

**SWEN30006**

# **Project 2: Report**

---

Team Members (Team 1 Tuesday 5:15pm)

Andrew Shen 982054

Khant Thurein Hain 1028138

Kaif Ahsan 1068214

## Introduction

For this project, we undertook a responsibility based approach to design where we divided the core functionalities of the game into the responsibilities of objects. Accordingly, based on the nature of the responsibility we attempted to implement an architecture, which not only makes our current design modular and clean but also easily extendable and highly scalable in the future.

### The responsibility of coordinating the GUI

The game needed to be able to display game information in a GUI. The majority of the functionality was achieved by the external *JCard Game* library. However, the application still needed a coordinator to position the different UI components and display the correct image assets. Because the UI component is not relevant to the domain, we did not have a domain element that directly corresponded with the UI. We had two options:

1. Assign the responsibility to the main *WhistGame* element
2. Create a class with the sole responsibility of coordinating the UI

The class that coordinates the UI will have to be used by other elements, like the Player class that needs to change the status text. If we kept assigned all the responsibilities that other classes needed to access to the main Whist element, we could have relatively low coupling. However, making Whist responsible for both coordinating the UI and coordinating the game logic leads to low cohesion.

Because the set of responsibilities for managing the UI is highly cohesive and all the information needed can be contained in one class, we applied the Pure Fabrication pattern and created a UI design element. We went with the second option because we decided that keeping game logic and UI logic separate was important to make the system easier to extend, understand and reuse. It also decreases coupling to the Whist element.

We also noted that the program will only have one GUI, only one instance of the UI element is needed. So we applied the *Singleton* pattern to the UI element.

## The responsibility of reading property files and storing game settings

We considered two options for storing the game settings

1. To keep it in the element that reads the property file
2. To store the game setting in each component that uses it

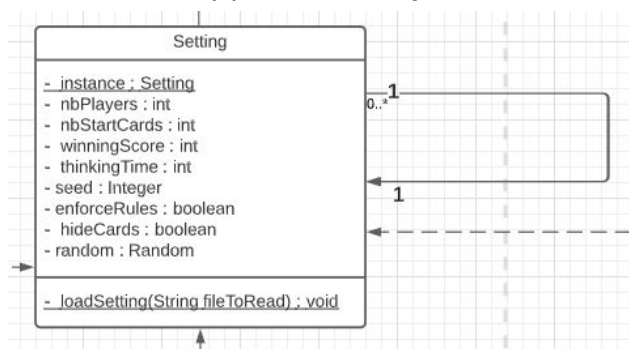
Using the information expert pattern, the element responsible for reading properties files will have access to all the game settings. It makes sense to store them in one common component, doing so increases cohesion and removes the possibility of having inconsistent copies of the same game value settings. We also felt that as the game is extended, more elements may need access to new game settings. For example, we found that in the process of improving our *Smart Selection Strategy* class we needed to access more game settings to better model the game.

We considered two options for assigning the responsibility of reading the property files

1. Assign the responsibility to the main *WhistGame* element
2. Create a class for reading and storing game settings

The setting variables are part of the Game element in our domain diagram, so the first option has a smaller representational gap to the domain. However, the responsibilities for reading property files and handling high-level game logic is not cohesive.

