Navigation de données complexes
Itérateurs et Streams

HAI401I
Modélisation et Programmation par Objets 2

Université de Montpellier

2022

Navigation de données complexes

Moyen *uniforme* de navigation (parcours) sur des données complexes qui peuvent être :

- collections, maps (dictionnaires associatifs),
- objets composites,
- flux, fichiers.

Deux stratégies en Java

Stratégies de navigation des données complexes en Java

Itérateurs	Streams
données plutôt finies	données finies ou infinies
itération externe	itération interne
sous le contrôle du programmeur	sous le contrôle de l'interprète
avec stockage des éléments	sans stockage des éléments
avec accès aux éléments	sans accès aux éléments

Itérateurs

Itérateur

Un itérateur est un objet qui permet :

- de visiter les éléments d'une collection ou d'un flux 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

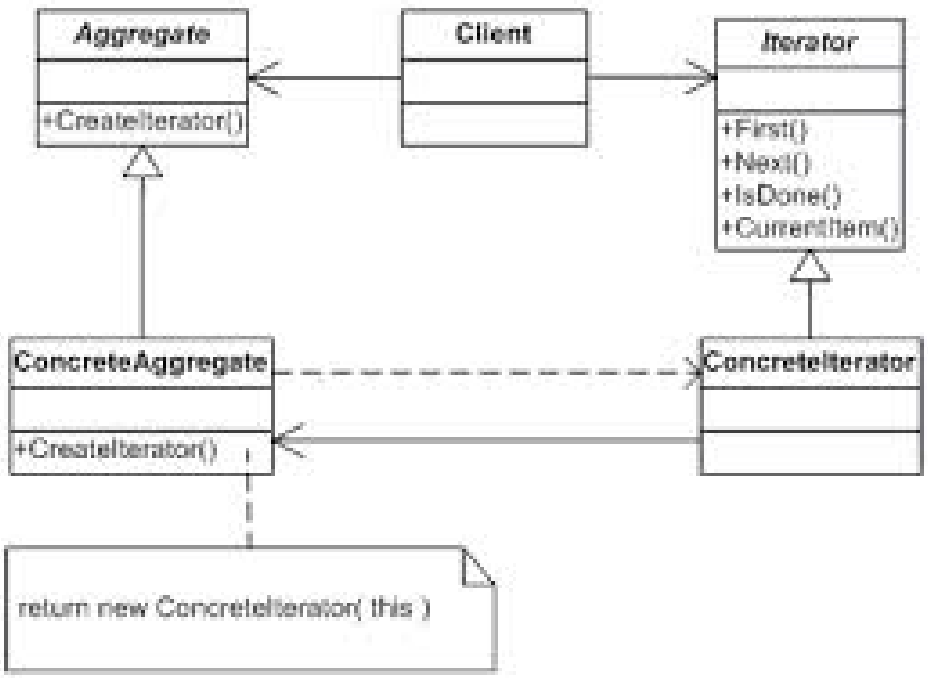
Patron de conception `Iterator`

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.

