Itérateurs et visite des collections (Java 1.7)
Avant-goût des Streams (Java 1.8)

HLIN505
Modélisation et programmation par objets 2

Université Montpellier 2

Octobre 2014

Itérateurs

Itérateur

Un itérateur est un objet qui permet :

- de visiter les éléments d'une collection un par un,
- plus généralement de visiter les éléments internes d'un autre objet complexe (qui est un composite).

Patron de conception `Iterator`

Présenté dans le GOF

Design Patterns : Elements of Reusable Object-Oriented Software

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Published Oct 31, 1994 by Addison-Wesley Professional.

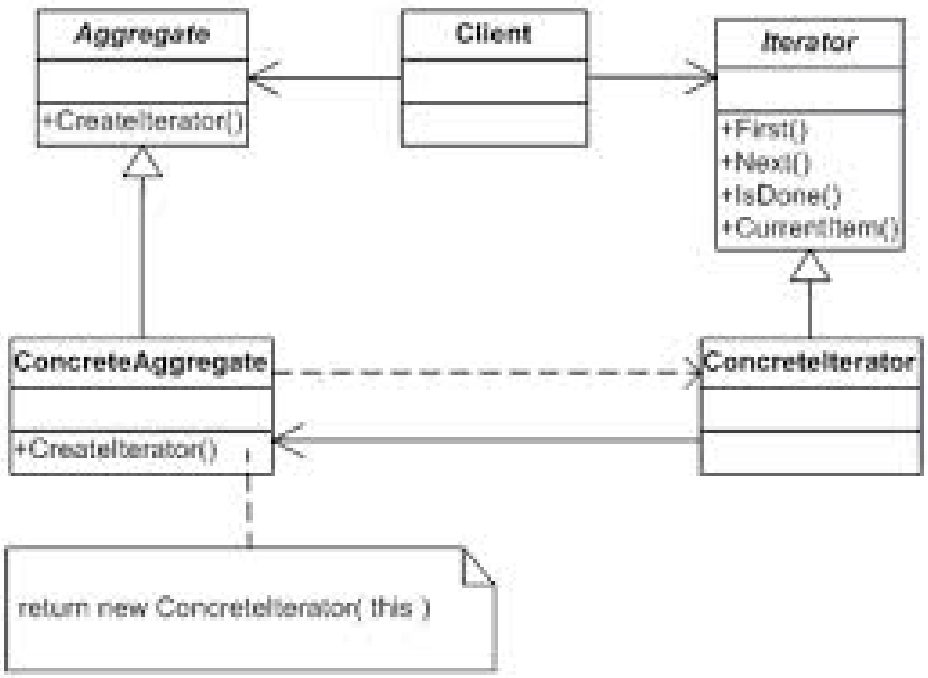
Patron de conception `Iterator`

Problème

- permettre à l'utilisateur d'un objet complexe (collection ou objet composite) de parcourir cette collection ou cet objet composite,
- au travers d'une interface uniforme (opérations de parcours standard),
- sans connaître les détails de l'implémentation,
- la structure interne de l'objet peut changer (ainsi que l'itérateur) sans que le programme utilisateur n'ait à changer.

