Project 2: Whist – Design Analysis

SWEN30006 Software Modelling and Design

Team 126

Samuel Clerke (913032), Zhili Chen(980950), Jiehuang Shi(980709)

1 Introduction

The task at hand is to develop the current Whist card game. The original system only supports two types of players which are: human interactive players and random NPC players. For enhancements, two additional NPC types need to be designed: legal NPC players and smart NPC players. The legal NPC players will only play the card within the rules. The smart NPC players will record information of cards which have been played in each round. Based on it, they will decide a most reasonable and legal play for the current round. Property files also were needed so that games can be configured to run with different parameters that are loaded from the file. Further design improvements are also needed on the base implementation of Whist, to allow for future development and expansion of the game.

To complete this we set out first pulling out logic from the whist class and putting it into other more relevant classes. In all our design work we considered various design patterns ensuring the final project state was as flexible as possible.

2 Solution design

The original files provided presented a poor design. The single Whist Class was responsible for far too much game logic creating a largely non cohesive class relying heavily upon if and switch statements. This design did not lend itself to the desired expansion of NPC players or the future proofing of the design. The first logical step taken to improve upon this was the extraction of the abstract player class. By utilising polymorphism in this hierarchy structure as seen in **Figure I** we are able to handle multiple player types through a common interface while not restricting our design to future variations. The game specifications make the distinction of two key types of players that need implementation, being the various NPC characters and being able to take input from a human player. Naturally these two character types were implemented as child classes to the Abstract player class Implementing two common overloaded methods to get the card that the given player is playing depending on if they are leading or following. From here the next challenge was the implementation of various strategies that each NPC type uses. To implement these strategies the Strategy design pattern was used.

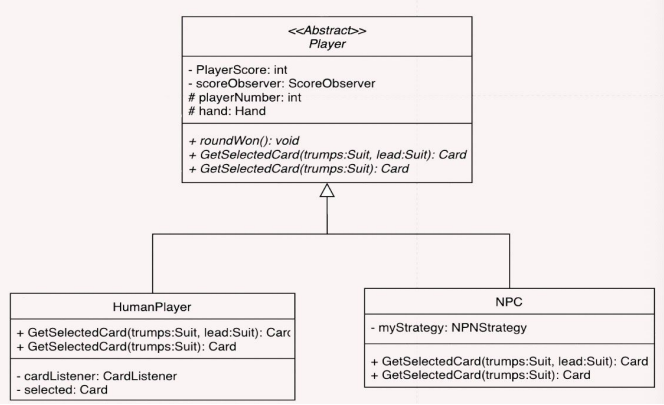

Figure I

The Strategy design pattern is a method where a common interface is implemented by a series of strategy classes depending upon predetermined conditions. This common interface allows for future proofing of design. To implement more strategies you simply have to add a new class that implements this interface to plug into your existing code with minimal interference. This is ideal for our system as the specifications state we must allow for a future addition of a super smart npc. Our implementation of the strategy design pattern can be seen in **Figure II.** Each NPC character in construction is assigned a strategy which it stores to call on when it needs to select a card. If the strategy pattern had not been used the logic for each strategy would need to be stored on the NPC class and differentiated between in a switch or if statement everytime the strategy needs to be called on. This would be a poorly cohesive solution with a massive file where every time a new strategy is added it needs to be inserted into further adding to the confusion.

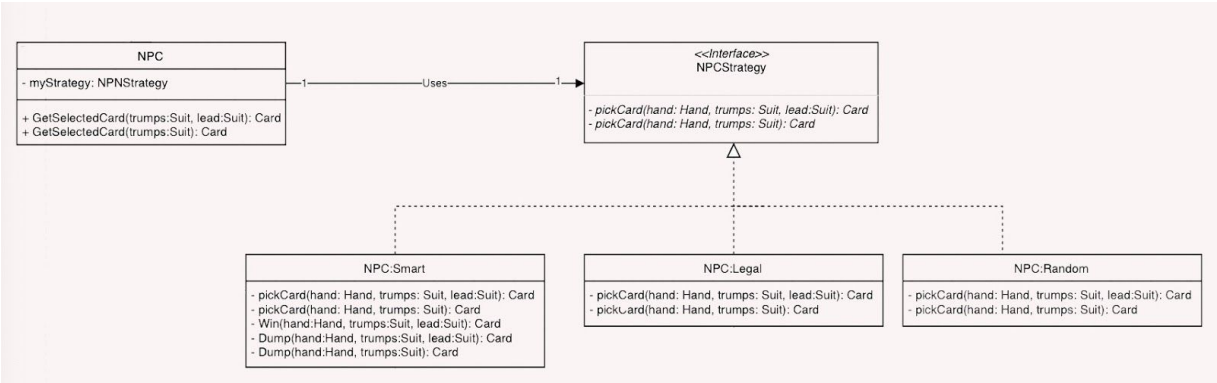

Figure II

With the architecture of the various strategies implemented the focus could be diverted to implementing some of these specific strategies. The logic for the random NPC was just pulled

out of the whist class but the legal npc had to be added by us. To implement this NPC it needed to have some concept of the game rules. This could have been hard coded as a logical statement checking which cards were legal to play and which were not. However this would be a poor design if future variations were needed to the game rules as the entire project would have multiple locations where these rules would need to be changed. This is inefficient and could lead to inconsistencies in game logic. The solution used was to create a new class responsible for handling all checks relating to the game rules and this class was accordingly named WhistRules. For efficient use of memory only one of these classes should ever exist. This lends itself to use of the Singleton design pattern. By using a static get instance function and private constructor we ensure that only a maximum of one instance is created at any given time and allow common access to this object and its methods. In this class we implemented a method to check if a given play is legal and a comparison function to determine between two different cards which was the winning one. With the implementation of this class the legal NPC could easily be implemented.

