Project 1: Whist – Design Analysis Report

SWEN30006 Software Modelling and Design

Team 28:

Jeff Yang (880512), Yutong Wu (880445), Han Liu (503931)

Introduction

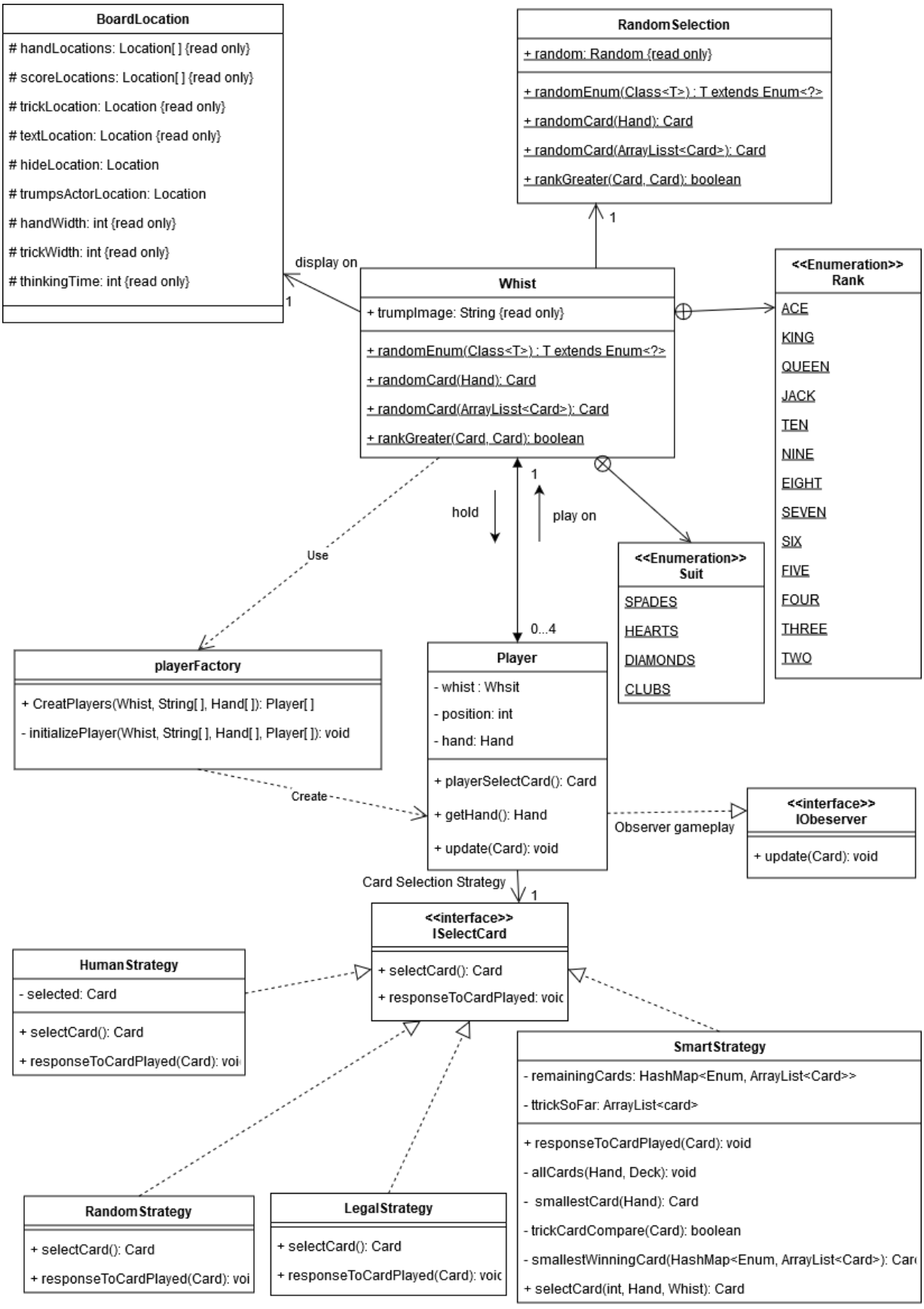
In this project, a simple version of Whist game implementation is given, which is later improved by adding an NPC system and optimizing configurability. There are four players sitting on four different position that playing the Whist game. According to the project requirement, three types of NPC player are included in the system which could be allocated freely along with the original interactive player. They are Random NPC who plays the game randomly, Legal NPC who follows the rules but plays randomly and Smart NPC who records relevant information and makes logical move based on that information. A property file format also been adapted to configure the system such as the winning score and the allocation of players.

