Report on Refactoring and Redesigning Game of Thrones

SWEN30006: Software Modelling and Design

Workshop 04 [Fri 9am], Team 04

Pratika Dlima 1268419

Shanaia Grace Chen 1214289

Xiaojiang Zheng 1302073

Introduction

While the digital version of Game of Thrones by Madness Industries works well, we redeveloped the program with GRASP and GoF principles and patterns for better maintainability and extensibility. This report documents the changes we have made.

Refactor and Design Decision

GameOfThrones Class

Prior to refactoring, the GameOfThrones class maintained all functionalities of setting up the game, dealing out initial hand cards and monitoring game progress. We needed **high cohesion** in place of dealing with all the logic inside GameOfThrones, which was why we created a GoTPlayMgr class that handles game logic. We let GameOfThrones monitor this object instead of maintaining game logic and executing each round itself. Initially, there was a huge, bulky GameOfThrones class which had too much work, this violates the **High Cohesion of GRASP principle**. We analysed the code and found some functions with related responsibilities, which is a hint to refactor them into a single class responsible for the related functionalities. We **grouped functions** into players and piles and created classes accordingly.

We also put player's cards into GoTPlayer class, which utilises **Information Expert**. Compared to GameOfThrones, GoTPlayer is more suitable for knowing what the hand cards are and dealing with the game logic accordingly. Using **Strategy pattern**, we created GoTPlayer as an abstract class, and implemented four types of Player control as subclasses. We also put common logic inside this abstract class and let the subclasses decide which functions to use based on their requirements.

The rest of the code is about play round execution and some static data. Firstly, we grouped all the data into another class specific for data management, apart from GoTPropertiesLoader. Some fields inside the GoTData are initialised automatically after GoTProperties. Additionally, some fields can be adjusted during runtime, while others cannot, which can be changed in future versions.

Using **pure fabrication**, we created a GoTPile class, which consisted of current piles and managed the calculation of current character status, which in turn utilised **Decorator pattern** and will be discussed later.

By putting all data into a GoTData class, we improved the **cohesion** of the program and made it easier for other classes to fetch configuration data and game setting data. While this has also caused high coupling between GoTData and all other classes that utilised the class, we prioritise cohesion more in this case. This is because high coupling with the GotData class is unlikely to be a problem as it is expected to have a stable interface over time.

GoTCharacter and Decorator pattern

In the initial design, the code for managing a GoT character was implemented in the GameOfThrones class. This was unrelated to the other responsibilities of the GameOfThrones class which contributed to the **low cohesion**. We strived to fix this problem by **encapsulating** code related to characters in separate classes. Our approach made use of the **decorator pattern** and is described below (see design class diagram for implementation details).

In our design, a team's character is an instance of the GoTCharacter class. According to the use case text, the attack and defence effects of a GoTCharacter should be continuously modified as cards are placed in a team's pile. We used the **decorator pattern** to dynamically modify attack and defence effects of a GoTCharacter by wrapping it with the appropriate GoTCharacterDecorator. This is delegated to the GoTCharacterFactory which depends on the card placed to the pile by the team. Assigning this complex responsibility to the GoTCharacterFactory, rather than the GameOfThrones class or some other existing class with unrelated responsibilities is an example of using **pure fabrication** to achieve **high cohesion**.

Furthermore, we used the decorator pattern so our design can be **easily extended**. This is because the decorator pattern uses **polymorphism**. That is, all effects **inherit** from the GoTCharacterDecorator class. Hence, more effected card behaviours can be easily added to the game by extending this abstract class and modifying the behaviour of `computeAttackAndDefend`.

For instance, we could increase the difficulty of the game by making a GoTTerrible class and directing the GoTCharacterFactory to wrap the character with this effect when a Jack is drawn. The GoTTerrible could easily be implemented to modify attack and defend through its `computeAttackAndDefend` method. Because **polymorphism**, such a change would have low impact to the current code and be easily implemented.

GoTPiles and Observer pattern

We created a GoTDisposePile instance to collect all the cards players have discarded from their hand. Aside from abiding to the **Information expert and Creator principles**, it also serves as an **observable** object. Players could implement an observer interface in case they want to know what cards have been played and what cards still remain unplayed. AI players can use this pattern to update their knowledge about current situations and decide the best moves for them.

GoTPlayer can also query GoTPiles, as illustrated later, for information of current piles, such that they can combine information obtained from GoTDisposePile to make decisions according to their implementation.

GoTPiles and Composite pattern

We made GoTPiles a **composite** of two piles using the library's Hand class, plus a GoTDisposePile. It's also a refactoring of GameOfThrones class because we put all the pile related functions into this class, thus increasing **high cohesion** of the program.

After the player plays a card, the pile adds that card to the disposal pile and current pile. When a GoTPlayer tries to make a decision, it can checkout the information disposal pile that has been updated for all the cards currently being played as well as checkout two piles on the board. Therefore, subclasses such as GoTSimplePlayer and GoTSmartPlayer can make decisions according to the information provided.

