# Project 2: Whist Game — Group 35

Glenn Phillips 820624 — Eldar Kurmakaev 1028326 — Michael Lowe 911365
SWEN30006 - Software Modelling and Design
University of Melbourne

June 5, 2020

## Introduction

The task is to design and improve on the innovative Whist card game that has been setup by New, Exciting, and Revolutionary Designer Games (NERD Games). The new features will expand on their design to allow for greater configurability and include new NPC players to play against. At the same time, these additional features must expect to have even further game additions in the future, so therefore, they will need to be implemented in a way that allows for easier extensions of the game in the future.
Before working on the code, drawing up a Domain Class Diagram was an important first step that allowed us to get a good idea of how we wish to implement the code and decide which classes should be talking to each other. From the Domain Diagram, we built a Design Class Diagram which would include all the links that we wanted from the Domain whilst also including in software concepts. From the Design model we could see that there were classes that could be split and relationships that could be redirected, when thinking about the design patterns. This report will look through the discussions of adding or modifying features through design patterns to allow for a better and highly configurable code.

## Analysis of Problem Domain

In our analysis of the problem domain the key change that NERD Games requested be implemented was a series of strategies for each NPC player that could be extended upon in the future. Upon reading this in the brief it was clear to the team that our solution must consider two alternate player types, NPC and Human, and so we represented these within our Domain Class Diagram as extending from the Player Class. Furthermore, we believed that the strategy that a NPC robot uses would be best represented as a class as well. Moreover, each type of strategy, random, legal and smart, should extend from this strategy class.
Another key decision made during our analysis were the relationships between hand, card, deck, suit and rank classes. In our domain class diagram we make the assertion that cards are dealt from the deck to the hand. Therefore whilst a deck is initialised as 52 cards it gradually decreases to zero. Similarly as cards are added and removed to an indefinitely sized hand, the number of cards within the hand changes invariably. Another key point illustrated within the domain class diagram is that there are four cards of each rank, and thirteen cards in each suit. This is shown via the cardinality between these classes.

## Design Process

In this section, we will go over the design problems that we faced and the patterns that we used to then overcome each problem. Below is a discussion of these problems and our decision making process in choosing a solution for each.

### Which class should be responsible for determining the winner of the Game?

We considered assigning this responsibility to Trick per information expert, as our initial implementation of Trick had all the necessary information to determine the winner of the game. It was responsible for incrementing the score of the players when a trick was won, meaning it would be the first class to know if a new winner emerged. However this led to low cohesion and a high representational gap. To solve this we instead chose Round to be responsible for checking if there was a winner. Trick would return the trickWinner to Round when the playTrick method was called. Round then has the responsibility to check if this trick winner is also a game winner. As Round is responsible for iterating over trick this can be considered more cohesive and to have a lower representational gap.

### Which class should be responsible for the initialization of the Round?

Our first consideration was if the factory pattern should be implemented for the creation of Round. We concluded that the creation of a RoundFactory was excessive, as the constructor logic was very simple and did not need a new class to manage it's creation. The existing WhistGame class was the only class that contains an instance of Round so it should be responsible for its initialization as per creator. This approach proved to be beneficial as it also helped maintain a low representational gap.

### Which class should be responsible for the initialization of the Trick?

The WhistGame has only part of the necessary data for the initialization of the Trick; it stores deck and players. The trump suit is also required but is stored within the Round class. Even though WhistGame has more of the necessary information for the creation of the Trick, having this as the creator would increase the representational gap and lower the cohesion. Therefore we assigned Round the responsibility of initializing Trick.

### Which class should be responsible for the initialization of the Deck?

This decision was not easy. Round indeed does use Deck the most out of all classes. Nevertheless assigning this responsibility to the round would decrease its cohesion as Round is already responsible for dealing the cards out to the players. Furthermore, this structure does not support real-world representation, in which Deck is part of the game, not the round. Therefore we placed this responsibility within WhistGame, since it supports lower representational gap and has higher cohesion.

