Université de Yaoundé I

Rapport Groupe 6

Patron iterateur

Membres de l'equipe :

LEUNA FIENKAK NKEHEUP 20U2698
NEGOUE MAFO PATRICIA 20U2603
KAMDA MALVINA EVA 20U2843
TEGUIMENE YENDJI FUREL DE CONSOL 22W2453

Sous la supervision de :
Dr VALERY MONTHE

Table des matières

Introduction

La perte d'objets est un problème courant dans de nombreux contextes, que ce soit dans les transports publics, les espaces publics ou d'autres environnements. Pour remédier à cette situation, la conception et la mise en œuvre d'un système orienté microservices peuvent offrir des solutions novatrices. Ce rapport explore les différentes facettes de la création d'un tel système visant à identifier et à récupérer les objets perdus de manière efficace et rapide.

I Contexte

Dans le développement logiciel, l'itération sur des collections de données est une tâche courante. Les langages de programmation modernes offrent souvent des mécanismes intégrés pour parcourir des listes, des tableaux, ou d'autres structures de données. Cependant, le patron Itérateur abstrait ce processus, offrant une interface uniforme pour parcourir des collections de manière indépendante de leur implémentation sous-jacente.

II Problème

Lorsque nous travaillons avec des collections d'objets, nous pouvons souvent avoir besoin de parcourir les éléments de la collection, mais il n'est pas toujours souhaitable ou pratique d'exposer la structure interne de la collection au code client. De plus, différentes collections peuvent avoir des structures internes différentes, ce qui complique la tâche du code client lorsqu'il doit travailler avec plusieurs types de collections.

Difficultés Rencontrées :

- **Dépendance à la structure interne :**Sans l'utilisation d'un itérateur, le code client pourrait devoir accéder directement à la structure interne de chaque collection, ce qui introduit une forte dépendance et rend le code client sensible aux changements de la structure interne des collections.
- **Code Client Non Générique :** Si le code client est spécifique à une collection particulière, il ne pourra pas facilement s'adapter à de nouveaux types de collections sans modification significative.

III Définition

Le patron de conception Itérateur est un modèle de conception comportementale qui propose une manière uniforme de parcourir séquentiellement les éléments d'une collection sans exposer la structure interne de celle-ci. Il fait partie des 23 patrons de conception originaux présentés dans le livre "Design Patterns : Elements of Reusable Object-Oriented Software" écrit par Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (le Gang of Four).

III.1 Objectifs

L'objectif principal du patron Itérateur est de fournir une interface standard pour traverser une collection d'objets, indépendamment de la manière dont la collection est mise en œuvre. Cela permet au code client d'itérer sur différentes collections de manière cohérente, sans avoir besoin de connaître la structure interne de chaque collection.

IV Solution

Le but du patron de conception itérateur est d'extraire le comportement qui permet de parcourir une collection et de le mettre dans un objet que l'on nomme itérateur.

