SWEN30006 Software Modelling and Design
Project 2 : Whist

Team 101:

- William Putra Intan (955545)
- Patricia Angelica Budiman (1012861)
- Franklin Aldo Darmansa (1025392)

Introduction

This project is based on the provided Whist game which implements the JGameCard library. Whist game is played using a standard 52-card deck involving 4 individual players in which the game ends when a player scores the determined number of scores. The provided Whist game currently supports two types of players which are human interactive players and a type of NPC player who selects random cards from its hand without following any rules. Our task is to enhance the current design and implement additional NPC types : Legal and Smart which we implement by applying the GRASP principles and GoF design patterns.

The Legal NPC plays a random card from its hand while following the set of rules and the Smart NPC plays the game by making a legal choice based on the relevant information and must be smarter than the Legal NPC and they must assume that other players are playing legally. The rule which the players must follow are: lead player may play any card from their hand, players must play a card of the same suit as the lead card if they have one or if they do not have the card, they may choose to play any card, the winner of a round will lead in the next round.

In this report, we will discuss the GRASP and GoF design patterns used in order to achieve the desired outcome. Firstly, we will discuss which patterns and the reasons and advantages of implementing the chosen patterns which is then followed by discussing the alternative patterns that we considered and the reasons of not implementing the patterns.

Solution Design

Summary

In this solution design, we decided to use 3 design patterns of the Gang of Four design patterns: Factory, Observer and Strategy. The factory design pattern is applied in creating the factory class which will be used by the player to create the strategy that they need. The observer pattern is used by the subject in which is the Whist game to notify the observers which in this case are the players in the case when there are changes to the cards displayed on the table such as when another player has put out a card. The strategy pattern is used in order to create the different strategies where the different types of players (NPCs) will use to select the card to be played.

For the GRASP principles, our team has applied 5 principles: Information expert, Creator, Low Coupling, Polymorphism and Protected Variations. The information expert is applied by creating the Player class who are responsible for playing the card and also getting the information of the round and the principle is also applied by creating a Game Properties class whose responsibility is to handle the currently used properties. The creator principle is used by creating the factory which is responsible for creating the different strategies depending on the player's type. The low coupling principle is applied in the case of the Rank and Suit enumerations as these enums are required by many classes in the solution. Polymorphism is applied by creating the SelectionStrategy interface which is implemented by the different strategies to choose the card to be played. Lastly, the protected variations principle can be seen through the use of the strategy design patterns allowing the system to hide specific information about each strategy.

The 3 design patterns and 4 principles applied can be seen from the Design Pattern diagram that our team has decided to include for easier understanding.

Patterns

Factory Design Pattern

The factory design pattern is used to create the different strategies of the different players such as the NPC who plays randomly, legal NPC and smart NPC because these NPC players implement different strategies in selecting the card to play. By implementing the factory pattern, which is used by the Player class, this allows the players to pass the responsibility of creating and deciding which strategy the player would have to use to the factory. The player would not need to know the type and details of strategy and how it is chosen and therefore, hide the logic behind the creation of the desired strategies. The players only need to have one instance of the factory inside the class which then the player needs to use the method of that factory to get the instance of the required strategy and the problem of choosing the appropriate strategy is no longer the player’s responsibility.

Strategy Design Pattern

Strategy design pattern has a main functionality to assist a class in managing the strategies that are implemented in that class. This implementation increases the flexibility of adding and removing strategies. In this project, the class that uses the strategy design pattern is player class. The reason behind is player class has various types of players, which consist of real player (human) and NPCs. As it mentioned above, the NPCs are divided into three different behaviours, that are random, legal and smart. Each of the NPC processes different algorithms on the input which results in different output. With strategy design pattern, the Player class only needs to access the interface that is implemented by those strategies classes called Selection Strategy interface, and therefore able to use those legal, smart and random strategies. However, it is a little bit different in this case, as the strategy design pattern is combined and under factory design pattern. Instead of Player class directly using the Selection Strategy interface to choose each NPC strategy, the Player class uses Selection Strategy Factory class to create a player, and the Selection Strategy Factory class accesses the Selection Strategy interface to select the intended strategy to apply to that player. Therefore, the developer will gain flexibility on creating or removing strategies and avoid having a big if-else statement or switch-case in the factory class.
Therefore, it will harden the development of the code in the future as having all of the NPCs strategies in the same class.

