Game of Thrones Extended

0.0 Introduction

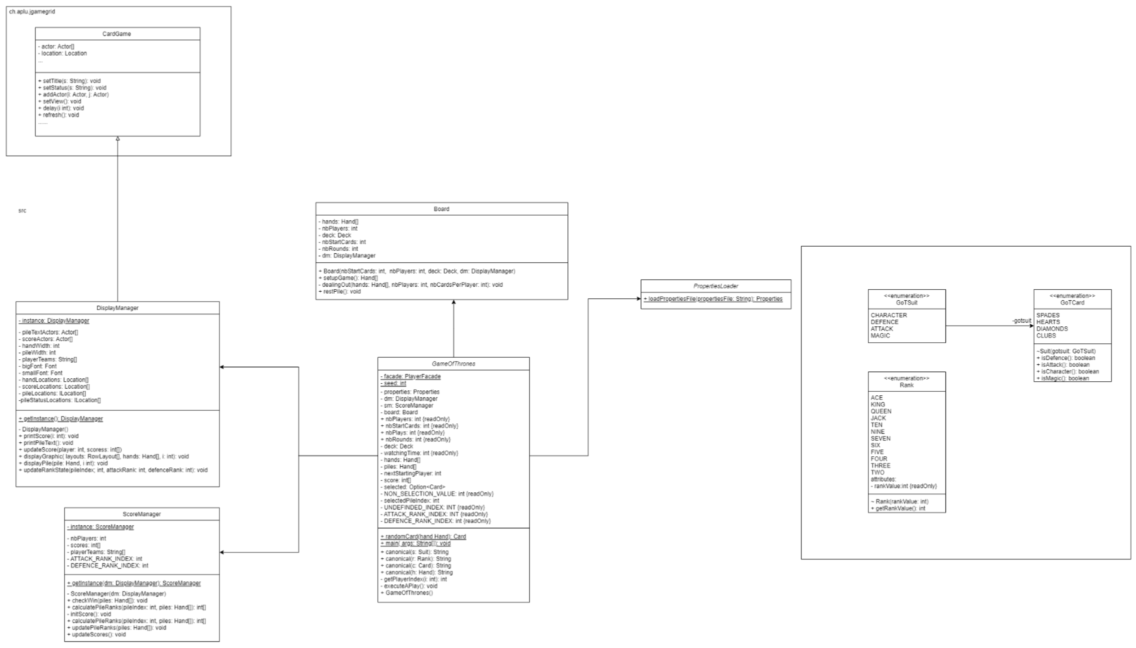
The original GoT framework had poor design as all the variables, methods and main were in a single class GameOfThrones. Low cohesion is also identified as the GameOfThrones class is responsible for all the tasks relating to the game, from card initializing, playing the turns and also running the game itself. This shows that there is no well-focused purpose for this class and these problems should be addressed before extending the code further. By separating the responsibilities in version 2.0, our code aims to satisfy non-functional requirements such as maintainability and reliability and provide clarity for future extensions.

Aspects of playing diamond cards correctly, evaluating the scores correctly and reading the properties files had to be addressed before having different types of players play the game. We constructed a UML to discuss the proper GOF patterns that needed to be implemented into the system.

1.0 Class Seperation: one class - one responsibility principle:

As aforementioned, one major problem with the current design is there is only one class (GameOfThrones) which takes responsibility for the whole entire program. This creates a bloated class with poor readability and is extremely hard to extend and maintain. Therefore, the first step of making improving GameOfThrone is to separate the bloated GameOfThrones class into many small classes; which we will be using the **"one class - one responsibility"** principle discussed in week 4.

