Software Engineering
Cédric Zanni

Design Pattern / Patron de conception

Certains patrons de conception (tel que **Iterator**) sont couramment utilisés par plusieurs bibliothèques standards (Java, Python, C++, …) afin de permettre de traiter plusieurs type de conteneur sans entraîner une trop grande redondance de code.

Iterator

Patron permettant de parcourir une collection d'objets indépendamment de la représentation interne de cette dernière. Cela peut permettre :

- d'implémenter certains algorithmes indépendamment de la représentation utilisée. Par exemple, chercher un élément vérifiant une propriété donnée dans un conteneur (personne d'âge supérieur à un seuil donné, …)
- de changer facilement la représentation d'un conteneur utilisé (à condition que l'on utilise uniquement l'interface commune à plusieurs conteneurs)

Ce concept nécessite l'utilisation de deux classes (ou interfaces) distinctes : l'une pour représenter l'objet contenant les éléments à parcourir (Iterable), l'autre pour représenter l'objet permettant d'effectuer le parcours des éléments dans la pratique (Iterator).

Iterable

```
Iterator iterator(); // permet de récupérer un objet de type Iterator pour parcourir les éléments du conteneur.
```

Iterator

```
E next(); // permet de récupérer l'élément suivant dans le conteneur
boolean hasNext(); // permet de savoir si il reste des éléments à parcourir
```

Attention : même si les itérateurs dans d'autres langages suivent le même principe général, certaines différences fondamentales existent.

Si l'on a un conteneur dérivant de Iterable on peut donc parcourir les éléments qu'il contient de la manière suivante :

```
MyContainer<Shape> container = new MyContainer<>();
// add content

Iterator<Shape> it = container.iterator();

while ( it.hasNext ()){
    Shape s = it.next();
    // do something with s
}
```

En Java les classes implémentant Iterable permettent l'utilisation de '*for-loop*' :

```
MyContainer<Shape> container = new MyContainer<>();
// add content

for (Shape s : container) {
   // do something with s
}
```

Template method

Patron permettant de paramétrer le comportement d'un algorithme (à ne pas confondre avec *template function* en C++ qui sont l'équivalent des fonctions génériques en Java). Le principe est d'implémenter le coeur de l'algorithme au sein d'une classe de base et de laisser l'utilisateur implémenter certaines portions du code en dérivant de la classe de base et en surchargeant des méthodes abstraites.

Exemple :

- recherche d'un élément dans un conteneur (méthode abstraite : tester si un élément correspond au critère souhaité)
- tri d'un conteneur (méthode abstraite : comparer deux éléments)

```
public abstract class FindIf<E> {
    protected abstract boolean testElement(E element);

    public E findIf(Iterable<? extends E> iterable) {
        for (E e : iterable) {
            if (testElement(e)) {
                return e;
            }
        }
        return null;
    }
}

public class FindGreater extends FindIf<Integer> {
    private final Integer val;
    public FindGreater(Integer n) {
        this.val = n;
    }

    protected boolean testElement(Integer element) {
           return element >= n;
    }
}
```

Plus l'implémentation est longue ou complexe plus cette approche prend de l'intérêt : pourquoi ré-implémenter un algorithme de tri lorsque l'on dispose déjà d'une implémentation optimisée.

Alternative : délégation

De manière similaire, il est possible de remplacer la méthode abstraite permettant de paramétrer l'algorithme par :

- un membre de la classe auquel on va déléguer la réalisation d'une tâche élémentaire
- un paramètre de la méthode auquel on va déléguer la réalisation d'une tâche élémentaire

Afin de laisser un large choix de paramétrisation, la seule contrainte sur le type de cet objet est qu'il implémente une interface donnée (par exemple Predicate ou Comparable). Il est alors possible de choisir le type de ce membre à la construction de l'objet / à l'appel de la méthode.

Si l'on reprend l'exemple précédent, on a alors :

```
public class FindIf {
    // WARNING : in this case the genericity is on the method because it is static
    // it would be on the class if we are implementing a container-like class as
    // done in the practical work
    public static <E> E findIf(Iterable<? extends E> iterable, Predicate<? super E> predicate) {
        for (E e : iterable) {
            if (predicate.test(e)) {
                return e;
            }
        }
        return null;
    }
}
```

Il suffit alors de choisir un type vérifiant l'interface lors de la construction de l'objet :

```
class Greater implements Predicate<Integer> {
    private final Integer limit;
    public Greater(Integer lim) {
        this.limit = lim;
    }
    public boolean test(Integer e) {
        return e >= n;
    }
}

// somewhere else :
ArrayList<Integer> list = new ArrayList<>();
// add content ...
FindIf.findIf(list, new Greater(5));
```

Utilisation d'interface fonctionnel

`Predicate` et `Comparable` sont des interfaces fonctionnelles, c'est à dire des interfaces ne proposant qu'une seule méthode abstraite. Pour simplifier la création de ce genre d'objet, il est possible d'utiliser des fonctions *lambda*, qui sont des objets anonymes. L'utilisation de la fonction `findIf` dans le cas précédent s'en retrouve alors encore plus simplifiée car la classe Greater devient superflue (car remplacer par une fonction lambda) :

```
ArrayList<Integer> list = new ArrayList<>();
// add content ...
FindIf.findIf(list, (Integer n) -> { return n>=5; } );
```

La fonction lambda doit bien entendu avoir la même signature que la méthode virtuelle de l'interface (même paramètre et même type de retour).