## Explaining the Software Design Patterns and Principles Used

### The Façade Controller Pattern

The design implements a façade controller (Whist class), which acts as the main interface between the user defined properties and the actual software. This can be seen from the “readPropertiesFile” method, which stores the user inputs into the corresponding class attributes. It is essentially the information expert with respect to the overall game state (it controls the necessary operations of the game such as executing a round, determining when to change the trump suit, and determining the winner, as well as updates the graphics when any event leads to a change in the game state). This makes the entire design cohesive by invoking other classes and delegating responsibilities to them. For example, instead of assigning the Whist class with the responsibility of instantiating the game actors, it delegates this task to a factory object. This is shown in **Figure 1** below:

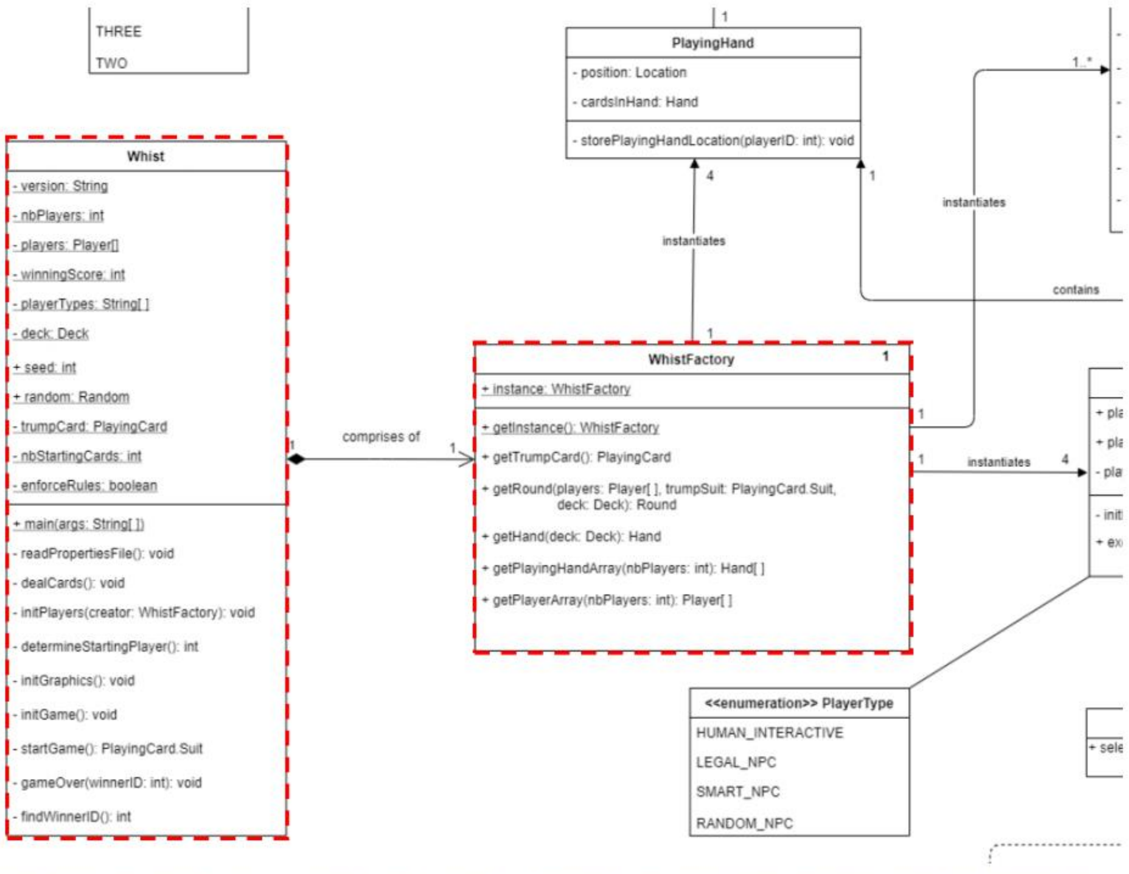
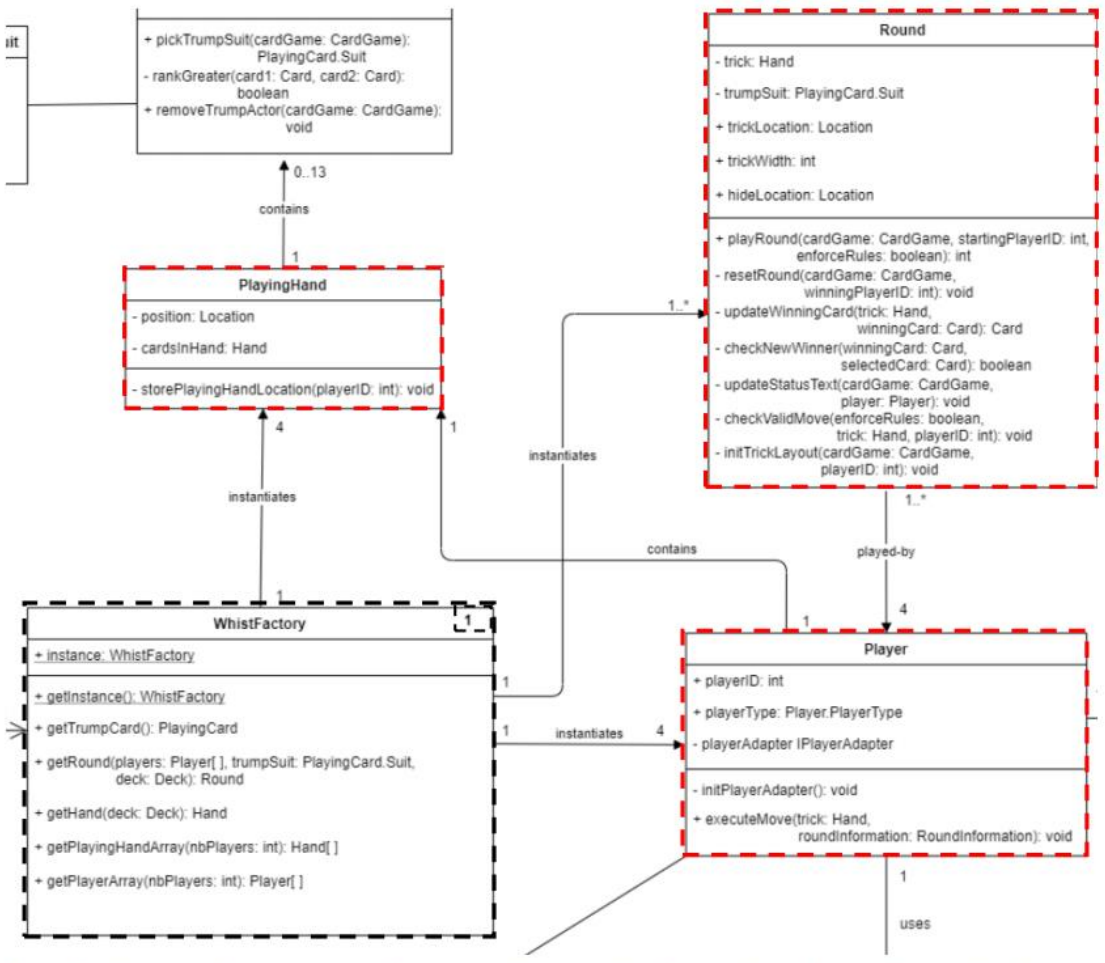


