Interfaces graphiques avec Swing et UML (programmation événementielle)
Corrigé

Exercice 1 : Morpion et UML

L'objectif de cet exercice est d'utiliser UML pour modéliser le jeu du Morpion.

Le jeu de Morpion est un jeu à deux qui se joue sur une grille de taille 3 cases par 3. Les cases de la grille sont initialement vides. Chaque joueur joue en alternance en inscrivant un symbole dans une case vide (l'un des joueurs est représenté par un anneau, l'autre joueur par une croix). Ce sont les croix qui commencent. Le jeu s'arrête dès que l'un des joueurs aligne trois symboles identiques (horizontalement, verticalement ou en diagonale) ou lorsque toutes les cases sont occupées.

1.1. Proposer un diagramme des cas d'utilisation. On commencera par rappeler les principaux éléments d'un diagramme de cas d'utilisation.

Solution :

Les principaux éléments d'un diagramme de cas d'utilisation sont :

1. le cas d'utilisation (ovale) : une fonctionnalité de l'application, un service rendu à un acteur du système.
2. l'acteur (petit bonhomme) : un rôle (personne, élément matériel, autre système informatique, etc) qui interagit directement avec le système.
3. les relations entre acteur et cas d'utilisation (quel acteur est impliqué dans quel cas d'utilisation ?) et les relations entre cas d'utilisation (héritage, «include», «extends»).

Une manière de construire le diagramme de cas d'utilisation est la suivante :

1. Identifier les acteurs : rôles qui ont une interaction directe avec le système
2. Identifier les cas d'utilisateur de chaque acteur : un cas d'utilisation correspond à un service que le système rend à cet utilisateur
3. Compléter les acteurs en listant, pour chaque cas d'utilisation, les autres acteurs concernés par ce cas.
4. En dernier lieu, identifier les relations entre cas d'utilisation. Attention, la relation «include» est à utiliser avec modération car elle risque à faire une décomposition fonctionnelles du système (en genre de raffinage) alors que chaque cas d'utilisation doit être déclenché par un acteur.

   Ici, on pourrait faire apparaître le cas d'utilisation « Cocher une case » qui serait déclenché par un joueur. Il y aurait alors une relation de type *include* entre « Jouer » et ce cas.

Cependant, ce qu'attend vraiment le joueur, c'est de jouer, pas seulement de cocher une case. On constate bien qu'ici on fait une décomposition fonctionnelle du case « jouer ». Ce n'est pas le but du diagramme de cas d'utilisation.

Si ce n'est déjà fait, essayez de construire le cas d'utilisation avant de lire la suite....

Ici, nous identifions l'acteur Joueur (qui peut être spécialisé en Joueur Croix et Joueur Rond). Le premier service que le jeu du Morpion lui rend, c'est justement de « jouer une partie ». Pendant cette partie, il peut la « quitter » ou la « recommencer » (i.e abandonner et recommencer une nouvelle partie). Ces deux derniers cas sont très basiques ici. Ils ne mériteraient certainement pas l'appellation de cas d'utilisation dans un vrai système. Notons que « quitter » et « recommencer » se fait pendant que la partie se joue, d'où les relations «extends» (à ne pas confondre avec l'héritage de Java !).