### Which class should be responsible for the creation of four different players?

To achieve this task, all needed information is contained within PropertyReaderSingleton. However designing only on the principle of information expert would lead to the substantial decrease of cohesion of PropertyReaderSingleton, as its primary responsibility is to read a properties file and provide information to other classes.
Another alternative is to assign the responsibility of Player creation to the Round class based on the creator principle as it is a class that closely uses the instances of Player. Initialising players involves complex logic, and having this logic in Round would have a negative impact on its cohesion, especially taking into consideration that it already has many responsibilities. Likewise if this responsibility is assigned to the WhistGame class, this complex creation logic would have a similar effect and lower the class's cohesion.

Therefore we created a Factory class called PlayerFactory. Via implementing the factory and pure fabrication design patterns higher cohesion has been achieved. The PlayerFactory class contains all complex creation logic and returns an ArrayList of Player instances through a getPlayers method. The negative impact of an increase in coupling is greatly outweighed by the positive change to cohesion.

**How to allow a player a different behavior type based on whether he is an NPC player or a human player?**

Because there are common class attributes (hand, score, player) that are present in both NPCPlayer and HumanPlayer, the usage of an abstract class instead of an interface is more appropriate as it allows for code reuse. The two current possible players are created in the PlayerFactory which splits Human and NPC players and if it is the latter, then it gives them a specific strategy as well. It is important to note that this approach, based on polymorphism pattern, allows for protected variations for future variations/instability as it provides a common interface via selectCard method.

**Which pattern to use in order to allow for NPCPlayer different behavior types while maximizing reuse of the code and allowing for future evolution?**

The first considered option was to create a Player abstract superclass with selectCard() method that would be overridden by hypothetical RandomNPCPlayer(), SmartNPCPlayer etc. The implementation of EvenSmarterNPCPlayer (required by specification) would be required to inherit from the SmartNPCPlayer and override the whole selectCard() method. Assuming that there would be a lot of shared code between the two discussed classes, this approach would lead to code duplication. Furthermore, we believe that possible very-very-very smart NPC player in the future is supposed to be able to adapt and pick from a number of possible smart strategies. But in this approach, there is a possibility only for one behavior.
As a solution, we implemented a Strategy pattern. This pattern enables for protected variations necessary for having games of different NPC players via polymorphism mechanics. We created an AbstractSelectCardStrategy abstract class. We decided to go with an abstract class instead of the interface because there are a number of attributes that are used by all implementations. Each NPC-Player class has an instance attribute of AbstractSelectCardStrategy but a specific implementation Smart.../Legal.../RandomSelectCardStrategy. Each implementation has a number of methods.
This allows an easy solution for an NPC player to select a card. This can be seen easily in the attached Design Sequence Diagram, in which the NPC player receives a notification that it needs to select a card and will then push that to message to the stored SelectCardStrategy class which then handles the selection of the card based on what type of NPC player it is.
Thus if a very smart NPC needs to be created it can inherit from SmartSelectCardStrategy and override one of the eight methods that control the behaviour of the player with better methods. Further, the strategy approach would allow in the future to implement potential very-very-very smart NPC by implementation of composite patterns. We did not implement the composite pattern in this iteration as Strategy is sufficient for current needs and the nearest future.

**Who should keep track of all the cards played?**

The smart strategy uses a few methods which take into account the previous cards that have been played in that round. This is used to check if the player has the top card or not, which can guarantee a win where possible. The original implementation had this stored in the NPC Player but after discussion it was found that this made the class less cohesive. We decided to implement

the observer pattern whereby an instance of Trick was the observable and multiple instances of SmartSelectCardStrategy were the Observer. Trick was the logical choice for to be the observable as it contained the method that set the value of the selected card. Upon a new card being selected for the trick, the trick would notify all observers, in this case each instance of SmartSelectCardStrategy. This then triggered the update method within each SmartSelectCardStrategy which would add a copy of the selected card to cards played. By moving this responsibility from the NPC player class, usage of the Observer pattern was able to increase the cohesion of the NPC player class. It also meant that the cards played were only accessed by its observers; those classes that were interested in updates to it. This meant that rather than all three strategies unnecessarily having access to updates in cards played, now only the smart strategy did.

