

# T4: Patrones de Diseño

## Patrón Iterator

IMPLEMENTA EL ITERADOR EXTENDIDO, Y REALIZA UN PROGRAMA SIMILAR AL DEL EJEMPLO QUE VISUALICE TODAS LAS PUJAS DE UNA SUBASTA EN ESE ORDEN.

Para implementar un iterador extendido se crea una nueva clase que implemente la interfaz *ExtendedIterator* que junto a los métodos de esta se añadirán los de *hasNext()* y *next()* del objeto *Iterator* de java. A la nueva clase se le pasa un *ArrayList* en el constructor con los datos, en este caso *Pujas*.

```
public class PujasIterator implements ExtendedIterator<Puja> {  
  
    private LinkedList<Puja> listaPujas;  
    private int pos;  
  
    public PujasIterator(LinkedList<Puja> listaPujas){  
        this.listaPujas = listaPujas;  
        pos = 0;  
    }  
  
    @Override  
    public Puja previous() {  
        if(pos < 0 || listaPujas.isEmpty()) return null;  
        return listaPujas.get(pos--);  
    }  
  
    @Override  
    public boolean hasPrevious() {  
        if(pos < 0 || listaPujas.isEmpty()) return false;  
        return true;  
    }  
  
    @Override  
    public void goFirst() {  
        pos = 0;  
    }  
  
    @Override  
    public void goLast() {  
        pos = listaPujas.size() - 1;  
    }  
  
    @Override  
    public boolean hasNext() {  
        if(pos > listaPujas.size() - 1 || listaPujas.isEmpty()) return false;  
        return true;  
    }  
  
    @Override  
    public Puja next() {  
        if(pos >= listaPujas.size() || listaPujas.isEmpty()) return null;  
        return listaPujas.get(pos++);  
    }  
}
```

En la clase Subasta se modifica el método `getSubastas` para que devuelva un nuevo objeto *PujasIterator* y se le pasa la lista de pujas.

```
public ExtendedIterator<Puja> getPujas() {
    return new PujasIterator(pujas);
}
```

Por último se realiza un programa para visualizar todas las pujas utilizando un *ExtendedIterator*.

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

    Usuario juan = new Usuario("Juan", "juanico@gmail.com", LocalDate.of(1981,12, 16), 100);
    Usuario enrique = new Usuario("Enrique", "kike@gmail.com", LocalDate.of(1982, 3, 19), 300);
    Usuario pedro = new Usuario ("Pedro", "perico@hotmail.com", LocalDate.of(1979, 6, 9), 150);

    Subasta subasta = new Subasta("Cerveza", pedro);
    subasta.pujar(enrique);
    subasta.pujar(juan);
    subasta.pujar(enrique);
    subasta.pujar(juan);
    subasta.pujar(enrique);
    subasta.pujar(juan);
    subasta.pujar(enrique);

    ExtendedIterator<Puja> it = subasta.getPujas();

    Puja p;
    it.goLast();
    System.out.println("Orden inverso:");
    while (it.hasPrevious()){
        p=it.previous();
        System.out.println("Puja actual: " + p.toString());
    }

    System.out.println("\nOrden normal:");
    it.goFirst();
    while (it.hasNext()){
        p=it.next();
        System.out.println("Puja actual: " + p.toString());
    }
}
```

Orden inverso:

```
Puja actual: t4patrones.Puja [pujador=Enrique, cantidad=7.0]
Puja actual: t4patrones.Puja [pujador=Juan, cantidad=6.0]
Puja actual: t4patrones.Puja [pujador=Enrique, cantidad=5.0]
Puja actual: t4patrones.Puja [pujador=Juan, cantidad=4.0]
Puja actual: t4patrones.Puja [pujador=Enrique, cantidad=3.0]
Puja actual: t4patrones.Puja [pujador=Juan, cantidad=2.0]
Puja actual: t4patrones.Puja [pujador=Enrique, cantidad=1.0]
```

Orden normal:

```
Puja actual: t4patrones.Puja [pujador=Enrique, cantidad=1.0]
Puja actual: t4patrones.Puja [pujador=Juan, cantidad=2.0]
Puja actual: t4patrones.Puja [pujador=Enrique, cantidad=3.0]
Puja actual: t4patrones.Puja [pujador=Juan, cantidad=4.0]
Puja actual: t4patrones.Puja [pujador=Enrique, cantidad=5.0]
Puja actual: t4patrones.Puja [pujador=Juan, cantidad=6.0]
Puja actual: t4patrones.Puja [pujador=Enrique, cantidad=7.0]
```

Process finished with exit code 0

## Patrón Adapter

REALIZAR EL DISEÑO, E IMPLEMENTAR UNA VENTANA DONDE APAREZCAN TODAS LAS PUJAS (NOMBRE DE USUARIO Y CANTIDAD) DE UNA SUBASTA EN UN JTABLE. NOTA: NO SE PUEDE MODIFICAR NINGUNA DE LAS CLASES EXISTENTES. DISEÑA E IMPLEMENTA LA SOLUCIÓN.

Para el patrón adapter la clase con la JTable debe usar un objeto de una clase que implemente la interfaz TableModel, para ello se crea una clase *PujasTableModel* que extiende de *AbstractTableModel*.

*AbstractTableModel* es una clase que ofrece java y que implementa *TableModel*, esta clase ya implementa parte de los métodos de la interfaz por lo tanto ahorra parte del trabajo.

La nueva clase recibirá por parámetro el iterador creado en el punto anterior, este iterador se convertirá en una lista para tratar los datos con mayor facilidad.

Se implementan las funciones para obtener el número de filas y columnas que no están implementadas en la clase padre y se sobrescribe la función *getValueAt()* que introduce los valores en la tabla por una que se adapte a nuestra lista.

```
public class PujasTableModel extends AbstractTableModel {
    private String[] colNames = {"Pujador", "Cantidad"};
    private List<Puja> data;

    public PujasTableModel(ExtendedIterator it){
        super();
        this.data = iteratorToList(it);
    }

    private List iteratorToList(ExtendedIterator it){
        List<Puja> lista = new ArrayList<Puja>();
        while (it.hasNext()) {
            lista.add((Puja) it.next());
        }
        return lista;
    }

    @Override
    public int getRowCount() {
        return data.size();
    }

    @Override
    public int getColumnCount() {
        return colNames.length;
    }

    public String getColumnName(int col) {
        return colNames[col];
    }

    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        Puja puja = data.get(rowIndex);
        switch (columnIndex){
            case 0:
                return puja.getPujador().getNombre();
            case 1:
                return puja.getCantidad();
            default:
                return null;
        }
    }
}
```

Por último en la clase para la interfaz se instancia un modelo del tipo *PujasTableModel* con un objeto subasta y se pasa como parámetro en el constructor del *JTable*.

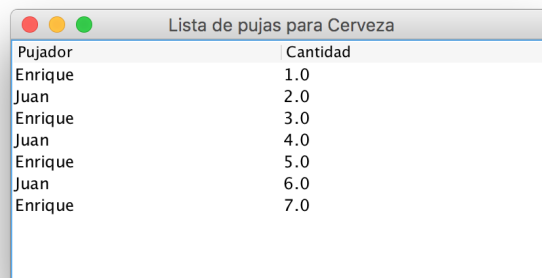
```
public class SubastaTable extends JFrame {
    public SubastaTable(Subasta subasta){
        super("Lista de pujas para " + subasta.getProducto());

        this.setSize(new Dimension(400, 400));
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        TableModel modelo = new PujasTableModel(subasta.getPujas());
        JTable tabla = new JTable(modelo);
        JScrollPane scroll = new JScrollPane(tabla);
        this.getContentPane().add(scroll);
        this.setVisible(true);
    }
}
```

Para probar se utiliza la función del primer punto y se crea un objeto *SubastaTable* con los datos de la subasta.

```
ExtendedIterator<Puja> it = subasta.getPujas();
new SubastaTable(subasta);
```



Pujador	Cantidad
Enrique	1.0
Juan	2.0
Enrique	3.0
Juan	4.0
Enrique	5.0
Juan	6.0
Enrique	7.0

## Patrón Observer

QUEREMOS EXTENDER EL SISTEMA, DE MANERA QUE, CADA VEZ QUE UN USUARIO REALICE UNA PUJA SOBRE UNA SUBASTA, SE ENVÍE UN CORREO A TODOS LOS USUARIOS QUE PREVIAMENTE HAYAN REALIZADO UNA PUJA SOBRE ESA SUBASTA. PARA SIMPLIFICAR EL EJERCICIO, EN VEZ DE ENVIAR EL CORREO SE IMPRIMIRÁ EL SIGUIENTE MENSAJE POR PANTALLA (POR CADA USUARIO QUE HAYA PUJADO PREVIAMENTE EN LA SUBASTA):

“CORREO ENVIADO A XXXX@YYY.ZZ”. EL USUARIO AAAAA HA PUJADO POR EL ARTÍCULO BBBB POR VALOR DE CCC EUROS.

Para empezar se creará una clase encargada de enviar los correos, esta clase implementa la interfaz *IObserver* que determina que está observando a la espera de cambios.