a. **DisplayManager:**
   We have separated the UI creation into a separate **singleton** class called DisplayManager. This class will take input from GameOfThrone class and be responsible for any printing information on the terminal or displaying graphics on the screen. This class extends the CardGame class from ch.aplu.jcardgame package.
b. **ScoreManager:**
   The ScoreManager class is responsible for all the tasks that are to do with managing the score. This includes storing, initialising, and calculating(extension) the score. We also made this class a Singleton since there should only be 1 single instance of ScoreManager for the whole program.
c. **Board**:
   The Board class is responsible for the logic of the game; namely, the number of players, the number of cards, setting up the game and resetting the pile.
d. **PropertiesLoader**:
   This class ensure all the given properties are being read correctly. If there is no given property, this class will set the game in default input.
e. **GameOfThrones (modification):**
   We have significantly lessened the amount of responsibility of the mother class to the aforementioned classes. Currently, GameOfThrones will only be responsible for executing the play and calling other classes for based on the given task.
f. **GoTCard**:
   We have separated GoTCard into a separate enum class.


ch.aplu.jgamegrid
CardGame
- actor: Actor[]
- location: Location
...
+ setTitle(s: String): void
+ setStatus(s: String): void
+ addActor(i: Actor, j: Actor)
+ setView(): void
+ delay(i int): void
+ refresh(): void
......
src
Board
- hands: Hand[]
- nbPlayers: int
- deck: Deck
- nbStartCards: int
- nbRounds: int
- dm: DisplayManager
+ Board(nbStartCards: int, nbPlayers: int, deck: Deck, dm: DisplayManager)
+ setupGame(): Hand[]
- dealingOut(hands: Hand[], nbPlayers: int, nbCardsPerPlayer: int): void
+ restPile(): void
PropertiesLoader
+ loadPropertiesFile(propertiesFile: String): Properties
DisplayManager
- instance: DisplayManager
- pileTextActors: Actor[]
- scoreActors: Actor[]
- handWidth: int
- pileWidth: int
- playerTeams: String[]
- bigFont: Font
- smallFont: Font
- handLocations: Location[]
- scoreLocations: Location[]
- pileLocations: ILocation[]
-pileStatusLocations: ILocation[]
+ getInstance(): DisplayManager
- DisplayManager()
+ printScore(i: int): void
+ printPileText(): void
+ updateScore(player: int, scoress: int[])
+ displayGraphic(layouts: RowLayout[], hands: Hand[], i: int): void
+ displayPile(pile: Hand, i int): void
+ updateRankState(pileIndex; int, attackRank: int, defenceRank: int): void
GameOfThrones
- facade: PlayerFacade
- seed: int
- properties: Properties
- dm: DisplayManager
- sm: ScoreManager
- board: Board
+ nbPlayers: int {readOnly}
+ nbStartCards: int {readOnly}
+ nbPlays: int {readOnly}
+ nbRounds: int {readOnly}
- deck: Deck
- watchingTime: int {readOnly}
- hands: Hand[]
- piles: Hand[]
- nextStartingPlayer: int
- score: int[]
- selected: Option<Card>
- NON_SELECTION_VALUE: int {readOnly}
- selectedPileIndex: int
- UNDEFINDED_INDEX: INT {readOnly}
- ATTACK_RANK_INDEX: INT {readOnly}
- DEFENCE_RANK_INDEX: int {readOnly}
+ randomCard(hand Hand): Card
+ main(args: String[]): void
+ canonical(s: Suit): String
+ canonical(r: Rank): String
+ canonical(c: Card): String
+ canonical(h: Hand): String
- getPlayerIndex(i: int): int
- executeAPlay(): void
+ GameOfThrones()
ScoreManager
- instance: ScoreManager
- nbPlayers: int
- scores: int[]
- playerTeams: String[]
- ATTACK_RANK_INDEX: int
- DEFENCE_RANK_INDEX: int
+ getInstance(dm: DisplayManager): ScoreManager
- ScoreManager(dm: DisplayManager)
+ checkWin(piles: Hand[]): void
+ calculatePileRanks(pileIndex: int, piles: Hand[]): int[]
- initScore(): void
+ calculatePileRanks(pileIndex: int, piles: Hand[]): int[]
+ updatePileRanks(piles: Hand[]): void
+ updateScores(): void
<<enumeration>>
GoTSuit
CHARACTER
DEFENCE
ATTACK
MAGIC
-gotsuit
<<enumeration>>
GoTCard
SPADES
HEARTS
DIAMONDS
CLUBS
~Suit(gotsuit: GoTSuit)
+ isDefence(): boolean
+ isAttack(): boolean
+ isCharacter(): boolean
+ isMagic(): boolean
<<enumeration>>
Rank
ACE
KING
QUEEN
JACK
TEN
NINE
SEVEN
SIX
FIVE
FOUR
THREE
TWO
attributes:
- rankValue:int {readOnly}
~ Rank(rankValue: int)
+ getRankValue(): int

