ITERATOR

Patrones de Diseño

Alumnos: Calcagno,Camila
Martínez,Agustín
Passalacqua,Santiago

Motivación

El patron de diseño iterator define una serie de interfaces que permite acceder a los elementos de una secuencia de objetos sin exponer su estructura interna (por ejemplo una lista). Es posible que se desee recorrer la lista de diferentes maneras, dependiendo de lo que se quiera lograr, pero no añadir operaciones a la lista por cada tipo de recorrido.

La idea de este patron es proveer los metodos de acceso y recorrido de una lista de objetos y poner esto en un objeto iterator. Se debe poder realizar varios recorridos simultaneamente.

Algunos ejemplos...

Por ejemplo una clase list puede crear una instancia de listiterator y utilizar los metodos que esta provee, y acceder a los elemntos de la lista.

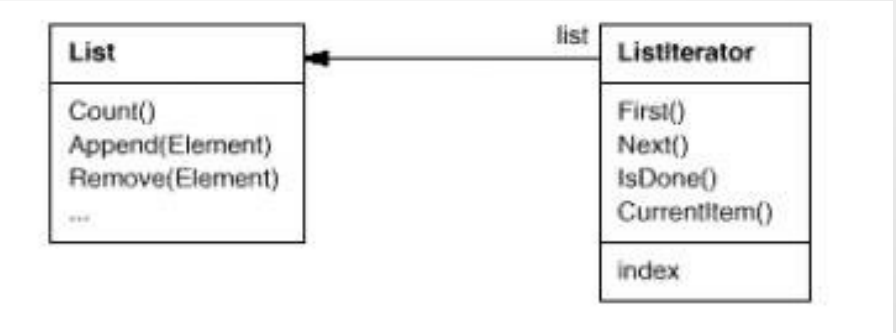
Los metodos que provee listiterator son:

currentitem() : devuelve el elemento actual de la lista

first() : inicializa el elemento actual al primer elemento

next(): avanza el elemento actual al sigiente elemento

isDone(): devuelve true o false según se halla terminado el recorrido o no.



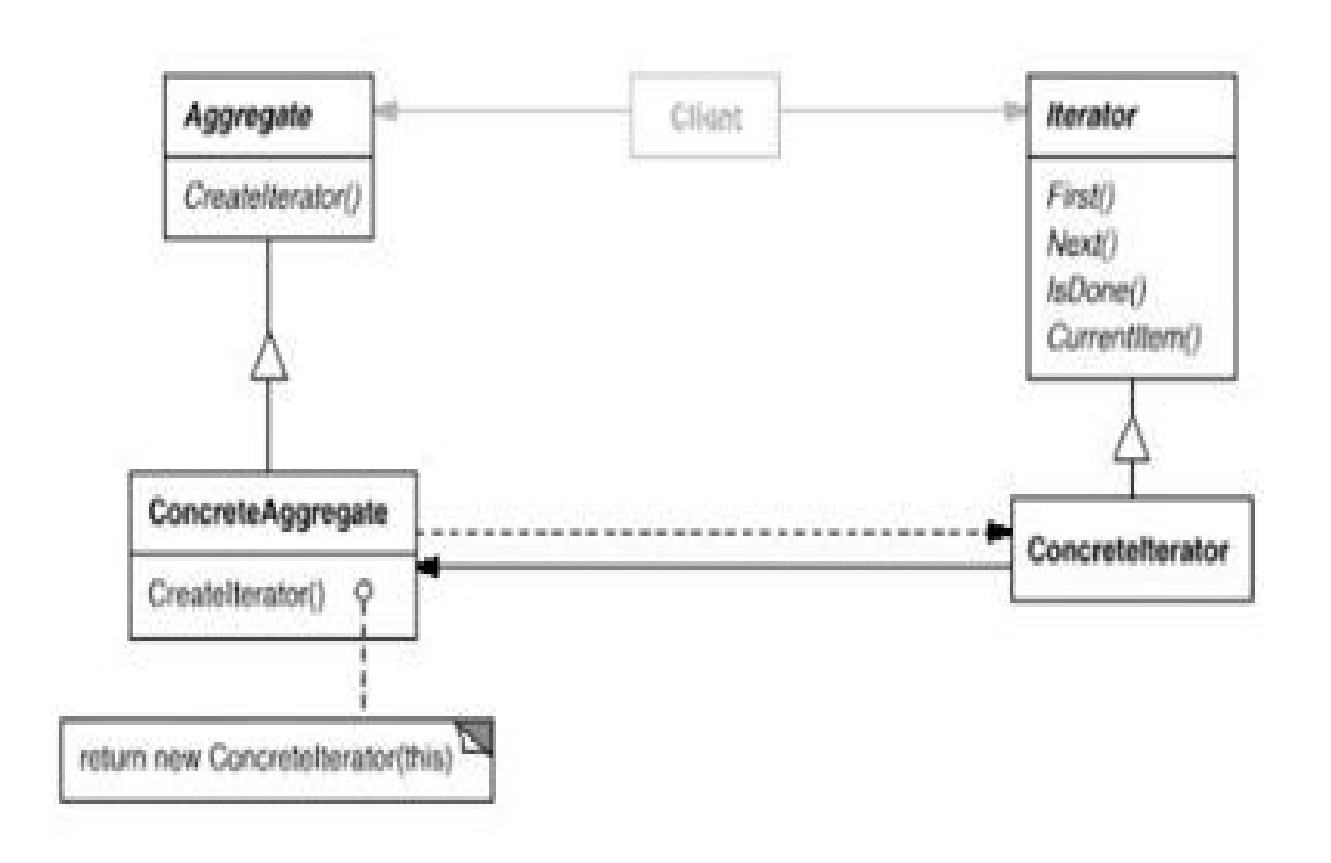
Aplicabilidad

-El patrón iterator se utiliza para acceder al contenido de un conjunto de objetos sin exponer su representacion interna.

