SWEN30006 Project 2: Whist – Report

Team 120
Abhilash Misra (1018969)
Fenghe Shen (1058605)
Ishan Goyal (1056051)

The University of Melbourne
2020

1. Introduction

Whist is played with a standard fifty-two card deck with four suits and thirteen ranks in each suit. The program currently supports two types of players: human interactive players and one type of NPC

Functionalities have been added/modified considering the need to facilitate the addition of other future changes:

1. Improved configurability
2. New NPCs, which allow for
3. New Strategies

The following report explains our implemented solution design by describing various changes made to the base structure as well as patterns and principles inculcated in our solution.

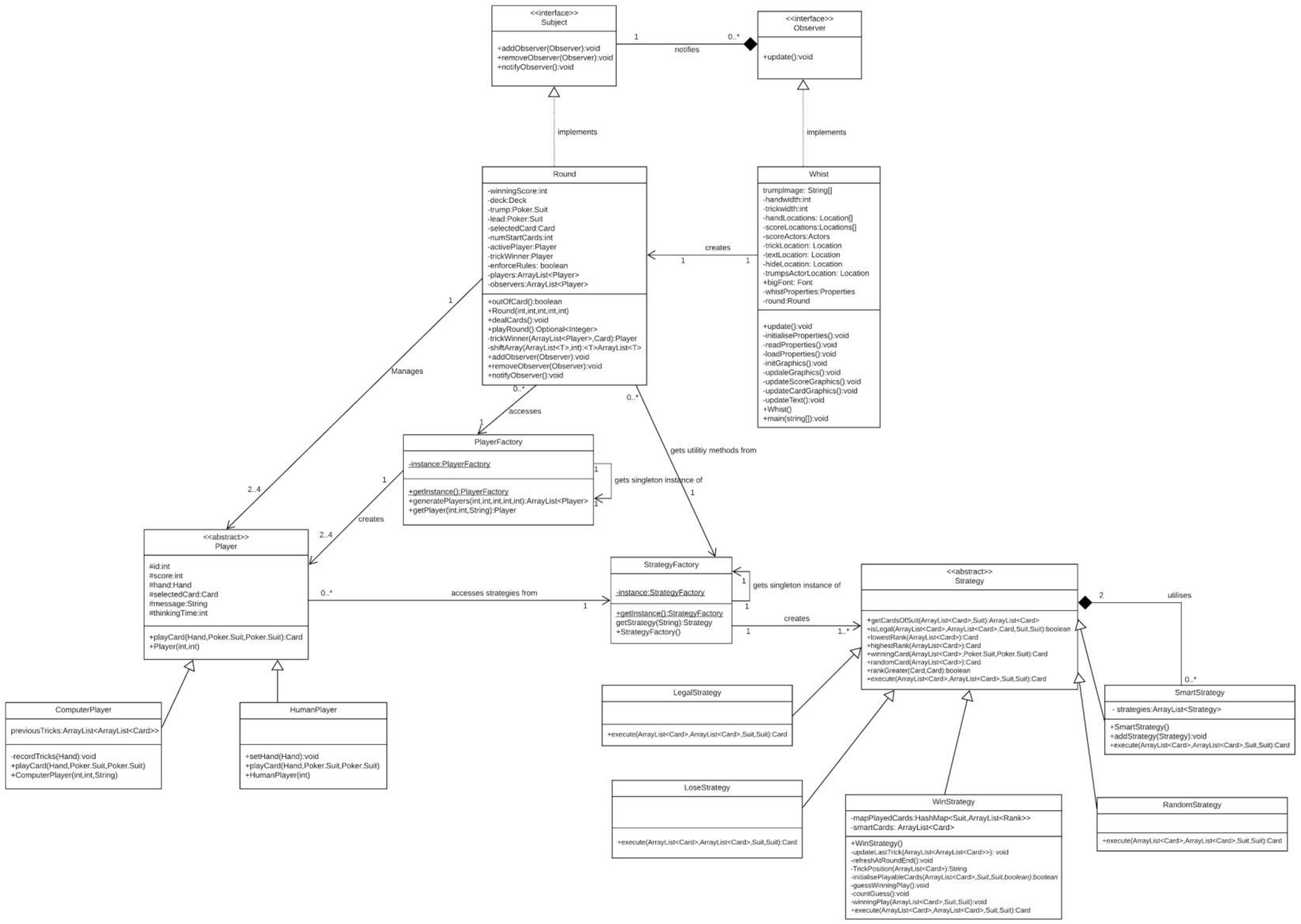
2. Solution Design

`Whist` class handles the initialisation of the game properties and the graphic rendering logic.

