Université de Montpellier / Faculté Des Sciences

Modélisation et Programmation par Objets (2e partie) 2017-2018 - HLIN 505

Enseignants
Marianne Huchard, Jessie Carbonnel, Clémentine Nebut, Abdelhak-Djamel Seriai

Ce document est un recueil de notes de cours. Il peut développer des aspects vus en cours plus succinctement ou inversement vous aurez en cours plus de détails sur certains points. Si vous y relevez des erreurs, n'hésitez pas à nous les signaler afin de l'améliorer.

1 Les itérateurs

Un itérateur est un objet permettant de parcourir les éléments d'une collection et plus généralement les éléments internes à un autre objet complexe (qui est un composite) sans exposer l'implémentation. Cette forme de navigation sur la structure interne d'un objet par un autre objet (l'itérateur) est un patron de conception connu (patron *Iterator*) et présenté dans le GOF [1].

Ce patron de conception permet à l'utilisateur de l'objet complexe (collection ou objet composite) de ne pas connaître les détails de l'implémentation lorsqu'il veut simplement accéder aux éléments internes comme s'ils étaient dans une liste. Un des avantages est que la structure interne de l'objet peut changer (ainsi que le fonctionnement interne de l'itérateur) sans que le programme utilisateur n'ait à changer. Gamma et al. parlent d'*itération polymorphe* pour indiquer que grâce aux itérateurs on dispose d'une interface uniforme pour traverser des agrégats (et donc notamment des collections).

L'objet complexe donne au programme utilisateur l'accès à un ou plusieurs itérateurs par l'intermédiaire d'une méthode. Un itérateur classique offre des méthodes permettant de savoir quel est le premier élément, l'élément courant, le prochain élément, s'il existe un prochain élément, ou pour détruire l'élément courant (méthode moins fréquente).

2 Itérateurs en Java

2.1 L'interface `Iterator` de Java 1.8

On peut imaginer l'itérateur comme un curseur qui se déplace sur les données internes auxquelles on accède. L'interface propose trois méthodes qui permettent ce déplacement, ainsi qu'une méthode de retrait et une méthode permettant d'exécuter un traitement passé en paramètre sur la fin d'un itérateur. Elle

1. Design Patterns : Elements of Reusable Object-Oriented Software, By Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Published Oct 31, 1994 by Addison-Wesley Professional

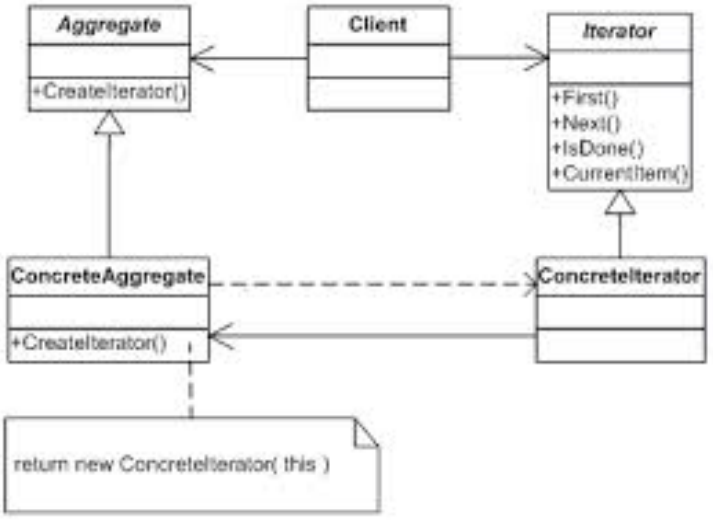

FIGURE 1 – **Patron Iterator (Gamma et al. 1994) : le programme client connaît l'agrégat et l'itérateur qui sert à le traverser. L'itérateur concret connaît ce qui lui est nécessaire de l'agrégat concret pour réussir à le traverser.**

définit donc le comportement des objets qui itèrent sur des objets complexes. L'interface est paramétrée par le type des objets retournés.

```
public interface Iterator<E>{
  boolean hasNext();  // Returns true if the iteration has more elements.
  E next();  // Returns the next element in the iteration.

  default void remove() {throw new  UnsupportedOperationException();}
  // Removes from the underlying collection the last element returned
                       // by the iterator (optional operation)

  default void forEachRemaining(Consumer<? super E> action)
      { while (hasNext())
         action.accept(next());} // Performs the given action for each remaining
         // element until all elements have been processed
         // or the action throws an exception.
}
```

