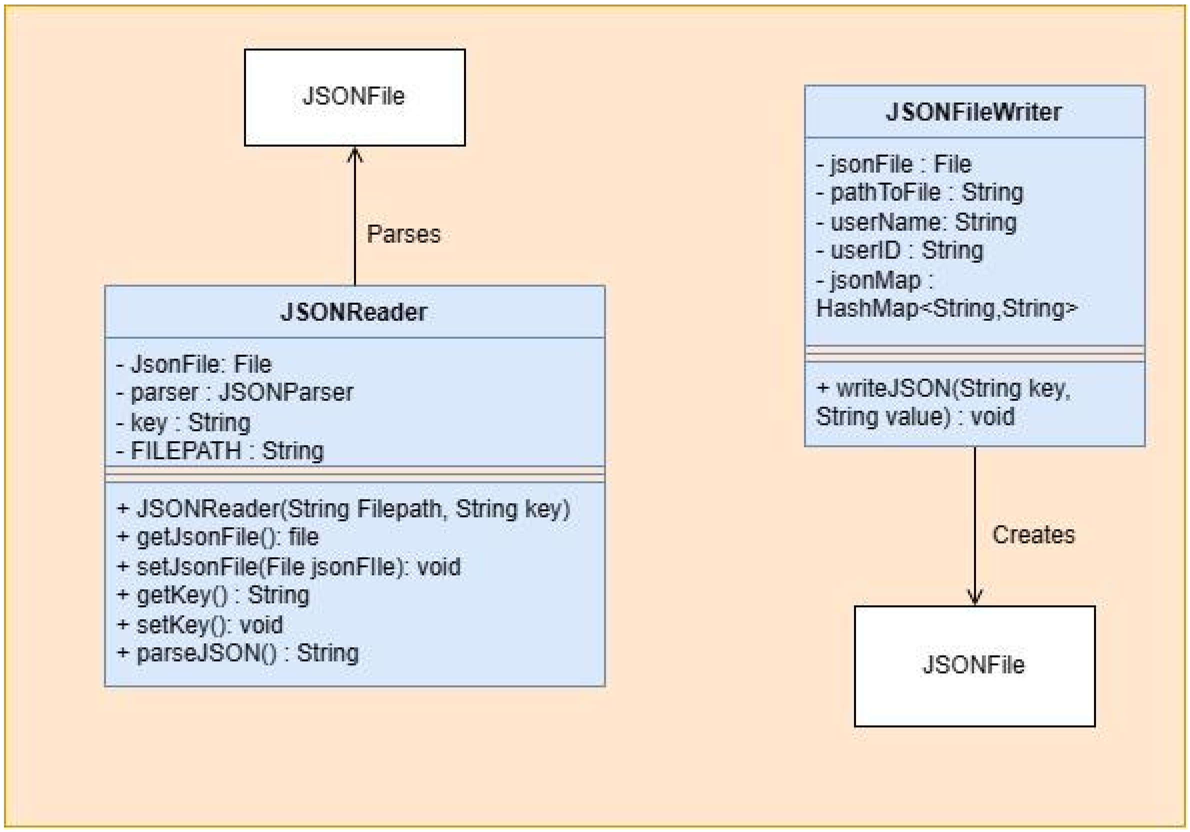
This section provides detailed design for specific subsystems described in the system architecture.

Name of the subsystem

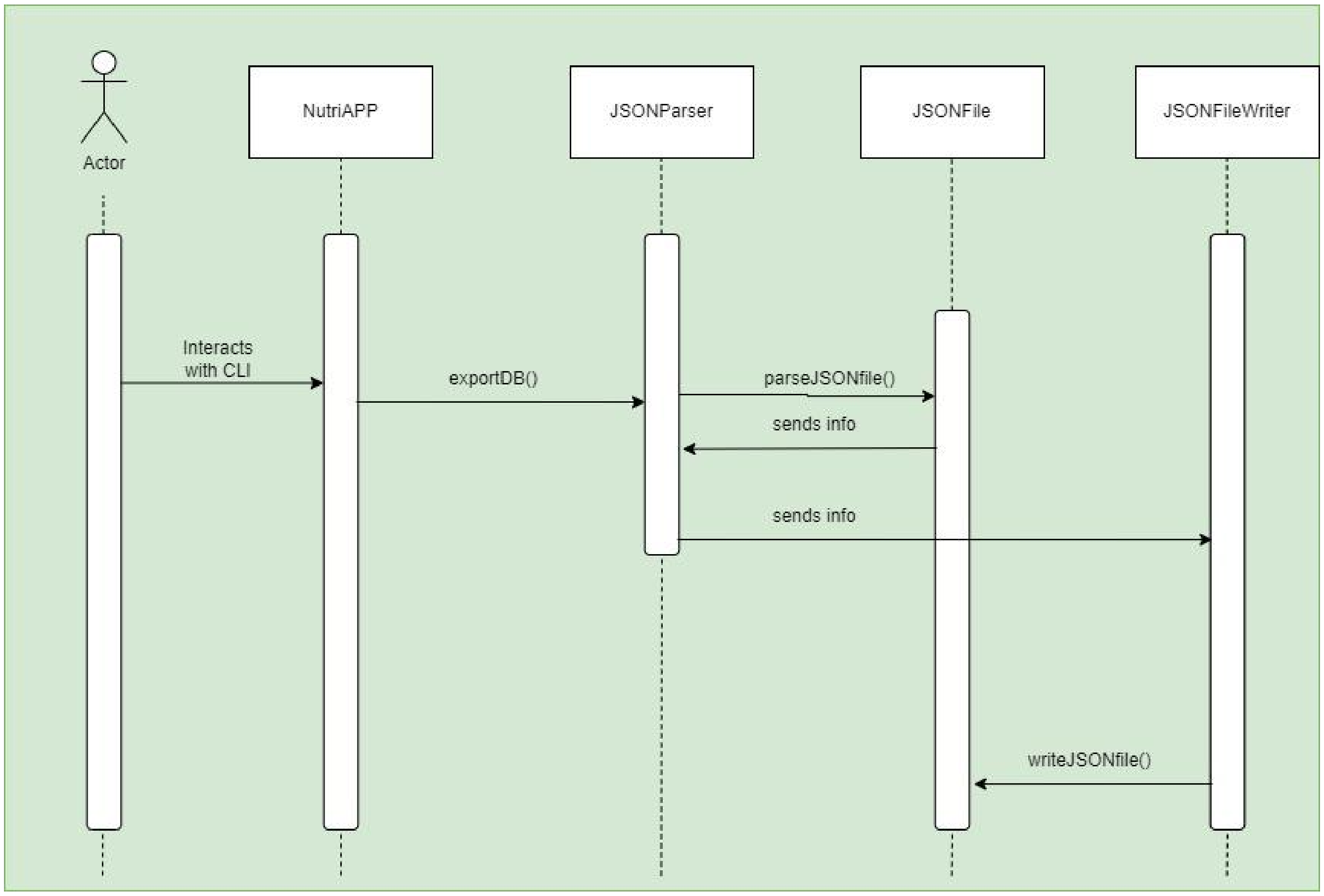
In this section, provide the following information for the first subsystem.

