# Group 1

# Demo-I : The Demo

Submission Date: 10 December 2023

Ata Tütek
Beril Bilici
Deniz Gülal
Diğdem Yıldız
Fatma Ceren Tarım
Muhammed Can Durmuş

# Table of Contents

**1 - Class Diagram**

**2 - The Logical Architecture of Our Design**

**3 - The GRASP Design Patterns for Each Use Case:**

**4- The Implementation of Design Patterns (GoF Design Patterns)**

**5- General Design & Git Usage**

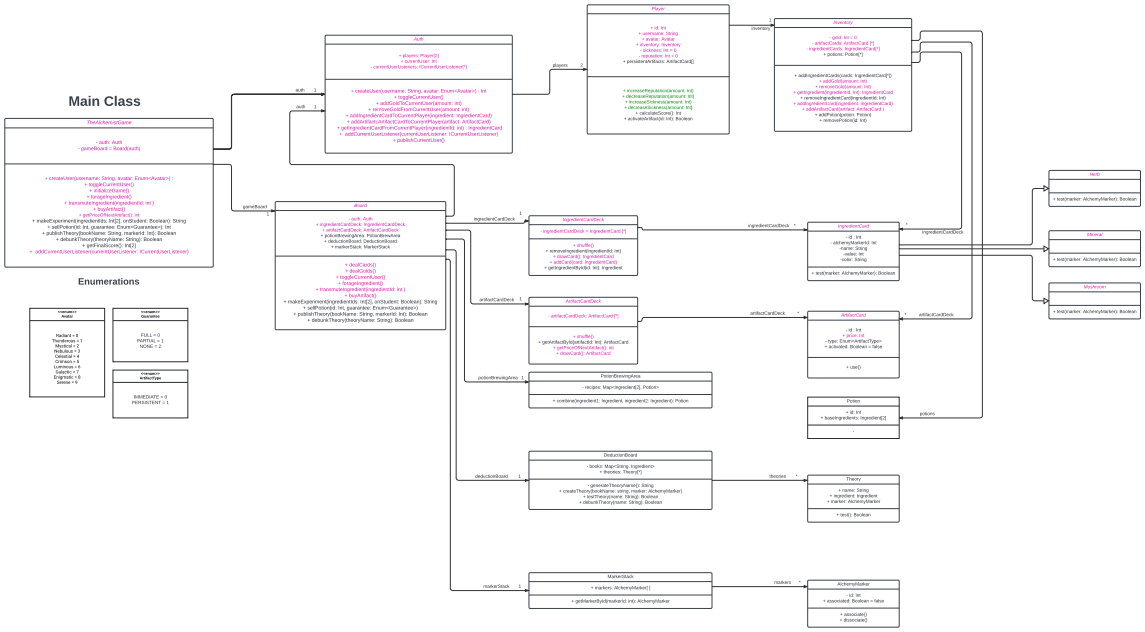
**6- The Path of Our Project**

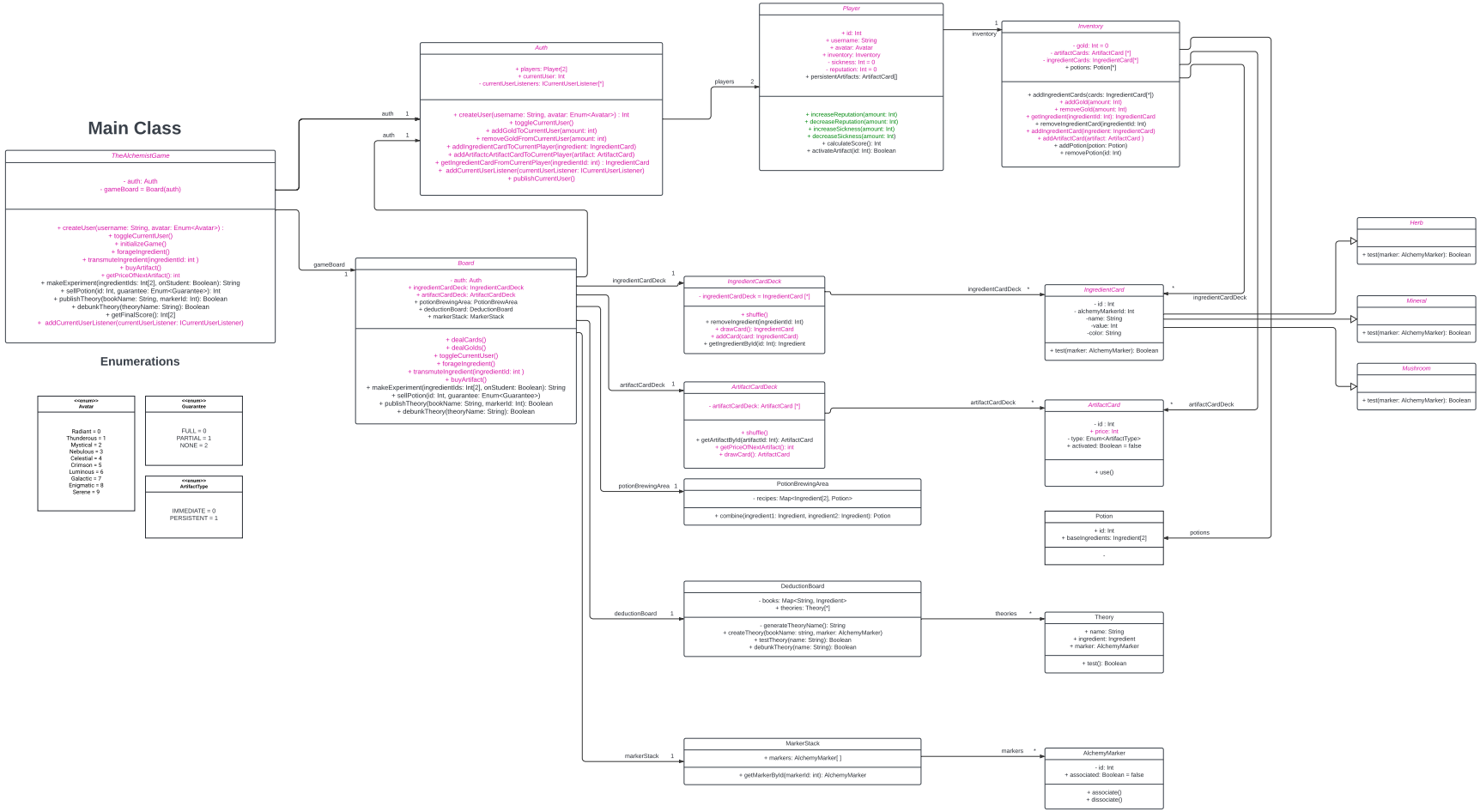
# Demo-1 Changes and Implementations

## Class Diagram

Firstly, we have designed our class diagram before we get into the actual