Manuel de développement

Programmation Orientée Objet Java

Hanabi

UNIVERSITE PARIS-EST MARNE-LA-VALLEE

25 janvier 2020

Créé par : SAVANE Kévin & BERGERON Youssef

Table des matières

I. Introduction : Hanabi

Ce projet, consiste à créer le jeu de carte Hanabi. Il s'agit d'un jeu de société stratégique où plusieurs joueurs doivent collaborer pour réaliser les plus beaux feux d'artifice possibles.

Ce projet est à réaliser en 4 phases :

- Une première phase où seul le principe de jouer des cartes et des jetons rouges est présent. Dans cette phase initiale, les joueurs ont encore la possibilité de voir leurs cartes, et de ce fait, les indices ne sont pas encore présents. Pour cette première partie, un affichage basique en ligne de commande sera mis en place ;
- La seconde phase ajoute les éléments manquants de Hanabi. A l'issue de cette étape, le jeu sera présenté dans une version complète en terminal. Il faudra alors mettre en place un affichage en ligne de commande plus poussé et élégant du plateau de jeu ;
- La troisième phase correspond à la création d'un affichage graphique du jeu, grâce à la librairie zen. Naturellement, l'affichage devra être aussi bien élégant que lisible et intuitif ;
- Enfin, la dernière phase correspond à une phase d'amélioration. Celle-ci pourra rajouter une IA et une gestion de fichier de paramètres ;

Ce projet va alors être divisé en quatre parties :

- La conception d'une architecture viable pour le projet ;
- Effectuer une interface console agréable ;
- Concevoir l'interface graphique ;
- Tenter de faire les améliorations proposées.

II. Partie 1 – Architecture du projet

La partie 1 consiste en une compréhension globale du sujet. Après avoir lu l'entièreté du sujet, un découpage se profile.

Pour commencer, la présence du modèle MVC implique la création des classes **SimpleGameData** (Gestion des données du jeu), **SimpleGameView** (Gestion de l'affichage) et **SimpleGameController** (Gestion de la boucle de jeu et des événements utilisateur).

De plus, un jeu de carte contient nécessairement une classe **Card**. Cette classe introduit alors les composants suivants :

- Un enum de **CardColor** ;
- Une classe **Number**.

Cette classe **Number** nous permet de vérifier l'indice d'une carte saisi par le joueur.

Ensuite, sachant qu'une pioche, une défausse ou même la main d'un joueur sont des paquets de cartes, une classe **Pack** verra le jour. Celle-ci contiendra un tableau de cartes et une méthode *sizeOf* donnant la longueur de ce tableau.

Comme indiqué, il faudra créer une classe pour la pioche : la classe **Deck**. Comme évoqué précédemment, elle héritera de le classe **Pack** et implémentera une méthode *shuffle* qui mélangera la pioche et une méthode *draw* qui permettra de piocher une carte.

Ensuite la défausse étant elle aussi un paquet de cartes, notre classe **Discards** héritera elle aussi de **Pack** et ajoutera une méthode *discard* pour défausser une carte et une méthode *lastDiscard* donnant la dernière carte défaussée.

Etant donné qu'un joueur possède une main, un champ *hand* de type **Pack** a été ajouté à une classe **Player**. Ici, cette classe n'hérite pas de **Pack**. En effet, un joueur n'est pas un jeu de carte. Il possède simplement une main.

De plus, un tableau de **Hint** a aussi été ajouté à cette classe permettant de stocker les indices obtenus par le joueur. Contrairement à la classe **Card**, la classe **Hint** peut avoir des champs null.

Nous avons aussi créé une classe **Coin** représentant les jetons possédant alors un champ de type **CardColor** (Blue ou Red) et un entier représentant le nombre de jetons disponibles.

Enfin, une classe **GameBoard** représentant le plateau de jeu a été créée. Celle-ci contiendra le plateau, les jetons bleu et rouge, la pioche et la défausse. Cette classe contiendra une méthode *draw* pour piocher une carte, une méthode *put* pour poser une carte et une méthode *discard* qui permettra de se défausser d'une carte.

