Développement orientée objet - Java
TP 06 - Partie 2

Lionnel Conoir, Isabelle Delignières, Wajdi Elleuch
Rémi Synave et Franck Vandewiele

Analyse de la situation

Les pièces du jeu d'échecs ont tous des déplacements différents en fonction de leur type. Les pions par exemple ne peuvent avancer que vers l'avant, les tours se déplacent d'autant de cases qu'elles veulent en restant sur la même ligne ou colonne, les fous vont en diagonale, etc. Ainsi, si on veut ajouter une méthode pour trouver tous les coups possibles pour une pièce, il faudra que cette méthode tienne compte de l'attribut `type`. Le développement deviendra rapidement chaotique et illisible.

La solution consiste ici à créer autant de classes différentes qu'il y a de types de pièces. Chaque classe correspondant à un type de pièce sera en charge de trouver tous les coups possibles pour son cas particuliers. En procédant de cette façon, nous allons reporter, dans les classes spécialisées, le code dépendant du type de la pièce. L'attibut `type` de la classe `Piece` devient alors inutile. La classe `Piece` comportera alors des méthodes dont le corps ne pourra pas être écrit directement mais sera écrit dans les classes spécialisées. La classe `Piece` contiendra donc des méthodes abstraites et deviendra, de fait, une classe abstraite. Les classes spécialisées correspondant chacune à un type de pièce particulier hériteront de cette classe abstraite et contiendront la redéfinition des méthodes abstraites.

Seule le pion nécessitera un traitement particulier. En effet, la couleur des tours, cavaliers, fous, rois et dames n'influent par sur leurs déplacements. Au contraire, en focntion de leur couleur, les pions possède un sens de déplacement.

TODO

Dans un premier temps, en partant du développement fait lors du TP1, vous allez procéder au changement de structure du projet en ajoutant toutes les classes correspondant à un type de pièce et en supprimant les références à l'attribut `type` dans la classe `Piece` qui deviendra une classe abstraite.

Ensuite, vous ajouterez dans chaque classe la méthode permettant de trouver tous les coups possibles.

Finalement, vous modifierez votre interface graphique afin de la rendre compatible avec le mouvement des pièces.

1 La classe `Piece`

Plusieurs actions à mener dans cette classe :

1. Pour préparer la suppression de l'attribut `type`, ajoutez/modifiez les méthodes suivantes :
   — La méthode `getType()` est dépendante de l'attribut `type`. Elle sera donc réécrite dans toutes les classes spécialisées. Définissez donc cette méthode abstraite et supprimez son implémentation. Bien qu'abstraite, cette méthode peut tout de même être utilisée dans cette classe.
   — Dans les méthodes `getNomLong`, `getNomCourt` et `toString`, si vous aviez utilisé l'attribut `type`, modifiez les afin d'utiliser la méthode `getType` à la place.
2. Supprimez toutes les références à l'attribut `type` :
   — Supprimez l'attribut `type`.
   — Modifiez tous les constructeurs pour supprimer le type passé en paramètre et/ou initialisé.
   — Supprimez le setter relatif au type.
   — Supprimez la condition utilisant le type dans la méthode `equals`.
3. Ajoutez une méthode abstraite `getDeplacementPossible`. Afin de trouver tous les déplacements possibles pour une pièce (prise d'une pièce adverse comprise - voir exemple sur la figure 1), il faut que cette pièce connaisse le plateau. Vous passerez donc le plateau en paramètre. Cette méthode retournera un tableau dynamique de toutes les `Position` possibles. Cette classe ne connaissant plus le type de la pièce, cette méthode ne peut être implémentée ici. Pour le type de retour, vous devrez importer la classe `ArrayList`. Voir la documentation de cette classe pour connaitre le bon package.

Les sous classes

La sous-classe `Pion` nécessitera un traitement particulier par la suite.

Pour toutes les sous-classes (`Tour, Cavalier, Fou, Dame, Roi, Pion`) de `Piece`, vous allez développer les méthodes suivantes :

— Un constructeur par défaut qui crée une pièce blanche en position A1.
— Un constructeur prenant en paramètre un objet de type `Position` et une couleur.
— La méthode `getType` qui retourne le nom complet du type de la pièce en minuscule. Cette méthode était abstraite dans la super-classe.

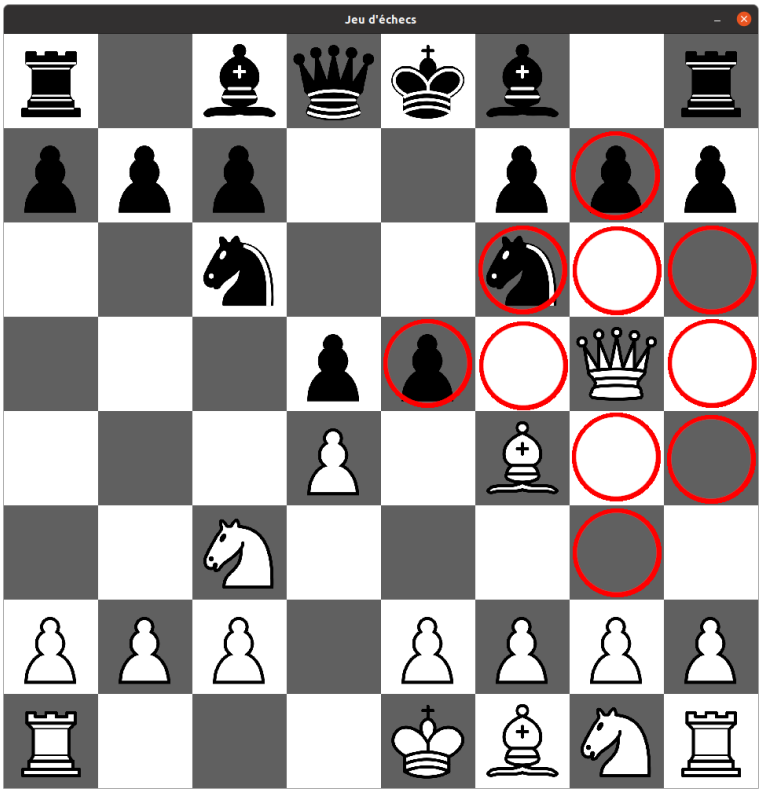

FIGURE 1 – Exemple de déplacements possibles de la dame blanche.

