Report - Refactoring Whist

Project Team 88
Aman Bhuyan (946264)
Josh Sanon (936965)
Lakshya Mittal (1009364)

Introduction

This project assigned us with the task of refactoring a readymade card game called Whist - using the JGameGrid and JCardGame libraries as the backend - and implementing some other features. The package provided supported the game with 1 interactive (human) player, and 3 NPCs (Non-Playable Characters) that played any random card. The additional features we had to include were:

1. More NPC types - Legal (which plays a random, but legal move) and Smart (which plays a decisive move based on the information from the game).
2. Customisable game parameters - some elements of the game of Whist are alterable but were kept constant in the original program.

In addition to these, we also had to supply our own “properties” files that set up the game in a specific way (with different types of players, number of starting cards, etc.).

Implementations & Justifications

On the first look through the provided package, we observed that every bit of code required for the game to run was handled by a singular class. This is a very bad practice in general - as that shows very low cohesion between the elements of the class. Thus, our first task was to separate out the sections of code into different classes.

Separation of tasks

The libraries that were provided with the game were unknown to us and are a subsystem that needed to be used by the game. We felt that this was a definitive case of requiring a Facade design pattern implementation. As a result, everything to do with the game interacting with the library now passes through a Facade called WhistUI. This Facade is also a Singleton, as we do not require multiple instances of this class, and there should be only one interface via which the game communicates with the JCardGame and JGameGrid libraries. This, in turn, implemented the Controller pattern for the main Whist class, as it controls all the aspects of the game, without any other code required to deal with the subsystem.

In order to obtain further abstraction in the information of the game itself, we have made a separate class to deal with the scores of the players. This is so that the game only knows of a score concept, but it should not have to deal with how the scoring really works.

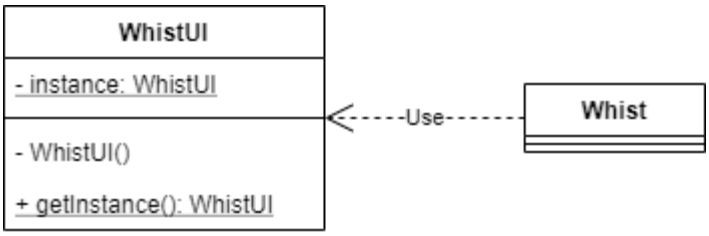

Figure 1: UML showing the Singleton Facade implementation in our program.

There are no real alternatives to this, as simply keeping all the code in one class can be - and usually is - quite messy. Thus, with no other choice, this was clearly the best option. We did leave the ability of other classes (Player, Strategy, etc.) to have instances of the types declared by the subsystem. This was done so as to not have unnecessary coupling between these other classes and the Facade. The main purpose of the Facade was to remove the code dealing with the subsystem from the main game class.

Type enumerations

The enumerations (enums) included in the original program were Suit and Rank (information pertaining to cards). However, as they were inside the main Whist class, no other class that we implemented could have direct access to this information. Thus, it was necessary for us to make these enums universal throughout the package, and so we brought them out into their own enums in separate files, instead of having to redeclare within each class or being dependent on another class for the information.

Player types

Our next step was to introduce the notion of different kinds of players. The ones already included were interactive players and NPCs that would always make a random (potentially illegal) move. However, as with the rest of the original package, this was implemented within the same class as the game.

So firstly, the code for the existing types of players was moved into separate classes. We realised that all players have similar behaviour, but work differently as such - which led to the formation of the abstract Player class. This class has general information for every type of player such as cards in their hand, and their player number. The classes NPC and InteractivePlayer inherit and make this class concrete with their own implementation of how they pick cards to be played. This allowed the game to handle different kinds of players using Polymorphism, with no extra code to deal with each type and covering the subclasses with a layer of abstraction.