The next NPC needed was the smart NPC. This NPC further expanded on the functionality of the legal NPC needing to also select which card was the best play to make given the hand it had. Because the play also needs to be legal and the whist rules singleton could be used for it's useful functions however this does not provide all information necessary to play efficiently and maximise wins. To do this the NPC also needed access to what cards have already been played in both the current trick and which are out of circulation entirely. To do this a 'pool' of cards was created as a singleton class called PlayedCards. This class is especially important that it is a singleton as if multiple instances existed the full information on what cards have been played may not be available leading to unexpected behaviour in the NPC. The PlayedCards class also provides some public methods to calculate useful information to the NPC combining the information available from the cards played and what the NPC has in their hand.

A simple algorithm was implemented on the smart NPC to increase its performance over the legal NPC's. An approximation of this algorithm can be represented by a state machine diagram as displayed in **Figure III**. The algorithm revolves around choosing one of two strategies: Win or Dump.

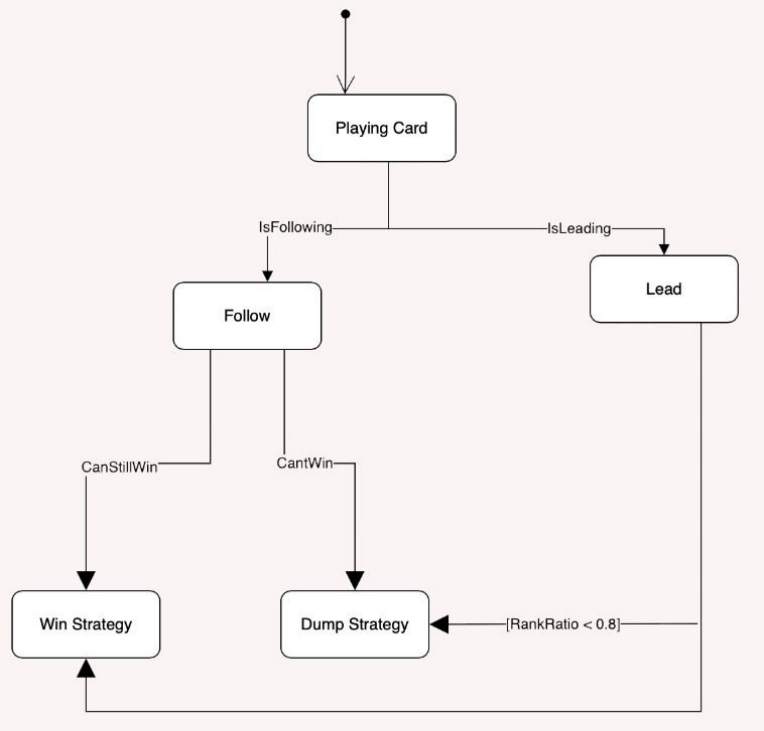

Figure III

For the Win strategy the NPC is aiming to maximise its chance of winning. This is done in 3 different ways depending upon how many other players have played cards in the given trick. If the NPC is playing last it can pick to play its lowest ranked legal card that will win the trick. If the NPC is leading it can look through all cards already played since the hands were dealt out. From this it calculates what we call the rank ratio comparing how many cards can beat the given card compared to how many of those were already played. If the majority or all of the cards were played the NPC knows it is relatively safe to play the card however this does not account for trumps. In the other cases the NPC is neither leading or finishing it will maximise its chances of a win by playing it's highest legal card as it does not know what players after it will play.

When the Dump strategy has been called the NPC is trying to set itself up to win future rounds. This is normally because it does not think it can win the round. When dumping the NPC will get rid of its lowest legal card. If the NPC can play any suit it will aim to clear out its suit that is not trumps and has the lowest number of cards. This is so that in future rounds it can legally trump more often.

To further increase the cohesion of the whist class more logic and parameters were pulled out of it. Because new player classes exist it is logical to store each player's score on that instance of the class. However moving the score off the Whist class means we need a way of determining when a player has won the game. The solution chosen to this problem was to implement an observer class that could notify the game when the win condition had been met. This class is the ScoreObserver class. When the game is set up this class is instantiated and in

its constructor it subscribes to the playerScore parameter. Everytime the score is updated on any player this observer class checks if that player has won and if so notifies the Whist class to end the game. This logic could have been implemented in the Whist class itself but would have been less cohesive than using an observer class.

Our other key feature included was the ability to use a seed for all random number generation. This was to increase our ability in testing the game ensuring repeatable results. If no seed is specified the game operates completely randomly. The major issue with implementing the seed was that the jcardgame package did not expose the seed with it's built in shuffling functions. To counter this we had to implement our own shuffler. This was easiest to do in it's own class to maintain cohesion of the Whist Class. To enhance efficiency the shuffler class was implemented as a singleton.

Our completed implementation of Whist meets the requirements set out in the specifications while enhancing the existing designs ability to implement new functionality in the future. The Whist class was refactored pulling out player logic and NPC strategies were implemented using the strategy design pattern. Parameters were set up to be read from a file to allow for configuring the game differently. Our overall final design is displayed in our design class diagram that has been provided along with this report.