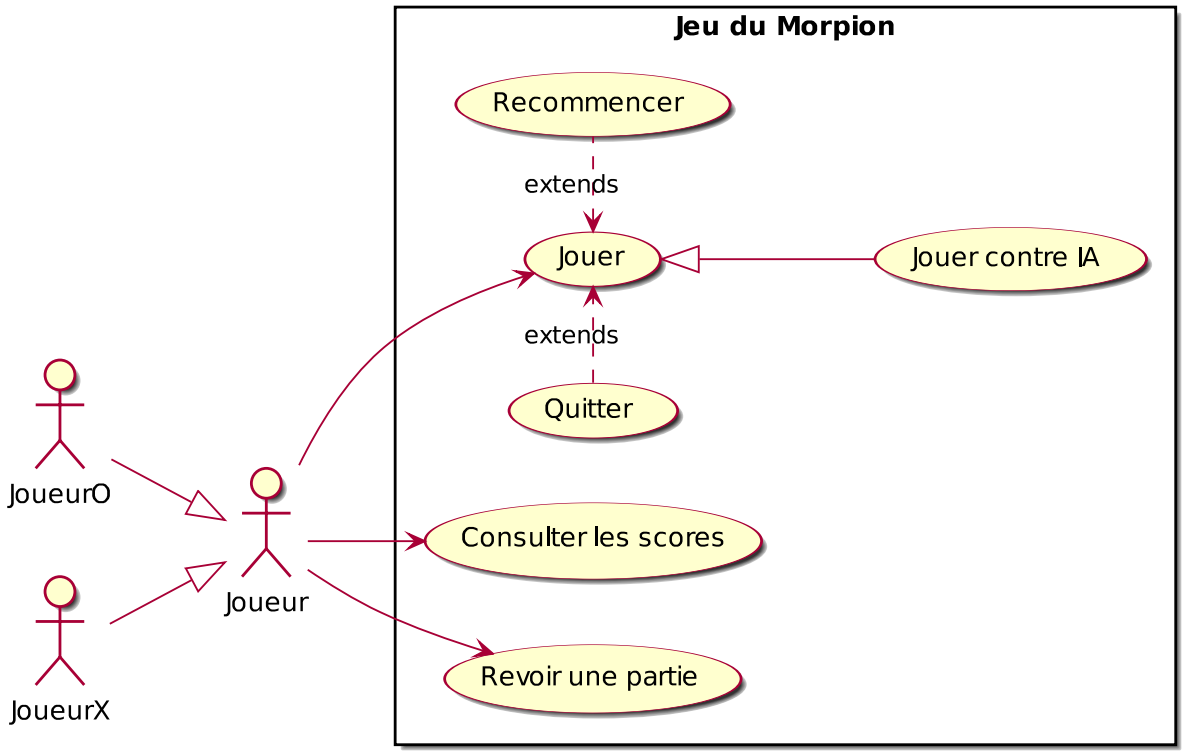
Le joueur peut souhaiter jouer contre la machine. On peut donc identifier un nouveau cas d'utilisation « Jouer contre IA » qui spécialise le cas d'utilisation « Jouer » (là c'est bien le sous-typage vu en programmation objet avec Java).

On peut envisager d'autres fonctionnalités comme par exemple avoir accès aux scores (« consulter les scores ») qui intéresse le joueur. Ceci signifie que l'application devra totaliser les parties gagnées par les croix, par les ronds et les partie nulles.

Le joueur pourrait aussi souhaiter pouvoir revoir une partie (les cases successivement cochées alternativement par les joueurs)...

Notons qu'il n'y a pas de relation entre ce cas d'utilisation et les autres. Cependant, c'est bien le déroulement des autres cas d'utilisation (Jouer, Recommencer) qui permettront de connaître le nombre de parties gagnées, de parties nulles, de parties abandonnées. Ce sont ces informations qui seront fournies par « Consulter les scores ».

On peut alors dessiner le diagramme de cas d'utilisation suivant.



1.2. Dessiner un diagramme de séquence contextuel qui montre le déroulement d'une partie. Il devra montrer que les règles du jeu doivent être respectées.

Solution :

On peut prendre une convention pour numéroter les cases de la grille :

```
    A   B   C
  +---+---+---+
3 |   |   |   | 3
  +---+---+---+
2 |   |   |   | 2
  +---+---+---+
1 |   |   |   | 1
  +---+---+---+
    A   B   C
```

Le diagramme de séquence suivant représente plusieurs aspects qu'il serait préférable de montrer sur des diagrammes différents :

— Chaque joueur joue à son tour.

— On ne peut pas cocher une case déjà cochée.