**How to separate business logic from view logic to allow for points of evolution within the GUI?**

It is a very common issue that as technologies develop the games need to be migrated to devices that use different GUI frameworks. Therefore it is necessary to have business logic and render logic separated according to Model-View separation principle.
The first pattern evaluated as a potential solution was the Observer pattern. The benefit of the pattern is that it allows several GUI to spy on whether there is a state change in a business logic component and update multiple GUIs once it is accrued. The negative of the pattern is that if there are states that are sometimes mutually exclusive (each time trick winner is determined, the game winner is not necessarily announced) it requires implementation of multiple observers/or states of application. Furthermore observer is generally best used when there is one observable for many observers. In our case there was only one observer, the WhistDisplay and many observables. Subsequently we evaluated that costs vastly outweighed the benefits of this approach.

We instead decided to use the facade pattern. The facade pattern was implemented via WhistDislplayFacade. Via indirection this class provides protected variation to a UI component (WhistDisplay class) for classes that need to update UI (Player,Round,WhistGame,Trick). WhistDisplay class is fully responsible for GUI view which allows us to separate logic from the view concerns.

Furthermore, WhistDislplayFacade is both a singleton and also a factory for the WhistDisplay. This is the case as there are multiple classes that use the UI. Through this implementation the UI, WhistDisplay, does not have to be passed via constructors and methods which would complicate the code. Instead they can be accessed and instanced by the WhistDisplayFacade in accordance with the Facade, Factory and Singleton design patterns.

As we asserted that the likely-hood of variations to the display logic or game logic were quite high, we concluded that the benefits of this implementation of WhistDisplayFacade outweighed the costs.

**Which class should be responsible for reading the properties file data and providing information to other classes?**

Placing a responsibility in each class that needs information from the properties file would violate high cohesion principle as these Round/PlayerFactory/RandomFactory are designed to maintain business logic and not file I/O operations which is another set of responsibilities. Also, there would be a duplicated code.
Creation of the class PropertyReaderSingleton allowed us to separate file I/O responsibility into

cohesive class based on pure fabrication. By making it a Singleton class with lazy initialization logic we allowed for generally computationally expensive I/O operation to be only performed once. We designed the class to be accessed via static getInstance() method thus avoiding unnecessary coupling that would occur because otherwise, the instance of this class would have to be passed via many classes in the system in order to provide visibility to all classes that need to use this class.

