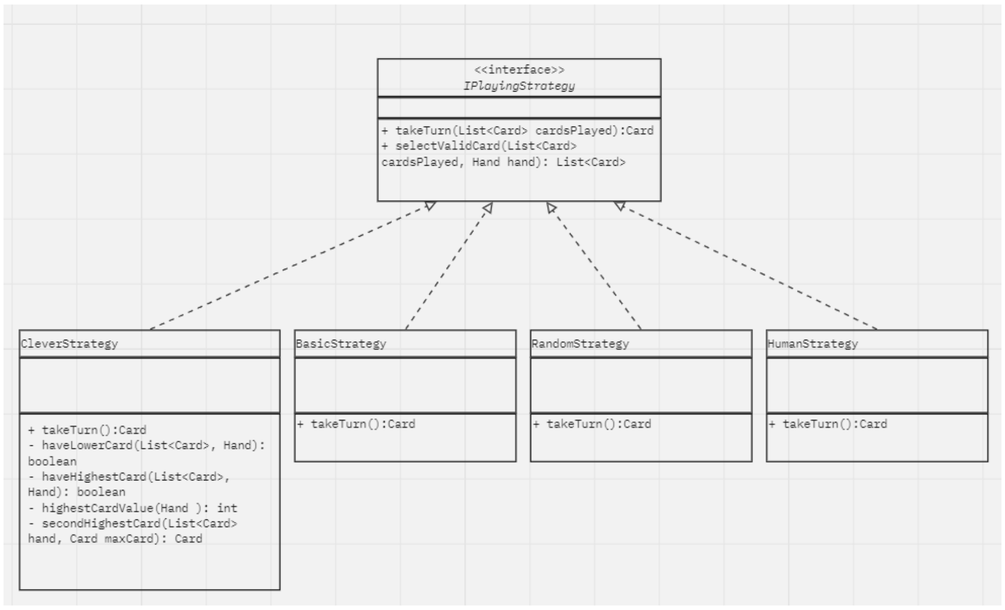
SWEN30006 Project 2

Thursday 14:15 Workshop - Team 12
William Grimsey - 1270044
Even Guo - 1307884
Sophie Su - 1273360

To preserve maintainability of the code base throughout the project, we implemented new features by adding new components to the given framework. Our goal was to implement design strategies outlined below to avoid overloading the original classes with responsibilities which would reduce their cohesion. The design class diagram on page 4 (and in the documentation) represents the whole implementation, where sections below give more detail into each account of modification.

Implementing Player strategies:

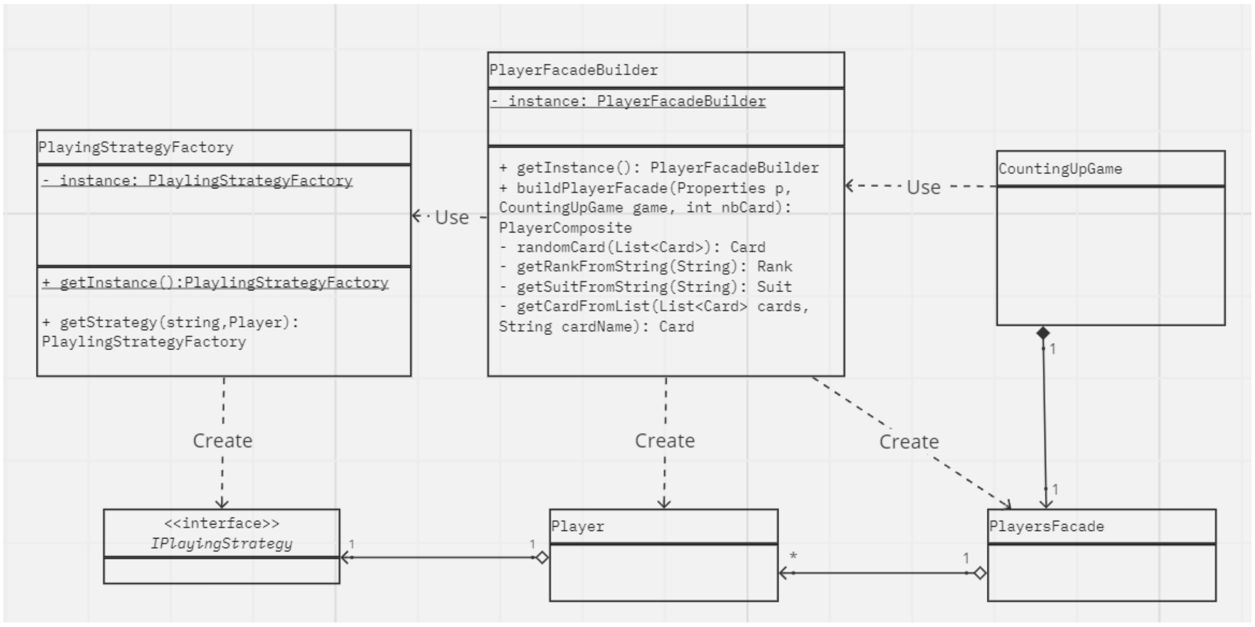
In the creation of the different types of players, we used a number of GRASP design patterns, all of which add to the simplicity of designing the card game:


