SWEN30006 Project 2

Workshop16, Team 9

Jiaheng Dong 1166436

Hanyi Gao 1236476

WAI KIN WILKIN CHOW 1068161

(3-member team)

Part 1: Rearrangement of the original code

1.1 PropertiesLoader

The system should be made more configurable through a property file. According to the high cohesion and information expert principle, we decide to make a properties loader class to take the responsibility to read in target files.
In this case, the game setting will be initialized by Oh_Heaven class through reading in property files (Figure 1).

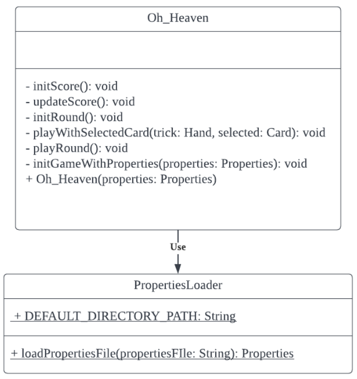

Oh_Heaven
- initScore(): void
- updateScore(): void
- initRound(): void
- playWithSelectedCard(trick: Hand, selected: Card): void
- playRound(): void
- initGameWithProperties(properties: Properties): void
+ Oh_Heaven(properties: Properties)
Use
PropertiesLoader
+ DEFAULT_DIRECTORY_PATH: String
+ loadPropertiesFile(propertiesFIle: String): Properties

Figure 1. Display the relationship between Oh_Heaven class and the PropertiesLoader class

1.2 GameRound

The Oh_Heaven class (Figure 2) handles all the interactions and game running rules in the original game design. Therefore, it leads to high coupling and low cohesion, which could be complicated when the specific game rules need to be modified. According to the GRASP information expert principle, the Oh_Heaven class is responsible to handle the game GUI settings and general game running logic. In this case, the responsibility of handling the specific game logic and rules should be delivered to another class. The GameRound class (Figure 2) is created to manage this purpose, the moves like initialize round, set players' hand cards, start lead and start after lead will be handled here. Moreover, the data of each player, such as scores, tricks, and bids will be stored in GameRound class, like a common information pool, which just is used to correctly run the game, so it will be easier to modify the data structure in the future.

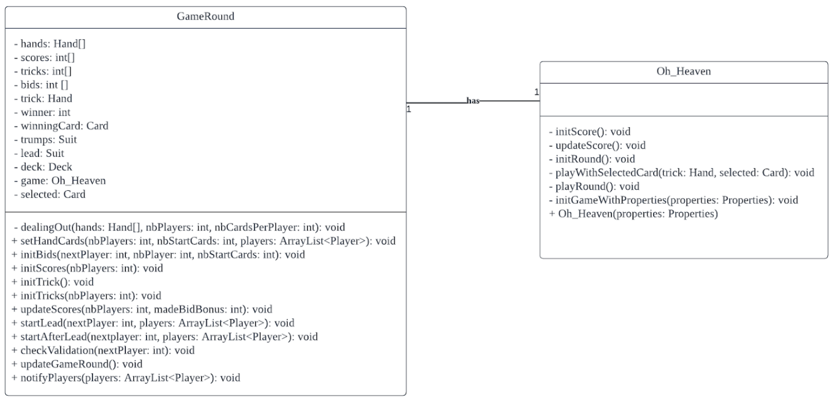

GameRound
- hands: Hand[]
- scores: int[]
- tricks: int[]
- bids: int []
- trick: Hand
- winner: int
- winningCard: Card
- trumps: Suit
- lead: Suit
- deck: Deck
- game: Oh_Heaven
- selected: Card
- dealingOut(hands: Hand[], nbPlayers: int, nbCardsPerPlayer: int): void
+ setHandCards(nbPlayers: int, nbStartCards: int, players: ArrayList<Player>): void
+ initBids(nextPlayer: int, nbPlayer: int, nbStartCards: int): void
+ initScores(nbPlayers: int): void
+ initTrick(): void
+ initTricks(nbPlayers: int): void
+ updateScores(nbPlayers: int, madeBidBonus: int): void
+ startLead(nextPlayer: int, players: ArrayList<Player>): void
+ startAfterLead(nextplayer: int, players: ArrayList<Player>): void
+ checkValidation(nextPlayer: int): void
+ updateGameRound(): void
+ notifyPlayers(players: ArrayList<Player>): void
1
has
1
Oh_Heaven
- initScore(): void
- updateScore(): void
- initRound(): void
- playWithSelectedCard(trick: Hand, selected: Card): void
- playRound(): void
- initGameWithProperties(properties: Properties): void
+ Oh_Heaven(properties: Properties)

Figure 2. Display the relationship between GameRound class and Oh_Heaven class, and the methods which are used to fit the design purpose.

