



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	1
2	Organisation du projet	2
A	Gestion du projet	2
i	Gestionnaire de version	2
ii	Trello	2
iii	Discord	3
B	Répartition des fonctionnalités	4
3	Architecture du projet	5
A	Arborescence du projet	5
B	Mise en place du pattern M-VC	6
4	Aspects techniques	7
A	TaquinGrid	7
i	Création de la grille	7
ii	Algorithme de mélange	7
iii	Déplacement d'une case	8
iv	Déterminer si la partie est terminée	9
B	Tests unitaires	10
i	Quel outil ?	10
ii	Mise en oeuvre	10
5	Conclusion	11
A	Avis général	11
B	Éléments à améliorer	11

1 Présentation du projet

Le taquin est un jeu de puzzle, constitué d'un rectangle dans lequel se trouve des cases. Ces cases contenant des lettres de l'alphabet, des nombres ou encore des morceaux d'images, peuvent glisser les unes sur les autres. Parmi les cases, l'une d'entre elles est vide. Le but du jeu du taquin consiste à reconstituer le puzzle en formant une image (lorsque les cases forment une image) ou de ranger les nombres par ordre croissant (lorsque les cases contiennent des nombres) en glissant l'un des éléments contigus à la case vide vers cette dernière. Avec l'interface graphique, l'utilisateur a la possibilité de choisir sa propre image pour le taquin qu'il devra reconstituer.

Le but de ce devoir à été de réaliser un taquin sous forme d'une application de jeu, dotée d'une interface graphique, tout en pouvant être utilisée sans interface graphique c'est-à-dire être jouable en ligne de commande. L'application a été intégralement réalisée avec le design pattern MVC, avec un modèle totalement indépendant de l'interface graphique.

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 fait le choix d'utiliser [Trello](#), une plateforme qui nous permet d'utiliser des tableaux pour planifier un projet.

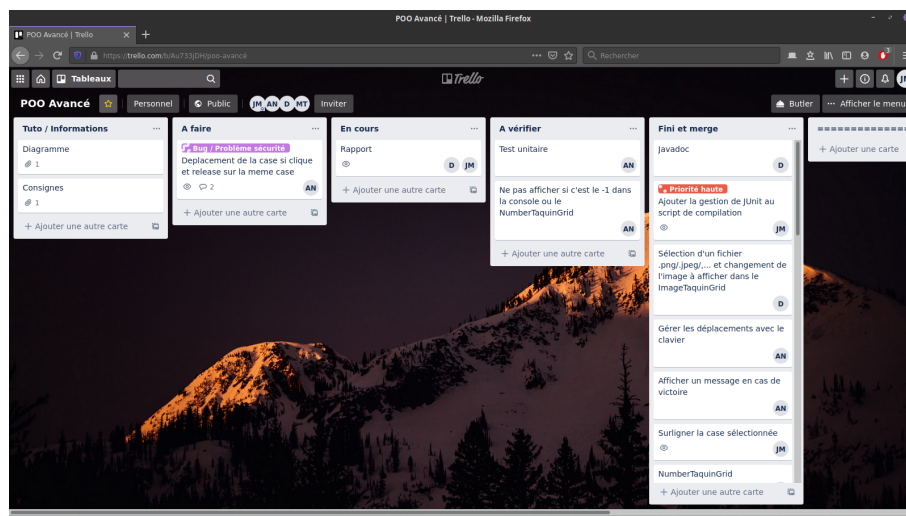


FIGURE 2.1 – Notre tableau Trello

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 vérifier" et "Fini et merge" nécessitent quelques précisions, les autres parlant d'elles-mêmes. 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 avons 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 discussion 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.

