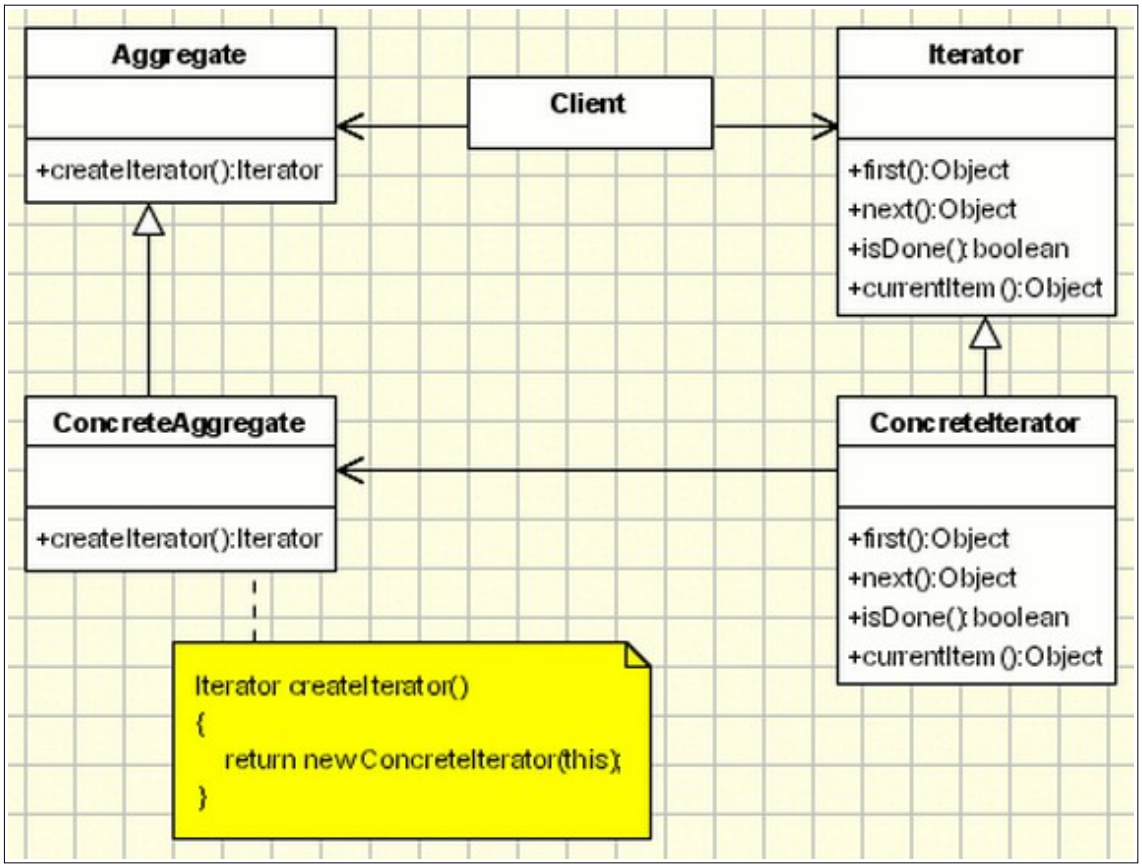
# Iterator

## Diagrama de clases e interfaces




## Intención

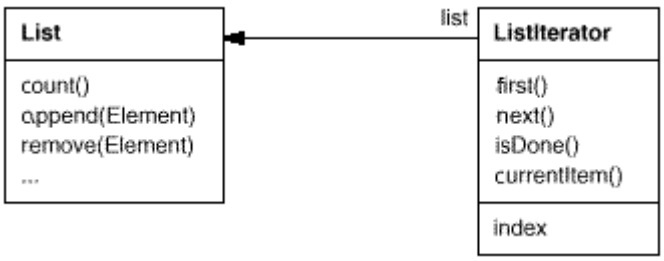
Permitir el acceso secuencial a los elementos de un objeto compuesto (agregado) sin que éste exponga su estructura interna.

## Motivación

Un objeto compuesto, como sería el caso de una lista, debería permitir el acceso secuencial a sus elementos sin exponer su estructura interna. Además, a veces podría interesar recorrer la lista en diferentes formas: en sentido inverso, sólo elemento pares o impares, etc.

La idea principal en este patrón es hacer que la lista no sea la responsable del acceso y recorrido de sus elementos. Para ello se utiliza un objeto iterador que llevará a cabo

este cometido. La clase Iterator define una interfaz para el acceso de los elementos de la lista. Un objeto iterador es responsable de mantener la pista del elemento actual, esto es, siempre sabe qué elementos ya han sido recorridos. Por ejemplo, supongamos las siguientes clases y la relación entre ellas:




Antes de poder instanciar ListIterator tendríamos que proporcionar la lista a recorrer. Una vez que tenemos la instancia de ListIterator podremos acceder secuencialmente a los elementos de List.

