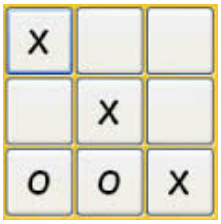
Conception et programmation orientées objets avancées
L'architecture modele-vue-controleur (2 séances)

Problème

Concevoir une application arbitrant une partie de *morpion* en utilisant l'architecture modèle-vue-contrôleur. L'application sera dotée d'une interface textuelle (nous n'utiliserons pas le design pattern *composite* pour cette interface).

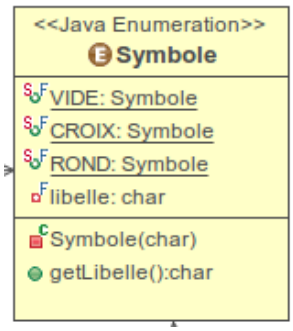
Règles du morpion

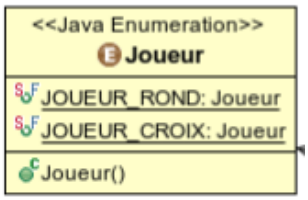
Il s'agit d'un jeu à deux joueurs, que nous nommons *JOUEUR_CROIX* et *JOUEUR_ROND*, qui doivent à tour de rôle déposer un symbole, une *CROIX* ou un *ROND*, dans une case encore *VIDE* d'une grille carrée (*JOUEUR_CROIX* commence). Le premier joueur qui remplit une ligne, une colonne ou la diagonale de la grille avec tous ses symboles a gagné ainsi que l'illustre la figure ci-dessous :



UML

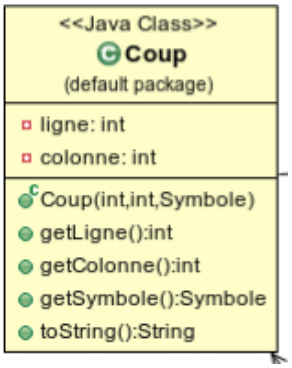
On dispose d'une énumération (cf annexe) *Symbole* pour représenter les 3 états possibles d'une case de la grille et d'une énumération *Joueur* pour représenter les deux joueurs d'une partie :



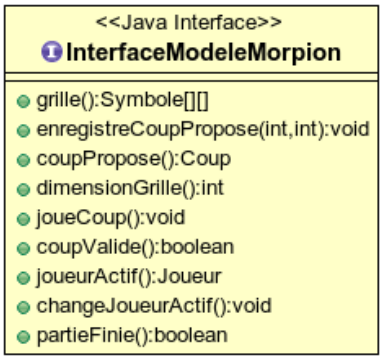


Modèle (gestion d'un jeu de morpion)

Soit la classe *Coup* utilisée pour enregistrer le dernier coup joué : un *Coup* est défini par une *ligne* et une *colonne,* déterminant la case jouée, et un *Symbole* dont la valeur est soit *Symbole.CROIX* soit *Symbole.ROND*.



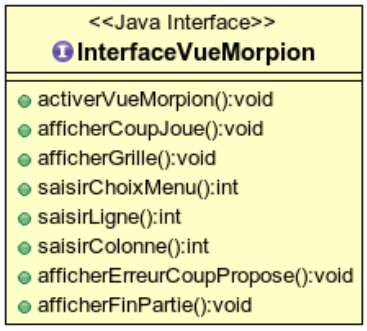
Soit l'interface *InterfaceModeleMorpion* :



- *enregistreCoupPropose(int ligne, int colonne)* enregistre dans le modèle le coup proposé par l'utilisateur (le symbole joué sera automatiquement déterminé selon le joueur actif).
- *coupPropose*() retourne le coup proposé qui a été enregistré.
- *coupValide*() détermine si le coup proposé enregistré est valide.
- *joueCoup*() joue le coup proposé en modifiant l'état de la grille en conséquence.
- *partieFinie*() détermine si une situation de fin de partie est apparue après le dernier coup joué.
- *joueurActif()* : retourne *Joueur.JOUEUR_CROIX* ou *Joueur.JOUEUR_ROND* représentant le joueur qui doit jouer.
- *changeJoueurActif()* : change le joueur actif (*Joueur.JOUEUR_CROIX* devient actif si *Joueur.JOUEUR_ROND* vient de jouer et inversement).
- *grille()* : retourne la grille courante.
- *dimensionGrille()* : retourne le nombre de lignes (et de colonnes) dans la grille.

Vue (interactions avec l'utilisateur)

Soit l'interface *vue* :



- *activerVueMorpion()* permet de lancer la vue qui gère les interactions avec l'utilisateur.
- *saisirChoixMenu()* demande à l'utilisateur d'entrer un numéro correspond à l'option choisie dans le menu .
- *saisirLigne()* demande à l'utilisateur un numéro de ligne.
- *saisirColonne()* demande à l'utilisateur un numéro de colonne.
- *afficherGrille()* affiche la grille du morpion associé à la vue.
- *afficheErreurCoupPropose()* affiche un message d'erreur quand le coup proposé est invalide.

