<NutriApp 2.0>

Design Documentation

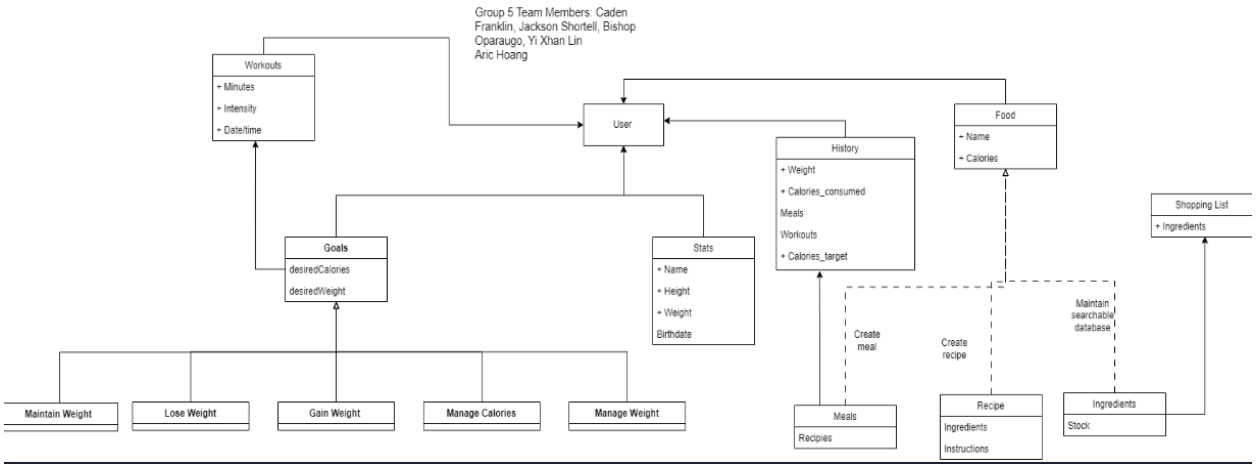
Prepared by CTN TEAM 5

- Ash Franklin
- Bishop Oparaugo
- Jackson Shortell
- Yi Xhan LIn
- Aric Hoang

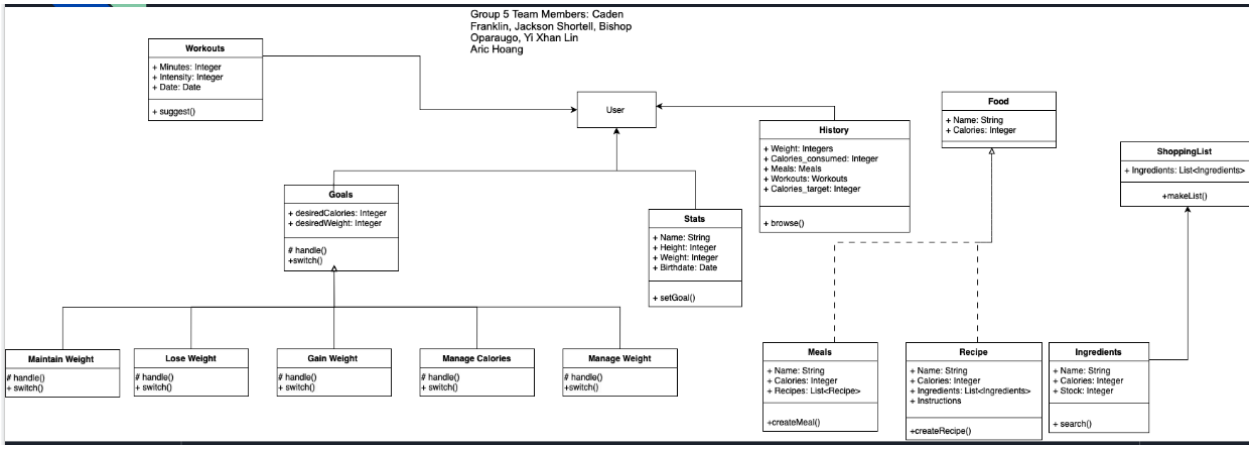
Summary

As a software developing team, we have been tasked with creating an app “NutriAPP“ that is used to track nutrition. The application includes features for allowing the user to input their information (statistics), choose a goal, and keep track of their development. It will also keep a record of the various forms of foods and properties included in the user’s diet, maintain a searchable database of ingredients, provide appropriate workout routines, and adapt to handle changes made to the user’s input. Additional functionality such as the shopping cart interface and treelike structure of the recipes, ingredients, and meal classes should also be implemented.

Domain Model



System architecture



Subsystem

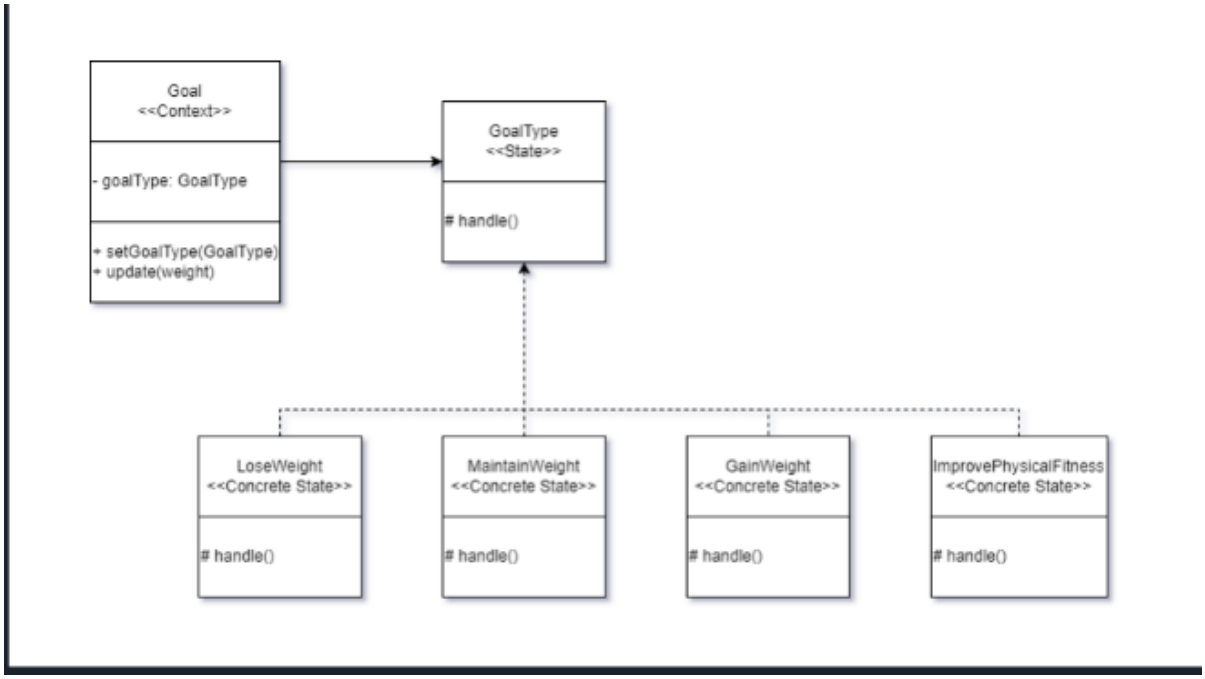
This section provides detailed design for specific subsystems described in the system architecture.