ListIterator proporciona las siguientes operaciones:

- currentItem(): Devuelve el elemento actual de la lista.
- first(): Hace que el primer elemento de la lista sea el elemento actual.
- next(): Avanza la posición del elemento actual al siguiente elemento de la lista.
- isDone(): Comprueba si hemos avanzado más allá del último elemento, es decir, si ya hemos terminado de recorrer la lista.

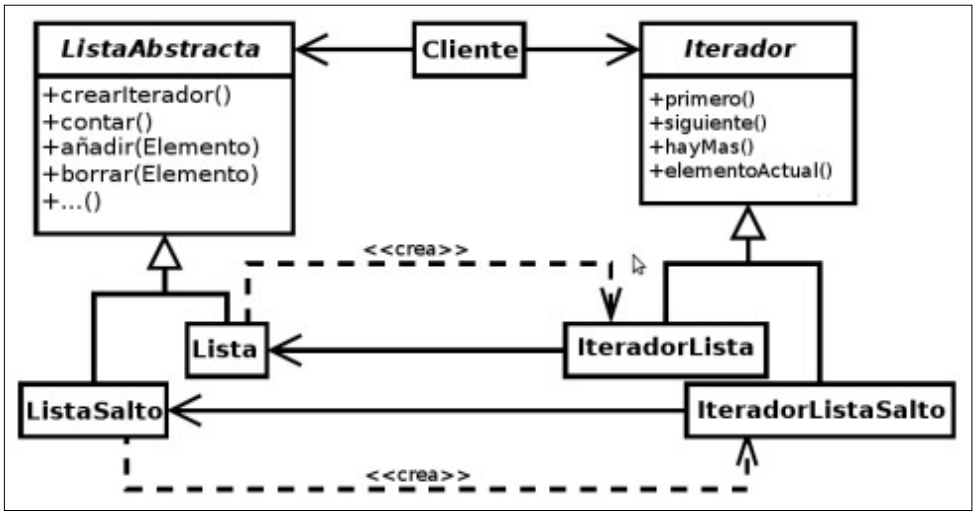
Separar el mecanismo que recorre la lista de la propia lista nos permite definir diferentes iteradores sin que sea necesario enumerarlos en la interfaz de la lista.

Hay que notar que el iterador y la lista están acoplados, y que el código cliente debe conocer que la estructura a recorrer se trata de una lista, y no de ningún otro tipo de estructura contenedora. Sin embargo, sería más ventajoso que se pudiera cambiar el tipo de estructura contenedora sin alterar el código cliente. Esto es posible si se generaliza el concepto de iterador para dar soporte a la iteración polimórfica.

Como ejemplo de esto último, vamos a asumir que tenemos una implementación de una lista llamada SkipList (ListaSalto) (una skiplist es una estructura de datos parecida a un árbol balanceado). Queremos crear una clase cliente que sea capaz de funcionar tanto con objetos List como con objetos SkipList.

También definiremos una clase abstracta llamada AbstractList (LsitaAbstracta), la cual servirá de interfaz para las operaciones de manipulación de las listas. Similarmente, necesitaremos un iterador abstracto que defina las operaciones a realizar por los objetos iteradores. A continuación, definimos diferentes subclases de implementación para el iterador abstracto, cada uno conveniente para cada tipo de lista.

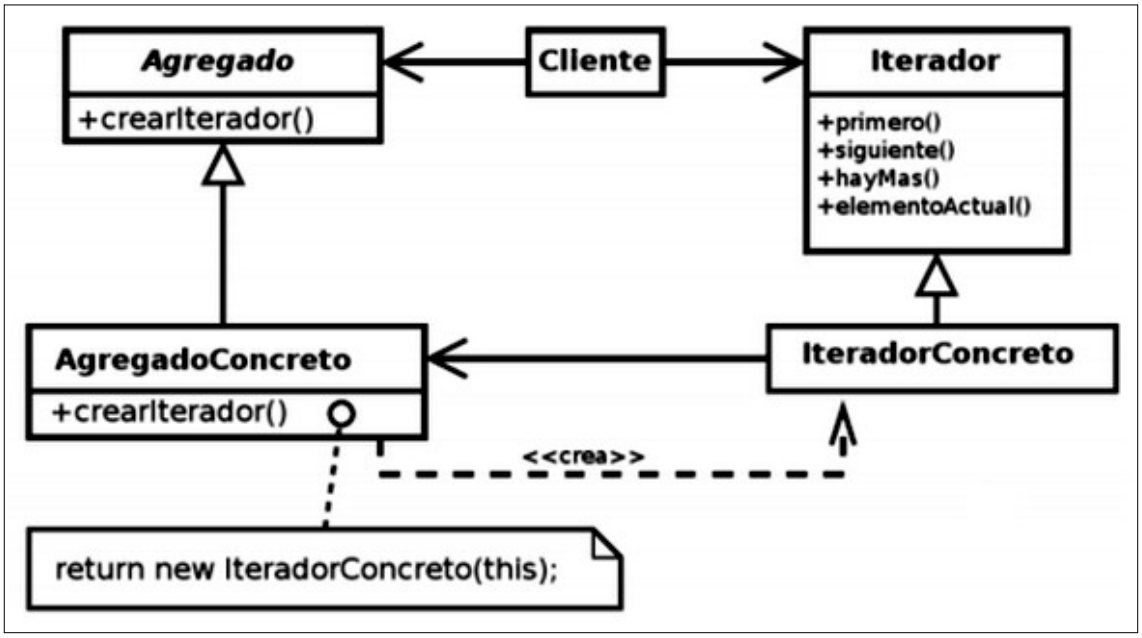
El resultado es el mostrado en la siguiente figura:




