

# Complément POO Rapport du Devoir

Beauchamp Aymeric 21301016  
Chagneux Dimitri 21606807  
Mori Baptiste 21602052  
Leblond Valentin 21609038

L2-Info-groupe-4A

# Table des matières

<b>Introduction</b>	<b>2</b>
<b>1 La conception du package model</b>	<b>2</b>
1.1 Organisation des classes . . . . .	2
1.2 Fonctionnement du modèle . . . . .	3
<b>2 Conception du package GUI</b>	<b>4</b>
2.1 Organisation des classes . . . . .	4
2.2 Fonctionnement GUI . . . . .	5
<b>Conclusion</b>	<b>6</b>

# Introduction

L'objectif de ce projet de POO est de réaliser un taquin : un casse-tête consistant à déplacer des cases d'un plateau afin de les replacer dans l'ordre et ainsi reconstituer une image ou un motif donné.

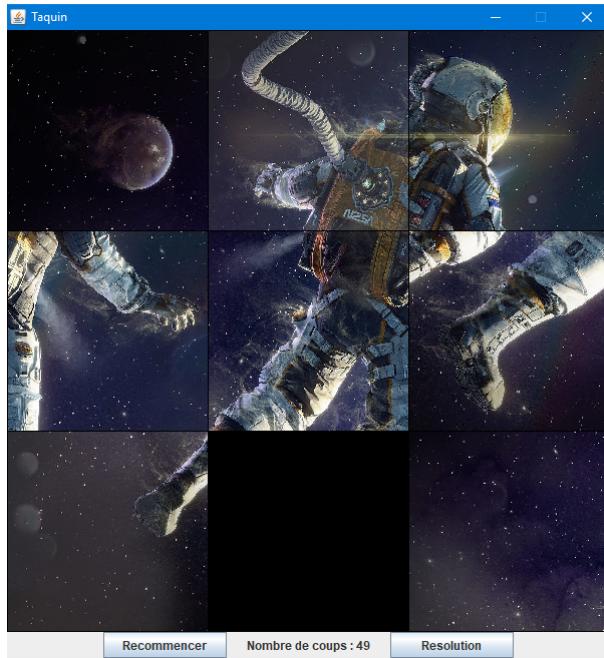


FIGURE 1 – Jeu du Taquin

Nous avons séparé le projet en deux packages, le premier comportant le modèle du jeu utilisable en version console (le package **model**), et le deuxième, l'implémentation de l'interface graphique (le package **GUI**). Nous utilisons également un troisième dossier **ressources** où sont stockées les images utilisées par l'application.

## 1 La conception du package model

Le modèle de base du taquin est constitué d'une grille d'objets représentant les différentes cases. Les cases pleines ont des identifiants pour permettre de mettre en place la condition de victoire : on gagne si chaque identifiant est placé aux bonnes coordonnées.

### 1.1 Organisation des classes

Tout d'abord, nous avons représenté les cases par deux classes, **FullTile** pour les cases pleines et **EmptyTile** pour la zone vide qu'on déplacera. Ces classes possèdent des attributs en commun qui sont les coordonnées X et Y dans la grille ; c'est pourquoi nous avons conçu une classe abstraite **Tile** qui possède ces coordonnées et dont héritent FullTile et EmptyTile. La classe FullTile diffère de EmptyTile en ce qu'elle possède en plus un identifiant entier que l'on rend unique pour chaque instance utilisée.

Ensuite, nous avons une classe **Board** qui décrit l'état du jeu et le fait évoluer.