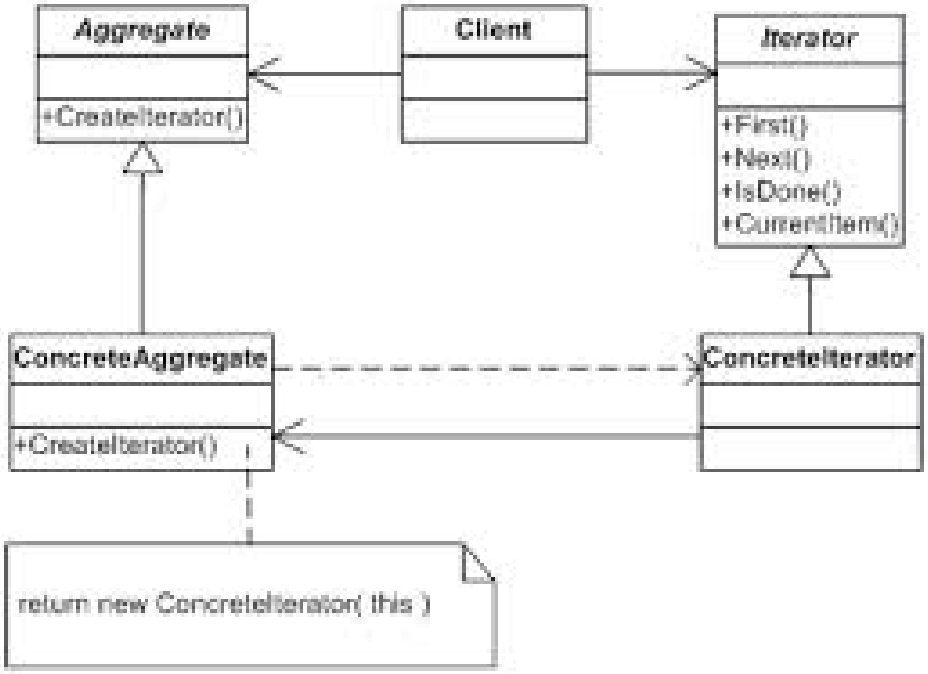
Patron de conception Iterator

Solution



Le programme `client` qui désire accéder à un `Aggregate` demande à ce dernier de lui procurer un distributeur de ses éléments (`Iterator`) par l'appel à la méthode `CreateIterator`.

Patron de conception Iterator



Ce distributeur d'éléments (`Iterator`) est créé par instanciation d'une classe `ConcreteIterator`, elle-même conforme à un type `Iterator` fournissant des opérations de parcours et de récupération des éléments.

Itérateurs

L'interface `Iterator` de Java

```
public interface Iterator<T> {
 T next();                  // retourne element courant
                            // et passe a l'element suivant
 boolean hasNext();         // teste s'il reste un element
 void remove();             // efface l'element visite
}
```

Correspondance avec le patron de conception

Java	*Patron du GOF*
l'itérateur est positionné par défaut au début	First()
next()	CurrentItem() et Next()
hasNext()	isDone()
remove()	pas d'équivalent

Itérateurs

Itérable

Un objet *itérable* est un objet sur lequel on dispose d'un `iterator`
c'est l'`Aggregate` du patron de conception `Iterator`

L'interface

```
public interface Iterable<T>
{
 Iterator<T> iterator();
}
```

Correspondance avec le patron de conception

Java	*Patron du GOF*
iterator()	CreateIterator()

Illustration avec les collections Java

Toutes les collections sont des objets itérables, notamment les listes.

Vue très simplifiée d'une liste

```
public class ArrayList<T> implements Iterable<T>{
    Iterator<T> iterator(){ .....}
}
```

Vue très simplifiée d'un itérateur de liste

```
public class ArrayListIterator <T>
                        implements Iterator<T>{
    T next(){.....}
    boolean hasNext(){.....}
    void remove(){.....}    ............
}
```

Exemple de parcours d'une liste d'étudiants

Création de la liste

```
List<Etudiant> listeEtu = new ArrayList<Etudiant>();
Etudiant zo =
        new Etudiant("Zoe", 12, 14, 17, 26, 1, 1);
Etudiant pa =
        new Etudiant("Paolo", 27, 1, 2);
Etudiant je =
        new Etudiant("Jean", 24, 1, 3);
listeEtu.add(zo);
listeEtu.add(pa);
listeEtu.add(je);
```

Itérateurs

Parcourir la liste avec une boucle `for` et une variable compteur

```
double moyenne = 0;

for (int i=0; i<listeEtu.size(); i++)
    moyenne += listeEtu.get(i).moyenne();

moyenne = moyenne/listeEtu.size();
```

Itérateurs

Parcourir la liste avec un itérateur

```
double moyenne2 = 0;
Iterator<Etudiant> ite = listeEtu.iterator();

while (ite.hasNext())
    moyenne2 += ite.next().moyenne();

moyenne2 = moyenne2/listeEtu.size();
```

Itérateurs

Parcourir la liste avec *for each* qui est traduit en un itérateur

```
double moyenne3 = 0;

for (Etudiant e : listeEtu)
    moyenne3 += e.moyenne();

moyenne3 = moyenne3/listeEtu.size();
```

Itérateurs

L'opération `remove`

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();
    }
```

Itérateurs

L'opération `remove`

Plutôt que ... dans le cas d'une boucle classique où on doit diminuer l'indice après le retrait

