# SWEN30006 Report –Project 2: Game of Thrones

## Part 1: Analysis of the current design

**There are some reasons that the current system is poorly designed and hard to maintain based on the GRASP principles:**

1. **Low cohesion**

   The most noteworthy problem in GameOfThrones is the lack of high cohesion. The GameOfThrones class has too many objects with more than 500 lines of code, making the class less focused, manageable and understandable. thus needs to be separate and refactored.

2. **Lack of functionality**

   The current design can not handle attack and defense info, properties for players and valid move checking, which needs to be added to extended GoT.

3. **No protected variations**

   The GameOfThrones class has all datas for hands, piles, and general information which needs to be protected and encapsulated by creating new classes and interfaces.

4. **No use of polymorphism**

   Current design checks player type and piles rank every time when a player needs to play a card which is not using the polymorphism principle. Subclasses can be made and handling new variations will become easy.

## Part 2: Proposed new design of GoT

**Based on the following issued we defined, we first simplify and refactor the system**

**and then add additional features.**

1. To obey the rule of pure fabrication, according to Project 1, a new class ‘propertiesloader’ is created to load all the properties of GoT, and new ‘initwithproperties’ method to call for initializing the data.

2. In order to reach the goal of high cohesion and low coupling, codes that are related to cards are encapsulated into a new class called CardUtils, and setting this as a final

class to limit its extensionality and avoid any misbehavior which could modify the existing card information from other classes.

3. To both reach creator and low coupling, we create a new class called playerfactory, to create different types of players. with a new polymorphism approach with AbstractPlayer, to separate the alternatives or behaviors vary by type, and store the same information as a players' parent class. On the other hand, the low coupling is also shown by creating a playerfactory, The got class no longer needs to access the constructor of each player, but can create players by accessing the playerfactory.

# Part 3: Analyzing different design patterns

**It is not difficult to find that this game has some flaws in coding design, so it is necessary to re-edit with other design pattern solutions.However, using too many design patterns will increase the number of classes.Below are some design patterns and corresponding diagrams.**

## Singleton pattern -- create a player (implemented from player factory)

To ensure that a class has only one player created at a time, while providing a global access point to this player.

Pros:

- can be sure the class has only one player
- gain a global access point to player

