Nutriapp™ R2

Design Documentation

Prepared by Aaron Santos, Himel Uddin, Abiel Yemane, Laxmi Poudel

Summary **3**

Domain Model **4**

System Architecture **5**

Rationale 5

View Controller Model Diagram 6

Subsystem Relationship Model 6

Subsystems **7**

Commands **7**

Rationale 7

Class Structure Diagram 8

Sequence Diagram 8

GoF for Command 9

User Authentication 11

Rationale 11

Class Structure Diagram 12

Sequence Diagram 13

GoF for Proxy Pattern 14

Database **15**

Rationale 15

Class Structure Diagram 16

Sequence Diagram 17

GoF for Adapter 18

Undo 20

Rationale 20

Class Structure Diagram 21
Sequence Diagram 22
GoF for Undo 22
Teams Subsystem **24**
Rationale 24
Class Structure Diagram 25
Sequence Diagram 25
GoF for Teams 26
Appendix **27**

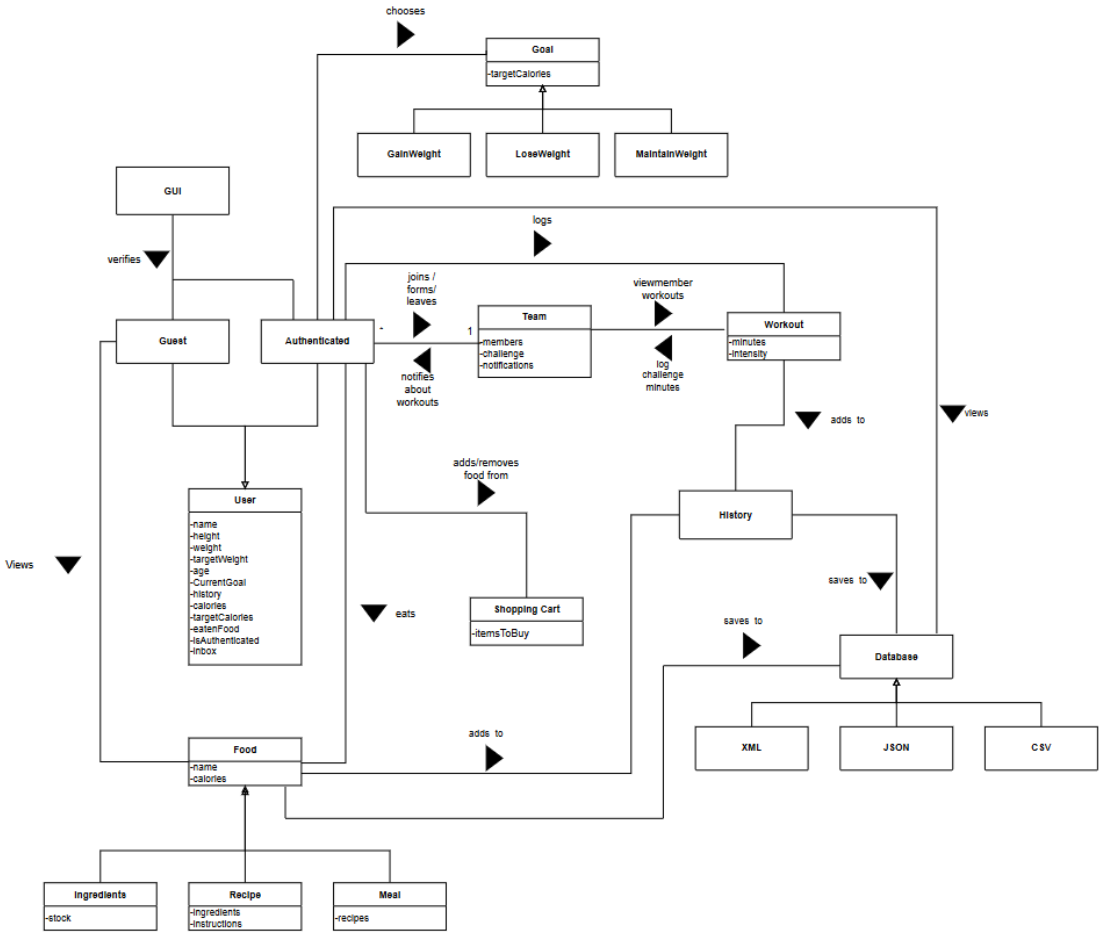
Summary

The NutriApp™ application went through its first iteration in the form of NutriApp R1. With the introduction of NutriApp R2, new sets of functional (and non-functional) requirements were introduced to our already existing system. To tackle these sets of requirements, the following subsystems were established: an update to Commands, User Authentication, Database, Undo, and Teams subsystems. Each of these subsystems play a crucial role in the overall system and in achieving the requirements of NutriApp R2, and each of these subsystems, one way or another, utilize one or more of the design patterns. With the patterns that were applied throughout the system, and onto already existing patterns from the past iteration, advantages and trade-offs needed to be weighed; this was all done to prioritize the requirements, and design. Such trade-offs that came with the application of the design patterns, and were deemed acceptable due to the advantages they brought in regards to meeting the requirements of NutriApp R2 and toward building a system that is designed well.

The respective subsystems cover their own unique advantages and disadvantages that they brought to the overall design but general design principles seen in the overall design can be noted. Many new classes were introduced that dealt with adhering to one specific role in the system, showing strong relations to the Single Responsibility Principle. Cohesion was also increased throughout the system as all the design patterns introduced many classes that were smaller in size and with more narrowly defined responsibilities. However, this increase in cohesion increased the coupling within the system due to all the new relationships that these smaller classes come with.

Not only did the addition of our new classes bring about more coupling to the system, so did the redesign of our previous subsystems, this however was needed as redesigning was an essential part for meeting the necessary requirements set by v2.0. This goes more in depth within our Commands and Undo Subsystem as both work on modifying and adding additional functionalities to the already existing subsystem. As for the other subsystems, there are crucial additions to the User object and many of them utilize it one way or another; be it having Teams of users or authenticating them when logging in.

Domain Model



System Architecture

Rationale

The five subsystems of R2 are Commands, User Authentication, Teams, Undoing, and Databases.

The Command subsystem from R1 is being updated to add commands from R2. Commands is the controller in between the GUI and the model.

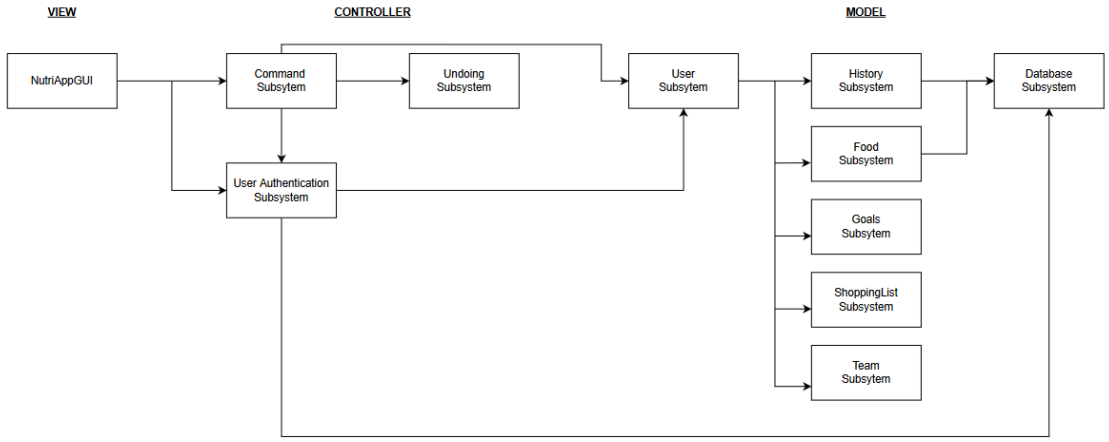
The User Authentication subsystem manages the authentication requirements, creating instances of users within the system and creating new users with new information. It creates a guest user if login information cannot be provided, and creates an authenticated user if a valid username and password combination is given. Login information and history, which is used to get the logged-in user's latest data (height, weight, etc) are stored in Databases.

The Teams subsystem manages the team requirements. It connects to Command because commands are sent to invite users to join teams.