2.2 L'interface `Iterable` (Java 1.8)

Cette interface définit le comportement des objets qui peuvent être la cible d'un itérateur, autrement dit sur lesquels on peut itérer. La méthode principale est `iterator()` qui retourne un itérateur sur l'objet complexe.

```
public interface Iterable<T>{
    Iterator<T>  iterator(); // Returns an iterator
```

```
    default void forEach(Consumer<? super T> action){
    // Performs the given action for each element of the Iterable until all
    // elements have been processed or the action throws an exception.
    for (T t : this)
         action.accept(t);
    }

    default Spliterator<T>  spliterator(){..}
    //Creates a Spliterator over the elements described by this Iterable.
}
```

2.3 Utilisation

Un premier exemple d'utilisation des itérateurs porte sur les collections de Java. Les collections que nous connaissons actuellement dérivent de l'interface `Collection`, qui dérive de l'interface `Iterable`. Dans une `ArrayList`, qui dérive de `Collection`, on héritera d'une implémentation de la méthode `iterator`. On y trouvera même une implémentation d'une forme spécialisée des itérateurs pour les listes (`ListIterator`).

On peut utiliser un itérateur de manière explicite, par exemple sur une liste d'étudiants `listeEtu` supposée non vide, contenant des objets d'une classe `Etudiant` disposant d'une méthode `moyenne()` :

```
List<Etudiant> listeEtu =  (...);
(...)
double moyenne=0;
Iterator<Etudiant> ite = listeEtu.iterator();
while (ite.hasNext())
   moyenne+=ite.next().moyenne();
moyenne = moyenne/listeEtu.size();
```

Il faut également savoir qu'il y a un itérateur caché (généré automatiquement) lorsque l'on appelle la boucle `"foreach"`. Cela signifie en particulier que dès que l'on définit un itérateur sur un objet complexe, on peut appeler sur celui-ci ce type de boucle.

```
double moyenne=0;
for (Etudiant e : listeEtu)
   moyenne+=e.moyenne();
moyenne = moyenne/listeEtu.size();
```

Un des gros avantages des itérateurs est que l'on peut utiliser la méthode `remove` pendant l'itération sans avoir de problème de changement d'indice. Par exemple, si on veut supprimer "Paolo" et "Jean", on peut écrire le code suivant.

```
Iterator<Etudiant> iter = listeEtu.iterator();
   while (iter.hasNext())
   {
      Etudiant e = iter.next();
      if (e.getNom().equals("Paolo")
               || e.getNom().equals("Jean") )
         iter.remove();
   }
```

Pour réaliser le même traitement avec une itération "for" sans itérateur (avec une variable compteur), on peut imaginer de diminuer l'indice, mais la documentation dit clairement qu'utiliser `remove` pendant une itération autrement que sur un itérateur provoquera un comportement non spécifié.

```
for (int i=0; i<listeEtu.size(); i++)
      if (listeEtu.get(i).getNom().equals("Paolo")
          || listeEtu.get(i).getNom().equals("Jean") )
         {listeEtu.remove(i); i--;} // DANGER
```

Les listes disposent de `ListIterator`(s) qui offrent plus d'opérations pendant la navigation et les opérations `add`, `set` et `remove` seront rendues plus efficaces car elles s'appliquent à un endroit déterminé sur la liste.

```
public interface ListIterator<E> extends Iterator<E> {
   boolean hasNext();
   E next();
   boolean hasPrevious();
   E previous();
   int nextIndex();
   int previousIndex();
   void remove();
   void set(E o);
   void add(E o);
}
```

3 Définir son propre itérateur

Dans cette partie, nous montrons comment définir des itérateurs sur un objet complexe, qui est ici une représentation d'un distributeur de briques PEZ.

On se propose donc de définir un distributeur de bonbons qui s'inspire du système des distributeurs de briques PEZ (voir figure 2). Il s'agit d'un tube de 12 bonbons, que l'on récupère par le haut du tube grâce à un mécanisme qui fait remonter les bonbons au fur et à mesure.

Une classe minimale est donnée ci-dessous avec un programme `main` montrant le fonctionnement de l'itérateur. Le programme suppose l'existence d'une classe `BonbonPez` disposant d'un constructeur sans paramètre.

```
public class DistributeurBonbonPez implements Iterable<BonbonPez> {

       private Color couleurTube = Color.pink;
       private String formeBouchon = "Minnie";
       private BonbonPez[] tube = new BonbonPez[12];

       public void remplit()
       {
              for (int i = 0; i < 12; i++) tube[i] = new BonbonPez();
       }
```