El último problema a solucionar es cómo crear el iterador. No podemos instanciar directamente una subclase de Iterador, dado que lo queremos es que el código cliente sea independiente de las subclases de ListaAbstracta. Por tanto, haremos que los propios objetos lista sean los responsables de crear su correspondiente iterador. Esto requiere definir una operación crearIterador() mediante la cual el código cliente pueda obtener una referencia a un objeto iterador. CrearIterador() es un método de factoría.

## Implementación

El funcionamiento del patrón es el mostrado en el diagrama de clases siguiente:




Las clases participantes en el patrón son las siguientes:

- **Iterador (Iterator):** Define una interfaz para acceder y recorrer los elementos de Agregado.
- **IteradorConcreto (ConcreteIterator)**: Clase que implementa la interfaz Iterador. Implementa las operaciones que permiten acceder a los elementos de Agregado, así como mantiene la posición del elemento actual.
- **Agregado (Aggregate)**: Define una interfaz para crear un objeto iterador. Aunque en la figura no se representa, también define operaciones para trabajar con estructuras compuestas (listas, mapas, etc) del tipo add(), remove(), etc.
- **AgregadoConcreto (ConcreteAggregate):** Implementa la interfaz Agregado.

De la figura anterior hay que observar que la clase AgregadoConcreto crea el iterador y se pasa así misma como parámetro. Esto es necesario para que el iterador pueda operar sobre la estructura compuesta.

## Aplicabilidad y Ejemplos

El patrón Iterador es adecuado cuando:

- Se necesita acceder a los elementos de un objeto compuesto sin que deba exponerse la representación interna del mismo.
- Se requiere recorrer un objeto compuesto de diversas maneras (secuencialmente de izquierda a derecha, de modo inverso, etc).
- Se necesita proporcionar una interfaz uniforme para recorrer distintas estructuras compuestas (listas, mapas, árboles, etc).

El patrón ofrece mucha flexibilidad, ya que:

- Admite diferentes variantes para recorrer objetos compuestos. Por ejemplo, un analizador sintáctico puede implicar recorrer una estructura tipo árbol en inorden o en preorden. Mediante el patrón Iterador, tan sólo tenemos que tener dos implementaciones, una clase que haga el recorrido en inorden y otra que lo haga en preorden. Además, se pueden definir subclases para soportar nuevas maneras de recorrer una estructura.
- Los iteradores simplifican la interfaz Agregado, ya que ésta sólo se limita a definir las operaciones de gestión de la estructura (add(), remove(), etc) pero delegan las operaciones del recorrido de la misma en los iteradores.
- Un agregado puede tener asociados varios iteradores, cada uno recorriendo simultáneamente el agregado. Cada iterador debe mantener su propio elemento actual.
- Filtros. El comportamiento básico de un iterador es devolver el siguiente elemento de un objeto agregado. Sin embargo, es posible extender este comportamiento para realizar algo más útil, como por ejemplo seleccionar sólo unos determinados elementos en lugar de todos los del objeto agregado. Este filtro puede estar basado en algún tipo de criterio establecido por el código cliente.

Veamos ahora algunos ejemplos.

*Ejemplo 1 – Implementación de Iterator como clase privada*

En este ejemplo creamos un objeto compuesto llamado Secuencia capaz de almacenar elementos de diferente tipo (Números, String's, etc). Por otro lado, definimos una interfaz llamada Iterador y dentro de Secuencia definimos una clase privada llamada IteradorImpl que implementa la interfaz Iterador, responsable de recorrer los elementos de Secuencia. Una clase cliente utiliza la clase Secuencia y la interfaz Iterador para añadir algunos elementos a un objeto Secuencia, recorrerlos y mostrarlos por pantalla.

