UNIVERSITÉ
CAEN
NORMANDIE

Taquin

Rapport de projet

ANDRÉ Lorada 21809742
DEROUIN Auréline 21806986
MARTIN Justine 21909920
THOMAS Maxime 21810751

Table des matières

1 Présentation du projet

A Présentation de l'application

2 Organisation du projet

A Gestion du projet

Afin de faciliter la communication et le bon déroulement de la conception de notre application, divers moyens ont été mis en oeuvre.

i Gestionnaire de version

Tout d'abord, nous pouvons citer l'utilisation d'un gestionnaire de version afin de permettre la centralisation du code et rendre le travail en équipe bien plus efficace. Le choix de celui-ci étant imposé (*Subversion*), il n'est pas nécessaire d'en parler plus longtemps.

ii Trello

Concernant la répartition et le "listing" du travail à effectuer, nous avons fais le choix d'utiliser Trello, une plateforme qui nous permet d'utiliser des tableaux pour planifier un projet.

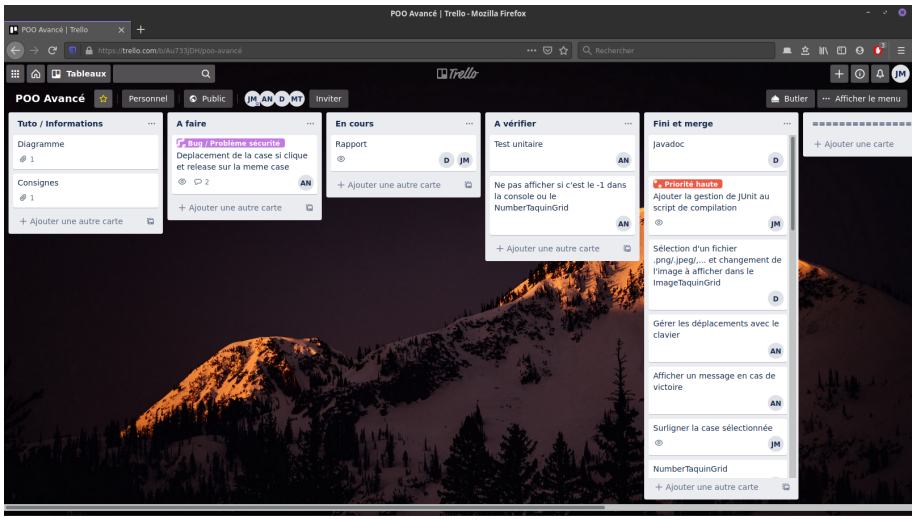

FIGURE 2.1 – Notre tableau Trello

Ainsi, comme nous pouvons le constater, les différentes tâches passent par différents états, "*A faire*", "*En cours*", "*A vérifier*", "*Fini et merge*". Enfin, bien que ce ne soit pas visible sur l'image 2.1, il existe un "*Backlog*" sur la droite qui contient les différentes tâches restantes à accomplir. Celles-ci peuvent ensuite être déplacées dans la colonne "*A faire*" au moment où nous jugeons qu'elles peuvent être réalisées.

Les colonnes "*A verifier*" et "*Fini et merge*" nécessitent quelques précisions, les autres parlant d'elles-même. Pour la première, lorsqu'une tâche est terminée, elle est soumise à évaluation et relecture. Cela permet d'obtenir un avis sur la fonctionnalité et d'éviter d'éventuels bugs par la suite mais aussi de garder une cohérence au travers du code. Raisons pour lesquelles les personnes qui effectuent cette relecture sont

souvent les mêmes. Enfin, quand celle-ci est vérifiée et validée, on peut alors la déplacer dans la seconde colonne.

iii Discord

Afin de faciliter la communication au sein du groupe, nous avons utilisé le service de messagerie Discord car tous les membres du groupe l'utilisaient déjà de manière personnelle. Celui-ci permet de parler par le biais de "serveurs" gratuits dans lesquels nous pouvons ajouter des salons textuels ou des salons vocaux à volonté. Ainsi, nous avions deux salons de discussion. L'un nommé "*important-taquin*" permet de transmettre des messages importants sur ce qui a été fait, sur des changements importants concernant le projet, etc. L'autre se nommant "*dev-taquin*" était une discution beaucoup plus générale dans laquelle on pouvait demander de l'aide, aider des membres en difficulté, ou même de discuter de certains choix à faire.

