IN201 : 4 – Interfaces

Solution

Author : Christophe Garion <garion@isae.fr>
Public : SUPAERO 2A
Date :

Résumé

Le but de ce TP est de manipuler la notion d'interface pour réaliser un itérateur sur un agrégat.

1 Contenu

Ce corrigé contient des commentaires aux questions du TP sur les interfaces. Vous pourrez trouver les sources des classes et interfaces sur le site web.

2 Présentation du problème

Supposons que l'on dispose d'une classe représentant une liste sous forme d'un tableau pour stocker des objets quelconques servant dans une application. L'accès aux éléments dans ce tableau se fait en utilisant un index et permet des recherches avec une *complexité* particulière à l'ensemble.
Supposons maintenant que l'on se rende compte qu'un arbre binaire serait beaucoup plus adapté pour l'application que l'on développe. Dans ce cas, l'accès aux éléments stockés dans l'arbre binaire se fait selon un autre mécanisme. Si l'on a construit l'application en ne considérant que l'ensemble implanté sous forme de tableau, on va devoir revoir une grande partie du logiciel.
Lorsque l'on dispose d'un agrégat d'objets (comme par exemple une liste implantée sous forme d'un tableau, une liste chaînée ou un arbre), il est utile de disposer d'un mécanisme d'accès commun à ses éléments (pour les afficher par exemple). Nous allons mettre en œuvre le mécanisme d'*itérateur*. Un itérateur permet de réaliser un parcours de tous les éléments de l'agrégat.
Des exemples d'agrégats d'éléments de type **`double`** et les itérateurs associés sont présentés sur la figure 1.

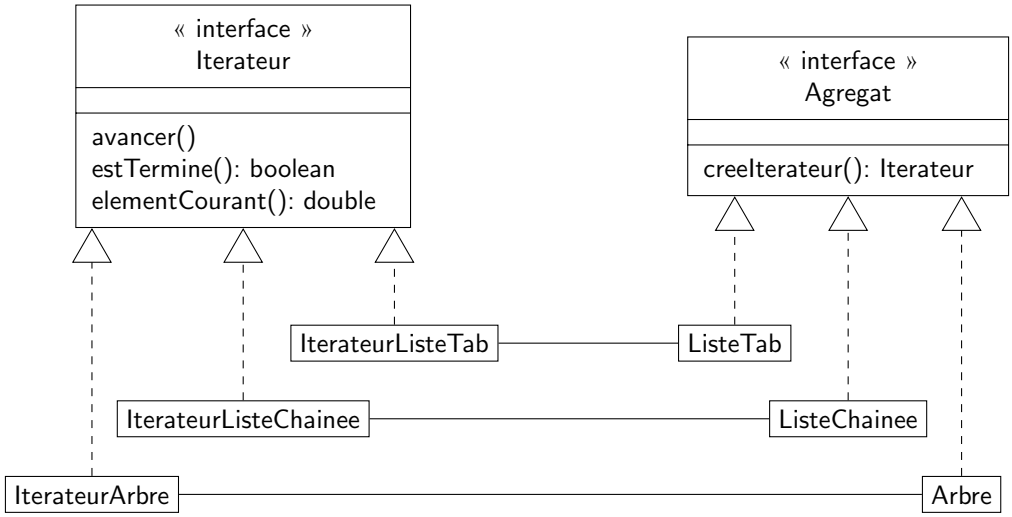

FIGURE 1 – Diagramme de classe présentant les interfaces `Iterateur` et `Agregat` et quelques agrégats et leurs itérateurs associés

Un itérateur sur un agrégat permet donc d'accéder aux éléments de l'agrégat suivant un parcours déterminé en protégeant la structure interne de l'agrégat. On peut ainsi disposer d'un mécanisme commun à tous les types d'agrégats et à tous les parcours possibles sur un même type d'agrégat. Cette solution de conception est ce que l'on appelle un *design pattern* (patron de conception), cf. [2, 1].

