SWEN30006
Project 2: Oh Heaven

Jiaqi Zhuang 1122155, Yuntao Lu 1166487, Yingyi Luan 1179002

Once we have received the code package, we firstly have decided to split them up into different classes in order to make the code more readable.

Property

The Property Class is mainly responsible for loading property files and configuring the parameters used in the game. ***Protected variation*** is applied in this case, so that the variation of property files will not have undesirable effects on other classes (Open-Closed Principle). Apart from configuring, the Properties class can also be used to provide an ArrayList of players in the *configPlayer() method* since Properties Class knows the number of players and the specified category of each player **(*Information expert*).**
Relevant property parameters are assigned to Property class(e.g. nbStartCards, handwidths) so that Oh Heaven can focus on game logic and UI jobs. It reduces the workload of Oh Heaven class and therefore, a lower coupling has been achieved.

Round

We decided to create a Round class by using ***pure fabrication***. This class is mainly responsible for handling all the information in each round. If we follow the original design where all of the information is stored in OhHeaven Class, that class would be too bloated and the cohesion would be poorer.

Apart from that, we need to consider what useful information is required to build up the legal and smart strategy. In other words, we need to pass enough useful attributes to the round class so that the player is able to use this information to make the right decisions (to win the bids). For example, attributes such as *tricks (details of cards played on the board), lead suit and trump suit* have to be given and updated on a regular basis where it could be easily accessed by the player.

Player

Human and NPC classes inherit the abstract class Player. It is an application of **polymorphism**, where players can behave differently depending on whether the player is human or not. For humanPlayers, cardListener is set up beforehand and the card can

be selected when the player double clicks the mouse. While for NPC, different strategies will be applied to determine which card to play next.
Attributes such as score, trick and bid are initiated once the player (instance) is created. These attributes will be constantly updated if the game is ended, the player wins a round or starts a new game.

StrategyFactory

StrategyFactory is designed in order to create strategy classes. Although it is not a problem if we have direct association between NPC class and the strategy class (as long as NPC is the **Creator** of strategy), It would be better if assigning **a pure Fabricated** StrategyFactory object to this job(**Factory Pattern**). Thus, direct coupling between NPC and IStrategy interface can be separated with the idea of **Indirection**. Not only will the coupling be lowered, the reusability could be improved as well. For instance, if we want to add more Strategy Classes or induce a more complicated Strategy creation job, we just have to modify *createStrategy* method within StrategyFactory class without affecting Player class, so that the Player class can focus on handling the it's own jobs(eg. NPC's behavior).

Furthermore, only one instance of StrategyFactory would be used, so we have a **Singleton Pattern** here. Static attribute *instance* and static method *getInstance()* is created within the class. In terms of class relationship, class NPC needs to use StrategyFactory in order to get the next card when NPC plays its turn (method *play()*) , and NPC needs attribute visibility StrategyFactory.

Choice of pattern for strategy

Now we are going to explain any features we have added in this project and the types of design patterns & principles that would be used for these features. First of all, we notice that there are 3 NPCs and 1 human player in total in this game. Then, we are going to select one NPC to support 3 different modes: original mode—play cards without any restriction, legal mode—have to play cards legally (the player who does not play the lead of a round must play the same suit card as the lead suit) and smart mode—advanced strategies that would be applied on one NPC so that it has more possibility to win the game. In this case, there will be 3 design patterns for us to choose, which are adapter, strategy and decorator respectively.

Alternative1: Adapter

Adapter is used to resolve incompatible interfaces problems which is generally used if the classes are unable to implement from the same interface (as they have similar

behavior but different methods). In our design model, it is not necessary to create intermediate adapters for the interface since different properties only affect the card playing strategy and does not change the behavior of game logic (such as play two cards at the same time, flip the cards etc.) We will ensure that all the implementing classes will have the same methods getNext(Player player, Round round) and the outcome of these classes will be the same—-a single piece of card. Thus, design pattern-Adapter will be discarded.

Alternative2: Decorator

Decorator contains the dynamic behaviors inherited from the abstract interface and this interface allows unlimited decorator layers to be added to the core object. In this case, we need to be aware of the relationship between our 3 properties. We treat the original property as the base class for decorators and in fact, we realized that legal property has more restrictions than the original property rather than adding more optional embellishments to the original property. Same idea with the smart property, the behaviors of smart property are getting more restricted from legal property, which conflicts with the idea of the decorator. Thus, design pattern-decorator will be discarded.

Our choice: Strategy

We should then consider whether design pattern-strategy is suitable in Oh Heaven game. If we look at the 3 different properties, the legal property algorithm is built based on the random property (but with more restrictions) while smart property requires the NPC player to play cards legally as well as more cleverly. Based on this information, we could conclude that the algorithms of these three properties are different to each other, but they are still related. The solution is to define each algorithm in a separate class with a common interface, which is the idea of strategy design pattern. We decided to create three different classes which are RandomStrategy, LegalStrategy and SmartStrategy respectively and they will implement the interface called Istrategy, with one abstract method getNext(Player player, Round round). Each Strategy class behaves differently based on its category and if we use the if-else method within just one class, it will be too bloated. Polymorphism is introduced in order to reduce coupling and increase cohesion.

Smart Strategy

So far, the overall logic is clear, what we need to do next is to modify the smart strategy so we need to make sure that there will be one NPC who has a higher chance to win the game (let us call it NPC-smart). We decided to create two new classes called

notReachedBidStrategy and ReachedBidStrategy which *implement* interface IBidStrategy. This is because card selecting algorithms vary in different scenarios, former is the strategy when the number of tricks is not reached to the bid and latter is when it's reached. If the number of tricks has not been reached, we should do as much as possible to help the player win each round. While if it's reached, we should do our best to avoid winning the game. Thus, we need to ensure that notReachedBidStrategy could help the NPC to win the trick and ReachedBidStrategy could help NPC to lose the trick. With *polymorphism*, we can make the class less bloated and increase cohesion.

Association relationship between SmartStrategy and IBidStrategy interface is given since SmartStrategy has to use a selection method in IBidStrategy interface in order to get the next card.

The basic logic of *notReachedBidStrategy* is: if NPC-smart plays the lead of a round, it would play the largest (rank) card among all suits (except the trump suit) so it has a high possibility of winning a trick. If the NPC-smart does not lead the round, the NPC-smart will check the winning card on the board, then make decisions. If the NPC-smart does not have a card that can beat the winning card, it will play the smallest rank card (must be legal). Otherwise, it will play the largest rank card (must be legal).

Meanwhile, if NPC-smart's trick reaches the bid, the NPC-smart will try its best to lose the all following round. Thus, the logic of ReachedBidStrategy is: always play the card just smaller than winning card(with lead suit), or play the biggest non-lead suit card (if there is no lead suit in hand and except trump suit card) so that not only NPC-smart could get rid of other big cards, but also lose the trick (in order to win the bid).