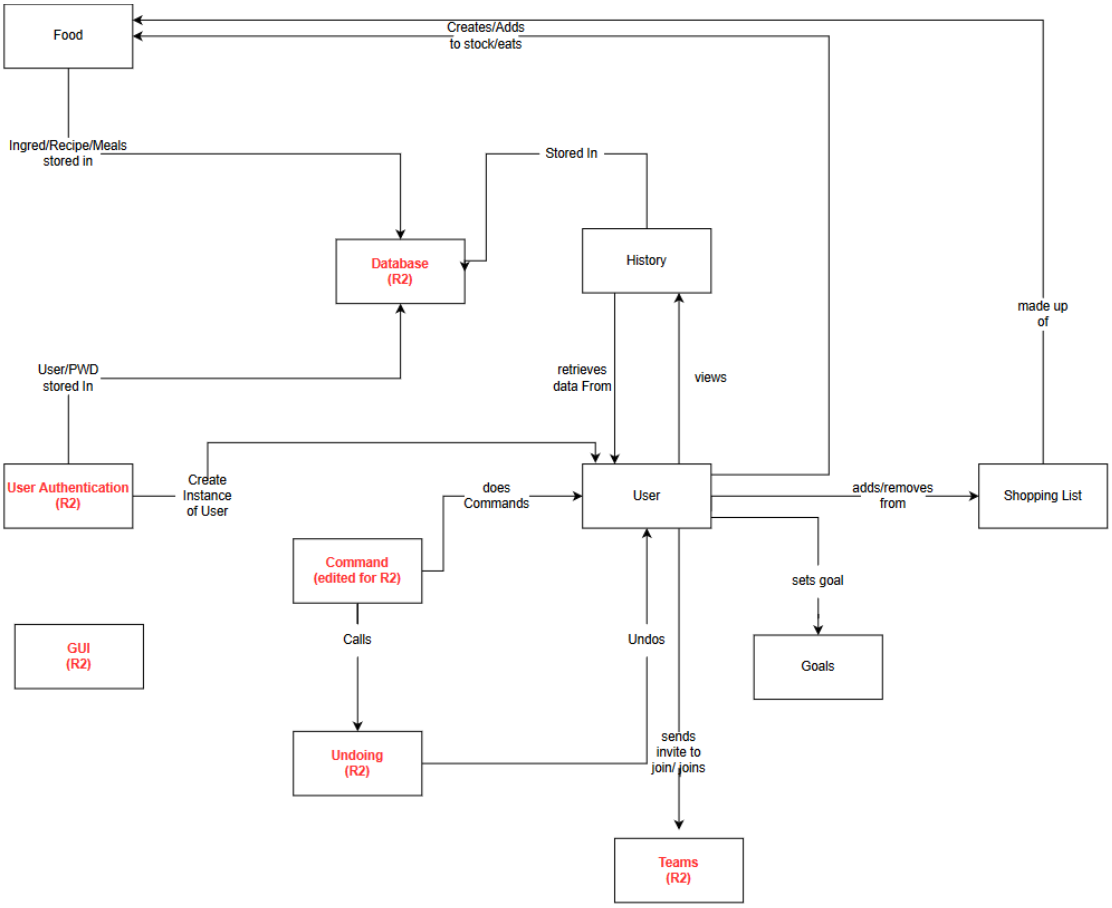
The Undoing Subsystem manages the undoing requirements. It connects to the user as it undoes information within the user, and it connects to command because it undoes commands.

The Database subsystem manages database requirements. It connects to User Authentication because login information is stored in the database, it connects to History because historical information is stored in the database, and it connects to Food because the three food types are stored in the database.

View Controller Model Diagram



Subsystem Relationship Model



Subsystems

Commands

Rationale

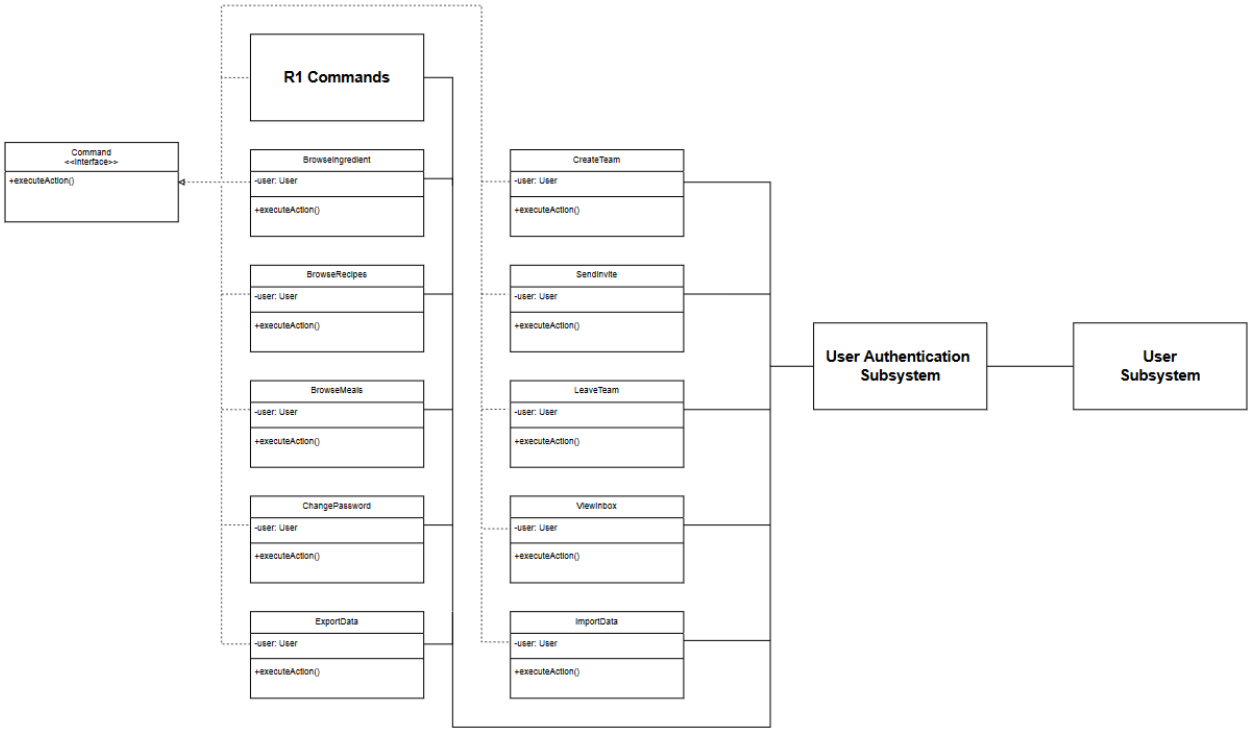
The *Command* subsystem from R1 is being expanded upon to include additional functionality that is described in R2.

Commands that are being added in R2 are: *Browse Ingredients, Browse Recipes, Browse Meals, Change Password, Create Team, Send Invite (*to join team), *Leave Team, View Inbox* (invites are accepted here), *Export Data*, and *Import Data*.

The entirety of the *Command* subsystem is the implementation of the Command pattern. The Command pattern has several positive consequences. It becomes the controller tier, and it helps the system adhere to the design principles of high cohesion and open-closed principle. The Command pattern becomes the controller tier because it decouples the object that invokes an operation, which would be in the view tier, from the object that knows how to perform it, which would be in the model tier. The Command pattern promotes high cohesion because it creates smaller classes with more narrowly defined responsibilities, these classes being the individual command classes. Open-closed principle can be maintained by the Command pattern because in most cases all that needs to be done in order to implement new commands would be to create a new class that extends the command interface.

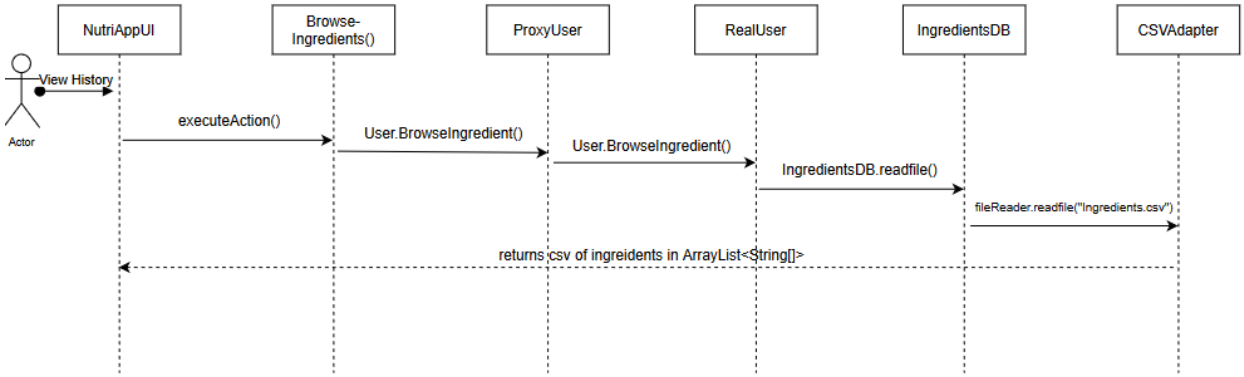
The implementation of the Command pattern does have a major adverse effect however. It increases the coupling in the system. Coupling is increased because each command creates an extra relationship between the invoker, the GUI / client, and the receiver, the User. This is acceptable because high cohesion and an increase in coupling go hand in hand, and in such cases high cohesion should also be prioritized. I also find this increase in coupling to be necessary because as described in the previous paragraph the Command pattern, specifically the concrete commands that it produces act as the controller tier in between the view and model tiers.

