Taquin

Rapport de projet

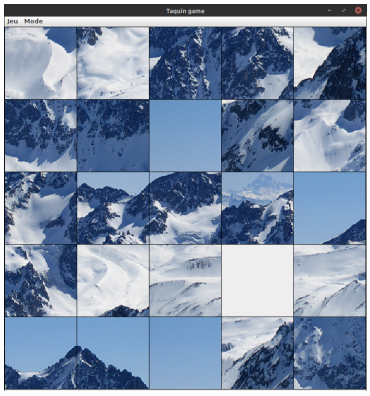
ANDRÉ Lorada 21809742
DEROUIN Auréline 21806986
MARTIN Justine 21909920
THOMAS Maxime 21810751

Table des matières

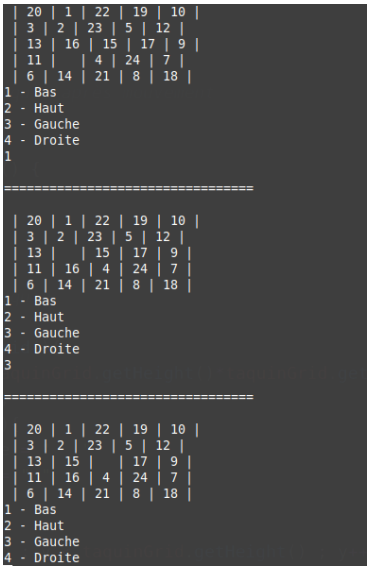
1 Présentation du projet

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 celle-ci. 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 a été de réaliser un taquin sous forme d'une application de jeu, dotée d'une interface graphique (voir image 1.1a) , tout en pouvant être utilisée sans interface graphique (voir image 1.1b) 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.



(a) Image du jeu avec le mode Image



(b) Image du jeu en console

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 œuvre.

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 Compilation

Nous avons décidé de réutiliser un script de compilation déja créé lors d'un précédent TP noté en Java. Il s'agit de **build.sh**. Celui-ci possède différentes sous commandes très utiles et qui permettent d'effectuer les taches courantes sur le projet. En voici une liste reprenant les plus importantes de ces sous commandes :

build Compile le projet

clean Supprime tous les dossiers générés (build, doc, dist, ...)

deploy Compile et package le projet en un jar exécutable

javadoc Génère la Javadoc du projet

run Lance le projet. On peut passer des arguments en utilisant *–args "arg0 arg1"*. Exemple : *./build.sh –args "10 10"*

test Lance les différents tests unitaires.

Le script est facilement adaptable. Il suffit de modifier les quelques constantes au début de celui-ci.

iii Trello

Concernant la répartition et le listage 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.

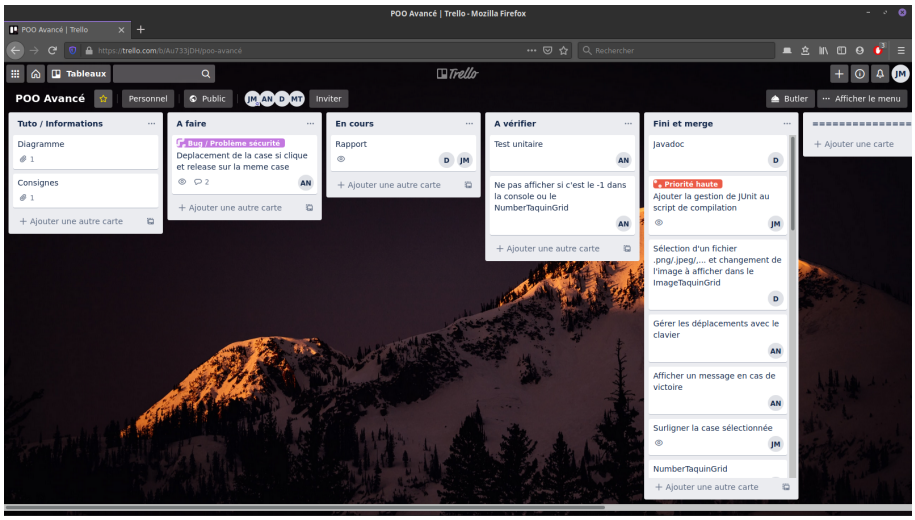


FIGURE 2.1 – Notre tableau Trello

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 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, nous pouvons alors la déplacer dans la seconde colonne.

iv 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*", est une discution beaucoup plus générale dans laquelle on peut demander de l'aide, aider des membres en difficulté, ou même de discuter de certains choix à faire.

