RAPPORT

Programmation orienté objet Java

Introduction :

Ce projet consiste à programmer le jeu de société : **hanabi**.

Les règles sont expliquées dans « user.pdf ».

Le projet :

Phase 1 : Pour commencer nous avons créé les classes correspondant aux éléments du jeu c'est-à-dire :

- Une classe Card avec 2 champs (final car une carte ne change pas au cours du jeu) :
  - Un champ « color » pour la couleur de la carte.
  - Un champ « num » pour le numéro de la carte
- Une classe Deck avec 2 champs et 2 constantes :
  - Un champ « deck » pour un tableau de 50 cartes soit le deck en lui-même
  - Un champ « nb_card », un entier, pour gérer le nombre de cartes restantes dans le deck.
  - Les deux constantes entières NB_MAX et SWAP servent respectivement à s'assurer que le nombre de carte ne dépasse pas 50 quel que soit le nombre de cartes entrées et à définir combien d'échange de cartes faire pour le mélange du deck.
- Une classe Hand avec 4 champs :
  - Un champ « hand » pour un tableau de 4 ou 5 cartes pour les mains des joueurs

    - Un champ « info_color » pour un tableau d'entiers pour gérer les informations des indices sur les couleurs
    - Un champ « info_num » comme pour « info_color » mais pour les numéros
    - Un champ « nb-cards », un entier, pour gérer le nombre de cartes en mains
- Une classe « Color » pour un enum pour gérer les couleurs
- Une classe « SimpleGameControler » pour la boucle qui permet de lancer et de jouer au jeu
- Une classe « SimpleGameData » pour réunir toutes les données du jeu et mettre les méthodes pour gérer le jeu, elle possède 6 champs et deux constantes :
    - Un champ « deck » pour récupérer un deck construit grâce à la classe Deck
    - Un champ « discarded_pile » pour un deck vide afin de faire une défausse
    - Un champ « hands » pour un tableau de mains de taille égale au nombre de joueurs où chaque main est construite grâce à la classe Hand
    - Un champ « board » un tableau de 5 cartes pour gérer le plateau du jeu et les scores de fin de jeu
    - Un champ « clue », un entier, pour gérer le nombre d'indices
    - Un champ « life », un entier, pour gérer le nombre de vies
    - Deux constantes entières MAX_PLAYER et MIN_PLAYER pour gérer les limites de joueurs

Une fois tout cela mis en place, nous avons programmé le jeu en lui-même, nous avions commencé par gérer l'affichage des mains, du deck, du plateau et des cartes avec les « toString » et dans

SimpleGameData nous avons donc repris les affichages et rajouté les informations qui manquaient comme le plateau ou les vies dans une méthode « printPlayer » nommée ainsi car la grosse partie de cette méthode permet d'afficher les mains des joueurs.

Nous avons fait une petite méthode afin de mélanger le deck que nous avons appelé « mix ».

Nous avons ensuite géré le fait de jouer ou de défausser une carte avec respectivement les méthodes « playCard » et « throwCard », utilisant des méthodes venant d'autres classes comme la méthode « discard » permettant de mettre la carte de la main désirée (via son emplacement dans le tableau de cartes qu'est la main du joueur) dans la défausse.

Nous avons également géré le fait de piocher une carte grâce à la méthode « pick » qui prend le deck et l'emplacement de la carte dans la main du joueur afin de remplacer la carte qui était jouée par la première carte du deck et de réduire ce dernier d'une carte.

Pour permettre au joueur d'entrer ses actions nous avons utilisé un BufferReader qui permet de lire le premier caractère de la ligne grâce à sa méthode « read » ou de lire une ligne entière grâce à sa méthode « readline ». Ainsi nous avons créé la méthode « readInt » qui permet de lire le premier entier de la ligne et de le renvoyer (même si un caractère est entré, celui-ci est lu comme un entier). Nous avons également créé « readJump » afin de pouvoir avoir une phase ou il suffit d'appuyer sur entrée pour passer au tour du joueur suivant.

Afin que le joueur puisse savoir quel action faire et choisir la carte à jouer nous avons fait une méthode « cardChoice » qui demande un nombre pour identifier quelle est la carte à jouer (le nombre demandé est l'emplacement de la carte dans la main du joueur) et

une méthode « actionChoice » pour demander quel action faire (encore une fois on demande un entier : 1 pour jouer une carte ou 2 pour défausser une carte). A partir de là nous avons convenu que nous n'utiliserons que des entier pour que le joueur choisisse parmi les choix possibles qui lui sont donnés.

Pour finir, cette première phase il suffit de faire la boucle pour gérer le jeu, nous avons donc inclus dans cette boucle les actions possibles et la gestion des contraintes de fin de jeu. Pour cela nous avons fait deux méthodes : « mainLoopTerminal » et « lastTurn » afin de gérer respectivement la boucle de jeu principale et enfin la gestion du dernier tour de jeu si le deck est vide. Le tout utilisé dans la méthode « mainTerminal » qui permettra de différencier les méthodes utilisées par le jeu sous forme texte dans le terminal et le jeu sous forme graphique.

Une fois tout cela combiné la première partie était ainsi terminée.

Phase 2 : A partir de là, nous avons modifié certains affichages afin de pouvoir utiliser les indices donnés par les autres joueurs, il y a donc dans la classe « Hand » le « toString » qui sert maintenant à afficher la main du joueur en fonction des indices qu'il possède et la méthode « printHand » pour afficher la main complète d'un autre joueur.