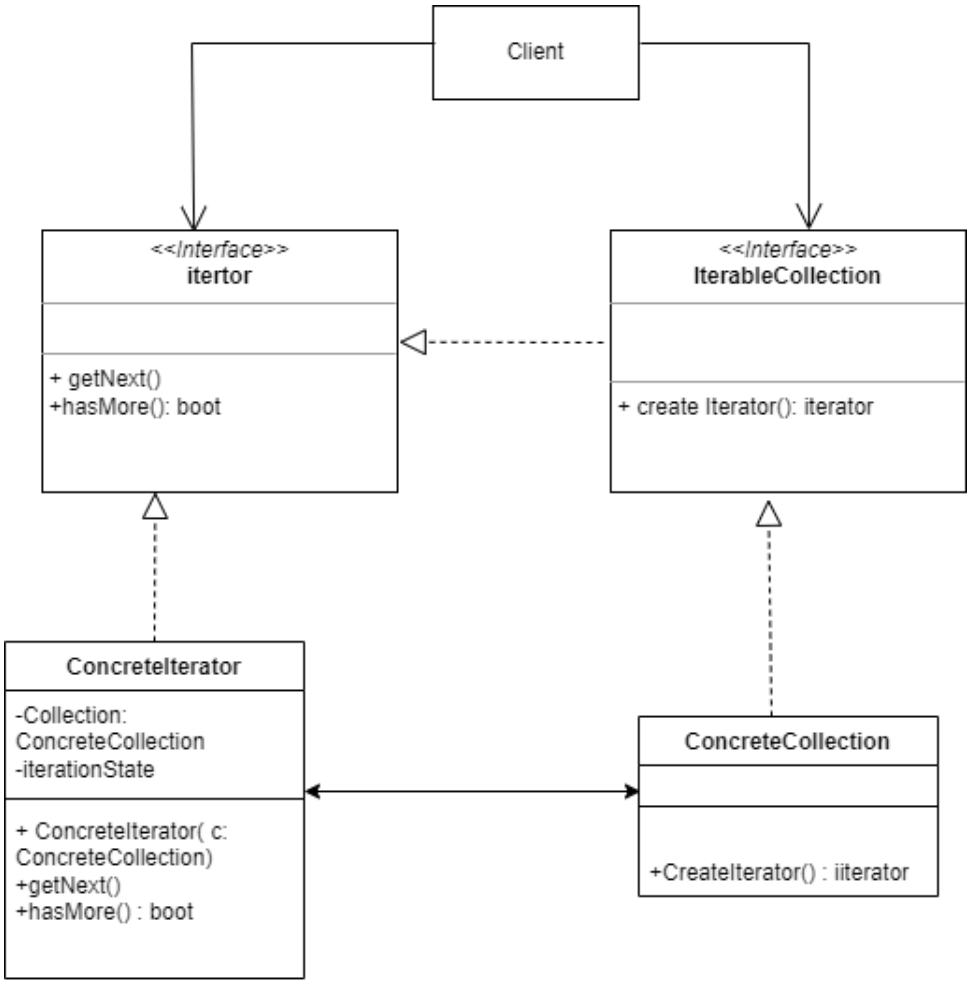
En plus d'implémenter l'algorithme de parcours, un objet itérateur encapsule tous les détails comme la position actuelle et le nombre d'éléments restants avant d'atteindre la fin. Grâce à cela, plusieurs itérateurs peuvent parcourir une même collection simultanément et indépendamment les uns des autres.

En général, les itérateurs fournissent une méthode principale pour récupérer les éléments d'une collection. Le client peut appeler la méthode en continu jusqu'à ce qu'elle ne retourne plus rien, ce qui signifie que l'itérateur a parcouru tous les éléments.

Les itérateurs doivent tous implémenter la même interface. Le code client est ainsi compatible avec tous les types de collections et tous les algorithmes de parcours, tant que l'itérateur adéquat existe. Si vous voulez effectuer un parcours un peu spécial dans une collection, il vous suffit de créer une nouvelle classe itérateur, sans toucher à la collection ni au client.

V Structure

V.1 Représentation graphique



V.2 Liste des participants

- **L'interface Itérateur** déclare les opérations nécessaires au parcours d'une collection : récupérer le prochain élément, donner la position actuelle, recommencer la boucle depuis le début, etc.
- **Les Itérateurs Concrets**implémentent les algorithmes qui servent au parcours d'une collection. L'objet itérateur doit garder la trace du parcours actuel. Grâce à cela, plusieurs itérateurs peuvent parcourir la même collection de manière indépendante.
- **L'interface Collection** déclare une ou plusieurs méthodes pour récupérer des itérateurs compatibles avec la collection. Le type de retour de la méthode

doit être l'interface de l'itérateur afin de permettre aux collections concrètes de renvoyer tous types d'itérateurs.

- **Les Collections Concrètes**renvoient les nouvelles instances de la classe concrète de l'itérateur lorsque le client en demande une. Vous vous demandez peut-être où l'on doit mettre le reste du code de la collection. Ne vous inquiétez pas, il devrait être dans la même classe. Ces détails ne sont pas très importants pour ce patron de conception, je me permets donc de les omettre.

- **Le Client** manipule les collections et les itérateurs grâce à leurs interfaces. Ceci permet de ne pas coupler le client avec les classes concrètes et d'utiliser différents itérateurs et collections avec le même code client.

  En général, les clients ne créent pas les itérateurs, ils les récupèrent auprès des collections. Mais dans certains cas, le client peut en créer un directement. Le client peut par exemple définir son propre itérateur spécial.

V.3 Domaines d'application

Le patron Itérateur peut être appliqué pour simplifier le processus de parcours des résultats d'une requête dans une base de données. Plutôt que d'exposer la structure interne de la base de données au code client, un itérateur peut être utilisé pour encapsuler le processus d'itération sur les enregistrements de la requête.

