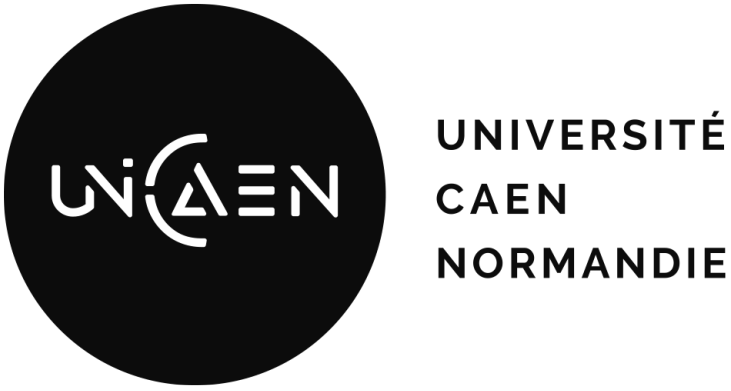
Université de Caen



Complément POO

L2 - Informatique

Taquin - DM

Auteur :
Nathan SAKKRIOU - 22003438
Yanis HABAREK - 22010593
Alan LECOT - 22003887

3 mars 2022

Taquin

Modèle

Comme le veut le principe d'un application architecturé en MVC, le modèle gère toute la partie logique de l'application.

Pour notre jeu, il faudra alors qu'il puisse gérer :

- La création de la grille
- Trouver les mouvements licites
- Effectuer un mouvement
- Si la grille est complétée

Création de la grille

Une grille prend évidement un nombre de ligne et un nombre de colonne. Elle sera représenté par une matrice de taille (`ligne x colonne`). Chaque case sera un chiffre, qui représente l'ordre dans lequel ils doivent être replacé (ils commencent par `1`).

La case vide sera représenté par le `0`, et placé initialement dans le coin en bas a droite.

Dans notre cas, cette action se passe dans le constructeur du modèle, car toute les actions citées précédemment s'appuie sur des interactions avec cette grille.

> Nous avons aussi fait le choix de surcharger le constructeur. Ce nouveau constructeur créera une grille de `3` lignes et `3` colonnes → Grille de base

Pour le taquin il faut dans l'ordre:

- Créer les cases
- Retirer le dernier coin
- Mélanger le tableau
- Ajouter le case vide

Pour simplifier la réalisation des deux premières étapes, nous avons fait le choix de tout gérer avec une liste simple (sur une ligne), dans laquelle nous ajoutons `nbLigne * nbColonne - 1` éléments, ce qui nous crée une liste d'éléments sans la dernière case.

Ensuite nous mélangeons cette liste grâce à la méthode statique

```
Collections.shuffle(ArrayList list);
```

Nous ajoutons ensuite la valeur `0` à la fin de la liste.

Nous avons terminé toutes les étapes cité plus haut, mais dans notre cas, étant donné que nous avons fait le choix de faire ces étapes préliminaires dans une liste simple, il nous suffit plus que de placer tout cela dans une matrice `ligne x colonne`

Algorithme utilisé :

```
LISTE matrice = []
INT compteur = 0
POUR i ALLANT DE 0 à nbLigne
            LISTE row = []

            POUR j ALLANT DE 0 à nbColonne
                row AJOUTE listeElementMelange[compteur]

                compteur ++

            matrice AJOUTE row
```

Nous obtenons une matrice de valeur mélangée, avec une case vide dans le coin en bas à droite.

Trouver les mouvements licites

Pour trouver les mouvements possibles, nous allons nous concentrer sur la case vide, car les mouvements ne peut venir que des cases adjacentes.

On compte 4 cas maximums :

- Il y a une case sur la même colonne mais sur la ligne du dessus
- Il y a une case la même colonne mais sur la ligne du dessous
- Il y a une case sur la même ligne mais sur la colonne de gauche
- Il y a une case sur la même ligne mais sur la colonne de droite

Il y a quelques subtilités à ces conditions :

1. La case vide se trouve sur une bordure de ligne
2. La case vide se trouve sur une bordure de colonne

Il faudra faire attention à bien regarder les déplacements possible de le cadre de la matrice.

Avec toutes ces informations il nous est facile d'écrire l'algorithme qui nous retourne les mouvements :

1. Trouver les coordonnées i (ligne) et j (colonne) de la case vide.

   > Un simple parcours de la liste en regardant la valeur de case suffit, les coordonnées sont ensuite stocké dans des variables

2. Créer un liste permettant de stocker tout les mouvements

3. Regarder les quatre cas

Dans notre programme nous avons décidé de représenter un mouvement de la façon suivante :

```
LIST [
        <valeur de la case qui peut être déplacée>,
        <coordonée i de la case>,
        <coordonée j de la case>,
        <coordonée i de la case vide>,
        <coordonée j de la case vide>,
      ]
```

Cela nous permet de conserver toutes les informations nécessaires au même endroit

A chaque comparaison, si celle-ci s'avère vraie, nous créons un mouvement et l'ajoutons dans la liste prévue à cet effet.

Cette méthode nous retourne une liste de mouvement facilement utilisable.

Effectuer un mouvement

La méthode "effectuer un mouvement" (`doMove()` dans notre code), prend en argument un mouvement.

Etant donné que toutes les informations nécessaire a une transposition entre une case pleine et la case vide sont stockés dedans, il est très simple de l'effectuer.

```
# Rappel de la structure d'un mouvement
LIST [
        <valeur de la case qui peut être déplacée>,
        <coordonée i de la case>,
        <coordonée j de la case>,
        <coordonée i de la case vide>,
        <coordonée j de la case vide>,
      ]

matrice[mouvement[1]][mouvement[2]] = 0
  -> Déplacement de la case vide

matrice[mouvement[3]][mouvement[4]] = mouvement[0]
  -> Déplacement de la case pleine
```

Si la grille est complétée

Dernière méthode charnière du modèle.