```
public interface IObserver {
    void update();
}
```

```
public class MailSender implements IObserver {
    Subasta subject;

    public MailSender(Subject subject){
        subject.attach(this);
        this.subject = (Subasta) subject;
    }

    @Override
    public void update() {
        Puja puja = subject.getPujaMayor();
        System.out.println("Correo enviado a xxxx@yyy.zz". El usuario"
            + puja.getPujador()
            + " ha pujado por el artículo "
            + subject.getProducto()
            + "por valor de "
            + puja.getCantidad()
            + " euros.");
    }
}
```



Luego se crea una clase *Subject* que tiene la responsabilidad de añadir a una lista los observers, quitarlos y notificar de un cambio.

```
public class Subject {
    private List<IObserver> observerList = new ArrayList();

    public void attach(IObserver observer) {
        observerList.add(observer);
    }

    public void detach(IObserver observer) { observerList.remove(observer); }

    public void toNotify() {
        Iterator it = observerList.iterator();
        while (it.hasNext()) {
            ((IObserver) it.next()).update();
        }
    }
}
```

*EmailSender* está pendiente de que se añadan pujas a una subasta por lo que el objeto a observar es una Subasta, un sujeto concreto, por lo tanto *Subasta* hereda de *Subject*. Cuando se añade una puja a la lista se invoca al método *toNotify()* que avisa a todos los observers de que ha ocurrido algún cambio y entonces ejecutan su función *update* que se definió en la interfaz *IObserver*.

```
public class Subasta extends Subject{
    private final String producto;
    private Usuario propietario;
    private boolean abierta;
    private LinkedList<Puja> pujas;
```

```
    public boolean pujar (Usuario pujador, double cantidad){
        if (pujador == null)
            throw new IllegalArgumentException("El usuario que hace la puja no puede ser null");

        if (cantidad < 1)
            throw new IllegalArgumentException("La cantidad tiene que ser positiva");

        if (isAbierta() &&
            pujador.getCredito() >= cantidad &&
            pujador != propietario &&
            (getPujaMayor() == null || (getPujaMayor() != null && getPujaMayor().getCredito() < cantidad))) {

            Puja puja = new Puja(pujador, cantidad);

            pujas.add(puja);
            toNotify();
            return true;
        }
        else
            return false;
    }
}
```

Después de haber creado una subasta y para finalizar se crea un objeto *EmailSender* al que se le pasa como parámetro dicha Subasta, en el constructor de *EmailSender* se puede ver como se llama a `subject.attach(this)` que está indicando a *Subject* que añada ese objeto a la lista de observadores.

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

        Usuario juan = new Usuario("Juan", "juanico@gmail.com", LocalDate.of(1981,12, 16), 100);
        Usuario enrique = new Usuario("Enrique", "kike@gmail.com", LocalDate.of(1982, 3, 19), 300);
        Usuario pedro = new Usuario ("Pedro", "perico@hotmail.com", LocalDate.of(1979, 6, 9), 150);

        Subasta subasta = new Subasta("Cerveza", pedro);
        new MailSender(subasta);
        subasta.pujar(enrique);
        subasta.pujar(juan);
    }
}
```

Después de ejecutar el programa principal, el mismo que para el patrón Iterator, se ve como por cada puja se ha imprimido un mensaje de correo enviado.

```
Run MainSubastas
/Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java ...
"Correo enviado a xxxx@yyy.zz". El usuarioEnrique ha pujado por el artículo Cervezapor valor de 1.0 euros.
"Correo enviado a xxxx@yyy.zz". El usuarioJuan ha pujado por el artículo Cervezapor valor de 2.0 euros.
"Correo enviado a xxxx@yyy.zz". El usuarioEnrique ha pujado por el artículo Cervezapor valor de 3.0 euros.
"Correo enviado a xxxx@yyy.zz". El usuarioJuan ha pujado por el artículo Cervezapor valor de 4.0 euros.
"Correo enviado a xxxx@yyy.zz". El usuarioEnrique ha pujado por el artículo Cervezapor valor de 5.0 euros.
"Correo enviado a xxxx@yyy.zz". El usuarioJuan ha pujado por el artículo Cervezapor valor de 6.0 euros.
"Correo enviado a xxxx@yyy.zz". El usuarioEnrique ha pujado por el artículo Cervezapor valor de 7.0 euros.
```