Université Paris Diderot **Programmation Orientée Objet**
L2, EIDD et M1 linguistiques **Année 2017-2018**

TP n° 4

Jeu d'échecs

Le but de ce tp est la réalisation d'une version simplifiée d'un Jeu d'échecs. Les règles complètes sont disponibles ici : `https://fr.wikipedia.org/wiki/Échecs`. Vous pouvez proposer votre propre modélisation, potentiellement différente de celle proposée ci-dessous.

Exercice 1 Pièces sans Contraintes

On commence par la définition des pièces d'échecs.

1. Écrire une classe `Piece`, qui a comme attribut un booléen qui correspond à la couleur de la pièce, où *true* indique 'blanc', et *false* indique 'noir' (vous pouvez aussi introduire des constantes :
   `public Static final boolean BLANC = true, NOIR = false;`).
2. Créer un constructeur qui initialise la couleur avec une valeur *true* ou *false* prise comme paramètre.
3. Redéfinir la méthode `toString()`, pour qu'elle affiche "p" (pour 'pièce'), soit en minuscule (pour 'blanc'), soit en majuscule (pour 'noir').

Remarque : Quand on exécute `System.out.println(p);` où p est une référence qui n'est pas nulle, la méthode `p.toString()` est invoquée, en affichant la String définie dedans. Si p est nulle, `System.out.println(p);` va afficher *null*.

Exercice 2 Pièces spécifiques

Définir les types *Pion*, et *Roi* en étendant la classe *Piece*. (Facultatif : définir les quatre types restant *Tour, Cavalier, Fou, Dame*.)

— Définir le constructeur de chaque classe de prendre un paramètre booléen et initialiser la couleur.

— Redéfinir la méthode toString() dans chaque sous-classe pour qu'elle affiche la première lettre du nom de la classe (soit en minuscule, soit en majuscule).

Exercice 3 Modélisation du Plateau

1. Écrire une classe Case, qui a comme attributs : un booléen indiquant si elle est blanche ou noire, et un attribut de type *Piece* donnant la pièce qui est sur la case (`null` si la case est vide)). Ecrire le getteur `Piece getPiece()` ainsi que des méthodes `boolean estVide()`,
   `void remplirPiece(Piece p)`, `void enleverPiece()`.
2. Écrire une méthode toString() dans Case, qui renvoie soit la couleur de la Case (si vide), soit la pièce dessus.
3. Écrire une classe Plateau qui a comme attribut :

— des entiers final correspondant à la longueur et la largeur du plateau.

— un tableau de Case

4. Ajouter à la classe Plateau :
   — Un constructeur `public Plateau(int longueur, int largeur)`, qui génère un plateau de taille variable et de couleur alternées.
   — Un getteur `public Case getCase(int x,int y)`, qui renvoie la Case de coordonnées x et y, ainsi que des méthodes `void videCase(int x, int y)` et `void remplirCase(int x, int y, Piece p)` pour interagir avec le Case.
   — Une méthode `public void afficher()`, qui affiche le plateau (utiliser la mèthode `toString()` de *Case*).
   — Une méthode `public boolean horsLimite(int x, int y)` qui testant si le point (x,y) est bien un carré du tableau.

Exercice 4 Gestion du jeu

Écrire une classe *Echecs* qui gère un jeu d'échecs. Dans cette classe, au début, vous devrez juste créer un plateau (préférablement de petite taille, disons 4×4, voir les règles de mini-échecs ici : https://fr.wikipedia.org/wiki/Mini-échecs), l'initialiser avec des pièces diverses dessus et l'afficher. On réexaminera cette classe ci-dessous.

Exercice 5 Le déplacement

On veut maintenant ajouter la possibilité de déplacer les pièces d'une case à l'autre.

1. Ecrire la classe *Deplacement* ayant quatre attributs `int x0,y0,x1,y1`.
2. Définir des méthodes divers pour faciliter la vérification de la validité d'un déplacement selon les pièces décrites. Vous remplirez la fonctionnalité fournie par cette classe à la demande tandis que vous programmez la classe Echecs. Choix possibles à considérer :
   — le type du déplacement `char quelType()` qui retourne un caractère décrivant le type ('v' : vertical, 'h' : horizontal, 'd' : diagonal, 'c' : cavalier, 'x' : aucun des précédents).
   — la distance absolue entre le point de départ et le point d'arrivée si le déplacement est du type 'h','v' ou 'd'. Par exemple :
     ◦ pour un déplacement m1 entre les cases (1,5) et (3,3) de type 'd', la méthode retourne 2.
     ◦ pour un déplacement m2 entre les cases (1,5) et (1,2) de type 'v', la méthode retourne 3.
     ◦ pour un déplacement m3 entre les cases (1,1) et (2,3) de type 'c', la méthode retourne -1.
   — autres que vous avez besoin...
3. Penser à surcharger quelques méthodes de la classe Plateau pour prendre un déplacement comme argument au lieu d'un carré (x,y), par exemple la méthode *horsLimite* (`public boolean horsLimite(Deplacement d)`).

