Hakim AOUDIA
Walid BAKHTI

2022 / 2023
Licence 3 Semestre 5

Projet jeu Patchwork - Programmation en Java
Manuel Développeur

Sommaire :

Phase 1 : La base

1. Organisation

Avant tout, nous avons commencé par décider comment on allait mettre en place notre architecture. On s'est alors basé sur l'architecture MVC (Modèle - vue - contrôleur).
Pour cela on a créé plusieurs package, chaque package sert a une fonction bien définie.

Dans le package "Data" on a :

- Patch
- Player
- Quilt Board
- TimeBoard

Dans le package "Display":

- Console
- Graphics (on le mettra en place lors de la phase 3)

Dans le package "Main":

- Game
- Main

Dans le package "Files":

- 4 fichiers correspondant aux données utilisées pour initialiser le time board et les patchs selon les versions.

2 - Classe ou Record

Ensuite une fois les grands fichiers trouvés il a fallu déterminer lesquelles seront des Classe et lesquelles seront des Record.
Tous sont des Classe à l'exception de Patch et Player étant donné qu'on souhaite pouvoir accéder à leurs paramètres. Par exemple pour Patch il y a 4 paramètres et on créer plusieurs objets en modifiant ses 4 paramètres, il est alors plus facile d'accéder à ses paramètres avec un Record.

3 - Paramétrage des Classe/Record

Une fois les grands types de fichier trouvés il a fallu trouver de bons paramètres. On a alors lu les règles du PatchWork et visionner des vidéos expliquant les règles du jeu pour avoir une idée de comment on allait remplir nos Class/Record.

Par exemple dans QuiltBoard.java on a utilisé deux variables

```
private Player playerOne;
private Player playerTwo;
```

Cela nous permet de connaître la position des joueurs sur le plateau. Et aussi à savoir quel est le joueur qui est au-dessus de l'autre pour ensuite déterminer à qui est le prochain tour. Avec ces variables on a pas besoin de parcourir notre plateau d'entiers à chaque fois.

2. Choix des paramètres de la classe QuiltBoard

Dans notre classe QuiltBoard nous avons

private final ArrayList<int[][]> patchForm2DList;

Cela représente la position des Patch sous forme de coordonnées. Nous avons fait cela car nous aidons à placer les patchs sur le plateau et aussi pour détecter d'éventuelles collision entre les pièces.

On aussi ajouter un :

private int profitButton;

C'est pour éviter de parcourir la liste du plateau quilt board a chaque fois qu'on souhaite compter le nombre de boutons que rapporte les patchs du plateau. Il suffit d'incrémenter cette variable a chaque fois qu'on ajoute un patch.

3. Pose de la pièce sur le plateau

Lorsque on ajoute un Patch dans le plateau avec la méthode ***addPatch(Patch patch)*** on ajoute aussi ses coordonnées qu'on obtient grâce à la méthode ***convertFormPatchTo2DList()***.

Par exemple, un carré de taille 2x2 représenter de cette façon {{1,1}, {1,1}} dans un objet Patch est converti en coordonnées comme ceci : [[0, 0], [1, 0], [0, 1], [1, 1]].
On utilisera ensuite cela lors du déplacement des pièces ou de l'affichage du quilt board.

4. Vérification piece ne sorte pas du plateau

A ce stade du développement il nous manque de faire bouger les pièces sur le plateau grâce aux touches du clavier. On a essayé de faire bouger les pièces grâce aux flèches directionnelles. Le problème étant que nous n'avons pas réussi à implémenter la méthode **KeyEvent**. On s'est alors rabattus sur l'utilisation des **Scanner** que nous avons vu au début des cours.

Dès que appuyons sur une de ces touches **Z, Q, S, D**, nous vérifions au préalable que la pièce ne sorte pas du plateau du joueur avec la méthode ***moveEachCase()***.

Problème 1 :

Le problème est que nous n'avons pas réussi à respecter la limite imposée de 20 lignes par méthode. Notre méthode ***moveRotatePatch()*** dans la class QuiltBoard dépasser cette limitation et nous avons vaguement une idée de comment réduire le switch.

5. Vérifier qu'une pièce n'est pas superposé

Nous vérifions aussi si la pièce est superposée avec une des autres pièces posées dans le plateau.
A chaque tour de boucle on met à jour le plateau avec la méthode ***updateQuiltBoard()*** pour faciliter l'affichage et la détection d'une quelconque superposition.

6. On a corrigé nos variable en private

On s'est rendus compte que c'est une mauvaise façon de programmer que d'utiliser des champs non privés. On a alors modifier notre code en changeant certaines variables pour avoir des **private final** le plus possible. Ça nous permet de savoir que nos variables restent inchangées dans les autres classes/records.

