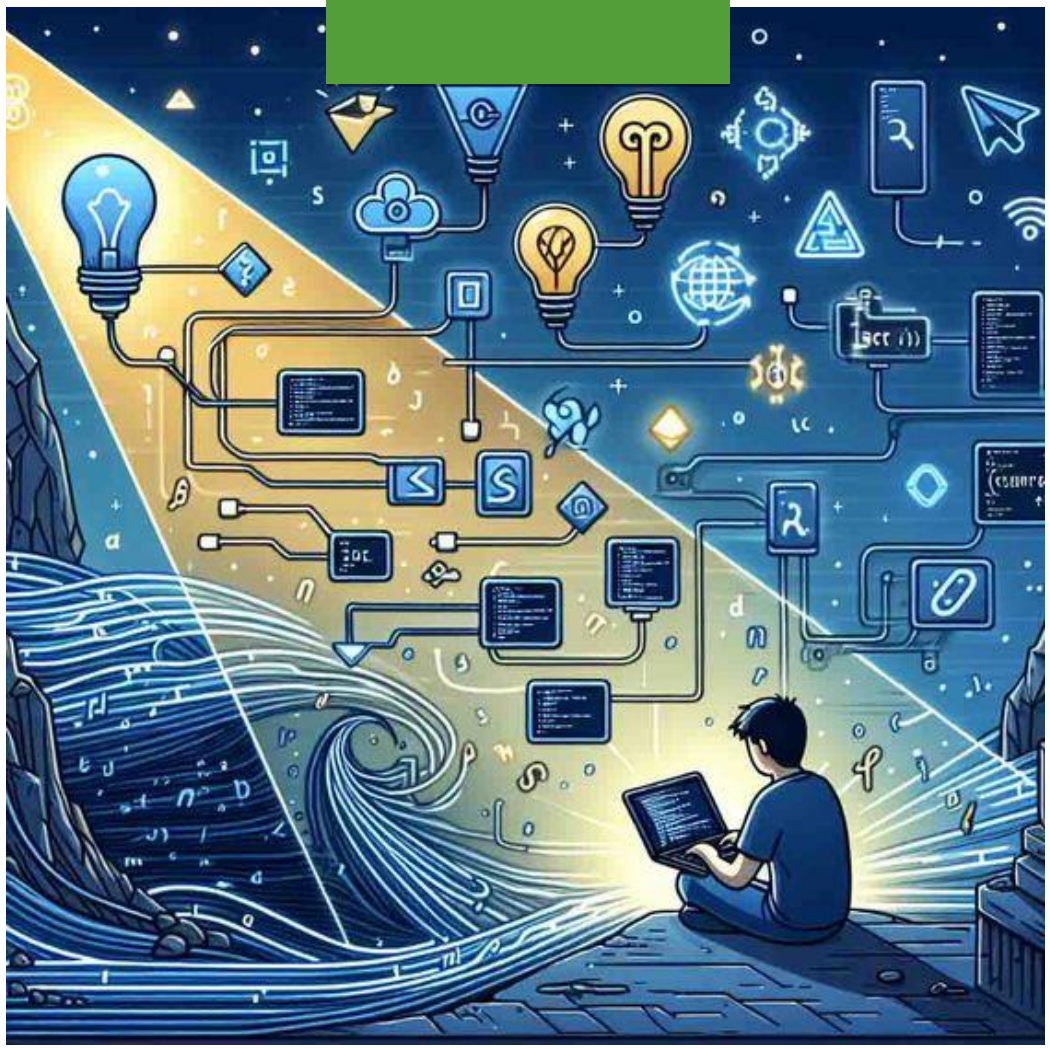


PATRON ITERATOR

Patrones de comportamiento

Por Pilar Fdez Adillo

PATRONES DE COMPORTAMIENTO

1

INTERACCION

2

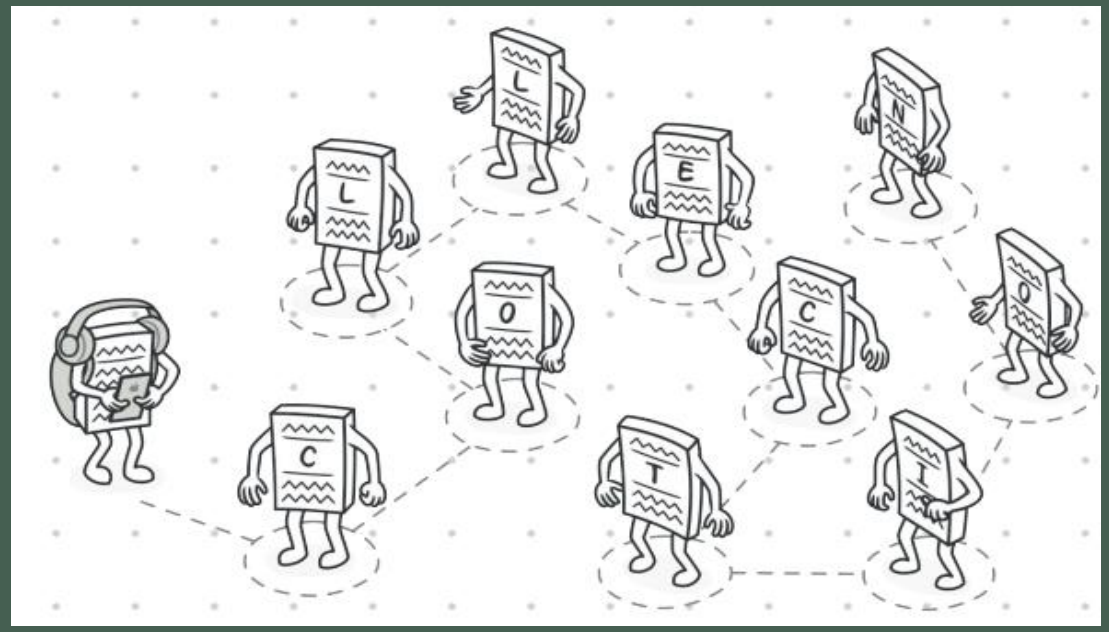
FLEXIBILIDAD

3

REUTILIZAR

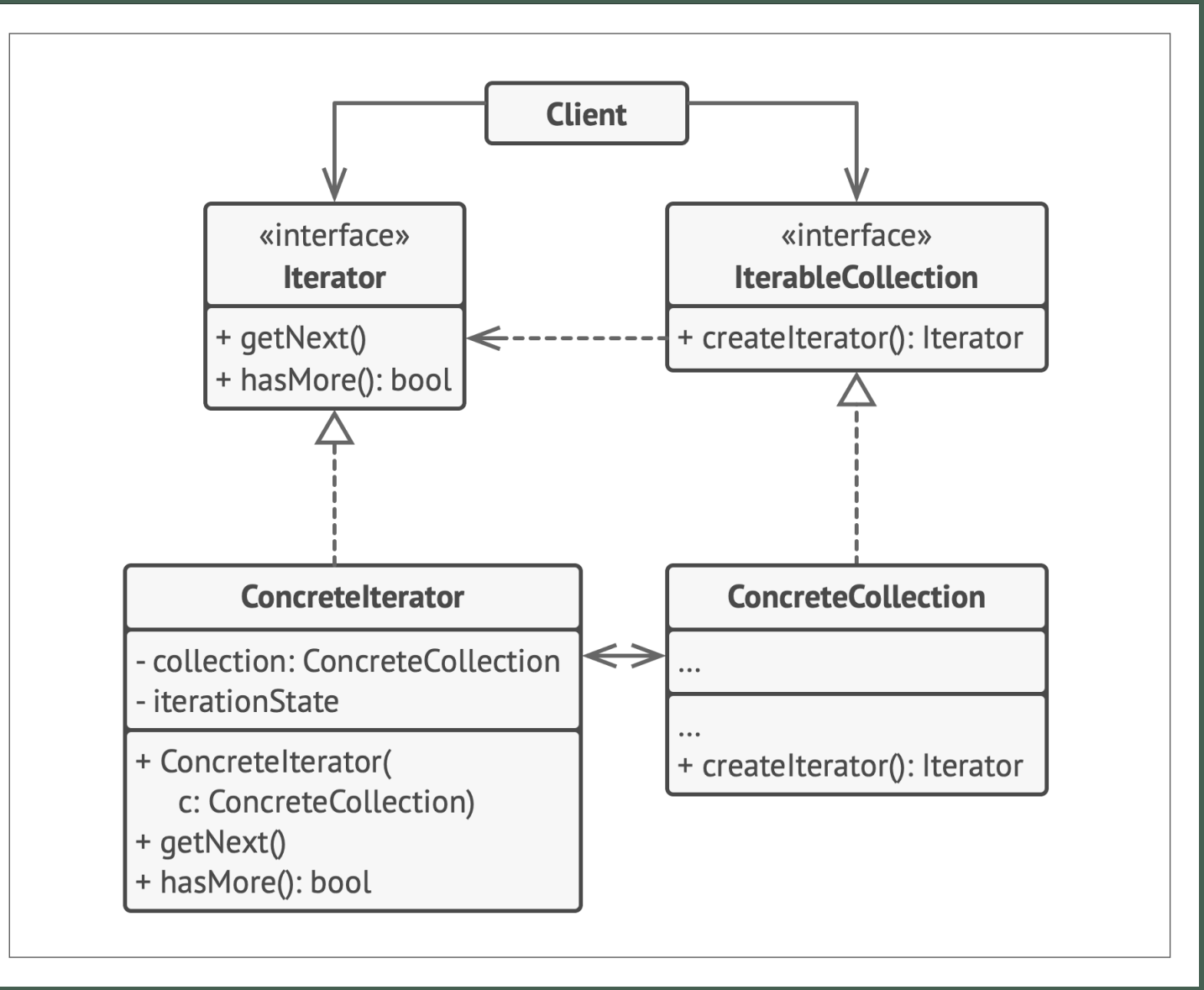