By separating into these classes, we ensure the readability of the code. Furthermore, this will also improve the cohesion of the program which makes any extension or maintenance in the future will become much easier.

2.0 Following GRASP principles for extension

We were tasked to extend the program as follow:

- All players must follow the given GameOfThrones rules, if illegal move was made, through an exception.
- The given score displaying system is incorrect, we have to fix it.
- Make 4 types of players: Human (Manual), Random, Simple, Smart (Auto) given the specs
- Add a property reader class and ensure it reads the properties correctly; if property was not given, set the game as default.

While adding the extensions, we made sure we were implementing different patterns namely Strategy Pattern (section 4.0), Facade Pattern (section 5.0) to follow the core principles:

a. **Single responsibility principle:**

Every class created should have one well-focused responsibility. Current class that has too many responsibilities will be modified to be simplified. GameOfThrones' responsibilities were spread to new classes following creation and information expert principles 1.0.

b. **Open-closed principle:**

New strategies are open to unique algorithms however should still be able to make legal moves. Protected Variation is seen as new strategies have to implement PlayerStrategies meaning they have to return a valid card and valid pile in the correct form. Illegal cards will not be played and will not break the game as GameOfThrones uses Rules.validate to check if the selected card can be played on the pile. This means that new strategies are free to use whatever algorithm they want as they follow the rules (addressed in BrokenRuleException). If the game rule changes, they can make new Rules by adding only to the Rules.validate method.

c. **Interfaces segregation principles:**

Player strategies do not have to use shortlist strategies if they do not need it. This is shown as only smart players use shortlist strategies. New player strategies are allowed to use shortList strategies if they want to.

d. **Dependency inversion principle:**

All strategies depend on PlayerStrategies which is an abstract class with abstract methods (no concretion). This is also why PlayerStrategies do not inherit from each other as if they do they would inherit concretions.

3.0 BrokeRuleException

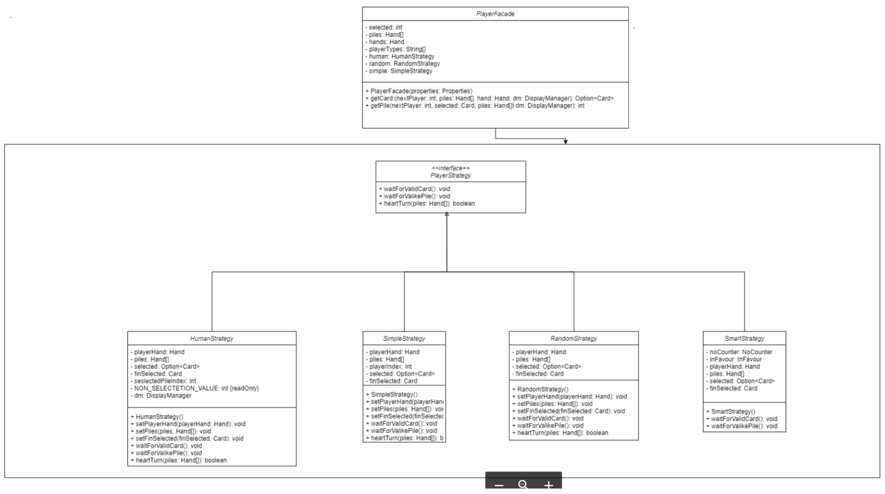
The rule of the game is as follows:

