SWEN30006

Software Modelling and Design

Project 2: Whist – Design Analysis

Wilbert Cargeson (1039809), Oliver van Ingen (992712), Mitchell Needham (823604) - Team 7

1 Introduction

Our aim was to extend *NERD Games Inc.*'s Whist game to allow more advanced AI, to help provide the player with more of a challenge. *NERD Games* desired a solution based on a two tiered approach. Each NPC would have a filtering step and then a selection step. Both of these would be configurable based on a properties file. We did our best to follow GRASP principles and allow future changes. The game should be able to support new AI decisioning strategies without modifying existing code.

2 Solution Design

We aimed to produce an easily extensible solution and take advantage of polymorphism. After much deliberation, we settled on using the strategy pattern, achieved through the use of the FilterStrategy and SelectionStrategy interfaces. This allowed us to decouple strategy logic from the rest of the code. Furthermore, as the creation of each player requires some complex creation logic in order to instantiate the strategies, we implemented the factory pattern.

We aimed to leave the main business logic largely untouched. PlayRound() for example was modified only as necessary. The only mention of strategies in the main Whist.java file is from the reading in of the properties.

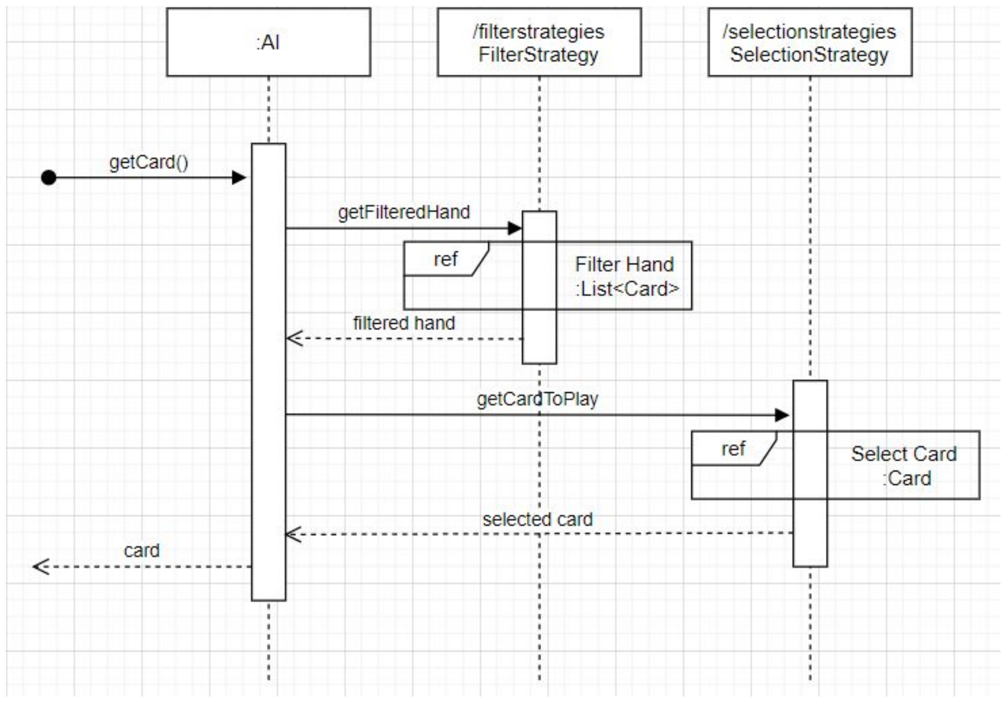
We did discuss many alternative approaches, but settled on this design for a number of reasons, as we will elaborate on. We hope *NERD Games* is happy with our solution.

2 Patterns and Principles

Our modifications were simple enough that we did not need to use a large number of patterns. However, GRASP principles were paramount to use when designing and implementing our code and classes.

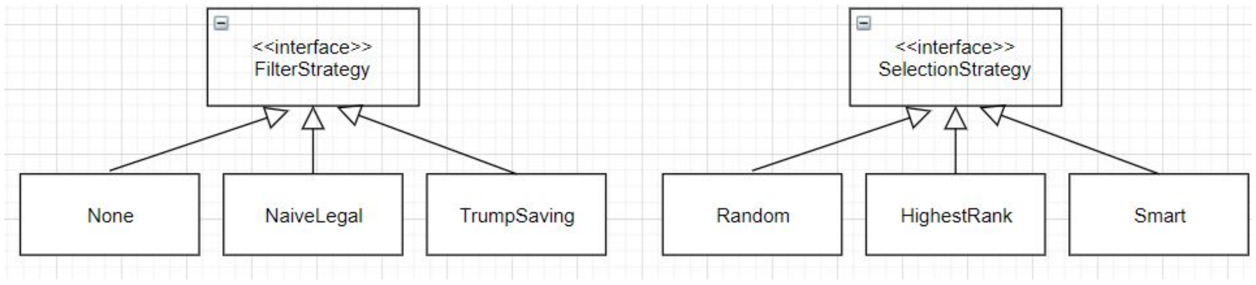
The main pattern we made use of was the strategy pattern. It made sense to us that the decision making logic or algorithms used by the AI would operate in this manner. We decided to have two interfaces, FilterStrategy, and SelectionStrategy. FilterStrategy has the method getFilteredHand, which takes a Hand (essentially a list of Cards), and the trump and lead suits, and then returns a list of cards.