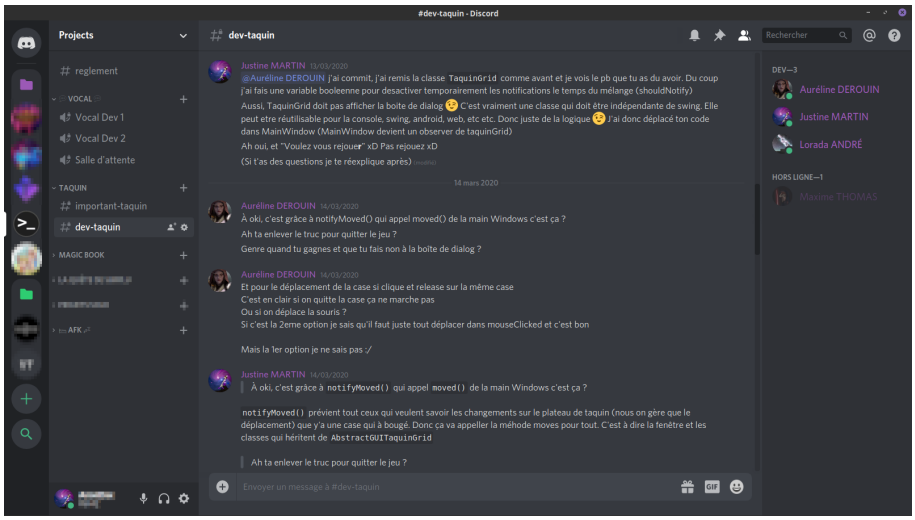


FIGURE 2.2 – Notre serveur Discord

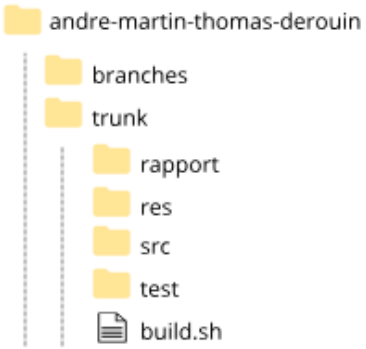
B Répartition des fonctionnalités

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 unitaires - 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, événement, 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

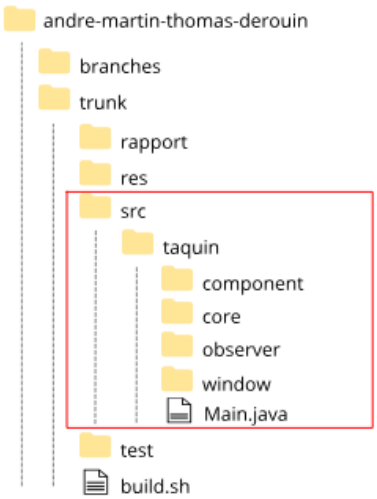
Notre application est organisée de la manière suivante :



On y retrouve 4 dossiers et 1 fichier :

rapport : Contient ce rapport sous latex
res : Contient les ressources du jeu, c'est-à-dire l'image par défaut pour le taquin
src : Contient le code source de l'application
test : Contient le code se chargeant des tests unitaires
build.sh : Fichier de compilation du taquin.

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



component : Contient les différents composants de l'application.

core : Contient le cœur du jeu du taquin, non dépendant de l'affichage graphique, ainsi que la classe qui énumère les directions

observer : Contient les classes relatives à l'implémentation du pattern Observer

window : Contient les classes permettant de gérer ce qui se rapporte à la fenêtre de jeu et des boites de dialogues

Main.java : Classe exécutable du projet.

