

Author : Christophe Garion <garion@isae.fr>
 Public : SUPAERO 2A
 Date :

Résumé

Le but de ce TP est de construire une interface graphique permettant de jouer à un jeu de morpion dont le modèle (au sens du patron de conception MVC) est fourni.

1 Objectifs

Les objectifs du TP sont les suivants :

- construire la vue de l'interface graphique du jeu de morpion ;
- utiliser les composants Swing pour construire la vue ;
- construire les contrôleurs nécessaires pour interagir avec le modèle.

2 La classe VueMorpion

Une vue de l'interface graphique vous est présentée sur la figure 1. Le diagramme de classe de VueMorpion est représenté sur la figure 2 (on remarquera que classiquement comme dans tout MVC la vue a une référence vers le modèle via une association).

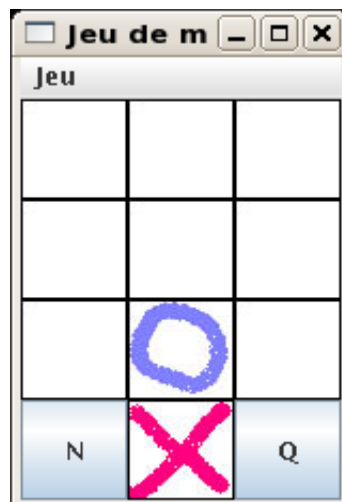


FIGURE 1 – Vue de l'interface

La classe VueMorpion hérite de la classe JFrame. On pourra donc accéder directement aux méthodes comme getContentPane() depuis les méthodes de VueMorpion.

Ses attributs (tous privés) représentent :

- trois constantes (donc déclarées **final static**) qui sont des instances de ImageIcon représentant les trois images disponibles sur le site (rond.jpg, croix.jpg et blanc.jpg) ;
- un menu Jeu qui comportera deux items "Nouveau" et "Quitter" ;
- une matrice d'instances de JLabel qui représente l'« aire » du jeu. Cette matrice sera une matrice 3*3 ;
- deux boutons "Quitter" et "Nouveau" ;
- un JLabel indiquant le joueur en cours (croix ou rond).

Le constructeur de VueMorpion devra :

1. construire le menu et le positionner sur la JFrame. Ce menu s'appelle Jeu et possède deux items Nouveau et Quitter ;
2. positionner un GridLayout sur la JFrame. Cette grille aura pour dimensions 4*3 (il faut une ligne supplémentaire pour les deux boutons et le label) ;
3. placer les JLabel contenus dans la matrice cases ;
4. placer le bouton Nouveau ;

5. placer le label indiquant le joueur en cours ;
6. placer le bouton Quitter ;
7. initialiser la vue grâce à la méthode recommencer ;
8. afficher la JFrame.

Les méthodes publiques de VueMorpion sont :

- recommencer() qui réinitialise le jeu. On affichera donc l'image blanc sur les JLabel de la matrice et le joueur courant (par défaut, le premier joueur sera toujours croix);
- setJoueurCourant(**boolean** b) qui affiche le « joueur » passé en paramètre. Par convention, si b est faux, alors le joueur est croix;
- setCase(**int** i, **int** j, **boolean** b) qui affiche sur une case en position (i,j) le joueur b (même convention que ci-dessus). Attention, on commencera numéroté les cases à partir de 0. Si on peut effectivement jouer sur la case, la méthode renverra **true** sinon **false** lorsque la case est déjà occupée.

Il faudra faire attention au positionnement des cases avec la méthode setCase (inversion des lignes et des colonnes).

Vous disposez d'un squelette de classe sur le site.

Vous testerez la classe VueMorpion en créant une instance de VueMorpion et en positionnant une image rond en position (1, 2). Lisez la section 5 pour avoir des détails concernant l'implantation, en particulier où placer les images dont vous avez besoin.