coding section. In such a way, we first figured out how we could implement the strategies within the actual code.

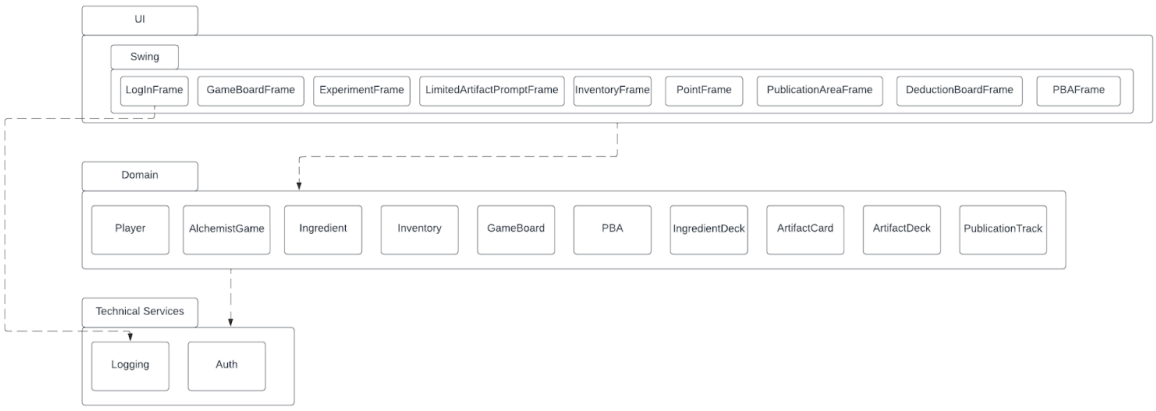
Through such a perspective we have implemented the required sections for Demo-1 for our KU Alchemists Game. On the other hand, as we did not apply the “Waterfall Effect” but rather we have organized our path along our journey; we have changed our class diagram to satisfy the required outputs. Therefore, along our journey, we revise our Class Diagram in a parallel manner to achieve a synchronized Class Diagram and code. Here is the revised Class Diagram (larger version in the next page):