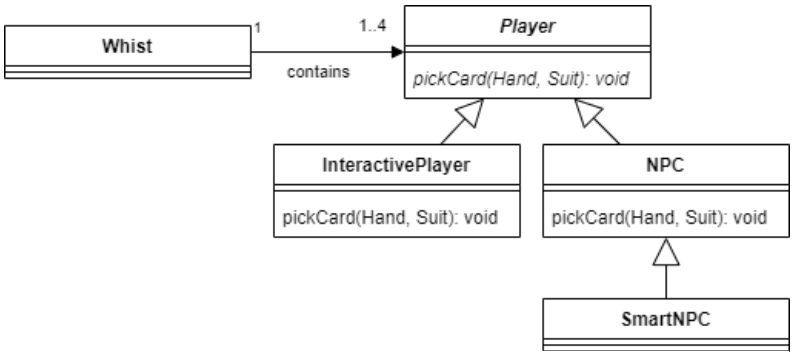

Figure 2: UML showing the abstract player class using Polymorphism to "hide" its subclasses to the main game class.

Interactive Player

The information regarding the input for the human players is now handled by WhistUI, which ensures that the InteractivePlayer class itself isn't dealing with the creation of how it deals with input using CardListener (which is a part of UI), but only with how it enables that. We decided against putting that information in the InteractivePlayer class itself, as that would introduce avoidable extra coupling between the class and the subsystem. WhistUI already held most of the information regarding the input listeners, so based on Expert pattern, it only made sense to transfer that responsibility to the WhistUI class.

NPC

The specification required us to have different kinds of NPCs, so first, we needed a class that would cover all NPCs in general. The difference between these and InteractivePlayers is that they have no "listener" attached to them, and the cards played by them are automatic and dependent on the type of NPC they are.

"Smart" NPC

We also decided to have a "smart" NPC class (inherited from NPC) that stores its own information about the game being played. This made sense to us, instead of having every single NPC store information. This was also in an effort to make the program extendable, with usable information readily available to make an "extremely smart" NPC.

NPC strategies

Coming back to the point of having different types of NPCs, we thought it would be best to implement different strategies that are used by an NPC. We did consider making concrete classes like RandomNPC, LegalNPC, etc., that extend from NPC, but that idea was rejected. This was because it would mean that if we wanted to have a different kind of NPC, we would have to extend the NPC class again and have all the necessary implementations yet again which is just not good coding practice nor future-proof.

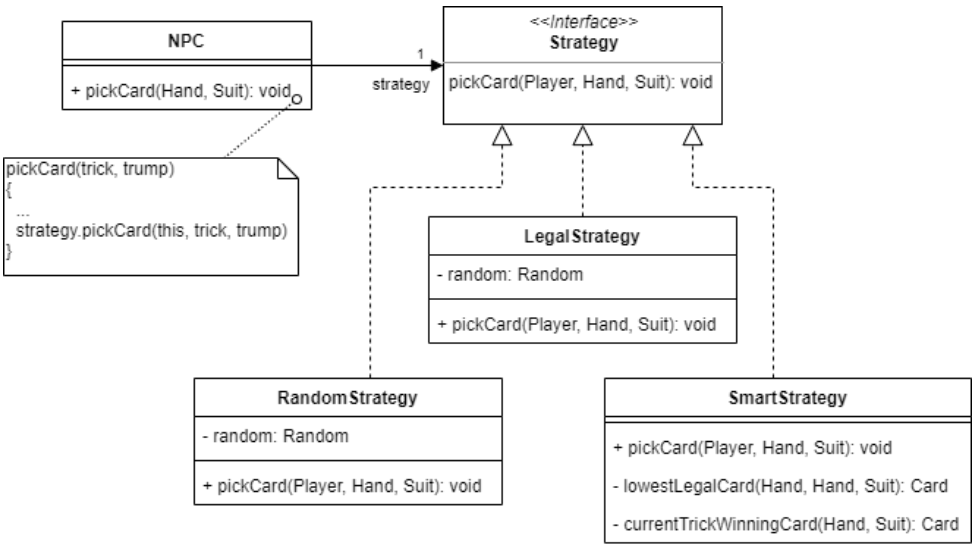

Figure 3: UML showing the application of the Strategy pattern in our implementation.

Thus, we decided to implement a general interface called Strategy that each NPC used to make its decision as to what card to play. Like its namesake, it also implements the Strategy pattern. A strategy's only task is to pick a card to play for the NPC, and as the different strategies are under the umbrella of a single interface, it makes this task much easier regardless of the underlying type of strategy. This means that if there was another type of NPC to be implemented (or rather another type of card-picking strategy), then no new actual types of NPCs need to be created - just a different kind of strategy would have to implement the interface.

