

# Proyecto Patrones de diseño

## (Mikel Calleja)

Links: <https://github.com/kroko8/labpatterns.git>

### 1. - Patrón Factory method

#### Cambios / Implementaciones:

Cree la clase que funcionará como factory, llamada BLFactory, que implementa el patrón Factory para crear una instancia de BLFacade, la lógica de negocio de la aplicación. Además `getBusinessLogicFactory()` decide si se debe crear una instancia local o remota de BLFacade, según la configuración en `config.xml`.

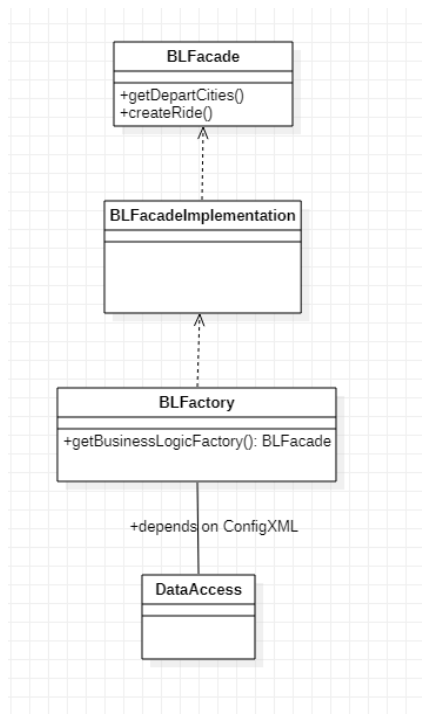
```
public class BLFactory {
    public static BLFacade getBusinessLogicFactory() {
        ConfigXML config = ConfigXML.getInstance();
        if (config.isBusinessLogicLocal()) {
            // Lógica de negocio local
            DataAccess dataAccess = new DataAccess();
            return new BLFacadeImplementation(dataAccess);
        } else {
            // Lógica de negocio remota
            try {
                String serviceName = "http://" +
config.getBusinessLogicNode() + ":" + config.getBusinessLogicPort()
                + "/ws/" + config.getBusinessLogicName() +
"?wsdl";

                URL url = new URL(serviceName);
                QName qname = new QName("http://businessLogic/",
"BLFacadeImplementationService");
                Service service = Service.create(url, qname);
                return service.getPort(BLFacade.class);
            } catch (Exception e) {
                e.printStackTrace();
                return null;
            }
        }
    }
}
```

Modifique la clase de applicationLauncher para que hiciese uso de factory:

```
BLFacade appFacadeInterface = BLFactory.getBusinessLogicFactory();
```

Y por último modifique ligeramente BLFacadeImplementation añadiendo un constructor que recibe una instancia de DataAccess como parámetro. Para que BLFactory pudiera añadir DataAccess al crear una instancia local de BLFacadeImplementation.



## 2. - Patrón Iterator

### Cambios / Implementaciones:

Añadí a la clase BLFacadeImplementation un nuevo metodo `getDepartingCitiesIterator` que crea un iterador para las ciudades de salida llamando al método `getDepartCities` de **DataAccess**, y devuelve un **CityIterator** con la lista de ciudades.

```
@Override
public ExtendedIterator<String> getDepartingCitiesIterator() {
    dbManager.open();
    List<String> cities = dbManager.getDepartCities();
    dbManager.close();
    System.out.println("Cities for iterator: " + cities);
    return new CityIterator<>(cities);}
```

Además cree una interfaz ExtendedIterator para implementar el patrón Iterator de una manera extendida, añadiendo métodos adicionales que no están en la interfaz estándar de Iterator. Que luego la use en CityIterator

```
public interface ExtendedIterator<T> extends Iterator<T> {  
  
    T previous();  
    boolean hasPrevious();  
    void goFirst();  
    void goLast();  
}
```

```
public class CityIterator<T> implements ExtendedIterator<T>{  
    private List<T> items;  
    private int currentPosition = -1;  
    public CityIterator(List<T> items) {  
        this.items = items;  
        if (!items.isEmpty()) {  
            this.currentPosition = 0;  
        }  
    }  
    @Override  
    public void goFirst() {  
        if (!items.isEmpty()) {  
            currentPosition = -1;  
        }  
    }  
    @Override  
    public void goLast() {  
        if (!items.isEmpty()) {  
            currentPosition = items.size();  
        }  
    }  
    @Override  
    public boolean hasNext() {  
        return currentPosition < items.size() - 1;  
    }  
    @Override  
    public T next() {  
        if (hasNext()) {  
            currentPosition++;  
        }  
    }  
}
```

```

        return items.get(currentPosition);
    }
    throw new NoSuchElementException();
}
@Override
public boolean hasPrevious() {
    return currentPosition > 0;
}
@Override
public T previous() {
    if (hasPrevious()) {
        currentPosition--;
        return items.get(currentPosition);
    }
    throw new NoSuchElementException();}

```

Para comprobar el correcto uso del método, cree en el paquete de Test la clase IteratorTest

```

public class IteratorTest {

    public static void main(String[] args) {

        BLFacade blFacade = BLFactory.getBusinessLogicFactory();
        ExtendedIterator<String> i = blFacade.getDepartingCitiesIterator();
        String c;

        System.out.println("_____");
        System.out.println("FROM LAST TO FIRST");
        i.goLast();
        while (i.hasPrevious()) {
            c = i.previous();
            System.out.println(c);
        }

        System.out.println();
        System.out.println("_____");
        System.out.println("FROM FIRST TO LAST");
        i.goFirst();
        while (i.hasNext()) {
            c = i.next();
            System.out.println(c);
        }
    }
}

