SWEN30006 Project 2, Game of Thrones

Monday Workshop, Group 7
Mohamad Danielsyah Mahmud 1190847
Khai Syuen Tan 1190030
Bhavika Shrestha 1065511

The original code design for Game of Thrones had all the code packaged into a single *GameOfThrones* class, making it overloaded with responsibilities and therefore exhibiting low cohesion. We aim to resolve this problem by creating new, smaller classes while maintaining the functionality and stability of the program, thus allowing us to apply the necessary extensions for the game.

Redesigned *GoT* Class

To decouple the *GameOfThrones* class, we broke down several methods and attributes into new submethods, which were then delegated to these smaller classes. These include the following:

- The rank and suit enumerators, the canonical string methods for each rank, suit, hand, and deck, and the logic of dealing out the cards and choosing a random card from a hand, are now handled by the *GoTCard* class.

- The *executeAPlay*() method was broken down into three methods: *playCharacters*(), *playEffectCards*(), and *finalizeScore*(), all of which are now implemented in the *GameLogic* class. Additionally, the setup of the hands array from *setupGame*() in *GoT* was moved to a new *setupHands*() method, which was also included in *GameLogic*.
    - In addition to this, portions of original code for different steps of a game were further developed and split into other classes such as the *Score* class and *Player* strategy classes, as discussed below.
    - The use of an array of bools for human and bot players was changed to an array of player strategies instead.

- All methods and attributes related to the locations of each game object and setup of the graphics, as well as adding and removing actors from Game of Thrones, including the *updatePileRankState*() method, were delegated to a *GameGraphic* class.

- All methods related to the piles on the centre of the game, such as setting and resetting the piles and calculating the pile ranks, were moved to a *GoTPiles* class.

- A new *Player* interface class was created, which was implemented by the *Human* and *Bot* classes.
    - The *pickACorrectSuit*() method was put in a new *playSuit*() method in the *Bot* abstract class. The abstract *Bot* class was extended to the *RandBot* Class and

    abstract *BrainBot* Class, which was then further extended to the *SimpleBot* and *SmartBot* children classes.
  - As a result, the *PlayerFactory* class was created to handle these player types.

- All score-related methods (e.g. *updateScore*()) were moved to a new *Score* class. Additionally, a portion of the code from the *finalizeScore*() method was moved to a new *battleScores*() method in that *Score* class.

- The final design for the *GameOfThrones* class therefore only has the responsibility of setting up the game and containing game objects such as the hands, the players, and the selected card to be used by other classes. Other classes handle the other core responsibilities and the logic on how to properly interact and use the game objects. Other design proposals were considered, but could not be implemented into the final design as discussed below.

GoTCard

The ***GoTCard*** class is the Information Expert on the logic and properties of the cards in the game, such as the values of each suit and rank and the logic of dealing cards at the beginning of the game. We initially considered separating the card attributes and logic into two different classes (*CardInfo* and *CardLogic*) from the main *GameOfThrones* class; however, we found that these classes would be far too coupled with one another and the GameOfThrones class, leading to high dependency.

The *GoTCard* class also applies the Singleton pattern, as every card in the game shares the exact same metadata information and game interaction logic, which does not change throughout the game. Therefore, only one instance of the class is allowed, which was achieved through the static *getInstance*() method that creates a GoTCard instance if it does not exist and then returns it.

GameLogic

The ***GameLogic*** class is the Information Expert on the logic of the overall game. It is responsible for setting up the hands array, the interaction between the hands and the human player, as well as the game's runtime logic.

Initially, the *executeAPlay*() method from the original *GameOfThrones* class design was significantly bloated and hard to read. So, it was broken down into three submethods, corresponding to different parts of a round. These methods are then executed in the new *playARound*() method to increase cohesion and code readability. These methods are:

1. *playCharacters*(), which contains the logic for playing the first two hearts,
2. *playEffectCards*(), which contains the logic for playing the effect cards after playing the heart cards, and

3. *finalizeScore*(), which calculates and updates the score based on the attack and defence of each card.

Furthermore, the majority of the code from *finalizeScore*() was moved to the *battleScores*() method in the *Score* class to further increase cohesion in the *GameLogic* class and allows prediction based on scores in the newly added SmartBoth.

GameGraphic

The ***GameGraphic*** class serves to initialise and update the graphics of the overall game, specifically in the scores and piles, and be the Information Expert on the attributes of those graphics. It is kept separately from *GameLogic* to achieve higher cohesion for the latter class, despite creating higher coupling with *GameOfThrones*.