Class Structure Diagram



Sequence Diagram

Sequence Diagram for the BrowseIngredient Command



GoF for Command

Name: Command Subsystem		**GoF pattern:** Command
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
User	Receiver	Referenced in commands. Knows how to perform the operations associated with carrying out all the commands.
Command	Command	Defines the interface for all user actions with the NutriApp. The execute() method is invoked each time that the command should perform its related task
BrowseIngredient	Concrete Command	A concrete command that retrieves the ingredients from Ingredient.csv for the user to browse. Calls user.BrowseIngredient().
BrowseRecipes	Concrete Command	A concrete command that retrieves the recipes from Recipe.csv for the user to browse. Calls user.BrowseRecipe().
BrowseMeals	Concrete Command	A concrete command that retrieves the ingredients from Meal.csv for the user to browse. Calls user.BrowseMeal().
ChangePassword	Concrete Command	A concrete command that changes the current user's password. Calls user.ChangePassword().
CreateTeam	Concrete Command	A concrete command that creates a team. The creating user joins this team and if they were already in a team leaves that team that they were previously a member of. Calls user.CreateTeam().
SendInvite	Concrete Command	A concrete command that sends an invite to another user to join their team. Calls user.SendInvite().
LeaveTeam	Concrete Command	A concrete command that removes the user from the team that they are currently in. Calls user.LeaveTeam().
ViewInbox	Concrete Command	A concrete command that displays the contents of the user's inbox to them. Also, provides functionality for

		the user to accept invites, allowing them to join a team. Calls user.ViewInbox().
ExportData	Concrete Command	A concrete command that calls the adapter to read the current data file and then stores it in a file type the user wants to export.
ImportData	Concrete Command	A concrete command that uses the adapter object to read the given file by the user. It will read from the file type it was given then it will store it in the 2d arraylist then it will store it in the default file type.
Deviations from the standard pattern:		
Requirements being covered: 2.b Change their password, send team invite, join team, leave a team; 3 (partly) team creation command; 5(partly) export and importing commands		

User Authentication

Rationale

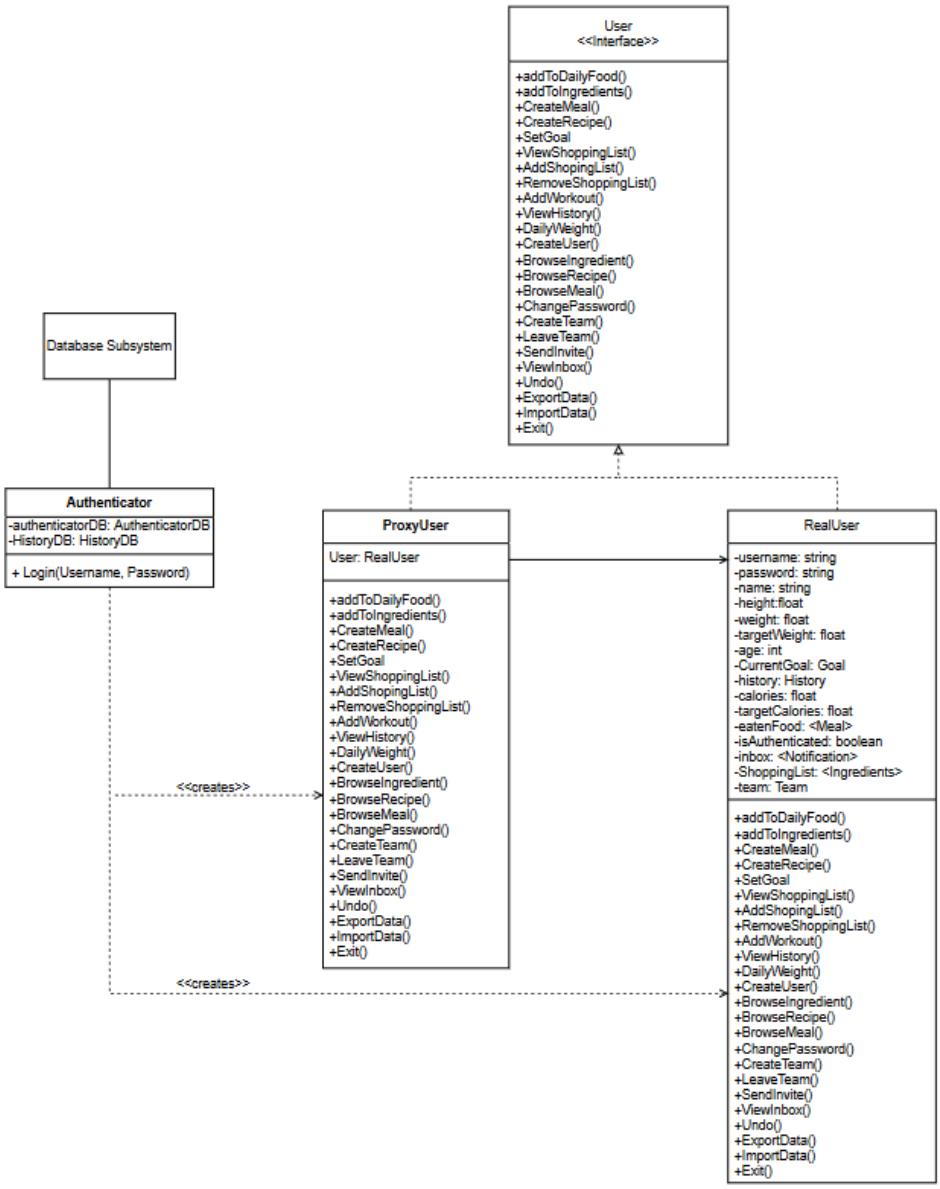
The *User Authentication* Subsystem has two main functions.

Firstly, it allows users to authenticate themselves by logging in using a username and password combination, which results in them being either an *Authenticated User* or a *Guest User* depending on whether or not the login information that they provided was valid. The two user types are denoted via the *isAuthenticated* field in user that is a boolean. Secondly, through the use of the *Proxy Pattern* it restricts access to commands based on whether or not a user is authenticated. The *User Authentication* Subsystem works toward fulfilling requirements 1 and 2, and non-functional requirement 2.

The application of the Proxy pattern benefits the system as it allows for additional housekeeping tasks before accessing an object, which in this system would be the *RealUser*. The housekeeping task that is being done is checking the user's authentication status on methods that are called by commands and determining whether or not their status and the command they are trying to perform is valid. If their request is invalid, the access to the *RealUser* object would be denied by the *ProxyUser*

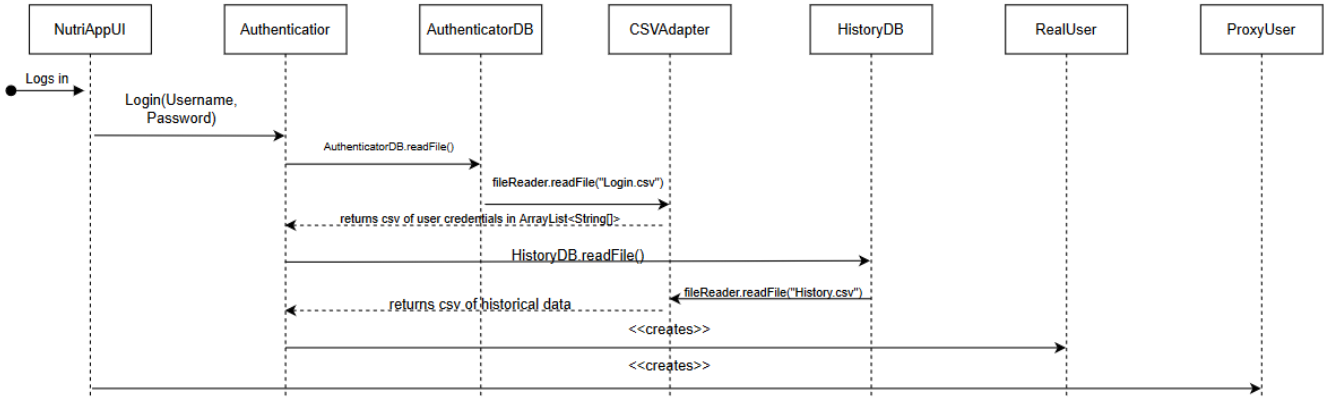
A drawback to using the proxy pattern is the increased coupling in the system. Low coupling, The proxy pattern goes against low coupling because it adds a relationship by creating the *ProxyUser* class which goes in between the various commands and the *RealUser*. However, this added relationship and increase in coupling is necessary, as checking the user's authentication status against the command they are trying to perform in the *ProxyUser* is necessary in order to best fulfill the requirement of restricting the guest user's accessibility.

Class Structure Diagram

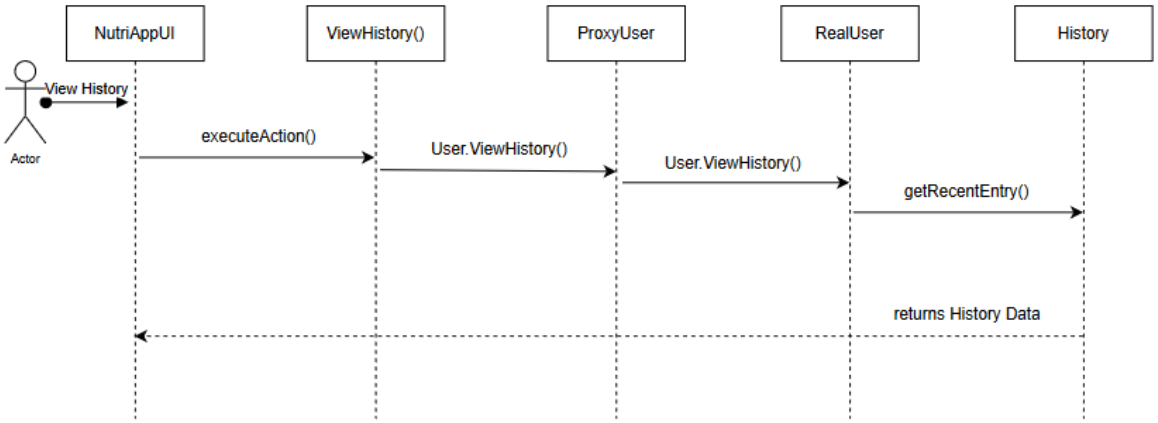


Sequence Diagram

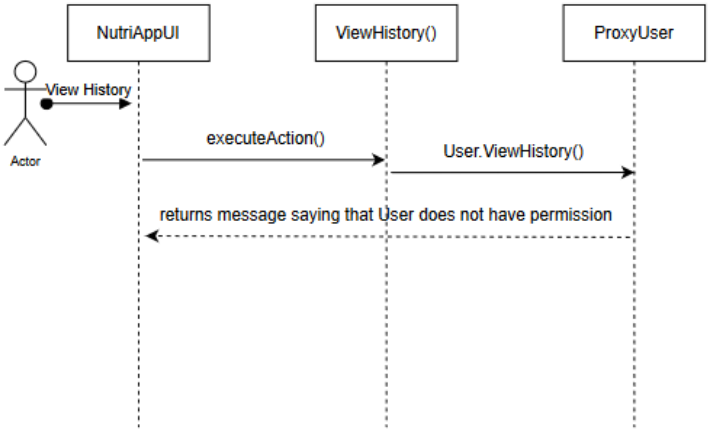
Sequence diagram for logging in as an Authenticated User