<<interface>>
IPlayingStrategy
+ takeTurn(List<Card> cardsPlayed):Card
+ selectValidCard(List<Card> cardsPlayed, Hand hand): List<Card>
CleverStrategy
+ takeTurn():Card
- haveLowerCard(List<Card>, Hand): boolean
- haveHighestCard(List<Card>, Hand): boolean
- highestCardValue(Hand): int
- secondHighestCard(List<Card> hand, Card maxCard): Card
BasicStrategy
+ takeTurn():Card
RandomStrategy
+ takeTurn():Card
HumanStrategy
+ takeTurn():Card

To handle the multiple different players that could be created for a game, a strategy interface, **IPlayingStrategy** , is used to encapsulate the different implementations of a player. This interface sets the rule that any player strategy object in the program must have a polymorphic operation takeTurn() which takes their turn based on the necessary strategy. This method takes a List of cards played this round, the player's hand therefore allows for the appropriate player's strategy to be used. It also allows for the total list of cards played to be accessed from the game object as this is necessary for the implementation of the clever player. By conducting it this way, the algorithm is flexible in choosing the types of players at runtime, which is necessary for this game. This interface allows for future implementations to create extra components for players that might be wanted, thus making the design polymorphic and extendable.

An alternative to implement the same set of features can also use inheritance. For example many subclasses(HumanPlayer, RandomPlayer class) can be created to implement different strategies. But since the players preserve all the same behaviors except for the playing strategy. The strategy pattern approach will be better than inheritance. New strategies that may require more information can be implemented without modifying the Player class.

In addition to this interface, a façade pattern is used to create a unified and simplified approach to a list of Player objects. By encapsulating the complex subsystem within a single interface, it creates ease of access by the **CountingUpGame** class and thereby promotes decoupling the subsystem. It takes the responsibility of manipulating players away from **CountingUpGame**, which already contains a lot of information and responsibility which allows for higher cohesion as it keeps the objects focused and manageable.


PlayerFacadeBuilder
- instance: PlayerFacadeBuilder
+ getInstance(): PlayerFacadeBuilder
+ buildPlayerFacade(Properties p, CountingUpGame game, int nbCard): PlayerComposite
- randomCard(List<Card>): Card
- getRankFromString(String): Rank
- getSuitFromString(String): Suit
- getCardFromList(List<Card> cards, String cardName): Card
PlayingStrategyFactory
- instance: PlaylingStrategyFactory
+ getInstance():PlaylingStrategyFactory
+ getStrategy(string,Player): PlaylingStrategyFactory
CountingUpGame
Use
Use
Create
Create
Create
<<interface>>
IPlayingStrategy
Player
PlayersFacade
1
1
1
1
*
1

A builder pattern is used in a flexible way to create the **PlayerFacade** object and a Factory class that creates the strategy. This is a way to create the different players based on the property file from **CountingUpGame**, as the building process is long and complex. This simplifies it to make the construction streamline and manageable, thus providing high cohesion.

