Hanabi

Documentation développeur

Jeu codé en JAVA
Par

SAADAOUI Oussama
LEBON Nicolas

SOMMAIRE

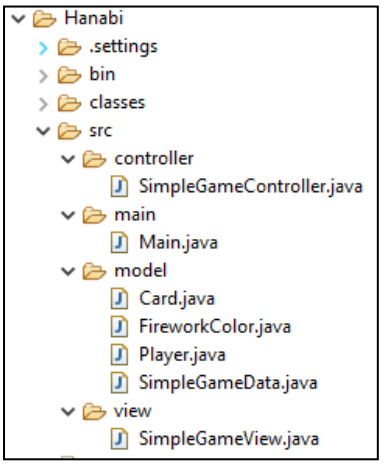
1. Introduction

Ce programme a été codé par deux étudiants de première année d'école d'ingénieur. Etant la formation INFORMATIQUE de l'ESIPE, nous avons du coder ce jeu à l'occasion d'un projet de programmation JAVA. Le but était de recréer le jeu de société Hanabi sortie en 2010 par le français Antoine Bauza.
Dans ce manuel, nous allons expliquer nos choix techniques, notre architecture, nos améliorations, …

2. Architecture

a. Organisation

Nous utilisons un GIT pour faciliter notre travail en coopération.
Notre projet est organisé de la manière suivante :



Controller :

Dans le répertoire *controller* nous avons *SimpleGameController.java*, qui est la pièce maitresse dans notre jeu. C'est lui qui va orchestrer les différentes actions de notre jeu, de l'initialisation du jeu, à la gestion des tours, en passant par la création des joueurs.

Main :

Le *main.java* va être exécuté et va lancer notre SimpleGameController.java pour démarrer et initialiser notre jeu.

Model :

Le répertoire *Model* contient toutes les classes essentielles pour faire fonctionner notre jeu. Nous y retrouvons *Card.java* qui est la classe qui définit une carte du jeu, ou encore *SimpleGameData.java* qui va se charger de l'initialisation du jeu, notamment du paquet de cartes et de la constitution des main des joueurs, mais aussi de la gestion des jetons d'indice, le score, … C'est l'une des pièces principale du jeu.

View :

En ce qui concerne la classe SimpleGameView.java nous n'avons pas su l'implémenter pour la phase 1 et 2, mais nous nous en serviront pour la phase 3.

b. Organisation

Dans cette partie nous allons voir les principales structures utilisées pour permettre à notre jeu de fonctionner correctement.

- **Card** (carte de jeu) Type → `Card(FireworkColor color, int valeur)`

- **DiscardZone** (défausse) Type → `Map<FireworkColor, ArrayList<Integer>`

Nous avons décidé d'utiliser un ArrayList<integer> car nous voulons garder et afficher toutes les cartes de valeurs et couleurs différentes qui ont été jetées, afin de faire un « historique ».

- **Field** (terrain de jeu) Type → `Map<FireworkColor, <Integer>>`

Nous avons décidé d'utiliser un integer simple car nous ne voulons afficher que la valeur de la carte de couleur FireworkColor qui a été la dernière à être posée par un joueur.

- **FireworkColor** (couleur feux) Type → `enum FireworkColor{white,blue,yellow,red,green}`

Nous avons choisit le type *enum* car cela évite qu'un utilisateur rentre lui-même la couleur, et cela évite notamment les noms de couleurs mal écrits.

- **Player** (joueur) Type → `Player(String name, SimpleGameData game, int handSize)`

Pour créer un joueur, nous avons besoin de son nom, de passer en argument les données de la partie pour que son main puisse être créé etc, et pour finir, la taille de sa main qui varie en fonction du nombre de joueurs.

b. Algorithme principal

Dans notre jeu l'algorithme principal va à chaque tour vérifier que le deck de carte n'est pas à 0, si il ne l'est pas alors nous pouvons pour chaque joueur appeler la méthode *turn* . Nous continuons d'appeler *turn*, tant que l'algorithme ne détecte pas que les trois jetons rouges ne sont pas perdus ou que tous les tas sur le tapis de jeu ne sont pas complets.

Nous allons parler de l'algorithme qui va nous permettre à chaque tour de jeu, de pouvoir effectuer les actions que le joueur souhaite faire. La méthode ***turn*** qui se situe dans ***controller->SimpleGameController.java.*** A chaque tour, nous affichons les informations du joueur c'est-à-dire, son nom pour qu'il remarque que c'est à son tour de jouer, nous affichons donc sa main, mais aussi des informations plus générales comme le nombre de jetons rouges et bleus, le nombre de carte restantes, le tapis de jeu et pour finir les cartes qui sont dans la défausse.
La seconde partie de ***turn*** prends en compte l'action du joueur et grâce à un switch qui va appeler les bonnes méthodes, l'action que le joueur souhaite, va être réalisée.
Pour finir, nous n'oublions pas de « clean » la console pour permettre une meilleure visibilité du jeu.

De retour dans notre boucle principale, nous allons vérifier le nombre de cartes dans le deck, puis nous vérifions allons donc afficher un message sur la console pour prévenir les joueurs que le jeu est finit. Un message différent s'affiche selon le nombre de points de l'équipe !

Pour résumer, une partie se termine si le tapis de jeu est complet, si le deck est vide ou alors si les trois jetons rouges sont perdus par l'équipe.

Victoire :

Si le tapis de jeu est complet ou que le deck est vide, alors nous allons calculer les points de l'équipe. Ces points correspondent à la somme de toutes les cartes visible sur le tapis (FIELD).
En fonction du score, un message différent s'affiche dans la console pour féliciter les joueurs sur leur victoire.

Défaite :

Si les joueurs ont perdus les trois jetons rouges, alors c'est une défaite, et nous affichons un message sur la console.

3. Evolutions

Il faut garder à l'esprit que notre jeu n'est qu'une version béta en ligne de commande. Des évolutions sont donc déjà prévus ou des méthodes sont vouées à potentiellement disparaitre.

Indice :

-Choisir la carte avant la valeur ou la couleur.

Liste évolutive.