Sequence diagram for an Authenticated User doing a command



Sequence diagram for a Guest User doing a command



GoF for Proxy Pattern

Name: Command Access Checking within the User Authentication Subsystem		**GoF pattern:** Proxy Pattern (Protection)
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
User	Subject	Defines the common interface for RealUser and ProxyUser so that a ProxyUser can be used anywhere a RealUser is expected.
ProxyUser	Proxy	Controls access to the RealUser. Maintains a reference that lets the ProxyUser access the RealUser. It checks that the User that performs a command has the access permissions required to perform that command. In most cases, If the User is not authenticated, then the ProxyUser will return a message saying that the command cannot be performed by a non-authenticated user.
RealUser	Real Subject	Defines the RealUser that the ProxyUser represents. If the User that performs a command is authenticated, then the ProxyUser directs the request to RealUser, which performs the requested action from the command.
Deviations from the standard pattern:		
Requirements being covered: 1-2. Restriction of access for non-authenticated users and allowing of access for authenticated users.		

Database

Rationale

The *Database* Subsystem has three main functions.

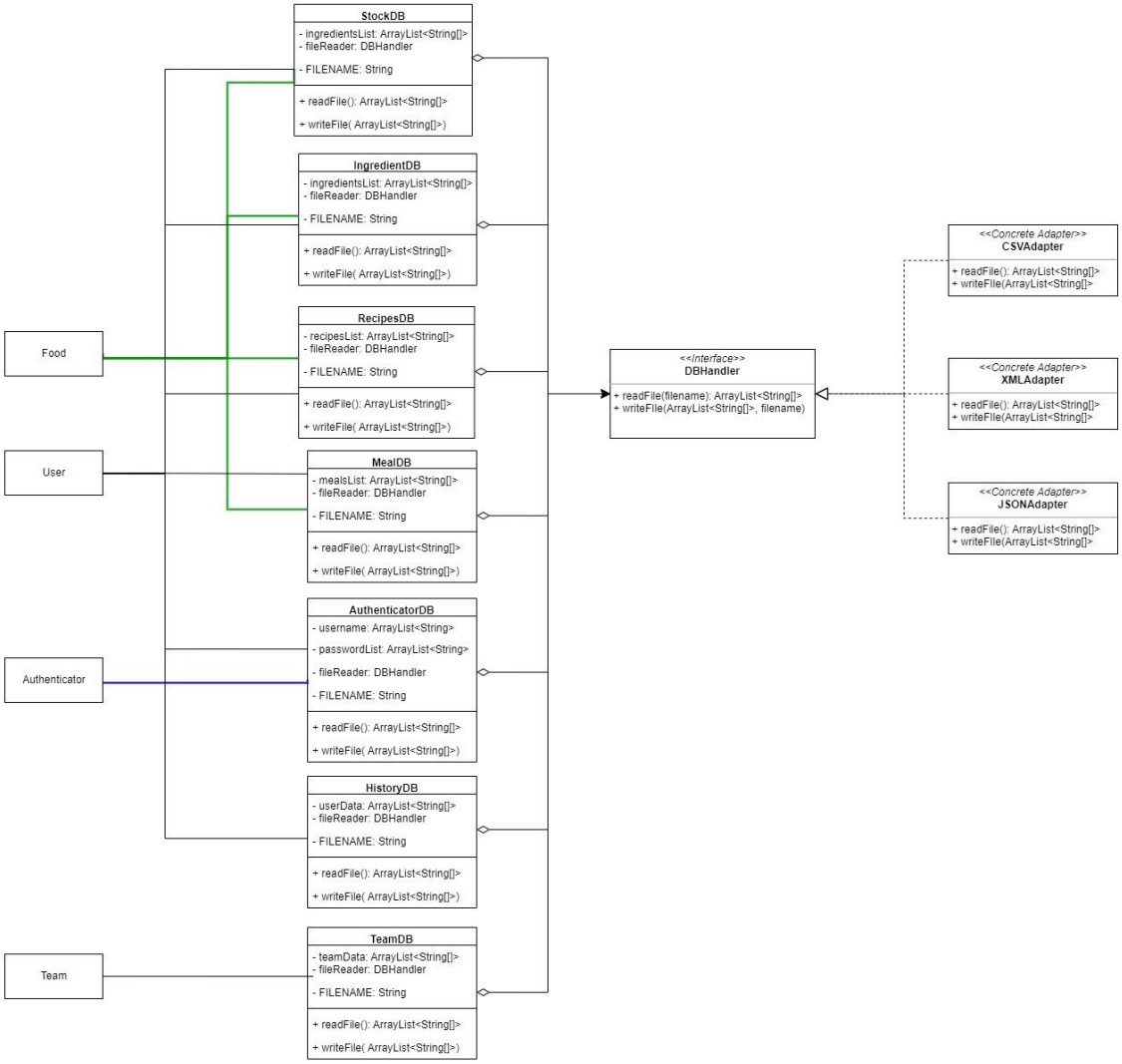
Firstly, it achieves data storage by creating CSV files for which data that needs to be stored persistently is housed. Data that needs to be stored are ingredients, stock ingredients, recipes, meals, login information, team, and user history.

Secondly, it provides functionality that allows a user to edit stored data by providing read and write operations for files that work toward adding and removing data from storage.

Thirdly, it provides functionality that allows users to export, import, and store data in CSV, JSON, and XML formats. The implementation also allows for the ability to save to or load from a real database if needed, which can be done by implementing the adapter.

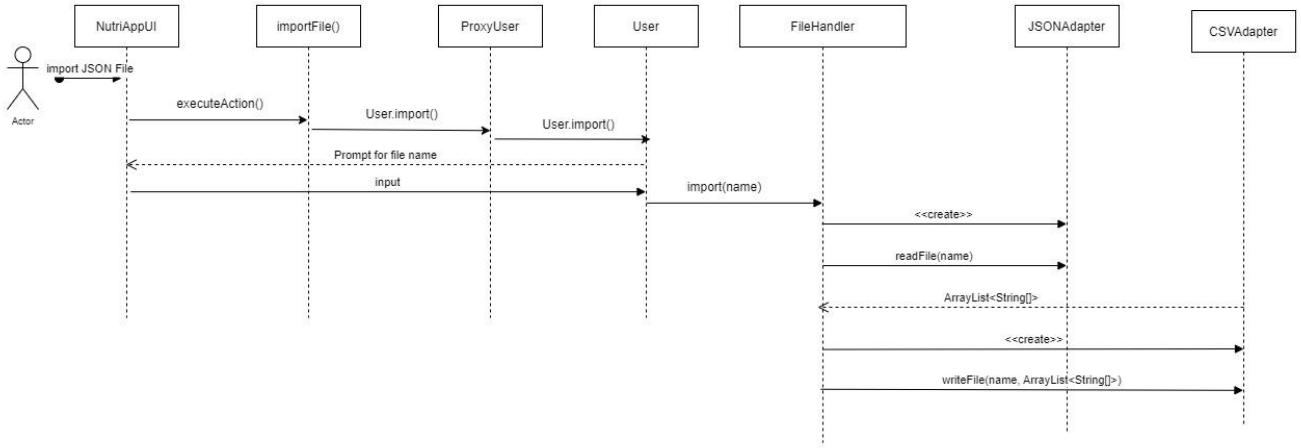
The *Database* Subsystem works toward fulfilling requirement 5, and non-functional requirements 2 and 3. Adding the adapter pattern to the database subsystem provides a layer of abstraction which allow to follow open-close principle and lowers the coupling between classes. Having an adapter for each types of files also follows single responsibility. It also allow to override some of the behaviors. Although this pattern increase the code complexity the benefit out weigh the cost.

