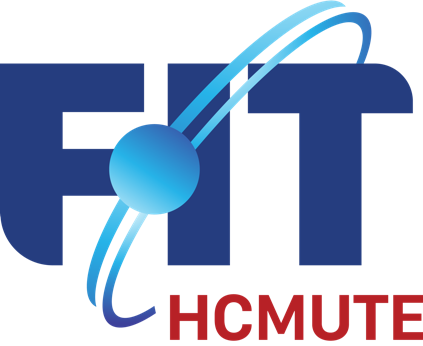
**A white rectangular frame with blue border

Description automatically generatedTRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HCM**

**KHOA CÔNG NGHỆ THÔNG TIN**



**BÁO CÁO**

*Đề tài:*

**ITERATOR PATTERN**

**Môn:** Mẫu Thiết Kế Phần Mềm

**Mã lớp:** DEPA330879\_03CLC

**GVHD:** ThS. Nguyễn Minh Đạo

**SVTH:** Nguyễn Hà Quỳnh Giao

**MSSV:** 21110171

*TP. Hồ Chí Minh, tháng 04 năm 2024*

MỤC LỤC

[**MỞ ĐẦU** 1](#_Toc166107042)

[**1.** **Khái niệm:** 1](#_Toc166107043)

[**2.** **Cấu trúc:** 1](#_Toc166107044)

[**3.** **Trường hợp sử dụng:** 1](#_Toc166107045)

[**4.** **Ưu điểm:** 1](#_Toc166107046)

[**5.** **Nhược điểm:** 2](#_Toc166107047)

[**THIẾT KẾ** 3](#_Toc166107048)

[**1.** **Ý tưởng thiết kế:** 3](#_Toc166107049)

[**2.** **Giải pháp sử dụng Iterator Pattern:** 3](#_Toc166107050)

[**3.** **Thiết kế các lớp:** 4](#_Toc166107051)

[**3.1.** **Lớp giao diện:** 4](#_Toc166107052)

[**3.2.** **Fibonacci triển khai Iterable và FibonacciIterator của nó:** 4](#_Toc166107053)

[**4.** **Sử dụng:** 5](#_Toc166107054)

[**5.** **Phân tích:** 5](#_Toc166107055)

[**KẾT LUẬN** 7](#_Toc166107056)

[**TÀI LIỆU THAM KHẢO** 8](#_Toc166107057)

**MỞ ĐẦU**

1. **Khái niệm:**

Iterator là một mẫu thiết kế thuộc nhóm Behavioral trong lập trình, được sử dụng để truy cập các phần tử của một tập hợp một cách tuần tự mà không cần biết cụ thể về cấu trúc nội bộ của tập hợp đó. Điều này giúp tách biệt trách nhiệm giữa việc duy trì dữ liệu và việc làm việc với dữ liệu, và cho phép xử lý nhiều loại tập hợp khác nhau bằng cách sử dụng cùng một phương pháp và cách thức định sẵn. Iterator tuân theo nguyên lý - Single Responsibility Principle, nơi mỗi lớp chỉ nên có một nhiệm vụ duy nhất.

1. **Cấu trúc:**

Một mẫu Iterator bao gồm:

* Lớp interface:

+ Iterator: chứa các phương thức tương tác với collection như hasNext() – kiểm tra phần tử nào còn tồn tại trong collection không, next() – phần tử tiếp theo trong collection, remove() – xóa phần tử hiện tại ra khỏi collection.

+ Iterable: chứa phương thức iterator(), cung cấp khả năng iterator cho đối tượng, trả về một Iterator.

* Đối tượng triển khai Iterable: triển khai phương thức iterator() trả về một đối tượng iterator mới.
* Iterator của đối tượng: triển khai các hàm trong iterator.

1. **Trường hợp sử dụng:**

* Muốn ẩn đi cấu trúc phức tạp của collection và tạo ra một giao diện đơn giản để truy cập các phần tử của nó.
* Giảm sự trùng lặp của mã trong ứng dụng, làm cho mã trở nên gọn gàng và dễ bảo trì hơn.
* Có khả năng duyệt qua các cấu trúc dữ liệu khác nhau hoặc khi loại cấu trúc này không được biết trước.

1. **Ưu điểm:**

* Đảm bảo nguyên tắc Single responsibility principle: ẩn đi hàm triển khai các phương thức của tập hợp và duyệt qua các phần tử theo từng lớp riêng lẻ.
* Đảm bảo nguyên tắc Open/Closed Principle: implement các loại collection mới và iterator mới, sau đó chuyển chúng vào code hiện có mà không vi phạm bất cứ nguyên tắc gì.
* Truy cập song song trên cùng một tập hợp vì mỗi đối tượng iterator có chứa trạng thái riêng của nó.