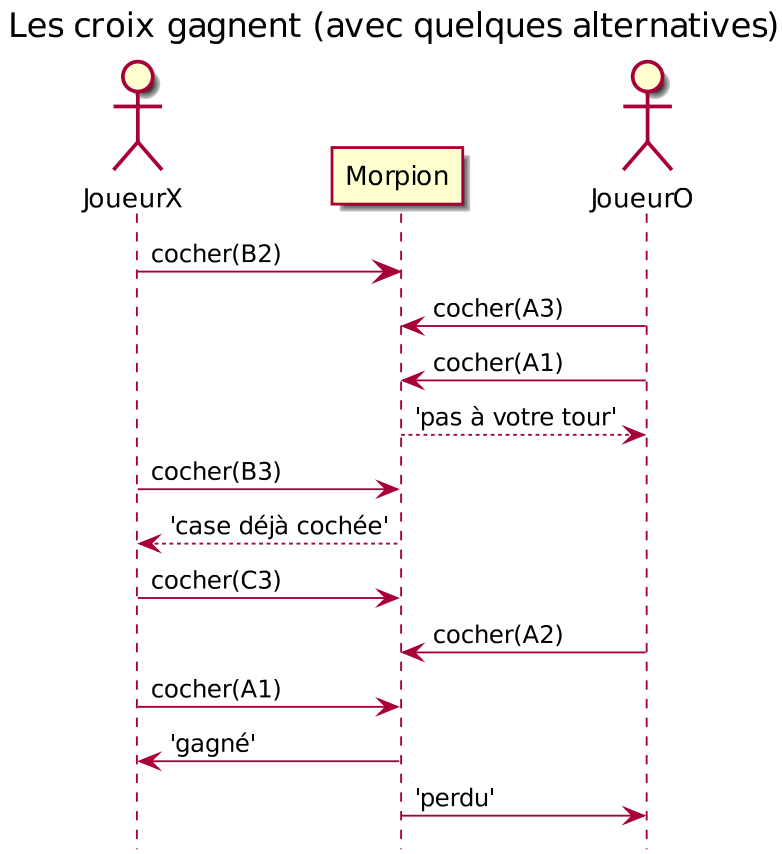
— La partie se termine dès que trois symboles identiques sont alignés

Il faudrait dessiner d'autres diagrammes pour envisager les scénarios suivants :

— les ronds gagnent

— il y a match nul

— les ronds ou les croix gagnent en posant le 9^{e} jeton



1.3. Lister les événements extérieurs, ceux déclenchés par l'utilisateur de l'application.

Solution : Les événéments utilisateurs sont :

- cocher une case de (i, j) ;
- recommencer la partie ;
- quitter l'application.

1.4. Dessiner un diagramme d'état du jeu de Morpion.

Solution :