Exercice 6 Déplacement est valide – héritage

La validité d'un déplacement dépend de la pièce dont on parle. Par exemple, le roi peut se déplacer horizontalement, verticalement, et en diagonale mais juste pour une distance de 1, alors que le fou peut juste se déplacer en diagonale mais en couvrant une distance ≥ 1 (seulement si les cases intermédiaires sont vides).

On veut séparer la partie du code qui s'applique à tout type de pièce considérée, de celle qui doit vraiment prendre en compte les spécificités de la pièce.

1. Ecrire une méthode `boolean estValide(Deplacement d, Plateau p)` dans *Piece*, testant si le déplacement est valide pour la pièce concernée et l'état du plateau actuel. Ici, vous pourriez tester les propriétés du déplacement qui s'appliquent sur tout type de pièce, par exemple si le point d'arrivée est occupé par une pièce de la même couleur.
2. Redéfinir la méthode `boolean estValide(Déplacement d, Plateau p)` pour chaque sous-classe de *Piece* que vous avez définie. Ici, on teste les propriétés qui s'appliquent aux types particuliers, par exemple si le type de déplacement ('h','v', etc) est en accord avec le type de pièce. N'oubliez pas d'ajouter l'instruction `super.estValide(d,p);` dans toute sous-classe appelant la version de classe *Piece*.

   Remarque : La méthode *estValide* est polymorphe, c'est-à-dire, quand on exécute `Piece p = ...; p.estValide(deplacement,plateau);`, la JVM décide le vrai (sous) type de p et exécute la bonne variante de la méthode (liaison dynamique).
3. (Facultatif : une autre modélisation) Vous pouvez introduire une 'grande' classe *HorVerDiag* qui remplace tous les classes représentant les pièces qui se déplacent horizontalement, verticalement ou en diagonale vers tous les quatre directions possibles (c-à-d les classes *Fou*, *Tour*, *Dame* et *Roi*). Utilisez des attributs booléens pour spécifier le bon type. Ça laisse seulement les classes Pion et Cavalier à définir et traiter séparément.

Exercice 7 Gestion d'un tour

Pour qu'un déplacement dans le tableau soit valide, il faut vérifier conditions suivantes :

— Les coordonnées indiquent bien deux cases dans le plateau.

— Il y a effectivement une pièce `p` sur la case de départ.

— Cette pièce appartient au joueur en train de jouer.

— Il n'y a pas de pièce appartenant à ce même joueur sur la case d'arrivée.

— Le mouvement est valide pour la pièce `p`.

1. Écrire une méthode `void jouerTour(Deplacement d, boolean joueur, Plateau p)`, qui teste les conditions ci-dessus et réalise le déplacement (vider la case de départ et remplir la case d'arrivée avec notre pièce en supprimant la pièce de l'adversaire si il y en a une dessus).
2. Vérifier le résultat pour des paramètres divers. Afficher le plateau à nouveau après chaque exécution de la méthode `jouerTour`.

Exercice 8 (bonus) Gestion d'une partie

Enfin, on prend soin des points fondamentaux de la gestion du jeu.

La communication entre l'utilisateur et le programme. On se concentre d'abord sur l'interaction entre l'utilisateur et le programme.

1. Créer une nouvelle classe *Communication* ayant un attribut de type *Scanner*. On utilise un objet de ce type pour communiquer avec l'utilisateur (demande de saisie et aussi peut-être affichage de messages).
2. Ajouter au moins une méthode
   `public Deplacement demanderDeplacement(boolean joueur)`,
   qui demande a l'utilisateur de donner au clavier les quatre coordonnées du déplacement proposé et retourne un objet de type *Deplacement* avec ces paramètres.

Les tours alternés. Un jeu commence par le premier joueur (habituellement le joueur *'Blanc'*), qui joue un tour. Ensuite, les joueurs alternent chacun son tour. Si le tour qu'on vient de jouer ne marque pas la fin de jeu, ce sera à l'adversaire de jouer.

3. Introduire une méthode `jouerPartie`, c'est-à-dire, une boucle où les joueurs alternent chacun leur tour jusqu'à la fin du jeu (voir les conditions ci-dessous). Le code qui correspond à cette boucle demande (à nouveau) au joueur courant les coordonnées des points de départ et d'arrivée jusqu'à obtenir un déplacement valide, et l'exécute.
4. Le jeu s'exécute dans *main* en déclarant un objet de type *Echecs* et en exécutant la méthode *jouerPartie*.

La fin du jeu. La fin du jeu est déclenchée par la capture du roi d'un des joueurs.

5. Ajouter dans la classe *Piece* une méthode `boolean estRoi()`. Il faut peut-être la redéfinir dans les sous-classes de *Piece*.
6. Définir les conditions pour sortir de la boucle et annoncer le gagnant.