*Figure 1: Façade Controller Implemented with Delegation of Object Creation to Factory Object (Excerpt from Static Design Model)*

Although the Whist class is responsible for instantiating the factory object, it is the factory object that instantiates the remaining primary game actors. The purpose of this design decision is to abstract responsibilities to the appropriate classes (responsibility-driven design) and reduce the degree of coupling between the façade controller and other classes.

**The Singleton (Concrete) Factory Pattern**

As mentioned above, a factory object is assigned the role of instantiating the primary game actors (utilizing the creator and pure-fabrication GRASP patterns). Since there is relatively complex logic associated with the creation of different player types and their respective card selection strategies, a purely fabricated object of type WhistFactory is necessary. This is shown in **Figure 2** below:



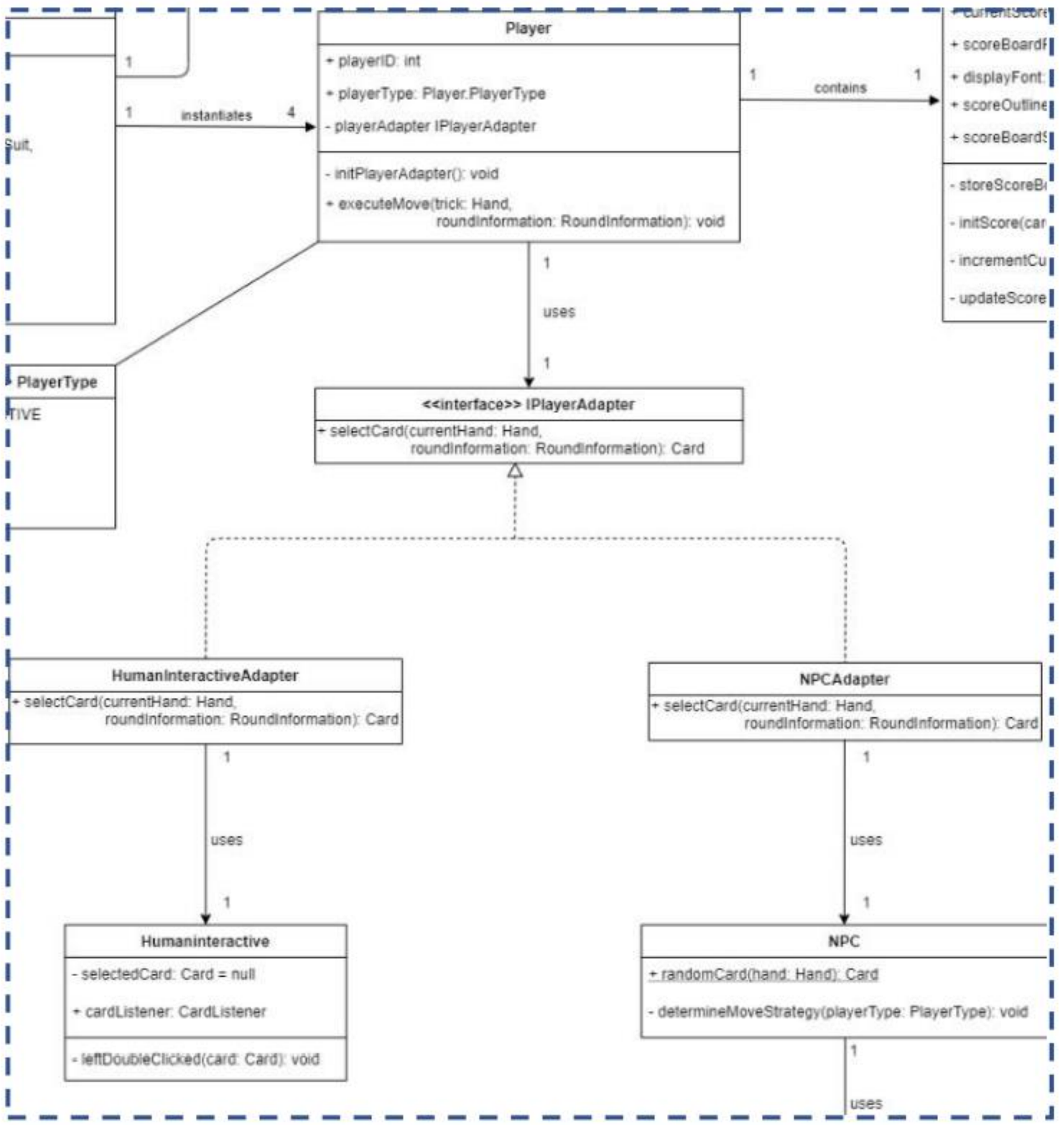
+ pickTrumpSuit(cardGame: CardGame): PlayingCard.Suit
- rankGreater(card1: Card, card2: Card): boolean
+ removeTrumpActor(cardGame: CardGame): void
0..13
contains
1
Round
- trick: Hand
- trumpSuit: PlayingCard.Suit
+ trickLocation: Location
+ trickWidth: int
+ hideLocation: Location
+ playRound(cardGame: CardGame, startingPlayerID: int, enforceRules: boolean): int
- resetRound(cardGame: CardGame, winningPlayerID: int): void
- updateWinningCard(trick: Hand, winningCard: Card): Card
- checkNewWinner(winningCard: Card, selectedCard: Card): boolean
- updateStatusText(cardGame: CardGame, player: Player): void
- checkValidMove(enforceRules: boolean, trick: Hand, playerID: int): void
- initTrickLayout(cardGame: CardGame, playerID: int): void
PlayingHand
- position: Location
- cardsInHand: Hand
- storePlayingHandLocation(playerID: int): void
1..*
4
1
instantiates
instantiates
contains
1..*
played-by
WhistFactory
+ instance: WhistFactory
+ getInstance(): WhistFactory
+ getTrumpCard(): PlayingCard
+ getRound(players: Player[ ], trumpSuit: PlayingCard.Suit, deck: Deck): Round
+ getHand(deck: Deck): Hand
+ getPlayingHandArray(nbPlayers: int): Hand[ ]
+ getPlayerArray(nbPlayers: int): Player[ ]
1
1
1
instantiates
4
1
4
Player
+ playerID: int
+ playerType: Player.PlayerType
- playerAdapter IPlayerAdapter
- initPlayerAdapter(): void
+ executeMove(trick: Hand, roundInformation: RoundInformation): void
1
uses