`Round` class is extracted from the base Whist class and consists of all gameplay-related elements of each round of the game. Its major responsibility is to execute a modified `playRound`*()* while simultaneously notifying the Whist class of the occurred changes to refresh the graphics.

`Player` Class allows us to define playable and non-playable characters (NPCs), each retaining its own hand of cards as well as other relevant information such as score. Their turns to play are managed by **`Round`** class

`Strategy` Class defines the strategic algorithms of card-play which are used by various NPCs. The child class **`SmartStrategy`** (inherits properties from **`Strategy`**) is a composite Strategy class which combines multiple strategies to implement a Smart strategy for the Smart NPC.



<<interface>>
Subject
+addObserver(Observer):void
+removeObserver(Observer):void
+notifyObserver():void
1
notifies
0..*
<<interface>>
Observer
+update():void
implements
implements
Round
-winningScore:int
-deck:Deck
-trump:Poker.Suit
-lead:Poker.Suit
-selectedCard:Card
-numStartCards:int
-activePlayer:Player
-trickWinner:Player
-enforceRules: boolean
-players:ArrayList<Player>
-observers:ArrayList<Player>
+outOfCard():boolean
+Round(int,int,int,int,int)
+dealCards():void
+playRound():Optional<Integer>
-trickWinner(ArrayList<Player>,Card):Player
-shiftArray(ArrayList<T>,int):<T>ArrayList<T>
+addObserver(Observer):void
+removeObserver(Observer):void
+notifyObserver():void
creates
1
1
Whist
trumpImage: String[]
-handwidth:int
-trickwidth:int
-handLocations: Location[]
-scoreLocations:Locations[]
-scoreActors:Actors
-trickLocation: Location
-textLocation: Location
-hideLocation: Location
-trumpsActorLocation: Location
+bigFont: Font
-whistProperties:Properties
-round:Round
+update():void
-initialiseProperties():void
-readProperties():void
-loadProperties():void
-initGraphics():void
-updaleGraphics():void
-updateScoreGraphics():void
-updateCardGraphics():void
-updateText():void
+Whist()
+main(string[]):void
1
Manages
2..4
0..*
accesses
1
0..*
gets utilitiy methods from
1
PlayerFactory
-instance:PlayerFactory
+getInstance():PlayerFactory
+generatePlayers(int,int,int,int,int):ArrayList<Player>
+getPlayer(int,int,String):Player
1
gets singleton instance of
1
1
creates
2..4
<<abstract>>
Player
#id:int
#score:int
#hand:Hand
#selectedCard:Card
#message:String
#thinkingTime:int
+playCard(Hand,Poker.Suit,Poker.Suit):Card
+Player(int,int)
0..*
accesses strategies from
1
ComputerPlayer
previousTricks:ArrayList<ArrayList<Card>>
-recordTricks(Hand):void
+playCard(Hand,Poker.Suit,Poker.Suit)
+ComputerPlayer(int,int,String)
HumanPlayer
+setHand(Hand):void
+playCard(Hand,Poker.Suit,Poker.Suit)
+HumanPlayer(int)
StrategyFactory
-instance:StrategyFactory
+getInstance():StrategyFactory
getStrategy(String):Strategy
+StrategyFactory()
1
gets singleton instance of
1
creates
1
1..*
<<abstract>>
Strategy
+getCardsOfSuit(ArrayList<Card>,Suit):ArrayList<Card>
+isLegal(ArrayList<Card>,ArrayList<Card>,Card,Suit,Suit):boolean
+lowestRank(ArrayList<Card>):Card
+highestRank(ArrayList<Card>):Card
+winningCard(ArrayList<Card>,Poker.Suit,Poker.Suit):Card
+randomCard(ArrayList<Card>):Card
+rankGreater(Card,Card):boolean
+execute(ArrayList<Card>,ArrayList<Card>,Suit,Suit):Card
2
utilises
0..*
SmartStrategy
- strategies:ArrayList<Strategy>
+SmartStrategy()
+addStrategy(Strategy):void
+execute(ArrayList<Card>,ArrayList<Card>,Suit,Suit):Card
LegalStrategy
+execute(ArrayList<Card>,ArrayList<Card>,Suit,Suit):Card
LoseStrategy
+execute(ArrayList<Card>,ArrayList<Card>,Suit,Suit):Card
WinStrategy
-mapPlayedCards:HashMap<Suit,ArrayList<Rank>>
-smartCards: ArrayList<Card>
+WinStrategy()
-updateLastTrick(ArrayList<ArrayList<Card>>): void
-refreshAtRoundEnd():void
-TrickPosition(ArrayList<Card>):String
-initialisePlayableCards(ArrayList<Card>,Suit,Suit,boolean):boolean
-guessWinningPlay():void
-countGuess():void
-winningPlay(ArrayList<Card>,Suit,Suit):void
+execute(ArrayList<Card>,ArrayList<Card>,Suit,Suit):Card
RandomStrategy
+execute(ArrayList<Card>,ArrayList<Card>,Suit,Suit):Card