Class Structure Diagram

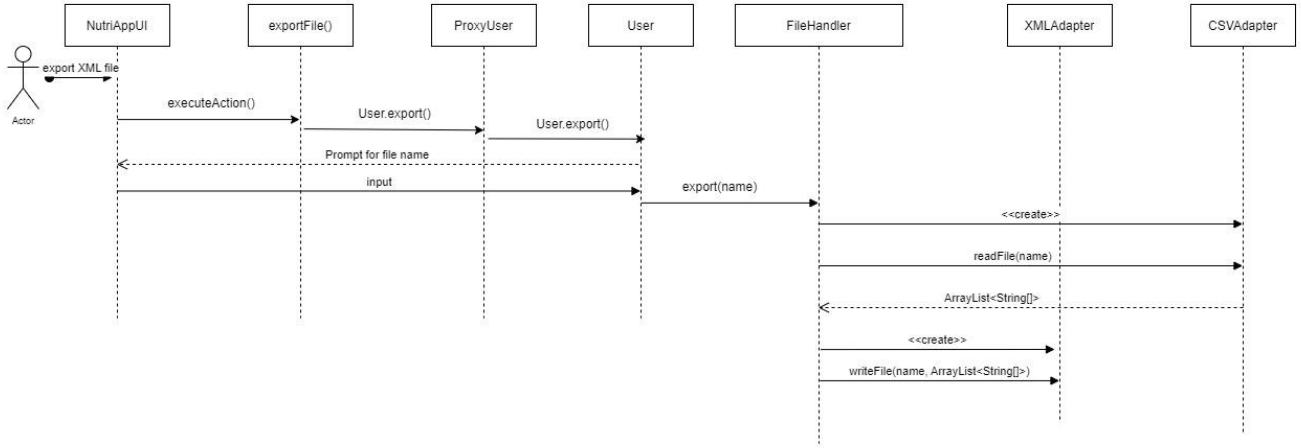


Sequence Diagram

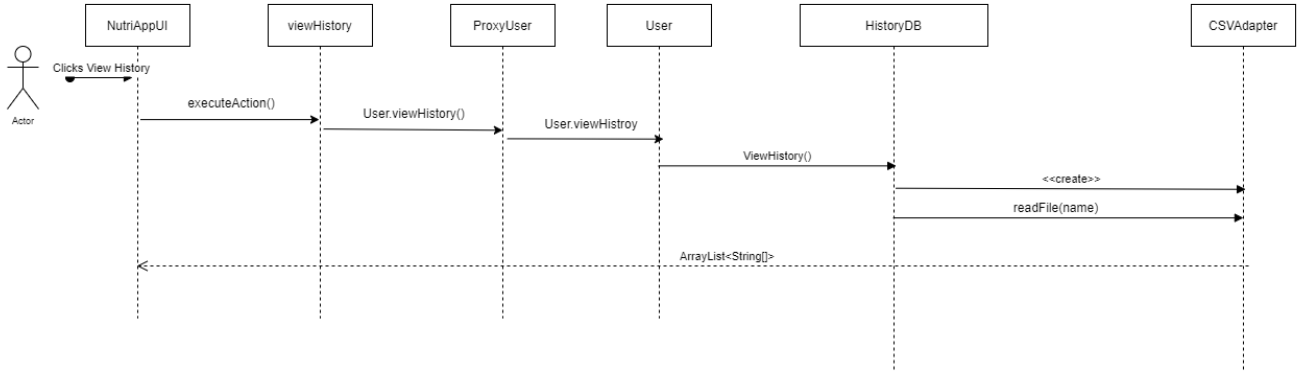
Importing a JSON file Sequence Diagram



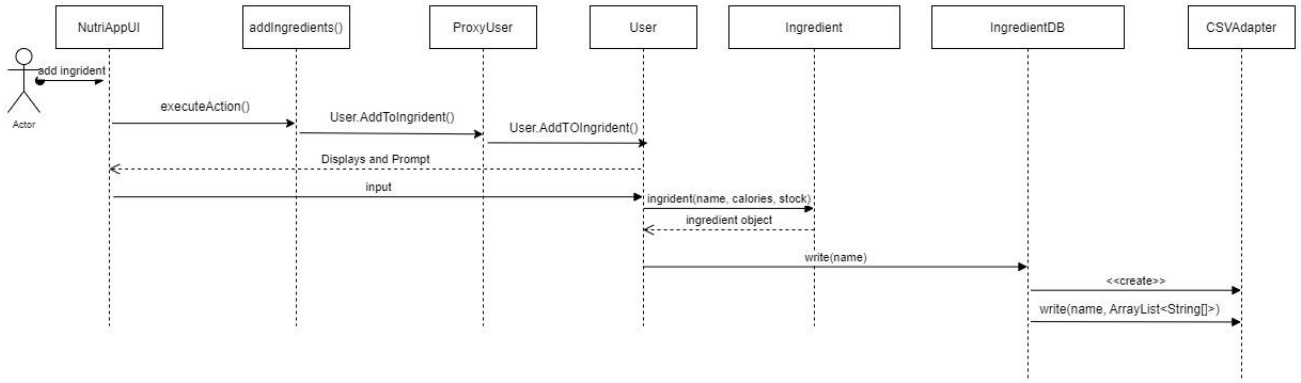
Exporting a XML file Sequence Diagram



Reading a file Sequence Diagram



Writing a file Sequence Diagram



GoF for Adapter

Name: DataBase		**GoF pattern: Adapter**
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
DBHandler	Adapter	Creates a layer of abstraction between the application and database API. Other classes will aggregate this interface as an instance. It follows the open-close principle and has low coupling and high cohesion.
CSVAdapter	Concrete Adapter	This class will use the CSV library to get the necessary functions to read and write CSV files. It will also implement DBHandler. Then it will use CSV library implementation to implement DBHandler.
XMLAdapter	Concrete Adapter	This class will use the XML library to get the necessary functions to read and write XML files. It will also implement DBHandler. Then it will use XML library implementation to implement DBHandler.
JSONAdapter	Concrete Adapter	This class will use the JSON library to get the necessary functions to read and write JSON files. It will also implement DBHandler. Then it will use JSON library implementation to implement DBHandler.
IngredientDB	Concrete Component	Uses the adapter pattern to read and write a file. Creates an instance of DBHandler type to use its function to read and write into ingredients file.

RecipesDB	Concrete Component	Uses the adapter pattern to read and write a file. Creates an instance of DBHandler type to use its function to read and write into recipes file.
MealDB	Concrete Component	Uses the adapter pattern to read and write a file. Creates an instance of DBHandler type to use its function to read and write into meals file.
AuthenticatorDB	Concrete Component	Uses the adapter pattern to read and write a file. Creates an instance of DBHandler type to use its function to read and write a login file.
HistoryDB	Concrete Component	Uses the adapter pattern to read and write a file. Creates an instance of DBHandler type to use its function to read and write a history file.
TeamDB	Concrete Component	Uses the adapter pattern to read and write a file. Creates an instance of DBHandler type to use its function to read and write a team file.
Deviations from the standard pattern:		
Requirements being covered: The user may export/import the database and they can opt into any supported formats (CSV, JSON, or XML).		