— **Sauf pour la classe** `Pion`, la méthode `getDeplacementPossible` qui était également abstraite dans la classe `Piece`.

La classe `Pion` ne donnant pas la redéfinition de la méthode `getDeplacementPossible`, cette classe est alors abstraite et doit être déclarée comme tel.

La sous-classe `Pion`

Alors que les déplacements possibles des pièces précédentes n'étaient pas contraints par leur couleur, le pion possède un sens de déplacement lié à sa couleur. Les pions blancs avancent de la ligne 2 vers la ligne 8 et, inversement, les pions noirs vont de la ligne 7 vers la ligne 1.

Avec la même logique que précédemment, nous allons donc créer des classes `PionBlanc` et `PionNoir` héritant de `Pion`.

Ces deux classes contiendront :

— Un constructeur par défaut qui crée un pion blanc/noir en A1/A8.

— Un constructeur prenant une position en paramètre.

— La méthode `getDeplacementPossible`. Attention, cette méthode n'est pas tout à fait la même en fonction de la couleur (voir figure 2). De plus, il est rappelé sur si le pion se trouve en ligne 2 (ou 7 pour les noirs), le pion peut avancer de deux cases (si aucune autre pièce ne l'en empêche évidemment). Il est rappelé également que le pion prend en diagonale. La règle de la prise en passant ne sera pas implémentée.

La classe `MainGraphique`

Modifiez votre classe `MainGraphique` afin que lorsque l'utilisateur clique sur une pièce, la liste des coups possibles appraissent.

Vous pouvez modifier le constructeur par défaut de la classe `Plateau` pour créer la configuration de départ de votre choix afin de vérifier si vos algorithmes fonctionnent correctement.

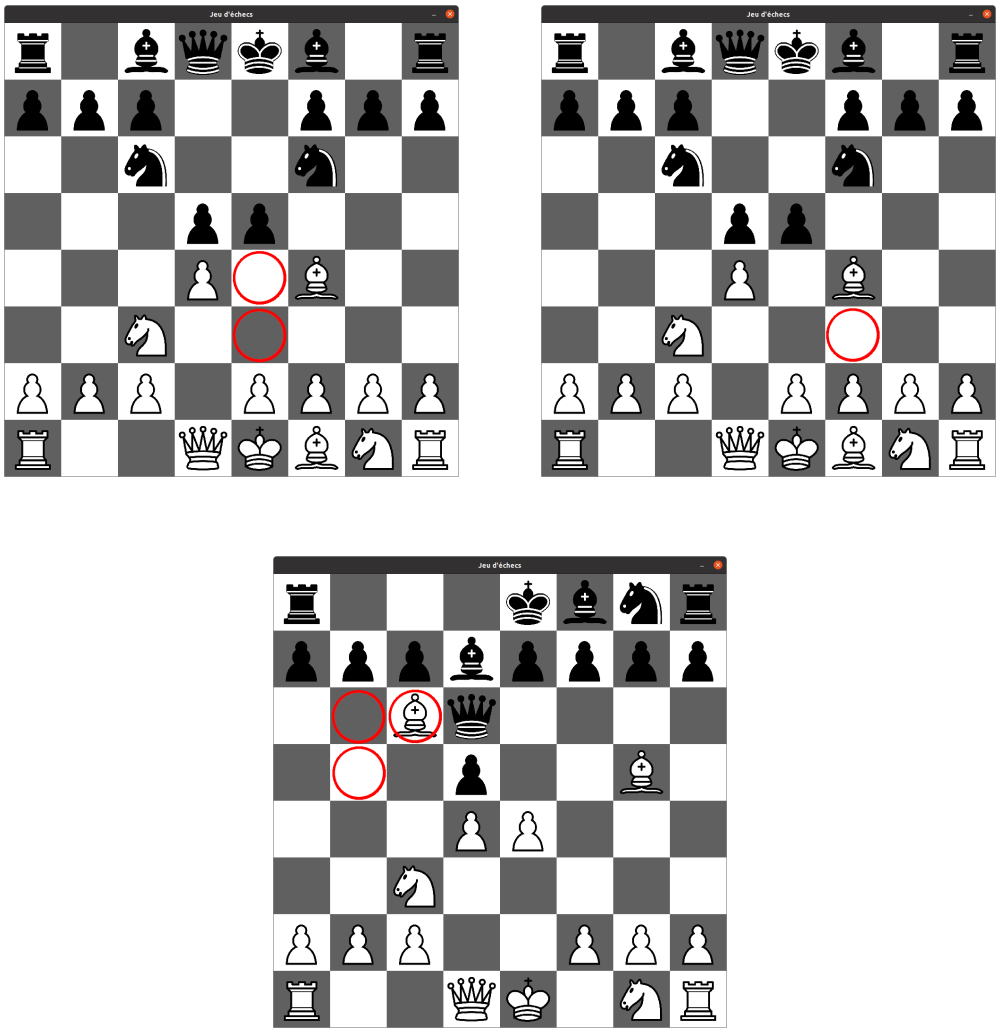

FIGURE 2 – Exemple de déplacements possibles pour les pions.