Implementation of Game Extension

Calculate Attack and Defend

As mentioned above, the decorator pattern was used to implement a GoTCharacter. Hence, the attack and defence effects of GoTCharacter were recursively calculated. Refer to the design sequence diagram: calculate decorated character's attack and defence for details.

GoTPlayer and GoTPlayerFactory

The GoTPlayer abstract class and GoTPlayerFactory were developed not only to lessen the burden on the GameOfThrones class, but also to account for differences in game logic between human and AI players. Nonetheless, much of their policies are similar, especially between non-human players whose requirements are built on top of each other. To **avoid replication of code** and to adhere to **high cohesion**, we decided to implement the abstract class through the **Strategy pattern** as mentioned above. This also demonstrates the use of **polymorphism** and **inheritance**. The former is more prominent in GoTHumanPlayer which has been **extended** for a more user-friendly gameplay experience, with the user having the chance to keep trying until they either make a valid move or pass their turn. On the other hand, the latter is mostly shown between the AI player classes: GoTRandomPlayer, GoTSimplePlayer, and GoTSmartPlayer. GoTRandomPlayer and GoTSimplePlayer implement the exact same method for selecting a pile. Additionally, we made use of **Pure Fabrication** through the GotPlayerFactory which is responsible for creating the different player types. Finally, in order to make 'smart' decisions on which card to play or whether to pass a turn, the GoTSmartPlayer class implements the GoT**Observer** interface and receives information from the disposal pile as well as the scores which extend from the GoT**Observable** abstract class.

Initially, we wanted GoTPlayer to be the one to 'play' or execute the game but decided against it as it would have been bloated with the various rules for the player types. Furthermore, we thought about having one big block of code for checking the legality of the player movements (heart and diamond cards), but this would have been difficult to

implement and extend, particularly if we wanted to allow human players to keep trying their turn.

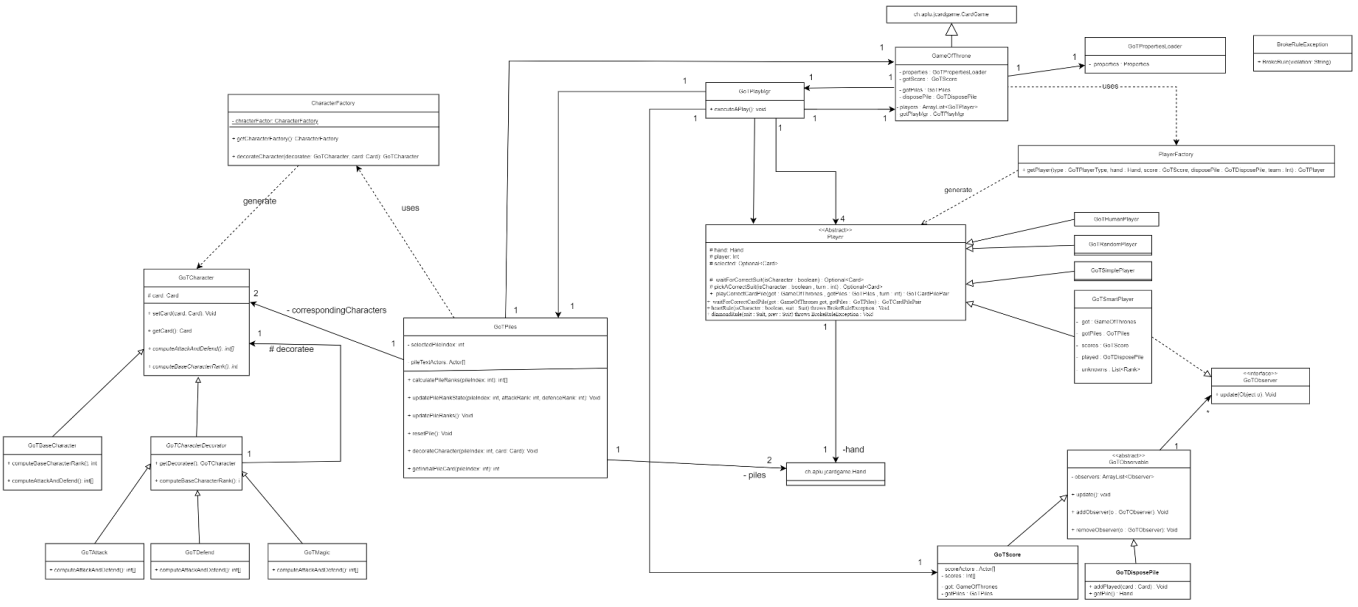
Load Property

Loading property is handled by the PropertiesLoader class, which was initialised in the main entry function inside GameOfThrones, and is passed in as an argument to GameOfThrones' constructor. The only argument it has is the property file path. And inside the constructor of the PropertiesLoader, it initialised the default properties and also initialised some values inside singleton data class DoTData.

The main responsibility of PropertiesLoader is to maintain the game's configuration and initialise important game data. It provides a query of values through the getProperty function, which returns a string representation of the value.

Diagrams

Design Class Diagram



Sequence Diagram for Recursively Calculating Attack and Defence Value