- Class structure diagram and a narrative that describes the structure of this subsystem
- Sequence diagrams with associated narratives that describe the dynamic behaviors that are primarily located within this subsystem. Within your subsystem design descriptions, you must make sure to provide sequence diagrams for all features listed in the design project problem statement. You may also decide that other features require documentation within the subsystems.
- A description of all design patterns that are primarily located within this subsystem. Use the table below to describe each design pattern. If a design pattern cuts across the boundary of subsystems, place the pattern usage table in the section for the subsystem that holds the majority of pattern participants.

JSONFiles



The Design Pattern that would have been used in this subsystem is the adapter design pattern if the requirements of R2 did not change.
Below is the sequence diagram.

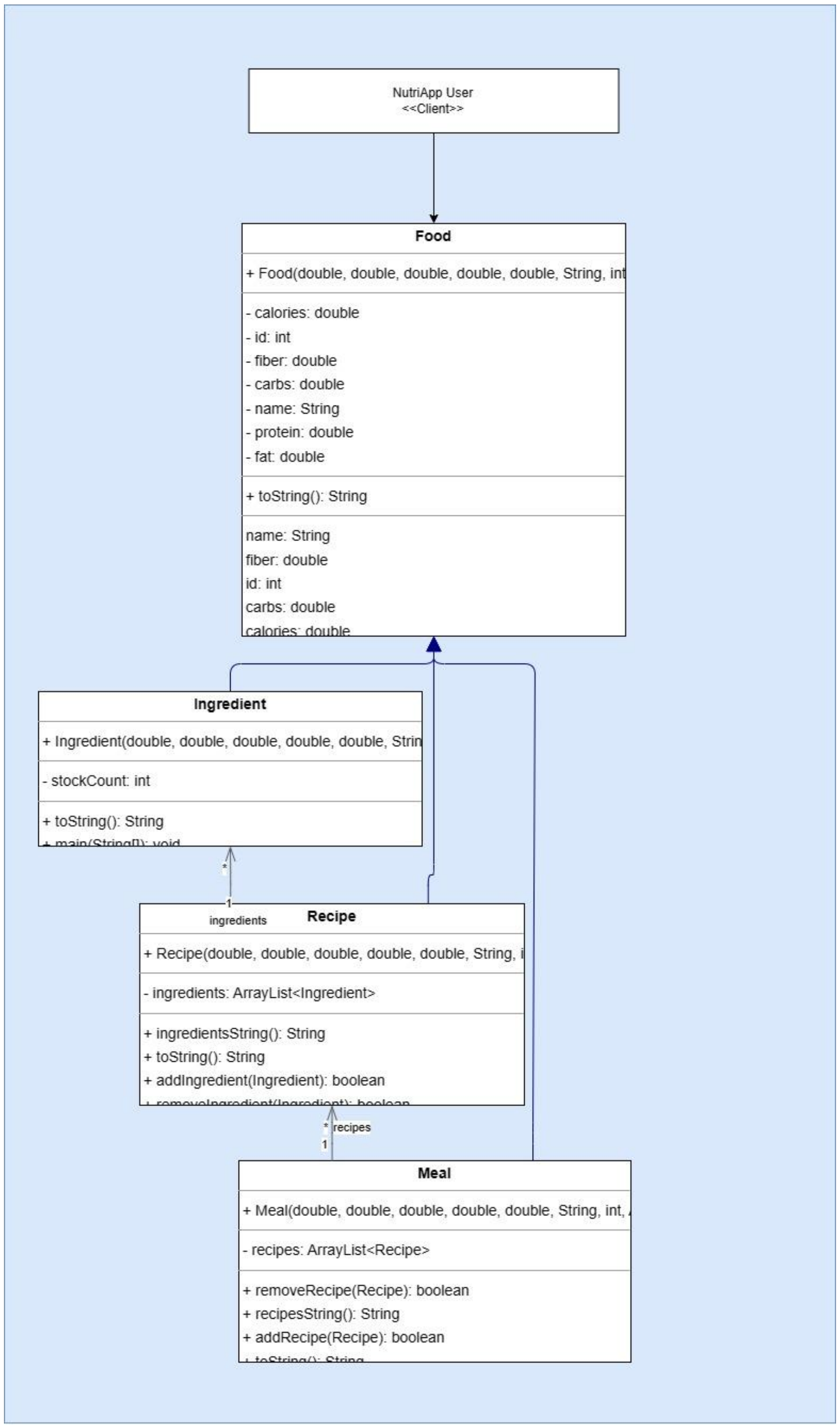

Actor
NutriAPP
JSONParser
JSONFile
JSONFileWriter
Interacts with CLI
exportDB()
parseJSONfile()
sends info
sends info
writeJSONfile()

Food Subsystem

The first subsystem is the Food subsystem. The Food subsystem contains the functionality of the food class, which contains other classes such as Recipes, Ingredients, and Meals. The design pattern that can be found in this subsystem is the composite pattern, as they all build on one another.