Veamos el código. Comenzamos por la interfaz que todo iterador debe implementar. Notad que expone las operaciones que permiten saber si se recorrido totalmente la colección -end()-, obtener el elemento actual -current()- y avanzar una posición dentro de la colección -next():

```
package comportamiento.iterator.privado;
public interface Iterador {
	boolean end();
	Object current();
	void next();
}
```

A continuación vemos la clase Secuencia. Esta clase:

- Define un array de objetos -items- que servirá de colección, cuyo tamaño la define el código cliente mediante el constructor de Secuencia.
- Dispone de un método para añadir nuevos elementos a la colección -add().
- Define la clase privada IteradorImpl, la cual encapsula la lógica para recorrer la colección de Secuencia -items.
- Define un método para crear un iterador, abstrayendo por completo al código cliente sobre la implementación del mismo.

```java
package comportamiento.iterator.privado;

public class Secuencia {
	private Object[] items;
	private int next = 0;

	public Secuencia(int size) {
		items = new Object[size];
	}
	public void add(Object x) {
		if (next < items.length)
			items[next++] = x;
	}

	private class IteradorImpl implements Iterador {
		private int i = 0;

		public boolean end() {
			return i == items.length;
		}

		public void next() {
			if (i < items.length)
				i++;
		}

		public Object current() {
			return items[i];
		}
	} // fin clase privada
```

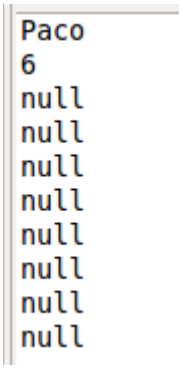
```
	public Iterador iterador() {
		return new IteradorImpl();
	}
}
```

Por último, veamos la clase cliente. Esta clase:

- Crea un objeto Secuencia de 10 elementos, aunque sólo ocupa con datos las dos primeras posiciones.
- Obtiene una referencia a un iterador a partir del objeto Secuencia. Notad que lo único a lo que puede acceder el código cliente es al tipo interfaz Iterador, ya que la clase IteradorImpl es privada. Esto es una muy buena práctica de programación, con muchas ventajas.
- Recorre la colección, mostrando cada uno de sus elementos por pantalla. Notad que en este ejemplo el bucle se hará 10 veces, por lo que obtendremos 8 null's.

```
package comportamiento.iterator.privado.client;

import comportamiento.iterator.privado.Iterador;
import comportamiento.iterator.privado.Secuencia;
public class MainClient {
	public static void main(String[] args) {
		Secuencia secuencia = new Secuencia(10);
		secuencia.add("Paco");
		secuencia.add(6);
		Iterador iterador = secuencia.iterador();
		while (!iterador.end()) {
			System.out.println(iterador.current() + " ");
			iterador.next();
		}
	}
}
```

Salida:




*Ejemplo 2 – Iteradores en Java*

Este ejemplo está compuesto por una serie de fragmentos de código que demuestran las diferentes alternativas que proporciona Java para la iteración.

Alternativas de iteración:

- *for* y *while*; habitualmente usando índices de tipo entero
- La interfaz *java.util.Enumeration*
- La interfaz *java.util.Iterator*, añadida para admitir colecciones en el JDK 1.2
- La interfaz *java.util.Iterable.* Bucle *for* extendido (foreach), añadido en el JDK 1.5
- La clase *java.sql.ResultSet*, utilizada para la recuperación de registros de una base de datos. *java.sql.ResultSet* proporciona un método *next()* para navegar a través de las filas y un conjunto de métodos *get()* para posicionarse en las diferentes columnas.

java.util.Enumeration:

| Método | Valor de retorno | Descripción |
|---|---|---|
| hasMoreElements() | boolean | Comprueba si aún quedan más elementos por recorrer en la colección. |
| nextElement() | Object | Devuelve el siguiente elemento en la colección. |

Diferentes clases del API de Java implementan la interfaz *java.util.Enumeration* . Por ejemplo, la clase *java.util.Vector* proporciona el método:

*public final synchronized Enumeration elements()*

el cual devuelve una *Enumeration* de objetos que funciona como un iterador sobre el objeto *java.util.Vector*. Los método *hashMoreElements()* y *nextElement()* pueden entonces utilizarse en la *Enumeration* devuelta para recuperar secuencialmente los elementos del vector. Por ejemplo, para imprimir todos los elementos del vector v:

```
for (Enumeration e = v.elements() ; e.hasMoreElements() ;) {
        System.out.println(e.nextElement());
}
```

java.util.Iterator:

La funcionalidad de *java.util.Enumeration* está duplicada en la interfaz *java.util.Iterator*. No obstante, la segunda aporta un método *remove()* y nombres más breves (luego más cómodos). Por tanto, se recomienda el uso de *java.util.Iterator* sobre j*ava.util.Enumeration*.

| **Método** | **Valor de retorno** | **Descripción** |
|---|---|---|
| hasNext() | boolean | Comprueba si aún quedan más elementos por recorrer en la colección. |
| next() | Object | Devuelve el siguiente elemento en la colección. |
| remove() | void | Elimina de la colección el último elemento devuelto por el iterador. |

Todas las colecciones implementan la interfaz *java.util.Iterator* por lo que proporcionan un método *iterator()* que permite recorrer secuencialmente cada uno de sus elementos. En el siguiente ejemplo se utiliza la clase *java.util.ArrayList* para ilustrar su funcionamiento:

```
List nombres = new ArrayList();
nombres.add("Antonio");
nombres.add("Juan");
nombres.add("Ricardo");
```

```
System.out.println("Estilo de iteracion JDK 1.2:");
for (Iterator it = nombres.iterator(); it.hasNext();) {
        String nombre = (String) it.next();
        System.out.println(nombre);
}
```

java.util.Iterable. Bucle for extendido (foreach)

Las clases que disponen de este tipo de bucles extendidos implementan la interfaz *java.util.Iterable* e implementan el método *iterator()*.