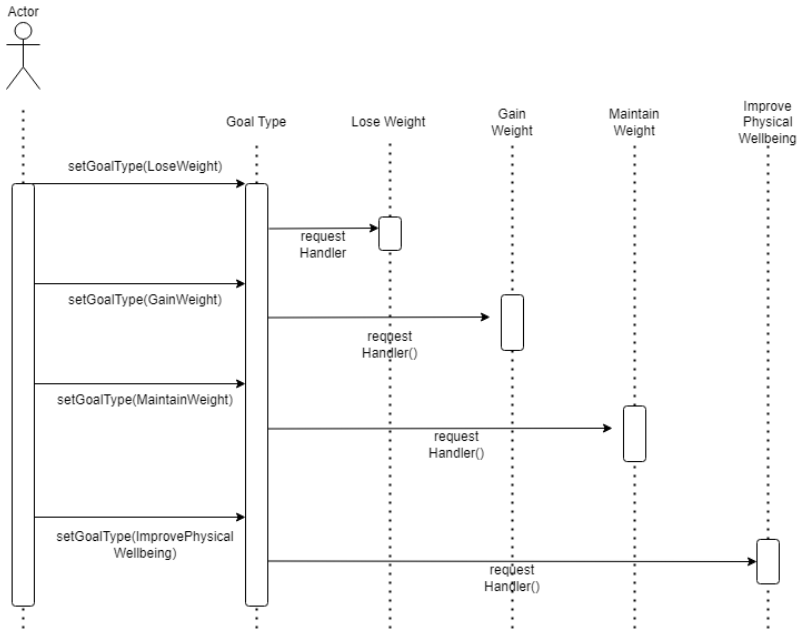
Name of the subsystem

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.

STATE DESIGN PATTERN

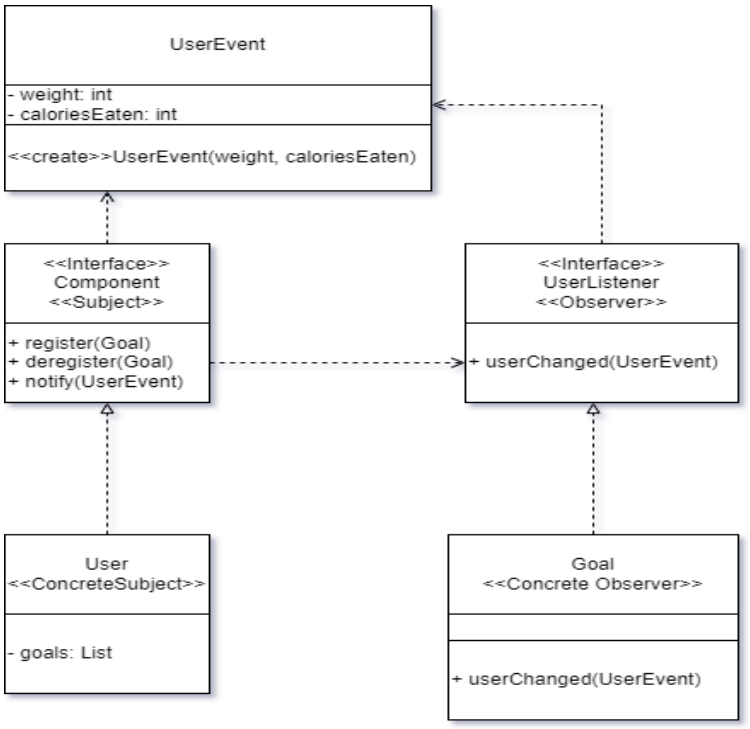


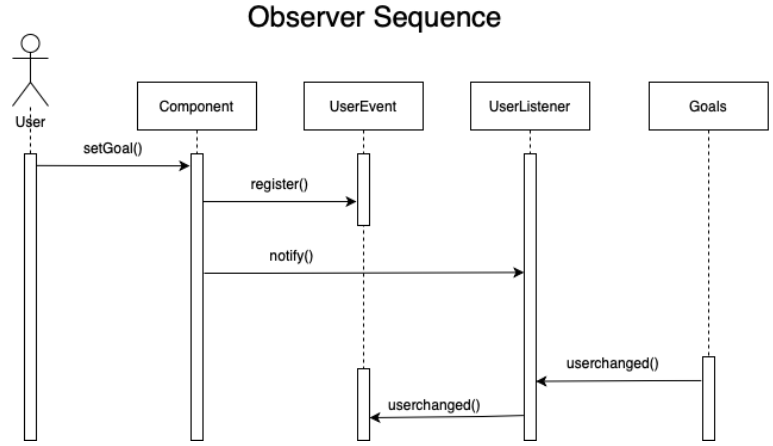


Generic GoF Design Pattern Name: State Design Pattern	
Pattern Name in terms of system context: SavedHistory	
List one per line the participants from the GoF class structure (add rows, if needed)	Nouns from your table that take on that participant role. If there is no noun for a participant role explain why.
State	History
State Context	Goal Type
ConcreteStateA	Lose Weight
ConcreteStateB	Gain Weight
ConcreteStateC	Maintain Weight

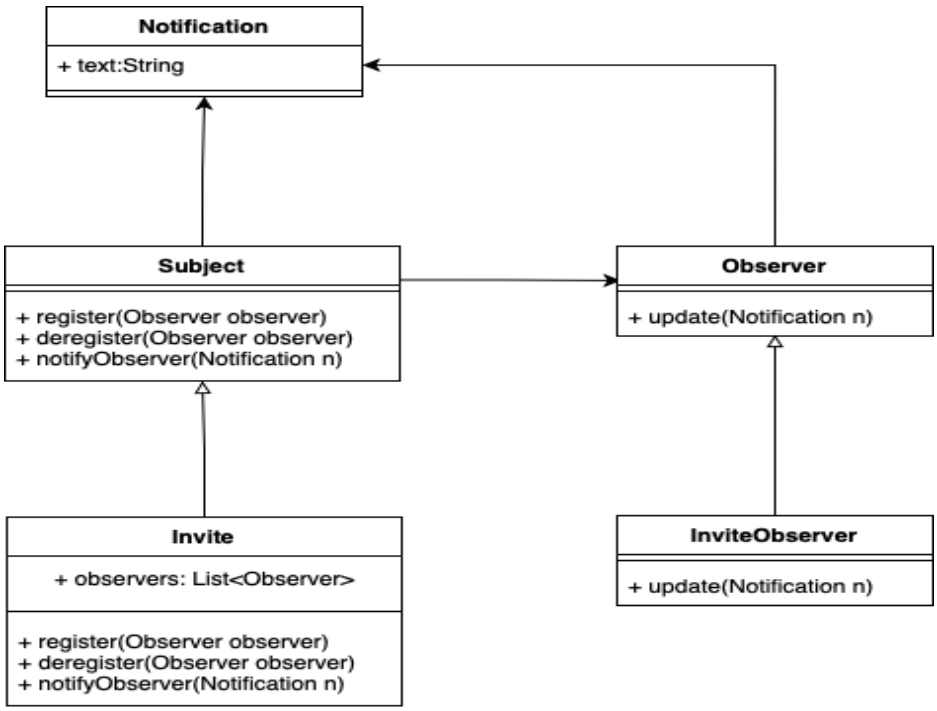
ConcreteStateD	Improve Physical Wellbeing
Short (several sentences) narrative description of how the design pattern is being used in this application. Should be tied to system description and requirements. The State Design Pattern is used to set the state of the goals between the different goals. After setting a state, the system will then implement the strategy design, which actually creates things in the goal. If something is automatically detected about a person's weight, the state will automatically change depending on the weight of the user. Also in the UserStats class, the user inputs a height and weight which directly influences the latter concrete classes.	