1. Each pile starts with a heart card
2. A diamond can not be placed on top of a heart card
3. A heart card can not be played on top of any card

The concerns of the rules implies knowledge of the **Card** played, **SelectedPileIndex** and the state of the **Piles**. Instead of having the rules checked every time, we wrote a class called Rules that, given the appropriate parameters, will validate whether the card can be played; if not it will throw an exception. Rules.validate is a public static function and allows users to use this function without needing to create any classes. This means that there is no unnecessary coupling and allows each strategy to check if their card can be played. Rules.validate works by taking each pile's top card suit and comparing it to the suit of the card being played on top of it. If any of the piles[i].getLast() is null, this exception is caught and if the card played is not a character card, it creates a new BrokeRuleException as the card played in an empty pile must be a heart card. If we are able to get to the top of the piles and try to place a **magic card on top of a heart card**, or a **heart card anywhere on an existing pile** they both violate the rules of the game and a BrokeRuleException is thrown each time. Having this function is useful when trying to implement the strategies later on in the game. During GameOfThrones.executeAPlay() this function is used to check before transferring a selected card to the pile if the card can be played on the pile; this means that if in future strategies they try to make an illegal move, the card will not be able to be played for that player. This makes sure that the game will only be played with players that make legal moves meaning that illegal players won't be able to break the game and legal players can still play the game.

4.0 Appling a strategy pattern

Strategy pattern solves the problem for designing varying yet related algorithms by defining each algorithm separately in the class but all the classes will implement a common interface. By sharing the same interface, each class must implement the same methods (with appropriate return types) and this further allows each algorithm to have their own algorithms implemented however they want making it more extendable. Our strategy pattern is shown in the figure below:


PlayerFacade
- selected: int
- piles: Hand[]
- hands: Hand
- playerTypes: String[]
- human: HumanStrategy
- random: RandomStrategy
- simple: SimpleStrategy
+ PlayerFacade(properties: Properties)
+ getCard:(nextPlayer: int, piles: Hand[], hand: Hand, dm: DisplayManager): Option<Card>
+ getPile(nextPlayer: int, selected: Card, piles: Hand[]l dm: DisplayManager): int
<<interface>>
PlayerStrategy
+ waitForValidCard(): void
+ waitForValikePile(): void
+ heartTurn(piles: Hand[]): boolean
HumanStrategy
- playerHand: Hand
- piles: Hand[]
- selected: Option<Card>
- finSelected: Card
- seslectedPileIndex: int
- NON_SELECTETION_VALUE: int {readOnly}
- dm: DisplayManager
+ HumanStrategy()
+ setPlayerHand(playerHand: Hand): void
+ setPiles(piles: Hand[]): void
+ setFinSelected(finSelected: Card): void
+ waitForValidCard(): void
+ waitForValikePile(): void
+ heartTurn(piles: Hand[]): boolean
SimpleStrategy
- playerHand: Hand
- piles: Hand[]
- playerIndex: int
- selected: Option<Card>
- finSelected: Card
+ SimpleStrategy()
+ setPlayerHand(playerHanc
+ setPiles(piles: Hand[]): voi
+ setFinSelected(finSelectec
+ waitForValidCard(): void
+ waitForValikePile(): void
+ heartTurn(piles: Hand[]): b
RandomStrategy
- playerHand: Hand
- piles: Hand[]
- selected: Option<Card>
- finSelected: Card
+ RandomStrategy()
+ setPlayerHand(playerHand: Hand): void
+ setPiles(piles: Hand[]): void
+ setFinSelected(finSelected: Card): void
+ waitForValidCard(): void
+ waitForValikePile(): void
+ heartTurn(piles: Hand[]): boolean
SmartStrategy
- noCounter: NoCounter
- inFavour: InFavour
- playerHand: Hand
- piles: Hand[]
- selected: Option<Card>
- finSelected: Card
+ SmartStrategy()
+ waitForValidCard(): void
+ waitForValikePile(): void

PlayerStrategy is an abstract class has two abstract methods:

1) Optional<Card> waitForValidCard() = each strategy must have a function that returns a valid card or an empty card (players are allowed to skip after the second turn).
2) int waitForValidPile() = each player must have a function that chooses a valid pile for their chosen card.