Este tipo de bucle tiene la forma:

*for (Tipo elemento: colección)*

Esto crea un bucle sobre la colección, tomando un elemento en cada iteración (en este ejemplo se ha llamado 'elemento'). También se puede utilizara para recorrer *arrays*. Además de su sencillez, otra ventaja de este tipo de iteradores es que no es necesarios realizar ningún *cast* para recuperar el elemento de la colección.

```
List<String> nombres = new ArrayList<String>();
nombres.add("Antonio");
nombres.add("Juan");
nombres.add("Ricardo");
System.out.println("Estilo de iteracion JDK 1.5:");
for (String nombre: nombres) {
        System.out.println(nombre);
}
```

## Problemas específicos e implementación

**Iteradores internos vs. externos**

Un iterador puede diseñarse como iterador interno o bien como iterador externo.

*iteradores internos*

- Una colección proporciona métodos que permiten al código cliente recorrer los diferentes elementos de la colección. Por ejemplo, la clase j*ava.util.ResultSet* contiene los datos recuperados de una base de datos y también ofrece métodos, tal como *next()* para navegar a través de las filas.
- Simultáneamente sólo puede haber un iterador recorriendo una colección.
- La colección es responsable de mantener el estado de la iteración.

*iteradores externos*

- El mecanismo de iteración existe separadamente de la colección, esto es, hay un objeto llamado iterador que encapsula toda la funcionalidad necesaria para iterar la colección. La propia colección es capaz de devolver un iterador al código cliente, dependiendo del tipo de petición que éste haga. Por ejemplo, la clase *java.util.Vector* tiene su iterador definido como un objeto separado del tipo *java.util.Enumeration*. Este objeto es retornado al código cliente como respuesta a la llamada al método *elements()*.
- Puede haber múltiples iteradores simultáneamente sobre una misma colección.
- La sobrecarga asociada al almacenamiento del estado de la iteración no corresponde a la colección, sino exclusivamente al objeto Iterador.

**Ejemplo de iterador interno**

Como ejemplo para comprender este tipo de iteradores, vamos a crear una pequeña aplicación que muestre los datos de un fichero llamado "candidatos.txt", el cual contendrá los detalles de diferentes profesionales informáticos que aspiran a un puesto de trabajo en un proceso de selección abierto. Por simplicidad, sólo vamos a

considerar tres atributos: nombre, empresa actual (o última empresa en la que trabajó) y localidad.

El fichero candidatos.txt queda como sigue:

candidatos.txt

```
Enrique,SoftCut s.a,Barcelona
Luis,IT Consulting s.a,Madrid
Carlos,Digit s.l,Zaragoza
Raquel,Microsoft,Madrid
David,Oracle Iberica,Madrid
Vanesa,SoftCut s.a,Barcelona
Armand,Oracle University, Barcelona
```

Veamos ahora la clase que representa un Candidato:

Candidato.java

```
package comportamiento.iterator.candidatos.interno;


public class Candidato {
	private String nombre;
	private String empresa;
	private String localidad;


	public Candidato(String nombre, String empresa, String
localidad) {
		this.nombre = nombre;
		this.empresa = empresa;
		this.localidad = localidad;
	}


	public String getNombre() {   return nombre;    }
	public String getEmpresa() {  return empresa;   }
	public String getLocalidad() { return localidad; }
```

```
}
```

En el caso de seguir un diseño mediante iteradores internos, el contenedor o colección es responsable de proporcionar la interfaz que permita a las clases clientes navegar a través de los elementos del contenedor o colección. Por tanto, vamos a definir también una clases contenedora llamada Candidatos que:

- En su constructor lea los datos del fichero de texto anterior y los utilice para crear objetos Candidato que almacenará en un atributo de instancia de tipo Vector.
- Implementará la interfaz *java.util.Iterator* y proporcionará una implementación para sus métodos tal que así:
  - ✔ hasNext(): Verifica si hay más candidatos en la colección o ya se ha alcanzado el último.
  - ✔ next(): Devuelve el siguiente candidato en la colección, si lo hay.
  - ✔ remove(): no se implementa, pues nuestro ejemplo no aborda el tema de la eliminación de candidatos.