3 Implantation des interfaces

1. le diagramme UML de la figure 1 présente deux interfaces spécifiant les services rendus par deux catégories d'objets :

- les itérateurs, qui fournissent les services suivants :
  - `avancer()` qui positionne l'élément courant sur l'élément suivant de l'agrégat ;
  - `estTermine()` qui renvoie un booléen qui est vrai si on a terminé le parcours de l'agrégat. Cela signifie que *`estTermine()` sera vrai si et seulement si l'on est au delà du dernier élément de l'agrégat* ;
  - `elementCourant()` qui renvoie un **`double`** qui est l'élément courant dans l'agrégat.
- les agrégats, qui fournissent un seul service, `creeIterateur()` qui renvoie un `Iterateur` sur l'agrégat.

Écrire le code Java correspondant aux deux interfaces.

Solution :
Les sources sont disponibles sur le site. Pas de problème particulier a priori, il suffisait d'utiliser la syntaxe vue en cours.

2. écrire une classe `Utilitaire` possédant une méthode `nbElements` prenant un `Agregat` en paramètre et renvoyant le nombre d'éléments de l'agrégat.

Solution :
Le but de l'implantation de cette classe était de vous montrer que l'on pouvait commencer à préparer des tests à partir des interfaces (et donc sans connaître l'implantation des méthodes). On peut ainsi tester le comportement général des classes. C'est la **bonne** méthode à appliquer ! Il faut écrire les tests avant d'écrire les classes.
Plus techniquement, la méthode `nbElements` est statique car elle ne manipule pas d'instance de la classe.

4 Réalisation d'un agrégat et de son itérateur

On va maintenant réaliser un agrégat particulier, `ListeChainee` et son itérateur associé, `IterateurListeChainee`. Vous trouverez la classe `ListeChainee` dans votre dépôt (répertoire `src` et les autres classes nécessaires sont disponibles dans l'archive `lab4.jar` disponible sous le répertoire `lib`. `ListChainee` réalise l'interface `Liste` vue en cours.

1. écrire le diagramme UML présentant les relations existant entre les différentes classes et interfaces du problème ;

Solution :
Le diagramme est présenté sur la figure 2. Pas de difficulté particulière, mis à part le fait qu'il fallait bien penser à créer une association entre `IterateurListeChainee` et `ListeChainee`, sinon l'itérateur n'a aucun lien avec l'agrégat (cela était indiqué par le premier diagramme quand même) ! On peut remarquer également que `ListeChainee` réalise deux interfaces.

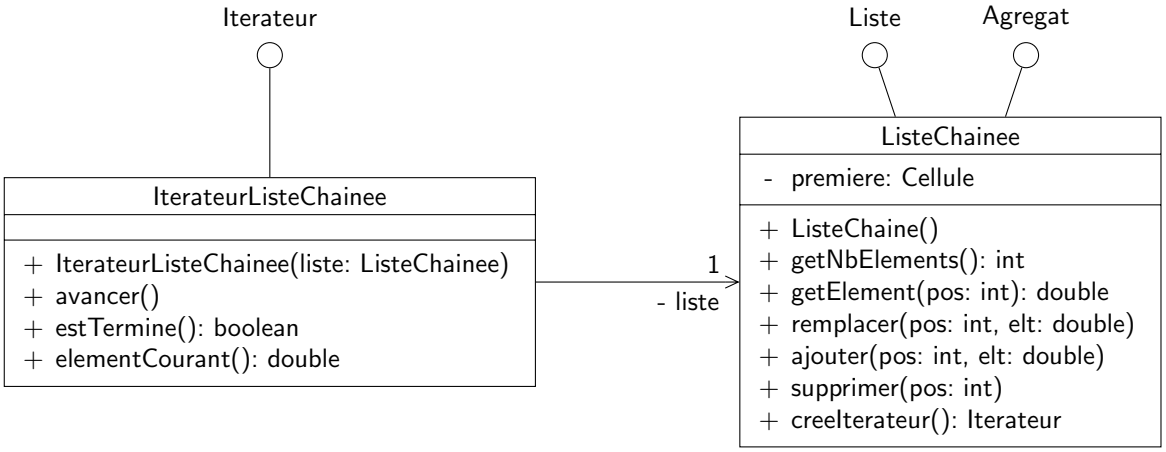

FIGURE 2 – Diagramme de classes présentant les classes `IterateurListeChainee` et `ListeChainee`

Remarque importante : on aurait bien sûr pu créer un itérateur commun à toutes les listes (comme par exemple `ListeTab`), car les méthodes spécifiées dans `Liste` suffisent à écrire l'itérateur. La solution choisie en TP n'est donc pas la meilleure. En particulier, on va devoir écrire une classe de test pour chaque itérateur créé, ce que l'on aurait

pu éviter en créant un itérateur commun à toutes les listes. Je propose sur le site une classe `IterateurListe` qui est un itérateur pour toutes les listes. Un programme de test (qui est en fait le même programme que celui pour `IterateurListeChainee`) l'utilise avec une instance de `ListeChainee`.