OBSERVER DESIGN PATTERN



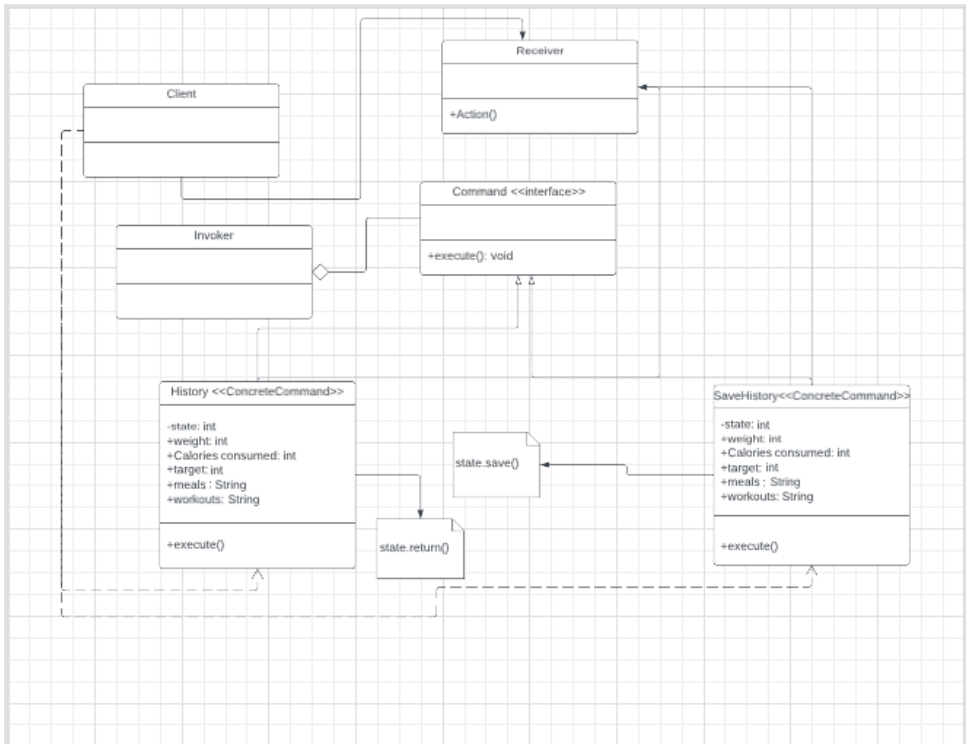


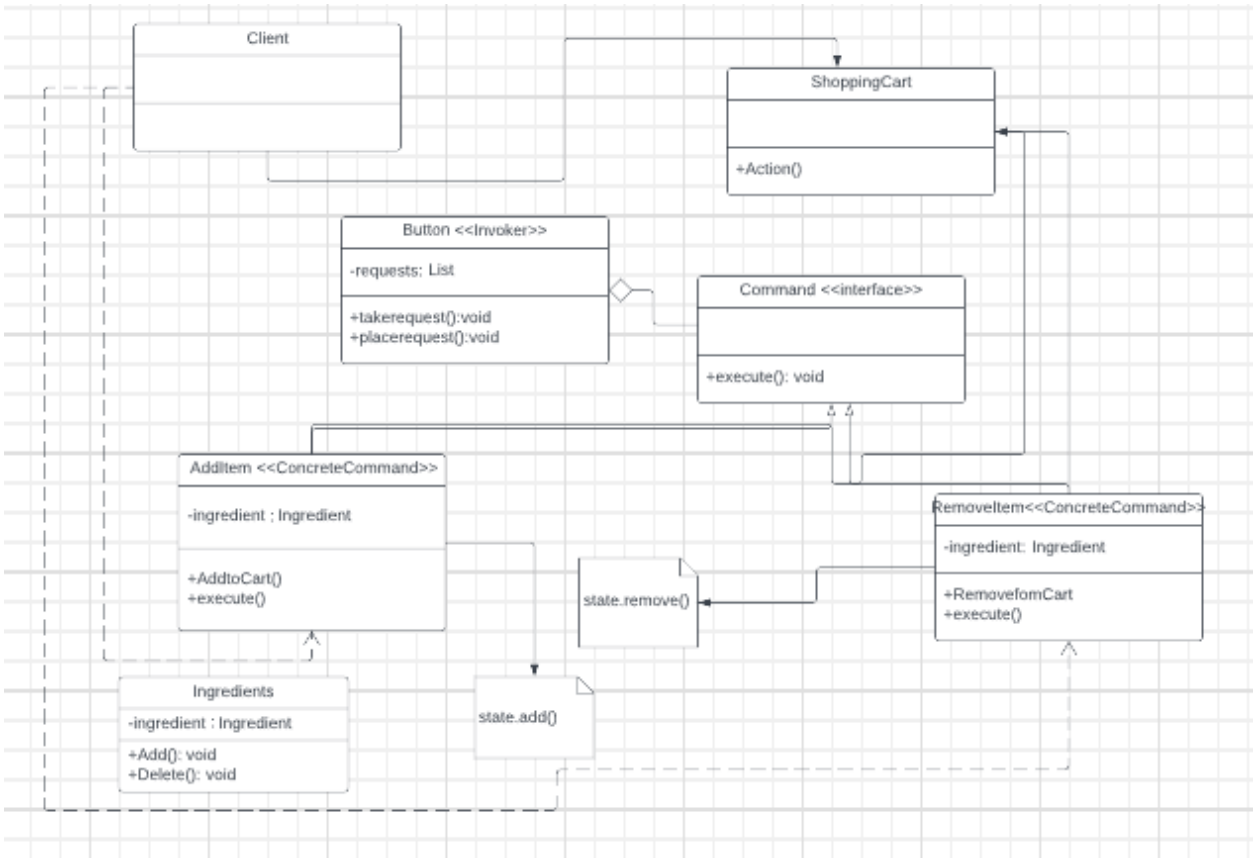
Generic GoF Design Pattern Name: Observer Pattern	
Pattern Name in terms of system context: updateGoal	
List one per line the participants from the GoF class structure (add rows, if needed)	Nouns from your table that take on that participant role. If there is no noun for a participant role explain why.
Subject	Goal
Interface	notifyObservers()
ConcreteObserverA	Food[meals, recipes, and ingredients]
ConcreteObserverB	Workouts
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 Observer Pattern will be used in the design of the NUTRIX app to notify its list of dependents of any internal changes made to the userGoal class object. Notified objects will then modify their behaviors based on the new state/behavior of the Object class “userGoal”.	

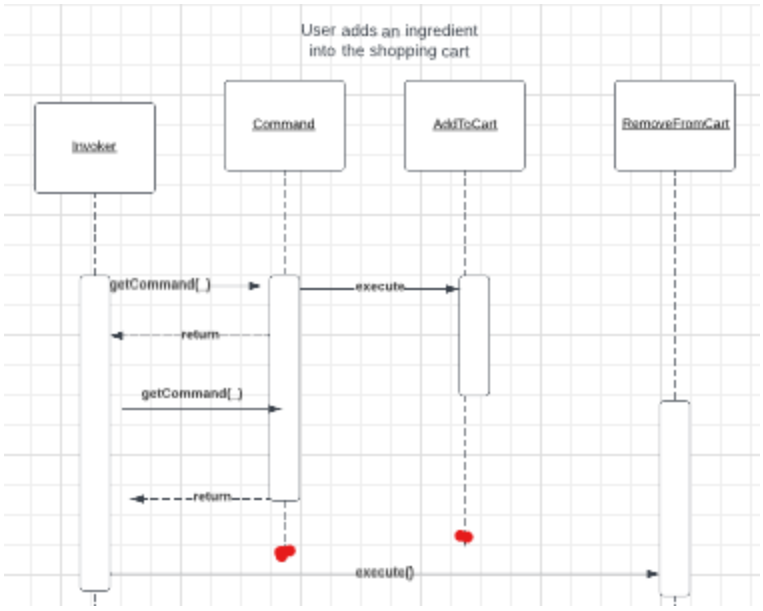


COMMAND DESIGN PATTERN

History Instance:

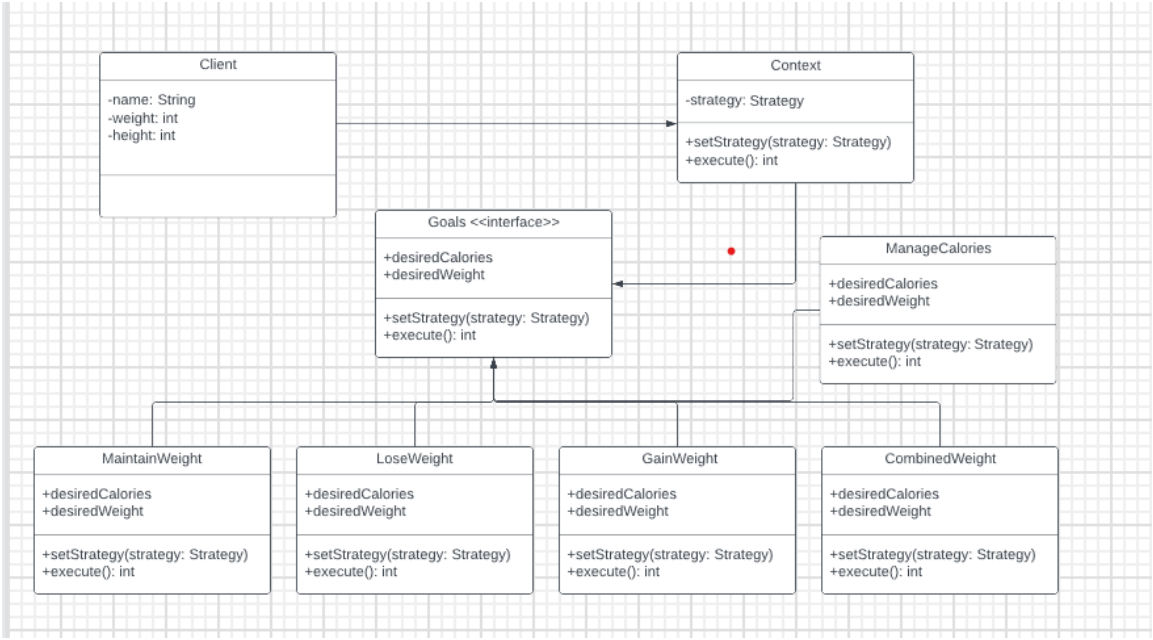

Receiver
+Action()
Client
Command <<interface>>
+execute(): void
Invoker
History <<ConcreteCommand>>
-state: int
+weight: int
+Calories consumed: int
+target: int
+meals : String
+workouts: String
+execute()
state.save()
state.return()
SaveHistory<<ConcreteCommand>>
-state: int
+weight: int
+Calories consumed: int
+target: int
+meals : String
+workouts: String
+execute()

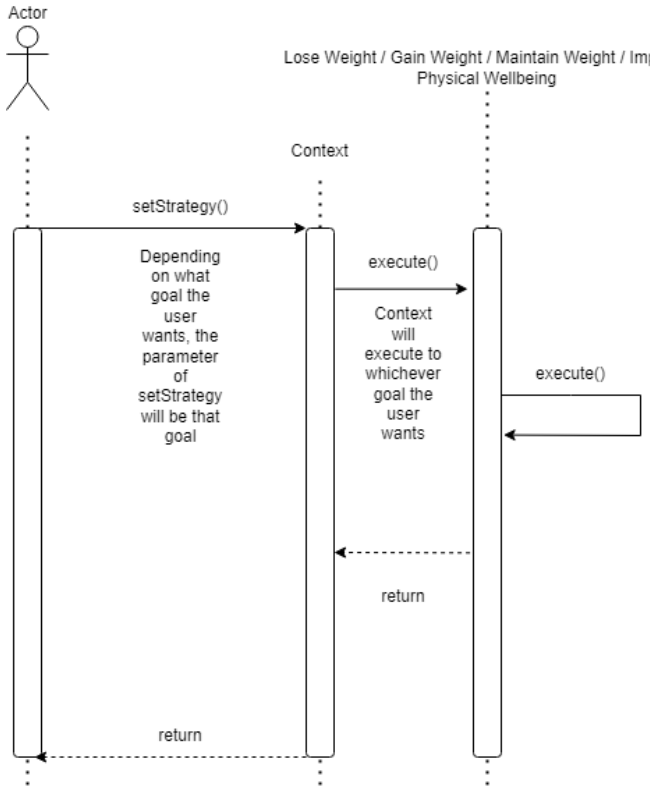

Client
ShoppingCart
+Action()
Button <<Invoker>>
-requests: List
+takerequest():void
+placerequest():void
Command <<interface>>
+execute(): void
AddItem <<ConcreteCommand>>
-ingredient : Ingredient
+AddtoCart()
+execute()
RemoveItem<<ConcreteCommand>>
-ingredient: Ingredient
+RemovefomCart
+execute()
state.remove()
Ingredients
-ingredient : Ingredient
+Add(): void
+Delete(): void
state.add()



Generic GoF Design Pattern Name: Command Design Pattern	
Pattern Name in terms of system context: Shopping List	
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.
Receiver	Shopping List
Invoker	Button
ConcreteCommand	Checkout (This is a verb not a noun, an invoker activities this)
ConcreteCommand	Remove From Cart (This is a verb not a noun, an invoker activities this)
ConcreteCommand	Add To Cart (This is a verb not a noun, an invoker activities this)
Invoker	Ingredients
Short (several sentences) narrative description of how the design pattern is being used in this application. Should be tied to system description and requirements. Include the idea of how you can add/remove items from the list and also store information of code The idea would be that everytime an ingredient is selected, there would be a widget that executes a command to store the ingredient somewhere. Adding to cart would take that active ingredient and add it into the cart. From there, you can view your cart and use more widgets to interact with it. You remove items from your cart, or you can checkout and buy them. All of these functions will take commands and widgets. The widgets act like invokers that initiate the changes in the shopping cart. The client and receiver would be the shopping list because it will constantly receive data and will act as the main hub for most of the shopping cart commands.	

STRATEGY DESIGN PATTERN

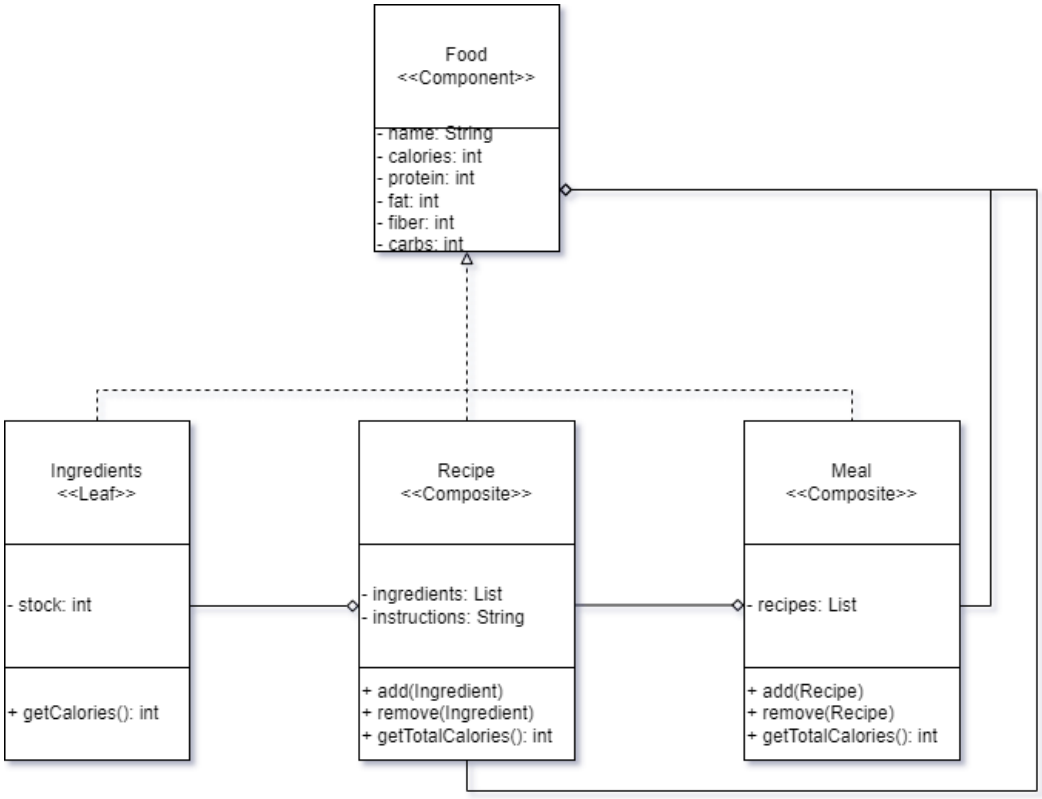




Generic GoF Design Pattern Name: Strategy Design Pattern	
Pattern Name in terms of system context: userGoal	
List one per line the participants from the GoF class structure (add rows, if needed)	Nouns from your table that take on that participant role. If there is no noun for a participant role explain why.

