Explication Architecture

Explication des principales fonctionnalités :

Utilisons le pattern MVC pour organiser notre code source.

Le Modèle gère les données de notre appli, elle contiendra donc toutes les classes qu'il sera nécessaire de créer.

La *View* se charge de l'affichage et se contente de récupérer des variables pour savoir ce qu'elle doit afficher. Nous y implémenterons alors les méthodes d'affichage suivantes :

- printPlayerDeck pour afficher la main des joueurs
- printFirework pour afficher à chaque instant l'état d'avancement du feu d'artifice
- render pour afficher
- void printHints
- clearScreen
- int promptNbJoueurs qui retourne la saisie du nombre de joueur choisi
- GameAction promptAction(Player p)
- private GiveHint checkColorHint
- private GiveHint checkNumberHint
- private Discard checkDiscardAction
- private AddCard checkAddCard
- private void printInvalidIndex
- void printScore

Le *controller* gère la logique du code qui prend des *décisions*, il va donc contenir :

- un tableau de String rassemblant les cinq couleurs par défaut du jeu
- 8 jetons bleus par défaut
- Les instances de joueurs
-

Nous avons créé une interface nommée GameAction qui exécute une des trois actions possibles sur la partie en cours. Ces trois actions seront donc modélisées par trois classes qui implémenterons GameAction :

-Discard modélise l'action de défausser une carte

-AddCard modélise l'action d'ajouter une carte au feu d'artifice (Firework) par implémentation d'une méthode (du même nom) et de la redéfinition de l'exécution de cette action sur le feu d'artifice.

-DemandInfo modélise l'action de recevoir une information sur le n° ou la couleur d'une carte d'indexe donné, l'ensemble étant contenu dans un tableau.

Selon les règles du jeu, la pioche contient 50 cartes réparties, pour chacune des cinq couleurs, comme suit : trois cartes 1, deux cartes 2, deux cartes 3, deux cartes 4 et une carte 5.

Nous avons donc choisi de créer une classe Game qui contiendra tout le matériel pour jouer afin de représenter le concept de *partie* ainsi que la classe Card, pour représenter celui de *carte* avec tous les

attributs la caractérisant ainsi que les manières de l'utiliser (constituer une pioche « deck », mélanger un paquet de cartes).

La classe Game possèdera alors :

- les attributs :

-NB_COLORS qui initialise le nombre de couleurs à 5

-DEFAULT_DISTRIBUTION, un tableau contenant pour chaque couleur, le nombre d'exemplaires de carte par numéro, allant de 1 à 5

- les types :

- une pioche, (deck) contenu dans une *interface Queue<>* pour rendre compte que chaque sortie d'élément dans la *queue*, corresponde à un tirage de la dernière carte de la pile

- discardDeck dans un tableau de type *ArrayList*, pour représenter le tas de cartes défaussées, qui doit pouvoir apparaître (au fur et mesure des actions générées) dans l'ordre d'arrivée de cartes défaussées.

- «givenRedTokens » de type *int* qui représente le nombre de jetons rouges suite à une erreur ou une défausse par un joueur

- « maxPossibleScore » de type *int* pour compter, à la fin de la partie, le nombre de points qui donnera lieu à une mention correspondante

- « totalBlueTokens » de type int pour compter le nombre jetons bleus utilisés à la fin du jeu

- « remainingBlue » de type *int* afin de pouvoir compter le nombre de jetons bleus dans la boîte

- «currentPlayerIndex » de type int , (initialisé à zéro)

- « gameRound » de type int, pour commencer une partie le jeu

- «lastPlayerToPlay» de type int,

- «lastRoundNumber » de type int

- Les variables de Classe

- « players », déclarée *final*, mise dans un tableau, afin de rendre constant le nombre de joueur et les faire jouer dans un ordre qui suivra le sens des aiguilles d'une montre

- «firework » déclarée *final* pour l'initialisation du feu d'artifice

- Les méthodes :
  - Le constructeur pour commencer une partie, prenant en paramètres « players », « r », et « totalBlueTokens »
  - Distribuer qui distribue 4 ou 5 cartes par joeurs selon qu'ils soient au nombre de 2 ou plus
  - fish de type Card qui retourne la carte qui vient d'être tirée de la pioche
  - boolean canGiveHint qui affiche le nombre de jetons bleus restants, au cours d'une partie
  - canDiscard
  - boolean sendHint(Player player, Hint h)
  - public boolean discard(Card card)