Smart Strategy

It must be mentioned that our implemented strategy for a "smart" NPC does not use the history of the game that it contains to make a substantially smarter move. However, the information is still retained by a smart NPC, just to show that it is usable. As such, our smart strategy can be used by the NPC class as well, which goes to show that our implementation is quite flexible with the strategies and a successful implementation of the pattern. Of course, if an "extremely smart" strategy was made which used the game history, then only the smart NPC class could use that strategy and not the basic NPC - another reason why we added the SmartNPC class. Another point to note is that the game treats all NPCs as "observers" in addition to players. So, at the end of each trick, the game notifies all NPCs about the trick that was played. Normal NPCs store no information, but SmartNPC accumulates this on every notification, and are thus implementing the Observer pattern.

Handling player creation

With the presence of many types of players, NPCs and their respective strategies, it is clear that the main game should not be bothered with the actual creation of the player instances, but only with how many of each type should be created. This is why we have the PlayerFactory class. The main game only has to tell the Factory what type of player it needs, and the Factory provides the exact type of instance of Player for the game to store in its internal list. In the case of NPCs, it also initiates them with the corresponding strategy as requested by the game.

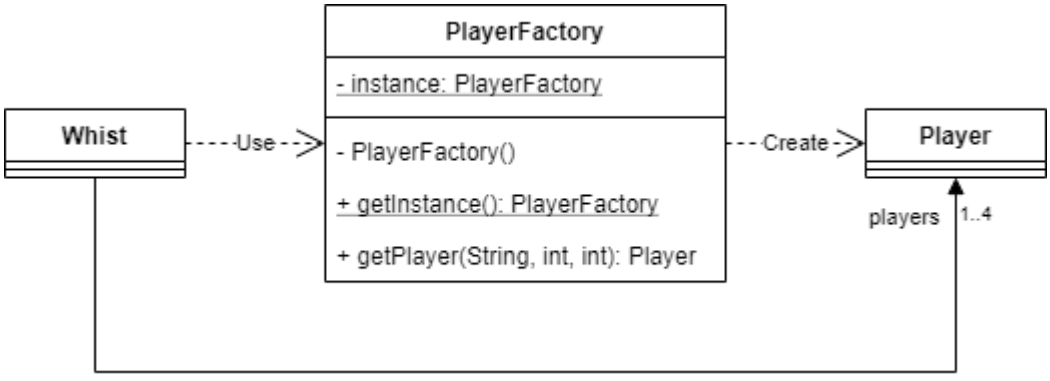

Figure 4: UML showing the Singleton Factory for the creation of the different types of players.

This also implies that if more strategies were to be implemented, this Factory would need to be correspondingly updated so as to be able to create that kind of player.

We also decided to make this Factory a Singleton, as there is no need for multiple creators, considering there would be absolutely no difference between each PlayerFactory instance.

“Random”-ness of the game

We have modified the code to now accept a random “seed” that decides the pseudo-randomness of the program. This ensures that the program will always have the same outcome, given the same seed. If no seed is provided, then every run of the game will be completely random.

Dealing of cards

In the original package, the cards were dealt to players using a function from the subsystem library. This function however, could not be customised to provide a specific outcome each time. We also kept in mind that the “deck” available to the game needed to remain constant, and simply shuffling it was not ideal. Thus, we wrote a custom function for dealing out the cards to the players, and this is dependent on the seed that is provided (otherwise completely random) - thus maintaining original behaviour.

Modifiable game properties

The configurable parameters that we have chosen to add to this game are:

- Seed representing the randomness of the game
- Number of each different type of player (Interactive, Random NPC, Legal NPC and Smart NPC)
- Starting cards
- Winning score
- Enforcing of rules (no illegal plays allowed)

We decided that these parameters should be made configurable for a simulation of the game. The 3 “properties” files that we have provided configure these parameters to simulate the game as tasked.

Conclusion

We believe that our implementation is able to perform the necessary tasks as laid out in the specification. Original behaviour has been maintained, while introducing other features and behaviours. We have refactored the original package in a way that it is now easier to understand (as seen from the final diagram), and also much easier to extend further, if necessary.