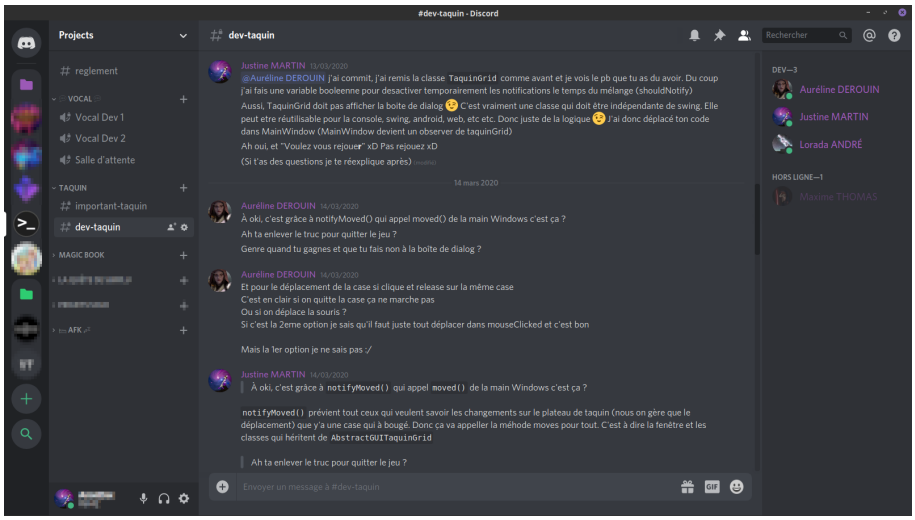

FIGURE 2.2 – Notre serveur Discord

B Répartition des fonctionnalités

Auréline DEROUIN	- Vue-Contrôleur du taquin avec les numéros (fenêtre) - Classe Modèle du Taquin - Évènements clavier - Énum Direction - Afficher un message en cas de victoire (fenêtre) - Tests unitaire - Rapport
Justine MARTIN	- Vue-Contrôleur du taquin avec une image - Vue-Contrôleur en mode console - Évènements souris - Mettre en évidence la case déplaçable - Déplacement des cases avec la souris - Boite de dialogue (classe mère et dialogue de nouvelle partie) - Script build.sh - Javadoc (relecture et précisions seulement) - Rapport
Lorada ANDRE	- Gestion de la fenêtre principale (conception, menu, évènement, etc) - Gestion des arguments dans la Main - Sélection et changement de l'image à afficher sur le Taquin - Javadoc - Commande pour zipper dans build.sh - Rapport
Maxime THOMAS	- Classe Modèle du Taquin - Interfaces pour le Pattern Observer - Remélange le taquin si une fois mélangé, il est déjà complété

3 Architecture du projet

A Arborescence du projet

B Mise en place de MVC

i Modèle

ii Pattern Observer

iii Vue-Contrôleur

4 Aspects techniques

A TaquinGrid

Tout d'abord, voici la classe *TaquinGrid* qui permet de créer et de gérer la grille, que cela soit pour mélanger, effectuer les déplacements, ou encore dire si le joueur à gagner ou pas.

i Création de la grille

Cette algorithme permet de créer la grille du taquin. Le constructeur de la class TaquinGrid est avant tout appelé permettant d'avoir les *width* et *height* de la grille. Puis, la grille est alors initialisé en fonction de ces deux variables. Deux boules **FOR** sont alors nécessaires afin de remplir la grille de numéro. La valeur *-1*, correspondant à la case vide, est alors initialisé à la dernière case de la grille.