Problème 2:

Comme on a mis la toute nos variable en **private**, il y avait certains cas où on pouvait pas accéder à des valeurs. Pour remédier a ce probleme on a créer des méthode commençant par ***getX()*** permettant de nous renvoyer la variable voulu. On s'assure alors qu'on lit seulement la variable et qu'on ne peut pas la modifier.

7. Problème dans récupération d'une pièce dans notre tableau

Problème 3 - Décalage lors de la suppression :

Lorsqu'on devait supprimer une pièce dans notre liste liste ***listTotalPatch*** de la classe TimeBoard.
Cela décaler naturellement toute notre ArrayList. Sauf que nous prenons ensuite la pièce à l'endroit de la pièce supprimée.
On avait alors provoqué un décalage car le joueur ne posait pas la bonne pièce dans son plateau.
On a corrigé ce problème, on s'est rendus compte que private final est très utile pour ce genre de problème car sans ***private int positionNeutralToken;***

On aurait pas pu connaitre ou on avait modifié la position du token rapidement étant donné qu'on a besoin de la position du neutral token dans plusieurs fichier.

Problème 4 - Neutral token arrive en fin de boucle :

un autre problème s'est manifesté quand le neutral token allait dépasser la taille de la liste total des patch, alors quand le joueur essayer de récupérer le prochain patch le neutral token dépasser le arrayList est donc provoqué une erreur. Pour résoudre ce problème on a créer la méthode ***updateNeutralToken()*** qui fait le modulo du neutral token avec le nombre de patch restants.

8. Difficulté dans la mise en place de rotation et retournement

On a tenté de mettre en place la rotation gauche, rotation droite et le retournement d'une pièce sans résultat probant. On a sous estimé ce problème qu'on a délaissé pour la fin cependant on a pas trouvé de bonne façon pour implémenter la rotation. On pense qu'en s'y attardant un peu plus on trouvera le moyen de mettre en place cela.

9. Difficulté à avoir des méthode de moins de 20 lignes

On sait que pour ce projet on nous demande d'avoir des méthodes de moins de 20 lignes cependant dans notre version intermédiaire il reste encore quelques méthodes qui ne

respectent pas cette règle. On a essayé au maximum pour l'instant et on espère pouvoir régler ce problème complètement d'ici la version finale et de n'avoir plus que méthodes courtes.

10. Le futur lors de la version graphique

On compte créer une classe Graphic.java qui nous permettra d'appeler des méthodes créant des affichages graphiques comme nous l'avons avec le fichier console.java.
Et on réfléchit aussi à créer une deuxième fichier Game qui sera nommé GameGraphics car le Game actuel appelle des méthodes de Console alors que le futur GameGraphics appelera des méthodes de Graphic.java

Correction du projet après la soutenance

Après la soutenance nous avions eu de nombreux conseils et donc devions changer certaines parties du code pour les améliorer.

1 - getQuiltBoard()

Parmis ces amélioration on peut parler de la fonction **getQuiltBoard()** dans la classe Quilt Board qui renvoie le tableau du quilt board ce qui n'était pas une bonne solution car on pouvait alors le modifier en dehors de sa Classe. Pour régler ce problème on a modifier la fonction pour copier toutes les valeurs du quilt board dans un autre tableau. On renvoie alors la copie et la version originale ne peut plus être modifiée de l'extérieur.

2 - Créer une classe enum

Dans notre précédente version on créait les structures enum directement dans les classes. Mais le professeur nous a conseillé de créer directement un fichier qui contiendra seulement nos enum. On s'est rendus compte que Eclipse possédait déjà cette option que nous n'avons pas vu avant. On a alors créé un enum pour les cases du QuiltBoard, un autre pour les cases du TimeBoard et enfin un enum pour les versions du jeu.

3 - On a simplifié des fonctions

Certaines de nos fonctions ressemblait à ceci if(1 == 1) return 1 sinon return 0. Ce qui est on le conçoit pas la version la plus courte à écrire. On a corrigé ceci sur quelque fonction pour retourner directement la condition ce qui retourne la même chose mais la fonction est plus courte.

4 - switch dans l'affichage de quiltBoard

Dans notre fonction d'affichage de quiltBoard version console on utilisait des if pour comparer chaque case du plateau. On nous a alors conseillé d'utiliser un switch case ce qui rend le code de la fonction d'abord plus agréable à lire et ensuite une petite amélioration de vitesse car le switch une fois une de ses conditions vrai va alors passer à l'itération suivante.