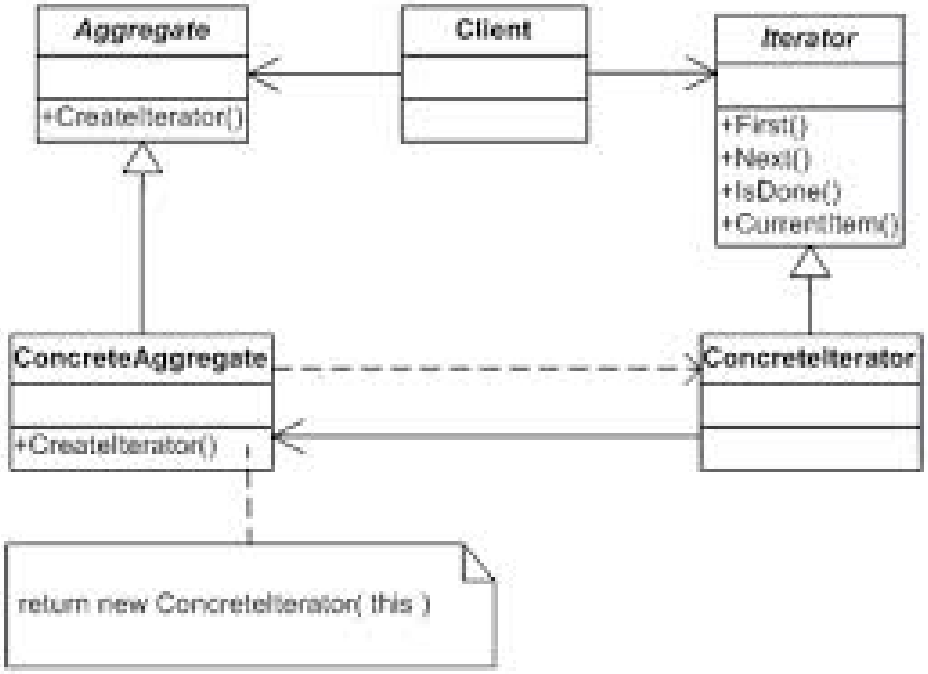
Patron de conception Iterator

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 l'element courant
                            // et passe a l'element suivant
 boolean hasNext(); // teste s'il reste un element
 default void remove(){...} // efface l'element visite
 default void forEachRemaining
                  (Consumer<? super E> action){...}
                  //  execute l'action
                  // pour tous les elements
                  // ou signale une exception
}
```

Itérateurs

Correspondance avec le patron de conception

Java	*Patron du GOF*
l'itérateur est positionné au début à la création	First()
next()	CurrentItem() et Next()
hasNext()	isDone()
remove()	pas d'équivalent
forEachRemaining(Consumer< ? super E> action)	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 la forme d'itération *for* qui est traduite 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

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<T>(elements);
   }
}
```

Une classe Itérateur de pile

```
public class IteratorPile<T> implements Iterator<T>{
// on stocke un iterateur sur la liste interne
   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);
}
```

Motivation pour faire des ... Streams

Introduction

Pour réaliser des itérations avec des traitements, différentes approches

- Itérer avec un itérateur et à chaque élément de la collection effectuer une opération
- Diverses manières de généraliser cette itération
- Les streams et les collectors (à partir de Java 1.8)

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;
        .........
}
```

Lambdas-expressions, streams, agrégations

Imprimer les adresses emails des entreprises créées après 2012

```
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));

}
```

Lambdas-expressions

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, ...

Streams et 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

Streams et 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

Streams et opérations d'agrégation

map

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

```
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

Streams et opérations d'agrégation

forEach

applique une fonction aux éléments du troisième `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 (en 2020)

```
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 += 2020-element.getAnneeCreation();
                    nbr++;}
        return m / nbr;
    }

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

Java 1.8

Age moyen des jeunes entreprises (en 2020)

```
// main

        Pile<DossierEntreprise> p = new Pile<>();
.......
        System.out.println(
        p
            .stream()
            .filter(d -> d.getAnneeCreation()>=2012)
            .mapToInt(DossierEntreprise::getAnneeCreation)
            .map(i -> 2020 - i)
            .average()
            .getAsDouble());
```

Références de méthodes ou de constructeurs

Il s'agit d'une autre manière de passer des opérations en paramètre

elles sont formées à l'aide de l'opérateur :: (opérateur de résolution de portée)

```
System.out::println
List<String>::size
ArrayList<String>::new
```

Catégories d'opérations

- opérations intermédiaires (créant d'autres *streams*)
  - le filtrage :
    - par un prédicat `filter(predicate)`,
    - en enlevant les doublons `distinct`,
    - en réduisant la taille `limit(n)`,
    - en sautant les *n* premiers éléments `skip(n)`
  - la projection,
    - qui se base sur une fonction, `map(fonction)`,
    - ou qui transforme en *streams* spécialisés comme `mapToInt` ou `mapToDouble`

Catégories d'opérations

- opérations terminales
  - Application d'une procédure (de type `void`) à tous les éléments du flot : `foreach(fonction)`
  - Recherches et appariements (sous forme d'opération terminale) : par un prédicat, pour vérifier
    - que tous les éléments le satisfont `allMatch(predicate)`,
    - qu'un élément le satisfait `anyMatch(predicate)`,
    - pour récupérer le premier élément qui le satisfait `findFirst(predicate)`
    - pour récupérer n'importe quel élément qui le satisfait `findAny(predicate)`

Catégories d'opérations

- opérations terminales
  - Réduction par `collect(Collector)`, où `Collector` est une opération de réduction ou de regroupement d'éléments de la collection
  - Réduction qui applique une opération de manière répétitive : `reduce(valeur d'accumulation, fonction d'accumulation)`
    sur les *streams* numériques on dispose d'opérations comme `sum()` ou `average()`

Interfaces fonctionnelles

Exemple :