**Which class should be responsible for the generation of random numbers based on a seed?**

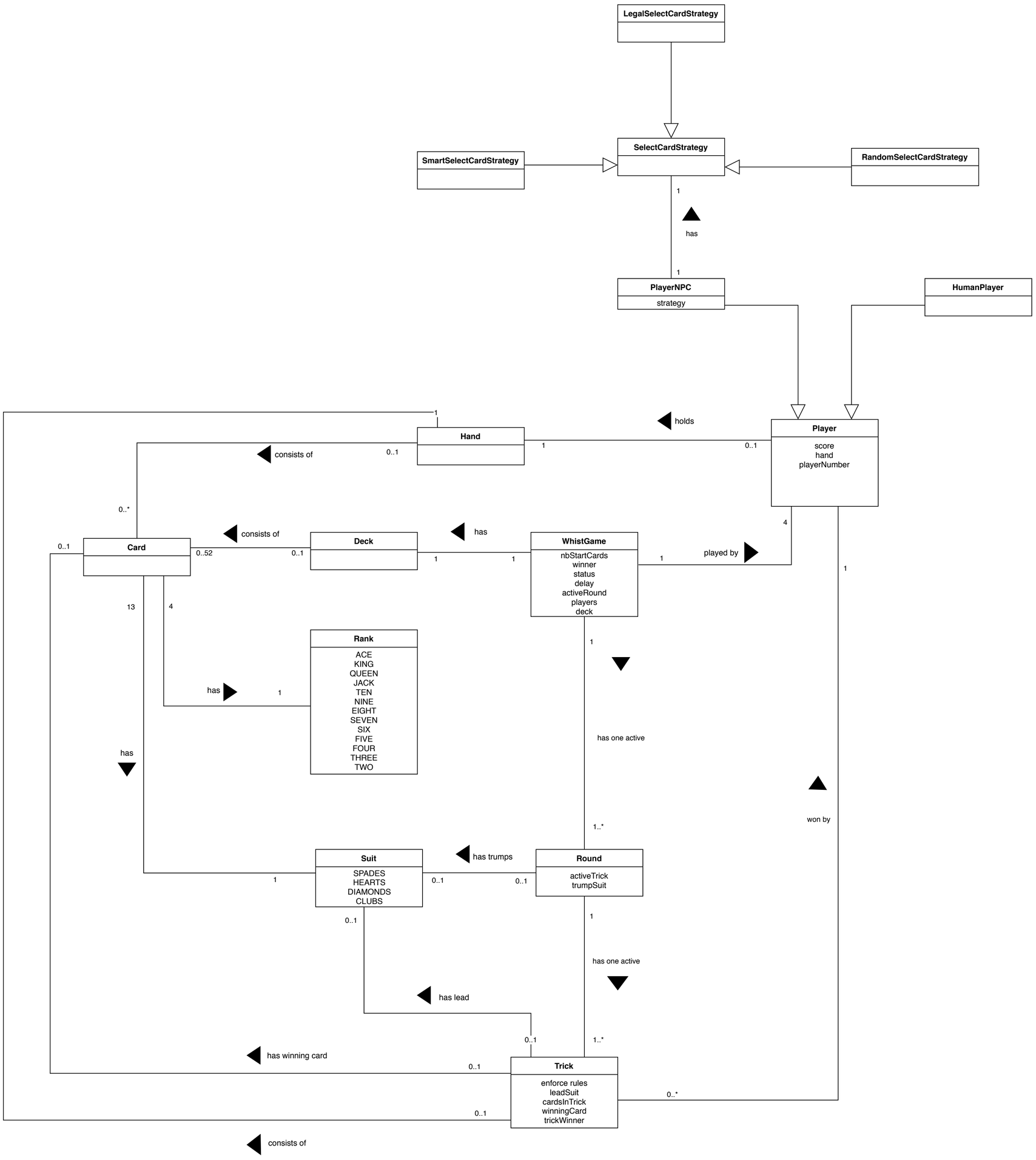
The PropertyReaderSingleton was an ideal choice to generate the random seed when using the information expert pattern. However, adding this feature to the class would decrease its cohesion, as its primary function is input operation. Therefore we wrote a RandomFactory class based on the factory pattern and pure fabrication principle, that is responsible for the creation of Random() instance, and is responsible for the logic of whether Random() is based on seed, or is purely random. This allows for low coupling and reuse.