- boolean addCardToFirework(int card)
- boolean isFinished
- boolean lastRoundEnded
- public int[] getFireworkState
- Player[] getPlayers
- int getScore
- Player findPlayer(String playerName)
- GiveHint createColorHint(String playerName, int color, int gameRound)
- Hint colorHint(Player player, Card c)
- Hint numberHint(Player player, Card c)
- GiveHint createNumberHint(String playerName, int number, int gameRound)
- void nextPlayer
- int currentGameRound
- getCurrentPlayer
- int getNbColors
- void sendColorHint(Player player, Card card)
- boolean playCard(Card card)
- void sendNumberHint(Player player, Card card)
- int getRedTokens
- int getRemainingBlue

- (Player player, int cardIndex) pour défausser puis piocher une carte tant que le jeu n'est pas fini
- public boolean addCardToFirework(Player player, int card) qui permet d'ajouter une carte au feu d'artifice si cette carte est correcte, en cas d'échec, on incrémente de un la variable « givenRedTokens » et si la pioche n'est pas vide on tire une carte
- public boolean isFinished() qui pose les conditions d'arrêt du jeu.

Concernant les joueurs, ils sont humains, au nombre de deux en phase 1, et caractérisé par un nom et un jeu de carte distribué au nombre de 5 chacun (puisqu'ils sont deux).

Nous allons donc créer une classe Player caractérisé par :

- des attributs :
  - name de type String qui sera final car constant
  - ownDeck qui sera une liste contenant des cartes de type Card, qui sera final car la distribution aléatoire de sa *main* de cartes lui est imposée
  - id de type int qui représente l'identité du joueur
  - receivedHints de type liste contenant des indices Hint, représente les indications reçues sur la main d'un joueur
- un constructeur qui prend en paramètre le «name » du joueur avec création d'une liste de carte dont la position importe, on stockera donc une instance de LinkedList dans la variable ownDeck
- une méthode dropCard pour enlever une carte de la main

Si tous les jetons bleus sont dans la boite alors on ne peut plus défausser de carte, nous allons donc traduire cela par un test de référence entre le nombre de jetons bleus dans la boite et celui au total, via l'opérateur « == »

Toujours concernant la main des joueurs, il faut pouvoir afficher les cartes situées dans la main du partenaire ainsi que le nombre de jetons bleus et rouges qui peuvent encore être utilisées.

Nous ajouterons alors, dans la classe joueur, une liste d'indices qui gardera en mémoire ceux reçus par les joueurs et qui s'afficheront à chaque passage du joueur ayant reçus ces indices.

Concernant le feu d'artifice il se complète par une suite de cartes par couleur, allant de 1 à 5, et ceci pour toutes les couleurs. Ainsi la carte jouée doit compléter le feu d'artifice soit, être supérieure de 1 à la dernière carte jouée d'une couleur, ou être une carte 1 si aucune carte de la couleur n'a encore été posée.

C'est pourquoi nous allons créer la classe Firework qui prend comme attribut son *etat.* Elle va aussi contenir entre autres

- un constructeur qui prend en paramètre le nombre de couleur que constitue le feu d'artifice
- une méthode ajouterCarte de type booléenne qui permettra d'ajouter une carte au feu d'artifice s'il elle est compatible avec l'état du feu sinon celle-ci est jeter est défaussée et on perd un jeton rouge
- une méthode currentScore qui compte les points en fin de la partie, en fonctions de la composition du feu d'artifice, (son affichage restera implémenté dans la View).
- Une méthode state qui rend compte de la constitution du firework
- getNbColors

Il faut aussi que le joueur puisse indiquer quelle information (couleur ou valeur) il souhaite donner à l'un de ses partenaires. Nous allons donc créer une classe mère Hint qui sera héritée par trois classes filles : ColorHint, NumberHint et GiveHint qui implémente l'interface GameAction.