This can be explained using the sequence diagram below



We implemented three different concrete FilterStrategies. These were NaiveLegal, TrumpSaving and None.

This implementation allowed us to retrieve a filtered list of cards for an NPC without including the filtering algorithms within the game context itself, instead we have a self contained class which does one thing and does it well.



It was essentially the same for the NPCs' selection step. We made a SelectionStrategy interface, and again gave it the three concrete implementations from the spec, while leaving it open for more ideas. This lets us decouple the NPCs' decision making logic from the main core loop.

However, we had to decide where to store all this information. We eventually resolved to encapsulate this data within a Player class. This class would store that player's type, HUMAN, DISABLED or AI, and if the latter, its filtering and selection strategies. This avoids the issues from storing in the main class (for example).

Due to the relative complexity of dynamically loading in classes and enums from a properties file, we found that the constructor for the Player class had begun to become unwieldy. We decided to utilise the Factory pattern in order to separate the creation responsibilities, promoting high cohesion. Our PlayerFactory class has a getPlayer function containing all the error handling necessary to ensure the Player instance that is returned was instantiated correctly. Implementing the PlayerFactory eases in creating more player objects in the

3 Alternative Solutions

There were a number of varied approaches that we considered. For example, rather than the Strategy interface pattern, we considered simply having a switch statement which would go through the three different strategies specified. We decided against this because it would mean low extendability and as the number of strategies increased, the function would grow longer and less maintainable.

We also considered folding the different Hand instances into the Player class as well. While we did debate this heavily, it was decided that it wasn’t worth modifying the existing code dramatically to achieve this, and the large number of Player.getHand type calls or iterations through the Players and their respective Hands would decrease code readability.

We were also uncertain of what to do when the properties file had invalid data. For example, what if the Player1 property was HUNAN instead of HUMAN, or the class for the specified SelectionStrategy was not found. We did consider having fallback values, for example setting to AI if the enum was not recognised, or using Random as a SelectionStrategy, but we instead decided to follow the example already set in the code (on the rules broken violation), and instead simply exit the game. The same approach was used for the StartingCards and WinningScore variables as well. We felt this allowed the user of the program to be able to better understand and see where all the variables were coming from. However, the random seed can be omitted to instead use the OS’ own randomness.

4 Smart Selection Strategy

We decided on a fairly simple algorithm for the Smart SelectionStrategy. The strategy has its own custom sorter, which sorts the cards in order, taking into account the trump and lead suits. The strategy uses the comparison function implemented by that comparator, and removes all cards that do not beat the current highest card in the trick. It then sorts these cards using the comparator, and selects the *lower* one.

This behaviour is demonstrated in Figures 1 and 2. Spades is the trump *and* lead suit, and thus the jack of spades is the current highest card in the trick. It is the player on the left's (Player1 internally) turn to play. The selection strategy discards all cards that will not beat the jack of spades, leaving it to consider the Ace, King and Queen of spades. It correctly identifies the 9 and 10 as losing cards, despite being part of the trump suit. It then selects the *lowest* of the winning cards it has, meaning it can save the King and Ace for later. It attempts to not waste high cards unnecessarily.

In the case that the player has no winning cards, it instead plays its lowest card. This means it gets rid of less useful cards at more optimal times.

In the case the player is starting, and has no current trick to base its decision off, it simply selects a random card.

The strategy performs best when the player is last, in which case it will always make the best decision. We considered implementing some kind of behaviour for when the player isn't last, perhaps some kind of semi-intelligent decision making process based on the number of turns left, number of cards left and even current scores, almost like a card counting strategy. This might allow it to make calculated decisions about how high to go in order to try and win the round. However, we decided this was beyond the scope of the project, and it would be difficult to produce a strategy which would actually be more efficient.