The Oh_Heaven class logically includes each game round, so it has the ability to create GameRound class based on the GRASP creator pattern (Figure 2).

1.3 Player

According to the game rules, there will be generally two types of players, which are the human players and NPC players. When handling alternatives based on type, we decide to follow the GRASP polymorphism pattern, which suggests we assign responsibility for the behavior by using polymorphic operations to the type for which the behavior varies (Figure 3). Moreover, the Oh_Heaven class will read in the initializing type of each player, so the Oh_Heaven class will take the responsibility of creating the players based on the GRASP creator pattern (Figure 3).

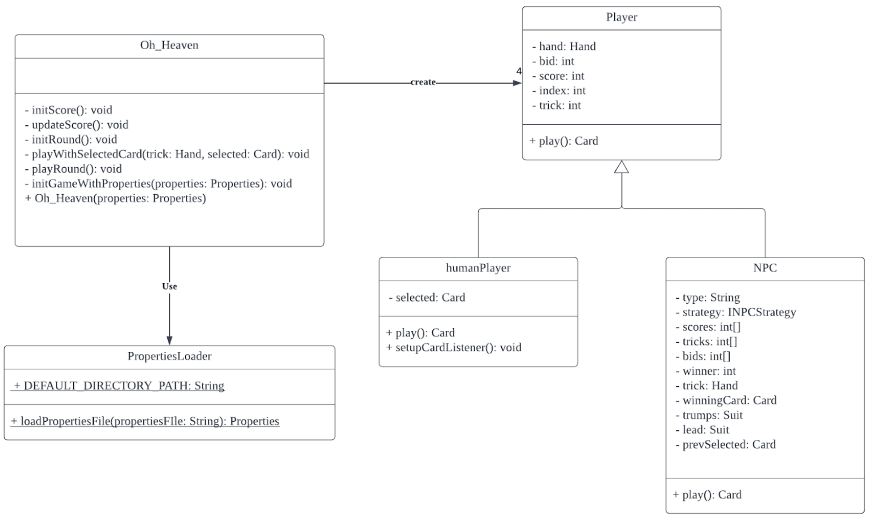


Figure 3. Display the relationship between Oh_Heaven class and Player class, and the polymorphism pattern of the player class

1.4 CardUtil and RandomUtil

In our design, we consider the basic game setting of cards and random methods might need to be changed in the future. In the real world, the card game might not need four suits, and the rank of the card is different in different games. Therefore, we separate the card class and random methods from the Oh_Heaven based on the GRASP high cohesion and low coupling principles (Figure 4).

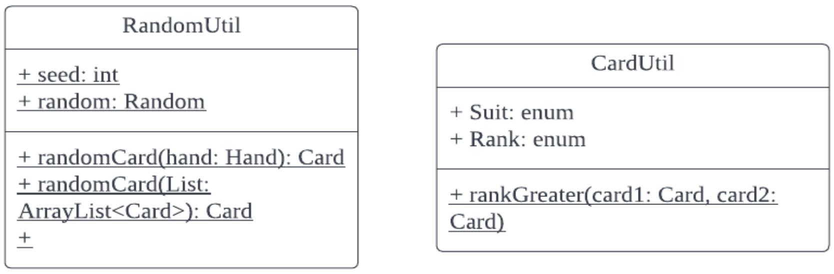

RandomUtil
+ seed: int
+ random: Random
+ randomCard(hand: Hand): Card
+ randomCard(List: ArrayList<Card>): Card
+
CardUtil
+ Suit: enum
+ Rank: enum
+ rankGreater(card1: Card, card2: Card)

Figure 4. Display the random class and card class

Part 2: GoF patterns in the current design

2.1 Observer Pattern

To ensure the fairness and security of the game, instead of making all players access a shared information pool, we create personal information pools for all NPC players. In this case, the transparent information, such as the scores, tricks, bids, trick, winner, and

winner cards will be stored in each NPC player. They can access their own information pool to apply different strategies.
Due to this decision, the NPC players act like observers which need to know the latest news on the game board, and the GameRound class works like an observable subject, which has been subscribed to by the players. In this case, we use the GoF observer pattern (Figure 5), the GameRound has to notify all NPC players when there is any new information that appears, and the information in each NPC player will be updated automatically.

