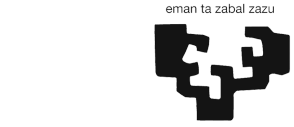
Grado en Ingeniería Informática
Facultad de Informática
UPV/EHU

PROYECTO RIDES
PATRONES DE DISEÑO



Mikel Pallin
Iker López

Índice

1. Patrón Factory Method

El proyecto Rides24 se basa en una arquitectura de 3 niveles, donde la capa de Presentación ApplicationLauncher accede a la capa de Lógica de Negocio BLFacade y esta accede a la base de datos, en este caso ObjectDB.

El problema era que el cliente contenía el código que corresponde a la creación del BusinessLogic. Esto generaba alto acoplamiento y baja flexibilidad, ya que ApplicationLauncher estaba ligado a los detalles de cómo se generaba el BusinessLogic.

1.1. Modificación de Código

Para aplicar el patrón Factory Method, se debe crear una clase que sea la encargada de crear la BusinessLogic, desacoplando así la creación de la clase ApplicationLauncher. La cual ahora pedirá la lógica de negocio a la clase Factory.

Los pasos seguidos para aplicar el patrón han sido los siguientes:

- **Paso 1:** Se ha eliminado la creación de la lógica de negocio de la clase ApplicationLauncher, para acto seguido crear la clase BLFactory, añadiendo a esta lo anteriormente suprimido.

BLFactory.java

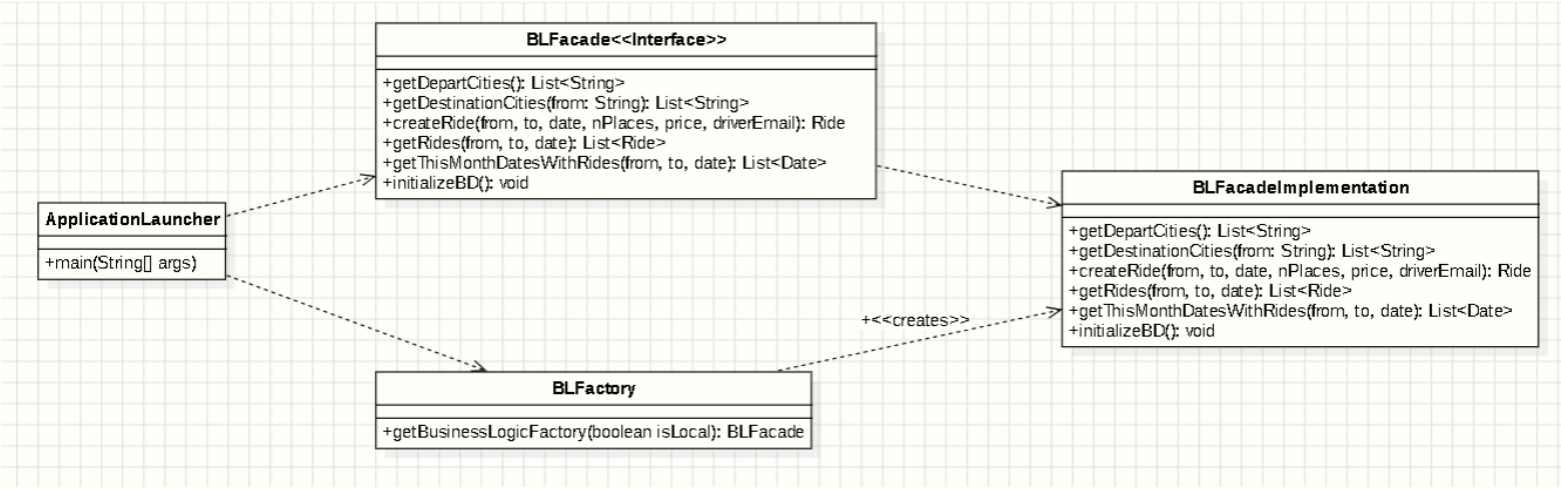
```java
public class BLFactory {

    public static BLFacade getBusinessLogicFactory(boolean isLocal) {
        try {
            ConfigXML c=ConfigXML.getInstance();
            BLFacade appFacadeInterface;

            if (isLocal) {
                DataAccess da= new DataAccess();
                appFacadeInterface = new BLFacadeImplementation(da);
            }

            else { //If remote
                 String serviceName= "http://"+c.getBusinessLogicNode() +":"+ c.getBusinessLogicPort()+"/ws/"+c.getBusinessLogicName()+"?wsdl";
                URL url = new URL(serviceName);
                QName qname = new QName("http://businessLogic/", "BLFacadeImplementationService");
                Service service = Service.create(url, qname);
                appFacadeInterface = service.getPort(BLFacade.class);
            }
            return appFacadeInterface;

        }catch (Exception e) {
            System.out.println("Error in ApplicationLauncher: "+e.toString());
            return null;
        }
```

- **Paso 2:** Una vez teniendo la Factory, se ha cambiado la clase ApplicationLauncher para que esta llame a la clase que ahora es encargada de crear la lógica de negocio (BLFactory), pasándole como parámetro un booleano para que esta sepa si la lógica a crear debe de ser de tipo local o no.