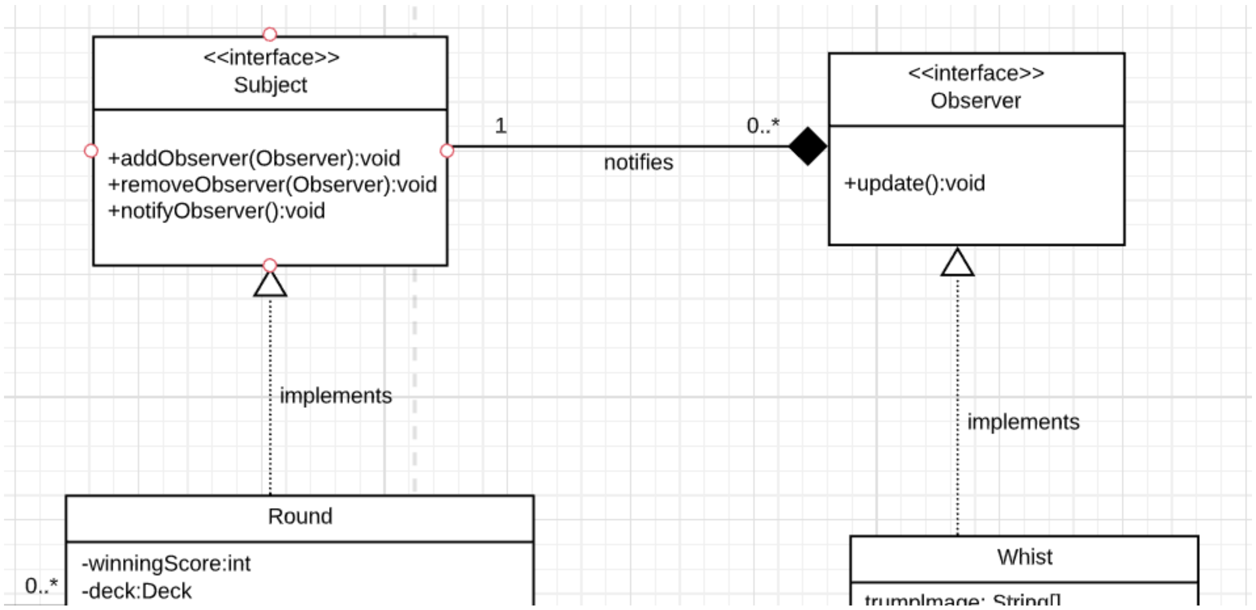
Design Class Diagram

3. Patterns and Principles

3.1. Observer Pattern

The observer pattern is implemented between `Round` and `Whist`, in which `Round` publishes events that *Whist* observes.