Both of the methods are abstract meaning they must be implemented in the subclasses. The functions also do not take any parameters meaning that each strategy is allowed to have their unique setters for their variables. Each strategy needs different variables and therefore the functions will not take in variables. PlayerStrategy also has a function heartTurn(Hand[] piles) that given the state of the piles will return true or false; this is because all players should be aware that if it is a heart turn they are not allowed to pass.

5.0 PlayerFacade

As seen in the strategy pattern above, it is clear that the subsystem of player strategies are very complex as each one requires knowledge for different variables. As discussed before the GameOfThrones class is in charge of reading the properties file and therefore has the knowledge of the types of player, however if we let the GameOfThrones handle each player's unique strategy everytime we create a new strategy we have to update their moves GameOfThrones and this my case other aspects of the game to be broken; this is poor design as it will increase the complexity of our code and cause low cohesion. Our solution was to create a class called PlayerFacade that is incharge of handling the PlayerStrategy subsystem. The type of Player (outlined in properties) information is set to initialize the PlayerFacade. This means that the PlayerFacade has knowledge of each player's strategies and records them in a String[4] array called playerTypes. PlayerFacade has 3 methods:

1. Optional<Card> getCard (int nextPlayer,….)
    a. PlayerFacade will call the playerTypes[nextPlayer] that gives the type of player for that index. For each type of player, PlayerFacade will set the appropriate variables needed for the strategies waitForValidCard() function. Since multiple players have the same type of strategy, PlayerFacade sets the variables everytime the function is called.
2. int getPile(int nextPlayer, Card selected, piles…)
    a. Similar to getCard() handling.
3. void updateFacade(Hand[] piles)
    a. This void function is called by GameOfThrones everytime the board resets. Although currently updateFacade only updates the smart player, this kind of function can be reused to send the previous piles to other PlayerStrategies in the future without much change.

GameOfThrones only uses the facade 4 times in the entire code.

1. Initialise PlayerFacade with player properties
2. Call getCard for the next player
3. Call getPile for the next player
4. Update PlayerFacade when the pile resets

With PlayerFacade, GameOfThrones is given a single access point to the PlayerStrategies yet it does not have to deal with each strategy's complex setters. PlayerFacade wraps the subsystem and handles the collaboration with the PlayerStrategy's components. PlayerFacade acts as a controller class for GameOfThrones and handles all the nextPlayer strategy requests for executeAPlay(). Since PlayerFacade only takes in certain parameters, future players are not able to do certain things (as they are controlled by PlayerFacade to be able to be played on GameOfThrones) such as look at other players hands and play a card that is not in their hand (as PlayerFacade only takes in the current players hand and the state of the piles.

6.0.1 RandomStrategy

The random player uses Rules.validate on every card in their hand and on every pile to see if the card is playable. This means that it will only shortlist hearts on heart turn, and it will never shortlist a magic card right after a heart turn. After the random player shortlists its cards, if the list is empty the selected card is empty. The selected card is also empty as the player will randomly skip 1 in 4 turns if it is a skippable turn (not a heart turn). Otherwise, out of the shortlisted cards, it will pick a random card to play.

For pile selection the card is validated against each pile, and the array of piles that it can be played on is recorded; the size of the array tells us how many piles it can be played on. If the number of piles is 2, the random player will randomly choose one of the two piles. If the player can only play on 1 pile, the random player has to choose that pile for the turn to be valid.

