Project 2: Whist Card Game – Design Analysis

SWEN30006 Software Modelling and Design

Team 118
Aneesh Chattaraj-826860
Ian Wen Jing Teh- 950479

1 Introduction

The aims of this project is to modify the design and improve the configuration of a company's new card game Whist. The current version is not up to the mark and does not provide extendibility. It only supports a human interactive player and basic NPCs who play any random card from their hand, even breaking the rules. This is a serious issue and we have been asked to provide extendibility for future additions to the game since the current version of the game handles all game activities in one class, this is a serious flaw and not a good design to go forward with because of its very low cohesion. We have also been asked to add two new types of NPCs, Legal player and Smart Player. Their description is as follows:

- *Legal: plays a random selection from all cards that can legally be played.*
- *Smart: a player that records all relevant information and makes a reasonable, legal choice based on that information. A Smart player must produce smarter play than a legal player and have the necessary information available in a suitable form so that an extremely good NPC player could be developed based on the Smart player. A Smart player should assume that other players are playing legally.*

In addition to these requirements, we have also been asked to make seed maps for testing the correctness of our code by re-running the same dealings of cards, so that it is easy to follow up.

2 Design and Modifications

2.1 Summary

Our system incorporates the following patterns and GRASP techniques:

- *Factory Pattern*
- *Strategy Pattern*
- *Singleton Pattern*
- *Higher Cohesion*
- *Lower Coupling*
- *Polymorphism*
- *Controller*

2.2 Patterns and Principles

Initially, before we started to modify the code, we analysed the given code and realised that it would be a good design strategy to separate the NPC and interactive player. We decided to go with the factory pattern by creating abstract classes and then via polymorphism we can add the additional required players (Smart and Legal NPCs) along with the basic NPCs and human/interactive player. The factory pattern would provide higher cohesion and lower coupling hence the reason why we decided to go forward with this approach. This would help us provide a better-designed code which would be easier to maintain and configure in future changes. Since factory pattern lets class defer instantiation to subclasses it provides extendibility in future versions to add more versions of NPCs (one of the asked requirements).
Then on it was decided to implement the strategy pattern to NPC class rather than creating a different kind of player, it worked better to treat them as a different behaviour of a single NPC class. This was done because the same strategy can be used across different players in future versions and thereby preserving the extendibility for future changes. This does come with a side effect of higher coupling.
Our system also works along the lines of a ***singleton pattern*** *because of this it lowers coupling and complexity of the code. Its main intent is to provide a global point of access to the GUI class.*

2.3 Legal and Smart strategy

2.3.1 Legal NPC

Additional non-playable characters or NPCs needed to be added. In the case of a legal player, it was fairly simple we just created a strategy where rather than playing a random card like a basic NPC in any case, it played a card depending on if it has to lead or follow. If the legal player had to lead it plays a random card from its hand(a legal move since the lead player can play any card and is not restricted by the trump card). If the legal player had to follow it would play a card available in his hand to follow the lead suit, in case no cards belonging to the lead suit are available in the hand it is allowed to play any random card. This strategy helps it maintain all legal plays.

2.3.2 Smart NPC

Smart NPCs need to be able to play smarter moves than a legal NPC. This strategy is divided into two cases similar to Legal NPC, lead and follow. If the NPC has to lead, we decided to play the highest ranking card available in the NPCs' hand. In doing this we increased its chances of winning the round as a lead.
If the player has to follow it would play a card depending on the cards already played in that round. If there is a higher ranking card on the table than its highest ranking card of the lead suit in his hand, it would play the lowest ranking card of that suit or else the highest card to increase its chances of winning.
If the NPC does not have a card of the same suit as the same suit, it would try to do the same thing with cards available of the trump suit in its hand to increase its chances of winning the rounds. If it has neither a lead nor a trump suit card it would just play a random card from its hand. This strategy does not break any rules or give exceptions and hence seems like the basis of a good NPC.

2.3.3 Alternative Smart NPC

In an alternative strategy for NPC, we had thought to record data of all the cards played in the previous rounds and then strategize a technique for the smart NPC to play a better card. We decided not to go ahead with the plan because in the project requirements each player needs to record any necessary information on its own and cannot access a common pool of data. Storing all cards played information for different users and not being able to use information expert techniques seemed impractical, that is why we could not go forward with this plan.