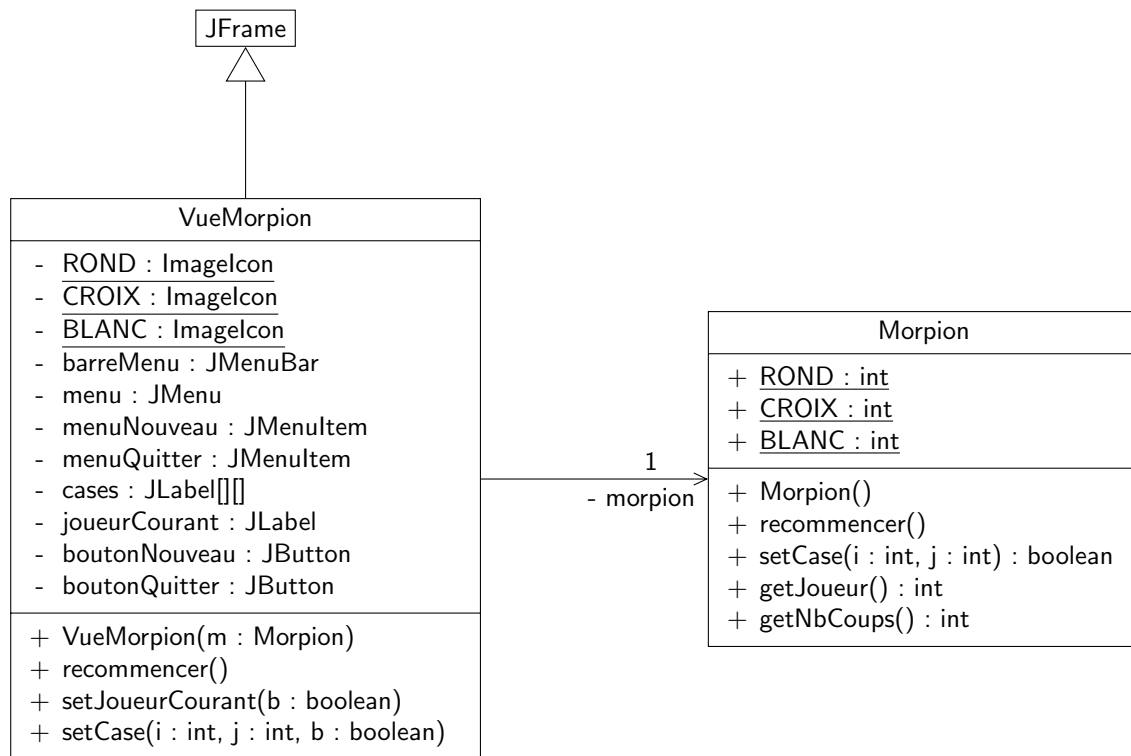


FIGURE 2 – Diagramme de classes du jeu de morpion

3 Le modèle du morpion

Le modèle du morpion est représenté par la classe Morpion que vous pouvez retrouver sur la figure 2. Vous trouverez également sur le site la documentation Javadoc complète de la classe.

4 Conception du contrôleur

Les événements que vous devez exploiter sur l'interface graphique sont les suivants :

1. l'utilisateur appuie sur le bouton « Quitter » ;
2. l'utilisateur appuie sur le bouton « Nouveau » ;
3. l'utilisateur utilise l'item « Quitter » du menu ;
4. l'utilisateur utilise l'item « Nouveau » du menu ;
5. l'utilisateur appuie sur une des cases.

La gestion de ces événements va se faire en Swing par l'intermédiaire de *listeners* que l'on va attacher aux composants de l'interface graphique `VueMorpion`.

Le fait d'appuyer sur un bouton ou sur un item d'un menu génère un événement de type `ActionEvent`. Cet événement peut être « intercepté » par un objet de type `ActionListener` associé au composant source de l'événement. `ActionListener` est une interface ne possédant qu'une seule méthode dont la signature est **public void actionPerformed(ActionEvent e)**. Cette méthode est appelée lorsque l'événement est généré par le composant associé au *listener*. Elle devra donc contenir le traitement nécessaire.