Generic GoF Design Pattern Name: **Composite Pattern**	
Pattern Name in terms of system context: **Food & its various subclasses.**	
List one per line the participants from the GoF class structure (add rows, if needed)	Nouns from your table that take on that participant role. If there is no noun for a participant role explain why.
Food	**Component**
Ingredients	**Leaf**
Recipe	**Composite**
Meal	**Composite**
Short (several sentences) narrative description of how the design pattern is being used in this application. Should be tied to system description and requirements. **This is a fundamental requirement of the NutriApp, as food, recipes, and meals are required in the nutriapp. This design pattern will be used as Food is present in Recipes and Meals. They are essentially the same object, just different applications of it. Ingredients are food, but are only found within Recipes when it comes to our design pattern. They store nutritional data, and stock count. Recipes are a child of Meals in this design as well. They contain a collection of ingredients as well as a list of steps (String) required to prepare the recipe. Meals are a collection of recipes with a unique name.**	

Below is the class diagram

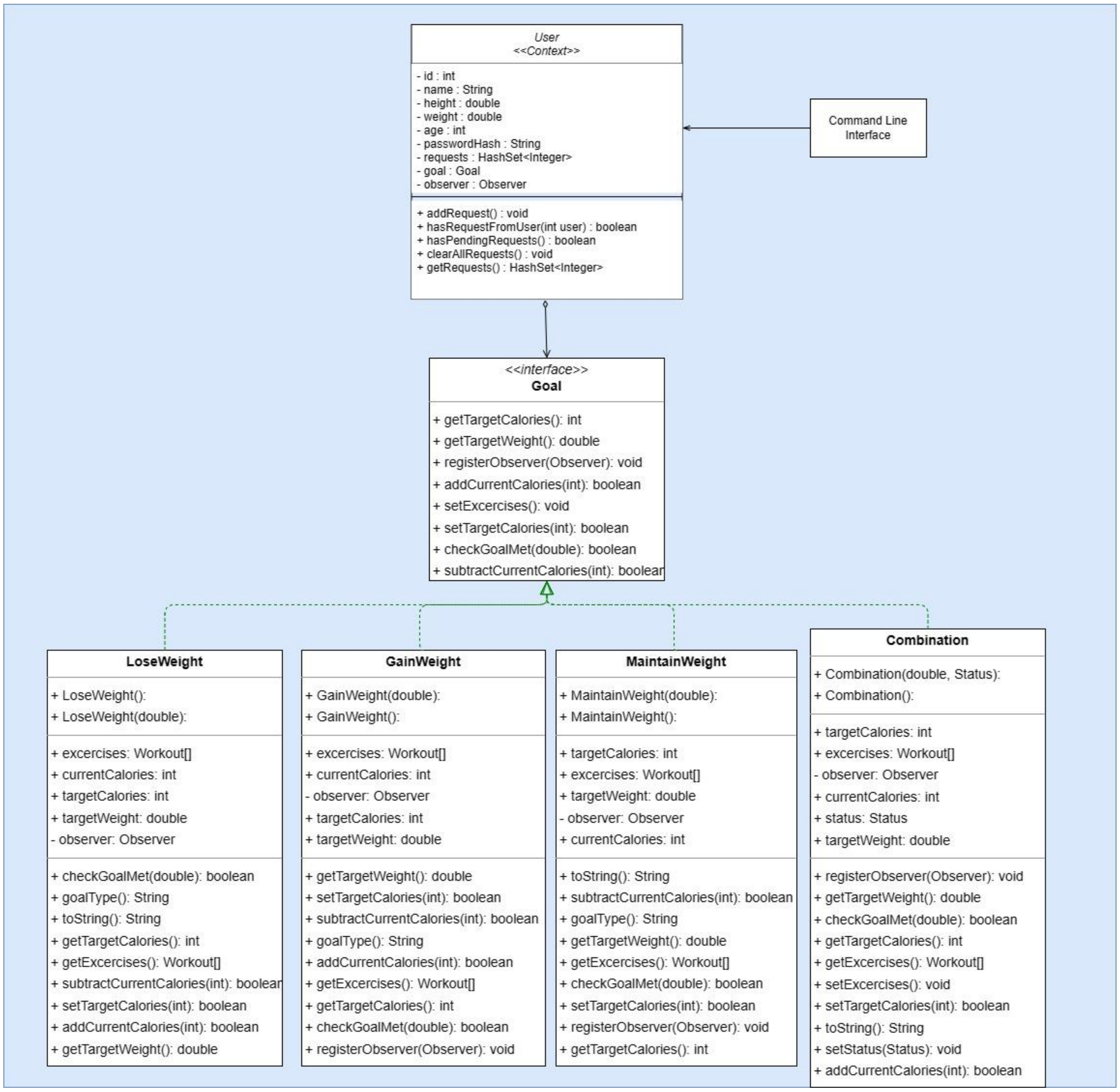

NutriApp User
<<Client>>
Food
+ Food(double, double, double, double, double, String, int
- calories: double
- id: int
- fiber: double
- carbs: double
- name: String
- protein: double
- fat: double
+ toString(): String
name: String
fiber: double
id: int
carbs: double
calories: double
Ingredient
+ Ingredient(double, double, double, double, double, Strin
- stockCount: int
+ toString(): String
+ main(String[]): void
*
1
ingredients
Recipe
+ Recipe(double, double, double, double, double, String, i
- ingredients: ArrayList<Ingredient>
+ ingredientsString(): String
+ toString(): String
+ addIngredient(Ingredient): boolean
+ removeIngredient(Ingredient): boolean
* recipes
1
Meal
+ Meal(double, double, double, double, double, String, int,
- recipes: ArrayList<Recipe>
+ removeRecipe(Recipe): boolean
+ recipesString(): String
+ addRecipe(Recipe): boolean
+ toString(): String