6.0.2 SimpleStrategy

Much like RandomStrategy, the simple player shortlists the cards it can play on each pile using Rules.validate. However, a magic card is only shortlisted if the pile that it can be played on is the enemy pile (e.g. simple players 0 on 2 should only play magic cards on pile 1 as pile 1 is the enemy pile). This means that magic cards that can be played on their pile (only) will not be shortlisted. A simple player also chooses the card and skips a turn similarly to the random player.

SimpleStrategy's pile selection is completely different from RandomStrategy's pile selection; for the simple player, if the card selected is a magic card, the pile will always be the enemy team's pile. Every other card (character, attack, defense) should be played on their own pile as simple players will play cards that boost their attack and defense on themselves and magic cards to decrease enemies attack and defense on the enemy player.

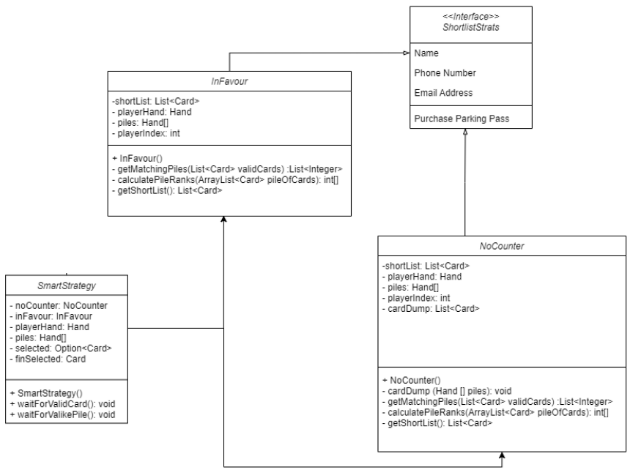
Although there is a little bit of code reuse between RandomStrategy and SimpleStrategy, we decided against have simple players inherit from RandomStrategy. This is because both strategies have their own unique aspects (simple player is aware of their own player index, and chooses turns to benefit themselves) and using inheritance may confuse the classes if changes were to be made in some strategy.

6.0.3 HumanStrategy

The HumanStrategy is different from the other players in that it requires mouse input rather than random generation. This means that HumanStrategy is also concerned with controlling the displayManager for the mouse click inputs. For the waitForValidCard() function, the HumanStrategy uses a cardListener for their hand. Then the HumanStrategy only selects a card if the card is allowed to be played in some pile. To implement that we had a shortListCards array for all the cards that can be played by looping through all the cards in the player's hand and all the piles and validating each one. A card is only selected if it is one of the cards in the shortListCard array. The player is only allowed to skip if it is not a heart turn or if the shortListCard array is empty.

For HumanStrategy pile selection, a pile listener is created. A pile is only selected if it is a valid pile. This means that if no pile has been selected or if the pile selected is not a pile the card and be played the waitForValidPile function will still be running.

6.0.4.1 SmartStrategy (pt 1)



The smart player strategy must first evaluate 2 different Shortlisting card strategies:

1. InFavour = a shortList of cards that will change the outcome of the battle if the battle took place after the card is played.
2. NoCounter = a shortList of cards that will not have a double diamond counter to that card. (no played has the equivalent value diamond card in their hand).

These two shortlisting techniques share the same interface as they both used to evaluate the hand and choose cards with these traits (return a shortlist of cards). This allows more shortListing techniques to be implemented in the future for other player's strategies. Or for other players to also be able to use these techniques. This is why we chose to separate these techniques and not make them exclusive to only the smart player.

6.0.4.2 InFavour

InFavour first evaluates all the valid cards it can play. InFavour has its own calculate score function that takes the hand as an array of piles instead (this is because we had some visual issues putting the card into the hand). By having its own calculate function, it is able to adjust it if it wants to change its strategy without being directly linked to the ScoreManager's calculate function (even though they are the same thing). InFavour then calculates the outcome if the battle took place without playing the card. Then the card is added to the appropriate pile and InFavour then calculates the outcome of the battle if the card was played. If the battle was lost before the card was played and then battle is won after the card was played, the card is shortlisted to cards that will be InFavour of the player.