Undo

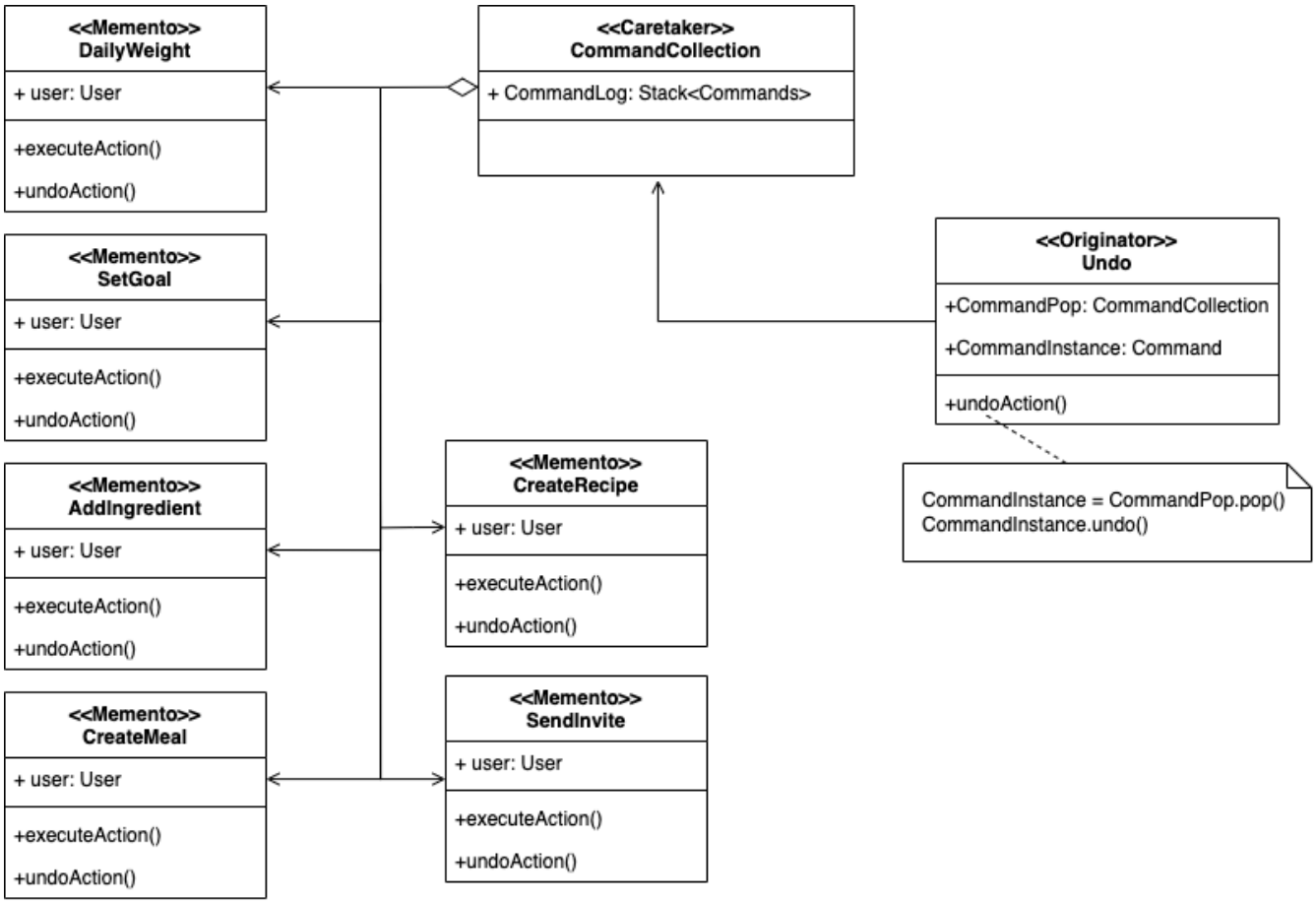
Rationale

With the new set of requirements that came with the R2 of NUTRiAPP, we dedicated this Undo Subsystem to encompass what's needed to be done to satisfy all the requirements that comes with 04 in the Design Problem Document. Our design targets not only the four required undoable commands (entering daily weight, setting/changing goals, adding/removing ingredients from stock, and adding users to their team) but all encapsulated commands. As this design was built upon our already existing subsystem (Command from R1), some changes were made to the original subsystem. To be specific, an undoAction method was added to our Command interface, which later gets used by our Undo class <<Originator>> . Thanks to this addition to a previous subsystem, we are able to utilize the new method by popping Commands from the CommandCollection <<Caretaker>>. The requirements of all the four of the commands have each do different things and their respective functionality can be found in their GoF section.

The introduction of this modified subsystem not only brings the same adherence (and/or lack) of design principles from the previous subsystem (Command Subsystem from R1) but also new sets of design principles. The scope we'll cover are only the new design principles that this subsystem adheres to but mainly focusing on the Memento Pattern's involvement in it. With both the CommandCollection and the Undo classes, we strictly follow Single Responsibility Principle and Information Expert. Single Responsibility is adhered to as CommandCollection is solely responsible of keeping a collection of commands while Undo has the role of undoing whatever the latest command may be, nothing else. Information Expert can be seen within Undo as it uses the information of all the stored commands (found in CommandPop) to do what it wishes, in this case being the action of undoing.

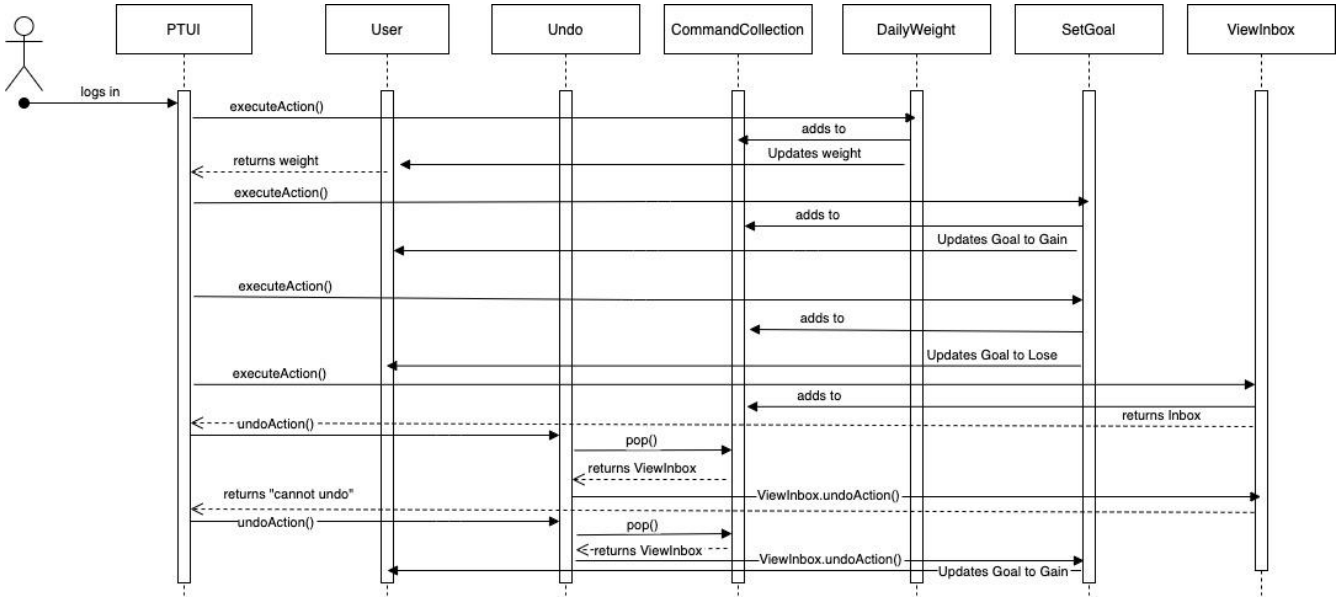
With any new implementation, one has to weigh the advantages and trade-offs that comes with anything and the Undo Subsystem isn't an exception. One of the SOLID principles that this subsystem clearly breaks is the Interface Segregation Principle. Only four commands, from the requirements, requires one to undo their action while the rest don't, hence not every command in the Command Subsystem needs to exposed to the undoAction method. We collectively decided to break this principle due to the simple fact that it would introduce complexity to the system to not only the team developing but also the client (the client needs to be told or just needs to know what command are/aren't undoable). The single reroute we chose is to have all of the commands implement the undo but have it so that if it's not one of the four required, it either informs the clients that previous command can't be undone or implement something similar that makes sense (changePassword Command doesn't require an undo but we can call the same executeAction method to allow the user to change their password as their "undo").

Class Structure Diagram



Sequence Diagram

Performing four commands and undoing two of them (the first undoing is a command not specified in the requirements that needs to be undone)



GoF for Undo

Name: Undo		**GoF pattern: Memento**
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
CommandCollection	Caretaker	Plays the role of “history log” and keeps a Stack of Commands that the clients performs. Does not judge what kinds of Commands get added to it and works at hand with the Originator in implementing the Undo Feature
Undo	Originator	The main player in the whole undo feature. When a user performs the Undo Command, it accesses the most recent command from the CommandCollection and both pops it out of the history log and performs that Command’s undo operation(if it has one, otherwise it tells the user there’s nothing to undo. Has

		an instance of both the list and an instanceCommand it uses to undo
CreateRecipe	Memento	One of the commands that has a designated undo operation, in this case being that it deletes the most recent instance of a recipe and informs the user what it just did
SendInvite	Memento	The undo operation for the SendInvite it to retrieve back all the notifications it sent to all the users (or user) and inform the user that the invite has been retracted
DailyWeight	Memento	For the daily weight the users are required to input their daily weight at the start of a new day. If any user, by chance, happens to mess up their input they're able to undo their command. This undo just calls the command again and asked the user to input their daily weight. Previous entry is deleted and the new data is set to User
SetGoal	Memento	For setGoal the user's goal is simply rev erted back to the previous goal they had. No manipulation of data is needed. For instances of having only one goal set ever, it returns that there are no previous goals to revert to
AddIngredient	Memento	The UndoAction of AddIngredients is responsible for not only deleting any instance of created ingredients added to the database but it's responsible for the user to be notified of the food that got deleted
CreateMeal	Memento	Similar to AddIngredients, the Undo of CreateMeal deletes of the created meal from the database and informs the user of the recent meal that got deleted
Deviations from the standard pattern: The main deviation is that it's a combination of the Command Pattern and the Memento Pattern. The Mementos are the Commands themselves and the rest are applicable elements for the Memento Pattern.		
Requirements being covered: 4a, 4b, 4c, 4d		

