BERNIER Valentin
VILAYVANH Mickaël

Manuel du développeur

Sommaire

BERNIER Valentin
VILAYVANH Mickaël

I. Introduction

Pour réaliser ce jeu du Patchwork, nous avons défini **2 records et 7 classes par rapport a la mutabilité des champs leur visibilité.** Ces deux records servent d'objets primaire pour les objets principaux du jeu. Seule la classe Main ne possède pas de choix d'implantation spécifique.

II. Choix de l'architecture

Le record Coordinate

Lors de l'écriture de la classe QuiltBoard, on souhaitait d'abord avoir un **tableau de Integer contenant deux éléments** pour représenter un couple de coordonnée. Pour rendre cela un peu plus propre, nous avons crée ce record qui a pour vocation de **rendre la compréhension et les manipulation plus simple** que d'accéder tout le temps a deux éléments d'un tableau.

Le record Patch

Nous avons choisi de représenter la forme de chaque patch dans un tableau a deux dimensions de booléen car il a pour vocation d'être représente sur un plateau a deux dimensions et cela facilite donc la pose du patch ainsi que sa rotation. Les autres champs sont les informations sur le patch crée.
Il y a une **méthode static createPatch** qui renvoie un objet **Patch** a partir d'une description donnée en **String**. Nous savions déjà a cette étape que nous souhaitions lire un fichier dont le format de patch serait défini par des caractères. Cela a donc **facilité l'insertion des patches une liste de Patch car nous comptions procéder par lecture de fichier.**

La classe PatchesList

Une patcheslist (liste de patches) est **une collection de patches.** Nous avons donc décidé de l'implanter avec une **ArrayList** de **Patch**.

Pour pouvoir afficher un objet **PatchesList**, nous avons écrit plusieurs méthodes intermédiaires utilisées dans le toString afin l'assouplir et le rendre lisible.

La classe TimeBoard

Le timeboard est une classe car même si tous ses champs sont final, on ne souhaite pas rendre explicitement visible les positions des cases spéciales et seule sa taille doit être visible. N'étant en faite qu'un ligne droite si on se représente fictivement le plateau, nous avons décidé de ne pas avoir de tableau pour le représenter car cela n'est finalement pas nécessaire. En effet, seule la taille, la position des éléments et leurs nombres était nécessaire pour représenter le timeboard.

Un joueur n'aura jamais besoin de savoir qu'il parcourt une case vierge, seules les cases spéciales ont une interaction direct avec lui. On conserve donc la position de ses cases spéciales dans des HashMap afin de garder une recherche en temps rapide.

La classe QuiltBoard

Nous avons choisi de représenter le board par un **LinkedHashMap** dont les clés sont les coordonnées de l'objet **Coordinate** et la valeur est booléenne, false pour une case vide, true pour une case pleine, et un **HashMap** pour conserver les positions des patches présents sur le board, gardant donc une collection des patches possédés cela servira notamment dans l'implantation de la version graphique.

Un **LinkedHashMap** a été choisi car l'ordre d'insertion était important **pour la représentation de la grille sur le terminal.** En particulier, un **HashMap** ici servait a rendre l'insertion et **la recherche d'un élément plus rapide**. On profite ainsi pleinement de la fonctionnalité de pouvoir remplacer la valeur a une clé spécifique et facilite la lecture du code et sa compréhension plutôt qu'un tableau a deux dimension.

La classe UserInterfaceTerminal

Cette classe permettait de **traduire toutes les données du jeu dans le terminal**, que cela soit les entrées de l'utilisateur ou bien les différents affichages. Il y a un seul champ prive non final qui est un objet **Scanner** permettant de lire les entrées. Cela a été choisi pour n'avoir qu'un scanner utilise par toutes ses méthodes **sans avoir a en définir un en dehors**, qui n'aurait pas de sens car la **seule concernée est cette classe**.

Nous avons ajouté **une fonctionnalité qui permet de jeter une pièce.** Cela pourrait survenir si le plateau personnel est suffisamment rempli au point de ne plus pouvoir poser une pièce sélectionnée. Cela ajoute aussi un autre aspect stratégique au jeu (par exemple, prévoir les prises de l'adversaire pour l'empêcher de réaliser son plan).

Nous comptons intégrer une interface UserInterface qui serait implanter pour la version terminal et la version graphique qui facilitera l'implantation de la version graphique sans faire subir de modification aux parties déjà intégrées.

La classe Player

Étant donne que le plateau du jeu n'est en aucun cas un tableau de case mais ne possède que sa taille pour se définir sur sa longueur et le nombre de case, nous avons pris le choix de représenter l'emplacement du joueur par une **valeur numérique de 0 jusqu'à la taille** du plateau. Nous pouvons ainsi savoir ou se trouve le joueur sur le plateau et les différentes cases spéciales qu'il a parcouru sans avoir eu explicitement spécifier qu'il est sur une case. Le joueur ne sait en effet pas **s'il est sur une case spéciale ou non**, c'est le plateau qui le sait.

La classe Patchwork

Pour représenter les joueurs, nous avons fait un **HashMap** qui a en clé l'identifiant du joueur (joueur 1 ou 2) en nombre et en valeur un objet Player étant le joueur respectif.

Nous avons mis un champ **playerTurn non final qui switch entre les valeurs 1 et 2** pour indiquer quel joueur doit jouer. Elle sert notamment a **généraliser la mis a jour au joueur concerné** en accédant a la **HashMap** grâce a sa clé qui désigne le joueur concerner par le tour.

III. Changement apportées au cours du développement

Pour effectuer des rotations, nous nous appuyons sur le fait que **l'endroit de référencer a la pose est le coin en haut a gauche.** De ce fait, lorsque nous effectuions nos rotations dans le tableau a deux dimensions du patch, nous avions une désynchronisation car **la forme se retrouvait dans les autres coins du tableau.** La forme était toujours formée a partir du point de référencer haut gauche. Afin de palier a ce souci, nous avons ajouté deux champs **width** et **height** afin de pouvoir toujours rogner la forme après la rotation dans le bon coin.

IV. Corrections et amélioration depuis la soutenance β

Une correction a été apporte a la méthode ***isInTheThreeNext*** dans la classe PatchesList qui avait un comportement qu'on ne voulait pas une fois que le neutral token avait effectuée un tour de la liste de patch.

Nous avons ajoute la version graphique du jeu. Pour cela, nous avons créer une interface UserInterface qui possède deux membres, UserInterfaceGraphic et UserInterfaceTerminal. Cela nous a permis de créer tous les affichages graphiques sans modifier la boucle principale et tout ce qui a été effectue en mécanisme de jeu et données. Il suffit juste de spécifier le bon UserInterfaceTerminal selon le choix effectue pour jouer au jeu soit en mode graphique, soit en mode terminal. Les deux étant membre de l'interface, nous avons pu utiliser le polymorphisme pour maintenir la même boucle de jeu.

V. Problème rencontré

Nous avons rencontré un problème pour lire des fichiers textuels et images. La méthodes getRessources, qui nous permet de charger nos images, ne fonctionne pas de la même façon que les bufferedReader pour le chemin spécifié. Ce problème nous a vraiment gêné pour la création et l'exécution dans le .jar. La solution actuelle fonctionne et le jeu est jouable mais ce n'est pas optimal.