```
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--;}
```

Itérateurs

Un itérateur spécifique pour les listes

Où les opérations prévues dans l'interface seront spécialement efficaces

```
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);
}
```

Un itérateur de Pile

Créer un itérateur pour sa propre structure, par exemple une pile simplifiée

```
public class Pile<T> {
    private ArrayList<T> elements;
    public Pile(){initialiser();}
    public T depiler() {
        if (this.estVide()) return null;
        T sommet = elements.get(elements.size()-1);
        elements.remove(sommet);
        return sommet;
    }
    public void empiler(T t) {
        elements.add(t);
    }
    public boolean estVide() {
        return elements.isEmpty();}
```

Un itérateur de Pile

Suite de la définition de la pile

```
public class Pile<T> {
    ........
    public void initialiser() {
        elements = new ArrayList<T>();
    }
    public T sommet(){
        if (! this.estVide())
            return elements.get(elements.size()-1);
        else return null;
    }
    public String toString(){
        return "Pile = "+ elements;
    }
}
```

Pour rendre la pile itérable

```
public class Pile<T>
                implements Iterable<T>{
                .........
   public Iterator<T> iterator() {
          return new IteratorPile(elements);
   }
}
```

Une classe Itérateur de pile

```
public class IteratorPile<T> implements Iterator<T>{
    private Iterator<T> iterateur_elements;
    public IteratorPile(ArrayList<T> elements) {
        this.iterateur_elements = elements.iterator();
    }
    public boolean hasNext() {
        return this.iterateur_elements.hasNext();
    }
    public T next() {
        return this.iterateur_elements.next();
    }
    public void remove() {
        this.iterateur_elements.remove();
    }
}
```

Un main avec l'itérateur utilisé explicitement

```
public static void main(String[] a)
 {
  Pile<String> p = new Pile<String>();

  p.empiler("a"); p.empiler("b"); p.empiler("c");

  Iterator<String> it = p.iterator();
  while (it.hasNext())
    System.out.println(it.next());
 }
```

Un main avec foreach

```
public static void main(String[] a)
 {
  Pile<String> p = new Pile<String>();

  p.empiler("a"); p.empiler("b"); p.empiler("c");

  for (String element : p)
        System.out.println(element);
 }
```

Faire différents traitements ...

Introduction d'une nouvelle classe pour représenter des données entreprise

```
public class DossierEntreprise {
    private String identification;
    private int anneeCreation;
    private String emailAddress;
.........
}
```

Faire différents traitements ...

Introduction d'une classe utilitaire pour les piles avec des traitements spécifiques pour les piles de dossiers d'entreprises

```
public class PileParcours {

public static<T extends DossierEntreprise>
    void printMailJeunesEntreprises(Pile<T> p)
    {
        for (T element : p)
            if (element.getAnneeCreation()>=2012)
                System.out.println(element.getEmailAddress());
    }
...
}
```

Faire différents traitements ...

Manque de généralité du code précédent : à chaque nouveau traitement (imprimer identifications, calculer moyenne âge des jeunes entreprises, etc.) on écrira un nouveau parcours
Solution en Java < 7 : spécifier des traitements dans des classes

traitement de type test

```
public interface TestDossier {
    boolean test(DossierEntreprise d);
}

public class TestJeuneEntreprise implements TestDossier {
    @Override
    public boolean test(DossierEntreprise d) {
        return d.getAnneeCreation() >=2012;
    }
}
```

Faire différents traitements ...

traitement de type action

```
public interface ActionDossier {
    public void agit(DossierEntreprise d);
}

public class PrintEmail implements ActionDossier {
    @Override
    public void agit(DossierEntreprise d) {
        System.out.println(d.getEmailAddress());
    }
}
```

Faire différents traitements ...

Version généralisée du `print`

```
public class PileParcours {
    public static<T extends DossierEntreprise>
            void printEntreprises
                (Pile<T> p, TestDossier t, ActionDossier a)
    {
        for (T element : p)
            if (t.test(element))
                a.agit(element);
    }
}
 }
```

Faire différents traitements ...

Version généralisée du `print`