Strategy Interface	Goal (selecting the current goal state desired by the user)
Context	UserInput
ConcreteStrategyA	LoseWeight
ConcreteStrategyB	GainWeight
ConcreteStrategyC	MaintainWeight
ConcreteStrategyD	Combined
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 Strategy design pattern could be used in this application to determine the differences in the goal the user wants to achieve. Currently, the major differences in goals for the Nutrix app are losing, gaining, and maintaining weight. Each of these separate goals can be represented as a separate strategy. All having one behavior similar to the other listed possibilities. Incorporating this design pattern will help reduce list conditionals, duplicate code, and help preserve the open close design principle with its drawback only being an increased amount of class objects.	

COMPOSITE DESIGN PATTERN

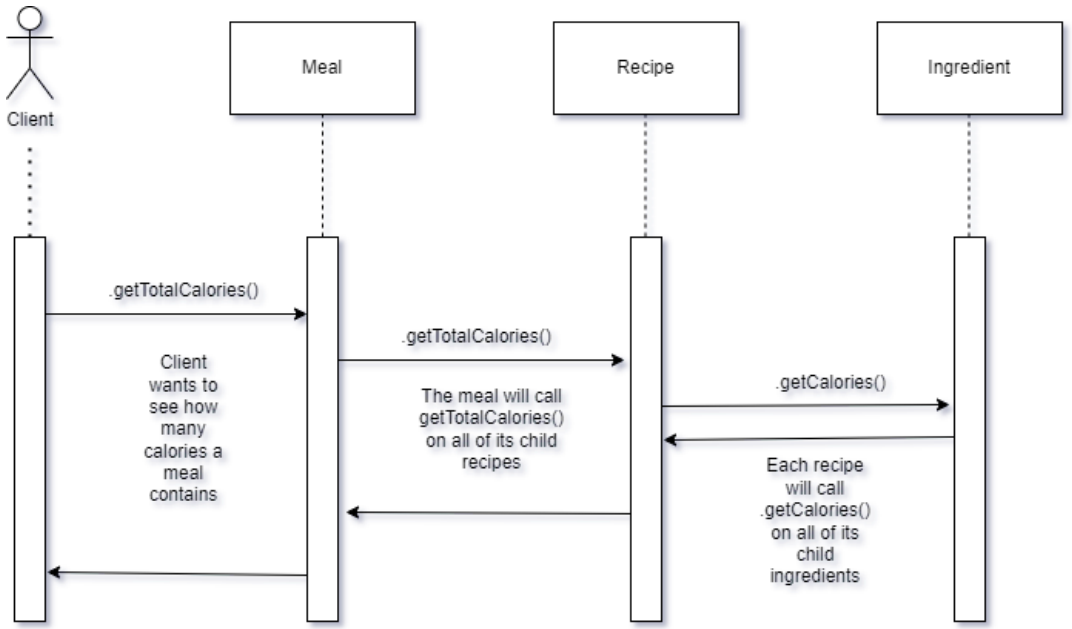


Generic GoF Design Pattern Name: Composite Design Pattern	
Pattern Name in terms of system context: Recipes and meals organization	
List one per line the participants from the GoF class structure (add rows, if needed)	Nouns from your table that take on that participant role. If there is no noun for a participant role explain why.
User	Client
Component	Food

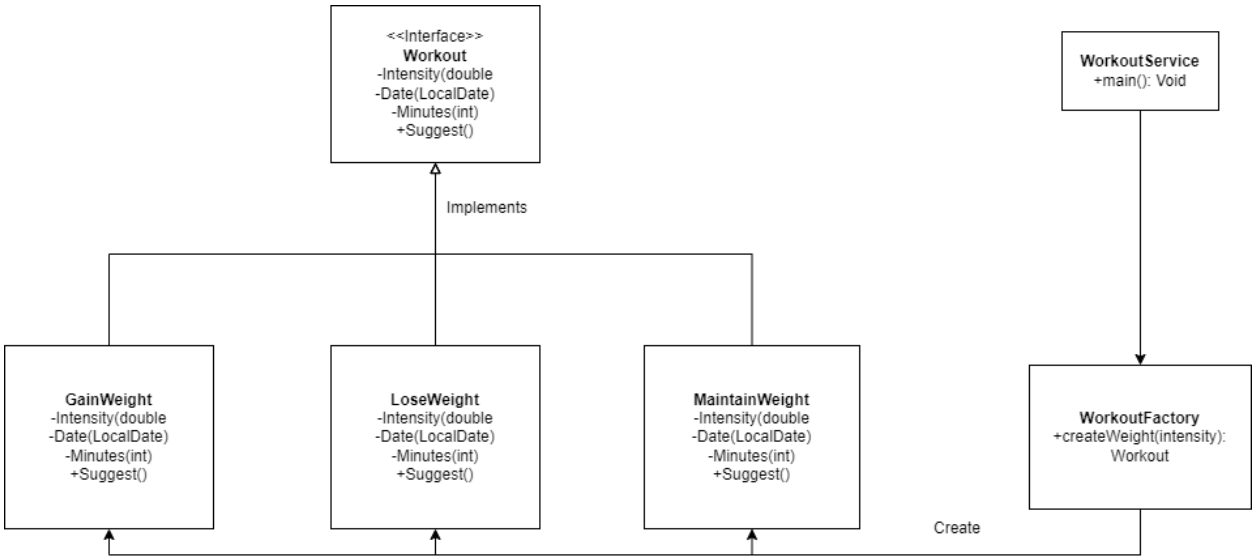
Composite	Recipes
Composite	Meals
Leaf	Ingredients

Short (several sentences) narrative description of how the design pattern is being used in this application. Should be tied to system description and requirements.

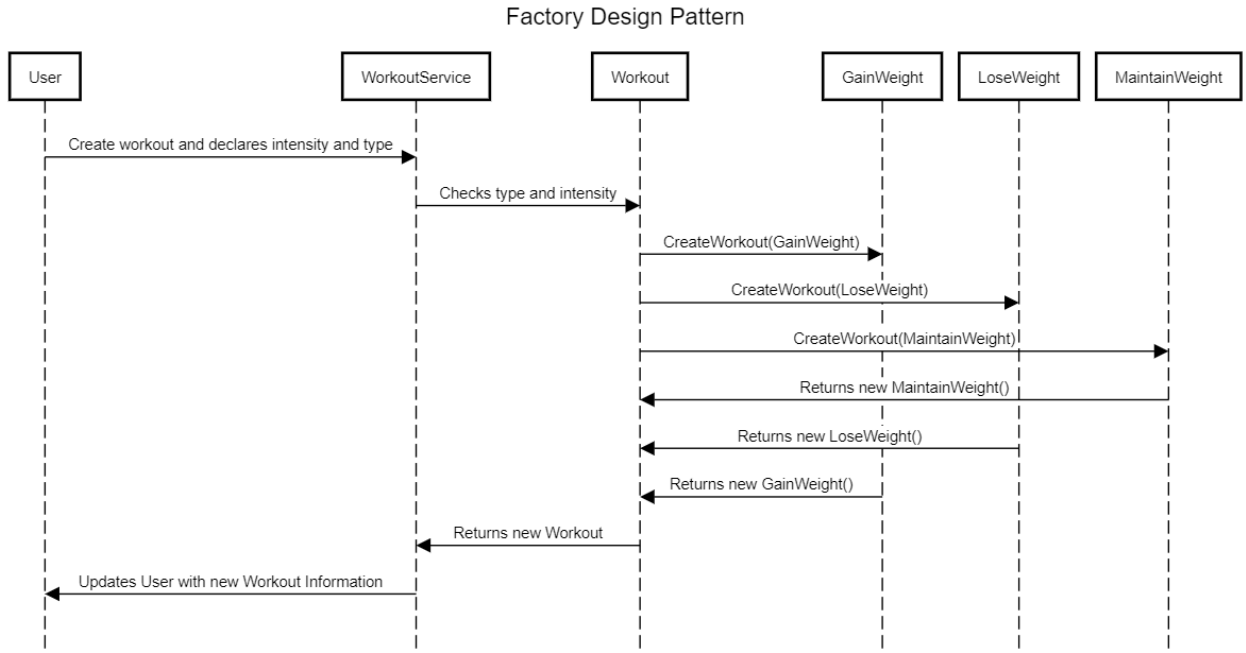
Recipes, meals, and ingredients are all foods but are each different in terms of what they are and how they should be treated and created. Since ingredients are a stand alone food while recipes are a collection of ingredients and meals are a collection of recipes we saw how each of these things would fit nicely into the composite design pattern. This design pattern will be useful to achieve some of the requirements in the design problem such as when a user prepares a meal the app knows how many calories to subtract from their daily total. To access the caloric information of a meal you would need to access all of the recipes within that meal and then all of the ingredients in each recipe. With this design pattern we would easily be able to access all of that information and help users easily keep track of their daily total.



