Fiche 4 : conception objet

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
`https://www.lri.fr/~blsk/IPO/`

V1, automne 2022

Pointeur nul Rappel sur l'organisation des données en Java : chaque objet est matérialisé par une structure en mémoire, allouée dans le *tas*. On manipule un tel objet par l'intermédiaire d'un *pointeur* (autrement appelé une *référence*) vers la structure sous-jacente : lorsque l'on stocke un objet dans une variable ou dans un tableau, ou lorsque l'on passe un objet en paramètre à une méthode, on stocke ou on passe le pointeur, et non la structure entière. Lorsqu'une variable `a` désigne un objet, l'accès à un champ `a.x` consiste donc à aller chercher le champ `x` dans la structure désignée par le pointeur associé à la variable `a`.

Le pointeur **`null`** est un pointeur particulier, qui ne pointe vers *rien*. Il apparaît notamment lorsque l'on crée une structure destinée à contenir un objet, mais que l'objet en question n'a pas encore été fourni. Avec une définition comme

```
Personnage tab = new Personnage[5];
```

on a créé un tableau destiné à stocker cinq personnages, mais ces derniers n'existent pas encore. À défaut d'objets associés en mémoire, les cinq champs de `tab` contiennent pour l'instant le pointeur nul. On l'observe dès que l'on tente d'accéder à un attribut ou une méthode de l'un de ces objets, par exemple avec `tab[3].x`.

```
Exception in thread "main" java.lang.NullPointerException:
  Cannot read field "x" because "<local1>[3]" is null
```

Wrappers En Java, toutes les données manipulées sont des objets, exceptées les données de certains types de base comme **`int`** ou **`double`**. Du fait de l'omniprésence des objets et des classes, certains mécanismes ne fonctionnent qu'avec des classes, et pas avec les types de base. Par exemple : les classes des tableaux redimensionnables `ArrayList` ou des tables de hachage `HashMap` sont paramétrées par des classes, de sorte qu'on puisse écrire des déclarations comme :

```
ArrayList<Personnage> tab;
HashMap<String, ArrayList<Personnage>> map;
```

mais *pas* `ArrayList<int>`.

Pour corriger cela, Java propose des classes appelées *wrappers* associées à chacun des types de base, qui permettent d'utiliser ces types de base comme des classes ordinaires (et les valeurs comme des objets). On pourra donc écrire la déclaration

```
ArrayList<Integer> tab;
```

pour un tableau redimensionnable contenant des entiers.

Un objet `obji` de la classe `Integer` peut être :

— manipulé comme un objet, notamment avec des appels de méthodes

```
obji.compareTo(obj)
```

— ou utilisé comme un entier ordinaire, auquel cas il est effectivement converti en **`int`**.

```
obji + 3
```

Note : la fonction `Integer.parseInt` déjà croisée est donc une fonction fournie par cette classe *wrapper* `Integer`. Elle renvoie une valeur de base de type **`int`**.

Conception La *conception* d'un programme consiste à définir la structure générale du code et des données manipulées, pour réaliser un projet donné. Dans un cadre de programmation objet, on cherche notamment à :

— identifier les classes à définir et leurs attributs,

— déterminer comment chaque action à réaliser va se décomposer en appels de méthodes sur des objets des différentes classes.

On considère l'exemple d'un jeu de plateforme en deux dimensions où évoluent plusieurs personnages. On suppose le terrain divisé en cases. Un personnage est tourné soit vers la gauche, soit vers la droite, et avance d'une case par tour dans cette direction, à moins qu'il ne se trouve au-dessus du vide. Dans ce dernier cas il tombe d'une case par tour, tout en continuant à regarder du même côté. Enfin, le personnage change de direction s'il ne peut pas avancer. On cherche à définir un ensemble de classes permettant de modéliser cette situation, avec des méthodes réalisant les déplacements.

Classes En première approximation, on peut introduire une classe pour chaque élément à manipuler. En l'occurrence, on peut citer : les personnages, le terrain où évoluent les personnages, et une classe principale pour le jeu lui-même. D'autres pourront être ajoutées en fonction des besoins, lorsque l'ensemble deviendra plus précis.

Pour chacune des classes, on énumère les différents éléments composant un objet.

— Pour la classe `Personnage`, on retient au minimum une position et une direction. On pourrait ajouter en fonction des besoins : un identifiant, un état de santé, un équipement...
— Pour la classe `Terrain`, il nous faut au minimum une carte du terrain (zones libres et obstacles) et l'ensemble des personnages présents.
— Pour la classe principale `Jeu`, on demande au minimum un terrain et l'ensemble des personnages actifs.

Ces éléments vont être matérialisés par des attributs, dont le type sera à choisir en fonction des manipulations envisagées.