6.0.4.3 NoCounter

NoCounter contains a cardDump, which means it contains all the cards that have already been played, from either previous turns or on the current pile. NoCounter's cardDump can be updated by its cardDump(Hand[] piles) function. Everytime NoCounter's shortlist is called, it first dumps all the cards in the current piles that have already been played around. Then it checks all the cards in the player's hand. For each one of these card it checks all the cards that have already been dumped, if one of the cards dumped is a magic card with the same value, there is no counter and this card can be shortlisted. It then checks the remaining cards in its hand, if the hand contains a card with the same value and is a magic card (meaning the player has their own counter) the card is also safe to play and can be shortlisted.

6.0.4.1 SmartStretegy (pt 2)

If the turn is a HeartTurn it will not need to apply any of the strategies. If it is not a heart turn, it will check all the cards that are InFavour and have NoCounter. This means it checks each card in InFavour.getShortList() to the card in NoCounter.getShortList() and if the card is in both, the card should be played. Then from all the cards that are InFavour and have NoCounter, the Smart player will randomly choose one. The smart player will skip if there are no cards that are InFavour and have NoCounter.

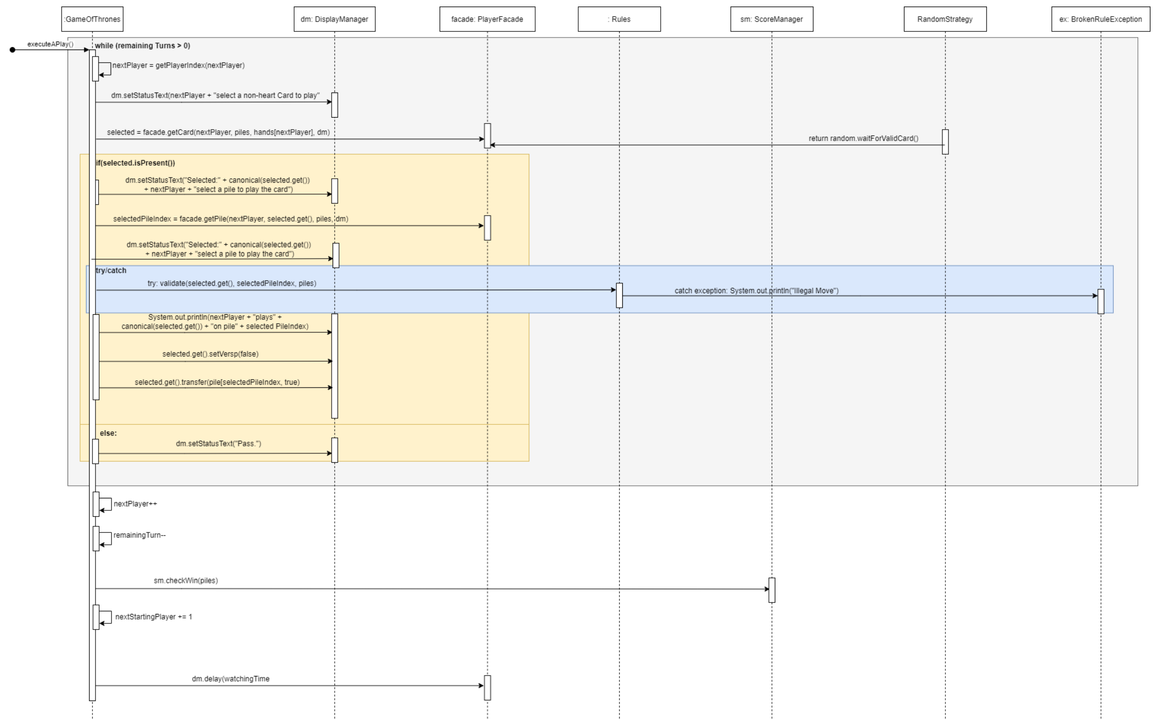
For the waitForValidPile() selection, the smart strategy will pick the same pile selection as the simple player; meaning they will only play attack and defense cards onto their pile and magic cards onto the enemy teams pile. This shows why it was better to have separate classes then to use inheritance even though there is a little bit of code reuse. Because a simple player shares some aspects to random and smart players yet smart players and random players have nothing in common. Having separate classes allows us to uniquely integrate our own strategies, meaning the future strategies do not have to rely on any other classes other than returning valid values (for card and pile selection).