```
@FunctionalInterface
public interface Predicate<T>{
  boolean test(T t)
  // Evaluates this predicate on the given argument
}
```

Ce sont les types des paramètres des opérations comme `filter`, `map` ou `forEach`. Exemple avec `filter` :

```
Stream<T> filter(Predicate<? super T> predicate)
```

Leurs instances sont des lambdas-expressions, des références de méthode ou de constructeur. Par exemple :

```
Predicate<DossierEntreprise> p1 = DossierEntreprise::sup2012;
Predicate<DossierEntreprise> p2 = d->(d.getAnneeCreation()>=2012);
```

Efficacité

Efficacité

- Calcul paresseux (Lazy)
  Le calcul des opérations intermédiaires est effectué si possible en une seule passe (c'est l'interpréteur qui s'en charge). Pour les pipelines de *streams*, le calcul est lancé lorsque l'opération terminale est atteinte.
- Parallélisation
  Remplacer `stream()` par `parallelStream()`. L'API des *streams* s'occupera de décomposer la requête pour qu'elle soit exécutée en parallèle si l'architecture machine le permet.

Création de Streams

D'où viennent les streams ?

- d'une collection comme vu précédemment
- de valeurs données en extension

  ```
  Stream<Integer> pairs = Stream.of(2,4,6,8);
  ```

  ou placées dans un tableau

  ```
  int[] tabDePairs = {2,4,6,8};
  IntStream pairs = Arrays.stream(pairs);
  ```

- d'un fichier
- en produisant des éléments à la demande à partir d'une fonction (cette production peut être "infinie")

Création de Stream à partir d'un fichier

Contenu et format du fichier (stocké dans ./data/books) : année/titre

```
1830/Le rouge et le noir
1837/Le rose et le vert
1830/le livre des oracles
```

Classe Paire utilitaire

```
class Paire{    public String premier, second;
   public Paire(String premier, String second) {
      this.premier = premier;  this.second = second;  }
   public String getPremier() {return premier;}
   public void setPremier(String premier) {this.premier = premier;}
   public String getSecond() {return second;}
   public void setSecond(String second) {this.second = second;}
   public String toString()
      {return "Paire [premier=" + premier +
                           ", second=" + second + "]";}
}
```

inspiré de http ://blog.paumard.org/category/java-8/stream/

Création de Stream à partir d'un fichier

Code qui récupère les lignes du fichier et les place dans une Map :

```
Map<Object,List<Paire>> m=
 Files
  .lines( Paths.get("data", "books"))
  .map(   (String line) -> {
   String[] elements = line.split("/") ;
   String annee = elements[0];
   String titre = elements[1];
   return new Paire(annee,titre);  }
   )
   .collect(Collectors.groupingBy(Paire::getPremier));
System.out.println(m);
```

Affichage résultant :

```
{ 1830=[Paire [premier=1830, second=Le rouge et le noir],
        Paire [premier=1830, second=le livre des oracles]],
   1837=[Paire [premier=1837, second=Le rose et le vert]]   }
```

Création de Stream à partir d'une fonction

Création du flot des premiers nombres pairs

```
Stream<Integer> pairs = Stream.iterate(0, n -> n + 2);
pairs.limit(4).forEach(System.out::println);
```

`limit(4)` sert à s'arrêter ... sinon le flot créé est infini et le programme ne s'arrête pas !

Synthèse

Naviguer des données complexes

- Itérateurs (qui sont des structures externes à la collection, avec accès explicite aux éléments)
- Streams (internes, optimisés, adaptés pour le calcul parallèle, avec accès implicite aux éléments)