Additionally, many of the methods in the original *GameOfThrones* class, such as *updateScores*() and *updatePileRanks*() had misleading names. For instance, one would assume from them that an element in the *Score* and *GoTPiles* classes would be changed, but upon closer inspection only the actors of those objects would be modified. Therefore, those classes were moved to *GameGraphic* and renamed to make it less deceiving.

GoTPiles

The ***GoTPiles*** class is the Information Expert for both piles located in the centre of the game. Multiple instances of the class can exist at the same time, so that each smart bot can keep track of both piles during the game. As the *SmartBot* class is an Observer of this class, *GoTPiles* contains an array of *SmartBot* objects. Whenever any of the contained piles is updated, *GoTPiles* will notify all SmartBots with the *notifySmartBot*() method to update their "knownMagicCards" array, thus acting as the publisher of the Observer pattern system. All SmartBots are registered to the GoTPiles instance through the *registerObservers*() method during initial game setup.

GoTPiles also uses the Indirection principle to serve as an intermediary between *GameOfThrones* and *GameGraphic*, so that if *GameOfThrones* needs to modify the piles, it must be done through that class, thus increasing stability in the program and creating low coupling.

Because of this, *GoTPiles* was separated from *GameOfThrones* to prevent unnecessary coupling and lower cohesion. If all classes and methods of *GoTPiles* were to remain in *GameOfThrones*, every instance of the *Pile* of type *Hand* would need to notify all *SmartBots* and be directly accessed by *GameLogic*, leading to higher coupling.

The *calculatePileRanks*() method was also expanded to allow for correctly calculating the attack and defence values of all cards on the pile, thus implementing the necessary extensions to *GameOfThrones*. Its design sequence diagram shows that it uses conditional statements based on the type of each card and whether the value of the previous card played is equal to the value of the current card. **(refer to "Calculating values(attack and defence) of a pile" in design sequence diagram)**

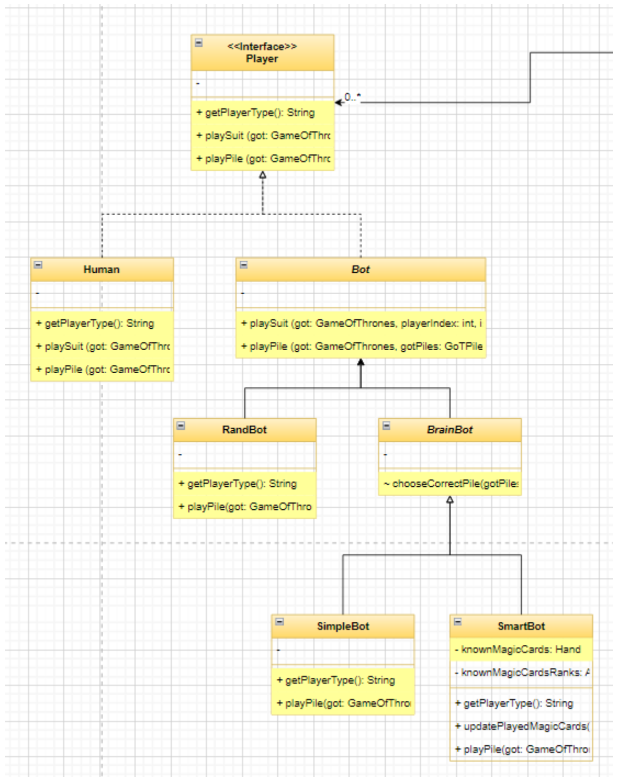
Player

The ***Player*** interface class was created using the Strategy pattern to achieve Protected Variation for each player strategy type, as each player type is meant to act separately from one another. This allows us to utilise polymorphism to account for each varying player strategy in the game while connecting it to the common *Player* interface class.

Human

The ***Human*** Concrete Strategy class was separated from the *Bot* class to allow interaction between the player and the program through input listener methods such as *setTouchEnabled*(). This allows for easier modification of the class and be independent from the other player types without affecting them, maintaining the Protected Variation principle.

Bot and *BrainBot*



<<Interface>>
Player
-
+ getPlayerType(): String
+ playSuit (got: GameOfThrc
+ playPile (got: GameOfThrc
0..*
Human
-
+ getPlayerType(): String
+ playSuit (got: GameOfThrc
+ playPile (got: GameOfThrc
Bot
-
+ playSuit (got: GameOfThrones, playerIndex: int, i
+ playPile (got: GameOfThrones, gotPiles: GoTPile
RandBot
-
+ getPlayerType(): String
+ playPile(got: GameOfThro
BrainBot
-
~ chooseCorrectPile(gotPile:
SimpleBot
-
+ getPlayerType(): String
+ playPile(got: GameOfThro
SmartBot
- knownMagicCards: Hand
- knownMagicCardsRanks: A
+ getPlayerType(): String
+ updatePlayedMagicCards(
+ playPile(got: GameOfThro

To account for multiple variants of the bots in the game, an abstract ***Bot*** class was created with the Strategy pattern also in mind. It holds a *playPile*() method from which each bot can implement its own strategy for playing an effect card on the pile. This allows us to easily modify each bot type while keeping them related to one another. It also serves as a Template for the *RandBot* and *BrainBot* classes, since all bot classes use the same implementation of the *playSuit*() method.