Main Class
TheAlchemistGame
- auth: Auth
- gameBoard = Board(auth)
+ createUser(username: String, avatar: Enum<Avatar>) :
+ toggleCurrentUser()
+ initializeGame()
+ forageIngredient()
+ transmuteIngredient(ingredientId: int )
+ buyArtifact()
+ getPriceOfNextArtifact(): int
+ makeExperiment(ingredientIds: Int[2], onStudent: Boolean): String
+ sellPotion(id: Int, guarantee: Enum<Guarantee>): Int
+ publishTheory(bookName: String, markerId: Int): Boolean
+ debunkTheory(theoryName: String): Boolean
+ getFinalScore(): Int[2]
+ addCurrentUserListener(currentUserListener: ICurrentUserListener)
Enumerations
<<enum>>
Avatar
Radiant = 0
Thunderous = 1
Mystical = 2
Nebulous = 3
Celestial = 4
Crimson = 5
Luminous = 6
Galactic = 7
Enigmatic = 8
Serene = 9
<<enum>>
Guarantee
FULL = 0
PARTIAL = 1
NONE = 2
<<enum>>
ArtifactType
IMMEDIATE = 0
PERSISTENT = 1
auth
1
auth
1
Auth
+ players: Player[2]
+ currentUser: Int
- currentUserListeners: ICurrentUserListener[*]
+ createUser(username: String, avatar: Enum<Avatar>) : Int
+ toggleCurrentUser()
+ addGoldToCurrentUser(amount: int)
+ removeGoldFromCurrentUser(amount: int)
+ addIngredientCardToCurrentPlayer(ingredient: IngredientCard)
+ addArtifactcArtifactCardToCurrentPlayer(artifact: ArtifactCard)
+ getIngredientCardFromCurrentPlayer(ingredientId: int) : IngredientCard
+ addCurrentUserListener(currentUserListener: ICurrentUserListener)
+ publishCurrentUser()
players
2
Player
+ id: Int
+ username: String
+ avatar: Avatar
+ inventory: Inventory
- sickness: Int = 0
- reputation: Int = 0
+ persistentArtifacts: ArtifactCard[]
+ increaseReputation(amount: Int)
+ decreaseReputation(amount: Int)
+ increaseSickness(amount: Int)
+ decreaseSickness(amount: Int)
- calculateScore(): Int
+ activateArtifact(id: Int): Boolean
1
inventory
Inventory
- gold: Int = 0
- artifactCards: ArtifactCard [*]
- ingredientCards: IngredientCard[*]
+ potions: Potion[*]
+ addIngredientCards(cards: IngredientCard[*])
+ addGold(amount: Int)
+ removeGold(amount: Int)
+ getIngredient(ingredientId: Int): IngredientCard
+ removeIngredientCard(ingredientId: Int)
+ addIngredientCard(ingredient: IngredientCard)
+ addArtifactCard(artifact: ArtifactCard )
+ addPotion(potion: Potion)
+ removePotion(id: Int)
gameBoard
1
Board
- auth: Auth
+ ingredientCardDeck: IngredientCardDeck
+ artifactCardDeck: ArtifactCardDeck
+ potionBrewingArea: PotionBrewArea
+ deductionBoard: DeductionBoard
+ markerStack: MarkerStack
+ dealCards()
+ dealGolds()
+ toggleCurrentUser()
+ forageIngredient()
+ transmuteIngredient(ingredientId: int )
+ buyArtifact()
+ makeExperiment(ingredientIds: Int[2], onStudent: Boolean): String
+ sellPotion(id: Int, guarantee: Enum<Guarantee>)
+ publishTheory(bookName: String, markerId: Int): Boolean
+ debunkTheory(theoryName: String): Boolean
ingredientCardDeck
1
IngredientCardDeck
- IngredientCardDeck = IngredientCard [*]
+ shuffle()
+ removeIngredient(ingredientId: Int)
+ drawCard(): IngredientCard
+ addCard(card: IngredientCard)
+ getIngredientById(id: Int): Ingredient
ingredientCardDeck
*
IngredientCard
- id : Int
- alchemyMarkerId: Int
-name: String
-value: Int
-color: String
+ test(marker: AlchemyMarker): Boolean
*
ingredientCardDeck
Herb
+ test(marker: AlchemyMarker): Boolean
Mineral
+ test(marker: AlchemyMarker): Boolean
Mushroom
+ test(marker: AlchemyMarker): Boolean
artifactCardDeck
1
ArtifactCardDeck
- artifactCardDeck: ArtifactCard [*]
+ shuffle()
+ getArtifactById(artifactId: Int): ArtifactCard
+ getPriceOfNextArtifact(): int
+ drawCard(): ArtifactCard
artifactCardDeck
*
ArtifactCard
- id : Int
+ price: Int
- type: Enum<ArtifactType>
+ activated: Boolean = false
+ use()
*
artifactCardDeck
potionBrewingArea
1
PotionBrewingArea
- recipes: Map<Ingredient[2], Potion>
+ combine(ingredient1: Ingredient, ingredient2: Ingredient): Potion
Potion
+ id: Int
+ baseIngredients: Ingredient[2]
-
potions
deductionBoard
1
DeductionBoard
- books: Map<String, Ingredient>
+ theories: Theory[*]
- generateTheoryName(): String
+ createTheory(bookName: string, marker: AlchemyMarker)
+ testTheory(name: String): Boolean
+ debunkTheory(name: String): Boolean
theories
*
Theory
+ name: String
+ ingredient: Ingredient
+ marker: AlchemyMarker
+ test(): Boolean
markerStack
1
MarkerStack
+ markers: AlchemyMarker[ ]
+ getMarkerById(markerId: int): AlchemyMarker
markers
*
AlchemyMarker
- id: Int
+ associated: Boolean = false
+ associate()
+ dissociate()