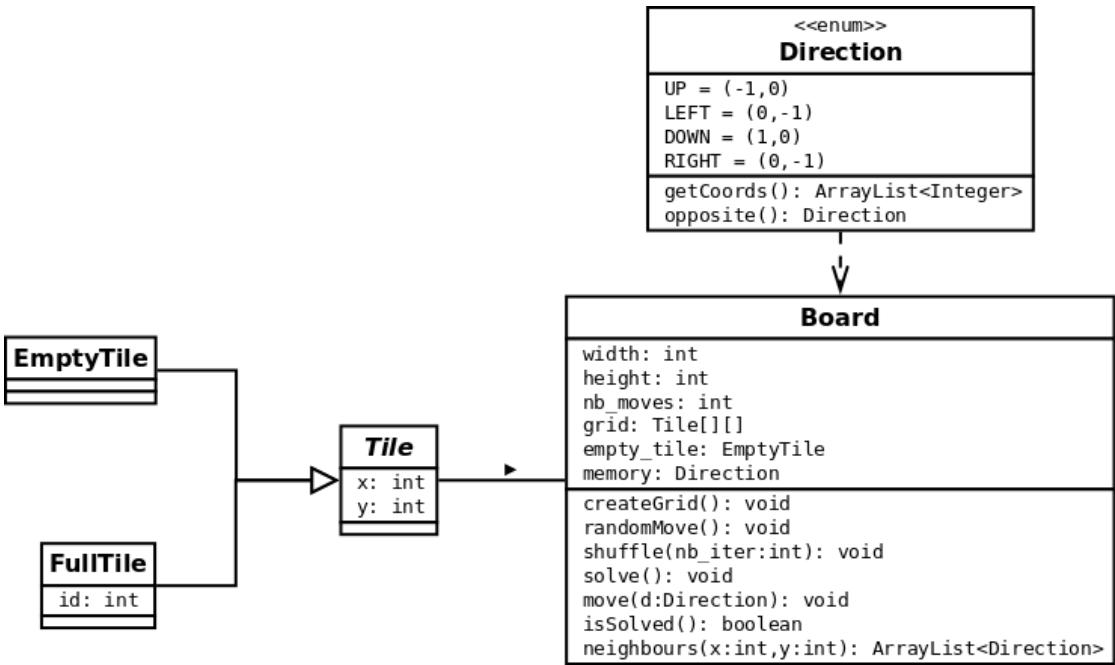


FIGURE 2 – Diagramme de classe du package model

## 1.2 Fonctionnement du modèle

Comme nous l'avons expliqué, l'initialisation et la mise à jour du modèle se font par le biais de la classe **Board**.

En premier lieu, nous avons une fonction **createGrid** qui permet d'initialiser une grille avec des **FullTile** et une **EmptyTile** et une fonction **toString** qui permet de l'afficher.

La grille ainsi créée est dans un état résolu où les identifiants des cases sont rangés dans l'ordre ci-dessous :

```
$ java model/Main
0 1 2
3 4 5
6 7
```

FIGURE 3 – Affiche d'une grille 3x3 initialisée

Puis, nous avons créé les déplacements de la case vide, nous avons d'abord commencé par créer une enum **Direction** avec quatre instances **UP**, **DOWN**, **LEFT**, **RIGHT** qui représentent une direction de déplacement. Nous utilisons cette enum dans la méthode **move** qui déplace la case vide dans la direction donnée en argument. Cette méthode vérifie également la validité du mouvement par rapport aux limites de la grille.

Pour mélanger le jeu, on choisit un mouvement aléatoire selon les mouvements possibles de la case vide puis on déplace la case vide jusqu'en bas à droite, à sa position initiale. On garde en mémoire le dernier coup effectué afin d'éviter des coups inutiles qui réduisent l'efficacité du mélange ; si l'on va vers le haut lors d'un coup, on interdit d'aller vers le bas au coup suivant. Le mélange est effectué par la fonction **shuffle** qui prend en argument le nombre de coups à réaliser avant de considérer le mélange comme terminé. Dans la version finale, nous faisons 10 000 mélanges. Cela assure une bonne dispersion des cases pour un temps de calcul négligeable. Comme la grille initiale est résolue, l'état obtenu par le mélange n'est pas bloquant puisqu'il existe une séquence de coups menant de cet état à la solution et qu'il est impossible au joueur

de bloquer le jeu en utilisant des déplacements valides.

La récupération des coups jouables de la case vide est un voisinage de taille 4 de rayon 1 qui correspond aux quatre directions haut, bas, gauche, droite. La fonction **neighbours** donne selon une coordonnée dans la grille du jeu toutes les directions possibles.

