Project 2: Whist-Design Analysis

SWEN30006 Software Modelling and Design

Team 3

Kuoyuan Li (1072843), Xuxu Xue(956895), Jiehuang Shi(980709)

Introduction

Our task for this project is to extend the current Whist card game. The original version only supports two types of players which are: human interactive players who play the game through the GUI, and random NPC players who play the game by randomly selecting a card without regard for the rules. For enhancements, a new type of NPC called advanced NPC needs to be designed. The advanced NPC will process cards filtering and then select the card to play from the filtered cards in each round. There are 3 filter approaches in our design, which are no filter strategy that indicates skipping the card filtering step, naive legal filter strategy and trump saving filter strategy. Advanced NPC has 3 selecting approaches which are random selection, highest rank selection and smart selection. The detailed implementation of these strategies will be illustrated in this report. Configurability also was required so that games can be configured to run with different parameters that are loaded from the property file. Modifications of the current design are also required to ensure the future development and expansion. 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.

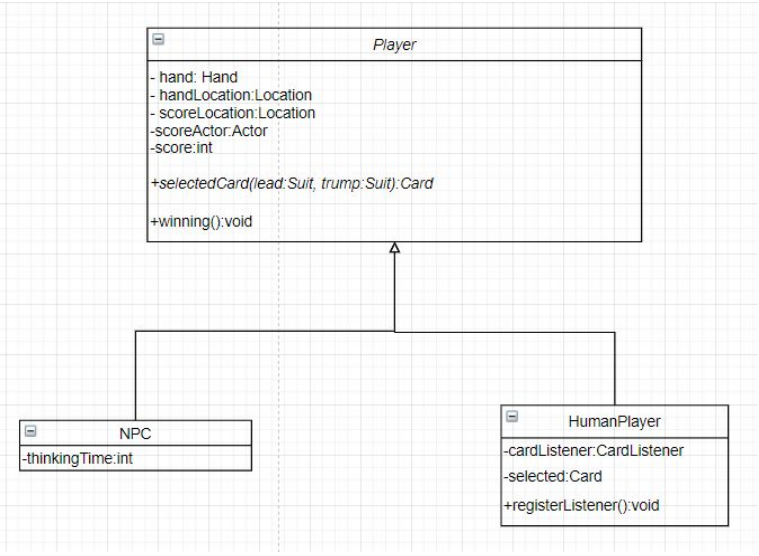

Figure I

Solution Design

The original system presented a poor design. The single Whist Class was responsible for the entire game logic, creating a largely non-cohesive class relying heavily upon conditional statements. This design did not lend itself to the desired expansion of advanced NPC or other future improvements. The first logical step taken 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 such as human player and NPC through a common abstract class Player. The abstract class defined their common behaviours such as select a card while not restricting our design to future variations. The project specification indicates the advanced NPC may need to follow a filter strategy to select a set of cards from the hand and must be capable of selecting a card based on its selection approach. From here the next challenge was the implementation of various filtering and selection approaches that advanced NPC uses. To implement these approaches the Strategy design pattern was used. For the creation of these strategy objects, the combination of the Factory and the Singleton design patterns was used.

As can be seen in our system sequence diagram (Figure II), as the game firstly loads, all players will be created based on the property file by using the PlayerFactory. PlayerFactory is a singleton class that takes the responsibility for creating player objects. The Factory design pattern provides a pure fabricated object that handles the creation. The logic of creating players is complex due to the diverse filtering and selection strategies. We can separate the creation responsibility into a cohesive helper factory to hide the complex logic, resulting in the high cohesion of Whist class. We only need one instance of the PlayerFactory to complete the creation work, Singleton pattern can ideally meet our requirement, ensuring only one PlayerFactory object being created to avoid redundant object instantiation.