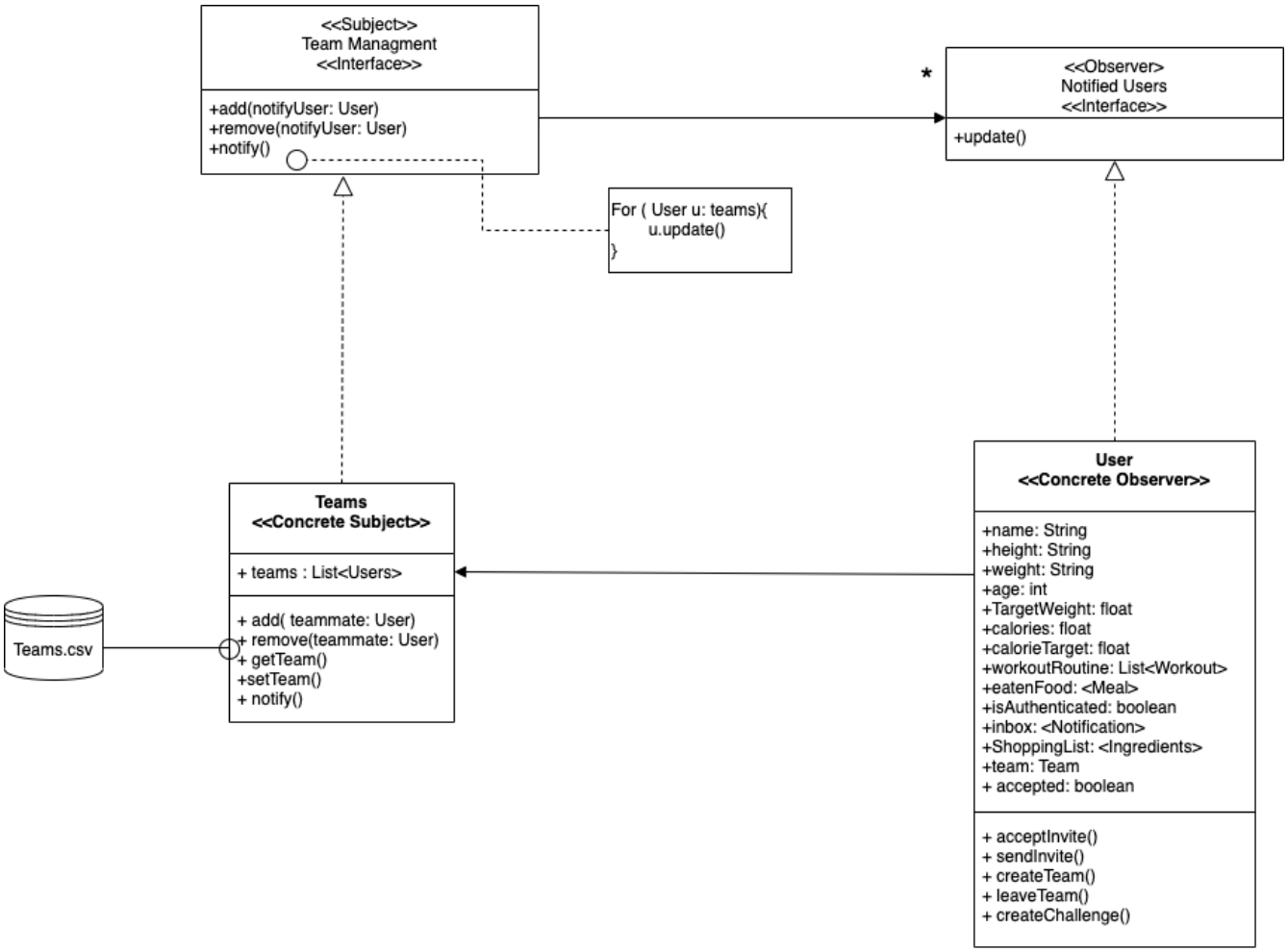
Teams Subsystem

Rationale

The NutriApp™ R2 requriements describe several features that require the application to respond to the changes in state of dependent objects. This includes “Registered users may form teams”, “Team members are notified whenever another team member logs a workout”, “Any team member may invite other users to join a team”, “Team members may issue week-long challenges to the team”, “Authenitcated users can accept requests from other users to join a team”, “Authenitcated users can leave a team”. The Teams subsystem is implementing the observer design establish a dependency between a subject (*Team*) and its Observer(s)(*User*) such that, when the state of the subject(*Team*) changes, the Observer(s)(*User*) is updated automatically. The previously mentioned features describe changes in state, i.e a user creates a team, a team member invites users to their team, a team member sets a weekly challenge, a member leaves the team and the actions that are taken in response, i.e all team members are notified when a new member is added to the team, all team members are notified when a new weekly challenge is issued, all team members are notified when a member leaves the team.

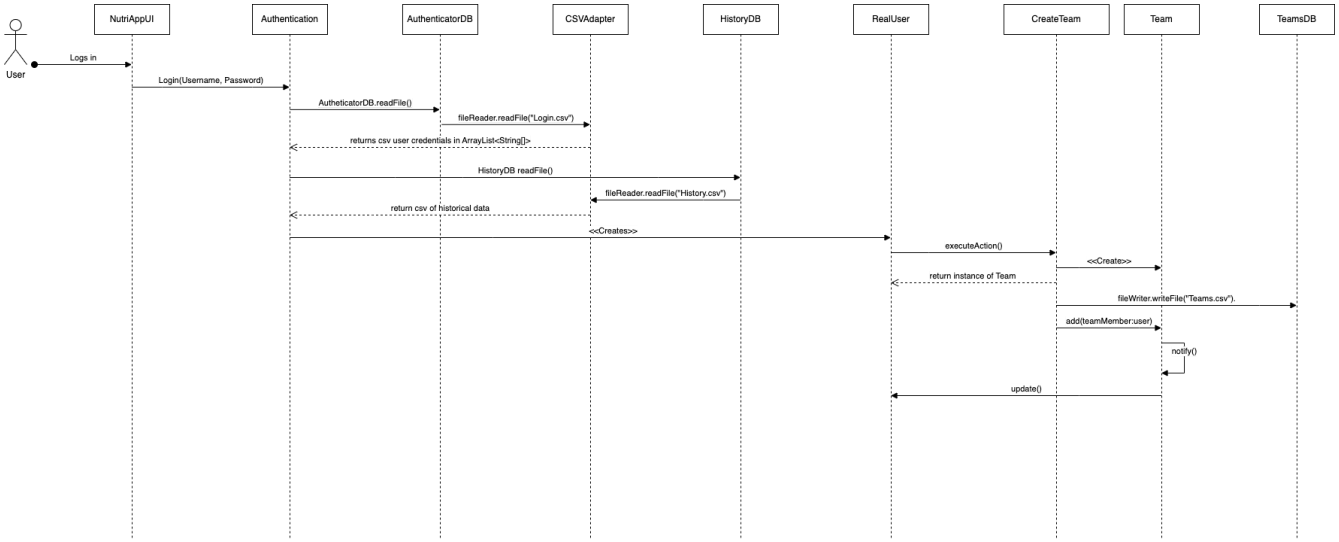
The Observer pattern has several positive consequences to the overall design of the Team subsystem. By defining interfaces for the Subject(*TeamManagment*) and Observer(*Notified Users*), the pattern conforms to the Dependency Inversion principle by inserting a layer of abstraction between the Concrete Subject(*Teams*) and its concrete Oberver(s) (User). The concrete observers (*User*) are registered with the concrete subject (*Teams*) by injecting them using the add method which is an example of the Dependency Injection principle. The pattern also conforms to the Open/Closed principle because we can introduce new observers without having to make changes to the subject. Additionally, the pattern also follows the Single Responsability principle because each of the participants has a specific non-overlapping responsibility. One trade-off is that the implementation of the Observer pattern can cause unexpected updates to occur because the observers are not aware of each others presence, and they are blindly to the costly changing of the subject. These tradeoff is acceptable given the benefits provided by the Dependency Injection principle, the Open/Closed principle, and the Single Responsability principle.

Class Structure Diagram



Sequence Diagram

Sequence of a user loging in an creating a team



GoF for Teams

Name: Team		**GoF pattern: Observer**
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
Team Managment	Subject	Defines the interface for the concrete subject that allows it to attach and detach the observer(s) from the list of observers that need to be notified.
Notified Users	Observer	Defines the interface for updating the observer(s) that should be notified of the changes that occur in the subject.
Teams	Concrete Subject	Stores the states of the concrete observer(s) objects using the add and remove method. It notifies the observers when its state changes.
User	Concrete Observer	Maintains a reference to the concrete subject object and stores the state that should stay consistent the subject by calling the update method.
Deviations from the standard pattern:		
Requirements being covered: 2)IV. Accept requests from other users to join a team. V. Leave a team 3). Registered users may form teams. a. A user may only be on one team at a time. b. Any team member may invite other users to join a team. c. Team members may see each other's workout history only. Team members do not have visibility into each other's goals or other personal information. d. Team members are notified whenever another team member logs a workout. If offline, they will receive these notifications upon logging into the system. e. Team members may issue week-long challenges to the team. Team members are ranked according to the number of exercise minutes that they log during the challenge. There may only be one challenge running at a time.		

Appendix

Class: Command	
Responsibilities: Defines the interface for all user actions with the NutriApp. The execute() method is invoked each time that the command should perform its related task	
Collaborators:	
Users: …	**Used by:** …
Author: Aaron Santos	

Class: BrowseIngredient	
Responsibilities: A concrete command that retrieves the ingredients from Ingredient.csv for the user to browse. Calls user.BrowseIngredient().	
Collaborators:	
Users: GUI	**Used by:** User
Author: Aaron Santos	

Class: BrowseRecipes	
Responsibilities: A concrete command that retrieves the recipes from Recipe.csv for the user to browse. Calls user.BrowseRecipe().	
Collaborators:	
Users: GUI	**Used by:** User

Author: Aaron Santos	

Class: BrowseMeals	
Responsibilities: A concrete command that retrieves the ingredients from Meal.csv for the user to browse. Calls user.BrowseMeal().	
Collaborators:	
Users: GUI	**Used by:** User
Author: Aaron Santos	

Class: ChangePassword	
Responsibilities: A concrete command that changes the current user's password. Calls user.ChangePassword().	
Collaborators:	
Users: GUI	**Used by:** User
Author: Aaron Santos	

Class: CreateTeam	
Responsibilities: A concrete command that creates a team. The creating user joins this team and if they were already in a team leaves that team that they were previously a member of. Calls user.CreateTeam().	
Collaborators:	

Users: GUI	**Used by:** User
Author: Aaron Santos	

Class: SendInvite	
Responsibilities: A concrete command that sends an invite to another user to join their team. Calls user.SendInvite().	
Collaborators:	
Users: GUI	**Used by:** User
Author: Aaron Santos	

Class: LeaveTeam	
Responsibilities: A concrete command that removes the user from the team that they are currently in. Calls user.LeaveTeam().	
Collaborators:	
Users: GUI	**Used by:** User
Author: Aaron Santos	

Class: ViewInbox	
Responsibilities: A concrete command that displays the contents of the user's inbox to them. Also, provides functionality for the user to accept invites, allowing them to join a team. Calls user.ViewInbox().	