- `Round` notifies `Whist` of any changes in the game, such as a card being played or change in player score
- When `Whist` is notified of such events, it updates and refreshes all graphic components to reflect the changes



Observer Pattern

By following the observer pattern, we are able to separate logic elements from the display elements. Display rendering is managed by `Whist`, while all the game-related functionalities are covered by `Round` allowing high cohesion. Additionally, this method allows us to remove any knowledge of `Whist` class from `Round` and allow it to update Whist's multiple graphic components without explicitly calling its graphic functions, therefore reducing coupling and improving cohesion.

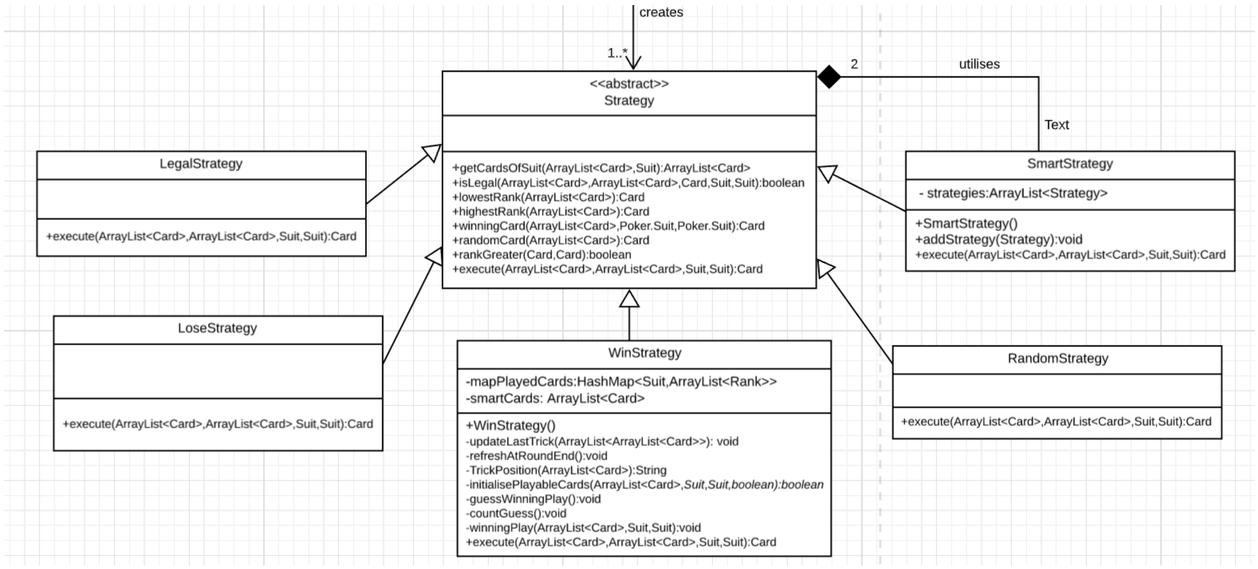
3.2. Strategy and Composite pattern

`Strategy` *class*

Following the Strategy pattern allows us to define algorithms for various play styles that a `Player` can resort to. Instead of creating various subclasses of `Player` and defining explicit play styles within them, this pattern enables flexibility in terms of a *Player's* choice of *Strategie*s as the varying Strategies can be used discretely or in combination.

In addition, through the use of inheritance and polymorphism, extensibility is achieved as it allows for new, independent subclasses of *Strategy* to be created to cater to the introduction of new play styles.

Last but not least, following the composite strategy pattern enables `SmartStrategy`, an intelligent play style that maximises the probability of winning, to combine and utilise multiple strategies in order to select the most optimal card to play.