```

Y estos son los resultados de la ejecución:

```
Console X
<terminated> IteratorTest [Java Application] C:\Users\mike\Downloads\jdk-17.0.11\bin\javaw.exe (10 nov 2024, 19:55:26) [pid: 23776]
Read from config.xml:      businessLogicLocal=true      databaseLocal=true      dataBaseInitialized=false
DataAccess opened => isDatabaseLocal: true
DataAccess created => isDatabaseLocal: true isDatabaseInitialized: false
DataAccess closed
Creating BLFacadeImplementation instance with DataAccess parameter
DataAccess opened => isDatabaseLocal: true
Cities retrieved: [Barcelona, Donostia, Irun, Madrid]
DataAccess closed
Cities for iterator: [Barcelona, Donostia, Irun, Madrid]

FROM LAST TO FIRST
Madrid
Irun
Donostia
Barcelona

FROM FIRST TO LAST
Barcelona
Donostia
Irun
Madrid
```

### 3. - Patrón Adapter

#### Cambios / Implementaciones:

Cree esta clase para que sea un adaptador que convierte los datos del objeto Driver y sus viajes en un formato que pueda entender JTable.

```
public class DriverAdapter extends AbstractTableModel {
    private static final String[] COLUMN_NAMES = { "from", "to", "date",
"places", "price" };
    private List<Ride> rides;
    public DriverAdapter(Driver driver) {
        this.rides = driver.getCreatedRides();
    }
    @Override
    public int getRowCount() {
        return rides.size();
    }
    @Override
    public int getColumnCount() {
        return COLUMN_NAMES.length;
    }
    @Override
    public String getColumnName(int column) {
        return COLUMN_NAMES[column];
    }
    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        Ride ride = rides.get(rowIndex);
        switch (columnIndex) {
            case 0:
                return ride.getFrom();
```

```

        case 1:
            return ride.getTo();
        case 2:
            return ride.getDate();
        case 3:
            return ride.getnPlaces();
        case 4:
            return ride.getPrice();
        default:
            return null;
    }
}

```

Luego cree la clase que se encargará de crear la ventana JFrame que muestra los datos:

```

public class DriverTable extends JFrame {
    private Driver driver;
    private JTable tabla;
    public DriverTable(Driver driver) {
        super(driver.getUsername() + "'s rides");
        this.setBounds(100, 100, 700, 200);
        this.driver = driver;
        DriverAdapter adapt = new DriverAdapter(driver);
        tabla = new JTable(adapt);
        tabla.setPreferredScrollableViewportSize(new
Dimension(500, 70));
        JScrollPane scrollPane = new JScrollPane(tabla);
        getContentPane().add(scrollPane,
BorderLayout.CENTER);
    }
}

```

Y por último en el paquete test cree una clase llamada TestAdapter para comprobar el correcto funcionamiento:

```

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

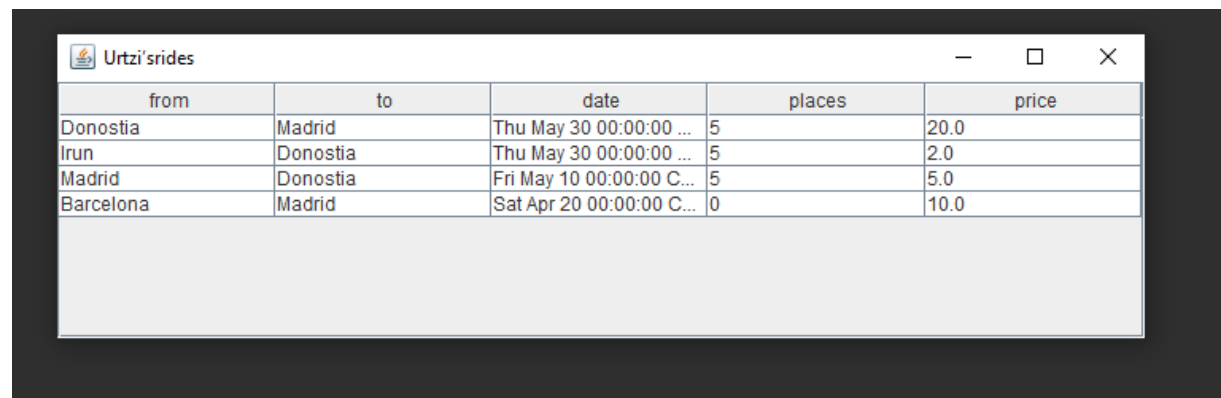
```

```

        boolean isLocal = true;
        BLFacade blFacade =
BLFactory.getBusinessLogicFactory();
        Driver driver = blFacade.getDriver("Urtzi");
        DriverTable driverTable = new DriverTable(driver);
        driverTable.setVisible(true);
    }
}

```

Resultados:



The screenshot shows a Java Swing window titled "Urtzi'srides" with a standard Mac OS X title bar (minimize, maximize, close buttons). Inside the window is a table with five columns: "from", "to", "date", "places", and "price". The table contains four rows of data. Below the table is a large, empty rectangular area, likely a placeholder for a list or further details.

from	to	date	places	price
Donostia	Madrid	Thu May 30 00:00:00 ...	5	20.0
Irun	Donostia	Thu May 30 00:00:00 ...	5	2.0
Madrid	Donostia	Fri May 10 00:00:00 C...	5	5.0
Barcelona	Madrid	Sat Apr 20 00:00:00 C...	0	10.0