Project 2 Report

SWEN30006 Software Modelling and Design

Junzhi Ning (1086241)
Ziqiang Li (1173898)
Lujia Yang (1174148)

May 28, 2022

In this project, we aim to extend and modify the existing version of the game NERDI so as to make improvements over the maintainability, readability and configurability by adhering to the GoF patterns and GRASP principles.

1 Task1

1.1 Rationale

In this task, we need to design several algorithms that allow the NPC to play the card legally, which means that NPC must follow the lead suit if they are not leading in a turn. Then For part b, the algorithm should be improved so that the NPC is able to play the game with some winning strategies.

1.2 Implementation

We created an abstract class called *player*, where *HumanPlayer* and *NonHumanPlayer* are the inheritance of the class *player*. We override the playOneCard method to perform different features of a real player and the NPC. If we create *HumanPlayer* and *NonHumanPlayer* seperately, it would lead to redundant code when we write their common mutators and accessors repeatedly. So, the implementation of polymorphism allows us to handle variations more easily by calling the playOneCard method straight away since we do not need to differentiate different players in the playRound method. To be more specific, the *playOneCard* method for the real player simply requires user's click, which is achieved by the function*setTouchEnabled* and the variable *selectedCard*. Whereas the NonHumanPlayer will play a card by the function *generateOneMove* from *PLAYERSTRATEGY*. Besides, the variables *score, nbWon, bid* and *hand* should be stored in the *player* class instead of the class *Oh Heaven* according to information expert principle.

Moreover, we employed the strategy pattern coupled with the factory method in order to bring out the concrete strategies designed for the NPC players. Factory method creates and generates different strategies, including *randomStrategy, legalStrategy* and *smartStrategy*. Specifically, The NPC player randomly plays a card when applying randomStrategy. When applying legalStrategy,

the NPC player has to follow the game rule if they have the corresponding suit (and not leading the turn).

While in smartStrategy, the NPC player will need to present some relevant techniques and approaches, such as sorting the cardList, come up with the biggest and smallest card and categorise lead suit and trump suit, in order to decide which card can be played to maximise the result. The full algorithm is to check whether the current player is the lead, if so, then check if the lead player can play the biggest trump suit card if he has one, Otherwise he plays a biggest card of any suit. If the player is not a "lead", then check if the player has any card with the lead suit. If the player has one, then it plays the biggest lead suit card if the biggest lead suit card greater than the current winning card. If not, then play the lead suit card with the lowest rank. If the player doesn't have any lead suit card, then the player checks whether the player has any trump suit card or not. If so, the game agent then plays the trump suit card with the largest rank. Otherwise, the lowest card on hand of any suit will be played

Building on this, the factory method is also a creator of those strategies, which increases the cohesion and avoid tight coupling between the class *NonHumanPlayer* and its concrete strategies.

Thus, strategy pattern and factory method not only assist us to switch and apply different algorithms without effort, but also make the code structure easier to maintain and present more tolerance for future variations. If new strategies are required, we can add them in without interfering the existing code. On top of this, we applied singleton pattern in *strategyFactory* and *playerFactory* to ensure the class has only one instance with a global access point.

To summarise for task 1, we have added in multiple patterns and methods, it makes the whole structure more focused, maintainable and reusable.