3 Code

***Whist class** acts as the **controller** since it receives information from the property files, creates the deck of cards, deals out the cards from the deck into each players hands, checks who wins the round and games and as a whole controls the workflow of the entire system.*

*We also separated the **Graphic User Interface** from Whist.java since this felt like an unnecessary task to be handled by the Whist class. In creating the **GUI class**, we increased **cohesion** of the classes again and also with this any additional graphic modifications in future versions of the game would be easier.*

***PlayerFactory.java** uses **factory patterns**, NPC and human player class implements it.*

*All the strategy classes BasicCardStrategy, LegalCardStrategy and SmartCardStrategy act like different behaviours of the NPC class. This **strategy pattern** is achieved via **polymorphism**.*

4 DOMAIN MODELS

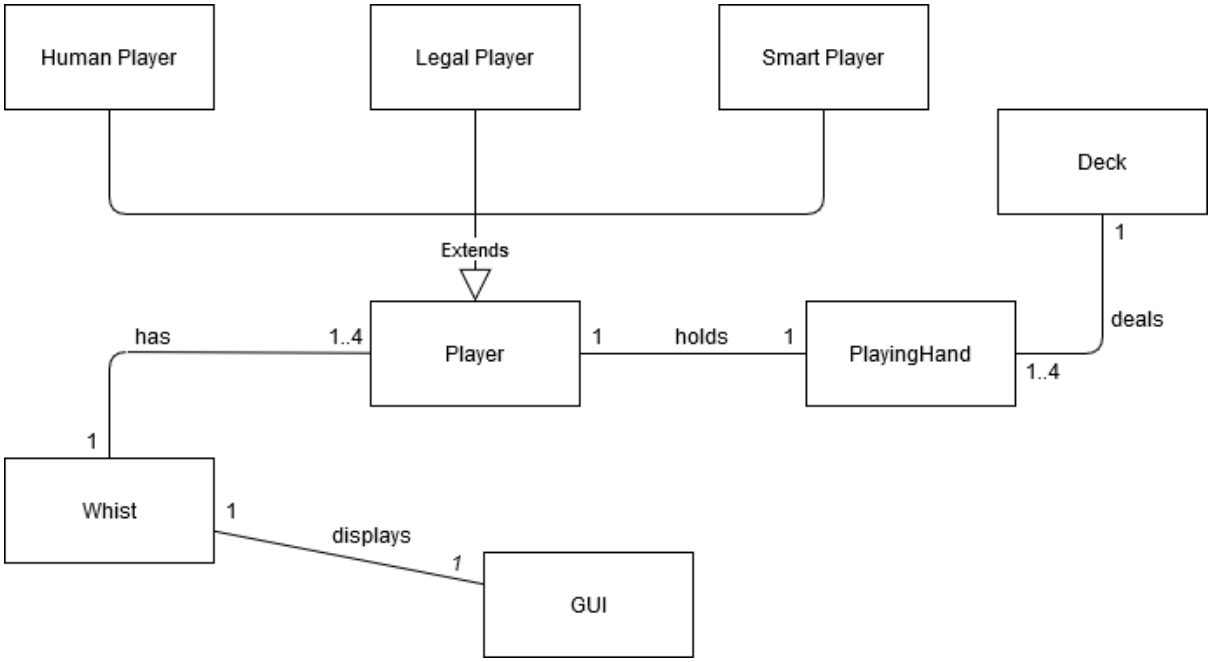

Figure 1 PREVIOUS DESIGN MODEL

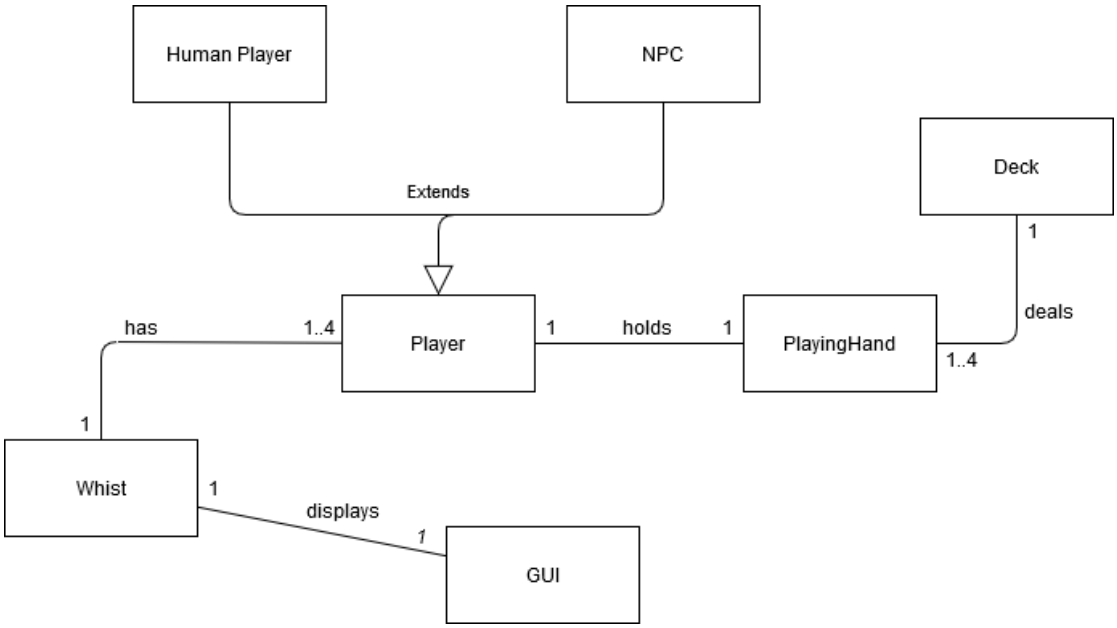

Figure 2 IMPROVED DESIGN MODEL

Figure 1 was our previous design model and we can see it just used factory pattern and in figure 2 we can see the difference in NPC class where applied displayed our strategy pattern, in essence, they have very similar functionality here but we realised for this project we should display more skills based on our knowledge.

FIGURE 3 describes our entire code and from this, we can see how the entire code was made more cohesive from the original provided code, separate classes work on separate issues of the system. This also adds the NPC's ICardStratergy interface which is not displayed in the domain model (figure 2), this clearly gives us a visual representation of how NPC will work.

To further clarify NPC and its strategy function a system design model (figure 4) has been included which provides us with a more detailed view how the system, more focused on NPC, works along with what functions called and what they return.