Patrón Iterator

- Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin saber cómo están organizados internamente
- A los elementos de una colección se puede acceder de manera uniforme, sin importar el tipo de colección



Aplicabilidad

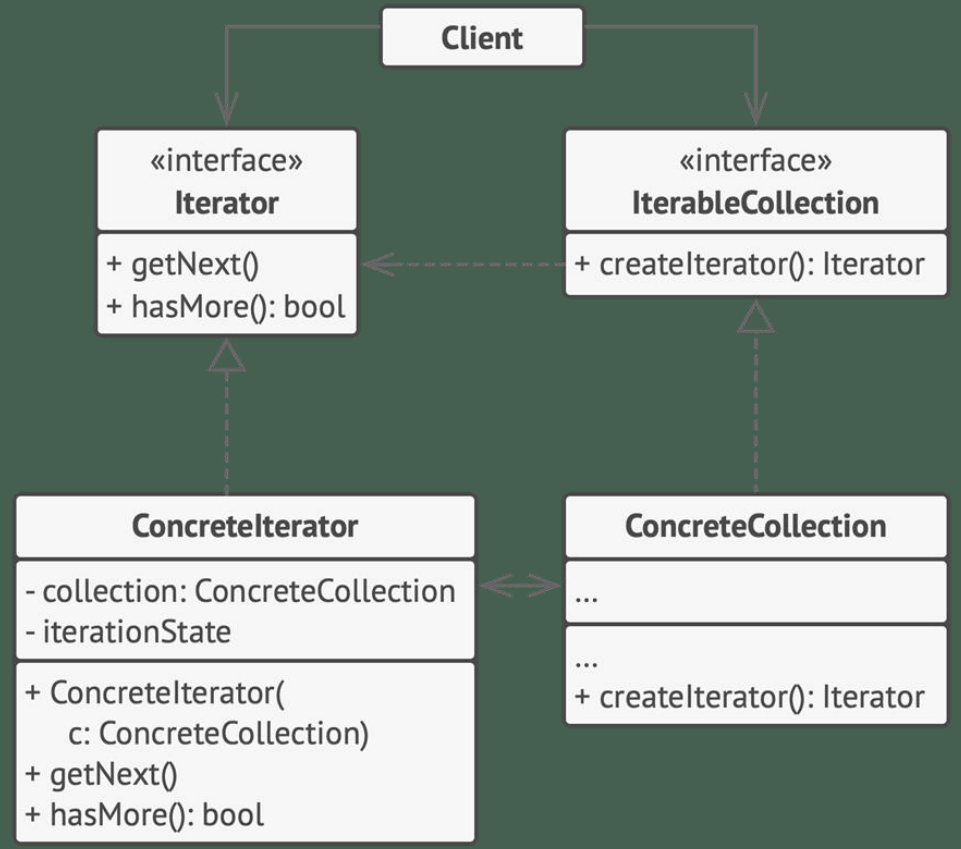
- Una clase necesita acceder al contenido de una colección sin tener que exponer su representación interna.
- Una clase necesita un modo uniforme de acceder al contenido de varias colecciones.
- Cuando se necesita soportar múltiples recorridos de una colección.