VI Exemple

Gestion d'une Bibliothèque avec le Pattern Iterator

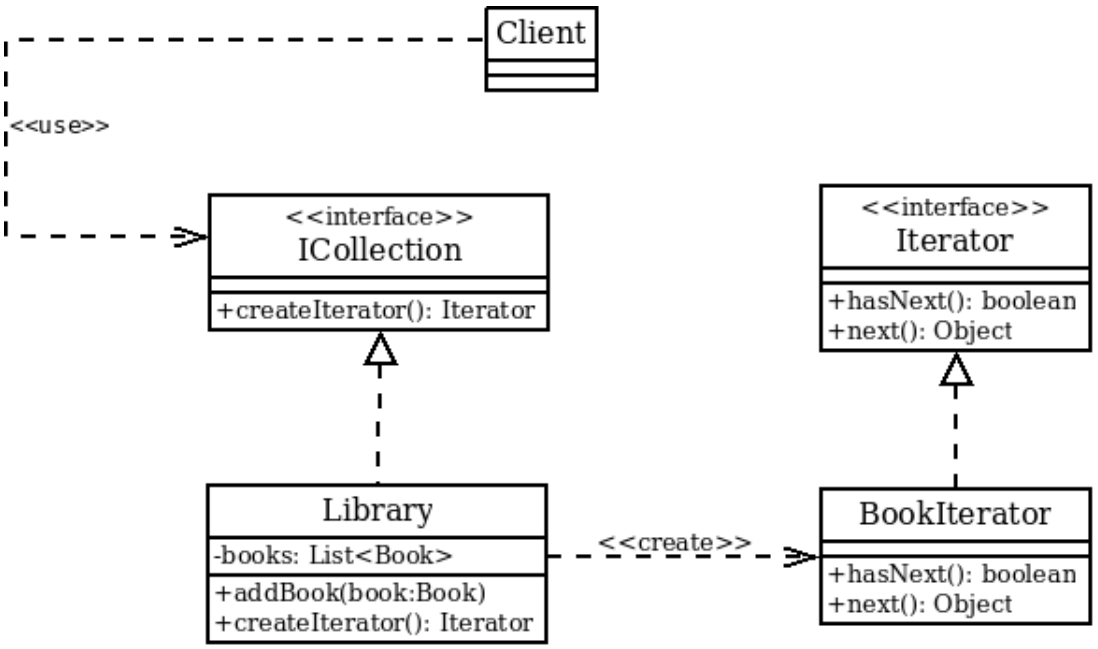
Contexte : Vous travaillez sur un projet de gestion de bibliothèque, où vous devez mettre en place une structure permettant de stocker des livres et de fournir une manière générique de parcourir cette collection.

Objectif : Concevoir un système de gestion de bibliothèque utilisant le pattern Iterator afin de permettre une itération facile et indépendante de la représentation interne de la collection de livres.

Contraintes :

- La solution doit utiliser le pattern Iterator pour assurer une itération propre et indépendante de la structure interne de la collection.
- Les classes doivent respecter le principe de séparation des préoccupations, assurant une bonne modularité et réutilisabilité du code.

Modélisation



Code source

ICollection.java

```java
public interface ICollection {
    public Iterator createIterator();

    // ------------------------------------
    public void addBook(Book book);
}
```

Library.java

```java
import java.util.ArrayList;
import java.util.List;

public class Library implements ICollection {
    private List<Book> books;

    public Library() {
        this.books = new ArrayList<>();
    }

    public void addBook(Book book) {
        books.add(book);
    }

    @Override
    public Iterator createIterator() {
        return new BookIterator(books);
    }
}
```

Iterator.java

```java
interface Iterator {
    boolean hasNext();
    Object next();
}
```

BookIterator.java

```java
import java.util.List;

public class BookIterator implements Iterator {
    private List<Book> books;
    private int position;

    public BookIterator(List<Book> books) {
        this.books = books;
        this.position = 0;
    }

    @Override
    public boolean hasNext() {
        return position < books.size();
    }

    @Override
    public Object next() {
        if (hasNext()) {
            Book book = books.get(position);
            position++;
            return book;
        }
        return null;
    }
}
```