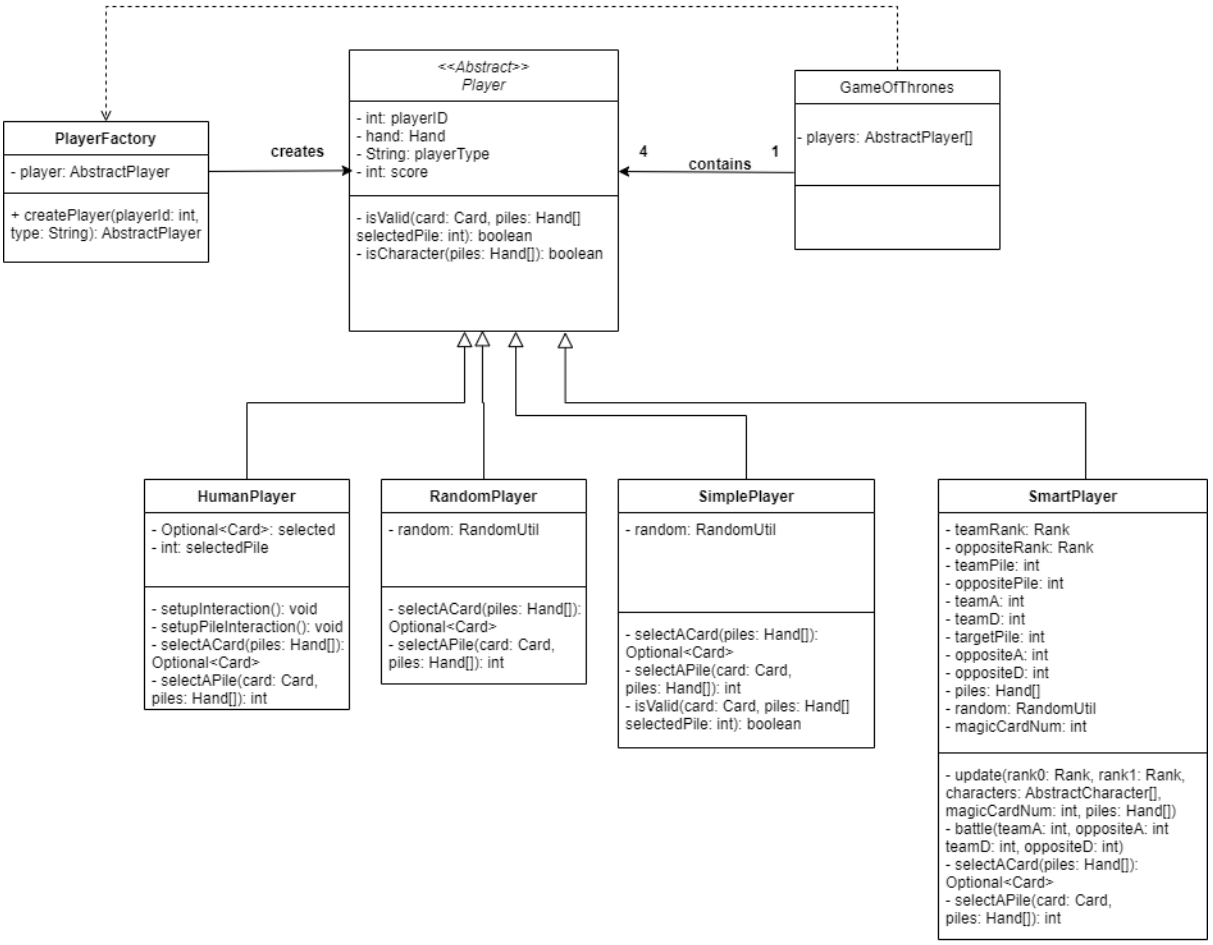
Cons:

- The Singleton pattern can mask bad design, for instance, when the components of the GoT know too much about each other.

If we use Facade patterns, a Facade class can also be transformed into a Singleton since a single facade object is sufficient in most cases.

## Factory Pattern -- player factory

In GoT, we have four different types of players: Human, Simple, Random, Smart, and they share many of the same properties like playerID, hand and score. So a factory pattern will be a great choice here to create these players, and each subclass can have these own information. By doing this,  polymorphism is also used to reduce the process of checking player types.




The playerfactory can also be declared as a Singleton class whose constructor is private as well to achieve better encapsulation.

Pros:

- avoid high coupling between the creator and each player
- polymorphism applied to reduce code redundancy.
- easy to maintain and extend.
- more customizable via subclasses

Cons:

- The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern.
- The smart player needs to observe more game stats than other players, but the strategy of playing is created and can not change after the player creation.

can evolve toward a Strategy pattern which is flexible, but more complicated, so the smart player can change its strategies at run time.