This report briefly explained how we come up with the design and implement the system. Including 2 parts, they are as following:

1. ***Solution Design***: brief explanation of our solution with the help of diagram.
2. ***Patterns & Alternation Solutions***: illustrate the application of GoF patterns in our design & the alternative solutions we have considered and why we did not adopt them.
   - ***Alternative solutions***
   - ***Final solutions***
   - ***Patterns & principles analysis***
     - ***Strategy Pattern***
     - ***Observer Pattern***
     - ***Singleton & Factory Pattern***

Solution Design

The design of our implementation is as shown on the static design diagram.


BoardLocation
handLocations: Location[ ] {read only}
scoreLocations: Location[ ] {read only}
trickLocation: Location {read only}
textLocation: Location {read only}
hideLocation: Location
trumpsActorLocation: Location
handWidth: int {read only}
trickWidth: int {read only}
thinkingTime: int {read only}
RandomSelection
+ random: Random {read only}
+ randomEnum(Class<T>) : T extends Enum<?>
+ randomCard(Hand): Card
+ randomCard(ArrayLisst<Card>): Card
+ rankGreater(Card, Card): boolean
display on
1
1
Whist
+ trumpImage: String {read only}
+ randomEnum(Class<T>) : T extends Enum<?>
+ randomCard(Hand): Card
+ randomCard(ArrayLisst<Card>): Card
+ rankGreater(Card, Card): boolean
<<Enumeration>>
Rank
ACE
KING
QUEEN
JACK
TEN
NINE
EIGHT
SEVEN
SIX
FIVE
FOUR
THREE
TWO
hold
play on
1
Use
<<Enumeration>>
Suit
SPADES
HEARTS
DIAMONDS
CLUBS
0...4
playerFactory
+ CreatPlayers(Whist, String[ ], Hand[ ]): Player[ ]
- initializePlayer(Whist, String[ ], Hand[ ], Player[ ]): void
Player
- whist : Whsit
- position: int
- hand: Hand
+ playerSelectCard(): Card
+ getHand(): Hand
+ update(Card): void
Create
Observer gameplay
<<interface>>
IObeserver
+ update(Card): void
Card Selection Strategy
1
<<interface>>
ISelectCard
+ selectCard(): Card
+ responseToCardPlayed: voic
HumanStrategy
- selected: Card
+ selectCard(): Card
+ responseToCardPlayed(Card): voi
SmartStrategy
- remainingCards: HashMap<Enum, ArrayList<Card>>
- ttrickSoFar: ArrayList<card>
+ responseToCardPlayed(Card): void
- allCards(Hand, Deck): void
- smallestCard(Hand): Card
- trickCardCompare(Card): boolean
- smallestWinningCard(HashMap<Enum, ArrayList<Card>): Car
+ selectCard(int, Hand, Whist): Card
RandomStrategy
+ selectCard(): Card
+ responseToCardPlayed(Card): voi
LegalStrategy
+ selectCard(): Card
+ responseToCardPlayed(Card): voic

Several factors are abstracted from *Whist* class compared to the original game to fulfill a better and more reasonable design. Position element of the game is separated to be an independent *BoardLocation* class accessed by the game to adjust and confirm windows location and size more easily.

The main game has four different types of player as required, *RamdomNPC*, *LegalNPC*, *SmartNPC* and *InteractiverPlayer*, which is defined by their card playing strategy. This is achieved through *Player* class keeping their own strategy which implements *ISelectCard* interface for possible further extension. Different strategies have their expected way to decide which card to play (mouse interaction, randomly, legally, or smartly) as stated in the project requirement. Also, there is an *IObserver* interface to enable all players to observe public information of the game to help their decision of which card to play. The actual behavior differs accordingly to the strategy.

HumanStrategy enables mouse interaction and play chosen card as human player double-clicks to select card. *RandomStrategy* ignores rules and play a card from hand randomly, no matter whether it violates the rules or not. *LegalStrategy* follows basic rules and play a random card that does not break the rules. *RandomStrategy* and *LegalStrategy* use *RandomSelection* class to pick a random class to from an array list of cards representing cards in hand. The *RandomSelection* class is built to convent further adjustment and application of random card playing. Lastly, *SmartStrategy* keep record of cards played and card pool so that it can calculate which cards are in other players' hand. In this way, *SmartStrategy* is able to find a card with very high win rate to play or play the less important (normally lower-rank non-trump card) card if there is no such a card. *Player*, with all types, is created by *PlayerFactory* instead of created directly in the main game. *PlayerFactory* creates number of players and assign their playing strategy as required by the properties file, which is read by the main game. By doing this, players' behaviors can be defined in separated class and the same type of player do not need to be defined repeatedly to avoid coupling.