Candidatos.java

```
package comportamiento.iterator.candidatos.interno;

import java.io.*;
import java.util.*;

public class Candidatos implements Iterator<Candidato> {
	private Vector<Candidato> candidatos;
	private Enumeration<Candidato> enumCandidatos;
	private Candidato siguienteCandidato;

	public Candidatos() {
		inicializar();
```

```
            enumCandidatos = candidatos.elements();
      }

      private void inicializar() {
            candidatos = new Vector<Candidato>();

            // Recuperar los datos desde el fichero de texto.
            Vector<String> filas = fileToVector("candidatos.txt");

            /*
             * Crear objetos Candidato a partir del vector de String's
y añadirlos
             * al Vector datos
             */
            for (int i = 0; i < filas.size(); i++) {
                  String str = (String) filas.elementAt(i);
                  /*
                   * String[] token = str.split(","); datos.add(new
                   * Candidato(token[0],token[1],token[2]));
                   */
                  StringTokenizer st = new StringTokenizer(str, ",");
                  candidatos.add(new Candidato(st.nextToken(),
                        st.nextToken(), st.nextToken()));
            }
      }

      public boolean hasNext() {
            siguienteCandidato = null;

            while (enumCandidatos.hasMoreElements()) {
                  Candidato tempObj = enumCandidatos.nextElement();
```

```
                siguienteCandidato = tempObj;
                break;
            }
            return siguienteCandidato != null;
        }

        public Candidato next() {
            if (siguienteCandidato == null) {
                throw new NoSuchElementException();
            } else {
                return siguienteCandidato;
            }
        }

        public void remove() {  }

        /*
         * Metodo que lee datos de un fichero dado y los devuelve en un
Vector
         */
        private Vector fileToVector(String fileName) {
            Vector v = new Vector();
            String inputLine;
            try {
                File inFile = new File(fileName);
                BufferedReader br = new BufferedReader(new
InputStreamReader(
                        new FileInputStream(inFile)));

                while ((inputLine = br.readLine()) != null) {
                    v.addElement(inputLine.trim());
                }
```

```
                br.close();
            }
            catch (FileNotFoundException ex) {
                ex.printStackTrace();
            } catch (IOException ex) {
                //
            }
            return v;
      }
}
```

Tenemos por último, una clase cliente llamada MainClient que utiliza la clase contenedor Candidatos para mostrar por pantalla los diferentes candidatos. Es importante advertir que para la clase MainClient la clase Candidatos es tanto un contenedor como un iterador.

MainClient.java

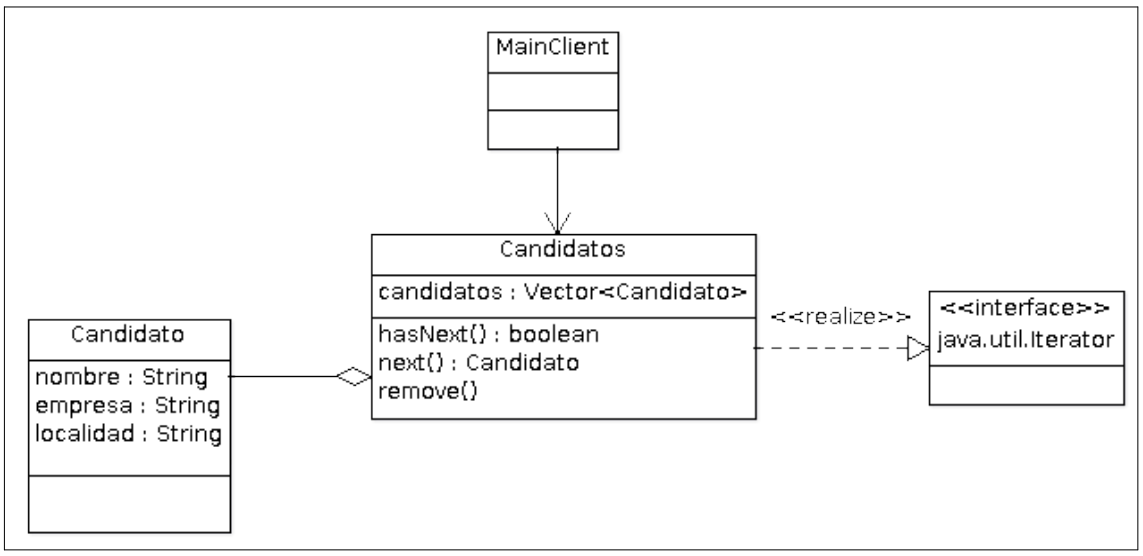
```
package comportamiento.iterator.candidatos.interno;

public class MainClient {

      public static void main(String[] args) {
            Candidatos candidatos = new Candidatos();
            String candidatosSeleccionados = "Nombre --- Empresa ---
Localidad"
                        + "\n" +
"-------------------------------------";

            while (candidatos.hasNext()) {
                Candidato c = (Candidato) candidatos.next();
                candidatosSeleccionados = candidatosSeleccionados +
"\n" + c.getNombre()
                            + " - " + c.getEmpresa() + " - "
```

```
                        + c.getLocalidad();
            }
            System.out.println(candidatosSeleccionados);
      }
}
```