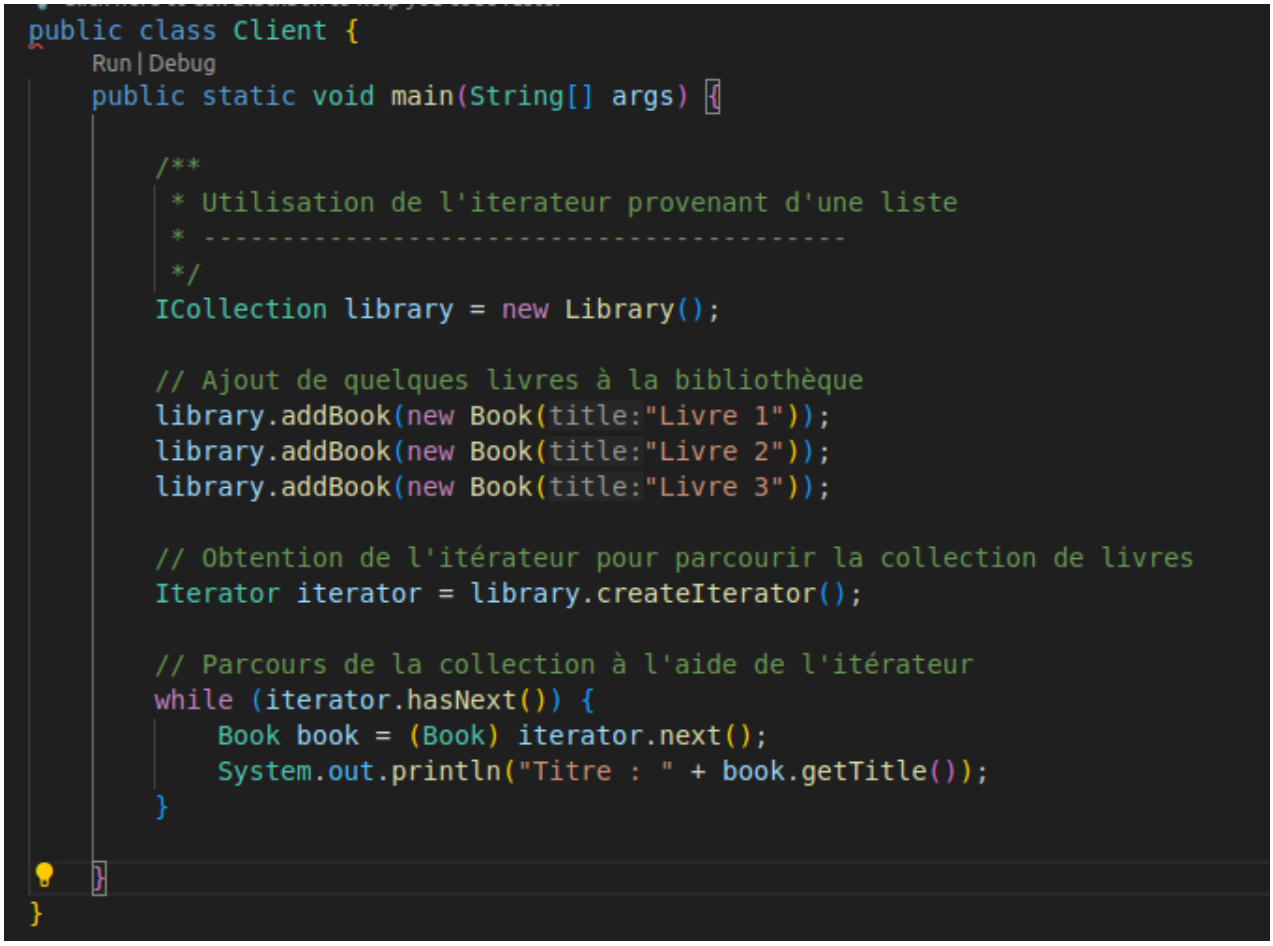
Book.java

```java
public class Book {
    private String title;

    public Book(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }
}
```

Client.java



```java
public class Client {
    Run | Debug
    public static void main(String[] args) {

        /**
         * Utilisation de l'iterateur provenant d'une liste
         * ----------------------------------------
         */
        ICollection library = new Library();

        // Ajout de quelques livres à la bibliothèque
        library.addBook(new Book(title:"Livre 1"));
        library.addBook(new Book(title:"Livre 2"));
        library.addBook(new Book(title:"Livre 3"));

        // Obtention de l'itérateur pour parcourir la collection de livres
        Iterator iterator = library.createIterator();

        // Parcours de la collection à l'aide de l'itérateur
        while (iterator.hasNext()) {
            Book book = (Book) iterator.next();
            System.out.println("Titre : " + book.getTitle());
        }

    }
}
```

Conclusion

Le patron de conception Itérateur offre une approche flexible et élégante pour traverser séquentiellement les éléments d'une collection, tout en maintenant une séparation claire entre le code client et la structure de la collection. Son utilisation améliore la maintenabilité, la réutilisabilité et la flexibilité du code, ce qui en fait un outil précieux dans la boîte à outils des concepteurs logiciels.