2. on va maintenant compléter la classe `ListeChainee` pour qu'elle réalise l'interface `Agregat` avant de construire la classe correspondant à l'itérateur associé, `IterateurListeChainee`. Réfléchir en particulier à ce dont on a besoin pour construire l'itérateur associé ;

   Solution :

   Il fallait écrire la méthode `creeIterateur`. Dans cette méthode, on doit renvoyer un itérateur sur l'agrégat, donc ici un `IterateurListeChainee` (le principe de substitution nous autorise à faire cela). Le diagramme de classe nous précise qu'il existe une association entre `IterateurListeChainee` et `ListeChainee`. La solution consiste donc à passer en paramètre du constructeur de `IterateurListeChainee` un objet de type `ListeChainee` qui sera stocké comme attribut de `IterateurListeChainee`. Le code de `creeIterateur` sera donc :

   ```
   @Override public Iterateur creeIterateur() {
       return new IterateurListeChainee(this);
   }
   ```

3. écrire la classe `IterateurListeChainee` et utiliser JUnit pour tester les méthodes de la classe ;

   Solution :

   Pas de problème particulier ici. Il fallait bien penser à stocker une cellule courante et utiliser la méthode `getPremiereCellule` de `Liste` pour obtenir la première cellule. Il suffisait ensuite d'utiliser la méthode `getSuivante()` de `Cellule` pour réaliser avancer par exemple.

   Le fait d'avoir des assertions permettait de débugger rapidement les classes.

   Pour la classe de test JUnit, j'ai essayé de tester le comportement de l'itérateur sur des listes particulières :
   - une liste vide
   - une liste avec un seul élément
   - une liste quelconque

   Ces données de tests sont intéressantes, car elles permettent de couvrir un nombre important de cas possiblement problématiques. C'est en effet souvent sur les cas « aux limites » (en particulier les ensembles vides ou les singletons lorsque l'on manipule des ensembles d'objets) que se produisent les problèmes. Vous remarquerez également que dans la classe de test de `ListeChainee`, je teste les méthodes de `ListeChainee` de la même façon : test à l'insertion d'un élément au début de la liste, à la fin etc.

   Puisque j'ai écrit un programme de test pour la classe `IterateurListe`, il serait intéressant de le réutiliser directement pour avoir les tests de `IterateurListeChainee` (ils sont identiques, on utilise simplement des instances de `IterateurListeChainee` au lieu de `IterateurListe`). Nous reparlerons lors du cours sur l'héritage. Enfin vous remarquerez que dans `IterateurListeTest` j'utilise le principe de substitution, puisque les poignées vers les listes sont déclarées comme étant de type `Liste` et que je leur associe des objets de type `ListeChainee`.

4. tester de façon élémentaire la classe `IterateurListeChainee` en utilisant la classe `Utilitaire`.

   Solution :

   Là encore, pas de problème particulier.

5 Les interfaces `Iterable<E>` et `java.util.Iterator<E>`

L'API standard de Java fournit en fait des classes et interfaces semblables à celles que nous venons de développer [1] :
- les *collections* représentées par l'interface `java.util.Collection` sont ce que nous avons appelé les agrégats
- les itérateurs sont représentés par l'interface `java.util.Iterator`

Reprenons notre exemple de liste chaînée. Si l'on veut développer un itérateur pour une liste chaînée, il suffit de créer une classe `ListeIterator` réalisant l'interface `java.util.Iterator`. De la même façon que pour `IterateurListe`, j'ai choisi ici de faire un itérateur « générique » pour tous les types de liste. C'est possible, car les méthodes fournies par l'interface `Liste` permettent de parcourir facilement une liste. Les méthodes de `java.util.Iterator` devant être implantées par `ListeIterator` sont
- `hasNext` qui correspond à `estTermine` (ou plutôt à sa négation)