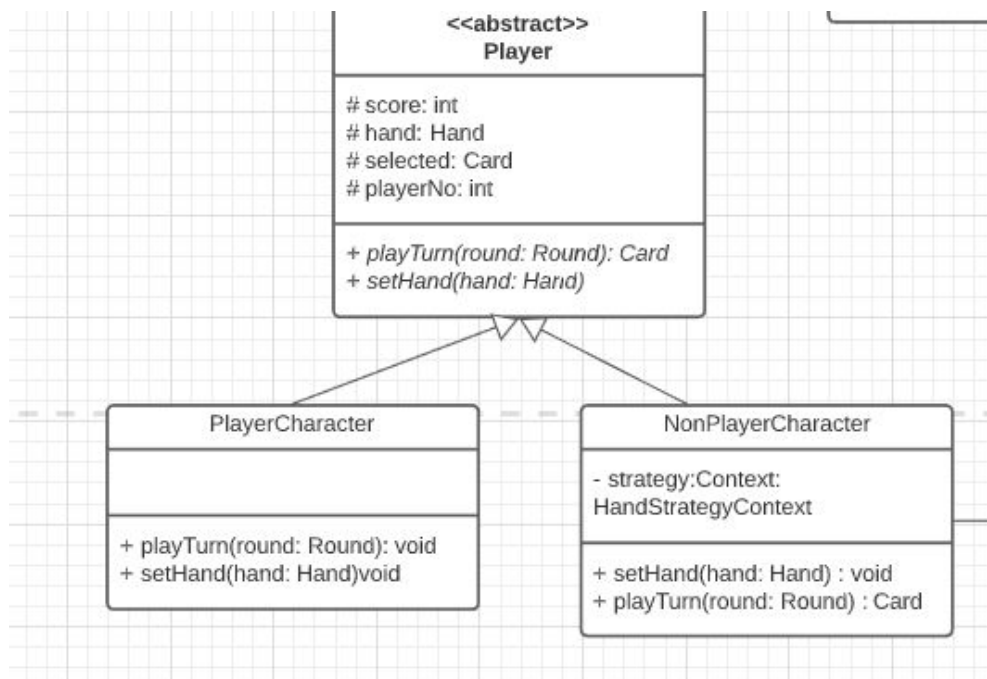
The second option is to apply the Pure Fabrication pattern and create a new Settings element. Both options have the same amount of coupling, so we went with the second option. Like the UI element, we also applied the Singleton pattern because the application only needs one instance of the Settings.



## The role of Player

The *Player* is an abstract class that defines the common behaviours between *PlayerCharacter* and *NonPlayerCharacter*. Ultimately, the role of any *Player* object is to select a card during *playTurn()* and thus, we declare an abstract method *playTurn()* in *Player*, which will be defined by its child classes.

This approach makes use of polymorphism, where the use of if-else statements was avoided to differentiate between *PlayerCharacter* and *NonPlayerCharacter*. The details of how every *Player* chooses a card is abstracted from the *Round* class, thus increasing cohesion and making the game highly scalable in scenarios where there are multiple *PlayerCharacters*.



## The responsibility of handling NonPlayerCharacter (NPC) behaviour

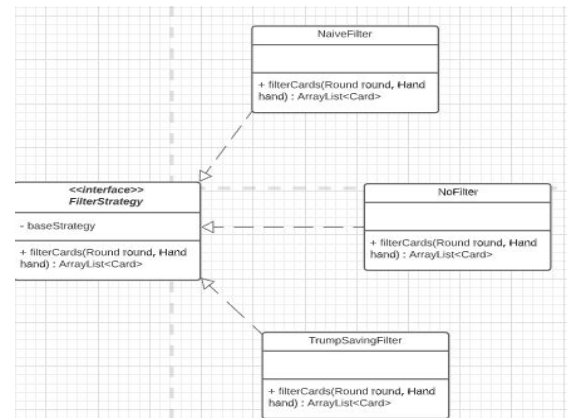
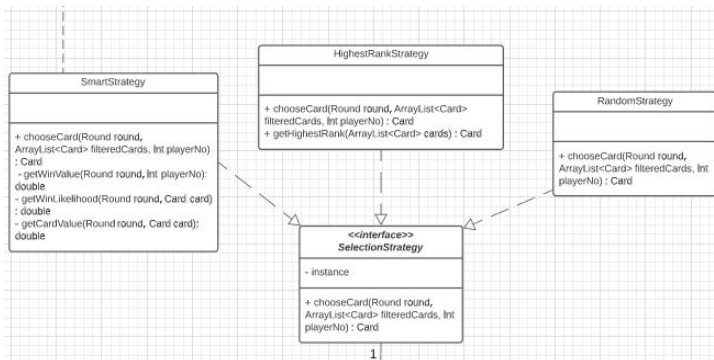
NPCs are a core part of the game and they mimic the real player's through an optional filtering stage and a mandatory selection step. As per system requirements, the NPC is required to implement a number of different algorithms for filtering and selection which are closely related to each other. Furthermore,