Overall, many game factors are separated out as independent class to achieve high cohesion and low coupling, and creator (*PlayerFactory*) is made to have a better design for further adjustment and development.

Patterns & Alternation Solutions

We have not made many changes in terms of the overall structure of the program. The workflow stays almost the same, despite we did consider make big changes in term of structure.

Alternative solution

once we considered divide the game into a couple of different components as following:

- **Component 1** - *whistGameFacade*. This object receive configuration and commands from property file and controls the inter-action of other objects.
- **Component 2** - *gameBoard*. This object contains all the information required to

display the game and draw elements during the game, such as hands, trick, trump, transfer of card, and so on.
 Component 3 - *helper classes*. Some of the information in original Whist class can be grouped together and become helper class. Some of this information is share among the whole program, many other classes will require it too.
 Component 4 - *validation*. This object checks if a card played follows the rules, and decide if end the game based on the Boolean object *enforceRules*.

The benefits of doing this way is that it increased cohesive, and makes future expansion or modification easier. However, there are drawbacks as well. This design will suit better for a general card game, but the project specifically requires a whist game program whose rules and display of elements are fixed. There are other reasons we abandoned this design. First, we recognize the Whist game requires information about what rules it follows, requires where and how its elements are displayed. Separating them will only increase coupling, but will not increase cohesive a lot (e.g. bad trade off), which violate the low coupling principle.
As the result, we only decide to group highly cohesive information together, and make them to become helper classes.

Final solution

The final solution present in our submission only makes several changes, we are keen to keep the workflow of running the game unchanged. Only a few changes have been made to meet the requirement that different types of player select cards based on their strategy. The changes are as follow:

1) We identify that there are four players in the game, the player object contains hand (e.g. a series of cards), its location in the game, the game it belongs to, and the strategy they adopted to select card.
2) We implemented the functionality that allows player receive updates from game. It allows players response to the information in their desired way.
3) We Grouped highly cohesive information together, and make them to become helper classes
4) We implemented the functionality that allows the program to be configured from a property file.

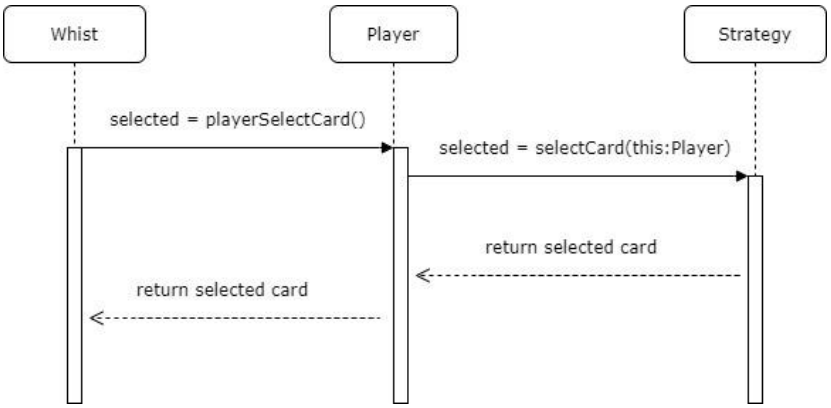
Following analysis is the details of how we implement above functionality and the alternative solution we have considered. Moreover, it analyses how the solution has impact on the program in terms of design principles.

Patterns & principles analysis

There are several problems arise when designing the program. In order to solve these problems, we have adopted difference pattern for each problem.

Strategy pattern

Strategy pattern has been adopted as different player has different algorithm to select cards to play.
Target: players who play the game (e.g. select card) based on their own algorithm /strategy.
Problem: How to design a series of varying but related car selection algorithm, and in the meantime has the ability for further expansion.



Design consideration
In our design, all types of player are the same to the system, they are only different at the way of choosing which card to play.
It obviously violates the high cohesive principle if we implement the algorithm in *Whist* class. the algorithms are not part of the game object. It is applied through players' hand, the players decide which algorithm/strategy to use, not the game.

Alternatively, we consider the design that different types of players (*legalNPC*, *smartNPC* etc.) extend from super class "player", and implement their selection method within their own class.
However, this kind of implementation lacks flexibility, and there is a representation gap between the reality and this implementation. For instance, a player plays legally at the beginning of the game and decides to play smart later. Under this scenario, previously implementation is not so good. The program must delete the *legalPlayer* object and re-create a *smartPlayer* object to represent this player.