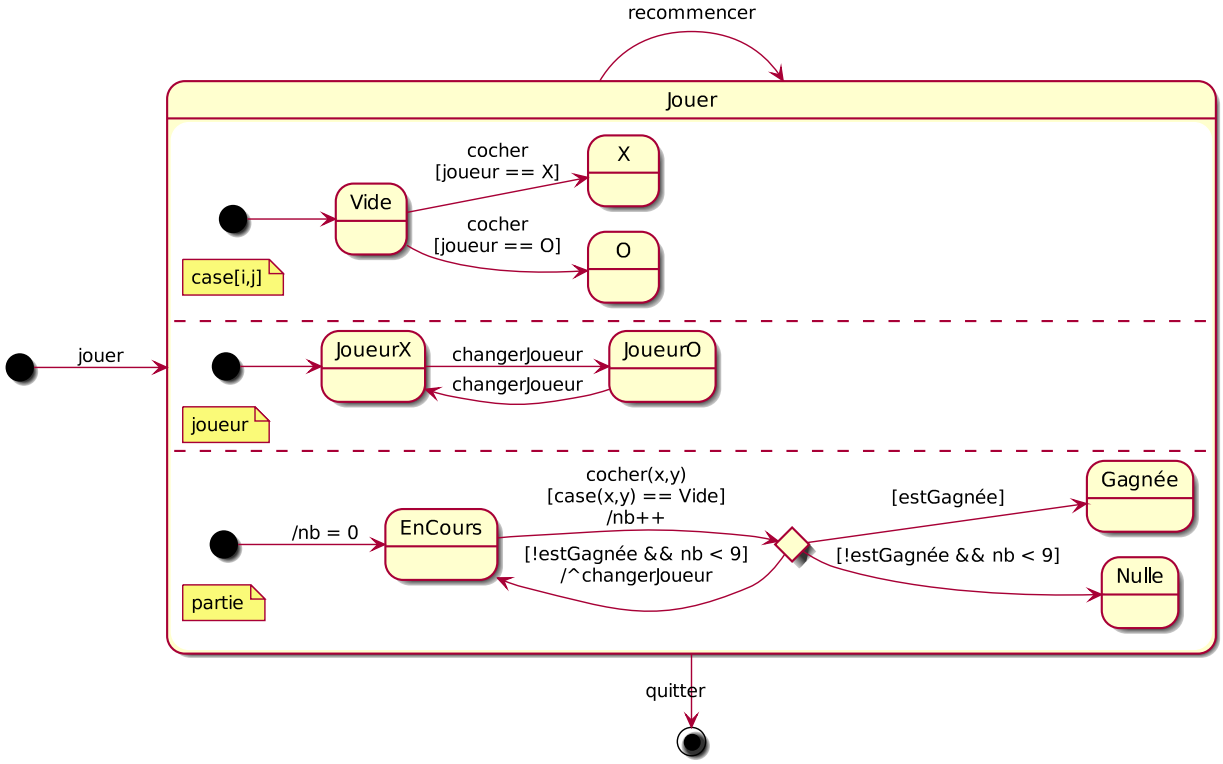
Pour dessiner le diagramme d'état du jeu du Morpion, on peut regarder les « objets » qui le composent. Le diagramme peut alors contenir autant de régions que d'objet, chaque région contenant le diagramme de machine à états de cet objet.

Quels sont les objets ?

Ici les objets sont les cases (il y en a 9 mais qui ont le même comportement), le joueur suivant (celui à qui c'est le tour de jouer) et la partie elle-même ?

Maintenant on devrait pouvoir dessiner le diagramme de machine à états de chacun de ces objets et ainsi en déduire le diagramme complet...

Voici le diagramme de machine à états complet.



Remarque : L'outil utilisé, plantuml, limite ce que l'on peut faire. En particulier, il serait plus logique d'avoir deux régions à gauche pour les cases et le joueur et une région à droite pour la partie.

Le losange désigne un pseudo-état dans lequel une décision sera prise. Les transitions qui en partent ne contiennent qu'une garde, pas d'événement car l'événement était sur la première partie de la transition. Ceci permet de factoriser l'événement, sa garde et son action qui auraient été dupliqués si on avait fait 3 transitions.

Notons qu'on aurait pu utiliser cette notation pour une case et ainsi factoriser l'événement « cocher ».

De plus, « évaluation » est représenté par un état alors qu'il pourrait être représenté par une décision (losange).

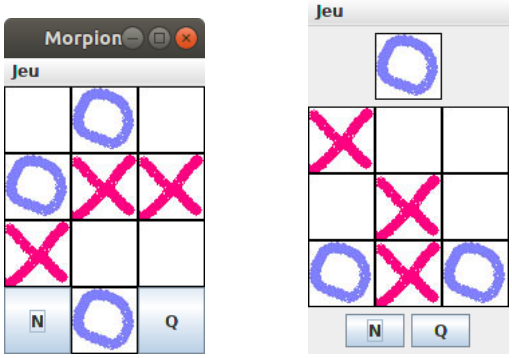
Exercice 2 : IHM graphiques pour le jeu du Morpion

On envisage deux IHM (Interface Homme Machine) pour le Jeu du Morpion présentées figures 1(a) et 1(b).