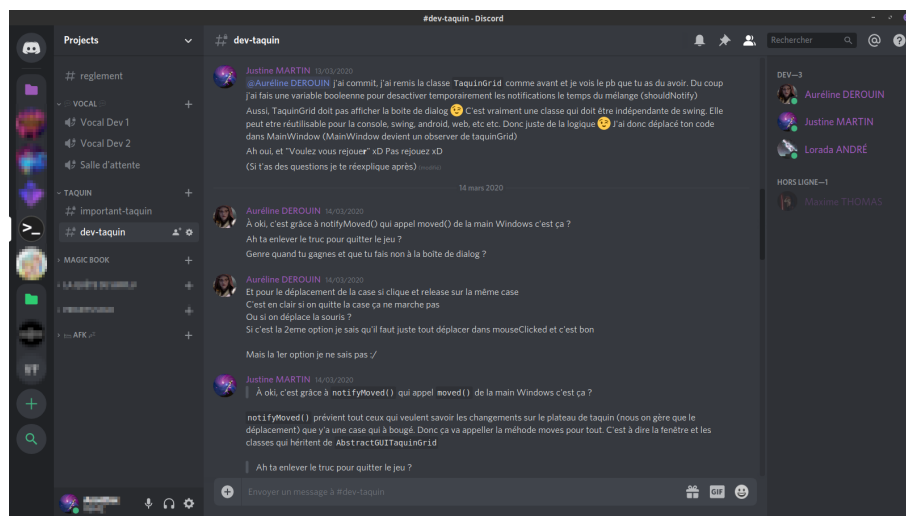


FIGURE 2.2 – Notre serveur Discord

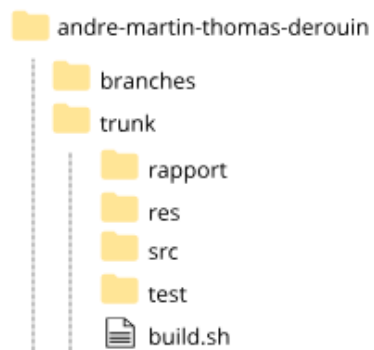
B Répartition des fonctionnalités

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
Justine MARTIN	- Rapport
	- 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
Lorada ANDRE	- Boîte de dialogue (classe mère et dialogue de nouvelle partie)
	- Script build.sh
	- Javadoc (relecture et précisions seulement)
	- Rapport
	- Gestion de la fenêtre principale (conception, menu, évènement, etc)
	- Gestion des arguments dans la Main
Maxime THOMAS	- Sélection et changement de l'image à afficher sur le Taquin
	- Javadoc
	- Commande pour zipper dans build.sh
	- Rapport
	- 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

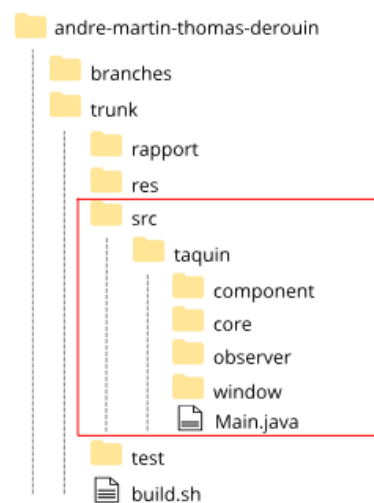
Notre application est organisée de la manière suivante :



On y retrouve 4 dossiers et 1 fichier :

- le dossier *rapport*, contenant ce rapport sous latex
- le dossier *res*, conteant les ressources du jeu, ce qui correspond à l'image par défaut pour le taquin
- le dossier *src* contenant le code source de l'application
- le dossier *test* contenant le code se chargeant des tests unitaires
- le fichier *build.sh*, qui permet l'exécution de l'application.

Le code source du projet est situé dans le chemin *src/taquin/*. Il est constitué par :



- un package *component*, contenant les classes des différents modes qui composent l'application.
- un package *core* contenant le cœur du jeu du taquin, qui est commun à tout les modes de jeu, ainsi que la classe qui énumère les directions
- un package *observer* qui constitue le pattern Observer
- un package *window* contenant les classes permettant de gérer ce qui se rapporte à la fenêtre de jeu et des boites de dialogues
- un fichier *Main.java*, qui est le fichier executable du taquin

B Mise en place du pattern M-VC

Dans notre application, le pattern M-VC à été mis en place de la manière suivante :