Algorithm 1: createGrid() :void

```
1 this.grid ←int [this.width][this.height]
2 for y = 0; y < this.height; y++ do
3 |   for x = 0; x < this.width; x++ do
4 |   |   this.grid[x][y] ← x + y * this.width + 1
5 |   end
6 end
7 this.grid[this.width − 1][this.height − 1] ← −1
```

ii Algorithme de mélange

La méthode randomizeGrid() permet de mélanger la grille *n* fois. Tout d'abord, un nombre aléatoire est initialisé à chaque tour de la boucle **FOR**, permettant ainsi de réaliser un mouvement grâce à la méthode move(). Si jamais un mouvement n'est pas possible, la boucle rajoute un tour de plus permettant que le mélange se fasse bien *n* fois. Une condition a été rajouté à ce mélange : si jamais la grille est déjà "terminer", la méthode est donc rappelé afin d'obtenir une grille mélangé.
Ce mélange évite les parties sans solutions. En effet, si on aurait utilisé un random() simple, il se pourrait que la grille n'aurais pas de solution.

Algorithm 2: randomizeGrid(int n) :void

```
1  r ← new Randow()
2  for int i = 0; i < n; i++ do
3  |   nbrRandom ← r.nextInt(4)
4  |   dir ← null
5  |   if nbrRandom == 0 then
6  |   |   dir ← HAUT
7  |   else if nbrRandom == 1 then
8  |   |   dir ← DROITE
9  |   else if nbrRandom == 2 then
10 |   |   dir ← BAS
11 |   else if nbrRandom == 3 then
12 |   |   dir ← GAUCHE
13 |   if !move(dir) then
14 |   |   i ← 1 − i
15 end
16 if finisehd() then
17 |   randomieGrid(intn)
```

iii Déplacement d'une case

La méthode move() permet d'effectuer des "déplacements" dans la grille grâce aux mouvement que l'on envoie dans la méthode (variable *direction*).
La direction voulu est tout d'abord vérifié. Si la direction n'est pas possible, le "déplacement" ne s'éxècute pas. Un false est alors retourné. Au contraire, si un déplacement est possible, la valeur de la case est alors "échanger" avec la case vide. En effet, la valeur de la case vide, soit *-1*, est alors égal à la valeur de la case voulu. Et la case que l'on veut déplacer est alors égale à la case vide (*-1*). La méthode renvoie alors true, notifiant la méthode appelant que le déplacement a bien été effectuer.

Algorithm 3: move(Direction direction) :boolean

```
1  if direction == HAUT && this.posYVide == this.height − 1 then
2  |  return false
3  else if direction== DROITE && this.posXVide== 0 then
4  |  return false
5  else if direction== BAS && this.posYVide== 0 then
6  |  return false
7  else if direction== GAUCHE && this.posXVide== this.width − 1 then
8  |  return false
9  if direction == HAUT then
10 |  this.grid[posXVide][posYVide] ← this.grid[posXVide][posYVide + 1]
   |    this.grid[posXVide][posYVide + 1] ← −1 this.posYVide + +
11 else if direction == DROITE then
12 |  this.grid[posXVide][posYVide] ← this.grid[posXVide − 1][posYVide]
   |    this.grid[posXVide − 1][posYVide] ← −1 this.posYVide − −
13 else if direction == BAS then
14 |  this.grid[posXVide][posYVide] ← this.grid[posXVide][posYVide − 1]
   |    this.grid[posXVide][posYVide − 1] ← −1 this.posYVide − −
15 else if direction == GAUCHE then
16 |  this.grid[posXVide][posYVide] ← this.grid[posXVide + 1][posYVide]
   |    this.grid[posXVide + 1][posYVide] ← −1 this.posYVide + +
17 return true
```

iv Déterminer si la partie est terminée

Cette méthode, finished(), permet de renvoyer True si la partie est terminée. Elle est exécuté à chaque mouvement effectuer (méthode move()).
Deux boucles **FOR** sont nécessaire afin de parcourir la grille. Si jamais la grille n'est pas une suite de nombre, donc si la grille n'est pas "terminer", la méthode renvoie false. A l'inverse, si la grille à une suite de nombre et que la dernière valeur est égale à *-1*, cela signifie donc que la grille est bien terminé. La méthode renvoie alors true.