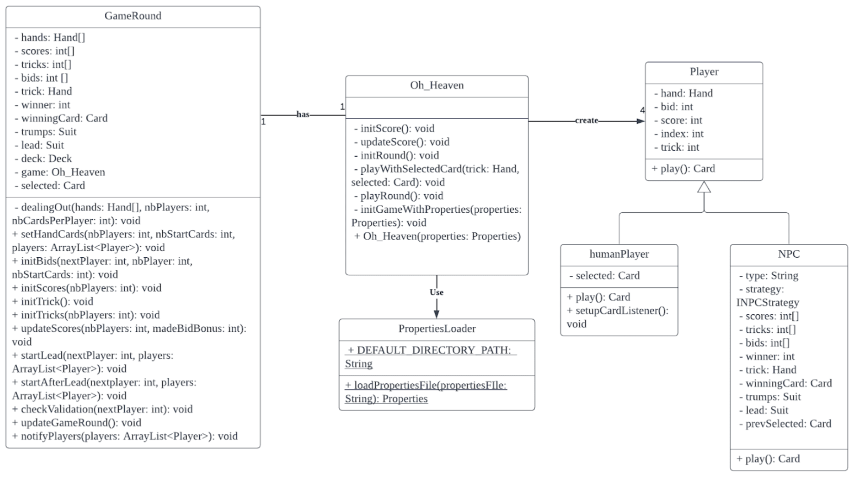
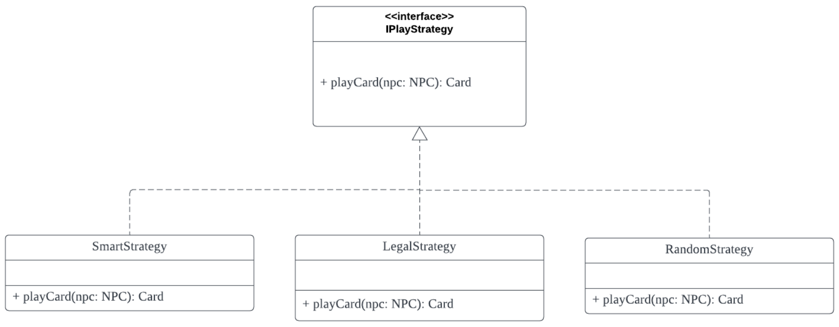

GameRound
- hands: Hand[]
- scores: int[]
- tricks: int[]
- bids: int []
- trick: Hand
- winner: int
- winningCard: Card
- trumps: Suit
- lead: Suit
- deck: Deck
- game: Oh_Heaven
- selected: Card
- dealingOut(hands: Hand[], nbPlayers: int, nbCardsPerPlayer: int): void
+ setHandCards(nbPlayers: int, nbStartCards: int, players: ArrayList<Player>): void
+ initBids(nextPlayer: int, nbPlayer: int, nbStartCards: int): void
+ initScores(nbPlayers: int): void
+ initTrick(): void
+ initTricks(nbPlayers: int): void
+ updateScores(nbPlayers: int, madeBidBonus: int): void
+ startLead(nextPlayer: int, players: ArrayList<Player>): void
+ startAfterLead(nextplayer: int, players: ArrayList<Player>): void
+ checkValidation(nextPlayer: int): void
+ updateGameRound(): void
+ notifyPlayers(players: ArrayList<Player>): void
1
has
1
Oh_Heaven
- initScore(): void
- updateScore(): void
- initRound(): void
- playWithSelectedCard(trick: Hand, selected: Card): void
- playRound(): void
- initGameWithProperties(properties: Properties): void
+ Oh_Heaven(properties: Properties)
create
4
Player
- hand: Hand
- bid: int
- score: int
- index: int
- trick: int
+ play(): Card
Use
PropertiesLoader
+ DEFAULT_DIRECTORY_PATH: String
+ loadPropertiesFile(propertiesFIle: String): Properties
humanPlayer
- selected: Card
+ play(): Card
+ setupCardListener(): void
NPC
- type: String
- strategy: INPCStrategy
- scores: int[]
- tricks: int[]
- bids: int[]
- winner: int
- trick: Hand
- winningCard: Card
- trumps: Suit
- lead: Suit
- prevSelected: Card
+ play(): Card