B Mise en place du pattern MVC

Dans notre projet, il nous a été demandé d'utiliser le design pattern MVC. Ce design pattern a été mis en place de la manière suivante :

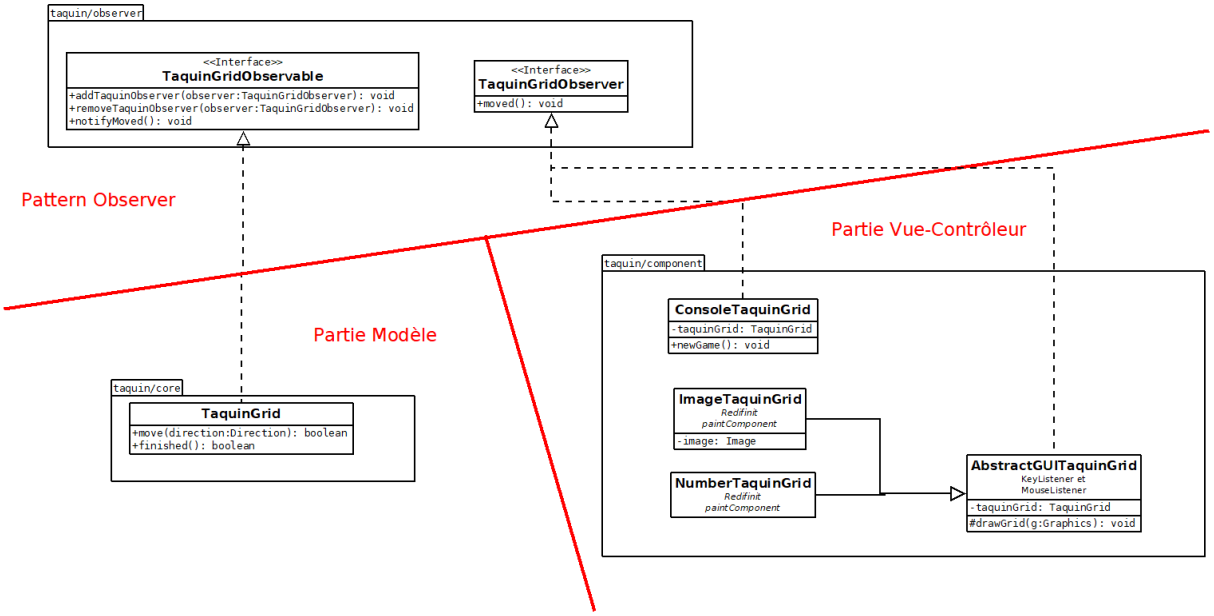


FIGURE 3.1 – Notre tableau Trello

Nous avons la classe **TaquinGrid** qui est le modèle pour une grille de jeu. Tous les traitements sont absolument identiques peu importe la vue. On utilisera la même méthode pour déplacer une case, par exemple, peu importe qu'il s'agisse d'un affichage en console ou en fenêtre.

Nous avons d'un autre coté, tout le package component, qui lui, représente à la fois les différentes vues de l'application qui jouent également le rôle de contrôleurs. Nous avons décidé de concevoir trois Vues-Contrôleurs différentes :

Pour le monde console : Vue-Contrôleur représentée par la classe **ConsoletaquinGrid**
Pour le mode image : Vue-Contrôleur représentée par la classe **ImagetaquinGrid**
Pour le mode nombre : Vue-Contrôleur représentée par la classe **NumbertaquinGrid**

Pour être plus précis, la classe mère **AbstractGUItaquinGrid** est un contrôleur parent aux classes **ImagetaquinGrid** et **NumbertaquinGrid**. Ces deux dernières ne font que de redéfinir la manière dont nous affichons l'information grâce à la méthode *void paintComponent(Graphics)*

4 Aspects techniques

A TaquinGrid

Commençons par parler de la classe *TaquinGrid* qui permet de créer et de gérer la grille, que cela soit pour mélanger, effectuer les déplacements, ou encore notifier à l'aide d'un booléen si le joueur à gagné ou non.