Ne devons pouvoir retourner si la grille est complétée, dans notre cas sous la forme du booléen.

Une grille est complétée si, en parcourant la matrice de droite a gauche et de haut en bas nous retrouvons une liste commençant par 1 et se terminant par `nbLigne * nbColonne-1`, si ce n'est pas le cas, ça veut dire que la grille n'est pas complétée

Pour cela nous parcourons la matrice (de la même manière que pour la crée précédemment), et nous comparons la valeur que nous regardons avec un compteur initialisé à 1, si les deux valeurs ne match

pas, nous retournons `false` sinon, nous continuons notre parcours en incrémentant le compteur de `1`.

Nous vérifions également une condition d'arrêt, car sans ça, même si la grille est complétée, il y aura toujours la case de valeur `0` en dernière position, et elle ne matchera évidement pas avec le compteur. nous vérifions alors si le compteur est égal à `nbLigne * nbColonne - 1`, et si c'est le cas, nous retournons `true`.

Une version en pseudo-code sera plus explicite :

```
INT compteur = 1
POUR i ALLANT DE 0 à nbLigne
      POUR j ALLANT DE 0 à nbColonne

            # Condition d'arrêt
            SI compteur == nbLigne * nbColonne - 1
            ALORS
                  break ou RENVOYER true
                  # Le language peut avoir besoin d'être sûr
                  d'avoir un retour

            SI matrice[i][j] != compteur
            ALORS
                  RENVOYER false

            compteur ++

RENVOYER true # Dans le cas cité dans la conition d'arrêt
```

Nous avons ici fait le tour des méthodes principales que nécessite un taquin.

Avec ces fonctionnalité il nous est possible de jouer en console, en implémentant des méthodes d'affichage est de demande de choix utilisateur.

Mais il nous est également possible de jouer avec une interface graphique. La vue et le controller ont tout ce qu'ils ont besoin pour afficher et jouer.

Vue et Controller

Pour pouvoir jouer nous allons avoir besoin de 2 vues :

1. Une "Home Page" permettant d'initialiser la grille avec les dimensions voulue par le joueur
2. Une interface de jeu → Une grille

Home Page

Comme explicité précédemment, notre modèle, pour être initialisé, demande 2 arguments :

1. Un nombre de ligne
2. Un nombre de colonne

L'objectif de cette vue sera de récupérer ces 2 informations afin de pouvoir initialiser un modèle et de le fournir à la prochaine vue.

Pour éviter tout problème nous avons fait le choix de lier ligne et colonne.

Pour ça nous avons fait le choix d'un `comboBox`, c'est le plus pratique pour des inputs de type `int`.

Coté controller, cela nous rend la tâches plus simple. Avec un `addActionListener` sur le bouton, et après une simple conversion `String → Integer`, le modèle peut être instancié peut plus de vérification sur les entrée, et l'interface de jeu peut être affichée.

Interface de jeu

Cette interface est totalement dépendante du modèle, elle demandera alors en argument, une instance du modèle, instance crée et passée dans le constructeur depuis le méthode du controller de Home Page.

L'interface sera composé de `nbLigne * nbColonne` boutons.

Le modèle à tout de fait pour nous. A l'aide d'un `Layout` sur notre `JPanel` de type `GridLayout`, nous fixons la taille de notre tableau. Il n'y plus qu'à créer un bouton par case, dans l'ordre que nous fournis le modèle et les placer.

Il ne suffit pas uniquement de placer les boutons, il faut maintenant les relier au controller afin d'effectuer un mouvement.

Tout les boutons, à un instant T, ne peuvent pas effectuer de déplacement, pour cela, nous avons désactiver les boutons qui ne sont pas présent dans les déplacement possible.

Nous avons décrit ici toute la vue.

Controller

D'après le cahier des charges, il nous ai demandé de pouvoir effectuer des mouvements de 2 façons :

1. A la souris
2. Au clavier

A la souris

Pour la souris c'est très simple, étant donné que l'on a mis un `ActionListener` uniquement sur les cases qui peuvent être déplacer, il nous suffit juste de comparer le numéro du bouton avec les numéros des coups jouables, et d'effectuer le mouvement correspondant.

Ensuite on actualise la vue, et on regarder si la position est gagnante où pas.

Au clavier

Dans ce cas, le problème numéro 1, c'est de savoir comment sont placées, par rapport à la case vide, les cases déplaçables.

La solution que nous avons trouvés, est celle de créer une méthode dans le modèle, nous permettant de la même manière que pour récupérer les mouvement licites, d'obtenir une listes de mouvement.

Mais la différence ici, est que la liste de mouvement sera toujours composée de 4 éléments, placés dans un ordre précis. Chaque emplacement correspond à une position :

- 0 → SUD → Flèche du haut
- 1 → NORD → Flèche du bas
- 2 → EST → Flèche de droite
- 3 → OUEST → Flèche de gauche

Les mouvements jouable, seront bien ajouté à leurs positions et complété de toutes les informations nécessaire (cf : composition d'un mouvement), et les mouvements non jouable seront composé uniquement de valeurs `null`.

Grâce cela nous savons quelles sont les mouvement jouable et quels sont leurs emplacements par rapport à la case vide.

Tout simple, sur le `KeyListener`, nous allons regarder si lors de la pression d'une touche directionnelle, le mouvement correspondant est différent de `null`. Si c'est le cas, nous effectuions le mouvement, actualisons, la vue, et regardons si le position est gagnante.