Figure 5. Display the Observer Pattern

2.2 Strategy Pattern

Different players have different types. Other than the interactive player, the NPC player can be “smart”, “legal” or “random". NPC players of different types have different strategies to choose the card to play. Therefore, we use the strategy pattern to define different strategies(algorithms) in different classes with a common interface. Using the strategy pattern lets the clients get a simple interface to execute different strategies and we can easily switch them at runtime. The strategy pattern also reduces the dependencies of different playing strategies from the rest of the code, which coincides with the **low coupling** and **high cohesion**. The class diagram of the strategy pattern in this project is shown in the figure below. The **SmartStrategy** class, the **LegalStrategy** class, and the **RandomStrategy** class are designed for the “smart” NPC, the “legal” NPC, and the “random” NPC respectively. They have a common interface called **IPlayStrategy** with a method **playCard**. This method returns a Card type, indicating which card needs to be played.



<<interface>>
IPlayStrategy
+ playCard(npc: NPC): Card
SmartStrategy
+ playCard(npc: NPC): Card
LegalStrategy
+ playCard(npc: NPC): Card
RandomStrategy
+ playCard(npc: NPC): Card

Figure 6. Display the strategy pattern

2.3 Factory Pattern & Singleton Pattern

Although the strategy pattern has been designed, it is worth thinking about how to create different strategy classes. We need to know which specific strategy should be used and created in the run time. With more player types and strategies in the future, the creation logic could be more complex. It is better to hide the creation logic from the NPC class and use a cohesive class to create the strategy object. Therefore, the Factory Pattern should be considered. We use a **Pure Fabrication** class called ***NPCStrategyFactory*** to handle the creation for better cohesion. There is a method called ***getStrategy*** in this Factory class to create a certain Strategy object based on the NPC the method receives as the input argument. As a result, the NPC class does not need to know how to create the strategy because the creation logic is in the factory class. The sequence diagram of the creation logic is shown in the figure below.

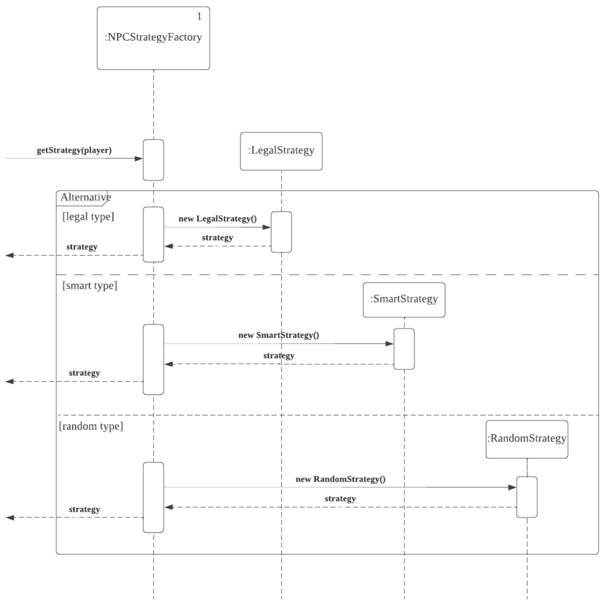


Figure 7. Sequence diagram of strategy creation in the Factory class

Besides, the **Singleton** pattern is used along with the Factory pattern. Since accessing the Factory by initializing the object with a reference adds a lot of complexity, we need to provide a global access point while ensuring the Factory class has only one instance. The way to do that is to define a static method ***getInstance*** in ***NPCStrategyFactory*** that returns the instance of its own class. And this static method is the only way of getting the Singleton Factory object.
The class diagram of the Factory pattern and the Singleton pattern is shown in the figure below. We can see that the NPC class has the strategy attribute. However, instead of creating the strategy in its own class, it uses the NPCStrategyFactory to create the strategy. And the NPC class creates and accesses the NPCStrateyFactory instance by using the static getInstance method, not through a constructor and a getter method.