ApplicationLauncher.java

```java
public class ApplicationLauncher {

    public static void main(String[] args) {

        ConfigXML c=ConfigXML.getInstance();
        System.out.println(c.getLocale());
        Locale.setDefault(new Locale(c.getLocale()));
        System.out.println("Locale: "+Locale.getDefault());

        LoginGUI a = new LoginGUI();
        a.setVisible(true);

        try {
            UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");

            BLFacade appFacadeInterface;
            appFacadeInterface = BLFactory.getBusinessLogicFactory(c.isBusinessLogicLocal());
            LoginGUI.setBussinessLogic(appFacadeInterface);

        }catch (Exception e) {
            System.out.println("Error in ApplicationLauncher: "+e.toString());
        }
    }
```

1.2. Diagrama UML



2. Patrón Iterator

La interfaz BLFacade inicialmente ofrecía el método getDepartCities(), que devolvía lista de las ciudades que estuviesen disponibles.

2.1. Modificación de Código

En el enunciado se pide implementar el Iterator extendido y realizar un programa similar al del ejemplo. Para ello, se han llevado a cabo los siguientes pasos:

- **Paso 1:** Se ha creado la interfaz ExtendedIterator, la cual extiende de la clase Iterator de java, y se han añadido cuatro métodos de manejo del index.

ExtendedIterator.java

```java
public interface ExtendedIterator<Object> extends Iterator<Object> {
    //return the actual element and go to the previous
    public Object previous();
    //true if there is a previous element
    public boolean hasPrevious();
    //It is placed in the first element
    public void goFirst();
    // It is placed in the last element
    public void goLast();
}
```

- **Paso 2:** Se ha creado la clase Extended Iterator Cities la cual extiende de la interfaz previamente creada. Esta clase, a parte de implementar los métodos de manejo del index, también implementa operaciones de avance y retroceso de la lista.

ExtendedIteratorCities.java

```java
public class ExtendedIteratorCities implements ExtendedIterator<String>{

    private List<String> DepartingCities = new Vector<String>();
    private int index;

    public ExtendedIteratorCities(List<String> cities) {
        this.DepartingCities = cities;
        this.index = 0;
    }

    @Override
    public boolean hasNext() {
        return index < DepartingCities.size();
    }

    @Override
    public String next() {
        String city = DepartingCities.get(index);
        index++;
        return city;
    }

    @Override
    public String previous() {
        String city = DepartingCities.get(index);
        index--;
        return city;
    }

    @Override
    public boolean hasPrevious() {
        return index >= 0;
    }

    @Override
    public void goFirst() {
        index = 0;

    }

    @Override
    public void goLast() {
        index = DepartingCities.size()-1;
    }
```

- **Paso 3:** Se ha creado el método encargado de crear y devolver el iterador creado. Este método se ha añadido en la implementación de la lógica de negocio.

