Project 2: Whist – Design Analysis

SWEN30006 Software Modelling and Design

Team team-number 49

Harshini Chidambara Raj (998465), Samer Adra (1007904), Paul Ou (888653)

1 Introduction

The task for this project was to modify the design of the existing Whist card game so as to improve its configurability, add new Non-Playable Cardplayers, and leave the design open for extension or modification in the future.

2 Solution Design

2.1 Summary

The Whist card game's design has been modified to increase the game's configurability and to facilitate the modification and/or addition of new game features in the future. In addition, two types of NPCs were implemented, a legal NPC that plays a random card from all the cards that can be legally played, and a smart NPC that chooses a card logically based on the information of the cards that have already been played.

Furthermore, three property files were added supporting repeatable runs: original.properties, legal.properties file and smart.properties. Each with its own number of NPCs, NPC types, number of start cards and winning score.

Moreover, with respect to software design, the Whist class was divided into different classes each with its own unique purpose, thus leaving the Whist class with only code related to the running of the game.

A graphics class was created to separate the game's logic and strategy from its layout and visual effects.

For higher cohesion and lower coupling, a player class was created. Having a player class also ensured a lower representational gap between software components and the domain. In addition, a PlayerFactory was implemented to create players using the factory pattern approach in order not to affect the Whist class if further player types or strategies were implemented.

Finally, a strategy pattern was used to implement the player strategies in order to separate the game's various strategies from the player class; that was done by creating a separate interface for strategies.

2.2 Graphics Class

The Whist class originally had many Location object coordinates relating to their positions on the screen the different game objects would appear while the game was running. This approach cluttered up the Whist class and reduced code reusability. Thus, we created a Graphics class as a pure fabrication object that contained only static functions and attributes so they could be accessed by the Whist game. Doing this resulted in higher cohesion and lower coupling for both the Whist game and Graphics classes.

Having all or most of the code relating to the Graphics meant separating the logic and strategy parts of the game from the look and layout of it. This leaves the game strategy open for extension without it affecting the graphics. It also means that if in the future, the graphics was to be changed, it could be done so without changing the game logic and strategy.

2.3 Player Class

To ensure minimal representational gap between the domain and the software, we created a player class while classes like the game and card classes already existed in the JCardGame package. In the future, if changes were to be made to any aspect of the players, they could be made in the player class without it affecting the program. The existence of the player class also means that the number of players can be customised if we choose to do so.

This is better than the original design where every attribute of the player was stored in arrays of size four in the Whist class. This means that if one was to change the number of players in the future, the size of every array would need to be changed, which would be tedious and makes the code less reusable. With the player class with all of the player attributes, there only needs to be one array of players in the game class. The size of the array could be configured at runtime if necessary. Separating the player attributes from the Whist game means that both the Whist and Player classes are more focused with more specific responsibilities, resulting in higher cohesion and lower coupling than before.

2.4 Factory Pattern for Creating Players

Based on the Creator Pattern, we would have created the players in the Whist class, since the game itself contains and uses the player objects. The creation of the player objects had the potential to become complex due to the player types and their strategies being one of the configurable aspects of the game. Future additions or changes to the player types would require changes to the Whist game class if the player objects were created there.

Thus, we used the Factory Pattern to create a pure fabrication object, the PlayerFactory, whose main goal was to deal with the creation of the players before the start of the game. The PlayerFactory would read and parse the property file since that was where the player types were specified. Then a static factory method,

createPlayers(), would be accessed from the Whist class and created there. This is a cleaner approach than reading property files and creating players in the Whist game class, since it is more cohesive as both classes have more focused responsibilities. If in the future changes need to be made to the player types or strategies or their creation, that can be done without affecting the main Whist class.

2.5 Strategy Pattern vs Inheritance for Implementing Player Strategy

We have chosen to implement the additional NPC types by adapting the strategy pattern, including a legal strategy, a smart strategy and a random strategy which preserves the original behavior where NPC just chooses a random card to play from their hand. To implement it, we added a public strategy interface and three public strategy classes that implements this common interface. In this case, every time we create a new player, we simply pass the type of the player as a parameter to the player factory, and the player created would hold an instance of the corresponding strategy class. In addition, this also makes the system more extensible. If we need to change some of the existing behavior in the future, we only need to change the algorithms in one of the strategy classes. Likewise if we need to add additional types of players, we simply add a new strategy class that holds the corresponding algorithm.

One of the options we considered for building the different NPC types is applying polymorphism and inheritance to the player class, with player class being an abstract class, and the concrete classes, smart player, legal player, and random player inheriting from it. However, in this case, the only difference between different NPC is the strategy they use to play the game, which is a single function, so it would be repetitive and unnecessary to create a new class for each of them. By using the strategy pattern, we can implement various strategies for playing the game in a separate interface, and allow the specific algorithm to be chosen at runtime by reading from the properties files. This also allows better decoupling between the strategy and the player class, so it is easier to maintain and refactor if any changes are needed.

2.6 Smart NPC Strategy

Experienced Whist players memorise the cards that have been played so that they can know if they possess the highest card of a certain suit and play it in the following rounds to secure the highest possible amount of points.

As such, a hand consisting of all 52 cards, called cardsThrown, was created to be accessed by the smart players. After each round played, the cards that have been thrown into the trick are removed from this hand, so that the smart players can know the current highest card of each suit and use this information to their advantage.

After a player initiates a round by throwing a card, the smart NPC checks if it has a card from the same suit as the lead suit. If it does, it checks if it currently possesses the highest card of this suit by looking at the cardsThrown hand. If the smart NPC

has the highest card in its hand, it plays that card; otherwise, it plays the smallest card it has in that suit.

However, if the smart NPC does not have a card from the same suit as the lead suit, it checks if it has cards of the trump suit. If it has cards from the trump suit, it will throw the smallest card it has in that suit. This approach is used because it will not be beneficial if the smart NPC plays a high card from the trump suit, since any card from this suit will beat a lead suit card.

On the other hand, if the smart NPC does not have neither a lead suit card nor a trump suit card, it plays the smallest card from another suit.

Note that this approach can be made more smarter in the future, and we have developed our code in such a way that it will be very easy for future developers to edit the currently existing smart NPC code, or simply just implement a new player type.

2.7 Refactoring

Once the code relating to the player attributes or strategy, as well as the graphics was removed from the Whist class, only the code relating to the running of the Whist game was left. Some of it was repeated and most of it was reusable and thus we split it into smaller private helper functions in the Whist class.

3 Conclusion

As you can see, the Whist card game now has increased configurability, and its design has been made such that new game features and strategies can easily be added in the future.

The game now also has two more NPCs, a legal NPC and a smart NPC. In addition, three property files were created, all supporting repeatable runs.

The open-close principle was applied, so the existing behaviour of the game was preserved but was left open for extension in the future.