Observer Design Pattern

The observer pattern has the functionalities to observe any changes during the game. After each player in the game has moved, all players in the observer list will get notified about the change. This will be very useful for each player to decide which would be the best move based on the observation. The pattern allows easy modification when other players decide to observe/stop observing the current game. The pattern sends data in an efficient manner where it only sends data when things get updated, instead of passing a class where there are only minor changes. Observer patterns provide a loosely coupled design between objects that interact. This enables a more flexible change if it is necessary.

Principles

Information Experts

Player Class
As the players need to choose which card to play, we introduced the method getCard whose responsibility is specifically to make a decision of which card to use in the current round based on the strategy created by the factory depending on the player's type. Thus, by applying the information expert principle, it is only natural for the player to have the responsibility of choosing the card for the round.

Game Properties
As the game will be played based on the different properties, we decided to create a class whose responsibility is to 'read' the chosen properties and store these properties in order to achieve the Information Expert principle.

Creator

There are different types of NPCs and they all have a different strategy depending on their type. We then decided to apply the creator principle to pass the responsibility of creating the different strategies to the factory class and thus achieve low coupling between the players and the strategies. As if this creator pattern is not applied, there will be high coupling between the player and the different strategies and thus this will provide a lower maintenance for the future in case more strategies are to be created.

Low Coupling

Initially, the enum 'Rank' and 'Suit' were in the Whist class. We decided to separate each enum to have their own enum class. By making their own enum classes, this would result in low coupling as there are other classes other than 'Whist' which will refer to the Suit and Rank enum, such as the smart strategy and Legal Strategy. If the enums do not have their own classes, other classes would have to access them through the Whist class and result in high cohesion between these classes and the Whist class.

Polymorphism

In order to achieve the different strategies where the players will select a card to be played depending on the chosen strategy, the polymorphism principle is applied on the solution. The principle is applied by creating the 'selectionStrategy' interface with the 'selectCard' method in which the different strategies will implement and each strategy will have different ways of choosing the card depending on whether the strategy follows the rule or not and information of the game. This principle allows more flexibility in making changes and adding more strategies in the future.

Protected Variation

We applied the protected variation principle by applying the strategy pattern as each pattern has different ways of selecting the card to be played depending whether they are following the rules and if they have the information to select the card. Thus, if there are any changes in any one of the strategies, we just have to make changes to that specific strategy class and do not impact any other classes. As other classes who would need this strategy would just need to call the factory pattern which will return an instance of the required strategy without knowing how the strategy works.

Alternative solutions

Using Inheritance instead of Interface for strategies

The first alternative solution that we were thinking of is using inheritance for the NPCs. Since smart NPC and legal NPC have something in common, where smart NPC must have a legal move, therefore the legal NPC could be the parent class and smart NPC inherit from the legal class and add its own feature to be smart. However, this model is not applicable to random NPC, where it does not play legally, and therefore, we need another new class which might increase coupling. In addition, this model is not flexible for the smart class, as if the smart class wants to play illegally at a time in future, we have to rearrange the class which is very inefficient. Therefore, inheritance is not a good option for this case.

Addition of “Game Information” class

We have thinked and tried to implement another class called “game information” which stores the data of the game information on the table at the current round. The aim was to make the code cleaner and more precise but this implementation would increase more coupling across class. This class would be updated every time a player generates their move. Players would need this information in order to make judgement on what card they should select for the play. The game information class may be coupled to the “whist” class and “player” class. This would be harder for further modification.