Collaborators:	
Users: GUI	**Used by:** User
Author: Aaron Santos	

Class: User	
Responsibilities: Referenced in commands. Knows how to perform the operations associated with carrying out all the commands.	
Collaborators:	
Users: …	**Used by:** …
Author: Aaron Santos	

Class: Authenticator	
Responsibilities: Takes in login information and determines whether or not it is valid. Creates instances of RealUser and ProxyUser.	
Collaborators:	
Users: GUI	**Used by:** AuthenticatorDB, CSVAdapter, HistoryDB
Author: Aaron Santos	

Class: User	
Responsibilities: Defines the common interface for RealUser and ProxyUser so that a ProxyUser can be used anywhere a RealUser is expected.	

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

Class: ProxyUser	
Responsibilities: Controls access to the RealUser. Maintains a reference that lets the ProxyUser access the RealUser. It checks that the User that performs a command has the access permissions required to perform that command.	
Collaborators:	
Users: ...	**Used by:** RealUser
Author: Aaron Santos	

Class: RealUser	
Responsibilities: Defines the RealUser that the ProxyUser represents. If the User that performs a command is authenticated, then the ProxyUser directs the request to RealUser, which performs the requested action from the command.	
Collaborators:	
Users: ProxyUser	**Used by:** ...
Author: Aaron Santos	

Class: DBHandler	
Responsibilities: Creates a layer of abstraction between the application and database API. Database adapter classes can implement this interface to have same methods.	
Collaborators:	
Users: CSVAdapter, XMLAdapter, JSONAdapter, StockDB, IngredientDB, RecipesDB, MealDB, AuthenticatorDB, HistoryDB	**Used by:**
Author: Laxmi Poudel	

Class: CSVAdapter	
Responsibilities: Implements the DBHandler using the CSV library functions. It has the functions to read and write the CSV files.	
Collaborators:	
Users: StockDB, IngredientDB, RecipesDB, MealDB, AuthenticatorDB, HistoryDB	**Used by:** DBHandler
Author: Laxmi Poudel	

Class: XMLAdapter	
Responsibilities: Implements the DBHandler using the XML library functions. It	

has the functions to read and write the XML files.	
Collaborators:	
Users: StockDB, IngredientDB, RecipesDB, MealDB, AuthenticatorDB, HistoryDB	**Used by:** DBHandler
Author: Laxmi Poudel	

Class: JSONAdapter	
Responsibilities: Implements the DBHandler using the JSON library functions. It has the functions to read and write the JSON files.	
Collaborators:	
Users: StockDB, IngredientDB, RecipesDB, MealDB, AuthenticatorDB, HistoryDB	**Used by:** DBHandler
Author: Laxmi Poudel	

Class: StockDB	
Responsibilities: Has instance of DBHandler which allows it to read and write user current ingredient list and stock amount. Has a 2d arraylist to keep track of the ingredient to read and write. Has a file name which contain the ingredient data.	
Collaborators:	
Users: Ingredient, User, ImportData, ExportData	**Used by:** DBHandler, CSVAdapter, XMLAdapter, JSONAdapter

Author: Laxmi Poudel	

Class: IngredientDB	
Responsibilities: Has instance of DBHandler which allows it to read and write ingredients list. Has a 2d arraylist to keep track of the ingredient to read and write. Has a file name which contain the ingredient data.	
Collaborators:	
Users: Ingredient, User	**Used by:** DBHandler, CSVAdapter, XMLAdapter, JSONAdapter
Author: Laxmi Poudel	

Class: RecipesDB	
Responsibilities: Has instance of DBHandler which allows it to read and write recipesList. Has a 2d arraylist to keep track of the recipes to read and write. Has a file name which contain the recipe data.	
Collaborators:	
Users: Recipes, User	**Used by:** DBHandler, CSVAdapter, XMLAdapter, JSONAdapter
Author: Laxmi Poudel	

Class: MealDB	

Responsibilities: Has instance of DBHandler which allows it to read and write mealList. Has a 2d arraylist to keep track of the meals to read and write. Has a file name which contain the meal data.	
Collaborators:	
Users: Meal, User	**Used by:** DBHandler, CSVAdapter, XMLAdapter, JSONAdapter
Author: Laxmi Poudel	

Class: AuthenticatorDB	
Responsibilities: Has instance of DBHandler which allows it to read and write username and password list. Has a 2d arraylist to keep track of the user login info to read and write. Has a file name which contain the user login info data.	
Collaborators:	
Users: Authenticator, User	**Used by:** DBHandler, CSVAdapter, XMLAdapter, JSONAdapter
Author: Laxmi Poudel	

Class: HistoryDB	
Responsibilities: Has instance of DBHandler which allows it to read and write user history list. Has a 2d arraylist to keep track of the history to read and write. Has a file name which contain the history data.	

Collaborators:	
Users: History, User	**Used by:** DBHandler, CSVAdapter, XMLAdapter, JSONAdapter
Author: Laxmi Poudel	

Class: MealDB	
Responsibilities: Has instance of DBHandler which allows it to read and write teamList. Has a 2d arraylist to keep track of the teams to read and write. Has a file name which contain the team data.	
Collaborators:	
Users: Team, User	**Used by:** DBHandler, CSVAdapter, XMLAdapter, JSONAdapter
Author: Laxmi Poudel	

Class: CommandCollection	
Responsibilities: Responsible of maintaining a ordered record of all the commands that a user may perform. This gets added to the Stack within this class.	
Collaborators:	
Users: DailyWeight, SetGoal, AddIngredient, CreateMeal, CreateRecipe, and SendInvite	**Used by:** Undo
Author: Himel Uddin	

Class: Undo	
Responsibilities: To undo the most recent command performed; this most recent command is the latest entry inside CommandCollection	
Collaborators:	
Users: CommandCollection	**Used by:**
Author: Himel Uddin	

Class: DailyWeight	
Responsibilities: This command is called upon at the start of every new day and is responsible for taking a weight from the user and setting it for the profile and adding it to the history database	
Collaborators:	
Users:	**Used by:** CommandCollection
Author: Himel Uddin	

Class: SetGoal	
Responsibilities: Whenever the user calls on this command, it's responsible for setting the Goal of the user and changing the attribute within User to whichever Goal was chosen	
Collaborators:	

Users:	**Used by:** CommandCollection
Author: Himel Uddin	

Class: AddIngredient	
Responsibilities: If an ingredient is not in the database (or the User chooses to add a new ingredient), this command is responsible for adding that new ingredient	
Collaborators:	
Users:	**Used by:** CommandCollection
Author: Himel Uddin	

Class: CreateMeal	
Responsibilities: Takes all the required recipe and creates a specific meal plan for the user. This meal that the user created gets added to the respective database.	
Collaborators:	
Users:	**Used by:** CommandCollection
Author: Himel Uddin	

Class: CreateRecipe	
Responsibilities: Takes not only a list of ingredients from the user but also String Instructions to build a specific recipe. This gets saved to the database meant for	

recipes.	
Collaborators:	
Users:	**Used by:** CommandCollection
Author: Himel Uddin	

Class: SendInvite	
Responsibilities: Creates a Notification instances that gets sent out to a list of users the client specifies. This command can only be done if the user created a team.	
Collaborators:	
Users:	**Used by:** CommandCollection
Author: Himel Uddin	

Class: Team Managment	
Responsibilities: Defines the interface that allows the cocrete subject attach and detach the observer(s) by predefining the add, remove, and notify methods.	
Collaborators:	
Users:	**Used by:** Team
Author: Himel Uddin	

Class: Notified Users	
Responsibilities: Defines the nterface for updating the observer(s) that should be notified of the changes that occur in Teams by predefining the update method.	
Collaborators:	
Users:	**Used by:** Team
Author: Abiel Yemane	

Class: Teams	
Responsibilities: Implements the Team Managment interface. Adds concrete observer(s) user to the team members list after they accept the team invitation using the add method. Removes users from the team when they leave the team using the remove method. Notifies users when a someone joins/leaves the team and when a new challenge is issued. Using the notify method it loops through the list of observers and calls the update function for each of them.	
Collaborators:	
Users:	**Used by:** Team
Author: Abiel Yemane	

Class: User	

Responsibilities: Implements the Notified Users interface to updating the concrete observer(s) User to be notified of the changes that occur in the concrete subject Teams. Stores a the state of the Teams object and updates it using the update method every time its state changes.	
Collaborators:	
Users:	**Used by:** Team
Author: Abiel Yemane	