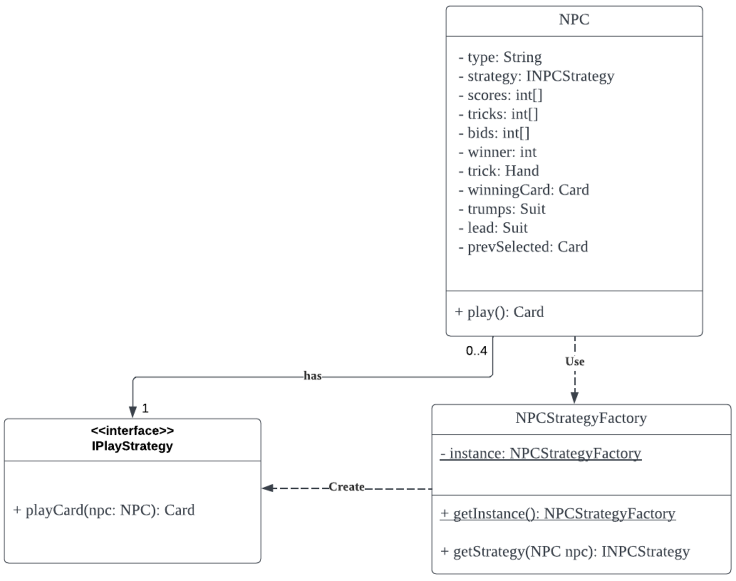


Figure 8. Display the Factory pattern and the Singleton pattern

Part 3: Future implementations

3.1 Smart strategy

The Smart strategy used in the game is to play the small card first and only win the game with the smallest card. So the opponent will waste their big card in a game that they could not win. Therefore, the smart NPC can maximize its winning rate.

3.2 Composite strategy pattern in the future

Based on our design of NPC types, we just make the NPC use a different strategy to implement a new type. In the future design, we consider the NPC could also use the composite pattern to combine more than two strategies to create a new NPC type. For example, if we need a semi-smart player, we can combine smart strategy and legal strategy in the composite strategy class.

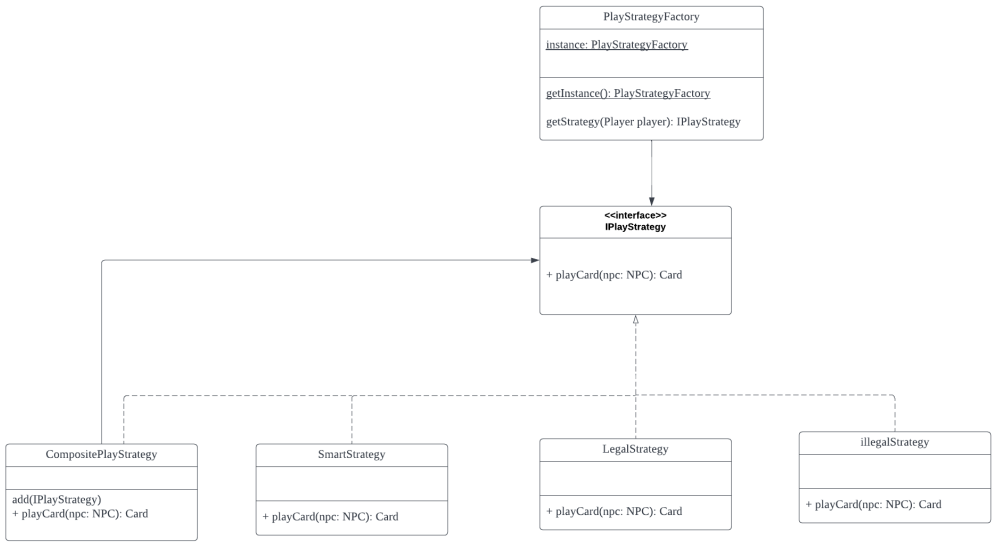

PlayStrategyFactory
instance: PlayStrategyFactory
getInstance(): PlayStrategyFactory
getStrategy(Player player): IPlayStrategy
<<interface>>
IPlayStrategy
+ playCard(npc: NPC): Card
CompositePlayStrategy
add(IPlayStrategy)
+ playCard(npc: NPC): Card
SmartStrategy
+ playCard(npc: NPC): Card
LegalStrategy
+ playCard(npc: NPC): Card
illegalStrategy
+ playCard(npc: NPC): Card