# The Logical Architecture of Our Design

## The Package Diagram



# The GRASP Design Patterns for Each Use Case:

## LOGIN

GRASPs:

1. **Creator**: Auth object is the creator because it is responsible for creation of a new user, using the createUser() method.
2. **Information Expert**:  Auth object is an information expert with the requisite knowledge to authenticate credentials and generate a user to authenticate credentials and generate a user. Player is also an information expert regarding avatars, as well as performing actions such as setAvatar() and markChosen().
3. **Low Coupling**: Actor, typically representing the end-user, demonstrates a state of Low Coupling with the system, since it engages in high-level actions without being dependent on the specifics of user creation or avatar choosing.
4. **High Cohesion**: Each class is dedicated to a distinct set of interconnected responsibilities. Auth class manages authentication, while the Player class manages properties that are specific to players, indicating a high cohesion.
5. **Controller**: TheAlchemistGame is the controller since it manages the login process, overseeing the sequence of steps involved in logging in, starting from acquiring login credentials and ending with user creation.

## GAME SETUP

GRASPs:

1. **Creator**: The Alchemist Game creates the board and the decks.
2. **Information Expert**: Ingredient Card Deck and the Artifact Card Deck hold the ingredient and the artifact information.
3. **Low Coupling**: Ingredient deck only interacts with the players and the board only interacts with the decks. Minimum number of relations between the classes.
4. **High Cohesion**: Decks are only responsible for distributing cards to the players and the game only shuffles and initializes the player stats. The tasks are distributed evenly.
5. **Controller**: Board shuffles cards and makes additions to the inventories according to the message from the Alchemist Game.

**MAKE EXPERIMENT**

GRASPs:

1. **Creator**: Creator is the GameBoard here. Because it creates the PotionBrewingArea (PBA). PBA is also a creator because it creates the potions.
2. **Information Expert**: Ingredient Storage holds the information of the ingredients of a player.
3. **Low Coupling**: Game board only interacts with PBA, PBA only interacts with the game board and the ingredient storage and the storage only interacts with the PBA, so low coupling is achieved.
4. **High Cohesion**: Game board is responsible for only moving to the related area of the game. PBA is only responsible for creation and display of the potion. And the ingredient storage is only responsible for holding the ingredient information.
5. **Controller**: Ingredient storage removes ingredients according to the message from the UI.

**FORAGE INGREDIENT**

GRASPs:

1. **Creator**: Alchemist Game creates the ingredient deck.
2. **Information Expert**: Ingredient deck holds the ingredient information.
3. **Low Coupling**: Alchemist Game only interacts with the ingredient deck, ingredient deck only interacts with the inventory. Low coupling is accomplished.
4. **High Cohesion**: The alchemist is only responsible for making the ingredient deck remove the ingredient card. Ingredient deck is only responsible for sending the card name and inventory is only responsible for adding the card to the inventory.
5. **Controller**: Inventory adds ingredients to the player's inventory according to the message from the UI.