2.1. Indiquer les composants graphiques à utiliser.

Solution : Il faut représenter la surface de jeu, les 9 cases à cocher. On peut donc prendre un tableau à deux dimensions. Reste à choisir ce que l'on prend pour représenter une case. On peut prendre un JLabel avec une image. On pourrait aussi prendre un JButton, mais on ne veut pas forcément avoir l'effet d'un bouton.

Il faut représenter un bouton pour quitter et un autre pour démarrer une nouvelle partie.



(a) version 1 (b) version 2

On fait aussi apparaître le joueur suivant (le joueur à qui c'est le tour de jouer). On utilise un JLabel.

2.2. Expliquer comment construire la partie présentation des deux IHM envisagées.

Solution : Pour la version 1, on remarque que les 12 composants ont tous les mêmes dimensions. On peut donc prendre un GridLayout(4, 3), 4 lignes de trois composants et on mettra dans l'ordre les JLabel des cases, le bouton Nouvelle partie, le JLabel du joueur courant et le bouton Quitter.

```
contentPane --- GridLayout(4, 3)
    +- les 9 JLabel des cases
    +- le JButon du bouton Nouvelle partie
    +- le JLabel du joueur courant
    +- le JButon du bouton Quitter
```

Pour la version 2, on remarque que la grille est dans la partie centrale, en bas on a les boutons pour recommencer ou quitter et en haut le symbole du joueur courant. On peut donc prendre un BorderLayout pour le volet principal de la fenêtre. Pour la grille, les boutons et le joueur suivant, on prendra un container (JPanel) avec comme gestionnaire de placement un GridLayout et deux FlowLayout.

```
contentPane --- BorderLayout
    +---CENTER--- JPanel --- GridLayout(3, 3)
        +- les 9 JLabel des cases
    +---NORTH---- JPanel --- FlowLayout
        +- le JLabel du joueur courant
    +---SOUTH---- JPanel --- FlowLayout
        +- le JButon du bouton Nouvelle partie
        +- le JButon du bouton Quitter
```

Exercice 3 : Différentes manières de réaliser les réactions du Morpion

Plusieurs solutions sont possibles pour programmer les réactions dans le jeu du Morpion. L'objectif de cet exercice est d'identifier ces solutions et d'indiquer leurs avantages et inconvénients.

3.1. Considérons d'abord les deux boutons Quitter et Recommencer. Expliquer comment il serait possible d'implanter les réactions correspondantes.

Solution : On peut écrire un classe (ActionListener) par bouton (donc réaction) et définir la méthode `actionPerformed` ou une seule. Dans ce dernier cas, il faut pouvoir identifier le bouton cliquer. On peut s'appuyer sur la méthode `getSource()` de l'événement de type `ActionEvent` en paramètre de `actionPerformed`. Si on veut associer la même réaction à un autre bouton ou une entrée de menu, il faut multiplier les tests sur la source. Une solution plus générale est d'utiliser les méthodes `get/setActionCommand` des `JButton` (qui existent aussi sur les entrées de menu).

3.2. Les cases de l'aire de jeu étant des JLabel, la réaction doit être définie comme un MouseListener. Discuter les avantages et inconvénients à utiliser MouseListener ou MouseAdapter.

Solution : `MouseListener` est une interface qui définit 5 méthodes. `MouseAdapter` est une classe qui réalise `MouseListener` et définit ses 5 méthodes avec un code par défaut vide. `MouseAdapter` est une classe abstraite même si toutes ses méthodes sont définies car il n'y a pas d'intérêt à avoir un `MouseListener` qui ne fait rien. Il faut au moins définir une de ses 5 méthodes !

Dans notre cas, une seule méthode nous intéresse :

```
public void mouseClicked(MouseEvent ev);
```

Si on décide de réaliser l'interface, nous serons obligés de définir les 5 méthodes même s'il n'y en a qu'une qui nous intéresse. Au donnera un code vide aux autres méthodes. Ceci est un peu fastidieux même si des outils comme Eclipse permettent d'engendrer automatiquement le squelette des méthodes abstraites et simplifient le travail. Si on oublie de définir l'une des méthodes, le compilateur nous dira que la classe doit être déclarée abstraite car elle contient des méthodes abstraites.