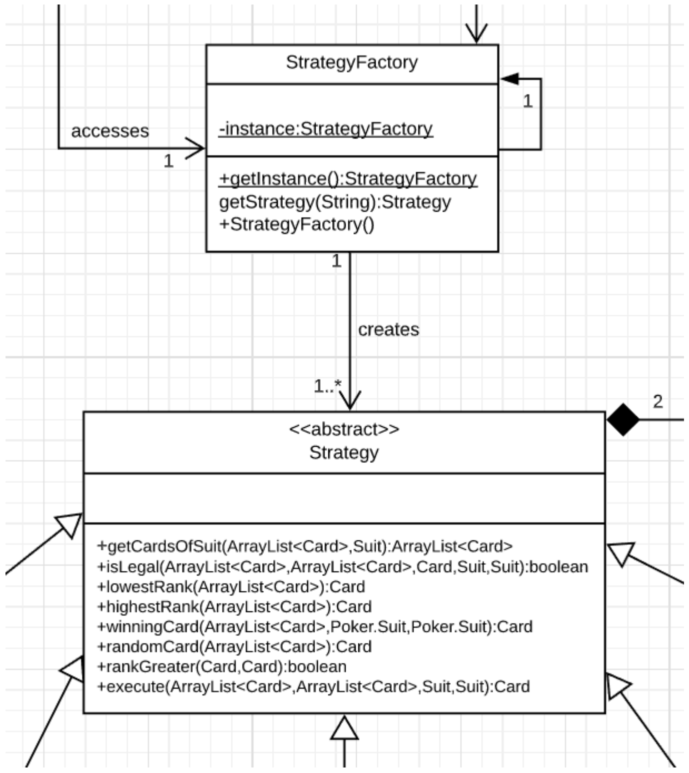

creates
1..*
2
utilises
Text
<<abstract>>
Strategy
+getCardsOfSuit(ArrayList<Card>,Suit):ArrayList<Card>
+isLegal(ArrayList<Card>,ArrayList<Card>,Card,Suit,Suit):boolean
+lowestRank(ArrayList<Card>):Card
+highestRank(ArrayList<Card>):Card
+winningCard(ArrayList<Card>,Poker.Suit,Poker.Suit):Card
+randomCard(ArrayList<Card>):Card
+rankGreater(Card,Card):boolean
+execute(ArrayList<Card>,ArrayList<Card>,Suit,Suit):Card
LegalStrategy
+execute(ArrayList<Card>,ArrayList<Card>,Suit,Suit):Card
SmartStrategy
- strategies:ArrayList<Strategy>
+SmartStrategy()
+addStrategy(Strategy):void
+execute(ArrayList<Card>,ArrayList<Card>,Suit,Suit):Card
LoseStrategy
+execute(ArrayList<Card>,ArrayList<Card>,Suit,Suit):Card
WinStrategy
-mapPlayedCards:HashMap<Suit,ArrayList<Rank>>
-smartCards: ArrayList<Card>
+WinStrategy()
-updateLastTrick(ArrayList<ArrayList<Card>>): void
-refreshAtRoundEnd():void
-TrickPosition(ArrayList<Card>):String
-initialisePlayableCards(ArrayList<Card>,Suit,Suit,boolean):boolean
-guessWinningPlay():void
-countGuess():void
-winningPlay(ArrayList<Card>,Suit,Suit):void
+execute(ArrayList<Card>,ArrayList<Card>,Suit,Suit):Card
RandomStrategy
+execute(ArrayList<Card>,ArrayList<Card>,Suit,Suit):Card