## Observer Pattern -- smart player observe the game states

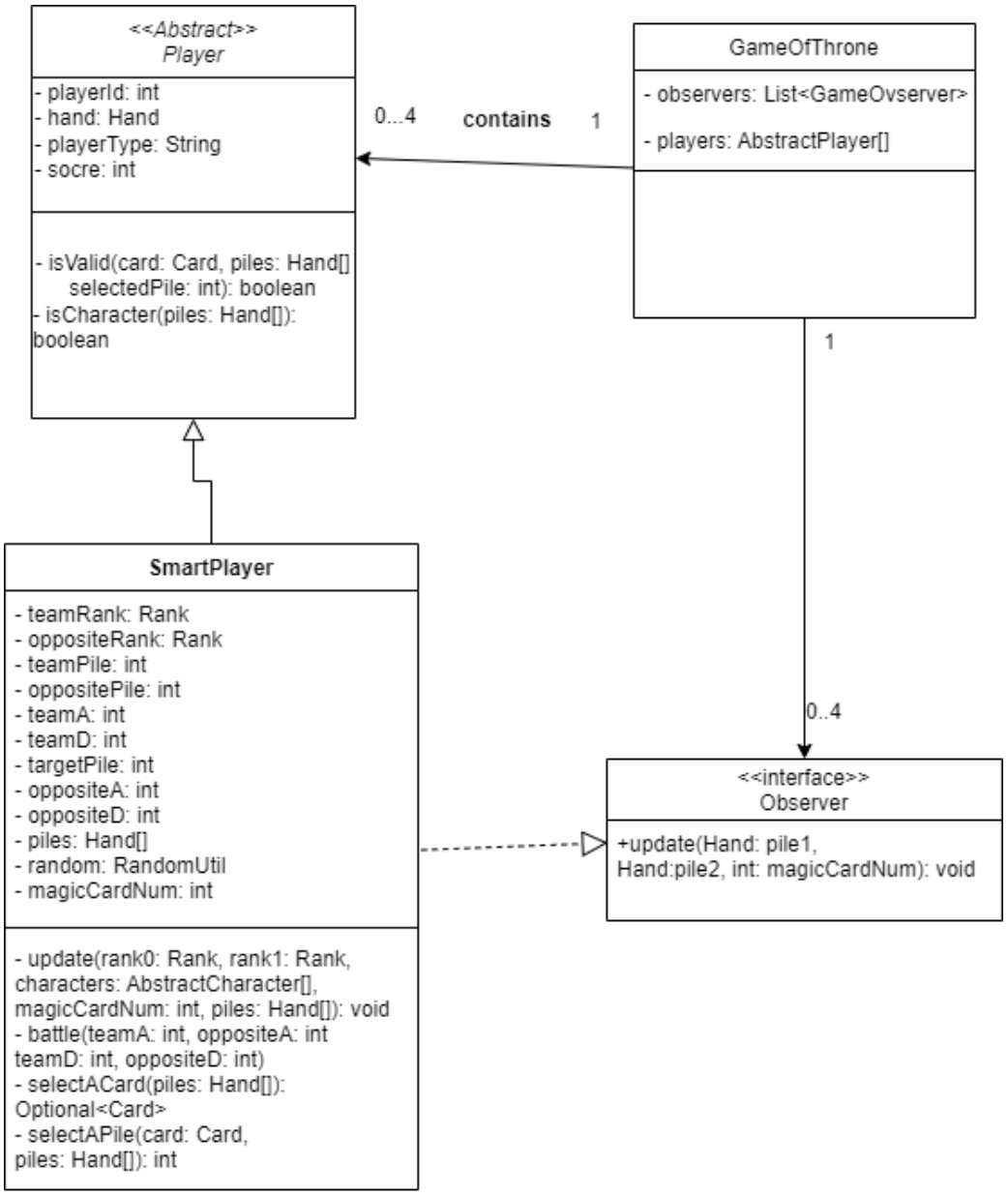
To implement the smart player strategy, an observer is used to notify players of the game state and the history of card deals, updating their internal states based on needs.

Pros:

- the player will record that information by itself instead of the GoT class, with high cohesion and low coupling achieved.
- can add more ‘subscriber’ class for player without having to change the publisher’s code

Cons:

- the strategy of a smart player can not change in run time.



This can be replaced with strategy pattern, so the strategy of the player can change at runtime, but with the smart player will need extra information, and the factory patterson will change to creating a different strategy of the player, loose cohesion will be shown as there is additional class.

## Decorator pattern/ helper function -- Calculate Attack and Defence

to calculate attack and defense data of the pile, we can use a helper function with output of a list of attack and defense. But this can also be done by using a decorator pattern which gets the attack and defence values from the latest decorator.