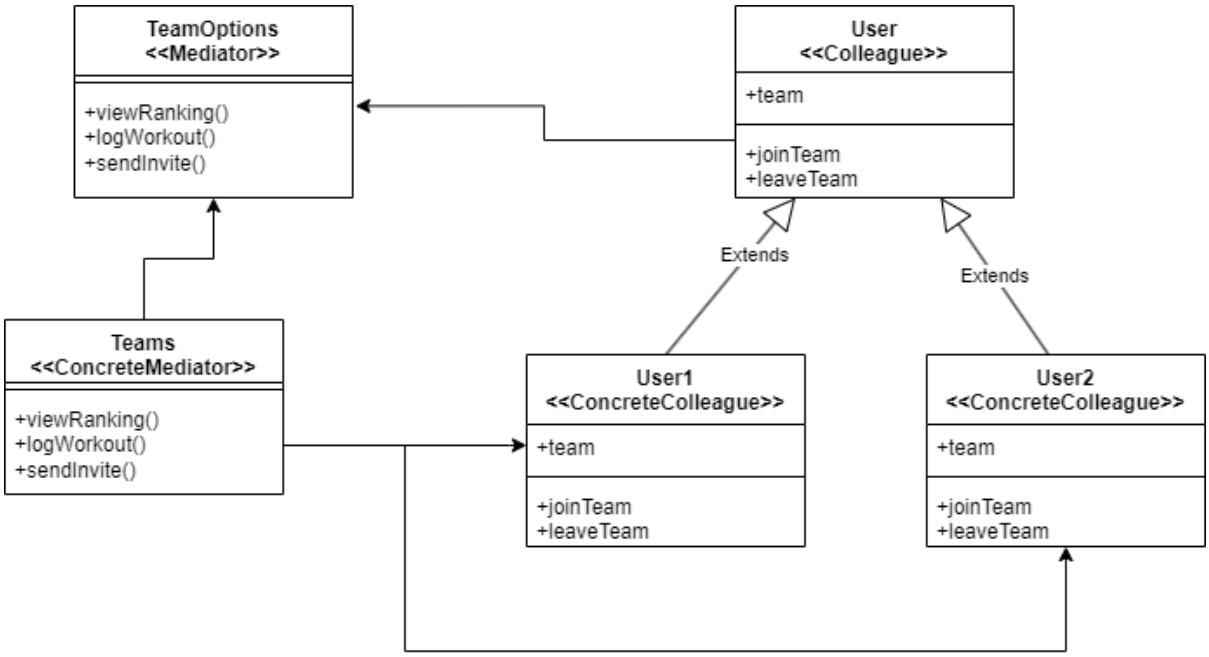
Factory Design Pattern



Interface	Workout
Service	WorkoutService
Factory	WorkoutFactory
Workout	Gain, Lose, Maintain Weight
There is an interface for workout, each type of workout has a compositor with a parameter of intensity. Then the factory creates the workout and the intensity is automatically adjusted based on giving it (low, mid, high) Calories and minutes are also calculated based off intensity.	



Mediator Design Pattern



Generic GoF Design Pattern Name: MediatorDesign Pattern	
Pattern Name in terms of system context: Communication between different users	
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.

Mediator	TeamOptions
ConcreteMediator	Teams
Colleague	User
ConcreteColleague	Not a defined class, but instance of User class
ConcreteColleague	Not a defined class, but instance of User

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 mediator here is being used to reduce the amount of relations users have between each other. Rather, they will communicate with each other mainly through the mediator class (through their team in this case). This will help them better communicate about challenges and the workouts of each other without having to constantly send information between the 2 user classes.

Status of Implementation

As of now the project is finished with the design phase. From the requirements several patterns were identified and subsystems were designed. Implementation is starting soon.

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	User
RESPONSIBILITIES	Knows, workouts Knows Goals Knows Stats Knows History
COLLABORATORS	History Stats Goals Workouts

USERS	Used By: User
AUTHORS	CTN

CLASS	Workout
RESPONSIBILITIES	Knows minutes Knows intensity Knows date
COLLABORATORS	user
USERS	Used By: User
AUTHORS	CTN

CLASS	Stats
RESPONSIBILITIES	Knows Name Knows Height Knows Weight Knows Birthdate
COLLABORATORS	User
USERS	Used By: User
AUTHORS	CTN

CLASS	History
RESPONSIBILITIES	Knows weight Knows Calories_consumed Knows Meals Knows Workouts Knows Calories_target
COLLABORATORS	User Food

USERS	Used By: User
AUTHORS	CTN

CLASS	Goals
RESPONSIBILITIES	Knows desiredCalories Knows desiredWeight Knows maintain weight
COLLABORATORS	History Stats Goals Workouts
USERS	Used By: User
AUTHORS	CTN

CLASS	Meals
RESPONSIBILITIES	Knows Recipes Create meals
COLLABORATORS	Food
USERS	Used By: User
AUTHORS	CTN

CLASS	Recipes
RESPONSIBILITIES	Knows Ingredients Create recipes
COLLABORATORS	Food
USERS	Used By: User
AUTHORS	CTN

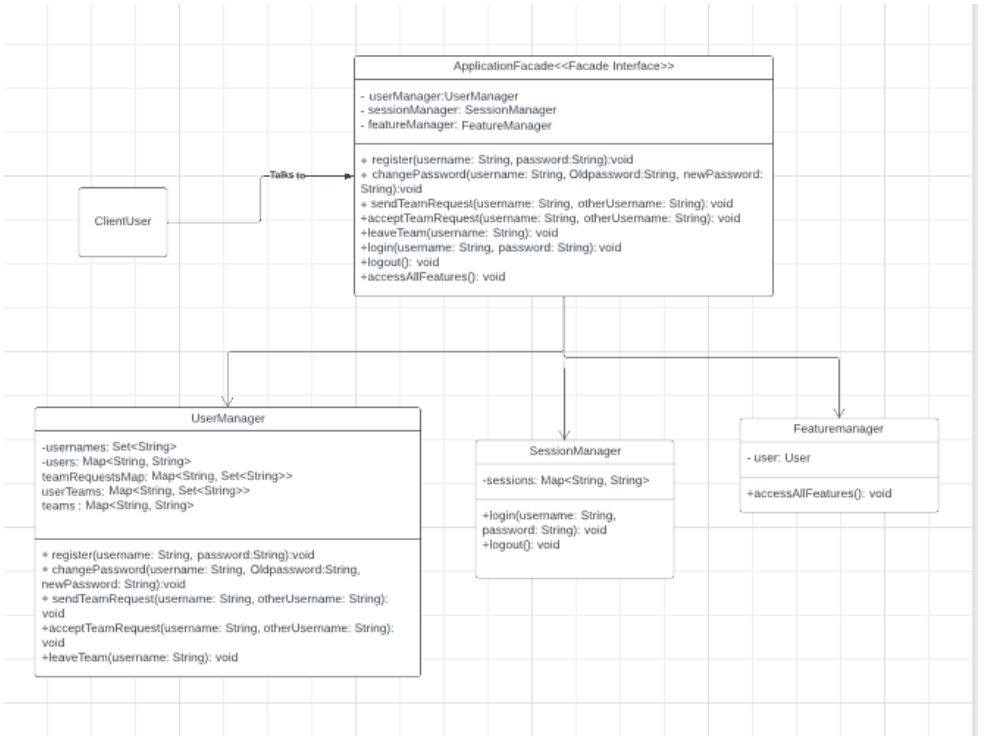
CLASS	Ingredients
RESPONSIBILITIES	Knows calories, carbs, fats, and proteins
COLLABORATORS	Shopping List
USERS	Used By: User
AUTHORS	CTN

CLASS	Shopping List
RESPONSIBILITIES	Search ingredients
COLLABORATORS	Shopping List
USERS	Used By: User
AUTHORS	CTN

USER ACCOUNT HANDLER (FACADEOPS)

For this feature's subsystem, we decided to go about using the facade because it allows our users with a much simpler and easy to use interface in our complicated Nutrix Application. It allows for a separation of complexities with managers that spearhead the handling of its unique features. By making use of the facade, we are ultimately able to improve the system's maintainability, and consistency over complex and complicated range of features.