Afin de centraliser les choix des joueurs, nous avons décidé de créer trois enum :

- **CardColor**, déjà évoqué précédemment ;
- **HintType**, regroupant, comme son nom l'indique, les types d'indice possibles ;
- **PlayerAction**, regroupant, comme son nom l'indique, les actions possibles.

III. Partie 2 – L'affichage Console

Dans un premier temps, nous allons décrire les actions du joueur dans la console :

Au commencement de la partie, le programme demande à l'utilisateur de saisir le nombre de joueur.

Ensuite, une fois que la partie a commencé, le joueur peut effectuer 3 actions différentes :

- Poser une carte (Play – p) ;

- Jeter une carte (Discard – d) ;
- Donner un indice (Hint – h).

Dans les deux premiers cas, le joueur devra simplement indiquer l'index de la carte qu'il souhaite jouer.

Dans le troisième cas, il devra indiquer le joueur auquel il souhaite donner l'indice, son type (Color ou Number) et suivant le choix précédent, la couleur (White, Red, Green, Blue, Yellow) ou la valeur (1, 2, 3, 4, 5) correspondante.

Ces interactions font l'objet d'une reprise sur erreur dans le cas où l'utilisateur entre une valeur incorrecte. Le programme ne s'arrêtera pas mais continuera en reprenant là où la saisie échouée.

Dans un second temps, nous allons décrire l'affichage console qui se découpe en plusieurs parties :

La première ligne indique la dernière carte défaussée. Les deux lignes suivantes indique le nombre restant de jetons bleu et rouge est affiché.

Ensuite, un affichage du plateau de jeu avec les cartes ayant été posées est proposé. On retrouve dans l'ordre de gauche à droite les couleurs White, Red, Blue, Yellow et Green, et de haut en bas les valeurs 1, 2, 3, 4 et 5.

La ligne suivante affiche les indices dont le joueur dispose.

Pour finir, les dernières lignes de l'affichage correspondent aux cartes des autres joueurs. Etant donné que les indices sont donnés sur les cartes, ils sont affichés de cette manière : **? ?** ; la première valeur représentant la valeur de la carte, et la seconde valeur représentant la couleur de la carte. Comme précisé précédemment, lorsqu'un joueur ne dispose pas d'indice sur certaines cartes, la valeur correspondante est *null* et est affiché comme un point d'interrogation.

IV. Partie 3 – L'affichage Graphique

L'affichage graphique que nous avons mis en place se base sur des images et non de simples formes. Ces images représentant les cartes et les jetons ont été entièrement réalisées par nous, à l'aide de Photoshop et Clip Studio Paint. Elles permettent de rendre notre projet un peu plus unique par rapport aux autres.

Afin de mettre en place cet affichage graphique, il a été nécessaire de faire de la surcharge de méthode. En effet, les méthodes d'affichage graphique ont été décomposées de la même manière que les méthodes d'affichage console, avec d'autres méthodes intermédiaires si nécessaire. Par conséquent, nous n'avons pas utilisé d'interfaces ni de classes abstraites, car elles se basent sur le sous-typage et le polymorphisme, alors que nous utilisions de la surcharge de méthodes. Nous aurions aussi pu séparer la classe **SimpleGameView** en deux classes, l'une regroupant les méthodes d'affichage console et l'autre regroupant les méthodes d'affichage graphique. Cependant, il a été

indiqué dans le sujet que la classe **SimpleGameView** centralisait déjà les méthodes d'affichage. En plus de cela, comme dit précédemment, nous ne pouvons pas utiliser d'interfaces ou de classes abstraites pour regrouper les méthodes communes d'affichage graphique et console.

Le système d'affichage des éléments du jeu repose sur le même principe celui d'affichage console. La différence principale réside dans l'utilisation de la bibliothèque Zen5, ainsi que de la classe **Graphics2D**.