Figure 9. The composite pattern in the future

However, the **Decorator** pattern is not suitable for our design. We treat the composite class, which is the combination of play strategies as our primary focus, there is no need to embellish the single play strategy. Moreover, according to the game rules, the player type will be static during the whole game after being initialized. In this case, there are no requirements for changing playing card strategies for each NPC during the game period.

3.3 How to extend bid behavior in the future

In the current gaming system, the bid of each player is made randomly by the system. However, in the future version, NPC players may have the option to make the bid based on a certain algorithm. Therefore, a new Strategy pattern for bidding can be designed in the future. For example, there will be a random bid strategy and a smart bid strategy (Figure 10). Using the strategy pattern lets the clients get a simple interface to execute different strategies and keep the code with low coupling and high cohesion since it can reduce the dependencies with other parts of the code.

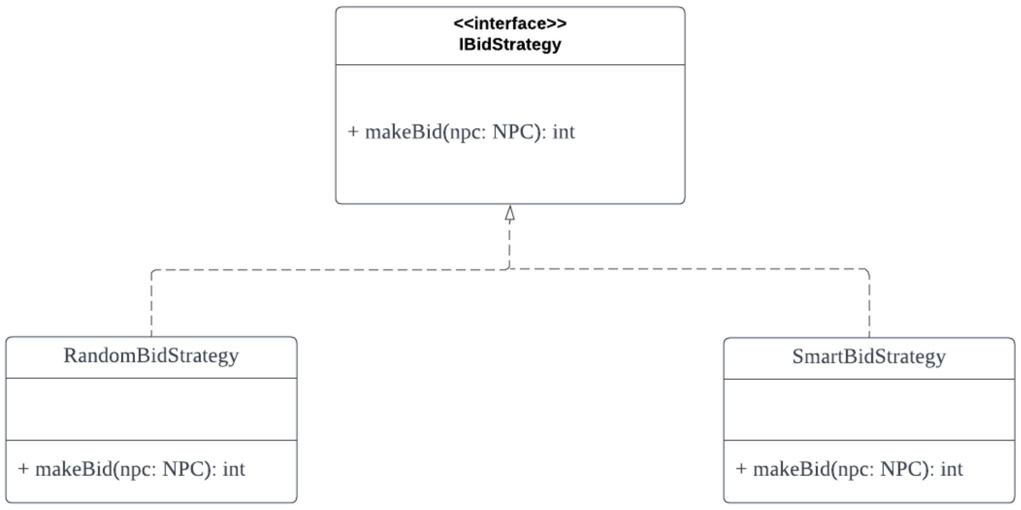
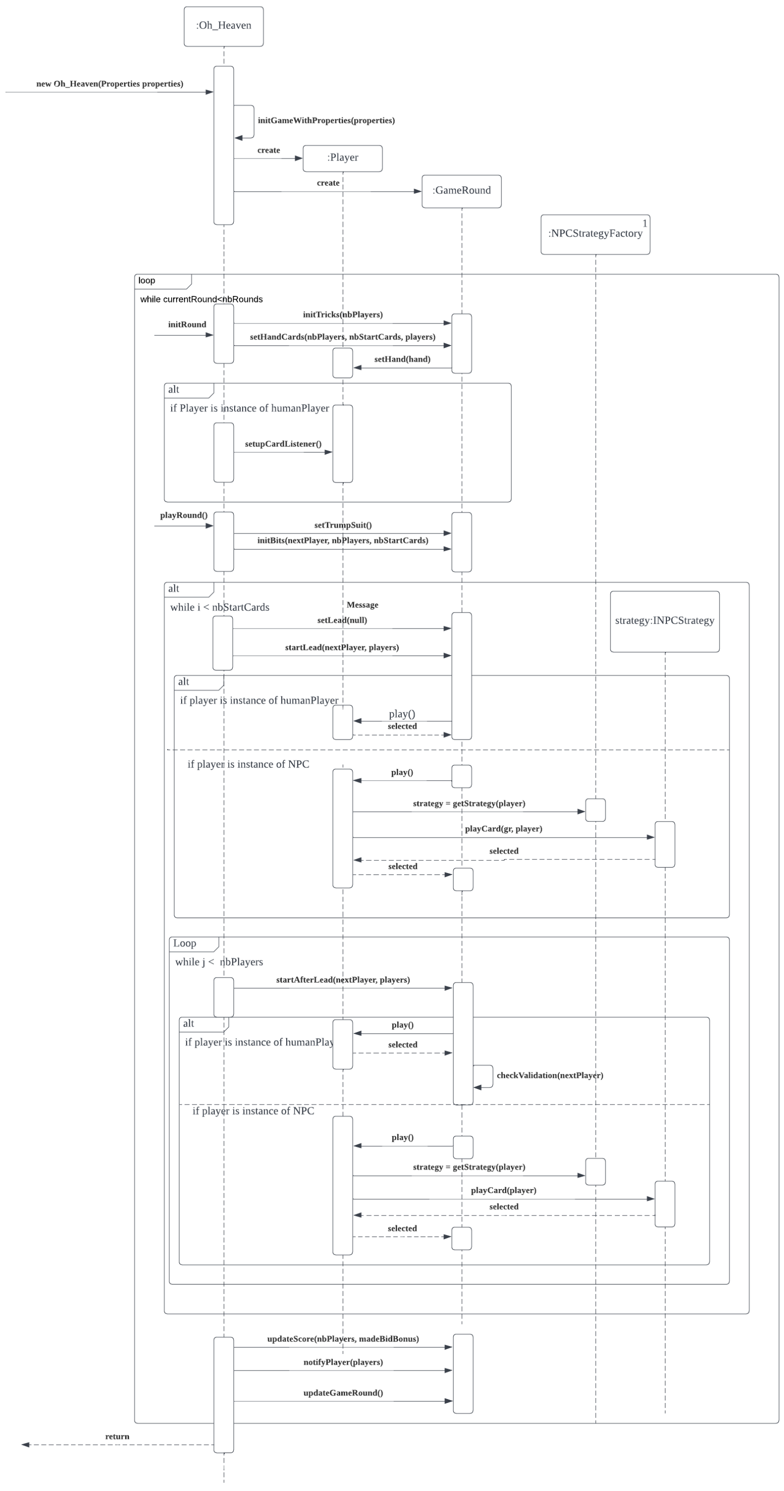