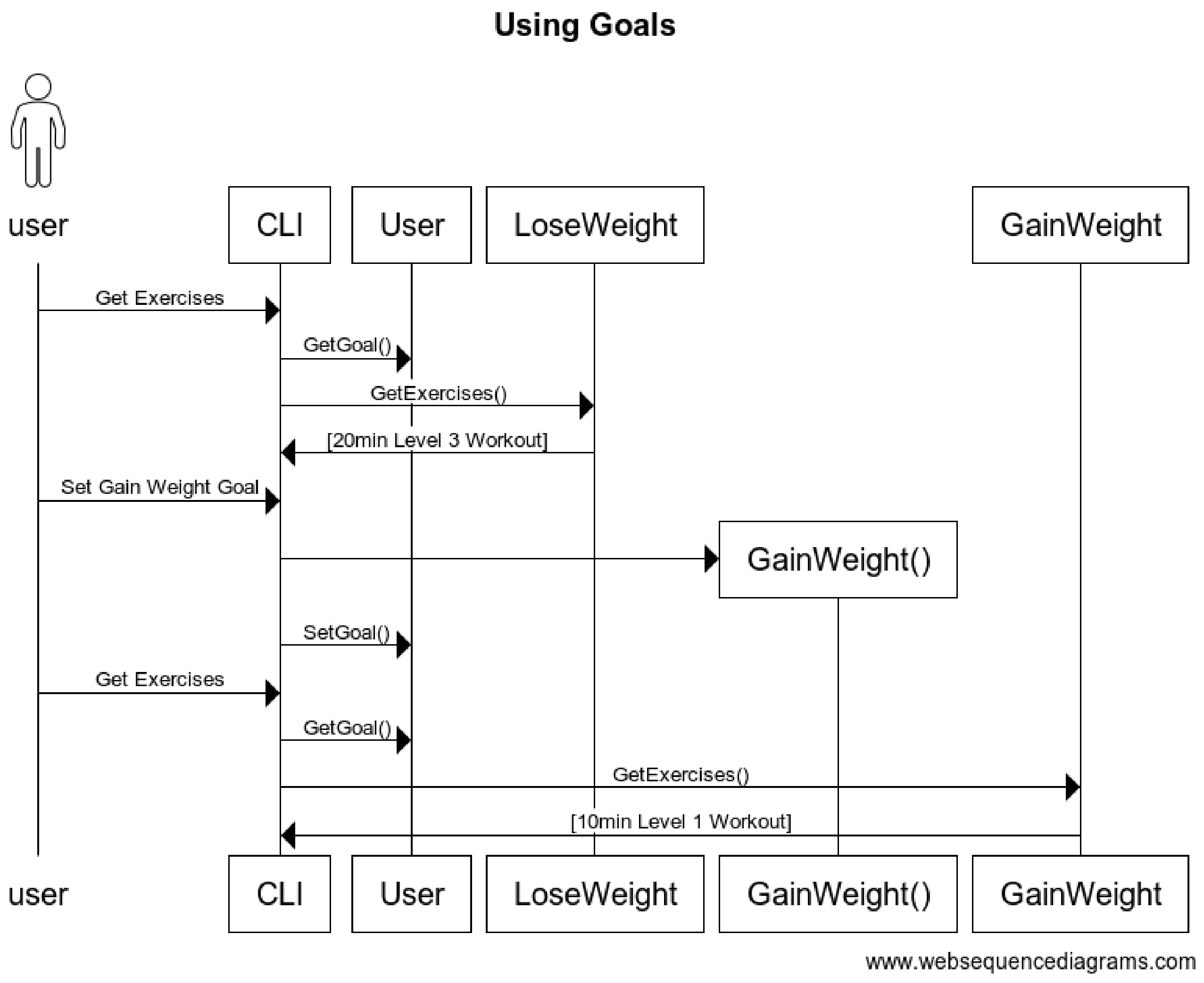
Goal Subsystem

The name of this subsystem is the Goal subsystem. This contains the goal class and all of its functionality, along with the different types of goals that can be set by the user. Ultimately, strategy was decided as the user is capable of changing the goal during runtime, as well as the goals changing themselves.

Generic GoF Design Pattern Name: **Strategy Pattern**	
Pattern Name in terms of system context: **Goal**	
List one per line the participants from the GoF class structure (add rows, if needed)	Nouns from your table that take on that participant role. If there is no noun for a participant role explain why.
Goal	**Strategy Interface**
Maintain Weight	**Concrete Strategy**
Lose Weight	**Concrete Strategy**
Gain Weight	**Concrete Strategy**
Combination	**Concrete Strategy**
User	**Context**
Client	**Command Line Interface**
Short (several sentences) narrative description of how the design pattern is being used in this application. Should be tied to system description and requirements. **How the app functions will change depending on the goal chosen, lighter meals or more intensive workouts might be recommended if the goal is to lose weight. This all changes if the goal changes, the goal will also automatically change itself depending on the state of the user. If the user is trying to reach a certain weight, after they reach that weight, the goal will change to maintain that weight level.**	

Class Diagram:


User
<<Context>>
- id : int
- name : String
- height : double
- weight : double
- age : int
- passwordHash : String
- requests : HashSet<Integer>
- goal : Goal
- observer : Observer
+ addRequest() : void
+ hasRequestFromUser(int user) : boolean
+ hasPendingRequests() : boolean
+ clearAllRequests() : void
+ getRequests() : HashSet<Integer>
Command Line Interface
<<interface>>
Goal
+ getTargetCalories(): int
+ getTargetWeight(): double
+ registerObserver(Observer): void
+ addCurrentCalories(int): boolean
+ setExcercises(): void
+ setTargetCalories(int): boolean
+ checkGoalMet(double): boolean
+ subtractCurrentCalories(int): boolean
LoseWeight
+ LoseWeight():
+ LoseWeight(double):
+ excercises: Workout[]
+ currentCalories: int
+ targetCalories: int
+ targetWeight: double
- observer: Observer
+ checkGoalMet(double): boolean
+ goalType(): String
+ toString(): String
+ getTargetCalories(): int
+ getExcercises(): Workout[]
+ subtractCurrentCalories(int): boolean
+ setTargetCalories(int): boolean
+ addCurrentCalories(int): boolean
+ getTargetWeight(): double
GainWeight
+ GainWeight(double):
+ GainWeight():
+ excercises: Workout[]
+ currentCalories: int
- observer: Observer
+ targetCalories: int
+ targetWeight: double
+ getTargetWeight(): double
+ setTargetCalories(int): boolean
+ subtractCurrentCalories(int): boolean
+ goalType(): String
+ addCurrentCalories(int): boolean
+ getExcercises(): Workout[]
+ getTargetCalories(): int
+ checkGoalMet(double): boolean
+ registerObserver(Observer): void
MaintainWeight
+ MaintainWeight(double):
+ MaintainWeight():
+ targetCalories: int
+ excercises: Workout[]
+ targetWeight: double
- observer: Observer
+ currentCalories: int
+ toString(): String
+ subtractCurrentCalories(int): boolean
+ goalType(): String
+ getTargetWeight(): double
+ getExcercises(): Workout[]
+ checkGoalMet(double): boolean
+ setTargetCalories(int): boolean
+ registerObserver(Observer): void
+ getTargetCalories(): int
Combination
+ Combination(double, Status):
+ Combination():
+ targetCalories: int
+ excercises: Workout[]
- observer: Observer
+ currentCalories: int
+ status: Status
+ targetWeight: double
+ registerObserver(Observer): void
+ getTargetWeight(): double
+ checkGoalMet(double): boolean
+ getTargetCalories(): int
+ getExcercises(): Workout[]
+ setExcercises(): void
+ setTargetCalories(int): boolean
+ toString(): String
+ setStatus(Status): void
+ addCurrentCalories(int): boolean