7.0 Discussion: why other patterns were considered and not used

Composite pattern for the ShortListStrats was considered as within InFavour it checks for the validCards which Random, Simple and Human also do in some way. However if we used a composite pattern the main focus would be shifted to the shortlisting and not the card selection; this means that the focus is too specific on a small aspect and would add more classes that are couplied. Instead we have every class implement their own algorithm for Card and pile selection. As discussed in the report, inheritance was considered however inheritance would have broken our dependency inversion principle as strategies based on concrete strategies increased coupling. Having SmartPlayers containing both InFavour and NoCounter was also considered as both techniques are only ever used for smart players. However this would've been a bad decision as future strategies may also want to implement these techniques and this allows smart players to also add more techniques in the future if SmartPlayers were modified. Although we created a lot of new classes for the display and the score, this was important to us. The calculation of the score would be separate from the display and therefore could be modified separately in the future.

8.0 How the responsibilities are delegated

From our sequence diagram below showing how a RandomStrategy would be used pick a card and a pile:


:GameOfThrones
dm: DisplayManager
facade: PlayerFacade
: Rules
sm: ScoreManager
RandomStrategy
ex: BrokenRuleException
executeAPlay()
while (remaining Turns > 0)
nextPlayer = getPlayerIndex(nextPlayer)
dm.setStatusText(nextPlayer + "select a non-heart Card to play"
selected = facade.getCard(nextPlayer, piles, hands[nextPlayer], dm)
return random.waitForValidCard()
if(selected.isPresent())
dm.setStatusText("Selected:" + canonical(selected.get()) + nextPlayer + "select a pile to play the card")
selectedPileIndex = facade.getPile(nextPlayer, selected.get(), piles, dm)
dm.setStatusText("Selected:" + canonical(selected.get()) + nextPlayer + "select a pile to play the card")
try/catch
try: validate(selected.get(), selectedPileIndex, piles)
catch exception: System.out.println("Illegal Move")
System.out.println(nextPlayer + "plays" + canonical(selected.get()) + "on pile" + selected PileIndex)
selected.get().setVersp(false)
selected.get().transfer(pile[selectedPileIndex, true)
else:
dm.setStatusText("Pass.")
nextPlayer++
remainingTurn--
sm.checkWin(piles)
nextStartingPlayer += 1
dm.delay(watchingTime

We can see here that GameOfThrones has no direct access to the selected card and only passes in parameters to the PlayerFacade. The PlayerFacade then reads the parameters and sets the variable to the appropriate nextPlayer and calls the waitForValidCard function itself. This hides the complex subsystem of PlayerStrategy and ensures that even in the future GameOfThrones should only ever need to send information to PlayerFacade (structure does not change).

After Pile selection, GameOfThrones validates if the card played on the pile is even legal before continuing the code. This means that if pile or card selection messed up or was illegal, a message would be printed and the player's turn would be skipped.

9.0 Conclusion

In conclusion we aimed to improve the old GameOfThrones code so that in the future it can be modified by others. We hope that by separating the responsibilities, setting specific Rules that are checked by GameOfThrones before each play and having abstract PlayerStrategies and a shortListStrat interface, future changes will not be able to affect the current code and will have to follow the same style. Future code is also able to utilize aspects of the current code.