-Soporta multiples recorridos sobre un conjunto de objetos.

-Proporcionar una interfaz uniforme para recorrer las diferentes estructuras de objetos (es decir para proveer iteración polimorfica).

Estructura


Aggregate
CreateIterator()
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
ConcreteAggregate
CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)

Clases participantes..

Iterator: define una interfaz para acceder y recorrer elementos.

ConcreteIterator: implementa la interfaz iterator. Realiza un seguimiento de la posicion actual del objeto en el recorrido.

Aggregate: Define una interfaz para crear un objeto iterator.

ConcreteAggregate: implementa una interfaz de creación

del Iterator para devolver la instancia de ConcreteIterator apropiada.

Interface Iterator Iterable

```
public interface Iterator<E> {
    //Comprueba si existe un elemento a continuación.
    boolean hasNext();

    //Devuelve el siguiente elemento de la iteración.
    E next();
}

public interface Iterable<E> {
    //Devuelve un iterador.
    Iterator<E> iterator();
}
```

Uso del Iterador

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

        Iterator it = new ImplementacionIterador();

        while(it.hasNext())
            it.next();
    }
}
```

```java
public class ZooBarcelona {
    private List<String> animales;

    public ZooBarcelona(){
        super();
    animales = new ArrayList<String>();
    }

    //En el recuento devuelven una lista.
    public List<String> getAnimales(){
        animales.add("gorilas");
        animales.add("osos");
        animales.add("hamsters");

        return animales;
    }
}
```

```java
public class ZooMadrid {
    private String[] animales;

    public ZooMadrid(){
        super();
    animales = new String[3];
    }

    //En el recuento devuelven un vector.
    public String[] getAnimales(){
        animales[0] = "elefantes";
        animales[1] = "hipopotamos";
        animales[2] = "jirafas";

        return animales;
    }
}
```

Solución Mala

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

		ZooMadrid zooMadrid = new ZooMadrid();
		String[] animalesMadrid = zooMadrid.getAnimales();

		//Animales del zoo de Madrid
		for(int i = 0; i < animalesMadrid.length; i++)
			System.out.println(animalesMadrid[i]);

		ZooBarcelona zooBarcelona = new ZooBarcelona();
		List<String> animalesBarcelona = zooBarcelona.getAnimales();

		//Animales del zoo de Barcelona
		for(int i = 0; i < animalesBarcelona.size(); i++)
			System.out.println(animalesBarcelona.get(i));

	}
}
```

Implementación Correcta

```
public class IteradorZooBarcelona implements
Iterator<String>{
        private int pos = 0;
        private List<String> animales;

        public IteradorZooBarcelona(List<String>
        animales){
                super();
                this.animales = animales;
        }
        @Override
        public boolean hasNext() {
        return pos+1 <= animales.size();
        }
        @Override
        public String next() {
                String animal = animales.get(pos);
                pos++;
                return animal;
        }}
```

```
public class IteradorZooMadrid implements
Iterator<String>{
        private int pos = 0;
        private String[] animales;

        public IteradorZooMadrid(String[] animales){
                super();
                this.animales = animales;
        }
        @Override
        public boolean hasNext() {
        return pos+1 <= animales.length
                        && animales[pos] != null;
        }
        @Override
        public String next() {
                String animal = animales[pos];
                pos++;
                return animal;
        }}
```

Acceso Común

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

        ZooMadrid zooMadrid = new ZooMadrid();
        ZooBarcelona zooBarcelona = new ZooBarcelona();

        mostrarAnimales(zooMadrid.iterator());
        mostrarAnimales(zooBarcelona.iterator());
    }

    //Con este método podemos mostrar tantos zoos como tengamos
    public static void mostrarAnimales(Iterator<String> iterator){
        while(iterator.hasNext())
            System.out.println(iterator.next());

    }
}
```

Ventajas y Desventajas

- Conseguir una forma común de acceder a los datos independientemente de si se ha utilizado un arreglo o una lista u otra colección.
- Nos proporciona una mayor facilidad a la hora de agregar mas datos.
- Menos código, ya que se extrae el recorrido al objeto iterator.

Conclusiones..

- Se puede hacer mas de un recorrido a la vez sobre una misma colección ya que cada iterator mantiene su propio recorrido.
- Simplifican la interfaz de las colecciones ya que el recorrido de los mismos se definen en una interfaces separada.
- Ahorramos el tiempo de estudiar cada colección para encontrar la manera de recorrerla, ya que en el patrón iterator existe una manera estándar de recorrer todas las colecciones de la misma manera.

Bibliografía

- http://www.uml.org.cn/c++/pdf/designpatterns.pdf
- http://arantxa.ii.uam.es/~eguerra/docencia/0708/03%20Iterator.pdf
- http://es.wikipedia.org/wiki/Iterador_%28patr%C3%B3n_de_dise%C3%B1o%29