II.1102 : Algorithmique et Programmation
TP 4: Programmation orientée objet

Patrick Wang

23 octobre 2020

Table des matières

1 Objectifs du TP

— Continuer de travailler sur la programmation orientée objet ;
— Se familiariser avec les mécanismes d'héritage et d'implémentation ;
— Mettre en pratique ses connaissances dans le cadre du développement d'un jeu de plateau.

2 Rappels

2.1 Héritage

Le principe d'héritage est un principe fondamental de la programmation orientée objet. Celui-ci nous dit qu'il est possible de créer une relation de **généralisation-spécialisation** entre plusieurs classes.

En effet, il est possible d'avoir une classe généralisant une idée ou un concept, par exemple, une classe `Mammifère`. Au sein de ce concept de mammifère, on trouve aussi différentes espèces qui appartiennent au groupe des mammifères, par exemple, les classes `Chien` ou `Baleine`. Ces deux espèces d'animaux, bien que mammifères, présentent ainsi suffisament de différences pour les différencier avec deux classes. Mais le fait est qu'un chien ou une baleine reste un mammifère, et que ce lien de **généralisation-spécialisation** est bien réel.

L'une des principales difficultés de la programmation orientée objet sera donc de savoir modéliser correctement son logiciel afin de créer les classes et *super-classes* utiles au développement de son projet.

Attention : En Java, une classe fille ne peut pas hériter de plusieurs classes mères.

2.2 Implémentation

Le principe d'implémentation est un autre principe fondamental de la programmation orientée objet. Celui-ci repose sur la création d'*interfaces* qui déclarent des méthodes publiques sans les implémenter.

Une interface permet donc de décrire un comportement qui peut être partagé par plusieurs classes n'ayant pas de relation de type généralisation-spécialisation.

Pour reprendre l'exemple du cours, il est possible de créer une interface `Surface` qui déclare une méthode pour calculer une surface. Cette interface peut ensuite être implémentée par un `Carré` ou un `Disque`. Ces deux formes géométriques ne présentent pas nécessairement de relation de type généralisation-spécialisation, mais il est possible de calculer la surface de ces deux formes géométriques.

Une autre difficulté de la programmation orientée objet est donc de bien distinguer la différence entre **héritage** et **implémentation**, puis de bien savoir les utiliser.

Attention : En Java, il est possible qu'une classe implémente plusieurs interfaces.

2.3 Polymorphisme

Le polymorphisme est une propriété qui repose sur les principes d'héritage et d'implémenation. Cette propriété nous aide à généraliser notre logiciel soit en instanciant des objets sur leurs super classe, soit en définissant des méthodes utilisant comme paramètres des super classes.

Listing 1 – Illustration du polymorphisme.

```
public class Main {
    public static void main(String[] args) {
        // Exemple de polymorphisme par heritage
        // On instancie sur la super classe
        Animal dog = new Dog();
        Animal cat = new Cat();

        feed(dog);
        feed(cat);
    }

    // On donne comme parametre Animal pour generaliser la methode
    private static void feed(Animal animal) {
        // Do something...
    }
}
```

3 Développement d'un jeu d'échecs

3.1 Rappels de la séance précédente

Lors de la séance précédente, nous avons commencé à construire les bases d'un jeu d'échecs en Java. Nous avions donc implémenté les classes suivantes : `Player`, `Position`, `Cell`, `Chess`, `Main`, ainsi que toutes les classes permettant de représenter les différentes pièces d'un jeu d'échecs.

Cependant, nous nous sommes heurtés à quelques problématiques d'implémentation, en particulier dans la classe `Cell`. L'objectif de ce TP est donc de voir comment l'héritage et le polymorphisme permettent de résoudre ces problématiques assez simplement.

À la fin de ce TP, nous aurons modélisé de façon assez précise le jeu d'échecs et vous pourrez, si vous le souhaitez, finaliser cette implémentation afin de rendre ce jeu fonctionnel.

3.2 Implémentation du jeu d'échecs

3.2.1 Création d'une classe `Piece`

Puisque les Rois, Reines, Fous, Cavaliers, Tours, et Pions sont tous des *pièces* du jeu d'échecs, on peut voir ici une relation de **généralisation**. Nous allons donc utiliser le mécanisme d'héritage pour modifier un peu la structure de notre programme.