To promote low coupling between the necessary information in the **CountingUpGame** and creation of the player's strategy objects, we made use of a Singleton Strategy Factory. This takes the creation away from the **Player** class which needs to interact with it. This increases cohesion by not overloading the **Player** class and, by splitting up the responsibilities, decreases coupling between classes.

Valid card play:

When implementing the checking valid card, we used the **IPlayingStrategy** interface as it already contains the information necessary to complete this task. As it is an information expert, it can use the player's hand and current card on top of the deck information to return a list of valid cards the player can play. Within the different players we can then use this list to pick a card from, rather than the player's whole hand. To follow the principle of protected variation, a final valid card check is also added in the **CountingUpGame** class to prevent wrong cards being played. Preventing incorrect strategy implementation which may cause the game to go wrong.

Clever Player Implementation:

To implement a strategy for the clever player, we decided to consider different situations the player would be in at different moments of the game, and then construct a 'best response' based on this. The two variables we considered are:

1. checkHaveHigherCard(): returns true when the player has every highest card by rank currently in play, and therefore can win the game.

2. haveLowerCard(): returns true when the player can play a card which isn't a highest card by rank currently playable in the game by anyone.

In using this is means we can consider 4 cases and have the clever player react accordingly:

<u>Case A: 1 is True and 2 is True</u>

- Play the lowest card so that the player can gain more points by the time it wins.

<u>Case B: 1 is True and 2 is False</u>

- Win the hand by playing one of the highest cards.

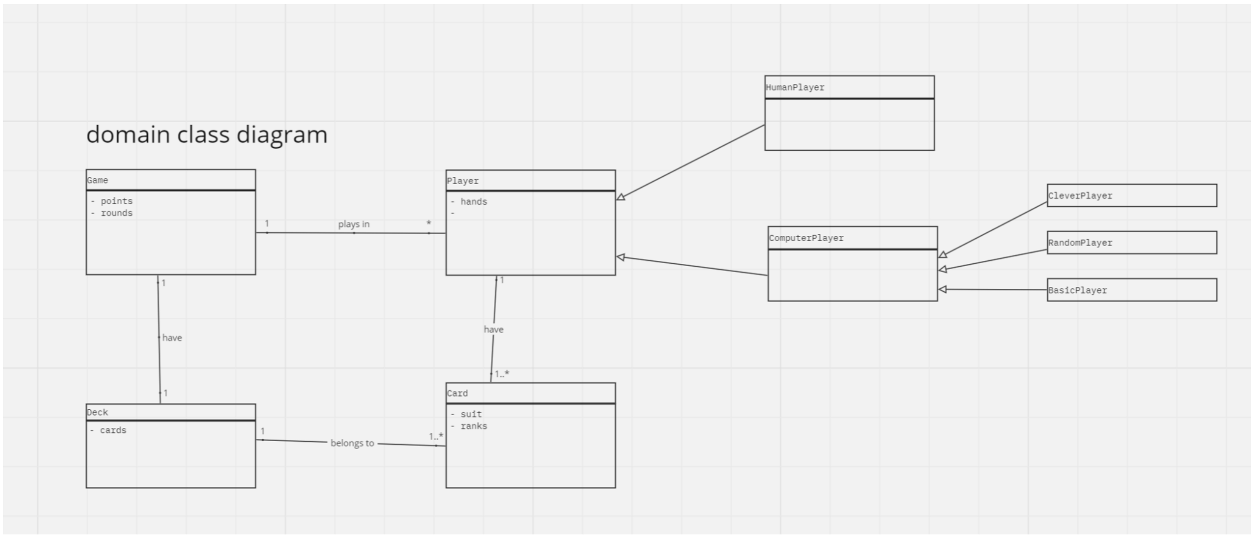
<u>Case C: 1 is False and 2 is True</u>

- Play the second highest card in hand to hold onto a potentially winning high rank card in the next round, as we try to get the other players to play their higher card on top of our card. This prevents a build-up of lower rank cards and forces other players to play potential winning cards.