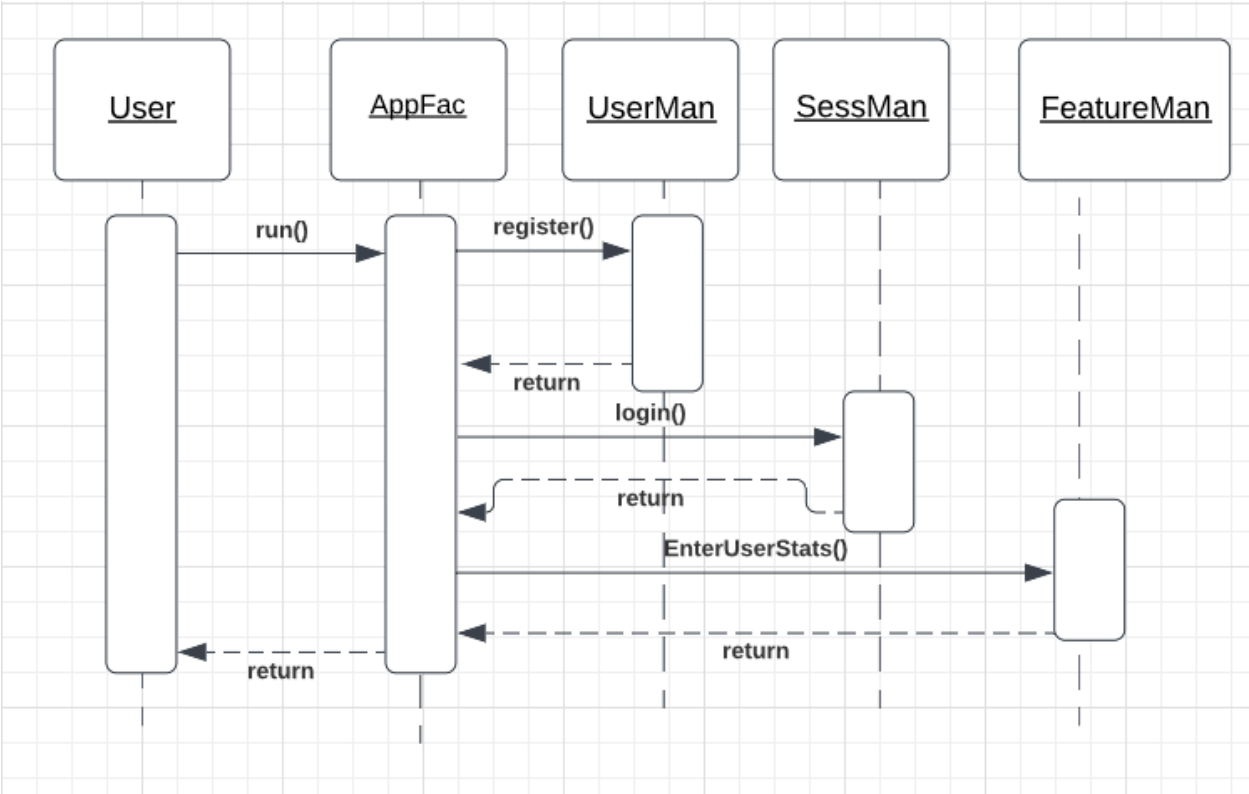
UML DiAGRAM AND CRC CARD (FACADE)



Subsystem Name: FacadeOps		GoF Pattern: Facade
Participants: User, etc.		
Class	Pattern Stereotype	Participant's contribution in the context of the application
ApplicationFacade	Facade interface	This class will manage the user features provided, user sessions with a session key, and ultimately handle any and all account operations. Acts as th interface for all other subsytems included in it's implementation.
UserManager	Complex Subsystem	This will manage user data, authentication, and user to teams operations regarding the acceptance and sending of requests.
SessionManager	Complex Subsystem	This will be in charge of managing session with a generated session key for the user after they log in and be deleted after log out.
FeatureManager	Complex Subsystem	• Manage and provide access to the features stated in R1 Requirements.

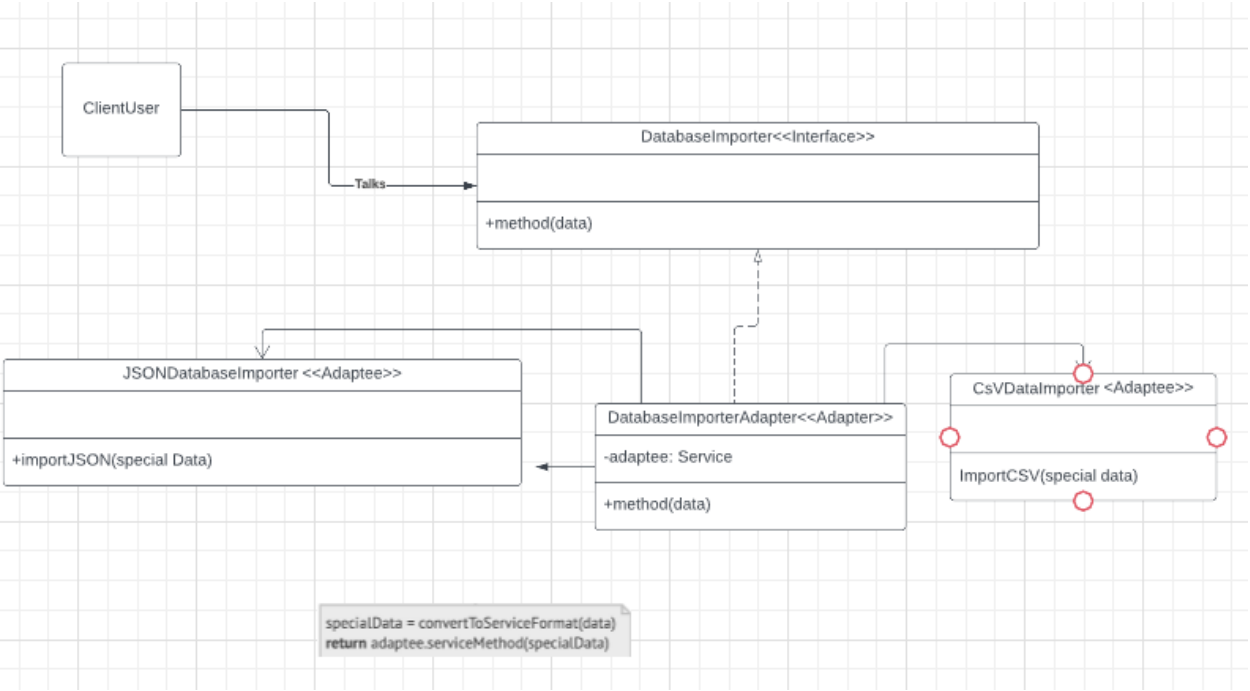
Deviations from the standard pattern: Not many deviations from the original design pattern model, in the sense of the need for multiple facade classes to handle specific subsystem.
Requirements being covered: support multiple registered users, register with a username and password, and upon verification will allows user access to other features in which ever Manager class.