Questions préliminaires :

- Quels sont les attributs communs à toutes les classes représentant les pièces définies lors du TP précédent ?
- Quelles sont les méthodes communes à toutes les classes représentant les pièces définies lors du TP précédent ?

En utilisant le mécanisme d'héritage, nous pouvons donc créer une classe abstraite `Piece` qui va déclarer les attributs et méthodes suivantes :

- `position` qui est de type `Position` ;
- `color` qui est un entier (0 pour blanc, 1 pour noir) ;
- `public abstract boolean isValidMove(Position position, Cell[][] board)` qui sera une méthode abstraite (et qui devra donc être implémentée par les classes héritant de la classe `Piece`) ;
- `public abstract String toString()` qui sera aussi une méthode abstraite.

Le contenu de la classe `Piece` peut être représenté par le programme suivant :

Listing 2 – Contenu de la classe `Piece`.

```
public abstract class Piece {
    // On note la visibilite des attributs definies a protected
    protected Position position;
    protected int color;

    public abstract boolean isValidMove(Postion position, Cell[][] board);
    public abstract String toString();
}
```

3.2.2 Modifications des classes représentant les pièces

Maintenant que la classe abstraite `Piece` existe, nous allons devoir modifier l'ensemble des classes représentant les pièces du jeu d'échecs et créées lors du TP précédent.

Ces modifications concernent :

— L'héritage de la classe `Piece` ;

— La suppression des attributs définis lors du TP précédent (ces attributs sont désormais directement hérités de la classe `Piece`) ;

— L'ajout de l'*annotation* `@Override` au-dessus des deux méthodes `isValidMove(Position position, Cell[][] board)` et `toString()` pour indiquer que ces méthodes *surchargent* celle de la superclasse `Piece`.

3.2.3 Modifications dans la classe `Cell`

Une autre modification à apporter concerne la classe `Cell`. Voici ce qui avait été implémenté la semaine dernière pour cette classe :

Listing 3 – Contenu de la classe `Cell`.

```
public class Cell {
    private final Position position;
    private boolean isEmpty;
}
```

Le sujet du TP de la semaine dernière mentionnait aussi un attribut permettant d'identifier la pièce se trouvant sur cette case de l'échiquier. Maintenant que nous avons déclaré la classe `Piece`, nous pouvons compléter cette classe avec l'attribut suivant : `currentPiece` de type `Piece`, et dont la valeur `null` indique simplement que la case est vide.

3.2.4 Implémentation de la logique du jeu

Lors du TP précédent, nous avons aussi mentionné quelques méthodes de la classe `Chess` qui pouvaient présenter des difficultés d'implémentation. Il y avait en particulier les trois méthodes suivantes :

— `private boolean isValidMove(String move)` qui va vérifier qu'une pièce se trouve bien dans la première partie de `move` puis qui va faire appel à la méthode `isValidMove()` de cette `Piece` afin de vérifierque le déplacement joué est valide (considérant la configuration actuelle du plateau) ;

— `private boolean isCheck()` qui vérifie si un joueur est échec ;

— `private boolean isCheckMate()` qui vérifie si un joueur est échec et mat.

Pour l'implémentation de la méthode `isValidMove(Position position, Cell[][] board)`, on pourra s'aider du pseudocode suivant :

```
private boolean isValidMove(Position position, Cell[][] board) {
    1. On va vérifier que la position finale est bien valide pour le type de
       pièce à déplacer ;
    2. On va vérifier que la configuration de l'échiquier autorise ce
       déplacement ;
    3. On retourne (résultat du premier test AND résultat du second test).
}
```

Pour l'implémentation du premier point, on pourra définir pour chaque pièce un algorithme permettant de valider les déplacements en fonction de sa position actuelle sur l'échiquier.

Pour l'implémentation du second point, on va s'assurer que les deux points suivants ne sont pas vrai :

— Il n'y a pas de pièce de même couleur déjà présent sur la position d'arrivée ;

— Il n'y a pas de pièce de couleur quelconque se trouvant sur le chemin (ce point n'est pas valable pour le déplacement des cavaliers).

Pour l'implémentation de la méthode `isCheck()`, on pourra s'aider du pseudocode suivant :

Pour l'implémentation de la méthode `isCheckMate()`, plusieurs solutions plus ou moins élégantes s'offrent à nous.