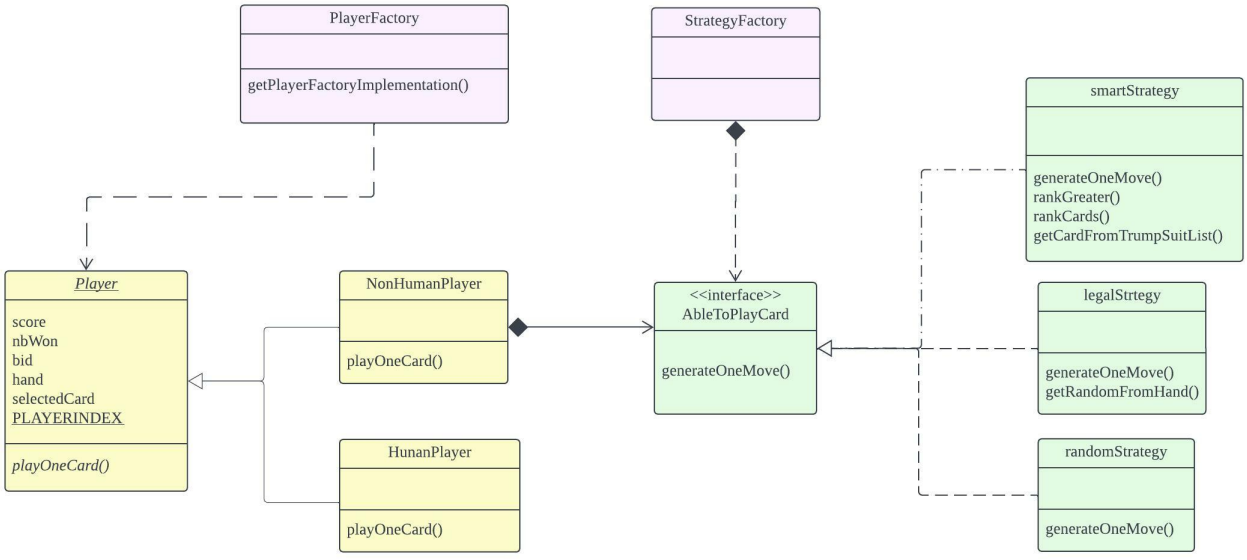

Figure 1: Task1: Skeleton of the design class diagram for Player and Strategy Types

2 Task2

2.1 Rationale

The configurablility of the system is another important aspect of improving the software's accessibility, flexibility and better user's experience in the software development. In this project, we tackled this problem by introducing additional module that allows the users to set up a list of predefined parameters to control the game behaviour and configuration of the game upon execution.

The following parameters are:

- **seed**: seed of the randomness generator in aim to support the repeatable runs.
- **nbStartCards** :the number of sub-round for each round.
- **rounds**: the number of rounds before the conclusion of a game.
- **enforceRules**: Indicator of applying checking the violation of game rules.
- **players.{playerIndex}**: player Types include a set *loadPlayerType* $\in$ {**legal, smart, random**}.
- **thinkingTime**: delay time to play a card for each player, control the pace of the game in (ms).
- **madeBidBonus**: Additional Bonus Score

To make it more user-friendly, we do not impose a strict requirement for users to specify these parameters on each run, which means that the program can still execute successfully without setting the predefined parameters. Instead, the set of default parameters will be used in such a case. In addition to that, in the light of aligning with the modern coding standard, we create our design by considering several grasp principles like creator, information expert as well as Pure Fabrication.

2.2 Implementation

The concrete implementation of this module is through defining a class called *PropertiesLoader* that encapsulates all variables and methods required for loading and processing the parameters from the corresponding .properties file under the folder of "/properties", this is an example of applying the GRASP principle of Pure Fabrication as the class PropertiesLoader is an artificial class which does not have any concrete representation in the real-world.

One advantage of using the .properties files to store the program parameters' configuration is that the programming language Java has a in-built class called "Properties" in the standard java.util library, this enables any proper-defined .properties files to be loaded into a "Properties" class which then can be used to retrieve the string value of those parameters by directly calling the method "getProperty(String propertiesName)". But it is noted that the method getProperty only returns string value given from the propertiesName in the properties file and therefore the conversion of types is needed to ensure that the correct type is assigned to the variables. For example, the

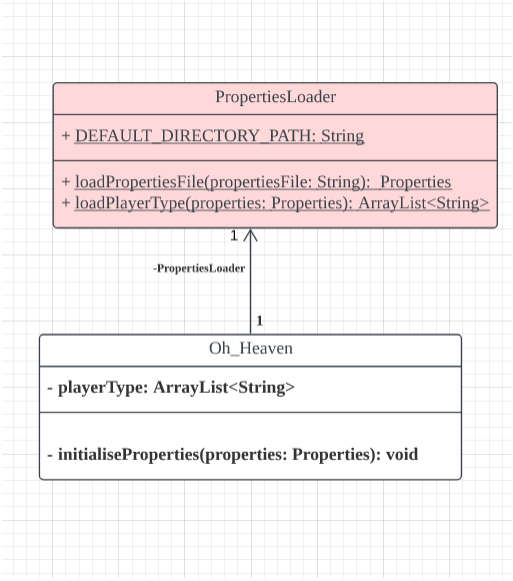

Figure 2: Task2: Interaction bewteen PropertiesLoader and Oh_Heaven

variable **enforceRules** required the Boolean value, whereas the variable **nbStartCards** requires the integer value.