***Figure 2:** Singleton (Concrete) Factory Pattern used to Instantiate Primary Game Actors (Excerpt from Static Design Model)*

The (concrete) factory pattern is used to hide the complexity associated with the creation of these game actors from the façade controller, thus leading to more cohesive class responsibilities. It also supports the open-closed software principle: other types of NPCs with different behaviours can easily be introduced in the future. The singleton pattern is used to give the factory object a global and single point of access, which supports concurrent access to this shared resource if the code were to be extended in the future. As a result, the WhistFactory class contains an attribute class "instance" and the static method "getInstance" which returns the object if it has already been instantiated in a class or returns a newly instantiated object of type WhistFactory if not.

**The Adapter Pattern**

In order to support the different behaviors/library of methods that each player category (Human Interactive and NPC) has, the adapter pattern is used. An interface called IPlayerAdapter is defined with an abstract method called "selectCard". Since the human interactive players select the card to play by performing a double left-click on the card in hand, they have their own library of methods to execute this. The state of the game is visible without requiring additional round information, unlike the NPC's. The NPC's use the round information in order to facilitate their decision making, and so, have their own library of functions to accomplish this. As a result, HumanInteractiveAdapter and NPCAdapter classes are also defined with the associations shown in **Figure 3** below:



Player
+ playerID: int
+ playerType: Player.PlayerType
- playerAdapter IPlayerAdapter
- initPlayerAdapter(): void
+ executeMove(trick: Hand, roundInformation: RoundInformation): void
instantiates
contains
uses
PlayerType
TIVE
Suit,
<<interface>> IPlayerAdapter
+ selectCard(currentHand: Hand, roundInformation: RoundInformation): Card
HumanInteractiveAdapter
+ selectCard(currentHand: Hand, roundInformation: RoundInformation): Card
NPCAdapter
+ selectCard(currentHand: Hand, roundInformation: RoundInformation): Card
HumanInteractive
- selectedCard: Card = null
+ cardListener: CardListener
- leftDoubleClicked(card: Card): void
NPC
+ randomCard(hand: Hand): Card
- determineMoveStrategy(playerType: PlayerType): void
+ scoreBoardF
+ displayFont:
+ scoreOutline
+ scoreBoardS
- storeScoreB
- initScore(car
- incrementCu
- updateScore

*__Figure 3:__ Adapter Pattern used for Different Player Categories (Excerpt from Static Design Model)*

The purpose of using the adapter pattern is to abstract behavior to a single method called "selectCard" so that this method can be invoked directly by creating an instance of the IPlayerAdapter in the Player class. Relating to the GRASP patterns, the adapter pattern uses polymorphism (through the implementation of the IPlayerAdapter interface) and an indirection mechanism achieved by pure fabrication (the HumanInteractive and NPC classes are not directly accessed – they are accessed through their respective adapters that are not part of the original problem domain). This reduces the degree of coupling by decreasing the potential dependencies between the Round class and the different types of players (no need to call each method of each player type separately), thus increasing the overall cohesion of the design solution.