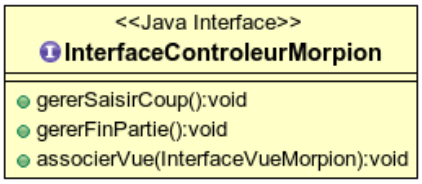
- *afficheFinPartie()* affiche un message lorsque la partie est finie.

exemple d'affichage proposé à l'utilisateur :

```
Grille courante :

   0 1 2
0 |O|-|-|
1 |-|X|-|
2 |-|-|-|

c'est a X de jouer :

1 - entrer un coup
2 - sortir du jeu


entrer votre choix (puis Entree)
```

Contrôleur (interprétation des services à l'utilisateur proposés par la vue)

Soit l'interface suivante où gererSaisirCoup() et *gererFinPartie()* contiennent les interprétations des choix *"entrer un coup"* et *"fin de partie"* que la vue propose à l'utilisateur :



associerVue(InterfaceVueMorpion) : void permet d'associer une vue au contrôleur.

1) Créer le diagramme des classes du Morpion selon l'architecture MVC (ajouter tous les éléments nécessaires).

2) L'opération *gererSaisirCoup()* procède selon l'algorithme suivant :
   Début
   - saisir la ligne et de la colonne
   - enregistrer le coup proposé
   - **si** le coup proposé est valide
     - jouer le coup
     - **si** la partie est finie
       - afficher la fin de partie
       - (sortir du programme).
     - **sinon** (partie pas finie) : changer le joueur actif.
   - **sinon** (coup invalide) : afficher un message d'erreur (coup invalide)

   Fin

   Écrire le diagramme de séquences correspondant (identifier les objets et les messages).

Java

Traduire l'ensemble des classes et les tester dans une classe *TestMorpion* (la classe *VueMorpionTextuel* est à compléter pour tenir compte de l'affichage du menu, de la gestion des choix de l'utilisateur relatif à ce menu, et de la sortie du programme).

Procéder dans l'ordre suivant :

1) faire le *modèle* et le tester entièrement (sauf la partie observable)

2) faire la *vue* et la tester (sauf la délégation au *contrôleur*)

3) faire le *contrôleur* et le tester avec la *vue*.

4) Implémenter la partie *observateur-observé* entre le modèle et la vue, et la tester.

Conseil :

- IL NE FAUT PAS ECRIRE LE MODELE, LA VUE ET LE CONTROLEUR ET TESTER L'ENSEMBLE APRES. IL EST IMPERATIF DE TESTER CHAQUE ELEMENT UN PAR UN COMME INDIQUÉ CI-DESSUS.
- employer l'accesseur *private* pour toute méthode d'une classe qui n'a pas vocation à être utilisée à l'extérieur de cette classe.

Question facultative

Créer une vue graphique du *morpion* (cf cours IHM) et un *contrôleur* associé. Modifier la classe *TestMorpion* en conséquence.

ANNEXE

Énumération

Une énumération représente une liste de valeurs, par exemple *{VIDE, CROIX, ROND}*. En java le type *enum* permet de gérer des énumérations.
ex)

```
public enum Symbole
{VIDE, CROIX, ROND}
```

définit une énumération de nom *Symbole* possédant les valeurs énumérées *VIDE, CROIX* et *ROND* (par convention les noms des valeurs sont écrits en majuscules).
Noter qu'une énumération est une classe et que les valeurs énumérées sont des objets pré-définis: ce sont les seuls objets possibles pour cette classe. Ici les objets (valeurs) de la classe *Symbole* n'ont pas d'attribut, seul compte le nom de ces objets.

Exemple d'utilisation d'une énumération :

```
{
      Symbole s1 = Symbole.CROIX;
      Symbole s2 = Symbole.ROND ;

      System.out.println(s1) ;            // affiche : CROIX
      System.out.println(s2) ;            // affiche : ROND

      if (s1 == s2)
            System.out.println("symboles identiques") ;
}
```

On voit que :

- Une variable de type *Symbole* référence UNE valeur (un objet) de l'énumération.
- On accède à une valeur (un objet) d'une énumération en la préfixant par le nom de l'énumération *(Symbole.CROIX)*
- On compare 2 valeurs d'une énumération en utilisant l'opérateur ==. Les valeurs d'une énumération étant des objets on pourrait les comparer par *equals()* avec le même résultat. Cependant l'usage est d'utiliser == car deux valeurs égales représentent le même objet. De plus la comparaison avec == est recommandée : il y a une vérification des types comparés à la compilation, et la comparaison avec la valeur *null* est valide.

Une énumération étant une classe elle peut être munie d'attributs, de constructeurs et d'opérations. ex)

```
public enum Symbole
{
  VIDE('-'), CROIX('X'), ROND('O') ;  // objets pré-definis

  private final char libelle ;        // attribut

  private Symbole(char unLibelle)     // constructeur
  {this.libelle = unLibelle ;}

  public char getLibelle() {return this.libelle ;} // opération
}
```

exemple d'utilisation :

```
{
      System.out.println(Symbole.CROIX.getLibelle()) ; // affiche : X
}
```