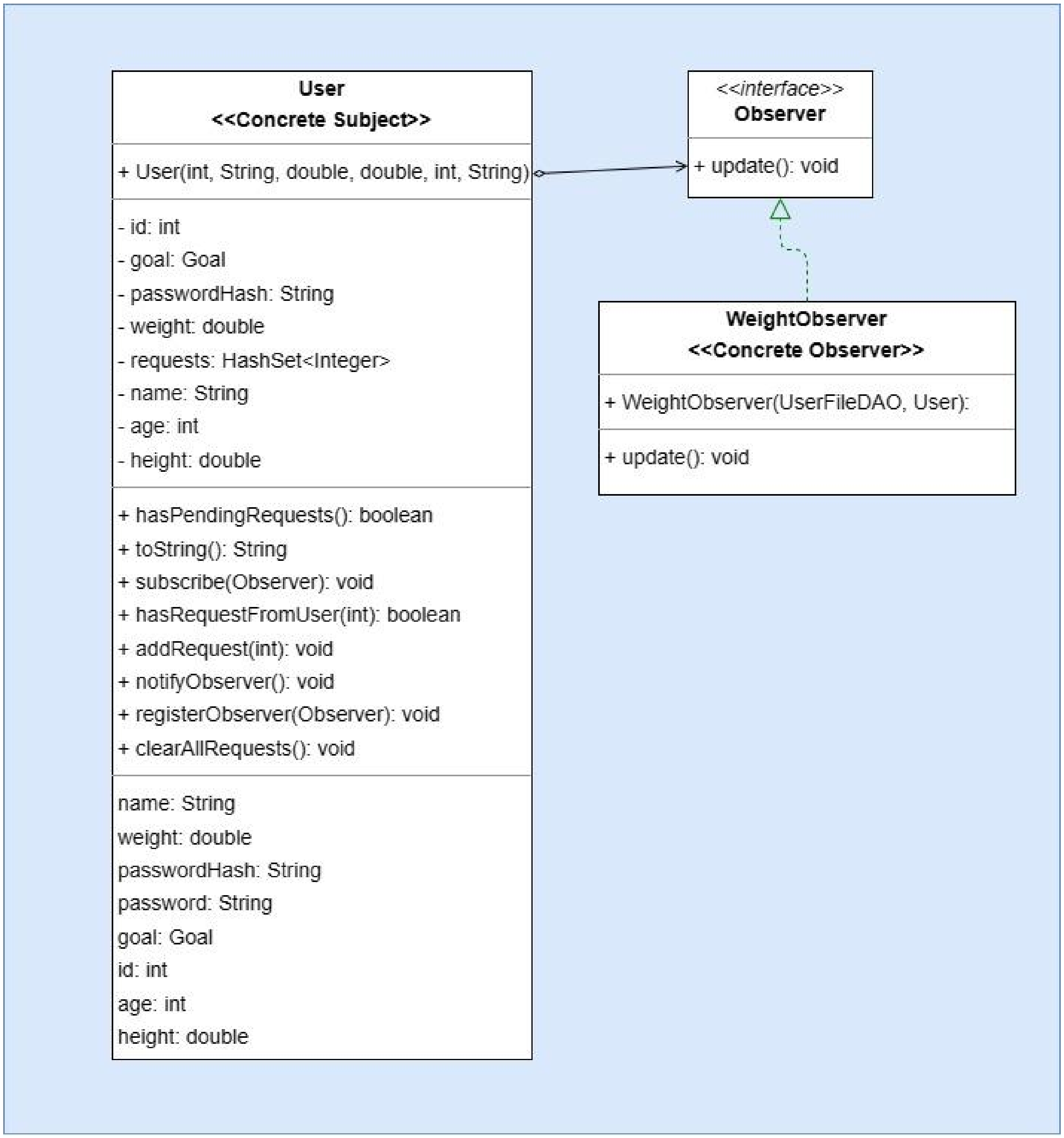
Goal Adapter

This subsystem is the Goal adapter. It contains the classes that deal with observing the goals. The design pattern used in this subsystem is the observer pattern as the app needs to observe when a user has met their goal, and notifies the goal to change accordingly.

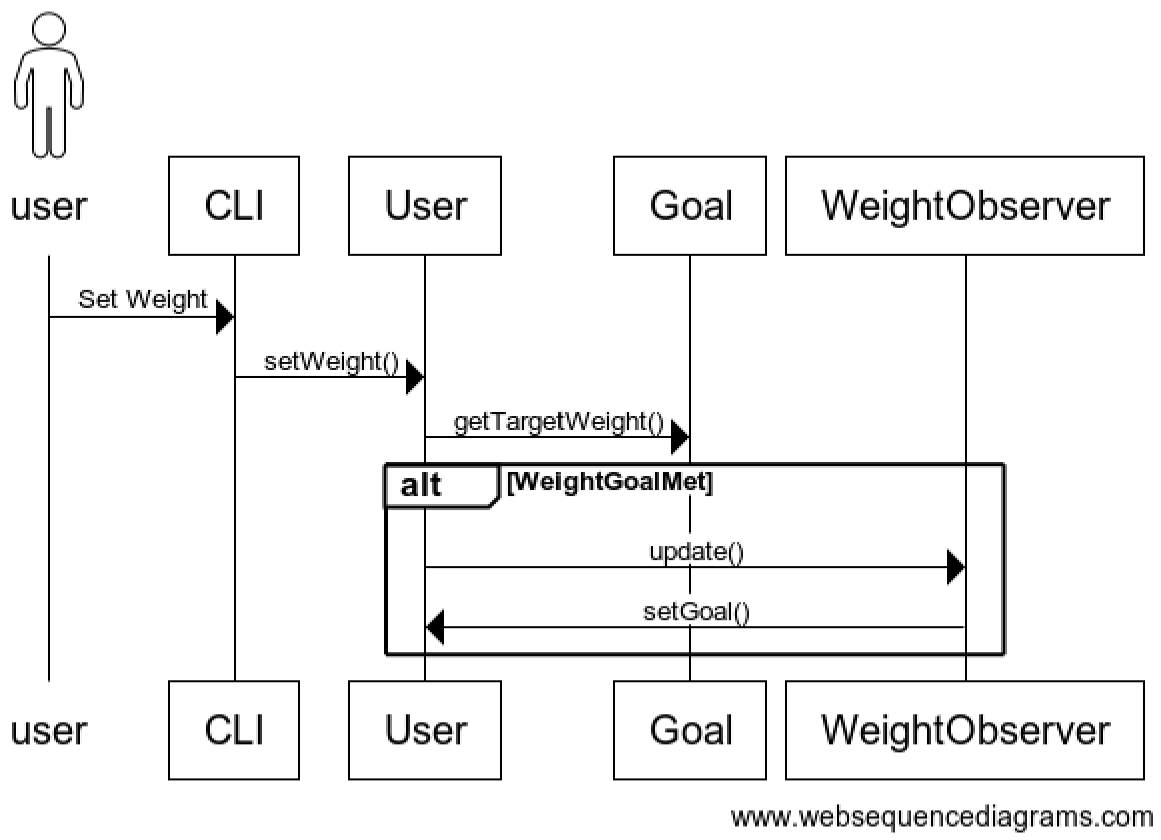
Generic GoF Design Pattern Name: **Observer pattern**	
Pattern Name in terms of system context: **Goal adapter**	
List one per line the participants from the GoF class structure (add rows, if needed)	Nouns from your table that take on that participant role. If there is no noun for a participant role explain why.
User	**Concrete Subject**
Observer	**Observer**
Weight Observer	**Concrete Observer**
Short (several sentences) narrative description of how the design pattern is being used in this application. Should be tied to system description and requirements.	