Cases libres Pour déplacer un personnage, on a besoin de pouvoir déterminer si une case est libre. Cette question est relative au terrain : on peut imaginer la traiter avec une méthode définie dans la classe `Terrain`, prenant en paramètres les coordonnées d'une case (numéro de ligne, numéro de colonne) et renvoyant un booléen.

```
public boolean estLibre(int lig, int col) { ... }
```

Cette méthode doit déterminer si la case ciblée est libre, c'est-à-dire si d'une part elle n'est pas un obstacle du terrain, et d'autre part elle ne contient pas déjà un personnage. Pour le faire efficacement, on propose que les attributs de la classe `Terrain` représentant le terrain lui-même et les personnages présents soient tous deux représentés par des tableaux à deux dimensions.

```
public class Terrain {
  private int hauteur, largeur;
  private boolean[][] carte;
  private Personnage[][] personnages;
```

Dans cette version, on propose de représenter le terrain par un tableau de booléens, indiquant par exemple **`true`** dans le cas d'une case libre, et **`false`** dans le cas d'un obstacle. On pourrait imaginer une version plus élaborée dans laquelle chaque case du terrain est représentée par un objet pouvant avoir des caractéristiques variées.

```
  public boolean estLibre(int lig, int col) {
    return this.carte[lig][col] && this.personnages[lig][col] == null;
  }
```

Déplacements Le déplacement d'un personnage concernant, justement, un personnage, on pourrait vouloir le réaliser par une méthode définie dans la classe `Personnage`. Cependant, l'application de l'algorithme de déplacement demande, entre autres choses, de consulter le statut libre ou non de certaines cases. Or, seul le terrain a cette connaissance, et la classe `Personnage` n'a, en l'état, pas d'accès au terrain. On peut donc plutôt définir une méthode au niveau du `Terrain`, ou du `Jeu` lui-même.

Le déplacement lui-même se manifeste à plusieurs endroits :

— il faut modifier les coordonnées du personnage `p` déplacé,
— il faut modifier le tableau `personnages` du terrain, pour retirer `p` de sa case d'origine et l'ajouter dans sa case de destination.

Ainsi, dans la classe `Jeu` on pourrait définir :

```
  public void joue(Personnage p) {
    int l = p.getLig();
    int c = p.getCol();
    if (this.terrain.estLibre(l+1, c)) {
      this.terrain.deplace(p, l+1, c);
      p.setPos(l+1, c);
    } else {
      ...
    }
  }
```

Cette méthode elle-même repose sur deux méthodes auxiliaires : `setPos` dans la classe `Personnage`, qui modifie les coordonnées d'un personnage,

```
  public void setPos(int lig, int col) {
    this.lig = lig;
    this.col = col;
  }
```

et `deplace` dans la classe `Terrain`, qui met à jour le tableau enregistrant les positions des personnages.

```
public void deplace(Personnage p, int lig, int col) {
  this.terrain[p.getLig()][p.getCol()] = null;
  this.terrain[lig][col] = p;
}
```

Pour capturer d'éventuels bugs, et assurer la robustesse de ce code, on peut ajouter une vérification du fait que les nouvelles coordonnées sont bien des coordonnées valides de notre terrain. Pour cela comme pour le reste, il faut se placer dans une classe disposant de suffisamment d'informations pour réaliser ce test. En l'occurrence, les personnages n'ont pas cette information, mais le terrain l'a. On pourrait donc ajouter la ligne

```
  assert (0 <= lig && lig < this.hauteur && 0 <= col && col < this.largeur);
```

au début de la méthode `deplace`. Ce faisant, on se rendra vite compte qu'il manque *quelque part* une précaution : aucune des méthodes écrites jusqu'ici ne teste si la case de coordonnées (`l+1`, `c`) vers laquelle le personnage est susceptible de chuter appartient bien aux limites du terrain. On a plusieurs manière de corriger cela. Par exemple :

- enrichir la méthode `estLibre` (classe `Terrain`) pour qu'elle ne renvoie **`true`** que si la case cible *existe* effectivement, ou encore
- introduire un test supplémentaire dans la méthode `joue` (class `Jeu`) pour traiter à part cette situation. Cela peut être pour l'interdire, ou au contraire pour déclencher un effet alternatif, comme retirer le personnage du jeu.

Bilan Dans la construction d'un programme, même d'ampleur modeste, on est amené à définir et à faire interagir plusieurs éléments. Le travail de conception demande d'identifier ces éléments, et de délimiter les contours *et les responsabilités* de chacun. Ce qui correspond, vu de haut, à une unique action, est généralement décomposé en plusieurs opérations élémentaires, s'appliquant à l'un ou l'autre des différents éléments présents. Pour faire cette décomposition, et répartir les tâches entre les différents acteurs, il faut observer ce à quoi chacun à accès.

Un problème d'origine étant fixé, il y a souvent de multiples manières légitimes de le résoudre, basées sur différentes organisations des données ou décompositions des actions. Autant que faire se peut, on favorise les solutions les plus *intelligibles*, dans lesquelles la structure générale et la répartition des responsabilités est claire (on affinera ces critères plus tard). Une fois ce découpage fixé, on cherche à l'intérieur de chaque classe les structures de données et algorithmes qui donneront à l'ensemble une efficacité raisonnable.