1. **Nhược điểm:**

* Sử dụng iterator có thể kém hiệu quả hơn so với việc duyệt qua các phần tử của bộ sưu tập một cách trực tiếp.
* Có thể không cần thiết nếu ứng dụng chỉ hoạt động với các collection đơn giản.

**THIẾT KẾ**

1. **Ý tưởng thiết kế:**

Trong bài toán này, hệ thống xây dựng cho phép tạo và duyệt chuỗi Fibonacci một cách dễ dàng và linh hoạt.

Ngôn ngữ lập trình: Java

1. **Giải pháp sử dụng Iterator Pattern:**

A diagram of a computer

Description automatically generated

* Lớp interface:
* + Iterator: chứa các phương thức tương tác với collection như hasNext() – kiểm tra phần tử nào còn tồn tại trong collection không, next() – phần tử tiếp theo trong collection.
* + Iterable: chứa phương thức iterator() - cung cấp khả năng iterator cho đối tượng, trả về một Iterator.
* Fibonacci triển khai Iterable: triển khai phương thức iterator() trả về một đối tượng iterator mới, chứa thuộc tính maxCount là là phần tử lớn nhất mà có thể cung cấp (phần tử thứ maxCount).
* FibonacciIterator trong Fibonacci: triển khai các hàm trongg iterator.

+ hasNext: nếu current hiện tại bé hơn maxCount.

+ next: tính và trả về phần tử tiếp theo.

1. **Thiết kế các lớp:**
   1. **Lớp giao diện:**

Iterator.java

package org.example;  
  
public interface Iterator<T> {  
 public boolean hasNext();  
 public T next();  
}

Iterable.java

package org.example;  
  
public interface Iterable<T> {  
 public Iterator<T> iterator();  
}

* 1. **Fibonacci triển khai Iterable và FibonacciIterator của nó:**

Fibonacci.java:

package org.example;  
  
public class Fibonacci implements Iterable<Integer> {  
 private int maxCount;  
  
 public Fibonacci(int maxCount){  
 this.maxCount = maxCount;  
 }  
  
 @Override  
 public Iterator<Integer> iterator() {  
 return new FibonacciIterator();  
 }  
  
 public class FibonacciIterator implements Iterator<Integer>{  
 private int current = 0;  
 private int next = 1;  
 private int count = 0;  
  
 @Override  
 public boolean hasNext() {  
 return this.count < maxCount;  
 }  
  
 @Override  
 public Integer next() {  
 int result = this.current;  
 int temp = this.current + this.next;  
 this.current = this.next;  
 this.next = temp;  
 this.count++;  
 return result;  
 }  
 }  
}

}

1. **Sử dụng:**

Main.java

package org.example;  
  
public class Main {  
 public static void main(String[] args) {   
 Fibonacci fibonacci = new Fibonacci(10);  
 Iterator<Integer> iterator = fibonacci.iterator();  
 while (iterator.hasNext()){  
 System.*out*.println(iterator.next().toString());  
 }  
 }  
}

1. **Phân tích:**

Kết quả sau khi sử dụng Iterator pattern là một cách linh hoạt và dễ bảo trì để duyệt qua các phần tử của chuỗi Fibonacci. Bằng cách triển khai các interface Iterator và Iterable, chúng ta tạo ra một cách tiếp cận chung cho việc duyệt qua các tập hợp các phần tử mà không cần biết cụ thể về cấu trúc nội bộ của chúng.

**KẾT LUẬN**

Trong ví dụ về Iterator Pattern, đã cho thấy việc sử dụng Iterator pattern mang lại nhiều lợi ích quan trọng. Bằng cách tách biệt trách nhiệm giữa việc duy trì dữ liệu và việc duyệt qua nó, chúng ta tạo ra một cách tiếp cận linh hoạt và dễ bảo trì cho việc duyệt qua các tập hợp các phần tử. Đồng thời, việc triển khai Iterator pattern cũng tuân theo các nguyên tắc thiết kế quan trọng như Single Responsibility Principle (SRP) và Open/Closed Principle (OCP), giúp mã nguồn trở nên linh hoạt và dễ mở rộng.

**TÀI LIỆU THAM KHẢO**

[1] Hoàng Đinh. (21-11-2021). Iterator Design Pattern - Trợ thủ đắc lực của Developers. Viblo. <https://viblo.asia/p/iterator-design-pattern-tro-thu-dac-luc-cua-developers-jvElaNwY5kw> (ngày truy cập 29-04-2024)