Etant un affichage graphique, il a été nécessaire de mettre en place un moyen de récupérer les actions des joueurs. Pour ce faire, nous avons récupéré les clics de souris, comme montré dans les exemples donnés. Il nous fallait ensuite localiser le clic et vérifier si sa position correspondait à un élément cliquable à l'aide de boucle for et de structures conditionnelles if / else if / else.

Il est évident que certains joueurs changent d'idée durant leur tour. Il était donc indispensable d'introduire un bouton *Back* permettant d'annuler l'action courante.

Rentrons désormais plus en détail dans le développement de l'affichage graphique :

Tout d'abord, afin de sauvegarder les images ainsi que d'éviter la duplication de variable, nous avons créé une nouvelle classe **GraphicsData** regroupant toutes ces données. Les champs de cette classe étant tous static, nous nous devions de les initialiser correctement, afin d'éviter quelconques surprises. La grande majorité des champs de cette classe sont constitués de variables définissant la taille ou la position des éléments à afficher à l'écran. Ces champs sont initialisés dès leur déclaration. Leurs valeurs ont été choisies par rapport à un écran de résolution 1920x1080. Comme nous ne travaillons pas sur le même type d'écran, il a été nécessaire d'adapter ces valeurs à l'aide d'un ratio de hauteur et de largeur. Ces ratios sont définis static dans la classe **SimpleGameView**, et sont présents dans une grande majorité des méthodes graphiques de cette classe.

Ensuite, pour initialiser les images, nous avons fait appel au bloc d'initialisation static qui appelle les différentes méthodes d'initialisation des images. Comme les appels à la méthode *ImageIO.read* ne garantissent pas une ouverture d'image réussie, ils ont tous été entourés d'un bloc try / catch.

Concernant l'affichage même du jeu, comme indiqué précédemment, nous faisons appel à des méthodes surchargées. De manière générale, au lieu de faire appel à la méthode *println*, nous appelons les méthodes *drawImage* ou *drawString* de la classe **Graphics**, en fonction de nos besoins. Bien évidemment, il a été nécessaire de faire des calculs intermédiaires afin de positionner correctement les images.

L'affichage de texte a été un peu plus compliqué à mettre correctement en place. Nous aurions très bien pu les afficher centrée sur leur boîte en tâtonnant, mais ce n'est pas une bonne approche. Il nous a donc fallu récupérer les informations de taille sur les polices utilisées grâce à la méthode *getFontMetrics*, puis appliquer une formule afin de trouver les coordonnées de positionnement des textes.

Le plus compliqué aura très certainement été le processus de conversion des clics de souris en action. Les méthodes *recoverCardsLocation* et *recoverPlayerNumberLocation* de la classe

SimpleGameData permettent, comme leur nom l'indique, de récupérer les clics de souris sur les cartes ou sur le choix du nombre de joueur en début de partie. Les cartes et les boutons de choix du nombre de joueur étant toutes sur la même colonne ou sur la même ligne, une simple itération suffit à récupérer les informations requises pour la suite du jeu.

En revanche, la méthode *recoverButtonsLocationOnButton* nous permettant de récupérer le clic de souris sur les boutons d'actions est bien plus complexe, étant donné que suivant le nombre de boutons ou l'action en cours, des conditions de calcul se présentent.

Dans le cas du nombre de boutons variant, s'il est supérieur à 3, les boutons s'affichent sur deux lignes. Nous devions alors être capable de récupérer correctement les clics sur deux lignes. Cette première partie s'est avérée relativement simple.

Cependant, dans le cas où le joueur actuel doit choisir à quel joueur il souhaite donner un indice, la récupération et le retour de l'indice du joueur s'est avéré être beaucoup plus compliqué à mettre en place. En effet, comme pour chaque état, les boutons s'affichent de la même manière à l'aide d'une itération simple, mais le retour peut varier. Le programme doit être capable d'attribuer à chaque bouton la valeur qui lui correspond. Or, cette valeur est modifiée suivant quel joueur est actuellement en train de jouer. Il nous a alors fallu rajouter des conditions pour incrémenter correctement des variables afin de sauter la valeur et le bouton qui devrait correspondre au joueur actuel.