Strategy and its subclasses

3.3. Factory Pattern

StrategyFactory

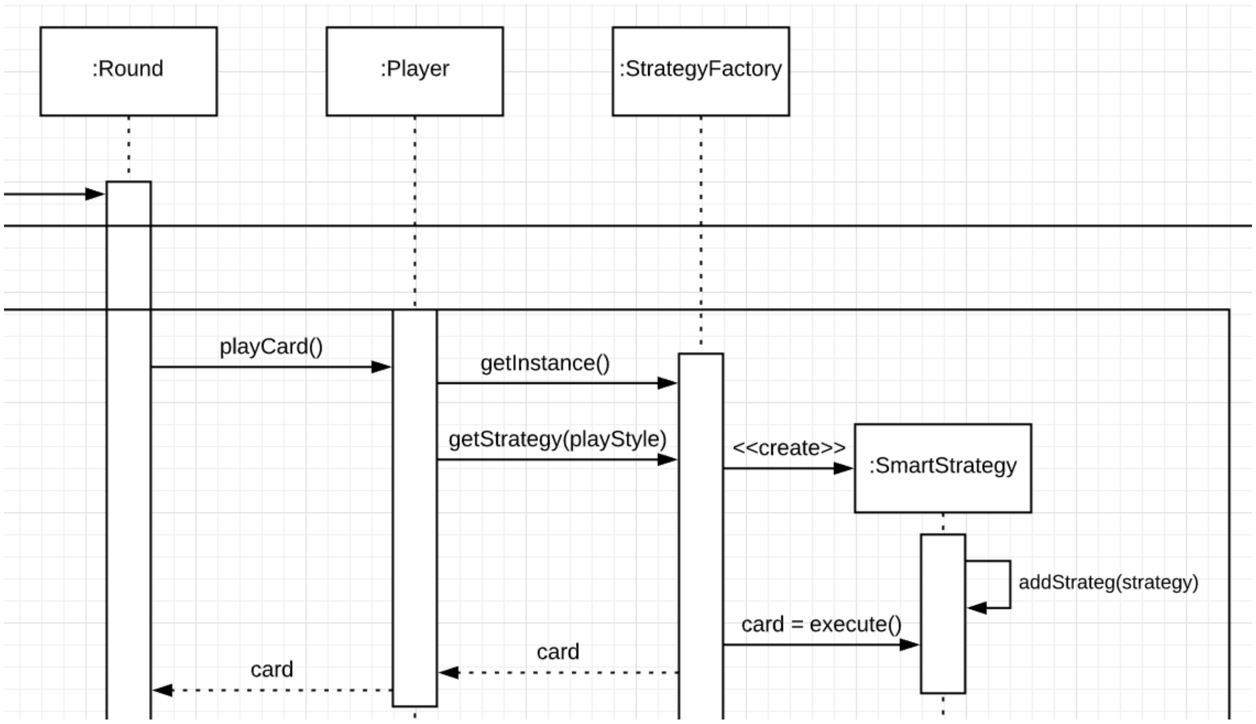
StrategyFactory is created as a layer of abstraction between **Player** and **Strategy**, which handles the selection and creation logic of *Strategy* for a *Player*.


StrategyFactory
-instance:StrategyFactory
+getInstance():StrategyFactory
getStrategy(String):Strategy
+StrategyFactory()
accesses
1
1
1
creates
1..*
2
<<abstract>>
Strategy
+getCardsOfSuit(ArrayList<Card>,Suit):ArrayList<Card>
+isLegal(ArrayList<Card>,ArrayList<Card>,Card,Suit,Suit):boolean
+lowestRank(ArrayList<Card>):Card
+highestRank(ArrayList<Card>):Card
+winningCard(ArrayList<Card>,Poker.Suit,Poker.Suit):Card
+randomCard(ArrayList<Card>):Card
+rankGreater(Card,Card):boolean
+execute(ArrayList<Card>,ArrayList<Card>,Suit,Suit):Card

StrategyFactory

Low coupling is achieved via this pattern as a **Player** is not required to explicitly call the constructor method of each Strategy, allowing the **Player** class to not know anything about **Strategy** class while still accessing its strategic algorithms. Also worth mentioning, **StrategyFactory** is Singleton ensuring that only a single object exists such that there is a global and single point of access for all Strategies.