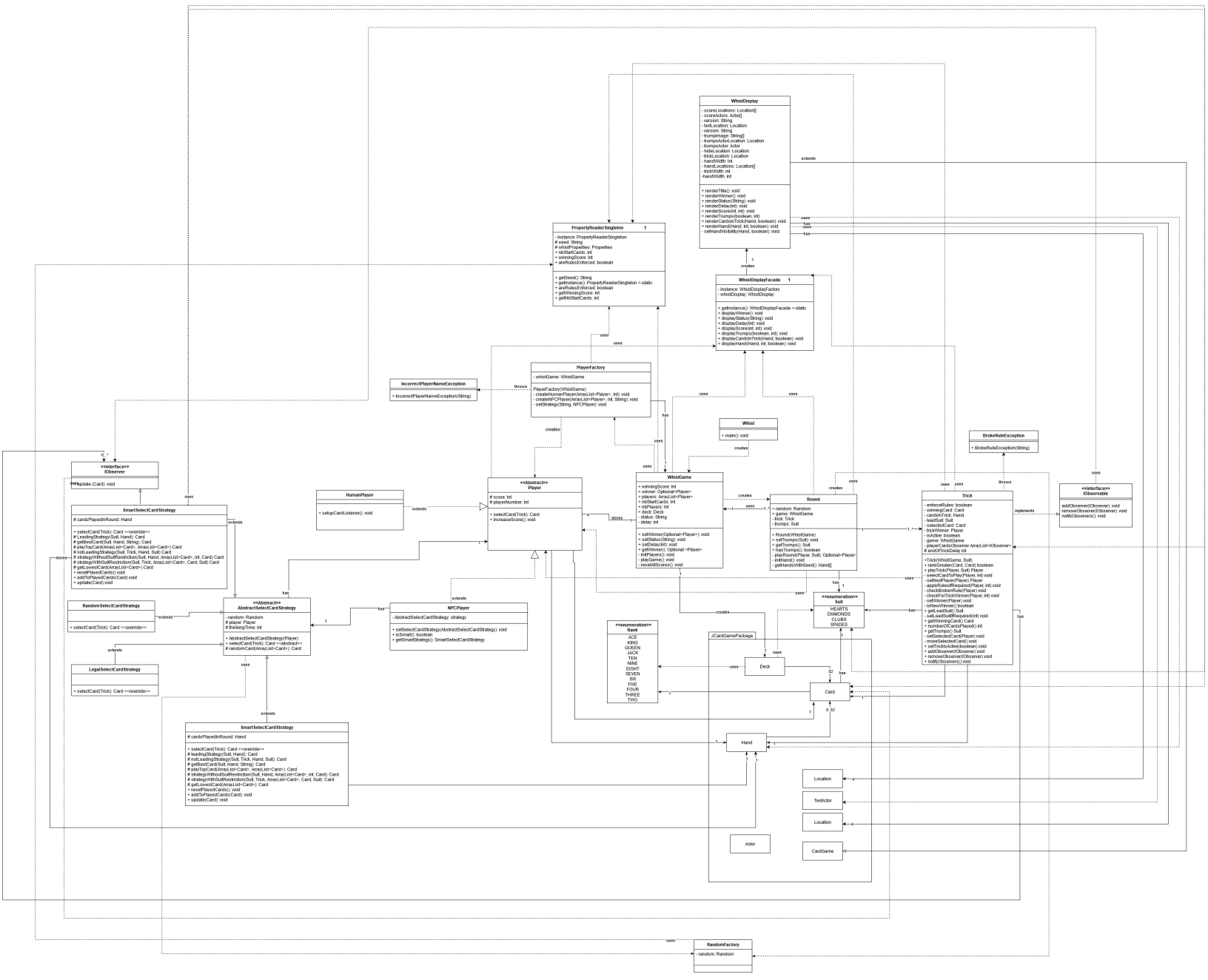
**How to handle use of the JCardGame library?**

It is important whenever designing a system to consider the impact that variations in a subsystems will have on the system as a whole. Subsequently we discussed the impact changes in JCardGame would have on our WhistGame. We further discussed if such changes were to take place, is it worthwhile trying to apply protected variations to prevent this. If we were to apply protected variation by perhaps creating a JCardGame facade it would require a lot of extra work and the resulting facade would almost represent a "God" class, where it would be doing too many different things. As our system was quite small we decided it would be easier to rework the existing design rather than implement a facade.
To further explain this point, imagine the system in question WhistGame has 'n' classes that depend on 'm' many classes within the JCardGame subsystem. At most the number of relationships would be n*m. If a facade were to be implemented, the number of relationships would be reduced to n+m. If n and m are small the difference between n*m and n+m is small, however as n and m increase the difference grows exponentially larger. Therefore for a smaller system, the benefit gained from implementing a facade is minimal, as was the case here, and so our final decision was not to include it.

## Conclusion

After carefully working through each problem we have extended on the original software to create a more cohesive code with can be easily extended in the future. This was achieved by working with common design patterns to come up with a solution to the problem initially asked by NERD Games.