Algorithm 4: finished() :boolean

```
1 for int y = 0; y < this.height; y + + do
2 |  for int x = 0; x < this.width; x + + do
3 |  |  if y == this.height − 1 && x == this.width − 1 && this.grid[x][y] == −1 then
4 |  |  |  continue
5 |  |  if this.grid[x][y] != x + y ∗ this.width + 1 then
6 |  |  |  return false
7 |  end
8 end
9 return true
```

B Tests unitaires

i Quel outil ?

Afin de réaliser les tests unitaires, nous avons décidé d'utiliser JUnit, l'un des meilleurs framework, si ce n'est le meilleur, dans le domaine des tests unitaires en Java. Pour être exacte nous avons utilisé la version 4 de celui-ci. Pour ne citer que quelques raisons concernant ce choix on peut parler de la simplicité remarquable concernant l'écriture des tests, l'apprentissage très rapide, 3 membres du groupe ne connaissait pas le framework et pourtant tous en on compris le fonctionnement de base. On peut également noter la forte intégration que possède ce framework sur diverses plateforme (GitHub, Jenkins, ...) ce qui peut être utile dans des projets utilisants ces services, bien que ce ne soit pas le cas de la forge, il peut tout de même être utile de noter celà.

Concernant l'éxecution de ces tests, le script build.sh possède une sous commande *test* permettant de lancer les différents tests. Ainsi il se charge lui même de télécharger les dépendances nécessaires, de compiler les classes de test présentes dans le dossier prévu à cet effet (*test* ici), puis d'effectuer les différents tests.

ii Mise en oeuvre

Nous avons décidé d'effectuer des test sur la class TaquinGrid car c'est une class principal, permettant de gérer tout le jeu. En effet, elle crée la grille, déplace les cases, recherche une valeur, mélange le jeu et regarde si la partie est terminer. Toutes ces méthodes sont donc très importante dans tout le déroulement principale du jeu.
Toutes ces méthodes citées sont alors testées et validées grâce aux testes unitaires.
Pour la méthode des déplacements notamment, nous avons procédé à des tests de directions ainsi qu'à une vérification de déplacement afin de savoir si la direction demandé était bien appliqué.

5 Conclusion

A Avis général

Le Taquin a été une approche pratique intéressante concernant le Pattern MVC. L'intérêt de celui-ci apparait très clairement lors du changement de JPanel où l'on conserve le même modèle (*TaquinGrid*) bien que la manière d'afficher l'information diffère, soit en affichant une image, soit en affichant des chiffres. Le pattern permet également une nette séparation entre les classes modèles qui se retrouvent réutilisables dans un autre contexte, dû au fait qu'elles n'embarquent pas de code spécifique au contrôle des évènements ou à l'affichage de la vue.

Le seul bémol de ce projet concerne la taille des groupes. En effet, cela a été très difficile de départager les différentes tâches à effectuer en 4. Plusieurs personnes devaient donc travailler ensemble sur un même code car il était impossible de travailler sur 4 tâches différentes en simultané.

B Éléments à améliorer

Bien que tous les éléments demandés soient remplis, voici une liste non exhaustive d'améliorations possibles :

- Afficher un compteur de coups
- Ajouter un système de score prenant en compte le temps mis pour résoudre la grille ainsi que le nombre de coups
- Sauvegarder les scores avec le nom d'utilisateur du joueur
- Ajouter un mode versus où 2 joueurs seraient en compétition pour finir le plus rapidement possible une même grille