**TRANSMUTE INGREDIENT**

GRASPs:

1. **Creator**: The alchemist game creates the inventory.
2. **Information Expert**: Inventory holds the ingredient information of the player.
3. **Low Coupling**: Game only interacts with the inventory, inventory interacts with the game and the ingredient deck. Low coupling is achieved.
4. **High Cohesion**: Inventory only makes ingredient deck add the ingredient back and the game makes inventory remove the ingredient. The tasks are distributed clearly.
5. **Controller**: Player removes the ingredient from its storage and gains gold according to the message from the inventory.

**SELL POTION**

GRASPs:

1. **Creator**: Game board creates the inventory.
2. **Information Expert**: Inventory holds the potion information.
3. **Low Coupling**: Inventory only interacts with the game. Minimum interaction between the classes.
4. **High Cohesion**: Inventory is only responsible for adding gold and removing potion. Tasks are distributed clearly.
5. **Controller**: Inventory changes its amount of gold according to the message from the UI.

**ACTIVATE IMMEDIATE ARTIFACT EFFECT**

GRASPs:

1. **Creator**: ArtifactDeck is in charge of creating instances of ArtifactCard, and places them in the inventory.
2. **Information Expert:** ArtifactDeck class is an information expert for maintaining the inventory of artifacts.
3. **Low Coupling:** Player has low coupling with the rest of the system as it only communicates with the ArtifactDeck through basic method calls. It is not involved in the internal processes.
4. **High Cohesion:** AlchemistGameBoard is characterized by high cohesion, since it seems to have the sole responsibility of confirming effect applications using confirmEffectActivation() method.
5. **Controller:** ArtifactCard is acting as a controller, overseeing the implementation of artifact effects. It is responsible for implementing the effects and regulating the progression of this action.

**STORE AND USE THE LIMITED ARTIFACT**

GRASPs:

1. **Creator**: ArtifactDeck is in charge of creating instances of ArtifactCard, and places them in the inventory.
2. **Information Expert:** ArtifactDeck class serves as the Information Expert for effectively handling the artifacts. Its responsibility is to retain the artifact card (storeArtifactInDeck()) and place the card into the deck (insertCardIntoDeck()).
3. **Low Coupling**: Player demonstrates a low level of coupling, as they interact with the system by making only a few straightforward method calls, such as validateArtifact() and activateArtifactEffect().
4. **High Cohesion:** AlchemistGame exhibits a strong sense of High Cohesion, prioritizing the orderly progression of gaming events while avoiding any unnecessary involvement with unrelated tasks.
5. **Controller:** AlchemistGame class functions as a Controller, overseeing the progression of the game. The artifact is validated using the validateArtifact() function, then it is initiated using the useArtifact() function, and finally, the artifact's effect is activated using the activateArtifactEffect() function.

**ACTIVATE THE PERSISTENT ARTIFACT EFFECT**

GRASPs:

1. **Creator:** ArtifactDeck is in charge of creating instances of ArtifactCard, and places them in the inventory.
2. **Information Expert:** AlchemistGame class serves as the Information Expert, since it possesses the necessary information to select and activate the enduring impact of the artifact (chooseCard and activatePersistentEffect).
3. **Low Coupling:** Player exhibits Low Coupling by engaging with the system through uncomplicated messages such as acquirePersistentArtifact, without requiring knowledge of the intricacies of artifact handling and effect execution.
4. **High Cohesion:** ArtifactDeck has high cohesion, because it only deals with the management of artifacts. (storeArtifactInDeck() and addCardToDeck().
5. **Controller:** AlchemistGame plays the Controller role, since it oversees the game’s progression, especially acquirePersistentArtifact() and activatePersistentEffect().

**PUBLISH A THEORY**

GRASPs:

1. **Creator**: TheAlchemistGame is the creator since it creates the other instances and classes, and it is a more general class.
2. **Information Expert**:
   - Board class serves as an Information Expert to the method publishATheory(player1), as it possesses the necessary expertise to handle the process of publishing theories. - PublicationTrack is the Information Expert that manages PublicationCards and is responsible for providing the available cards using the getAvailableCards() method.
   - Inventory class possesses knowledge of IngredientCards, hence establishing it as the Information Expert of accessing an ingredient through the getIngredient(id) method.