LegalSelectCardStrategy
SelectCardStrategy
SmartSelectCardStrategy
RandomSelectCardStrategy
1
has
1
PlayerNPC
strategy
HumanPlayer
1
holds
Hand
1
0..1
0..1
Player
score
hand
playerNumber
consists of
0..1
0..*
consists of
0..1
Card
0..52
0..1
Deck
has
1
1
WhistGame
nbStartCards
winner
status
delay
activeRound
players
deck
1
played by
4
1
13
4
Rank
ACE
KING
QUEEN
JACK
TEN
NINE
EIGHT
SEVEN
SIX
FIVE
FOUR
THREE
TWO
has
1
has
1
has one active
won by
1..*
Suit
SPADES
HEARTS
DIAMONDS
CLUBS
has trumps
1
0..1
0..1
Round
activeTrick
trumpSuit
0..1
1
has one active
has lead
0..1
1..*
has winning card
0..1
Trick
enforce rules
leadSuit
cardsInTrick
winningCard
trickWinner
0..*
0..1
consists of


WhistDisplay
- scoreLocations: Location[]
- scoreActors: Actor[]
- versionLocation: Location
- versionActor: Actor
- versionString: String
- trumpImage: String[]
- trumpsActorLocation: Location
- trumpsActor: Actor
- hideLocation: Location
- trickLocation: Location
- handWidth: Int
- handLocations: Location[]
- trickWidth: Int
-handWidth: int
+ renderTitle(): void
+ renderWinner(): void
+ renderStatus(String): void
+ renderDelay(int): void
+ renderScore(int, int): void
+ renderTrumps(boolean, int)
+ renderCardsInTrick(Hand, boolean): void
+ renderHand(Hand, int, boolean): void
- setHandVisibility(Hand, boolean): void
extends
PropertyReaderSingleton 1
- instance: PropertyReaderSingleton
# seed: String
# whistProperties: Properties
+ nbStartCards: int
+ winningScore: int
+ areRulesEnforced: boolean
+ getSeed(): String
+ getInstance(): PropertyReaderSingleton ~static
+ areRulesEnforced: boolean
+ getWinningScore: int
+ getNbStartCards: int
WhistDisplayFacade 1
- instance: WhistDisplayFactory
- whistDisplay: WhistDisplay
+ getInstance(): WhistDisplayFacade ~static
+ displayWinner(): void
+ displayStatus(String): void
+ displayDelay(int): void
+ displayScore(int, int): void
+ displayTrumps(boolean, int): void
+ displayCardsInTrick(Hand, boolean): void
+ displayHand(Hand, int, boolean): void
creates
PlayerFactory
- whistGame: WhistGame
PlayerFactory(WhistGame)
- createHumanPlayer(ArrayList<Player>, Int): void
- createNPCPlayer(ArrayList<Player>, Int, String): void
- setStrategy(String, NPCPlayer): void
IncorrectPlayerNameException
+ IncorrectPlayerNameException(String)
throws
Whist
+ main(): void
BrokeRuleException
+ BrokeRuleException(String)
<<interface>>
IObserver
+ update (Card): void
<<interface>>
IObservable
addObserver(IObserver): void
removeObserver(IObserver): void
notifyObservers(): void
implements
SmartSelectCardStrategy
# cardsPlayedInRound: Hand
+ selectCard(Trick): Card ~<override>>
# leadingStrategy(Suit, Hand): Card
# getBestCard(Suit, Hand, String): Card
# playTopCard(ArrayList<Card>, ArrayList<Card>): Card
# notLeadingStrategy(Suit, Trick, Hand, Suit): Card
# strategyWithoutSuitRestriction(Suit, Hand, ArrayList<Card>, int, Card): Card
# strategyWithSuitRestriction(Suit, Trick, ArrayList<Card>, Card, Suit): Card
# getLowestCard(ArrayList<Card>): Card
+ resetPlayedCards(): void
+ addToPlayedCards(Card): void
+ update(Card): void