Salida:

```
Nombre --- Empresa --- Localidad
-------------------------------------
Enrique - SoftCut s.a - Barcelona
Luis - IT Consulting s.a - Madrid
Carlos - Digit s.l - Zaragoza
Raquel - Microsoft - Madrid
David - Oracle Iberica - Madrid
Vanesa - SoftCut s.a - Barcelona
Armand - Oracle University -  Barcelona
```

El siguiente diagrama muestra las relaciones entre las clases anteriores:




**Ejemplo de iterador externo**

Vamos a extender el ejemplo anterior de los candidatos para permitir que el código cliente pueda filtrar candidatos por la empresa en la que actualmente trabajan o, en su defecto, en la última en la que trabajaron. Esta extensión puede conseguirse usando un iterador externo de filtrado. Del ejemplo anterior aprovechamos la clase Candidato, el resto de clases varían.

En un iterador externo, la implementación del iterador dispone de su propia clase, independientemente de la implementación del contenedor. Por tanto, vamos a crear una clase para el iterador externo llamada EmpresaCandidatos, la cual implementará la interfaz *java.util.Iterator*. Como parte de su constructor, el iterador EmpresaCandidatos acepta una instancia del tipo Candidatos y un String representando el nombre de la empresa. La misión de este iterador es filtrar los elementos de la clase contenedor Candidatos, devolviendo sólo aquellos cuya empresa corresponda con la recibida por parámetro. La implementación de *java.util.Iterator* se caracteriza por lo siguiente:

- hasNext(): Comprueba su hay más candidatos por recorrer cuya empresa coincida con la recibida por parámetro en el constructor.
- next(): Devuelve el siguiente candidato, si lo hay.
- remove(): No se implementa por ser irrelevante para el ejemplo.

El código es el siguiente:

EmpresaCandidatos.java

```
package comportamiento.iterator.candidatos.externo;

import java.util.Enumeration;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class EmpresaCandidatos implements Iterator<Candidato> {
	private String empresa;
	private Candidato siguienteCandidato;
	private Enumeration<Candidato> enumCandidatos;

	public EmpresaCandidatos(Candidatos candidatos, String empresa)
{
		this.empresa = empresa;
		this.enumCandidatos = candidatos.getCandidatos();
	}
```

```
	@Override
	public boolean hasNext() {
		boolean encontrado = false;
		while (enumCandidatos.hasMoreElements()) {
			Candidato tempObj = enumCandidatos.nextElement();
			if (tempObj.getEmpresa().equals(empresa)) {
				encontrado = true;
				siguienteCandidato = tempObj;
				break;
			}
		}
		if (!encontrado) {
			siguienteCandidato = null;
		}
		return encontrado;
	}

	@Override
	public Candidato next() {
		if (siguienteCandidato == null) {
			throw new NoSuchElementException();
		} else {
			return siguienteCandidato;
		}
	}

	@Override
	public void remove() {  }

}
```

A causa del nuevo diseño del iterador, la clase contenedora Candidatos tiene que modificarse, teniendo en cuenta lo siguiente:

- Es aun responsable de leer los datos del fichero y convertirlos en objetos del tipo Candidato.
- Debe proporcionar un método getEmpresaCandidatos(String empresa), el cual crea y retorna un iterador. El código cliente puede usar este método cuando quiera filtrar los candidatos por una determinada empresa.

Es importante advertir que cuando se crea una instancia de EmpresaCandidatos, el contenedor Candidatos se envía así mismo como un argumento para el constructor del iterador. El iterador utilizará esa instancia para acceder a los diferentes objetos Candidato.

Veamos el código:

Candidatos.java

```
package comportamiento.iterator.candidatos.externo;


import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.*;


public class Candidatos {
	private Vector<Candidato> candidatos;

	public Candidatos() {
		inicializar();
	}
```

```
	private void inicializar() {

		candidatos = new Vector<Candidato>();

		// Recuperar los datos desde el fichero de texto.
		Vector<String> filas = fileToVector("candidatos.txt");

		/*
		 * Crear objetos Candidato a partir del vector de String's
y añadirlos
		 * al Vector datos
		 */
		for (int i = 0; i < filas.size(); i++) {
			String str = (String) filas.elementAt(i);
			/*
			 * String[] token = str.split(","); datos.add(new
			 * Candidato(token[0],token[1],token[2]));
			 */
			StringTokenizer st = new StringTokenizer(str, ",");
			candidatos.add(new Candidato(st.nextToken(),
st.nextToken(), st
					.nextToken()));
		}
	}

	public Enumeration<Candidato> getCandidatos() {
		return candidatos.elements();
	}

	public Iterator<Candidato> getEmpresaCandidatos(String empresa)
{
```

```
            return new EmpresaCandidatos(this, empresa);
      }


      /*
       * Metodo que lee datos de un fichero dado y los devuelve en un
Vector
       */
      private Vector fileToVector(String fileName) {
            Vector v = new Vector();
            String inputLine;
            try {
                  File inFile = new File(fileName);
                  BufferedReader br = new BufferedReader(new
InputStreamReader(
                              new FileInputStream(inFile)));


                  while ((inputLine = br.readLine()) != null) {
                        v.addElement(inputLine.trim());
                  }
                  br.close();
            } catch (FileNotFoundException ex) {
                  ex.printStackTrace();
            } catch (IOException ex) {
                  //
            }
            return v;
      }
}
```