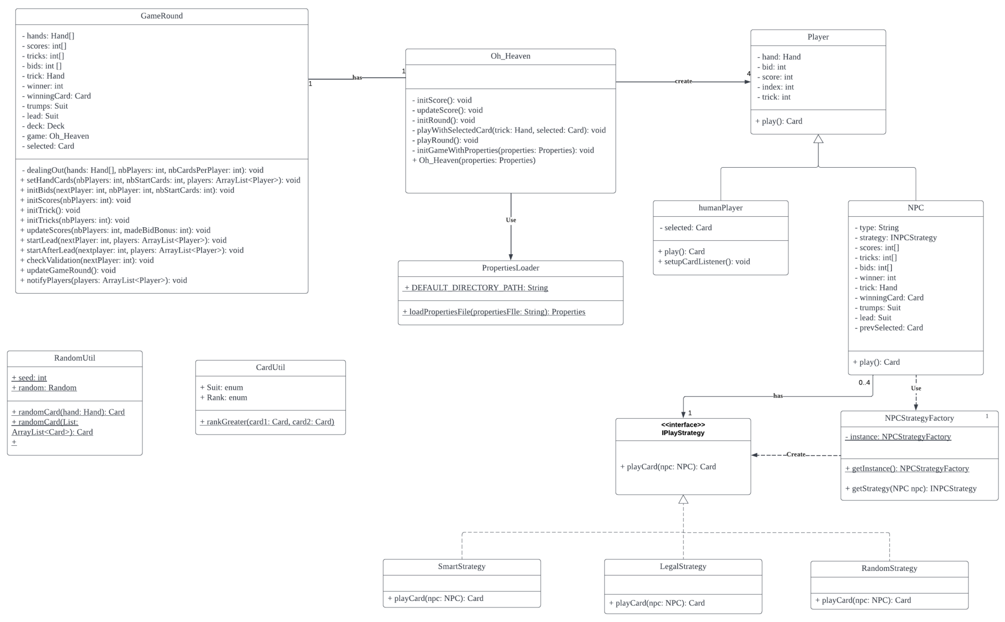
Figure 10. The Strategy pattern for bidding in the future