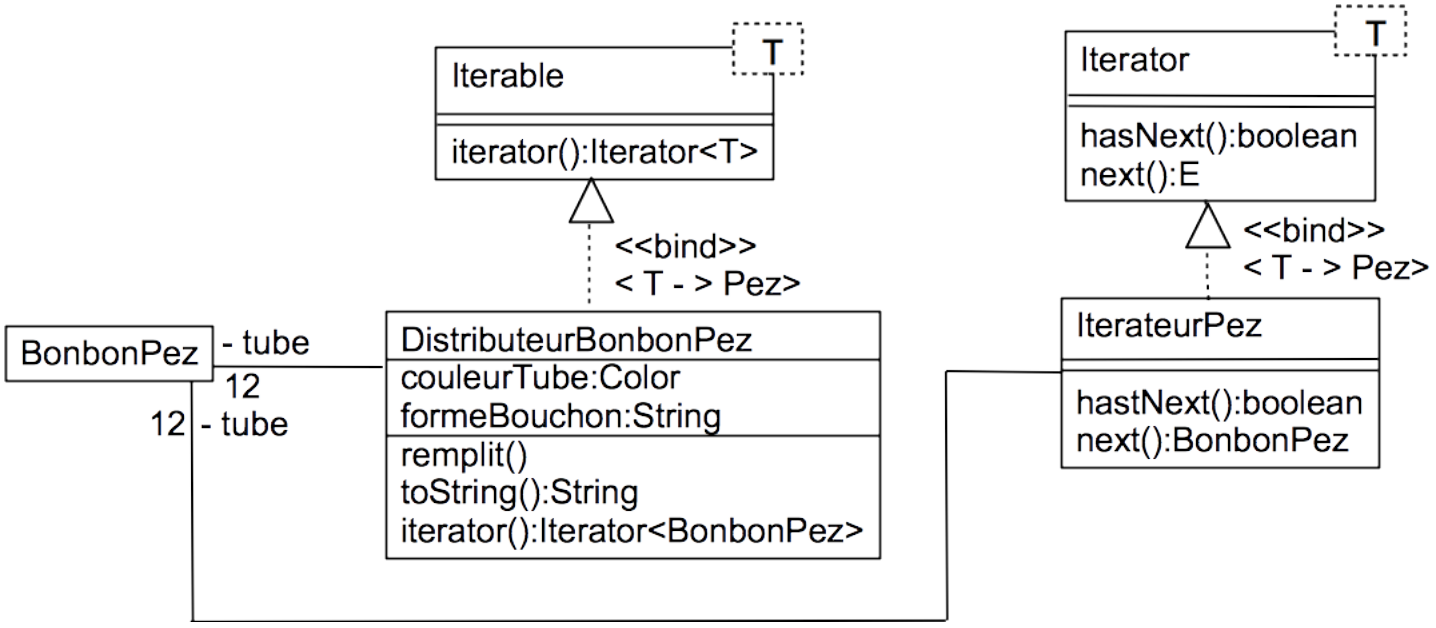

FIGURE 2 – **Patron Iterator appliqué au distributeur Pez**

```
@Override
public Iterator<BonbonPez> iterator() {
	return new IterateurPez(tube);
	// on donne parfois à l'itérateur la donnée sur laquelle il va travailler
	// (ici le tube)
}

public String toString()
{
	String s = "";
	for (BonbonPez b : tube) s+=b + " | ";
	return s;
}
```

L'itérateur est l'objet qui traverse le tube. Dans son implémentation, nous avons choisi d'effacer la case contenant la brique récupérée. C'est donc un itérateur "destructif". Si on désire qu'il ne le soit pas, on n'efface pas l'élément dans `next`. On peut le définir dans une classe à part mais on peut aussi en faire une classe interne ou une classe anonyme comme c'est souvent fait dans l'API Java.

```
public class IterateurPez implements Iterator<BonbonPez> {
	private BonbonPez[] tube; // la structure que l'on va traverser
	private int curseur = 0; // le curseur sur cette structure

	public IterateurPez(BonbonPez[] tube){this.tube = tube;}

	@Override
	public boolean hasNext() {
```

```
        return curseur < 12;
    }

    @Override
    public BonbonPez next() {
        BonbonPez bonbon = tube[curseur];
         tube[curseur]=null;
        curseur = curseur+1;
        return bonbon;
    }
}
```

Les deux classes une fois définies, on peut définir un programme simplement montrant leur utilisation.

```
    public static void main(String[] argv)
    {
        DistributeurBonbonPez d= new DistributeurBonbonPez();
        d.remplit();
        System.out.println(d);

         for (BonbonPez b : d)
         // boucle for each utilisant l'itérateur et vidant le tube
         // cette écriture est possible car on a définit un itérateur
        {
            System.out.println(b);
            System.out.println("Apres retrait " + d);
        }
    }
}
```