Puis nous avons inclus plusieures fonctions de choix comme « nbPlayer », « player Choice », « typeOfClue » et « clueChoice » afin de pouvoir, respectivement, choisir combien de joueurs pour la partie (et avec cela combien de cartes pour chaque joueur) puis pour les indices de choisir à quel joueur donner quel type d'indice et quel couleur ou numéro donner. Enfin une méthode « giveClueTerminal » afin de réunir tout ce qui fait l'action "donner un indice".

Nous avons de plus géré l'affichage des indices et de la défausse dans « printPlayer» en plus par rapport à la version précédente afin d'avoir toutes les informations nécessaires à la compréhension du jeu.

A partir de là, le jeu est jouable dans l'ensemble de ces règles.

Phase 3 : Pour l'affichage graphique il nous a suffi de créer des méthodes pour que le jeu se dessine en fonction de son état. Pour cela nous avons créé une méthode « drawGame » qui permettra l'affichage complet du jeu avec toutes les informations nécessaires comme l'action en cours, le nombre de joueurs et le joueur qui joue son tour ou encore les dimensions de l'écran de jeu (qui sont les dimensions de l'écran de l'utilisateur).

Nous avons ensuite fait beaucoup de fonction d'affichage différentes afin de séparer les différentes choses à dessiner. Le jeu commence par demander combien de joueurs on veut avec « drawNbPlayer ». Ensuite il a fallu dessiner le plateau de jeu où les cartes sont posées avec « drawBoard » qui affiche 0 sur les cartes si aucune carte de la couleur n'est sur le plateau, le nombre de vies et d'indices restants avec « drawLife » et « drawClue », le bouton de retour pour recommencer son tour avec « drawBack », le deck avec « drawDeck » qui affiche le nombre cartes restantes dans le deck ainsi que la défausse avec « drawDiscard » qui permet l'affichage cette défausse et qui affiche les cartes une par une lorsque l'on clique dessus lors du choix de l'action à faire. Pour finir il faut afficher les mains des joueurs avec toutes les informations nécessaires c'est-à-dire les cartes des joueurs avec « drawCard » pour les cartes des autres joueurs avec les informations qu'ils ont déjà eus, il est écrit en petit sur la carte si le joueur en connait la couleur et/ou le numéro, et « drawCardUnknown » pour les cartes du joueur qui joue son tour et qui verra les cartes en fonctions des informations dont il dispose. Les

mains des joueurs sont dessinées grâce aux méthodes « drawHand » et « drawHandUnknown », elles appellent donc respectivement « drawCard » et « drawCardUnknown » pour afficher les cartes des mains des joueurs en fonction de celui dont il est question. Chaque main s'accompagne aussi du numéro du joueur à qui appartient les cartes avec « drawName » et qu'elle est la 1ère, 2ème, 3ème, 4ème et (s'il y a) la 5ème carte du joueur avec « drawNameCard » pour faciliter le choix de la cartes à jouer. Pour finir, il reste à dessiner le choix des actions nous avons donc fait « drawChoiceColor » pour le choix de la couleur, « drawChoiceNum » pour le choix d'un numéro, celui-ci s'adapte à la situation en fonction de si on demande une carte, un numéro ou un joueur, « drawChoiceAction » pour choisir une action parmi les 3 disponibles et « drawChoiceBetween » pour choisir entre Couleur ou Numéro lorsque l'on veut donner un indice. Elles sont alors toutes appelées dans « drawAction » qui dessine d'abord une boite grise pour bien repérer la zone servant à afficher les actions possibles et elle écrit aussi un message afin de bien comprendre ce qui est demandé (les messages sont gérés grâce à un enum « Message » afin d'éviter de compliquer la méthode).

Les méthodes ainsi créées sont pour la plupart des fonctions qui permettent l'affichage des éléments du jeu, néanmoins il a été nécessaire de faire deux autres types de méthodes : les méthodes de détection pour détecter les cliques de la souris de l'utilisateur et les méthodes d'attentes qui permettent d'attendre une action particulière de l'utilisateur.

Il y a donc 4 méthodes de détection avec « detectBetween » , « detectAction » et « detect » qui permettent de renvoyer le choix du joueur en fonction l'action en cours. « Detect » est la plus utilisée car elle permet la détection des décisions les plus courantes dans le jeu et ce par le faite que la décision pour le choix d'un nombre, le choix

d'une carte, le choix d'une couleur et le choix d'un joueur se fait de la même façon. De plus « detectAction » permet de détecter si le joueur clique sur la défausse pour la regarder, ainsi on ne peut regarder la défausse qu'en début de tour lorsque l'on doit choisir son action. Le tout réuni dans une méthode « clickChoice » qui permet de renvoyer les choix du joueur en fonction de l'action en cours et donc de la détection utilisée.

Il y a 3 méthodes d'attentes : avec « waitPoint » qui permet d'attendre un clic de la souris afin d'avoir des coordonnées à analyser ; avec « waitNextPlayer » qui permet d'afficher une transition entre les tours de chaque joueur si on passe au joueur suivant ou d'afficher si on recommence notre tour car on n'a pas cliqué où l'on voulait ou encore de dire que l'on arrive au dernier tour s'il n'y a plus de carte dans le deck . Enfin, il y a « waitScore » qui permet d'afficher le score en fin de partie ou d'afficher le fait que la partie est perdue s'il n' y a plus de vie ; ici les messages sont gérés grâce à « drawLose » et « drawScore ».

Phase 4 : Nous avons malheureusement manqué de temps nous n'avons pas pu faire d'amélioration supplémentaire.

CONCLUSION

Nous n'avons pas eu de difficultés particulières.

ANNEXES

Pour les informations sur BufferReader, IOException, InputStreamReader et les classes graphics venant « java.awt » la javadoc contient toutes les informations nécessaires. De plus, la bibliothèque fournie « zen5 » et une doc attitrée pour chacune de ses classes.