3. **Low Coupling**: The player maintains a state of Low Coupling with the system, only triggering high-level activities. publishTheory(player), chooseIngredient(), and pickAlchemyMarker() require no knowledge of the specific implementation details of these processes.
4. **High Cohesion**: Each class has a well specified range of obligations: :Board for the publication of theories, :PublicationTrack for the management of publishing cards, and :Inventory for the management of ingredients, indicating a high level of cohesion.
5. **Controller**: TheAlchemistGame is the controller because it oversees the intricate process of disseminating a theory, which entails multiple stages and engagements with various components of the system.

**DEBUNK A THEORY**

**GRASPs:**

1. **Creator**: The alchemist game creates the publication area.
2. **Information Expert**: Player is the information expert that holds the reputation information of the players.
3. **Low Coupling**: The player maintains a state of Low Coupling with the system, only triggering high-level activities such as debunkTheory and adjusting reputation. Interactions are minimal between classes.
4. **High Cohesion**: Each class has a well specified range of obligations: :The alchemist game for the debunk of theories. Player for adjusting reputations.

**Controller**: Player is the controller, it adjusts the reputation points according to the message coming from the UI.

## The Implementation of Design Patterns (GoF Design Patterns)

We have implemented 2 design patterns within our code:

- Singleton Pattern
- Observer Pattern

### Singleton Pattern Implementation

We implemented Singleton Pattern in the Router class. There, as there can be only 1 instance of Router, we implemented the Singleton Pattern in such a class.

In addition, TheAlchemistGame is our controller (in terms of GRASP patterns) which enables us to connect the UI (View) to our domain (Model). In such a scenario, implementing Singleton pattern though the Router class is reasonable to prevent the possible conflicts that can occur in the interaction between our UI (View) and our domain (Model).

### Observer Pattern Implementation

In terms of Observer Pattern, we have implemented a ICurrentUserListener interface. In such a way, normally our Domain (Model) does not normally interact with the UI (View). However, when a change happens, we can inform our UI for it to act accordingly.

There, our VInventory class implements our ICurrentUserListener interface. There, TheAlchemistGame object adds VInventory into our currentUserListeners. There, when the domain intends to inform the UI, it also sends a message mentioning "I ınvented observer pattern

d'dem", where we also can observe that we are using the Observer pattern. Therefore, in the VInventory class we implemented the onCurrentUserChangeEvent method as the following:

```
@Override
    public void onCurrentUserChangeEvent() {
        // TODO Auto-generated method stub
        System.out.println("I invented observer pattern d'dem");
    }
```

Moreover, as we have implemented the toggleCurrentUser method in the Auth class, the currentUserListeners listen to the Auth class. There, in the Auth class we keep the list of currentUserListeners:

```
public class Auth {
    public ArrayList<Player> players = new ArrayList<>(2);
    public int currentUser = 0;
    private ArrayList<ICurrentUserListener> currentUserListeners = new ArrayList<>();
```

To add currentUserListeners, we implemented a method in the Auth class:

```
// Method for observer pattern
    public void addCurrentUserListener(ICurrentUserListener currentUserListener){

        currentUserListeners.add(currentUserListener);
    }
```

Also, in the Auth class we implemented a method to publish currentUserListeners:

```
// Method for observer pattern
    public void publishCurrentUser(){
        for(ICurrentUserListener listener: currentUserListeners){
            listener.onCurrentUserChangeEvent();
        }
    }
```

As a whole, we integrated the observer pattern into our design to inform our UI (View) with the changes occurring in our domain layer (Model).

## General Design & Git Usage

Throughout our general design in our code, we followed our path along the class diagram that we have implemented. There, when any changes have occurred, we integrated such changes to our class diagram to prevent any confusion.

Further, we used Git not only to assign responsibilities to our members equally, but also to co-operate in a team spirit. We first divided the workload to our members and put out issues on Git. There, we could not assign issues to multiple people due to the pricing strategy of Git. However, to overcome that problem, we included our members' names in the description of issues. Further, we developed our codes in separate branches where we elaborated on each branch as a team to let every member understand each segment of the code.

Additionally, we are expecting to merge our code in a branch called "development"; then at the last step, merge into the final branch "main" to minimize confusion and merging errors.

## The Path of Our Project

In our project, we followed such stages:

a. Design a class diagram to simply our coding experience and solidify our knowledge upon the Ku Alchemists game.

b. Assign each member responsibilities through Git with opening issues and weekly meetings.

c. Each member group working in separate branches.

d. Finalizing each branch with the entire team elaborating on it before the merging stage.

e. Merging our branches in a single branch to finalize our design.