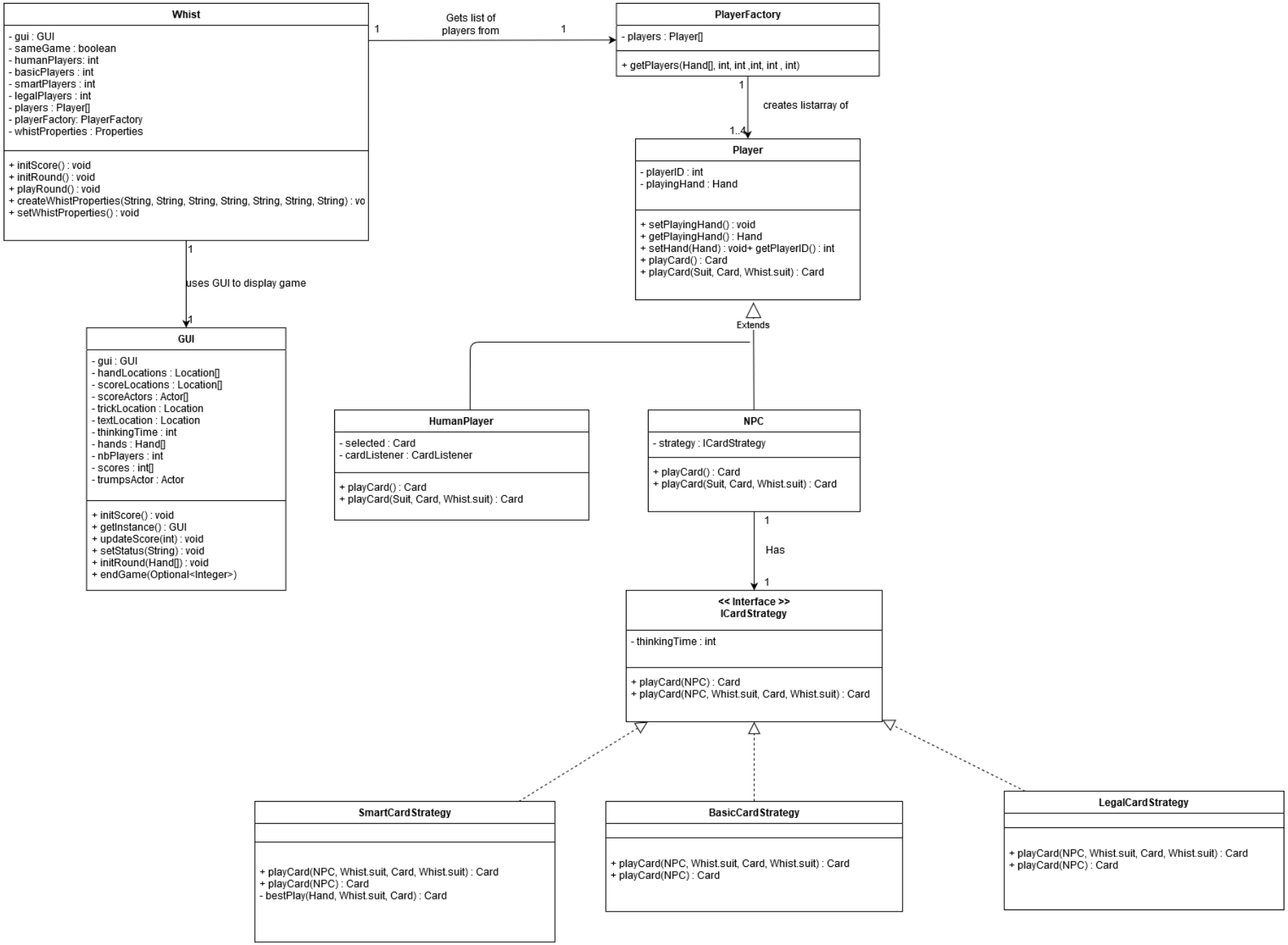

Figure 3 CLASS MODEL