these algorithms are also required to be easily configurable and swappable. With such premises, we decided to implement a *Strategy* pattern as it enables us to achieve the conditions mentioned above.

We explored two possibilities of implementing the Strategy for NPC. They are:


1. A unified interface which encompasses both the filtering and selection step.
2. Separate interfaces for the card filtering and selection.

After much consideration, we decided to proceed with separate interfaces for each one of the steps. As seen in our UML diagram, we have implemented *FilterStrategy* and *SelectionStrategy* interfaces which help us isolate the implementation of the individual algorithms. The reason to proceed with separate interfaces for each step was adopted keeping future extendability and breaking the code into functional



components. This way, each of the major steps, filtering and selection can be changed significantly without having to worry about its effect on the other step. That way, we ensured that the implemented architecture had low coupling and high cohesion. Hence, even though the alternative method a single unified interface did have the advantage of one single interface to handle all steps, we felt isolating them into their own categories would increase code maintainability and extensibility.

In addition to the interfaces, a Context class called *HandStrategyContext* has been implemented which maintains references to the concrete interfaces and communicates with the object, in this case, the NPC, that utilises the strategies. The context class was implemented as a wrapper to the interfaces which meant the NPC class itself does not need to worry about any future changes to the strategies as well as the exact implementation of the strategies. Instead of instances of strategies, the NPC keeps an instance of the context class and the context class becomes responsible for the execution of strategies.



This method of implementation allows us in the future, to add conditions and complex behaviour to filtering and selection logic without ever having to change the code inside NPC. Furthermore, the context class provides a setter to switch between strategies in future during runtime if necessary. Thus ensuring that our system has high cohesion and low coupling.

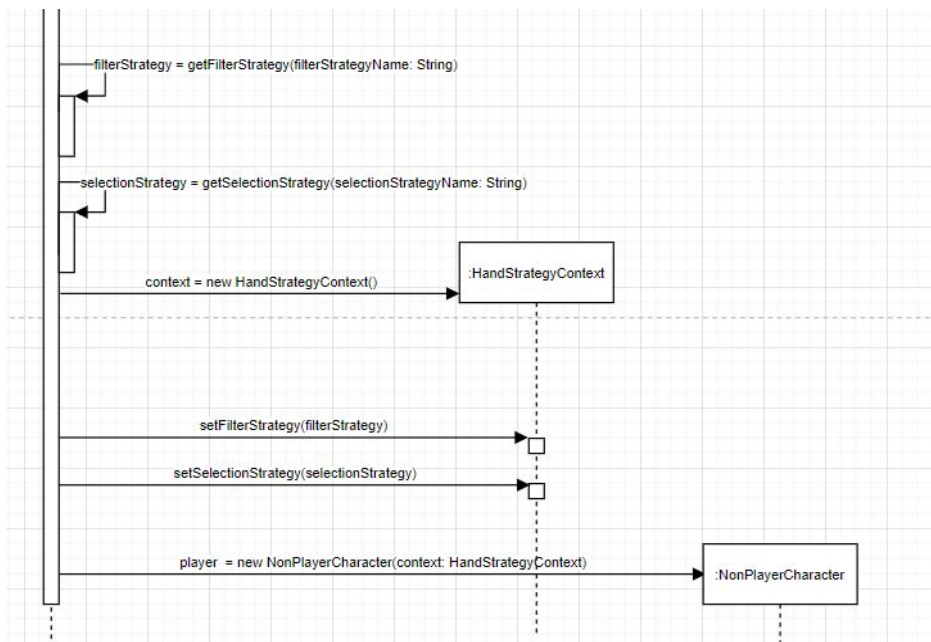
### **The responsibility of creating players**