Comme nous souhaitons définir qu'une seule méthode de `MouseListener`, on peut donc préférer hériter de `MouseAdapter` et ne redéfinir que cette seule méthode, les 4 autres conservant le code vide défini dans la super-classe. Il est important d'utiliser l'annotation `@Override` pour que le compilateur puisse vérifier que l'on a bien redéfini la méthode de la super-classe. Par exemple, ceci permettra de détecter l'erreur suivante :

```
@Override public void MouseClicked(MouseEvent ev) {
    ...
}
```

ou celle-ci :

```
@Override public void mouseClicked(ActionEvent ev) {
   ...
}
```

Quelles sont les erreurs ? La majuscule au début du nom de la méthode pour la première, le type du paramètre pour la seconde. Ce ne sont donc pas des redéfinitions mais la définition de nouvelles méthodes qui ne seront jamais appelées quand on cliquera sur le JLabel !

Sans `@Override`, on pourra chercher longtemps avant de trouver l'erreur.

3.3. Lorsque le joueur clique dans une case, il faut que dans la réaction on puisse retrouver les coordonnées de la case cliquée (ceci peut être nécessaire pour déterminer de manière efficace si la partie est terminée). Expliquer comment connaître cette information.

Solution : Voici les solutions que l'on peut envisager :

— Faire un calcul par rapport aux coordonnées de la souris. Attention, il faut associer le Listener au container des JLabel et non à chaque JLabel car la position de la souris est relative au composant qui a intercepté le clic.

Il faut faire attention lors du calcul aux dimensions de la fenêtre car elle peut avoir été agrandie ou diminuée.

Cette solution est donc assez fastidieuse mais efficace quand le nombre de case est très grand.

— Utiliser la méthode `getSource` du paramètre de `mouseClicked` pour retrouver le JLabel cliqué et ensuite le chercher dans le tableau des cases. Ceci fonctionne mais n'est pas efficace quand le tableau est grand !

— Ajouter dans le listener les informations manquantes. Elles sont stockées dans des attributs qui sont initialisés grâce au constructeur. Lors de la création du listener, on indique la ligne et la colonne des cases contrôlées.

L'inconvénient est qu'il faut autant d'instances du Listener que de cases dans le jeu !

— Pour éviter le défaut précédent, on peut mettre les informations du côté des cases. Il suffit alors de spécialiser un JLabel en Case, nouvelle classe, et d'ajouter deux attributs pour les coordonnées. On a alors un seul Listener qui va aller chercher les coordonnées dans les attributs de getSource (il faudra bien sûr le transtypé en Case).

Remarque : C'est un mécanisme similaire à l'« *actionCommand* » défini sur les `AbstractButton` et donc les `JButton`, mais qui n'existe pas pour les `JLabel`.

— Utiliser `getActionCommand` et `setActionCommand` pour coder les coordonnées, par exemple sous la forme "cocher ixj". Ceci ne peut pas fonctionner avec un `JLabel` qui ne définit pas « *actionCommand* ».

Il faut faire le codage et le décodage ce qui est source d'erreur et prend du temps. La bonne solution dans cette optique est de spécialiser le JLabel en Case pour ajouter les bonnes informations avec le bon type (solution précédente) plutôt que de faire un codage/une sérialisation ad'hoc.

— On peut utiliser une classe anonyme. L'intérêt est d'éviter de transmettre les coordonnées de la la case au Listener qui utilisera directement les variables des deux boucles for qui permettent de parcourir les JLabal. Cependant, une classe interne ne peut pas utiliser une variable modifiée. Il faudra donc déclarer deux variables locales déclarées finales (optionnel depuis Java8) initialisées avec les variables de boucles des for.