Sequence Diagram for Facade



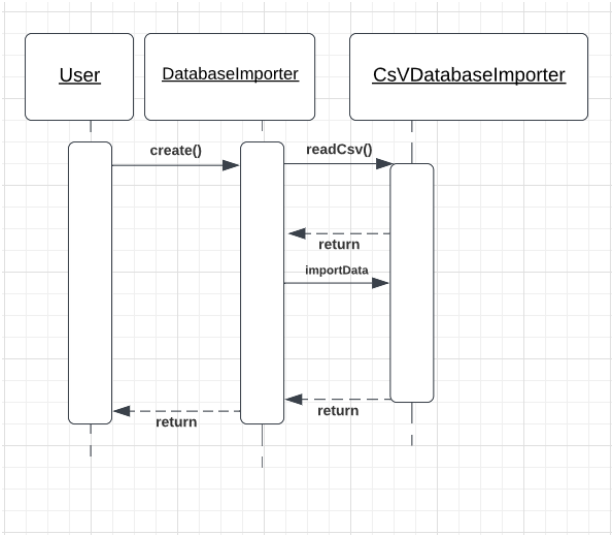
DATABASE (ADAPTER)

For the Database Importation/Exportation of the system, we choose to use the Adapter Design pattern. By Using the Adaptor pattern we are able to handle user requests regardless of the chosen format whether json or csv or any file try the Adapter will work with it created adaptee to handle those client requests. Overall, using this pattern allows the team to incorporate the importation and exportation of data within the system consistently, and easily based on the provided context of the request.



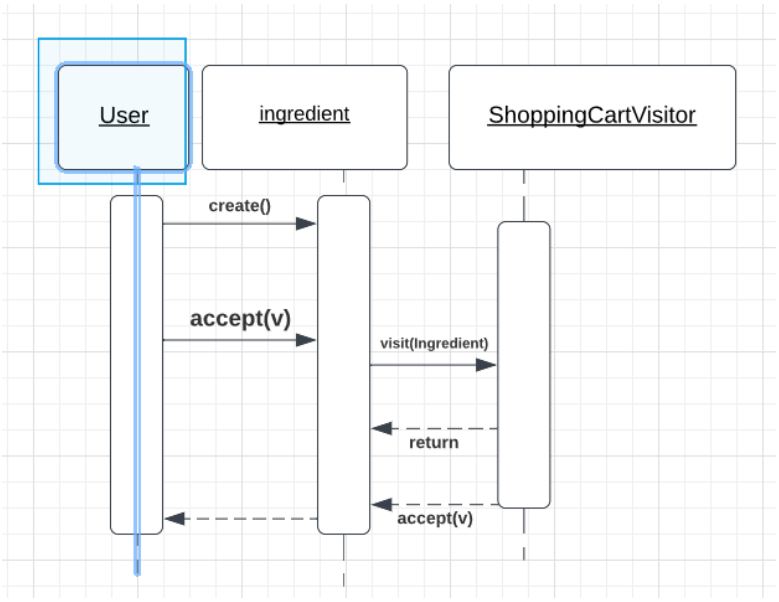
Subsystem Name: Database		GoF Pattern: Adapter
Participants: food, facadeOps		
Class	Pattern Stereotype	Participant's contribution in the context of the application
DatabaseImporter	interface	• Describes the precedent that other Adaptor classes must follow in order to communicate with the client. Defines an interface in the system. Responsible for stating the importData function as well as the other unimplemented methods that are implemented by its DatabaseImporterAdapter.
DatabaseImporterAdapter	Adapter	This will convert the data extracted from its database importer into the format that the client requested for. It will also adapt the interface to what's expected and easily accessible for the user.

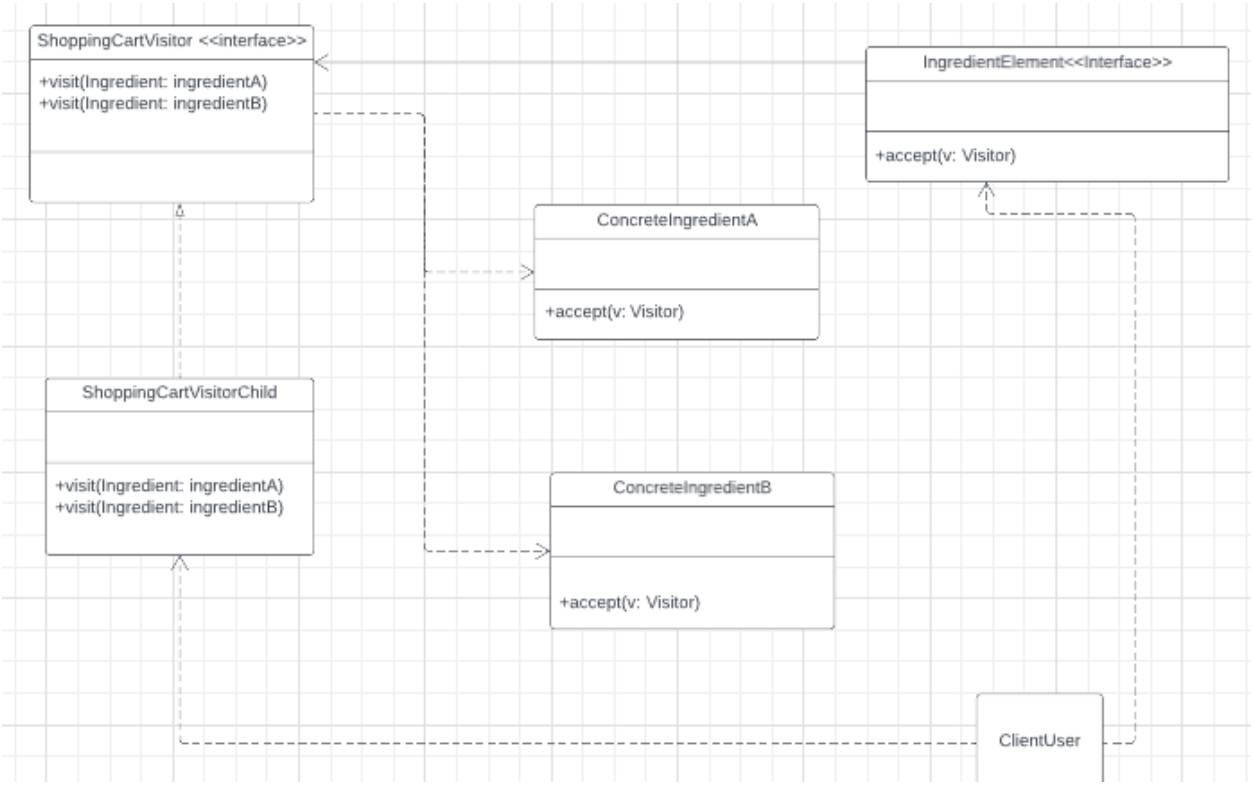
CSVDatabaseImporter	Adaptee	• Manage user session data • Imports data from a file • Converts the database into a csv file
JSONDatabaseImporter	Adaptee	• Imports data from a file given by its Adaptee • Converts the database into a json file.
Deviations from the standard pattern: Not many deviations from the original design pattern model, but in the application only one Adapter was made for multiple (2) Adaptees.		
Requirements being covered: Extract data from a random file and import the data into the format appropriated by the Adaptee to handle client request.		



ShoppingCart (Visitor)

For handling the shopping Cart feature we decided on the Visitor design pattern. The user will be able to visit multiple items/ingredient's for sale in the cart and making use of this pattern allows for future maintenance and development of other elements that will interact with that user cart. Overall, using this pattern allows the user to explore through various ingredient and access their details.


User
ingredient
ShoppingCartVisitor
create()
accept(v)
visit(Ingredient)
return
accept(v)


ShoppingCartVisitor <<interface>>
+visit(Ingredient: ingredientA)
+visit(Ingredient: ingredientB)
IngredientElement<<Interface>>
+accept(v: Visitor)
ConcreteIngredientA
+accept(v: Visitor)
ShoppingCartVisitorChild
+visit(Ingredient: ingredientA)
+visit(Ingredient: ingredientB)
ConcreteIngredientB
+accept(v: Visitor)
ClientUser