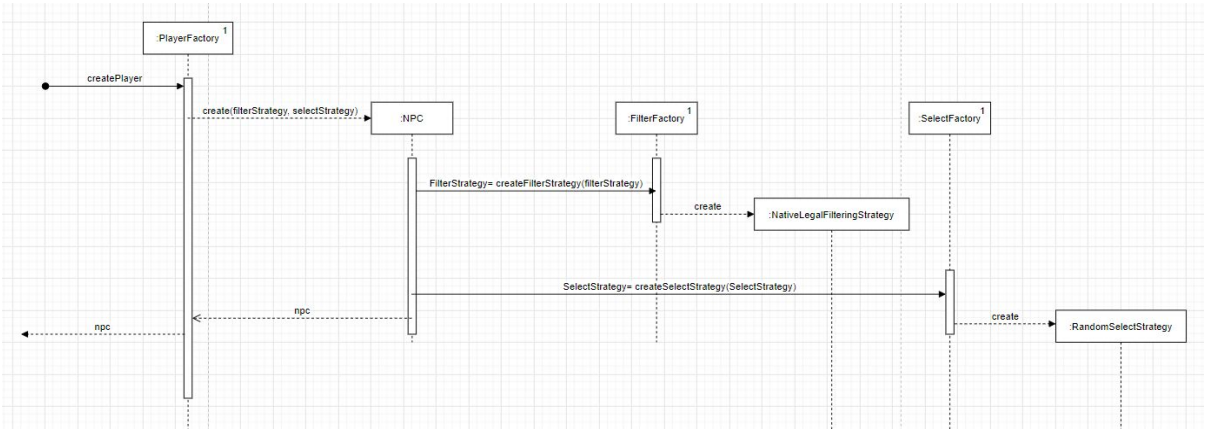

Figure II

As shown in Figure II, each NPC uses another two factories named FilterFactory and SelectFactory which are dedicated to the creation of NPC's filtering and selection strategies. Each factory has a method named createXXXStrategy, which takes a string parameter indicating the strategy type and uses the conditional statements (switch-case blocks) to determine which type of strategies to create. New strategies are created for every NPC, no strategy is stored in the factory. In our sequence diagram, we demonstrate a case where the property file specifies an NPC using the naive legal strategy for filtering and the random select strategy for selecting, leading to factories creating the nominated strategies. Similar to the PlayerFactory advantages, these two factories guarantee that the NPC class is highly cohesive by delegating the complex (it uses the switch-case block) creation responsibilities to the factory classes.

An alternative solution we've considered is removing the FilterFactory and the SelectFactory classes, implementing all creation logic in the PlayerFactory class. This benefits the reduction of the entire system coupling as the dependencies between factories vanish. However, the PlayerFactory would have a mix of creation jobs and no longer preserve highly focused and mangable. If we are required to modify or extend the current version of the Whist game with the introduction of new strategies, PlayerFactory would be constantly affected by the changes. Despite higher coupling, separating the creation responsibilities into several factory classes is more appropriate, keeping objects highly concentrated, understandable and manageable.

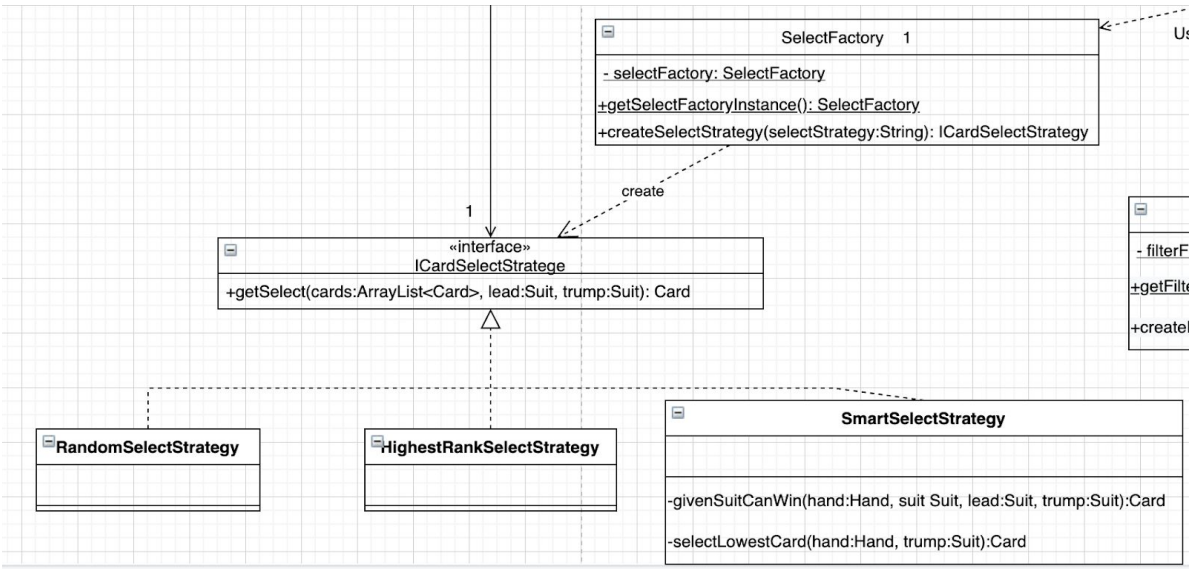

Figure III

In order to enable our Non-Player CardPlayers using different filtering approaches and selecting approaches, we decided to use Strategy design pattern. The strategy pattern defines a family algorithm, encapsulates each one and makes them interchangeable. The strategies let the algorithm change independently from the NPC that uses it. As

shown in Figure III, we separate the filtering approaches into 3 different filer strategy classes which all implement a common filtering strategy interface. Each filtering strategy class has a filter method which overrides the abstract method in the interface. The advantage of the strategy pattern is when an NPC is using these low-level encapsulations of behaviors(like NaiveLegalFilteringStrategy), the NPC itself doesn't need to know how the strategies are implemented or how they work. In addition, the strategy pattern is ideal for our system as it would be easier to implement more filtering or selecting approaches in the future design. Strategy pattern can help us to achieve high cohesion and reduce the coupling between the NPC and the filtering, selecting approaches.