To uphold the GRASP principle of Creator and promote better code maintenance, we consider the *Oh_heaven* class as the best place to initialize the class *PropertiesLoader* as the class *Oh_heaven* is the only class that requires the initialization of the variables for the entire program, so it needs the information to assign the values to variables if the user defines the set of predefined parameters within the .properties file.

However, the method of assigning the values from the *Properties* to variables used in the game called "initialiseProperties" is defined within the class *Oh_heaven* according to GRASP principle of information expert, the trade-off the cohesion and coupling certainly exists in this situation, but taking into account the fact that the class *Oh_heaven* has the most of information about the values of those variables and coupling might significantly increase otherwise, we think it is worth to choose our alternative solution in the context.

3 Task3

3.1 Rationale

To present smarter bid strategies for the NPC players, we need to do a more detailed analysis over the cards of the current NPC player on hand.

First, we would create a new interface called *BiddingStrategy* so that we can access to the *smartBidding* in the NonHumanPlayer by adding BiddingStrategy interface in the NonHumanPlayer class. In the NonHumanPlayer class, writing a initialiseBid method and use the smartBidding in the method. Then in the Oh_Heaven class, bid can be initialised in the playRound method by looping through all players and check whether the player is a NonHumanPlayer type.

Then, we can sort the cards by suit(trump suit ranks higher than other suits) and by the number on the card(2-Ace). If the NPC player has an Ace with the trump suit, definiteWin will plus 1. That is, the NPC will definitely win 1 trick and the numOfBid can start from 1. To make a more specific bidding, the NPC player can add the total number of Ace plus the number of trump suit card and conclude a roughly bid.

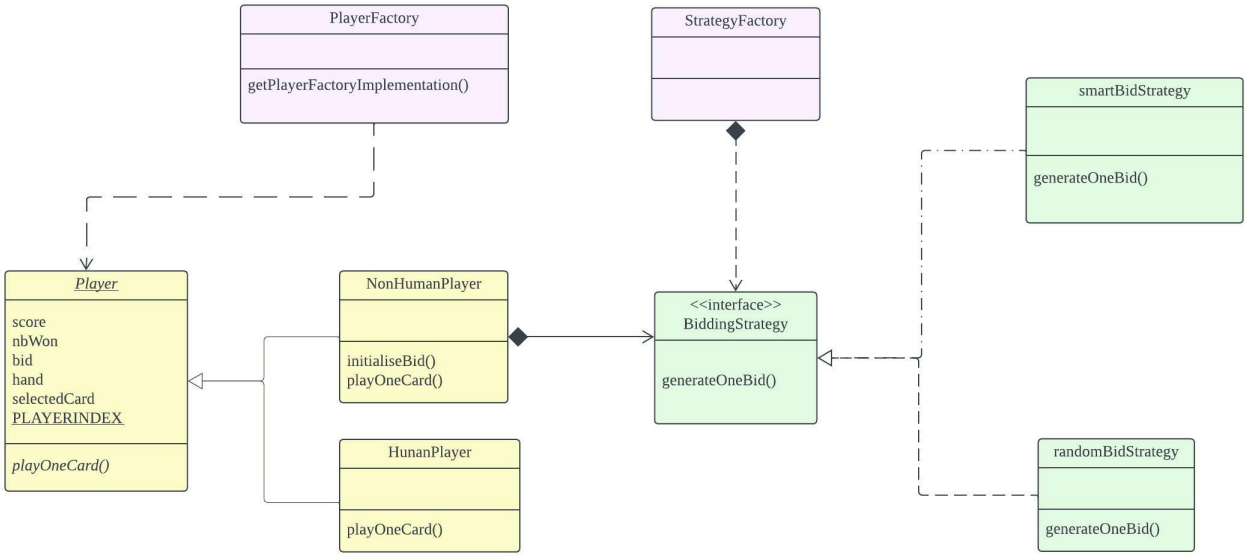

Figure 3: Task3: basic design of a smarter Bidding Strategy

4 Conclusion and Limitations

In this project, we have learnt the importance of implementing the GRASP principles and GoF patterns in various aspects of software design and modelling as they encourage software re-usability and accommodate possible alternative algorithms without breaking the existing functionalities. We faced some challenges when dealing with task 3 since it is hard to establish an algorithm that has obvious improvement over the bidding Strategy, but we managed to give a solution.