i Création de la grille

Cette algorithme permet de créer la grille du taquin. Lors de l'appel au constructeur de la classe **TaquinGrid**, nous lui fournissons une *width* et une *height* pour connaître les dimensions de la grille à créer. La grille est alors initialisée en fonction de ces deux variables. Deux boucles **FOR** sont alors nécessaires afin de remplir la grille avec les bons numéros. La valeur *-1*, correspondant à la case vide, est alors initialisée 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(int)* permet de mélanger la grille en déplaçant les cases *n* fois dans des directions choisies au hasard. Un nombre aléatoire est donc tiré à chaque tour de la boucle **FOR**, permettant ainsi de représenter un mouvement qui sera effectué grâce à la méthode *move()*. Si jamais un mouvement n'est pas possible, par exemple parce qu'il n'y a pas de case à déplacer dans la direction souhaitée, la boucle ajoute un tour de plus permettant ainsi, de vraiment mélanger la grille *n* fois. Une condition a été rajoutée à la fin de ce mélange : si jamais la grille est déjà "terminée", la méthode est de nouveau rappelée afin d'obtenir une grille mélangée. Ce type de mélange permet d'éviter une partie sans aucune solution. En effet, si nous avions placé les nombres de manière totalement aléatoire, il se pourrait

que la grille n'ait 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 ← i − 1
15 end
16 if finisehd() then
17 |   randomieGrid(n)
```

Comme nous pouvons le voir, la méthode move retourne un booléen afin de savoir si le mouvement a été effectué ou non.

iii Déplacement d'une case

La méthode *move(Direction)* permet d'effectuer des "déplacements" dans la grille grâce à la direction que l'on donne en argument.
Tout d'abord nous avons effectué des vérifications sur la grille en fonction du mouvement demandé. Si celui-ci n'est pas possible, le "déplacement" ne s'exécute pas et la méthode retourne false, signifiant que le mouvement n'a pas eu lieu. Au contraire, si un déplacement est possible, la valeur de la case est alors "échangée" avec la case vide. En effet, la valeur de la case vide, soit *-1*, est alors égale à la valeur de la case voulue. La case que l'on veut déplacer est alors égale à la case vide (c'est-à-dire *-1*, que nous avons nommé sous le nom de la constante *TaquinGrid.EMPTY_SQUARE*). La méthode renvoie alors

true, notifiant la méthode appelante que le déplacement a bien été effectué.

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;
12 |  this.posYVide + +;
13 else if direction == DROITE then
14 |  this.grid[posXVide][posYVide] ← this.grid[posXVide − 1][posYVide];
15 |  this.grid[posXVide − 1][posYVide] ← −1;
16 |  this.posYVide − −;
17 else if direction == BAS then
18 |  this.grid[posXVide][posYVide] ← this.grid[posXVide][posYVide − 1];
19 |  this.grid[posXVide][posYVide − 1] ← −1;
20 |  this.posYVide − −;
21 else if direction == GAUCHE then
22 |  this.grid[posXVide][posYVide] ← this.grid[posXVide + 1][posYVide];
23 |  this.grid[posXVide + 1][posYVide] ← −1;
24 |  this.posYVide + +;
25 return true
```

iv Déterminer si la partie est terminée

Cette méthode, *finished*, permet de renvoyer True si la partie est terminée. Elle est exécutée par les Vues-Contrôleurs à chaque mouvement effectué (méthode *move*) afin d'afficher un message de fin si la partie est terminée.
Deux boucles **FOR** sont nécessaires afin de parcourir la grille. Pour vérifier si la grille est terminée nous utilisons la même méthode que dans la création de celle-ci, mais cette fois-ci, nous comparons les nombres de la grille avec celui qui devrait y être présent.

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 exact, nous avons utilisé la version 4 de celui-ci. Pour ne citer que quelques raisons concernant ce choix nous pouvons parler de la simplicité remarquable concernant l'écriture des tests, l'apprentissage très rapide (trois membres du groupe ne connaissaient pas le framework et pourtant tous en ont compris le fonctionnement de base). nous pouvons 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 tests 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 il s'agit d'une classe principale, permettant de gérer l'ensemble du 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.
Les méthodes citées ci-dessus sont alors testées et validées grâce aux tests unitaires.