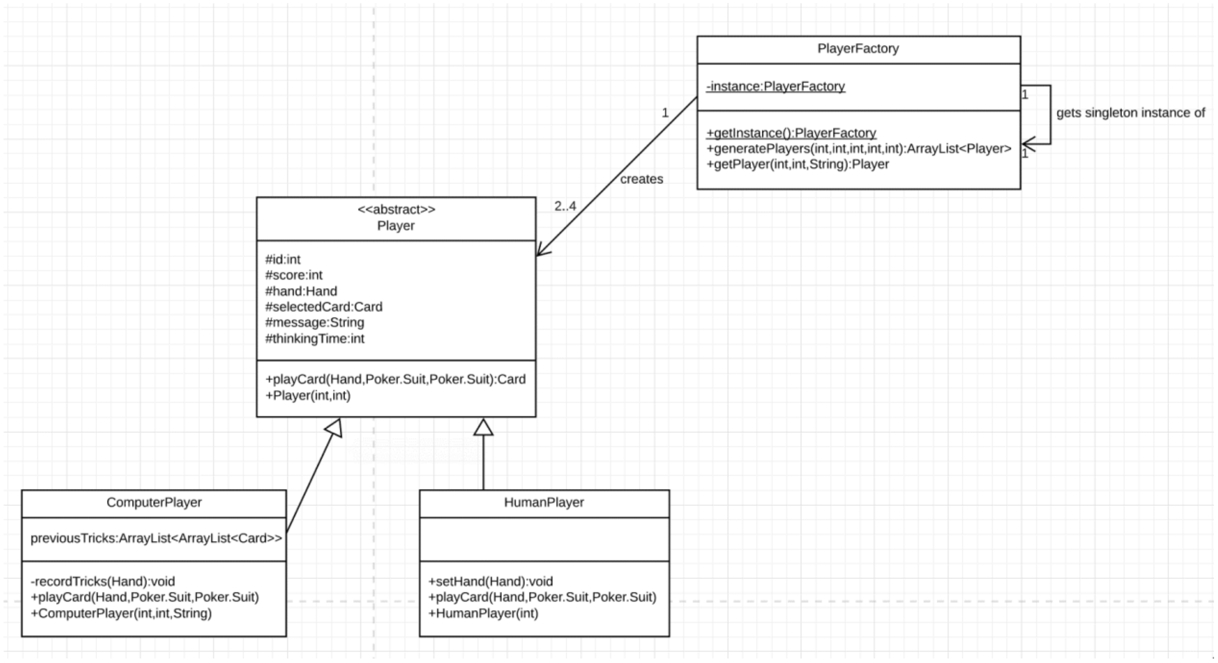
Therefore the procedure of a **Player**’s use of **Strategy** can be summarised as below



Player accessing a Strategy

PlayerFactory

This class hides the complex **Player** creation logic from **Round**. During the initialisation of each game, **Round** feeds the relevant user configuration information (number of NPCs, human players etc.) to **PlayerFactory**, which subsequently generates the correct number of **Player** entities, all provided with unique IDs, and sends them back to **Round**.


PlayerFactory
-instance:PlayerFactory
+getInstance():PlayerFactory
+generatePlayers(int,int,int,int,int):ArrayList<Player>
+getPlayer(int,int,String):Player
gets singleton instance of
1
1
1
creates
2..4
<<abstract>>
Player
#id:int
#score:int
#hand:Hand
#selectedCard:Card
#message:String
#thinkingTime:int
+playCard(Hand,Poker.Suit,Poker.Suit):Card
+Player(int,int)
ComputerPlayer
previousTricks:ArrayList<ArrayList<Card>>
-recordTricks(Hand):void
+playCard(Hand,Poker.Suit,Poker.Suit)
+ComputerPlayer(int,int,String)
HumanPlayer
+setHand(Hand):void
+playCard(Hand,Poker.Suit,Poker.Suit)
+HumanPlayer(int)

PlayerFactory

3. Alternative Solutions

One of our earlier considerations was to implement multiple strategies within the **Player** class and create NPC players accordingly. This would lead to low cohesion within the Player class. Logical changes would also become tougher to implement and maintain. By following the Factory pattern instead, we implemented **StrategyFactory** as a layer of abstraction between Player and Strategy.

Another solution explored by us while designing the NPC's Strategy was to implement a single **SmartStrategy** class rather than implementing various strategies and making SmartStrategy a composite class of various strategies. However, this solution seemed disadvantageous in the long run. We would have sacrificed cohesion of each strategy that we implemented in a single class. Moreover, the extensibility of the **SmartStrategy** would have been compromised. Later additions of possibly improved *winning/Losing strategies* could not be stacked on top of the current ones.