Pros:

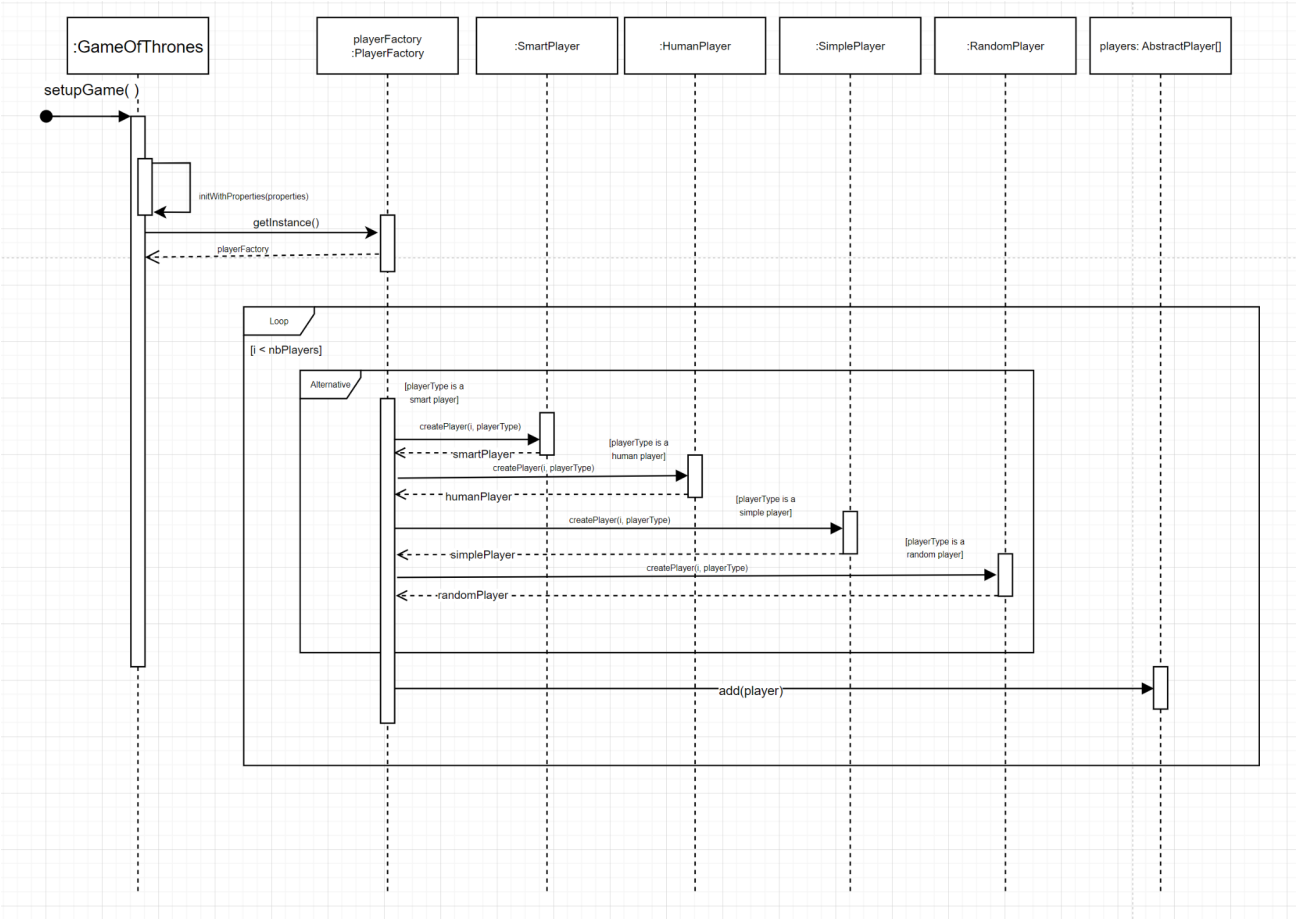
- can combine several game stat by wrapping an object into multiple decorators.
- can extend the way of calculation without making a new subclass.

Cons:

- In the GoT, we only need the attack and defense data of the pile, using a decorator pattern leads to a lot more classes created to store the information of three different effect cards.

For a better extentionality, we use decorator pattern to calculate attack and defense data of the pile, while as the helper function can do the same job but the design pattern suits future needs.

# Part 4: System sequence diagram of creating players




The system sequence diagram above shows how the GameOfThrones class creates different kinds of players. As can be seen from the diagram, the GameOfThrones first call the setupGame() function, and then get an instance of PlayerFactory. After that, using playerFactory to create different types of players by different properties and store them into the array players.