With the use of patterns (like Factory, Facade, etc.) we have achieved a significantly refined design of the overall program. These allow us to have low coupling between the classes, while maintaining high cohesion within each class. Lastly, we have included the final static design diagram of our implementation to give an overview of the full implemented system.

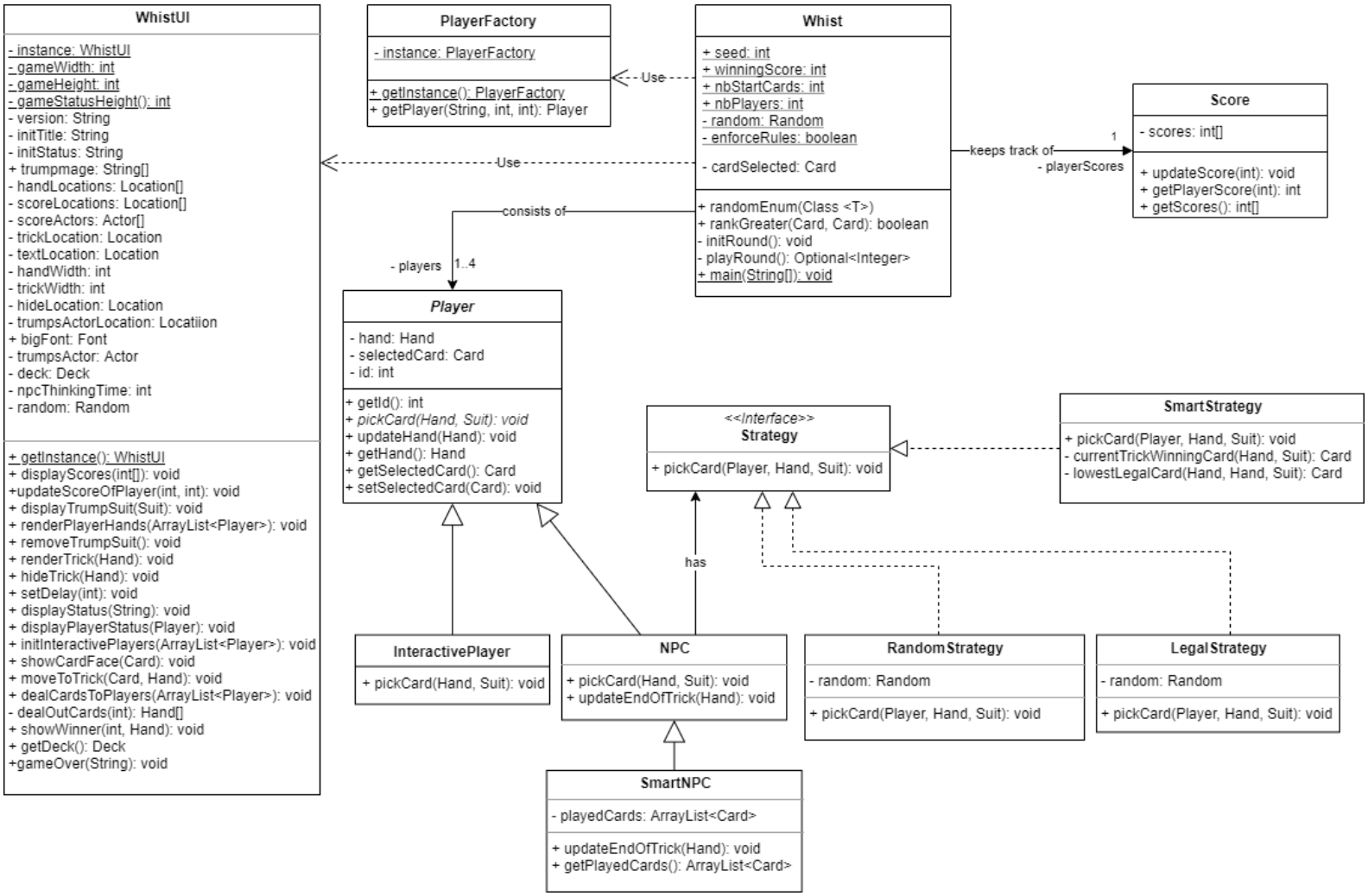

Figure 5: Final static design diagram of our implementation, provided only for overview purposes.