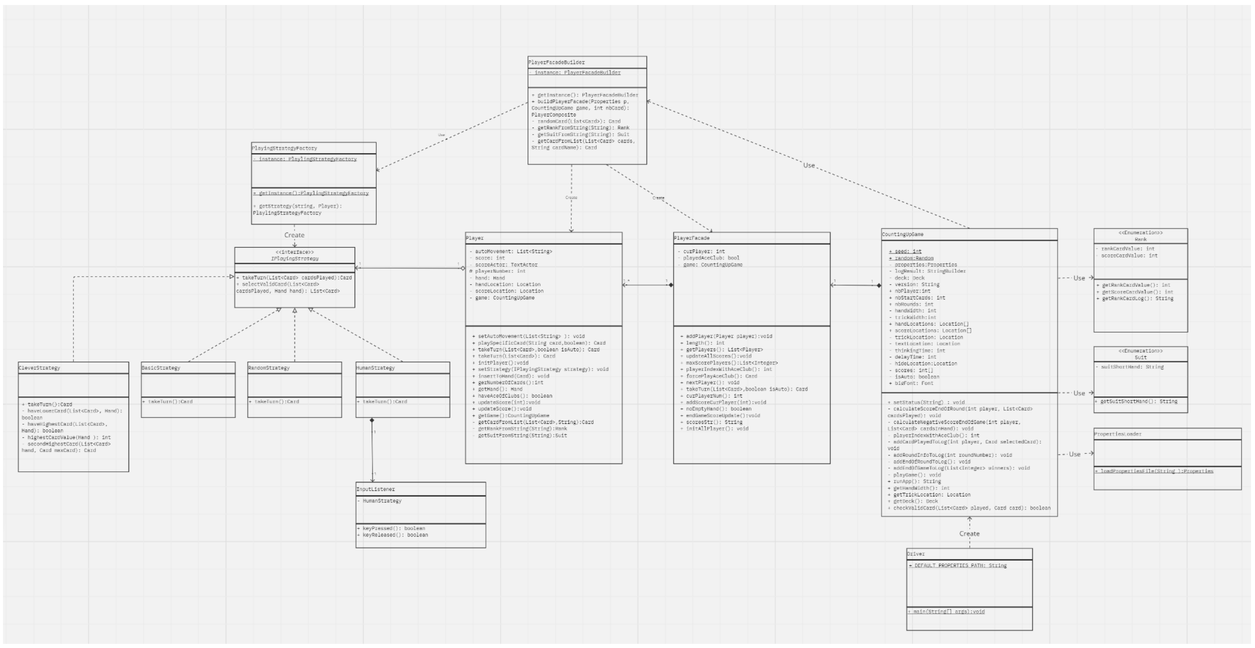
<u>Case D: 1 is False and 2 is False</u>

- Skip (Cannot play anything)