Appendix

1. Sequence diagram of the system


:Oh_Heaven
new Oh_Heaven(Properties properties)
initGameWithProperties(properties)
create
:Player
create
:GameRound
:NPCStrategyFactory
1
loop
while currentRound<nbRounds
initTricks(nbPlayers)
initRound
setHandCards(nbPlayers, nbStartCards, players)
setHand(hand)
alt
if Player is instance of humanPlayer
setupCardListener()
playRound()
setTrumpSuit()
initBits(nextPlayer, nbPlayers, nbStartCards)
alt
while i < nbStartCards
Message
setLead(null)
strategy:INPCStrategy
startLead(nextPlayer, players)
alt
if player is instance of humanPlayer
play()
selected
if player is instance of NPC
play()
strategy = getStrategy(player)
playCard(gr, player)
selected
selected
Loop
while j < nbPlayers
startAfterLead(nextPlayer, players)
alt
play()
if player is instance of humanPlay
selected
checkValidation(nextPlayer)
if player is instance of NPC
play()
strategy = getStrategy(player)
playCard(player)
selected
selected
updateScore(nbPlayers, madeBidBonus)
notifyPlayer(players)
updateGameRound()
return

2. Class diagram of the system


GameRound
- hands: Hand[]
- scores: int[]
- tricks: int[]
- bids: int []
- trick: Hand
- winner: int
- winningCard: Card
- trumps: Suit
- lead: Suit
- deck: Deck
- game: Oh_Heaven
- selected: Card
- dealingOut(hands: Hand[], nbPlayers: int, nbCardsPerPlayer: int): void
+ setHandCards(nbPlayers: int, nbStartCards: int, players: ArrayList<Player>): void
+ initBids(nextPlayer: int, nbPlayer: int, nbStartCards: int): void
+ initScores(nbPlayers: int): void
+ initTrick(): void
+ initTricks(nbPlayers: int): void
+ updateScores(nbPlayers: int, madeBidBonus: int): void
+ startLead(nextPlayer: int, players: ArrayList<Player>): void
+ startAfterLead(nextplayer: int, players: ArrayList<Player>): void
+ checkValidation(nextPlayer: int): void
+ updateGameRound(): void
+ notifyPlayers(players: ArrayList<Player>): void
1
has
1
Oh_Heaven
- initScore(): void
- updateScore(): void
- initRound(): void
- playWithSelectedCard(trick: Hand, selected: Card): void
- playRound(): void
- initGameWithProperties(properties: Properties): void
+ Oh_Heaven(properties: Properties)
Use
PropertiesLoader
+ DEFAULT_DIRECTORY_PATH: String
+ loadPropertiesFile(propertiesFIle: String): Properties
create
4
Player
- hand: Hand
- bid: int
- score: int
- index: int
- trick: int
+ play(): Card
humanPlayer
- selected: Card
+ play(): Card
+ setupCardListener(): void
NPC
- type: String
- strategy: INPCStrategy
- scores: int[]
- tricks: int[]
- bids: int[]
- winner: int
- trick: Hand
- winningCard: Card
- trumps: Suit
- lead: Suit
- prevSelected: Card
+ play(): Card
0..4
has
1
Use
<<interface>>
IPlayStrategy
+ playCard(npc: NPC): Card
NPCStrategyFactory
1
- instance: NPCStrategyFactory
+ getInstance(): NPCStrategyFactory
+ getStrategy(NPC npc): INPCStrategy
Create
RandomUtil
+ seed: int
+ random: Random
+ randomCard(hand: Hand): Card
+ randomCard(List: ArrayList<Card>): Card
+
CardUtil
+ Suit: enum
+ Rank: enum
+ rankGreater(card1: Card, card2: Card)
SmartStrategy
+ playCard(npc: NPC): Card
LegalStrategy
+ playCard(npc: NPC): Card
RandomStrategy
+ playCard(npc: NPC): Card