```
public static void main(String[] args) {
 Pile<DossierEntreprise> p = new Pile<>();
 p.empiler(new DossierEntreprise
       ("ChezJacques",2012,"cj@gmail.com"));
 p.empiler(new DossierEntreprise
       ("Laforet",2013,"lf@yahoo.fr"));
 p.empiler(new DossierEntreprise
       ("Ast",2010,"ast@astservice.com"));

 printEntreprises(p,
                  new TestJeuneEntreprise(),
                  new PrintEmail());
}
```

Vers Java 8 : lambdas, stream, agrégations

```
public static void main(String[] args) {

    Pile<DossierEntreprise> p = new Pile<>();
    p.empiler(new DossierEntreprise
        ("ChezJacques",2012,"cj@gmail.com"));
    p.empiler(new DossierEntreprise
        ("Laforet",2013,"lf@yahoo.fr"));
    p.empiler(new DossierEntreprise
        ("Ast",2010,"ast@astservice.com"));

    p
        .stream()
        .filter(d -> d.getAnneeCreation()>=2012)
        .map(d -> d.getEmailAddress())
        .forEach(email -> System.out.println(email));

}
```

Vers Java 8 : le goût des lambdas

Fonction anonyme

```
(liste de parametres)     ->      body
```

Quelques exemples

```
d -> d.getAnneeCreation()>=2012
(DossierEntreprise d) -> d.getAnneeCreation()>=2012
d -> { return d.getAnneeCreation()>=2012; }

(a,b) -> a+b
(int a, int b) -> a+b
```

Autres éléments

Capture, Utilisation de l'environnement, ...

Vers Java 8 : les Streams et les opérations d'agrégation

Stream

- Séquence d'éléments avec traitement séquentiel ou parallèle
- Ne stocke pas ses éléments mais décrit (de manière déclarative) sa source et les opérations qui seront effectuées
- Le traitement est pris en charge par l'interprète (et plus efficace)
- L'itération est interne (et non externe comme avec les itérateurs)

```
public static void main(String[] args) {
    Pile<DossierEntreprise> p = new Pile<>();
    ....
    p
        .stream()
        .....
}
```

Dossiers entreprise

Vers Java 8 : les Streams et les opérations d'agrégation

`filter`

retourne un second `stream` constitué des éléments du premier `stream` qui vérifient le prédicat

```
public static void main(String[] args) {
    Pile<DossierEntreprise> p = new Pile<>();
.....
    p
        .stream()
        .filter(d -> d.getAnneeCreation()>=2012)
.....
}
```

Dossiers entreprise dont l'année de création est postérieure à 2012

Vers Java 8 : les Streams et les opérations d'agrégation

map

retourne un troisième `stream` constitué des résultats de l'application de la fonction aux éléments du premier

```
public static void main(String[] args) {
    Pile<DossierEntreprise> p = new Pile<>();
.....
    p
        .stream()
        .filter(d -> d.getAnneeCreation()>=2012)
        .map(d -> d.getEmailAddress())
....
}
```

Adresses mails des dossiers entreprise dont l'année de création est postérieure à 2012

Vers Java 8 : les Streams et les opérations d'agrégation

foreach

applique une fonction aux éléments du `stream`

```
public static void main(String[] args) {
    Pile<DossierEntreprise> p = new Pile<>();
.......
    p
        .stream()
        .filter(d -> d.getAnneeCreation()>=2012)
        .map(d -> d.getEmailAddress())
        .forEach(email -> System.out.println(email));
}
```

Affichage des adresses mails des dossiers entreprise dont l'année de création est postérieure à 2012

Java 1.7

Age moyen des jeunes entreprises

```
public static<T extends DossierEntreprise>
            double ageMoyenJeunesEntreprises(Pile<T> p)
    {
        double m = 0;   int nbr= 0;
        for (T element : p)
              if (element.getAnneeCreation()>=2012)
                    {m += 2014-element.getAnneeCreation();
                    nbr++;}
        return m / nbr;
    }

//main
        System.out.println(ageMoyenJeunesEntreprises(p));
```

Java 1.8

Age moyen des jeunes entreprises

```
// main

        Pile<DossierEntreprise> p = new Pile<>();
.......
        System.out.println(
        p
            .stream()
            .filter(d -> d.getAnneeCreation()>=2012)
            .mapToInt(DossierEntreprise::getAnneeCreation)
            .map(i -> 2014 - i)
            .average()
            .getAsDouble());
```