Le modèle dispose aussi d'une fonction **solve** qui résout le puzzle. Nous avons utilisé une approche extrêmement simple consistant à jouer un coup aléatoire jusqu'à obtenir la configuration initiale de la grille.

Cette simplicité a un coût : le solveur peut réaliser jusqu'à 500 000 coups pour résoudre un simple puzzle 3x3, et devient très lent dès que l'on veut augmenter la taille. Nous sommes conscients qu'il est possible de réaliser un solveur plus efficace, par exemple via l'utilisation de l'algorithme A\*, mais nous avons préféré ne pas nous focaliser sur cet aspect du projet.

La classe principale permet de jouer au taquin en interface console. Le joueur peut déplacer la case vide avec Z,Q,S,D.

## 2 Conception du package GUI

L'interface graphique est composée de deux fenêtres, l'une permettant de choisir l'image du jeu et l'autre affichant le jeu en lui-même.

### 2.1 Organisation des classes

L'interface graphique est composée de la vue **View**, du modèle **Board**, **Tile** et du contrôleur présent dans **Interface** (clics de souris et touches du clavier). La vue implémente **ModelListener**, ce qui permet de l'actualiser lorsque le modèle est modifié. Le modèle (ici le **Board**) hérite de **AbstractModeleEcouteable** qui permet d'ajouter ou supprimer des écouteurs mais aussi d'actualiser tous les écouteurs lors de l'exécution de la fonction **move**. Les contrôleurs lancent le mouvement du modèle qui va lui actualiser la vue.

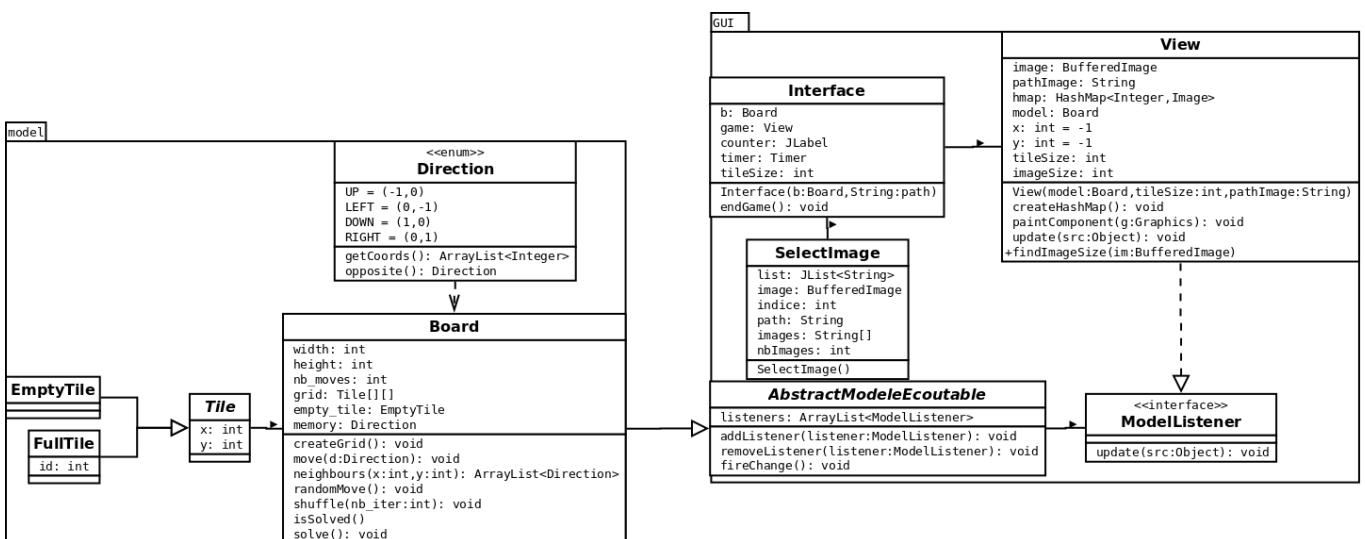


FIGURE 4 – Diagramme de classe MVC

## 2.2 Fonctionnement GUI

D'abord, pour représenter l'état du jeu, la classe **View** va, à partir d'un chemin, d'une image et d'un **Board**, découper l'image et lier pour chaque identifiant une partie de l'image qui lui correspond. La case vide ne possède pas d'identifiant, néanmoins elle a également une image qui est associée de façon à ce que l'image complète soit affichée lorsque le puzzle est résolu. On enlève aussi le quadrillage pour avoir une image plus esthétique.