En sortie de ces méthodes, de simples tests et conversions permettent de déterminer le choix effectué par le joueur. Les méthodes de conversion sont utilisées uniquement pour le choix de l'action, du type d'indice et de la couleur de l'indice.

Concernant les indices, nous avons décidé de les afficher explicitement sur les cartes. Nous avons utilisé des images représentant soit la couleur de l'indice, soit la valeur de l'indice. Ainsi, au lieu d'afficher une simple pastille ou autre méthode d'affichage banal, ce sont les cartes elles-mêmes qui changent. Cela implique que lorsqu'un joueur connaît toutes les informations sur une carte, il sait naturellement quelle carte il a en main. Dans ce cas-ci, nous avons décidé d'afficher directement la carte complète du joueur.

Enfin, lorsque la partie prend fin, le score s'affiche à l'écran avec une phrase personnalisée commentant le spectacle réalisé par les artificiers. Comme précisément précédemment, ces deux textes sont affichés simplement grâce à la méthode *drawString*.

V. Partie 4 – Améliorations

Malheureusement, nous n'avons pas traité les améliorations suggérées dans le sujet. Cependant, une réflexion à tout de même été menée.

Pour l'amélioration concernant le fichier de paramètres, il nous aurait fallu maitriser l'ouverture et la fermeture des fichiers. Les paramètres auraient pu être écrit comme dans un fichier de paramètres classiques d'un jeu, le nom du paramètre, suivi d'un égal et de la valeur du paramètre. Cependant, au vu de l'affichage graphique que nous proposons, il n'aurait pas été possible

d'implémenter le choix du nombre de couleur et la plage de valeurs, étant donné que nous utilisions des images. Bien sûr, une nouvelle classe **Options** aurait été la bienvenue.

Concernant l'amélioration IA, il nous aurait fallu une classe **IA** héritant de la classe **Player**. Le principe se baserait sur une imbrication de structure if / else if / else, agrémentée de calculs et conditions menant aux décisions les plus utiles possibles.

VI. Retour de la soutenance β

Comme demandé lors de la soutenance β, nos méthodes *toString* ont toutes été rassemblées dans la classe *SimpleGameView* afin de centraliser l'intégralité de l'affichage console en une seule et même classe.

De plus, nous avons regroupé nos classes dans des packages selon leur fonction :

- fr.umlv.hanabi.card : Regroupe les classes gérant les cartes ;
- fr.umlv.hanabi.coin : Regroupe les classes gérant les jetons ;
- fr.umlv.hanabi.gameBoard : Regroupe les classes gérant le plateau de jeu ;
- fr.umlv.hanabi.graphics : Regroupe les classes gérant les informations graphiques ;
- fr.umlv.hanabi.hint : Regroupe les classes gérant les indices ;
- fr.umlv.hanabi.mvc : Regroupe les classes gérant le modèle MVC ;
- fr.umlv.hanabi.pack : Regroupe les classes gérant les paquets de cartes ;
- fr.umlv.hanabi.player : Regroupe les classes gérant les joueurs.

VII. Anecdotes

"Hanabi" (花火) est un mot japonais signifiant "Feu d'artifice". Vous pouvez voir sur les cartes des Kanji représentant la signification des couleurs aux Japon :

- Pureté (精) pour le blanc ;
- Cerise (桜) pour le rouge (le fameux cherry blossom) ;
- Loyauté (忠) pour le bleu ;
- Soleil (陽) pour le jaune ;
- Energie (勢) pour le vert.

Enfin, les pièces ont le mot "花火" écrit dessus. Comme indiqué, ce mot se lie "Hanabi".

VIII. Conclusion

Ainsi, ce projet s'avéra être très formateur.

En effet, il était nécessaire de bien réfléchir à une architecture pouvant convenir et essayer de la rendre la plus efficace possible.

Ce projet nous a aussi permis de gagner de l'expérience en modularisation de projet et dans notre approche (l'implémentation peut différer fortement d'une personne à l'autre).

Enfin, un projet sans indication nous a permis de retrouver et améliorer notre autonomie et notre logique de programmation.