The App needs to be aware of the user's weight in order to effectively change the goals when they are achieved. A requirement of the app is the dynamic management of the goals outside of user intervention and an observer system would be the best way to do that. The observers would subscribe to the user and watch their weight attribute, and signal based on the user's current weight versus their target weight.

Below is the class diagram.


User
<<Concrete Subject>>
+ User(int, String, double, double, int, String)
- id: int
- goal: Goal
- passwordHash: String
- weight: double
- requests: HashSet<Integer>
- name: String
- age: int
- height: double
+ hasPendingRequests(): boolean
+ toString(): String
+ subscribe(Observer): void
+ hasRequestFromUser(int): boolean
+ addRequest(int): void
+ notifyObserver(): void
+ registerObserver(Observer): void
+ clearAllRequests(): void
name: String
weight: double
passwordHash: String
password: String
goal: Goal
id: int
age: int
height: double
<<interface>>
Observer
+ update(): void
WeightObserver
<<Concrete Observer>>
+ WeightObserver(UserFileDAO, User):
+ update(): void

Weight Observer



History Subsystem

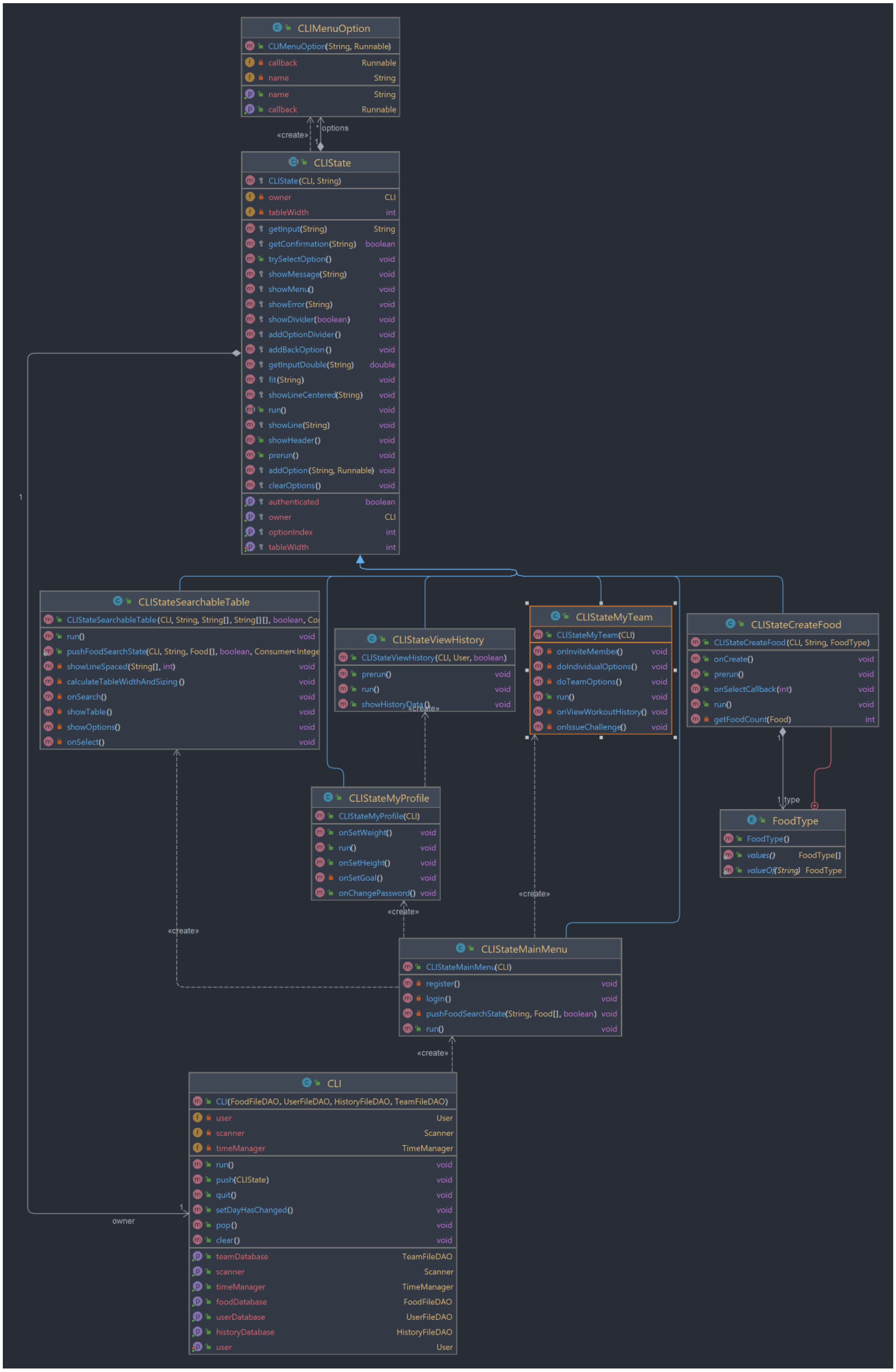
The name of this subsystem is the history subsystem. This has classes pertaining to the history of the user. The design pattern found in this is Memento. The user is the one who creates the history, the class called History is the one who manages the user History, and the class called Day contains the information about the user on that day.

