Project1: Whist – Design Analysis

SWEN30006 Software Modelling and Design

Author 1(Wei Ge), Author 2(Chang Shen), Author 3(Xubin Zou)

Alternative Solution:

For the alternative solution, we used the *Composite Pattern* to control all the NPC and players, tried to separate the operations on human and npc out from ***Whist*** class for better extension in the future. However, we found that the printing of all the information need to store in the ***Whist*** class. This is the reason we give up the *Composite Pattern* and further remoulding in ***Whist*** class.

Introduction:

This report focuses on the techniques we used and processed we had in the original Whist code and tried to improve the code to satisfy the GRASP by using different patterns.

Assumption:

We assume the maximum number of all players is four.

Solution Design:

Summary:

In this project, we used two design patterns to finish the task. They are ***Strategy Pattern*** and ***Decorator Pattern***. We also recognized four main elements in the project: Card, Human, NPC, and Rule.

At first, we ignore the Card part while poker is the standard Card; there are few opportunities for change in the future.

We separated the player's collection from the main class for the human parts, allowing us to add new players freely. We ignored the playing method's extension of players while thinking there are few opportunities to change its manipulation.

For the design, we decided to use the ***Strategy Pattern*** in our Card Selecting part and ***Decorator Pattern*** in our Card Filtering part. We think these two patterns are the most fitted for this task.

We generating a pure fabrication class to connect the main class with other component following the open-closed principle. If we make extension on human players and npc in the future, it will be the only class which will be impacted.

At last, we made some changes to some variable names and corrected the code style to formal.

Operation Process:

First of all, everything starts in the Whist class. In this class, we will read the property file first. There is all the information about each NPC. Then we will generate the initial score of each player includes NPC. After that, the Whist will create the NPC and regular player by the information we read from the property file. Here is the point that all the initial setting has been set up. Then the game start. The **playRound()** method will run.

Patterns and Principles:

As discussed above, we used the ***Strategy Pattern*** on our Card selecting part. Using this design pattern, we can use different Card selecting algorithms easily and add any new selecting algorithm without a bunch of code. This implements the low coupling. In our code, if the people want to add any new selecting algorithms into the game, they can create a new selecting algorithm strategy and extends from ***CardSelectingStrategy*** class.

In our task, we have three selection algorithms. They are **HighestSelect, RandomSelect, and SmartSelect**. We let them become the child class of

CardSelectingStrategy. By this pattern, the NPC does not need to know how each selecting algorithm works, but they can use it. It achieves the *Low Coupling*, *high cohesion*, *and Information expert.*

Also, this satisfied The *Open-Closed principle*. **If people want to make any changes, they do not need to change the code we already write. They can add the new one straightforwardly.**

Secondly, we used the ***Decorator Pattern*** in our card Filtering part. The reason that we chose ***Decorator Pattern*** is the Filtering part is optional. We create a ***FilterNPC interface*** to connect all the filtering algorithms with the game. Then we have ***FilteringDecorator*** class, which is the parent class of all the Filtering algorithms classes. This class is the base decorator in the ***Decorator Pattern***. The ***TrumpSavingDecorator*** and ***NaiveLegalDecorator*** are the Filtering algorithms classes. They all extend from the ***FilteringDecorator***. By using this pattern, we can install any Filtering algorithm on NPC. Keeping the *Open-Closed Principle* is another reason that we use this pattern, **people can add new decorator without changing anything which has existed.**

Then, we have ***ManipulatePlayer*** class, which is used to control all the NPC and regular players. This class makes our project with *Low Coupling* and *Pure Fabrication*. This class is also satisfied with the *Open-Closed Principle* because it is easy to do any extension on players but do not need to change the code we already write.

Property File:

We create an interface for creating property which named **File.** In each property file, for filtering part, 0 means non filtering method, 1 means naïve legal method, 2 means trump saving method. For selecting part, 0 means random select method, 1 means highest select method and 2 means smart select method. In this File, we only need to change the variable which named "*NUMBER_OF_FILTERING_METHOD"* and

"*NUMBER_OF_SELECTING_METHOD*" when we add new Filtering method and Selecting

Method.
"