extends
HumanPlayer
+ setupCardListener(): void
extends
<<Abstract>>
Player
# score: Int
# playerNumber: Int
+ selectCard(Trick): Card
+ increaseScore(): void
plays
WhistGame
+ winningScore: Int
+ winner: Optional<Player>
+ players: ArrayList<Player>
+ nbStartCards: Int
+ nbPlayers: Int
+ deck: Deck
- status: String
- delay: Int
+ setWinner(Optional<Player>): void
+ setStatus(String): void
+ setDelay(int): void
+ getWinner(): Optional <Player>
- initPlayers(): void
- playGame(): void
- revealAllScores(): void
creates
Round
+ random: Random
+ game: WhistGame
- trick: Trick
- trumps: Suit
+ RoundWhistGame)
+ setTrumps(Suit): void
+ getTrumps(): Suit
+ hasTrumps(): boolean
- playRound(Player, Suit): Optional<Player>
- initHand(): void
- getHandsWithSeed(): Hand[]
Trick
- enforceRules: boolean
- winningCard: Card
- cardsInTrick: Hand
- leadSuit: Suit
- selectedCard: Card
- trickWinner: Player
- isActive: boolean
- game: WhistGame
- playerCardsObserver ArrayList<IObserver>
# endOfTrickDelay: int
+Trick(WhistGame, Suit)
+ rankGreater(Card, Card): boolean
+ playTrick(Player, Suit): Player
- selectCardToPlay(Player, int): void
- setNextPlayer(Player): Player
- applyRuleIfRequired(Player, int): void
- checkBrokenRule(Player): void
- checkForTrickWinner(Player, int): void
- setWinner(Player): void
- isNewWinner(): boolean
+ getLeadSuit(): Suit
- setLeadSuitIfRequired(int): void
+ getWinningCard(): Card
+ numberOfCardsPlayed(): int
+ getTrumps(): Suit
- setSelectedCard(Player): void
- moveSelectedCard(): void
+ setTrickIsActive(boolean): void
+ addObserver(IObserver): void
+ removeObserver(IObserver): void
+ notifyObservers(): void
RandomSelectCardStrategy
+ selectCard(Trick): Card ~<override>>
extends
<<Abstract>>
AbstractSelectCardStrategy
- random: Random
# player: Player
# thinkingTime: int
+ AbstractSelectCardStrategy(Player)
+ selectCard(Trick): Card ~<abstract>>
# randomCard(ArrayList<Card>): Card
has
NPCPlayer
- AbstractSelectCardStrategy: strategy
+ setSelectCardStrategy(AbstractSelectCardStrategy): void
+ isSmart(): boolean
+ getSmartStrategy(): SmartSelectCardStrategy
LegalSelectCardStrategy
+ selectCard(Trick): Card ~<override>>
extends
SmartSelectCardStrategy
# cardsPlayedInRound: Hand
+ selectCard(Trick): Card ~<override>>
# leadingStrategy(Suit, Hand): Card
# notLeadingStrategy(Suit, Trick, Hand, Suit): Card
# getBestCard(Suit, Hand, String): Card
# playTopCard(ArrayList<Card>, ArrayList<Card>): Card
# strategyWithoutSuitRestriction(Suit, Hand, ArrayList<Card>, int, Card): Card
# strategyWithSuitRestriction(Suit, Trick, ArrayList<Card>, Card, Suit): Card
# getLowestCard(ArrayList<Card>): Card
+ resetPlayedCards(): void
+ addToPlayedCards(Card): void
+ update(Card): void
extends
<<enumeration>>
Suit
HEARTS
DIAMONDS
CLUBS
SPADES
<<enumeration>>
Rank
ACE
KING
QUEEN
JACK
TEN
NINE
EIGHT
SEVEN
SIX
FIVE
FOUR
THREE
TWO
JCardGamePackage
Deck
Card
Hand
Location
TextActor
Location
Actor
CardGame
RandomFactory
- random: Random
uses
has
creates
extends
implements
throws
plays
stores

# Selecting a Card - Sequence Diagram