Pour la méthode des déplacements notamment, nous avons procédé à des tests de direction (voir code 4.1). Pour cela, une grille non mélangée est créée. La case vide (*TaquinGrid.EMPTY_SQUARE*) est donc située à la case tout en bas à droite. Un **move** (voir code 3) est alors effectué en *Direction.BAS*. La case située en haut de la case vide (9,8) est alors déplacée à la place de la case vide (9,9). L'ancienne case (9,8) contient normalement la "case vide". Le déplacement peut alors être vérifié en regardant si la case vide est bien placée en (9,8) ainsi que la valeur, anciennement contenue dans la case (9,8), est bien située en case (9,9). Le même procédé est effectué pour toutes les **Directions** possibles : "Gauche", "Haut", "Droite".

```
1     public void move() {
2         TaquinGrid gridCreate = new TaquinGrid(10, 10, false);
3         int ancienneValeur;
4
5         ancienneValeur = gridCreate.getSquare(9,8);
6         Assert.assertTrue(gridCreate.move(Direction.BAS));
7         Assert.assertTrue(gridCreate.getSquare(9,8) == TaquinGrid.EMPTY_SQUARE);
8         Assert.assertTrue(gridCreate.getSquare(9,9) == ancienneValeur);
9
10        ancienneValeur = gridCreate.getSquare(8,8);
11        Assert.assertTrue(gridCreate.move(Direction.DROITE));
12        Assert.assertTrue(gridCreate.getSquare(8,8) == TaquinGrid.EMPTY_SQUARE);
13        Assert.assertTrue(gridCreate.getSquare(9,8) == ancienneValeur);
14
15        ancienneValeur = gridCreate.getSquare(8,9);
16        Assert.assertTrue(gridCreate.move(Direction.HAUT));
17        Assert.assertTrue(gridCreate.getSquare(8,9) == TaquinGrid.EMPTY_SQUARE);
18        Assert.assertTrue(gridCreate.getSquare(8,8) == ancienneValeur);
19
20        ancienneValeur = gridCreate.getSquare(9,9);
21        Assert.assertTrue(gridCreate.move(Direction.GAUCHE));
```

```
22          Assert.assertTrue(gridCreate.getSquare(9,9) == TaquinGrid.EMPTY_SQUARE);
23          Assert.assertTrue(gridCreate.getSquare(8,9) == ancienneValeur);
24      }
```

Listing 4.1 – Test Moved()

Nous avons aussi réalisé le test de la méthode *finished*, permettant donc de savoir si la partie est terminée ou non. Une grille non mélangée est d'abord créée ainsi qu'une grille mélangée. La grille non mélangée doit alors être "terminée", donc doit valoir *true*. Pour la grille mélangée, elle est alors non "terminée" et donc doit valoir *false*. Nous savons alors que la méthode *finished* fonctionne.

```
1       public void finished() {
2           TaquinGrid gridCreate = new TaquinGrid(10, 10, false);
3           TaquinGrid gridRandomize = new TaquinGrid(10, 10);
4
5           Assert.assertTrue(gridCreate.finished());
6           Assert.assertFalse(gridRandomize.finished());
7       }
```

Listing 4.2 – Teste de la méthode finished()

Un troisième test est également effectué afin de vérifier le bon fonctionnement de la méthode *getSquare*. En effet, l'affichage de la grille est réalisée par la méthode *getSquare* pour connaître la valeur d'une case. Pour faire ce test, une nouvelle grille est alors créée, puis la valeur sur la dernière case de la grille est demandée. Cette valeur doit être égale au *TaquinGrid.EMPTY_SQUARE* (case vide).

```
1       public void getSquare() {
2           TaquinGrid gridCreate = new TaquinGrid(10, 10, false);
3           Assert.assertTrue(TaquinGrid.EMPTY_SQUARE == gridCreate.getSquare(9, 9))
        ;
4       }
```

Listing 4.3 – Teste getSquare()

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 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 :

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