The alternative solution for the filtering and selecting strategy pattern is to store all the filtering or selecting approaches in the NPC class. All the filtering/selecting approaches will be differentiated by the switch cases or If statement. However, this would be a poorly cohesive solution because if we add some new filtering or selecting approaches in the future, we have to add them into the NPC class while the NPC class should not know how the approaches are implemented. In addition, it would cause high coupling problems. For example, if one of the approaches goes wrong, it may affect other components in the NPC class and thus the whole class will be out of order. Therefore we think it would be better to apply the strategy pattern on this problem.

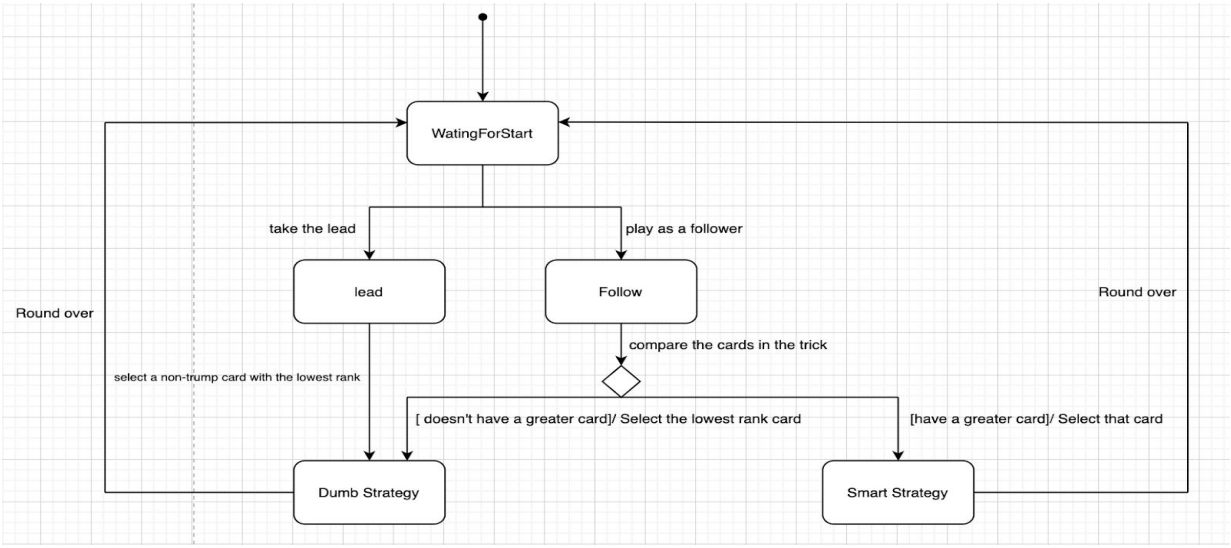

Figure IV

A simple algorithm was implemented on the smart selection strategy to increase NPC's performance over the combination of pre-defined select and filter approaches. Ours smart selection approach can be represented by a state machine diagram as displayed

in Figure IV. The algorithm revolves around choosing one of two strategies: Dump or Smart.

The Dump Strategy is aiming to keep the card with the highest rank or with trump suit in hand, for the sake of improving NPC's winning rate in the future rounds. When dumping, the NPC will get rid of its lowest card with a non-trump suit. There are two cases that the dump method will be called, when NPC is leading or no cards in hand can win in the current trick. According to the Whist rules, even if you played the highest card with a non-trump suit, others can beat you with the lowest card with trump suit. So our smart strategy would give up the leading position.

For the Smart Strategy, NPC will choose the lowest rank card which can win in the current trick. To save as many cards with trump suit as possible, the card with lead suit takes precedence over the trump suit in the selection. To do this a 'pool' of cards was created as a singleton class called PlayedCardHistory. This class is used to collect both the cards that have already been played in the current trick and are out of circulation entirely. In addition, PlayedCardHistory also provides some useful public methods for smart selection strategy.

The provided Whist game configurations are hard-coded in the system. We are required to make the system configurable through readling the property files. Our justified parameters in the property files are seed, winningScore, number of Start Cards, number of players, each player types, each NPC Filter strategy, select strategy, thinking time and whether to enforce rules. We deemed these parameters should be alterable based on player expectation on the game rules. For other parameters such as handWidth, trickWidth, player's hand and score locations in the GUI, since these elements are related to the rendering of the user interface instead of the logic and rules of the game, we decided to remain the original hard-coded style. For now, the constructor of Whist class reads the property file before the game starts, assigning the property values to the corresponding variables in the game. We also realized the scenario in which a property file does not list sufficient information to set up the game. If the required parameters do not exist in the property file, a set of default values will be loaded and the game is still runnable.

The last key feature in this project is the ability to support repeated runs with a fixed random seed. The game is operated randomly if there is no seed to control the shuffler. To maintain the cohesion of the Whist Class. We create a singleton class ShuffleToPlayer to assign cards to each player with a given value of seed.