Comme `ActionListener` est une interface, il faut créer une classe réalisant cette interface (et donc implémentant la méthode `actionPerformed`) pour les différents événements que l'on veut traiter. A priori, les événements 1 à 4 vont faire appel à un `ActionListener`. On devrait donc créer quatre classes réalisant `ActionListener` pour traiter ces quatre événements. Or le traitement des événements 1 et 3 (resp. des événements 2 et 4) est le même : il s'agit de quitter l'application (resp. de recommencer une partie). On ne va donc créer que deux classes réalisant `ActionListener` :

- une classe `ListenerQuitter` dont la méthode `actionPerformed` va permettre de quitter l'application ;
- une classe `ListenerNouveau` dont la méthode `actionPerformed` va permettre de recommencer une partie.

On utilisera donc des classes internes non anonymes pour ces *listeners* dont les instances seront associées aux composants graphiques de `VueMorpion` par l'intermédiaire de la méthode `addActionListener`.

La question à se poser maintenant est la suivante : de quoi a-t-on besoin pour construire une instance de `ListenerNouveau` et de `ListenerQuitter` ?

Le traitement effectué par la méthode `actionPerformed` de `ListenerNouveau` doit réinitialiser le jeu de morpion. On doit donc faire appel à la méthode `recommencer` de `VueMorpion` et de `Morpion` (car le contrôleur doit gérer les interactions entre la vue et le modèle). Il faut donc que les instances de `ListenerNouveau` « connaissent » la vue et le modèle qui leur sont associés. Comme `ListenerNouveau` est une classe interne, elle a accès directement à la vue à laquelle elle est associée. De la même façon, elle a accès directement à l'attribut `morpion` de la classe `VueMorpion`. On n'a pas donc besoin d'ajouter des attributs dans `ListenerNouveau`.

Le traitement effectué par la méthode `actionPerformed` de `ListenerQuitter` doit simplement quitter l'application (cf. section 5). Les instances de `ListenerQuitter` n'ont donc pas besoin de connaître la vue et le modèle qui leur sont associés. Reste maintenant à gérer les événements de type 5. Ils se produisent quand l'utilisateur clique sur une des cases du morpion. Il s'agit donc d'événements de type `MouseEvent`. Ces événements peuvent être récupérés par un listener de type `MouseListener`. L'interface `MouseListener` possède cinq méthodes publiques (cf. cours). Parmi celles-ci, une seule nous intéresse : **public void mouseClicked(MouseEvent e)**. On va utiliser la classe `MouseAdapter` qui réalise `MouseListener` pour n'avoir qu'à redéfinir la méthode `mouseClicked`. Nous allons utiliser une classe interne `ListenerCase` spécialisant `MouseAdapter`. Chaque instance de cette classe anonyme devra être associée à une case bien précise. Il faut donc qu'elle connaisse la colonne et la ligne de la case. On va donc ajouter deux attributs entiers à la classe. La vue sera accédée directement par la classe `ListenerCase` et le modèle via l'attribut `morpion` de la vue.

Le diagramme de classe final est représenté sur la figure 2. Les classes internes n'y sont pas représentées.

5 Implantation

Quelques précisions pour l'implantation :

- l'instruction permettant d'arrêter l'application est `System.exit(0)` ;
- pour afficher le résultat du jeu, on pourra utiliser la méthode *statique* `showMessageDialog` de la classe `JOptionPane` :

```
showMessageDialog(Component c, Object message, String titre,  
                  JOptionPane.INFORMATION_MESSAGE, ImageIcon i);
```

où :

- `c` est le composant devant contenir le panneau de dialogue (ici `null`) ;
- `message` est l'objet à afficher (on utilisera donc un objet de type `String`) ;
- `titre` est le titre du panneau de dialogue ;

- `i` est l'image à afficher dans le panneau.
- les images doivent être normalement placées dans le CLASSPATH. On peut donc placer les images individuellement dans le répertoire contenant les *bytecodes* (attention, il faut les placer dans le répertoire de plus haut niveau, `bin` par exemple) ou ajouter le jar fourni dans le CLASSPATH via les options du projet.