Por último veamos el código cliente. Notad que es necesario trabajar tanto con una referencia de Candidatos como con una de *java.util.Iterator*:

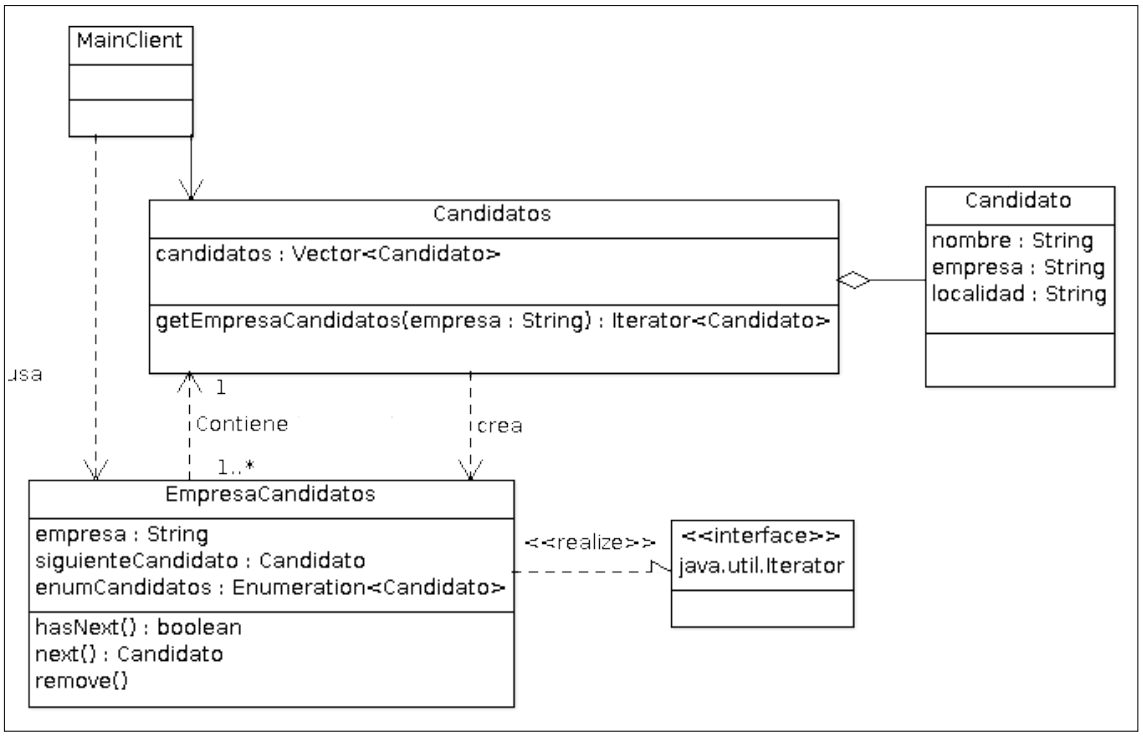
MainClient.java

```
package comportamiento.iterator.candidatos.externo;


import java.util.Iterator;


public class MainClient {


	public static void main(String[] args) {
		Candidatos candidatos = new Candidatos();
		Iterator<Candidato> empresaCandidatos =
				candidatos.getEmpresaCandidatos("SoftCut s.a");
		String candidatosSeleccionados = "Nombre --- Empresa ---
Localidad"
				+ "\n" +
"-------------------------------------";


		while (empresaCandidatos.hasNext()) {
			Candidato c = empresaCandidatos.next();
			candidatosSeleccionados = candidatosSeleccionados +
"\n" + c.getNombre()
					+ " - " + c.getEmpresa() + " - "
					+ c.getLocalidad();
		}
		System.out.println(candidatosSeleccionados);
	}


}
```

Salida:

```
Nombre --- Empresa --- Localidad
-------------------------------------
Enrique - SoftCut s.a - Barcelona
Vanesa - SoftCut s.a - Barcelona
```

El siguiente diagrama muestra las relaciones entre las clases anteriores:




## Patrones relacionados

- Composite: Los iteradores suelen aplicarse recursivamente a estructuras compuestas.
- Factory Method: Los iteradores polimórficos se construyen mediante métodos de factoría con tal de instanciar la apropiada subclase de iterador.
- Memento: Este patrón se suele utilizar conjuntamente con el patrón Iterator, ya que un iterador pueste utilizar un memento para obtener el estado de la iteración. El iterador almacena internamente al memento.