FIGURE 5 – Visuel de la résolution du taquin

Pour choisir l'image qui est en fond, nous avons fait une autre interface qui va lister tous les éléments qui sont dans le dossier **ressources** (ne contenant que des images) en utilisant **JList**, **Vector** et **JScrollPane** pour gérer une plus grande quantité d'images. On ajoute dans le **Vector** les noms de fichier sans extension et on crée la **JList** avec le **Vector** en argument.

Ensuite, on crée le **JScrollPane** et on ajoute la **JList**. Pour intégrer le tout, on ajoute le **JScrollPane** de la même manière qu'un **JPanel** avec un **add** dans la **frame**. Quand on sélectionne une image dans la liste, elle s'affiche en grand à côté. Enfin, une fois l'image voulue sélectionnée, il suffit d'appuyer sur le bouton **Jouer** pour lancer l'interface du jeu avec l'image choisie en fond. On peut utiliser n'importe quelle image, cependant il est évident que plus l'image est petite par rapport à la vue, plus elle sera pixelisée en jeu.



FIGURE 6 – Images dans le dossier **ressources**

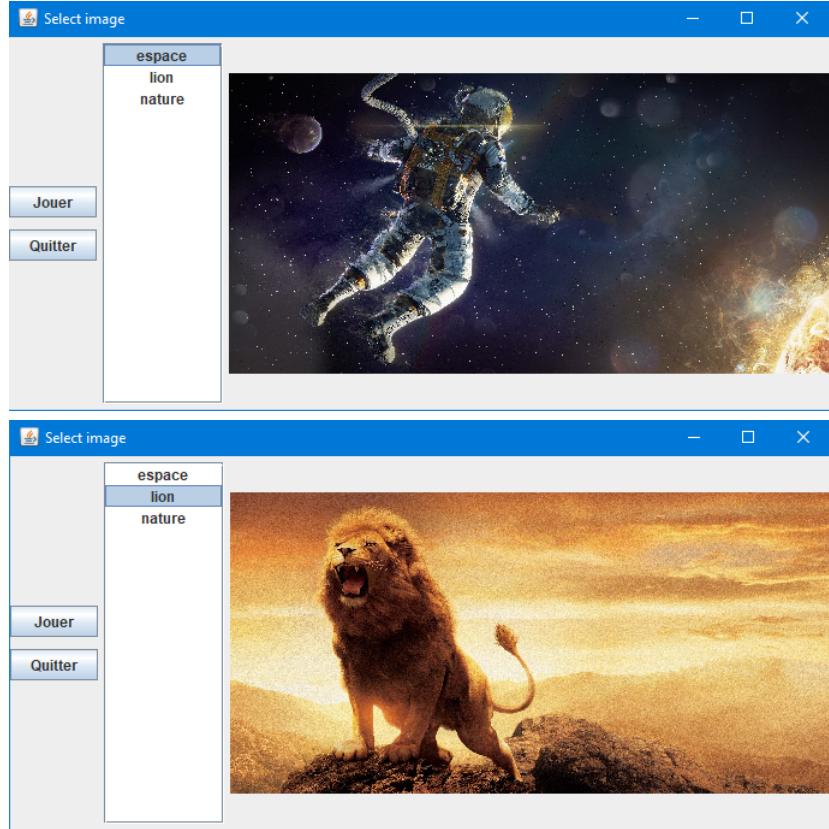


FIGURE 7 – Interface de sélection d'image

Les déplacements sont réalisables soit avec le clic de la souris sur une case pleine, soit avec les flèches directionnelles du clavier.