The abstract ***BrainBot*** class was designed with the Template pattern in mind by providing a protected *chooseCorrectPile*() method that chooses a pile based on the type of card selected to prevent any advantages to the enemy team or disadvantages to the player's team, from which both *SimpleBot* and *SmartBot* concrete strategy classes is extended from. It also utilises the Indirection pattern by serving as an intermediary between *Bot* and the two aforementioned bot classes to create lower coupling.

RandBot

The ***RandBot*** Concrete Strategy class serves as the random player type, which acts the same way as the original game design, except it is now modified slightly to pass if it attempts to play a diamond on top of a heart.

SimpleBot

The ***SimpleBot*** Concrete Strategy class serves as the simple player type. It acts like the random player type, except it implements *BrainBot* to choose the correct pile so that it can play fairly.

SmartBot

The ***SmartBot*** Concrete Strategy class handles the responsibilities for the smart player type. As demonstrated in its design sequence diagram for its *playPile*() method , it calculates the initial score and final score based on the battle outcome of each round, and then decide whether to play the card on hand or pass the turn based on the battle outcome and the type of card in play in another player's hand. Like *SimpleBot*, it implements *BrainBot* to choose the correct pile based on the type of card at hand. **(refer to "Play Effect Cards onto the piles" in design sequence diagram)**

It is an Observer of the *GoTPiles* class, acting as the subscriber to *GoTPiles* as a publisher. Through *GoTPiles* notifying *SmartBot*, *SmartBot* is able to update the number of magic cards played and thus play an effect card only when necessary.

PlayerFactory

The ***PlayerFactory*** class utilises the Singleton Factory pattern and the Creator principle in which it creates a new player type object based on the parameters set in the property file. Since the type of players do not change throughout the game, to maintain this single instance of the *PlayerFactory* class, we used that Singleton pattern by creating a *getInstance*() method and implementing it in the *GameOfThrones* class. This allows us to implement the properties extension for GoT and create new player types if needed.

Score

The ***Score*** class is the Information Expert in relation to the scores of the game. It is used by both the *SmartBot* and *GameLogic* classes and dependent on the *GameGraphic* class. It is separated from *GameOfThrones* because since *SmartBot* needs to be able to predict the score, the scores themselves cannot be placed in another class such as *GameLogic*. In addition, the *battleScores*() method is used both in *SmartBot* and *GameLogic*, and since it cannot be in *SmartBot*, it must exist in a different class. Furthermore, it utilises Indirection to facilitate access to *GameGraphic* from *GameOfThrones*, and so any changes to the score can be done through this class without affecting *GameGraphic* directly. Therefore, this creates higher cohesion and prevents unnecessary coupling between *Score* and any other class.

Like *GoTPiles*, multiple instances of the score object can exist simultaneously. This allows us to keep track of the initial score and final score so that the smart bot can accurately predict the outcome of the game.

Other Considerations

- We considered simply returning the “selectedPileIndex” variable inside of the *GameOfThrones* class instead of letting each player class directly modify it to achieve protected variation and lower coupling. However, since the *Human* class requires input from the computer, through methods such as *delay*() and *setTouchEnabled*() that required directly modifying “selectedPileIndex”, it would affect the guards used to return these methods inside of *Human*, and so all player type classes would have to follow the method implementations derived by *Human*.

- The hands array remained in *GameOfThrones*, since it is dependent on the *Player* class types, which are not dependent on *GameLogic*, so that the *Player* class types can access the hands from *GoT*, preventing unnecessary coupling.

- A *setupGUI* class was considered at one point to delegate the *setupGame*() and the graphics responsibilities into a separate class, aiming to create higher cohesion and increased readability in *GameOfThrones*. However, doing so would require increased dependency on the following:
    - the hands attribute from *GoT*,
    - *setupHands*() from *GameLogic*,
    - *setRandom*() from *GoTCard*, and
    - *resetPile*() from *GoTPiles*,

  creating increased coupling between these four classes. Therefore, *setupGame*() remained in *GoT*, while the graphics were delegated to a new class as discussed above, allowing only three classes to be dependent on that class.