5 - Piece en cercle autour du quiltBoard

Une amélioration qu'on a pas réussi à implémenter. On nous a conseillé d'afficher les pièces autour du quiltBoard dans la version de la console, on a essayé de le faire mais la gestion du quilt board ainsi que des patch dans la console rend la chose compliqué. On a néanmoins appliqué ces conseils dans la version graphique qui rend cela plus facile à implémenter que dans une console. Cependant on a ajouté dans une de nos fonctions d'affichage de epatch le nombre de patch à afficher. On est donc plus limité à voir les 3 prochains patchs mais autant que le développeur veut. Dans la version rendue on a mis ce nombre a 10. Le joueur peut donc voir les 10 prochains Patch.

6 - Liste de joueur

L'amélioration la plus importante selon nous est celle ci. Le professeur nous a conseillé d'utiliser une liste de Joueur plutôt que de créer 2 variable joueur comme nous l'avons fait précédemment. Cela permet de simplifier le code car on a plus à se coltiner deux variable joueur mais seulement une liste et comme on possédait déjà une variable turnPlayer qui indique quelle joueur pouvait jouer il était aisé de savoir à quelle joueur il fallait faire certains changement.

Phase Graphique

1 - Difficulté au début

La grande difficulté de la partie graphique a été la prise en main de la bibliothèque graphique. Comme c'était la première fois en Java qu'on utilisait une bibliothèque graphique on a dû bien comprendre la démo donner dans le sujet avant de s'attaquer à la version graphique de PatchWork. Mais une fois les méthodes principales comprises (objet/méthodes les plus utiles comme les ApplicationContext, Rectangle2D.Float, renderFrame() …) il était assez simple de dessiner certaines formes à l'écran.

Le dessin de plateau de jeu ou d'objet en cercle nous avons déjà fait cela dans des précédents projets et donc on avait l'habitude.

2 - Nouveau paramètre

On a créé une classe Graphic.java et une autre GameManagerGraphic.java pour cette phase. Graphic servait a mettre les méthodes de dessins de base comme le quitlboard, timeboard ou les patchs. Tandis que **GameMangerGraphic** gérait les touches du clavier de l'utilisateur principalement. Dans le fichier Game.java on a ajouté de nombreuses fois la variable **VersionGame version** et **ApplicationContext context**.
La variable version servait à connaître quel est le code à utiliser selon la version de jeu choisie. Et context servait à transmettre la fenêtre graphique ou on devait dessiner.

Phase Automa

Pour la phase Automa nous avons décidé de faire une Class Automa et un Record AutomaCards qui permet d'avoir une structure pour toutes les cartes du jeu Automa. Tout la gestion du joueur Automa est faite dans la Class Automa cela nous a rendu le travail plus simple a faire. Dans la Class Game il y a une fonction qui permet de faire jouer Automa. Dans cette phase, le plus compliqué était de comprendre comment fonctionne le joueur Automa.

Pour l'implémentation du code cela était simple car pour l'autre joueur rien ne change, il fallait juste ajouter la partie qui permet de faire jouer Automa. Le plus important dans cette phase c'était de ne pas modifier tout le code mais d'ajouter Automa comme une option mais quand on a voulu mettre en place Automa il a fallu ajouter récursivement Automa a chaque paramètre de chaque fonction qui l'utilise.

Rendu final

Des problèmes

On a eu **beaucoup** de difficulté à essayer d'implémenter le build.xml. On a compris qu'il agissait comme un makefile pour Java. On a réussi à faire la commande **compile**, **jar** mais pas **javadoc** ni **clean**. Le problème étant que lorsqu'on générait le fichier **Patchwork.jar** et qu'on lançait le programme cela mettait des messages d'erreurs comme celui ci :

```
udia_Bakhti_Patchwork$ java -jar Patchwork.jar
Error: Invalid or corrupt jarfile Patchwork.jar
```

Ou encore celui ci

```
hakim@hakim-ThinkPad-T450s:~/Documents/L3/Java/PROJET/Version final/V1$ java -jar PatchWork.jar
Erreur : impossible de trouver ou de charger la classe principale classes
Causé par : java.lang.ClassNotFoundException: classes
```

On a des vagues idées expliquant pourquoi cela ne marche pas mais même en cherchant à régler le problème on a pas eu de résultat probant. On a donc décidé de créer manuellement le **Patchwork.jar** comme pour la version intermédiaire. Et lorsqu' on créer manuellement on a plus eu de problème donc on a gardé cette version. On a aussi fait la javadoc manuellement car **ant javadoc** donnait aussi des erreurs.