Compared to final solution
Our solution to this problem is adopting strategy pattern. In our design, we created an interface *ISelectStrategy* and every strategy will implement this interface. Then each player contains an *ISelectStrategy* object. When a player needs to decide which card to play, it passes essential information (or itself) to the strategy, and the strategy returns the card to play.
It is flexible and easy to expand. If more strategy required, just create a class of that strategy and implement *ISelectStrategy* interface. If at some point, a player deicide to adopt different strategy, it can just update the strategy by creating a new strategy object, and assign to the *IselectStrategy* object (this functionality is not implemented, as project does not require it).

Patterns & principles analysis

In this way we increase the cohesion of the design and achieved information expert. all the players' information is in the *player* class, all the strategy's information is in *strategy* class.

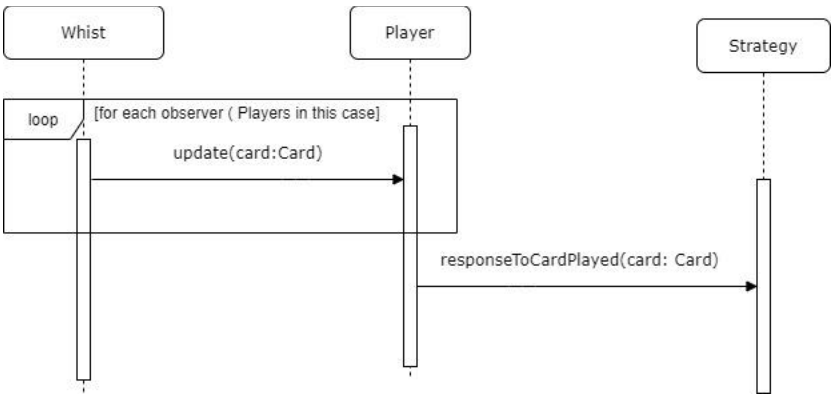
Moreover, we achieved polymorphism & Protected Variations through *ISelectStrategy* interface that every player can adopt a strategy that may vary to each other. Unfortunately, coupling is unavoidable somehow, we just have to keep it as low as possible.

Observer Pattern

Observer pattern has been adopted as players need to have access to the game, they are required to know what card has been played by other players.

Target: players gets update on the cards has been played (e.g. cards in "trick").

Problem: How to design program so that the game object notifies the player when cards has been played. Furthermore, the game class maintain low coupling to the player class.



Design consideration

First, the design should allow players have access to game's information.

Then, players should response to the information they acquired in some way.

More importantly, allowing for further expansion. What if there were other objects that are interested in updating events of the game. For instance, a player's friends watch him playing the game, and they want to be notified when a card has been played. These friends could be watching multiple games at the same time. It is not necessary and is a bad design for every game to contains these objects.

Alternative solution we have consider is that not using observer pattern. The "Whist" object contains "Player" objects itself, it can notify player directly, then player response to the change. However, in this way, only players will receive the updated information. The design is fixed and inflexible, no other object can receive updates and is hard to expand in future.

Compared to final solution
Our solution to this problem is using observer pattern. We created an *IObserver* interface. Every object who want to observe the game required to implement this interface. Then in the *Whist* class, contains a list of *IObserver* objects. Whenever there is a change in the game, the game invokes update method, and notify all the observers.
Moreover, in this way the game can comfortably manage who receive the updates. It can add and remove observers easily as well.

Patterns & principles analysis
By applying observer pattern, polymorphism and low coupling have been achieved. In regards to polymorphism, different type of object can observe the game through a single interface *IObserver*. Regarding low coupling, the game is not directly connected to all the observers, they connected through the interface. In this way, the game object can easily manage those observers, e.g. add, remove, and decide who gets updated.

Singleton & Factory Pattern

Singleton factory pattern is adopted to create players & strategies with input configuration.
Target: *factory pattern* creates players and the strategy along with each player without specify the type of strategy, hide complex creation process behind the class. Singleton pattern makes sure there is at most one and only one factory object available.
Problem: all the players are same in our design. However, the strategy along with them at the beginning is different. The creation process involves a number of if statement, and string manipulation.

Design consideration
 1) It is better to hide the creation process behind a factory class.
 2) There should be one only one on factory instance.
 3) The class must be able to create a player along with the strategy without specify the class of strategy.

Compared to final solution
We adopted singleton factory pattern, which build a *playerFactory* class. That allows us to input the configuration in property file into the factory, and output player along with each player's strategy

Patterns & principles analysis
Creator principle has involved in factory pattern. The *Whist* game contains players, and uses player object a lot, so the game object is where we should create players along with their strategy.

In conclusion, the implementation is completed successfully. A several tests with various parameter and seeds has been tried and our system works smoothly as expected. The configurability is ensured through the properties files and the expendability is built within the design. Several alternative solutions were considered but they all have some flaws compared to our final solution.