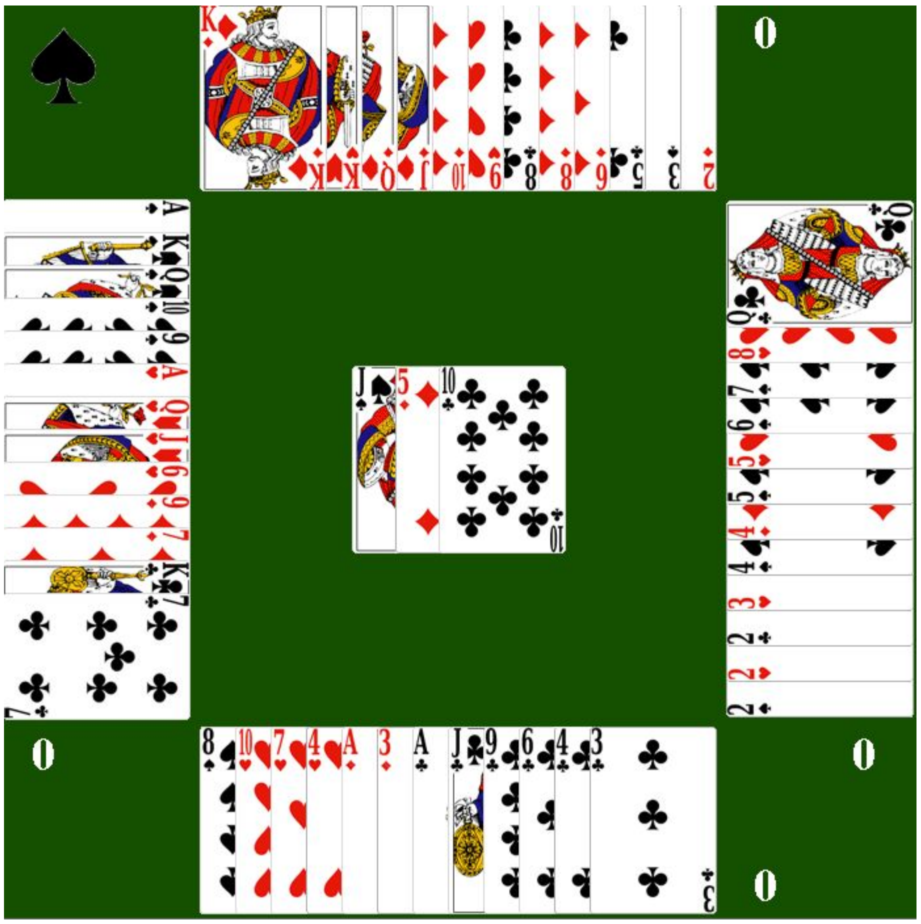


Figure 1

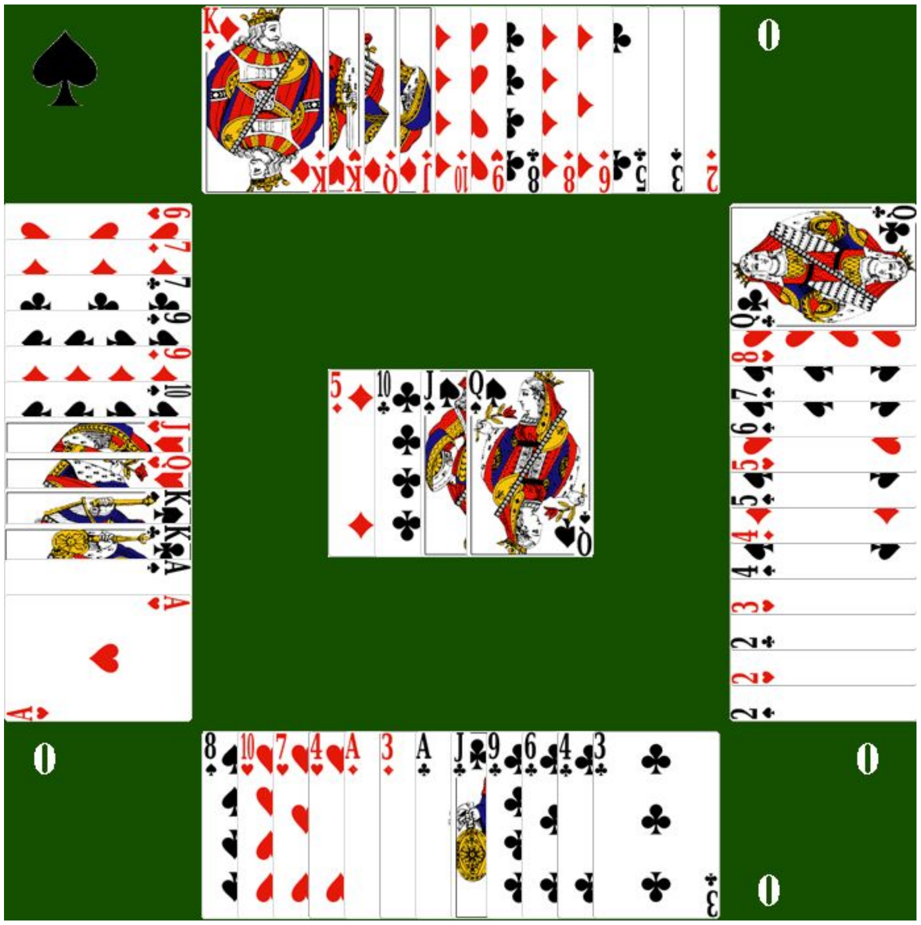
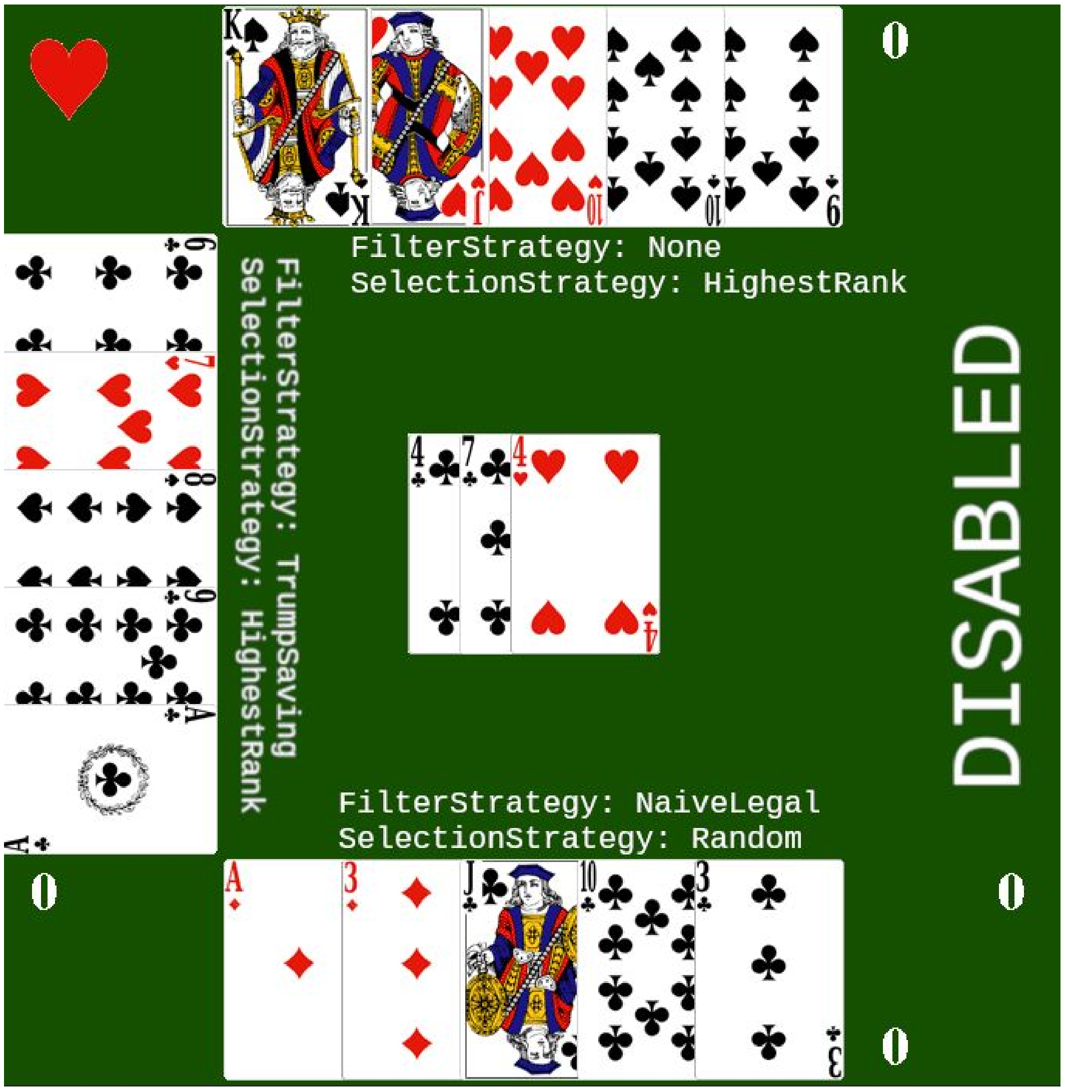


Figure 2

5 Legal Properties Diagram



```
FilterStrategy: None
SelectionStrategy: HighestRank
```

```
FilterStrategy: TrumpSaving
SelectionStrategy: HighestRank
```

```
FilterStrategy: NaiveLegal
SelectionStrategy: Random
```

This figure demonstrates how the game will look when legal.properties is used.
It shows :

- Player 0 (bottom) using NaiveLegal and Random strategies,
- Player 1 (left) using TrumpSaving and HighestRank
- Player 2 (top) using None and HighestRank.
- Player 3 (right) is disabled.

Worth noting is that this image was taken at the end of the first round, and thus each player has placed one card and has five remaining. This is because legal.properties sets StartingCards to 6. WinningScore is also 6.