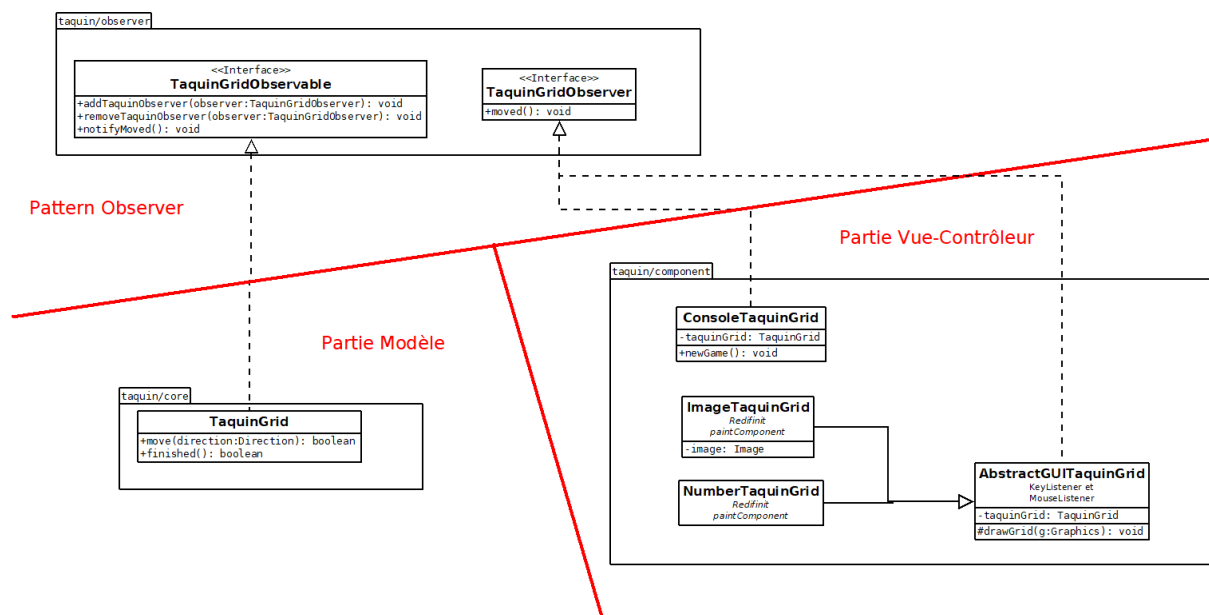


FIGURE 3.1 – Notre tableau Trello

On a d'une part la classe *TaquinGrid* représente le modèle de l'application. Il s'agit du coeur du jeu, du fonctionnement du taquin, peut importe le mode d'affichage du jeu.

D'un autre côté, nous avons tout le package compoment, qui lui, contient les différentes vues de l'application. On retrouve dans le taquin trois vues :

- le mode "console", représenté par la classe *ConsoleTaquinGrid*
- le mode "image", représenté par la classe *ImageTaquinGrid*
- enfin, le mode "nombre" représenté par la classe *NumberTaquinGrid*

La classe *AbstractGUITaquinGrid* est uniquement une vue qui regroupe les parties communes aux classes *ImageTaquinGrid* et *NumberTaquinGrid*.

Ces trois classes citées ci-dessus sont également des contrôleurs puisqu'elles

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 algorithmme 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   $r \leftarrow \text{new Random}()$ 
2  for  $\text{int } i = 0; i < n; i++$  do
3       $\text{nbrRandom} \leftarrow r.\text{nextInt}(4)$ 
4       $\text{dir} \leftarrow \text{null}$ 
5      if  $\text{nbrRandom} == 0$  then
6           $\text{dir} \leftarrow \text{HAUT}$ 
7      else if  $\text{nbrRandom} == 1$  then
8           $\text{dir} \leftarrow \text{DROITE}$ 
9      else if  $\text{nbrRandom} == 2$  then
10          $\text{dir} \leftarrow \text{BAS}$ 
11      else if  $\text{nbrRandom} == 3$  then
12          $\text{dir} \leftarrow \text{GAUCHE}$ 
13      if  $\text{!move}(\text{dir})$  then
14          $i \leftarrow 1 - i$ 
15  end
16  if  $\text{finisehd}()$  then
17       $\text{randomieGrid}(\text{intn})$ 

```

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'exécute 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]
11  | this.grid[posXVide][posYVide + 1] ← -1 this.posYVide ++
12 else if direction == DROITE then
13  | this.grid[posXVide][posYVide] ← this.grid[posXVide - 1][posYVide]
14  | this.grid[posXVide - 1][posYVide] ← -1 this.posYVide --
15 else if direction == BAS then
16  | this.grid[posXVide][posYVide] ← this.grid[posXVide][posYVide - 1]
17  | this.grid[posXVide][posYVide - 1] ← -1 this.posYVide --
18 else if direction == GAUCHE then
19  | this.grid[posXVide][posYVide] ← this.grid[posXVide + 1][posYVide]
20  | this.grid[posXVide + 1][posYVide] ← -1 this.posYVide ++
21 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 ont compris le fonctionnement de base. On peut également noter la forte intégration que possède ce framework sur diverses plateformes (GitHub, Jenkins, ...) ce qui peut être utile dans des projets utilisant ces services, bien que ce ne soit pas le cas de la forge, il peut tout de même être utile de noter cela.

Concernant l'exécution 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 tests sur la classe `TaquinGrid` car c'est une classe principale, 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 terminée. Toutes ces méthodes sont donc très importantes dans tout le déroulement principal du jeu.

Toutes ces méthodes citées sont alors testées et validées grâce aux tests 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 demandée était bien appliquée.

5 Conclusion

A Avis général

Le Taquin a été une approche pratique intéressante concernant le Pattern MVC. L'intérêt de celui-ci apparaît 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 événements 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 répartir 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