In the initial design, the responsibility of creating players was handled by the Whist game class. This implementation was very rigid as a minor change of the player class could potentially affect the core game class as well as extending the number of different player types would result in changing a lot of code. To alleviate this condition we decided to apply the *Factory* pattern and implement a *PlayerFactory*. The *PlayerFactory* would be responsible for handling the logic for creating the appropriate types of players. The real benefit of such an implementation is evident during the creation of an NPC. Apart from handling the PC and NPC characters, the *PlayerFactory* class also handles the logic of instantiating the filter and selection strategies according to the game configuration files.

As seen in our Dynamic Sequence diagram, the *PlayerFactory* creates the *Player* abstract class as a product. Within its creation method, it contains the business logic of creating concrete *PlayerCharacter* and *NonPlayerCharacter* classes. At the bottom half of our 'Initializing individual players' diagram the creation of *NonPlayerCharacter* class also encompasses the creation of proper *FilterStrategy* and *SelectionStrategy* interfaces as well as the *Context* class.

Such an implementation means, in the future, we can very increase the different types of player types, NPCs and strategies with a minimal amount of friction with the rest of the codebase.

Lastly, we decided to make our *PlayerFactory* class a *Singleton* in our Design Class Diagram due to the nature of the game. In our code, we have created and passed the instance of the *PlayerFactory* to the *Settings Class* which stores the instances of the games players.



## The responsibility of dealing out cards

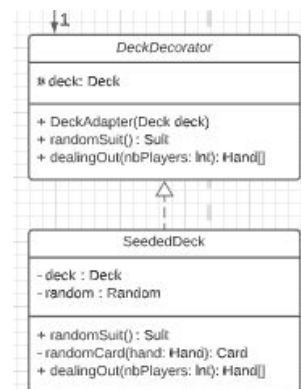
To be able to keep games consistent across the same seed, the game needed a custom logic for dealing out cards using a seeded random generator. We considered two classes to assign this responsibility to:

1. The *Round* class
2. A new *SeededDeck* class

The *Round* class has access to the *Deck* instance from the JCard Game library. Round also uses the card dealing logic, so assigning the responsibility to *Round* would have the lowest coupling.


However, the logic for dealing cards and the game logic in *Round* is not cohesive. So we decided to assign this responsibility to a new class. Because we needed to extend the functionality of the *Deck* class and be able to extend this at run time when the seed is provided, we decided to apply the *Decorator* pattern and create a *DeckDecorator* abstract class and a concrete implementation *SeededDeck*.

Because *SeededDeck* uses the *Decorator* pattern we can extend the code easily if we need more complicated logic for dealing out cards and do so at runtime.



## Smart Selection Strategy

As part of the system requirements, we are expected to develop a smart selection strategy. We have implemented a complex strategy which takes in multiple factors



such as the possibility of winning, availability of trump cards, player number during trick to determine the best card that needs to be played.

Our algorithm undertakes a value-based approach. For each card in the hand, the algorithm calculates the probability of winning the trick with that card. It takes into account the cards already played in the trick and how many players are still yet to play. The gain is equal to the probability of winning the trick. The cost of playing that card is set by the algorithm as a fixed cost. Cards of higher rank have higher costs and cards of the trump suit have higher costs. This incentivizes behaviour to reserve good cards. The algorithm then compares the value of each card (difference between gain and cost) and chooses the card with the highest value.

Our Smart Selection strategy was put against all other required algorithms and in all our simulations the smart selection algorithm consistently beat all the other algorithms consisting of a combination of various filtering and selection strategies.