Generic GoF Design Pattern Name: **Memento**	
Pattern Name in terms of system context: **History**	
List one per line the participants from the GoF class structure (add rows, if needed)	Nouns from your table that take on that participant role. If there is no noun for a participant role explain why.
Originator	**User**
Caretaker	**StateManager**
Memento	**State**
Short (several sentences) narrative description of how the design pattern is being used in this application. Should be tied to system description and requirements. **The history of the user is to be stored for them to look at later. In order for this to work, the history of the user has to be encapsulated into objects. The history is stored as “state” objects which are a generic type and can store any data type.**	

CLI State System

The name of this subsystem is the CLI State System. This subsystem has all the classes pertaining to the CLI. It utilizes the state design pattern, as there are different types of states for the CLI.

Generic GoF Design Pattern Name: **State**	
Pattern Name in terms of system context: **CLI State System**	
List one per line the participants from the GoF class structure (add rows, if needed)	Nouns from your table that take on that participant role. If there is no noun for a participant role explain why.
Context	**CLI**
Context	**CLIMenuOption**
State	**CLIState**
Concrete State	**CLIStateCreateFood**
Concrete State	**CLIStateMainMenu**
Concrete State	**CLIStateMyProfile**
Concrete State	**CLIStateMyTeam**
Concrete State	**CLIStateSearchableTable**
Concrete State	**CLIStateViewHistory**
Short (several sentences) narrative description of how the design pattern is being used in this application. Should be tied to system description and requirements. **The CLI is what the user interacts with. This has many states to meet the needs of the PTUI. This also the front end that the user sees.**	


CLIMenuOption
CLIMenuOption(String, Runnable)
callback Runnable
name String
name String
callback Runnable
«create»
options
CLIState
CLIState(CLI, String)
owner CLI
tableWidth int
getInput(String) String
getConfirmation(String) boolean
trySelectOption() void
showMessage(String) void
showMenu() void
showError(String) void
showDivider(boolean) void
addOptionDivider() void
addBackOption() void
getInputDouble(String) double
fit(String) void
showLineCentered(String) void
run() void
showLine(String) void
showHeader() void
prerun() void
addOption(String, Runnable) void
clearOptions() void
authenticated boolean
owner CLI
optionIndex int
tableWidth int
1
CLIStateSearchableTable
CLIStateSearchableTable(CLI, String, String[], String[][], boolean, Co
run() void
pushFoodSearchState(CLI, String, Food[], boolean, Consumer<Intege
showLineSpaced(String[], int) void
calculateTableWidthAndSizing() void
onSearch() void
showTable() void
showOptions() void
onSelect() void
CLIStateViewHistory
CLIStateViewHistory(CLI, User, boolean)
prerun() void
run() void
showHistoryData() void
«create»
CLIStateMyTeam
CLIStateMyTeam(CLI)
onInviteMember() void
doIndividualOptions() void
doTeamOptions() void
run() void
onViewWorkoutHistory() void
onIssueChallenge() void
CLIStateCreateFood
CLIStateCreateFood(CLI, String, FoodType)
onCreate() void
prerun() void
onSelectCallback(int) void
run() void
getFoodCount(Food) int
1
1 type
FoodType
FoodType()
values() FoodType[]
valueOf(String) FoodType
CLIStateMyProfile
CLIStateMyProfile(CLI)
onSetWeight() void
run() void
onSetHeight() void
onSetGoal() void
onChangePassword() void
«create»
«create»
«create»
CLIStateMainMenu
CLIStateMainMenu(CLI)
register() void
login() void
pushFoodSearchState(String, Food[], boolean) void
run() void
«create»
CLI
CLI(FoodFileDAO, UserFileDAO, HistoryFileDAO, TeamFileDAO)
user User
scanner Scanner
timeManager TimeManager
run() void
push(CLIState) void
quit() void
setDayHasChanged() void
pop() void
clear() void
teamDatabase TeamFileDAO
scanner Scanner
timeManager TimeManager
foodDatabase FoodFileDAO
userDatabase UserFileDAO
historyDatabase HistoryFileDAO
user User
owner
1

Status of the Implementation

Provide a complete description of the status of your implementation. This should specify all known defects in the system, and indicate requirements that your implementation does not cover.

NtruiApp’s implementation is complete, however we are missing the feature of members of teams being notified when other team members log a workout. Although the system can be difficult to navigate and understand, given that only numbers are used to enter commands. The system is completely functional, ingredients and other data can be imported, recipes and meals can be made and eaten, workouts can be performed. Teams can be joined.

Appendix

This section provides fine-grained design details for all of the classes in your design. You will capture this information using the CRC (Class-Responsibilities-Collaborators) card format below.

Class: `App`	
Responsibilities: Starts the application by instantiating the persistence file manager classes, and runs the command line interface.	
Collaborators: FoodFileDAO, UserFileDAO, HistoryFileDAO, TeamFileDAO, CLI	
Users: ...	**Used by:** ...
Author: Aidan Ruiz + Group 3	

Observers

Class: `Observer`	
Responsibilities: Interface which defines implementation for concrete observers	
Collaborators: User	
Users: DayObserver, WeightObserver	**Used by:** ...
Author: Logan Holmes + Group 3	

Class: `WeightObserver`	
Responsibilities: Updates a user's goal based on their weight changes.	
Collaborators:	
Users: User	**Used by:** ...
Author: ... Logan Holmes + Group 3	

Class: `DayObserver`	
Responsibilities: Notifies class when the end of the day has been reached.	
Collaborators: TimeManager	
Users:	**Used by:** ...
Author: Logan Holmes + Group 3	

Weight Goals

Class: `Goal`	
Responsibilities: Defines an interface for user weight goals.	
Collaborators: User	
Users: MaintainWeight, LooseWeight, GainWeight, Combination	**Used by:** ...
Author: Collin Cleary + Group 3	

Class: `MaintainWeight`	
Responsibilities: Defines workouts to be used to achieve goal, and caloric intake.	
Collaborators:	
Users: User	**Used by:** ...
Author: Collin Cleary + Group 3	

