**Travail synthèse**

**(partie 2: classes Intersection et Echiquier)**

**Jeu d’échecs : fonctionnalités de base**

***la classe Intersection (un seul champs; une piece)***

***si pas de piece, champ piece=null;***

Nous allons débuter par modéliser une Intersection de l’échiquier. Quelles seront ses données et ses méthodes ?

données \_piece\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

méthodes :constructeurs :\_\_\_\_\_\_\_\_tableau d’objet\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***la classe Echiquier***

• Un échiquier sera donc un tableau 10x9 ( 10 lignes, 9 colonnes ) d’objets Intersection ; il s’agit donc de ses seules données :

private Intersection[][] jeu ;

• Le constructeur d’Echiquier devra donc créer le tableau ( il a été seulement déclaré  ) et créer également chacun des objets Intersection qui en fait partie. On créera un échiquier avec des **intersections vides au départ.**

* Intégrez à votre projet l'interface MéthodesEchiquier située dans LÉA. Votre classe Echiquier devra donc mettre en oeuvre cette interface.

• Il nous faut une méthode getIntersection qui, à partir d’un i ( rangée /ligne ) et d’un j ( colonne ) donné, nous retournera l’intersection correspondante.

• Il nous faut également une méthode debuter() permettant d’installer chacune des pièces sur son intersection de départ. On pourra le faire à l’aide de la méthode setPiece de Intersection.

Dans le chemin possible, on verifie PAS si cest valide (fait e npartie1)

Depl hor, depl ver, depl diag;et cavalier Ligne+colonne ),

• La méthode cheminPossible (Position depart, Position arrivee) est certainement celle qui vous causera le plus de soucis à cause du nombre de cas à traiter. Cette méthode vérifie si la pièce peut se frayer un chemin entre sa position de départ et celle d’arrivée. De plus, on vérifiera si la position d’arrivée n’est pas occupée par une pièce de la même couleur ; si c’est le cas, le chemin n’est évidemment pas possible.

La méthode chemin possible sert principalement à vérifier la présence d’obstacles sur le chemin du déplacement. Seule la bombarde peut sauter par-dessus une autre pièce et dans certaines conditions seulement, toutes les autres pièces ne peuvent pas.

La méthode chemin possible retournera false s’il y a d’autres pièces sur le chemin emprunté par le déplacement et s’il y a une pièce de la même couleur à l’arrivée.

Le cas particulier de la bombarde doit être traité et n’oubliez pas qu’un déplacement de cavalier est d’abord horizontal/vertical, suivi d’une case en diagonale.

**IMPORTANT :**

* **On suppose le déplacement nul comme étant valide.**
* **On suppose le déplacement soumis à la méthode comme étant théoriquement valide.**
  + **Autrement dit, on ne traite pas un déplacement qui serait non valide pour une pièce.**
  + **Par exemple, s’il s’agit d’un déplacement horizontal, c’est nécessairement pour une pièce qui peut faire un déplacement horizontal.**

**CAS SPÉCIAUX :**

* **Bombarde : pour capturer une pièce, il doit y avoir exactement une pièce entre la Bombarde et la pièce à capturer.(en ligne droite)**
* **Cible :autre couleur**
* **Roi : en aucun temps, les Rois doivent se situer un en face de l’autre ( dans la même colonne, il doit y avoir une pièce entre les deux Rois ) 🡪 voir méthode suivante**

• Une dernière méthode, roisNePouvantEtreFaceAFace, sert justement à vérifier ce dernier fait. En fonction de la position de départ et de la position d’arrivée d’une pièce à déplacer, vous devrez vérifier que le déplacement de la pièce n’amènera pas les deux rois à être face-à-face ( dans la même colonne ) sans aucune pièce les séparant.

N’oubliez pas que, comme dans cheminPossible, vous ne devez pas déplacer les pièces pour le moment. On n’utilise pas la méthode roisNePouvantEtreFaceAFace dans cheminValide, c’est une validation séparée.

* Ces deux dernières méthodes seront passablement longues; vous devez les « découper » en plusieurs sous-méthodes. On fait appel à une méthode séparée lorsque : le code est assez long et/ou le code se répète et/ou la lecture d’un expression conditionnelle (if) est difficile à interpréter, etc.

FINALEMENT

La classe de test pour cette étape vous est laissée à votre discrétion. Soyez judicieux dans son utilisation, les tests JUnit sont recommandés.

Une partie importante de la correction est faite à partir d’un échiquier avec des pièces prédisposées et des tests de chemins possibles pour les diverses pièces et conditions. Il est permis pour vous d’en faire de même!

IMPORTANT

Dans la **troisième partie**, on utilisera l’interface graphique pour déplacer les pièces.

C’est à ce moment qu’on utilisera les validations : estValide (partie 1), cheminPossible (partie 2) et roisNePouvantEtreFaceAFace (partie 2) pour autoriser un déplacement ET mettre à jour l’Echiquier suite à un déplacement valide.

C’est à cette étape seulement qu’on pourra déplacer des pièces, capturer des pièces adverses etc.