1. Ces classes et interfaces sont dites *génériques* : nous reviendrons sur cette notion dans un prochain cours.

- next qui combine à la fois avancer et elementCourant
- remove qui permet de supprimer de la liste l'élément sur lequel l'itérateur est placé

Il n'y avait pas de difficulté particulière pour écrire la classe ListeIterator. Vous remarquerez que j'utilise la classe Double dans le paramètre de Iterator car les types génériques doivent être des classes. Double est une classe dite *wrapper* qui permet de représenter par des objets des types primitifs (**double** ici).
L'intérêt de posséder une réalisation de Iterator est de pouvoir faire réaliser l'interface Iterable par ListeChainee. J'ai choisi ici de créer une nouvelle classe, ListeChaineeIterable pour cela. Vous remarquerez que cette classe a un attribut de type ListeChainee et qu'elle délègue tous les services demandés par l'interface Liste à cet attribut. Iterable est une interface du paquetage java.lang. Ce paquetage contient des classes et interfaces « de base » pour le langage Java. Iterable ne contient qu'une seule méthode, iterator, qui doit renvoyer une instance de Iterator permettant de parcourir l'objet de type Iterable concerné. Une classe réalisant l'interface Iterable peut alors utiliser la boucle **for** particulière que nous avons utilisé par exemple sur ArrayList. C'est ce que je fais dans le programme de test TestListeIterator :

```
    public static void main(String[] args) {
        ListeChaineeIterable liste = new ListeChaineeIterable();

        System.out.println("Creation de la liste");

        for (int i = 0; i < 5; i++) {
            liste.ajouter(i, i);
        }

        for (Double d : liste) {
            System.out.println(d);
        }
    }
```

L'utilisation de ces interfaces nous permet donc de bénéficier d'un mécanisme de Java facilitant l'écriture de code. Vous pourrez utiliser Iterable et Iterator dans vos classes si le besoin s'en fait sentir.
Attention, il s'agit juste de *syntactic sugar*, i.e. d'une construction qui permet de rendre le code plus lisible. Le compilateur transforme cette boucle **for** en une boucle **for** utilisant un itérateur, comme le montre le résultat d'une décompilation de fr.isae.lists.TestListeIterator.**class** avec l'utilitaire javap (instructions 37 à 67) :

```
public class fr.isae.lists.TestListeIterator {
  public fr.isae.lists.TestListeIterator();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: new           #2                  // class fr/isae/lists/ListeChaineeIterable
       3: dup
       4: invokespecial #3                  // Method fr/isae/lists/ListeChaineeIterable."<init>":()V
       7: astore_1
       8: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
      11: ldc           #5                  // String Creation de la liste
      13: invokevirtual #6                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      16: iconst_0
      17: istore_2
      18: iload_2
      19: iconst_5
      20: if_icmpge     36
      23: aload_1
      24: iload_2
      25: iload_2
      26: i2d
      27: invokevirtual #7                  // Method fr/isae/lists/ListeChaineeIterable.ajouter:(ID)V
      30: iinc          2, 1
```

```
      33: goto          18
      36: aload_1
      37: invokevirtual #8                  // Method fr/isae/lists/ListeChaineeIterable.iterator:()Ljava/uti
      40: astore_2
      41: aload_2
      42: invokeinterface #9,  1            // InterfaceMethod java/util/Iterator.hasNext:()Z
      47: ifeq          70
      50: aload_2
      51: invokeinterface #10,  1           // InterfaceMethod java/util/Iterator.next:()Ljava/lang/Object;
      56: checkcast     #11                 // class java/lang/Double
      59: astore_3
      60: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
      63: aload_3
      64: invokevirtual #12                 // Method java/io/PrintStream.println:(Ljava/lang/Object;)V
      67: goto          41
      70: return
}
```

Références

[1] E. Freeman et al. *Head first design patterns*. O' Reilly, 2005.

[2] E. Gamma et al. *Design Patterns - Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, 1994.

License CC BY-NC-SA 3.0