Class: `LoseWeight`	
Responsibilities: Defines workouts to be used to achieve goal, and caloric intake.	
Collaborators:	
Users: User	**Used by:** ...
Author: Collin Cleary + Group 3	

Class: `GainWeight`	
Responsibilities: Defines workouts to be used to achieve goal, and caloric intake.	
Collaborators:	
Users: User	**Used by:** ...
Author: Collin Cleary + Group 3	

Class: `Combination`	
Responsibilities: Combines several aspects	

of gain weight, lose weight and/or maintain weight into a single goal.	
Collaborators:	
Users: User	**Used by:** ...
Author: Collin Cleary + Group 3	

Class: `User`	
Responsibilities: Stores various state attributes of the user. Goals, name, age, weight, password, id.	
Collaborators: Goal, Meal, Food, Workout	
Users: CLI	**Used by:** ...
Author: Collin Cleary + Group 3	

Class: `Team`	
Responsibilities: Contains a list of members (id's of users).	
Collaborators: User	
Users: CLI	**Used by:** ...
Author: Collin Cleary + Group 3	

Class: `Workout`	
Responsibilities: Contains info on duration, and intensity of a workout.	
Collaborators: Collin Cleary + Group 3	
Users: User	**Used by:** ...
Author: ...	

Food classes

Class: `Food`	
Responsibilities: Stores info on a food's nutritional value, calories, protein, carbs, fiber, fat, and name.	

Collaborators: ...	
Users: Ingredient, Meal, Recipe	**Used by:** ...
Author: Collin Cleary + Nikhil Patil + Group 3	

Class: `Ingredient`	
Responsibilities: Ingredients are a component of meals. Ingredients also contain a count to represent how many are in stock.	
Collaborators: Meal, Food	
Users: Recipe, Meal	**Used by:** ...
Author: Collin Cleary + Nikhil Patil + Group 3	

Class: `Meal`	
Responsibilities: A meal is a group of recipes, the resulting meal is a food which contains the nutrients of all of the recipes' results combined.	
Collaborators: Recipe, Food	
Users: User	**Used by:** ...
Author: Collin Cleary + Nikhil Patil + Group 3	

Class: `Recipe`	
Responsibilities: A recipe is a group of ingredients which contains the summation of all of the ingredient's nutrients.	
Collaborators: Ingredient, Food	
Users: Meal	**Used by:** ...
Author: Collin Cleary + Nikhil Patil + Group 3	

State, and history

Class: `StateManager`	

Responsibilities: Keeps track of previous states to allow undoing.	
Collaborators: ...	
Users: ...	**Used by:** ...
Author: Aidan Ruiz + Group 3	

Class: `State`	
Responsibilities: Keeps track of a single action performed by the user which can be undone.	
Collaborators: ...	
Users: ...	**Used by:** ...
Author: Aidan Ruiz + Group 3	

Class: `Day`	
Responsibilities: Keeps track of all of the actions which happened in a day. User weight, caloric intake, meals, and workouts.	
Collaborators: ...	
Users: ...	**Used by:** ...
Author: ...	

DateTime Serialization (for json)

Class: `LocalDateTimeSerializer`	
Responsibilities: Changes a DateTime object to format it into json.	
Collaborators: ...	
Users: ...	**Used by:** ...
Author: ...	

Class: `TimeManager`	
Responsibilities: Manages the current time, using a Java.util.Timer to measure the time. Which allows days to be shortened or lengthened.	

Collaborators: ...	
Users: ...	**Used by:** ...
Author: ...	

Class: `Crypto`	
Responsibilities: Given a string, it returns a hash of the string, used for storing passwords in persistent files.	
Collaborators: ...	
Users: ...	**Used by:** ...
Author: Aidan Ruiz + Group 3	

Class: `CLI`	
Responsibilities: Manages the current logged in user, and scans for incoming commands.	
Collaborators: CLIMenuOption, CLIState	
Users: ...	**Used by:** ...
Author: Aidan Ruiz + Group 3	

Class: `CLIMenuOption`	
Responsibilities: Defines a name, and runnable callback which is executed when the option is selected by the user through the CLI.	
Collaborators: ...	
Users: ...	**Used by:** ...
Author: Aidan Ruiz + Group 3	

Class: `CLIState`	
Responsibilities: Stores the state and options of the CLI, which returns the current input from the user that is authenticated.	
Collaborators: ...	

Users: CLIStateMainMenu, CLIStateMyProfile, CLIStateMyTeam, CLIStateSearchableTable	**Used by:** ...
Author: Aidan Ruiz + Group 3	

Class: `CLIMainMenuState`	
Responsibilities: Defines all the commands which can be done from the “main menu” state.	
Collaborators: ...	
Users: ...	**Used by:** ...
Author: Aidan Ruiz + Group 3	

Class: `CLIStateMyProfile`	
Responsibilities: Defines all the commands which can be done from the “my profile” state.	
Collaborators: ...	
Users: ...	**Used by:** ...
Author: Aidan Ruiz + Group 3	

Class: `CLIStateMyTeam`	
Responsibilities: Defines the commands which can be done from the “my team” state.	
Collaborators: ...	
Author: Aidan Ruiz + Group 3	**Used by:** ...
Author: ...	

Class: `CLIStateSearchableTable`	
Responsibilities: Defines the commands which can be done to search through a table,	

ex) ingredients.	
Collaborators: ...	
Users: ...	**Used by:** ...
Author: Aidan Ruiz + Group 3	

Class: `CLIStateCreateFood`	
Responsibilities: Defines the commands for creating food items, specifically recipes and meals.	
Collaborators: ...	
Users: ...	**Used by:** ...
Author: Aidan Ruiz + Group 3	

Class: `CLIStateViewHistory`	
Responsibilities: Defines the commands for browsing user history.	
Collaborators: ...	
Users: ...	**Used by:** ...
Author: Aidan Ruiz + Group 3	

Class: `JSONReader`	
Responsibilities: Parses a JSON file given a key. Returns all the values. Used in exporting the history database.	
Collaborators: ...	
Users: User	**Used by:** Backend
Author: Nikhil Patil + Group 3	

Class: `JSONFileWriter`	
Responsibilities: Parses a JSON file given a key. Returns all the values. Used in exporting the history database.	

Collaborators: ...	
Users: Nutriapp	**Used by:** Backend
Author: Nikhil Patil + Group 3	