## The Strategy Pattern

To accommodate for the different strategies of moves played by the different NPC types (random, legal, and smart), the strategy pattern was implemented. A separate interface called ICardChoiceStrategy was implemented, with an abstract method "executionAlgorithm". This is shown in **Figure 4** below:

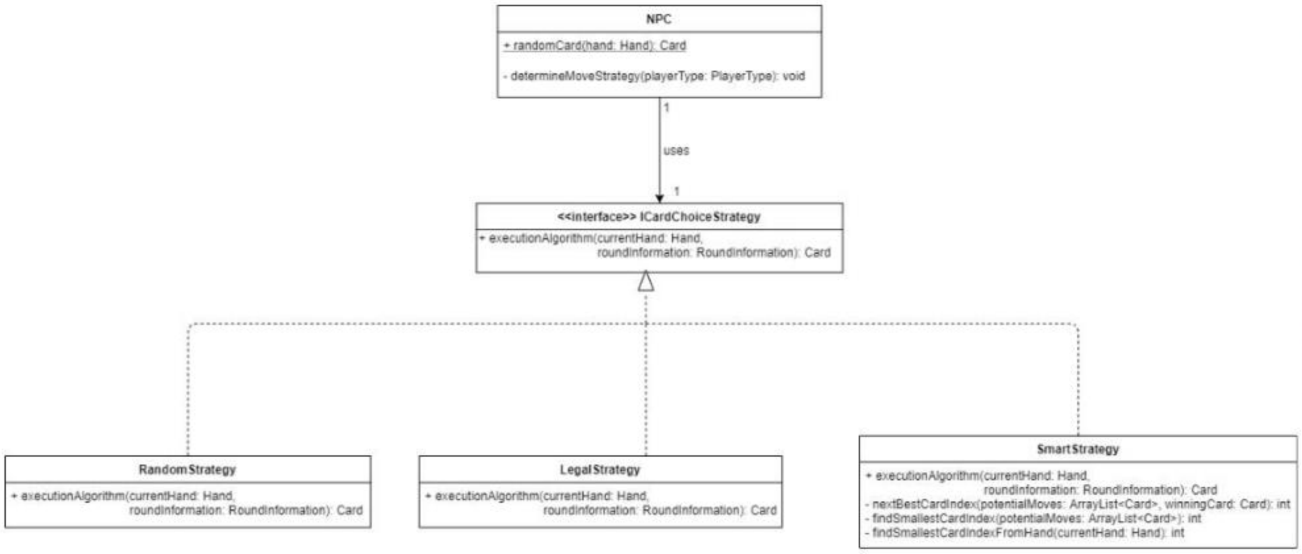

NPC
+ randomCard(hand: Hand): Card
- determineMoveStrategy(playerType: PlayerType): void
1
uses
1
<<interface>> ICardChoiceStrategy
+ executionAlgorithm(currentHand: Hand, roundInformation: RoundInformation): Card
RandomStrategy
+ executionAlgorithm(currentHand: Hand, roundInformation: RoundInformation): Card
LegalStrategy
+ executionAlgorithm(currentHand: Hand, roundInformation: RoundInformation): Card
SmartStrategy
+ executionAlgorithm(currentHand: Hand, roundInformation: RoundInformation): Card
- nextBestCardIndex(potentialMoves: ArrayList<Card>, winningCard: Card): int
- findSmallestCardIndex(potentialMoves: ArrayList<Card>): int
- findSmallestCardIndexFromHand(currentHand: Hand): int

***Figure 4:*** *Strategy Pattern used for Different Execution Algorithms for Different NPC Types (Excerpt from Static Design Model)*

Although each NPC has access to the round information, only the smart and legal NPC's use this information to make its decision on which card to select. As with the adapter pattern, the behavior of each NPC type is abstracted to a single method called "executionAlgorithm". An instance of the ICardChoiceStrategy interface is instantiated in the NPC class, and calling the "executionAlgorithm" will implement the specific behavior whilst hiding its complexity further up the player hierarchy. This increases the cohesion of the design, by giving each class a well-defined responsibility, and allows for easy extensibility (with minimum modification to existing code) if other NPC algorithms/playing strategies were to be implemented in the future.