Client
«interface»
Iterator
+ getNext()
+ hasMore(): bool
«interface»
IterableCollection
+ createIterator(): Iterator
ConcreteIterator
- collection: ConcreteCollection
- iterationState
+ ConcreteIterator(
c: ConcreteCollection)
+ getNext()
+ hasMore(): bool
ConcreteCollection
...
...
+ createIterator(): Iterator

ESTRUCTURA

Desglose de la estructura

- **Iterator:** Define las operaciones necesarias para recorrer la colección
- **ConcreteIterator:** Implementa la interfaz del iterador y mantiene el estado de la iteración.
- **I.Collection:** Define una interfaz para crear un iterador.
- **ConcreteCollection:** Implementa el método para devolver una instancia concreta del iterador y almacena los elementos a recorrer



¿Como funciona?

```java
package src;

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

```java
package src;

public class NumberIterator implements Iterator {
    private int[] numbers;
    private int position = 0;

    public NumberIterator(int[] numbers) {
        this.numbers = numbers;
    }

    public boolean hasNext() {
        return position < numbers.length;
    }

    public Object next() {
        return numbers[position++];
    }
}
```

¿Como funciona?

```java
package src;

public interface IterableCollection {
    Iterator createIterator();
}
```

```java
package src;

public class NumberCollection implements IterableCollection {
    private int[] numbers;

    public NumberCollection(int[] numbers) {
        this.numbers = numbers;
    }

    @Override
    public Iterator createIterator() {
        return new NumberIterator(numbers);
    }
}
```

¿Como funciona?

```java
package src;

public class Main {
    Run | Debug
    public static void main(String[] args) {
        // Crear una colección de números
        int[] numbers = {10, 20, 30, 40, 50};
        NumberCollection numberCollection = new NumberCollection(numbers);

        // Obtener un iterador para la colección de números
        Iterator iterator = numberCollection.createIterator();

        // Recorrer la colección de números
        while (iterator.hasNext()) {
            int number = (int) iterator.next();
            System.out.println(number);
        }
    }
}
```

```
10
20
30
40
50
PS C:\Users\adill\numeros_iterator>
```

Ventajas

- **Independencia**: Permite recorrer distintas colecciones de forma homogénea.
- **Encapsulamiento**: No es necesario exponer la estructura interna de la colección.
- **Simplicidad**: Hace que el código del cliente sea más simple.

Desventajas

- **Sobrecarga**: Añade complejidad innecesaria en colecciones simples.
- **Posible Incompatibilidad**: Si la colección cambia durante la iteración, podría ser necesario manejar condiciones adicionales.

Patrones relacionados

- **Factory Method**: Utilizado para crear iteradores específicos para cada tipo de colección.
- **Memento**: Permite guardar el estado de un Iterator y reanudar la iteración más tarde.

MUCHAS GRACIAS