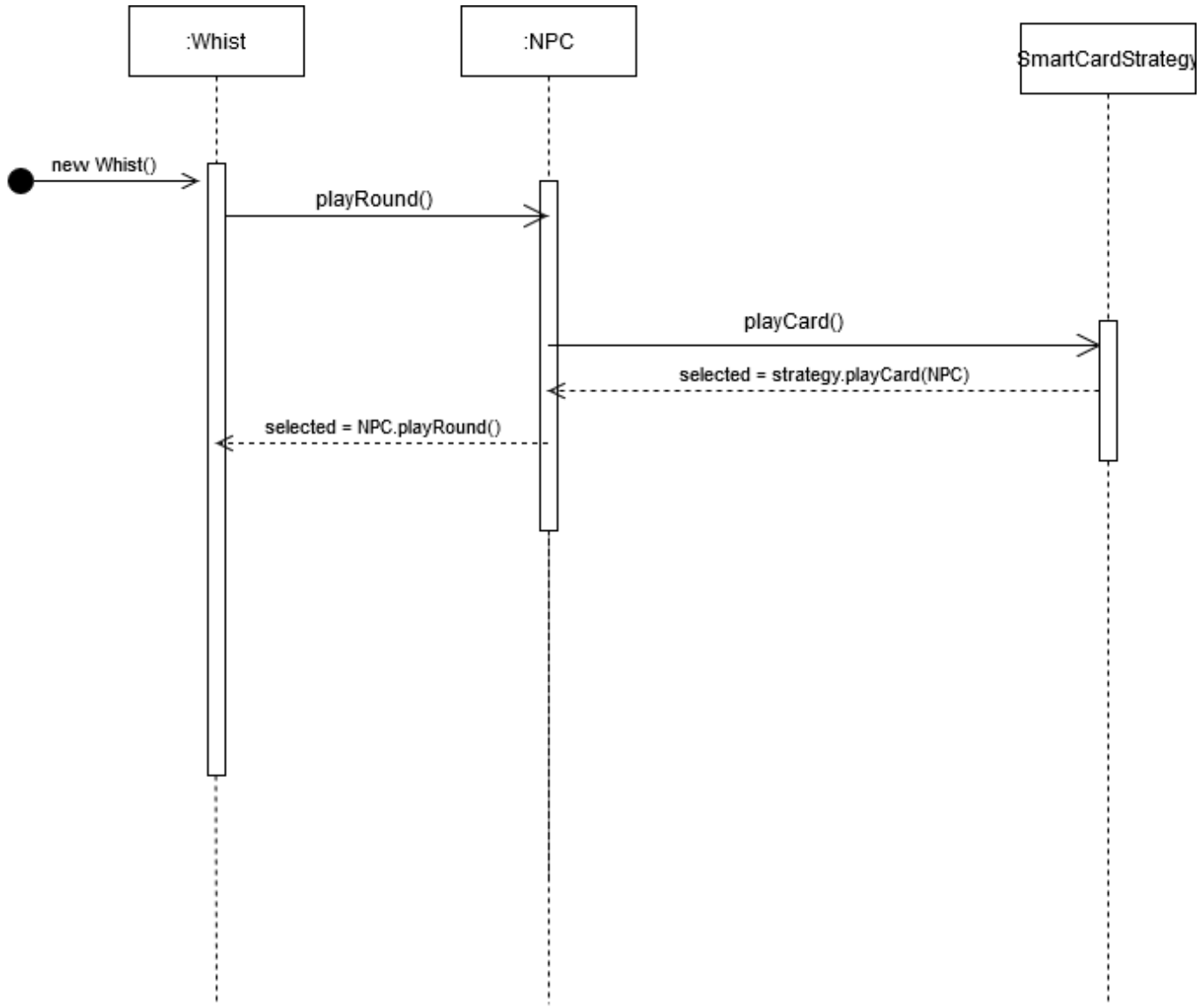

Figure 4 SYSTEM DESIGN MODEL