Pour le déplacement à la souris, on récupère la position du clic, on le divise par la taille des cases et on prend la partie entière. Ceci nous donne les coordonnées du clic dans la grille de jeu. Il nous reste maintenant à tester si le clic est valide. Pour cela, on soustrait les coordonnées de la case cliquée avec celles de la case vide pour obtenir le vecteur de déplacement ; si ce vecteur correspond à une instance de l'enum Direction, on effectue le mouvement, sinon on ne fait rien. Pour le déplacement au clavier, c'est plus simple, on teste quelle touche est pressée et on lance la fonction **move** du Board avec la direction adéquate en argument.

Deux boutons **recommencer** et **résolution** sont présents en bas de l'interface de jeu. Ils permettent respectivement de recommencer le jeu et de lancer le solveur. Pour recommencer le jeu, on a juste à mélanger le jeu. L'état obtenu est aussi soluble par propagation des propriétés énoncées en première partie.

## Conclusion

L'objectif du projet a été atteint, nous avons programmé le jeu du taquin jouable en console et avec une interface graphique et en suivant le modèle MVC.

Nous aurions pu améliorer ce projet en étoffant le solveur ou en proposant au joueur les dimensions de la grille de jeu - que nous avons mis par défaut en 3x3 - mais nous avons fait en sorte que le modèle et l'interface s'adaptent bien aux dimensions de la grille.

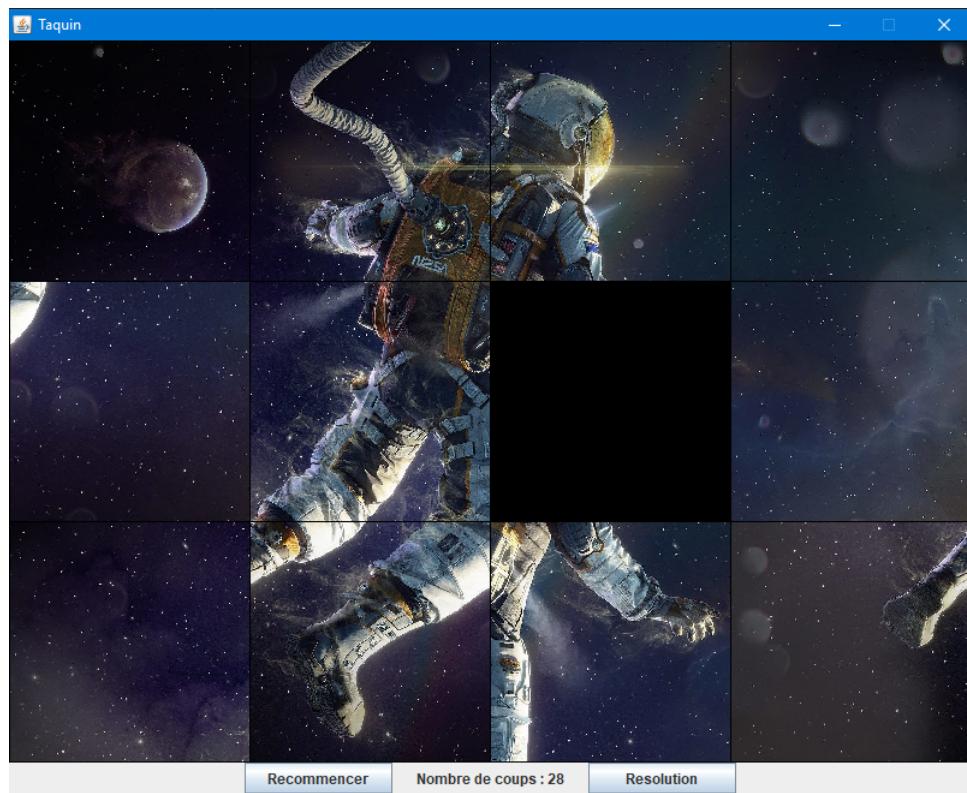


FIGURE 8 – Interface graphique avec une dimension 4x3