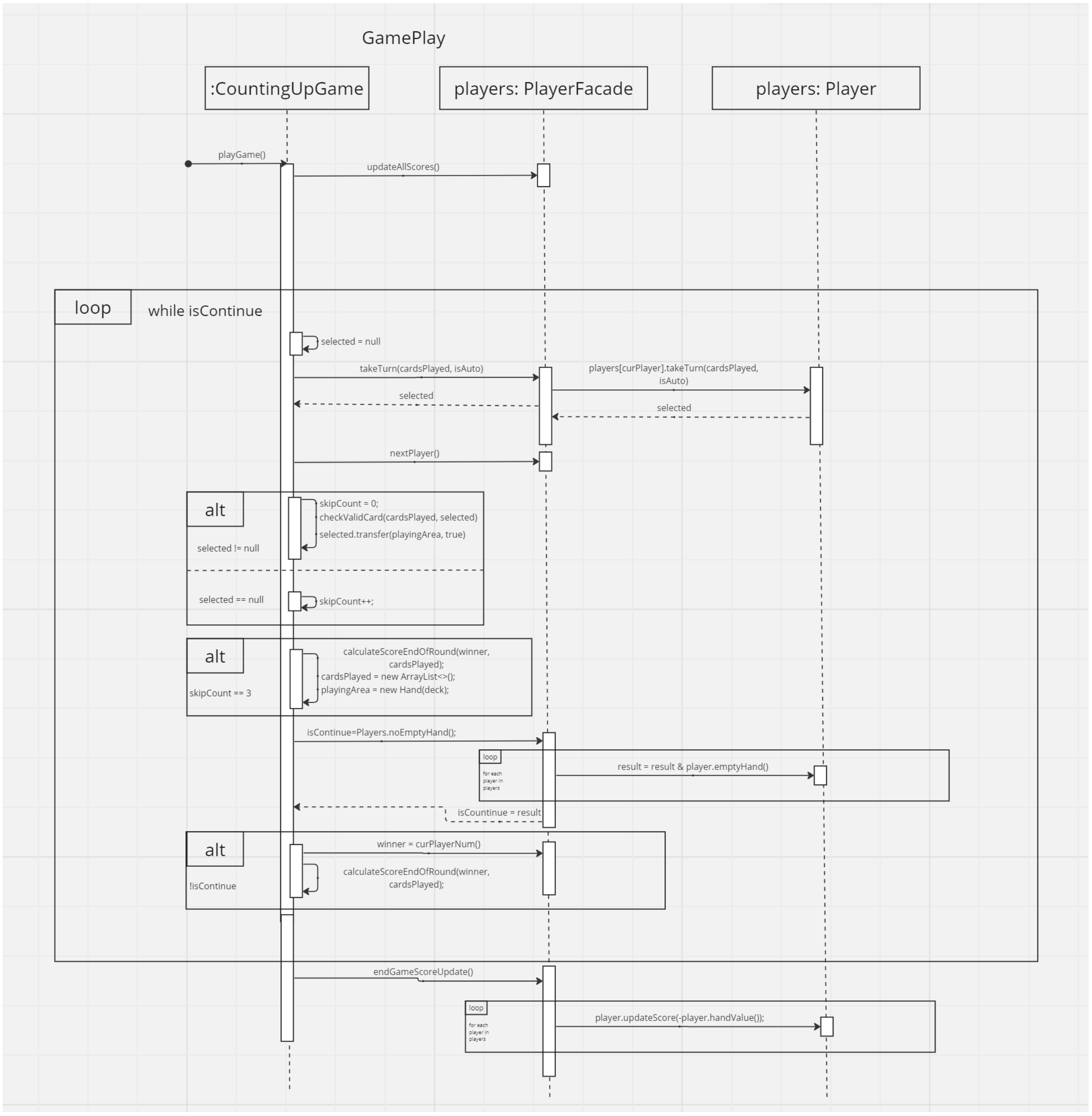
Below is the domain model, followed by the design model, containing the components that we created:



Design Class Diagram



System Sequence Diagram


GamePlay
:CountingUpGame
players: PlayerFacade
players: Player
playGame()
updateAllScores()
loop
while isContinue
selected = null
takeTurn(cardsPlayed, isAuto)
players[curPlayer].takeTurn(cardsPlayed, isAuto)
selected
selected
nextPlayer()
alt
selected != null
skipCount = 0;
checkValidCard(cardsPlayed, selected)
selected.transfer(playingArea, true)
selected == null
skipCount++;
alt
skipCount == 3
calculateScoreEndOfRound(winner, cardsPlayed);
cardsPlayed = new ArrayList<>();
playingArea = new Hand(deck);
isContinue=Players.noEmptyHand();
loop
for each player in players
result = result & player.emptyHand()
isCountinue = result
alt
!isContinue
winner = curPlayerNum()
calculateScoreEndOfRound(winner, cardsPlayed);
endGameScoreUpdate()
loop
for each player in players
player.updateScore(-player.handValue());