BLFacadeImplementation.java

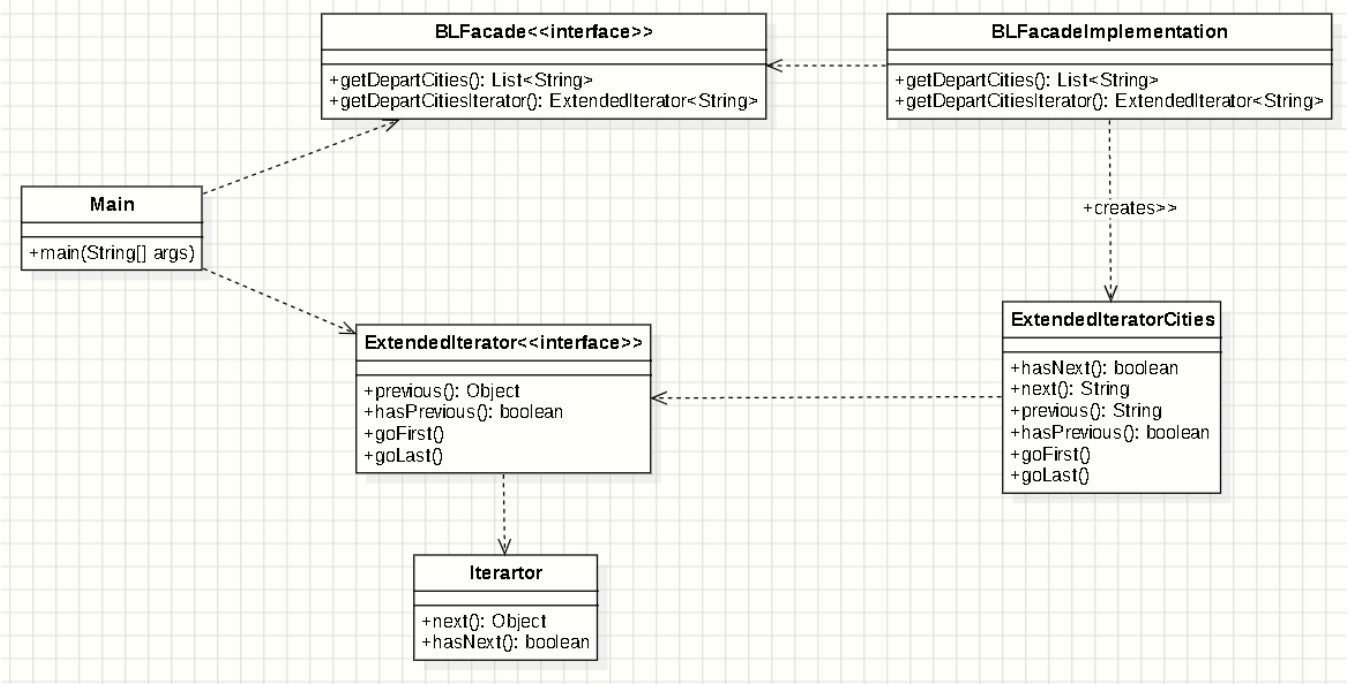
```java
@Override
public ExtendedIterator<String> getDepartingCitiesIterator() {
    List<String> cities = this.getDepartCities();
    return new ExtendedIteratorCities(cities);
}
```

- **Paso 4:** Por último, se ha creado la clase main proporcionada por el profesor, para comprobar el funcionamiento del iterador creado.

Iterator.Main.java

```java
public class main {

    public static void main(String[] args) {
        // the BL is local
        boolean isLocal = true;
        BLFacade blFacade = new BLFactory().getBusinessLogicFactory(isLocal);
        ExtendedIterator<String> i = blFacade.getDepartingCitiesIterator();

        String c;
        System.out.println("_____");
        System.out.println("FROM LAST TO FIRST");
        i.goLast(); // Go to last element

        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(); // Go to first element

        while (i.hasNext()) {
            c = i.next();
            System.out.println(c);
        }
    }
```

2.2. Diagrama UML



2.3. Ejecución de código implementado

La ejecución muestra el recorrido completo desde el último al primer elemento y viceversa, confirmando la funcionalidad del Iterador Extendido.

```
Read from config.xml:    businessLogicLocal=true databaseLocal=true      dataBaseInitialized=true
nov 14, 2025 8:53:02 PM dataAccess.DataAccess <init>
INFORMACIÓN: File deleted
DataAccess opened => isDatabaseLocal: true
Db initialized
nov 14, 2025 8:53:03 PM dataAccess.DataAccess <init>
INFORMACIÓN: DataAccess created => isDatabaseLocal: true isDatabaseInitialized: true
DataAcess closed
Creating BLFacadeImplementation instance with DataAccess parameter
DataAccess opened => isDatabaseLocal: true
DataAcess closed

_____________________
FROM LAST TO FIRST
Iruña
Gasteiz
Eibar
Donostia
Bilbo


_____________________
FROM FIRST TO LAST
Bilbo
Donostia
Eibar
Gasteiz
Iruña
```

3. Patrón Adapter

3.1. Modificación de Código

En el enunciado se pedía crear una tabla donde se muestren todos los viajes de un conductor, implementando el patrón Adapter. Para ello se han seguido los siguientes pasos.

- **Paso 1:** Se ha creado la clase DriverAdapter, la cual "traduce" el objeto Driver que se le pasa como parámetro de la constructora para que pueda ser visible en el componente JTable. Para ello, como se podrá ver a continuación, añade los datos necesarios del Driver en cada columna de la tabla.

DriverAdapter.java

```java
public class DriverAdapter extends AbstractTableModel {

    protected Driver driver;
    protected String[] columnNames = new String[] {"From", "To", "Date", "Places", "Price" };

    public DriverAdapter(Driver d) {
        this.driver = d;
    }

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

    @Override
    public String getColumnName(int i) {
        return columnNames[i];
    }

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

    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {

        List<Ride> rides = new ArrayList<>(driver.getRides());
        Ride r = rides.get(rowIndex);

        if (columnIndex == 0) return r.getFrom();
        else if (columnIndex == 1) return r.getTo();
        else if (columnIndex == 2) return r.getDate();
        else if (columnIndex == 3) return r.getnPlaces();
        else return r.getPrice();
    }
```

- **Paso 2:** Se han creado las clases DriverTable y Main proporcionadas por el profesor para poner a prueba el adaptador anteriormente creado.

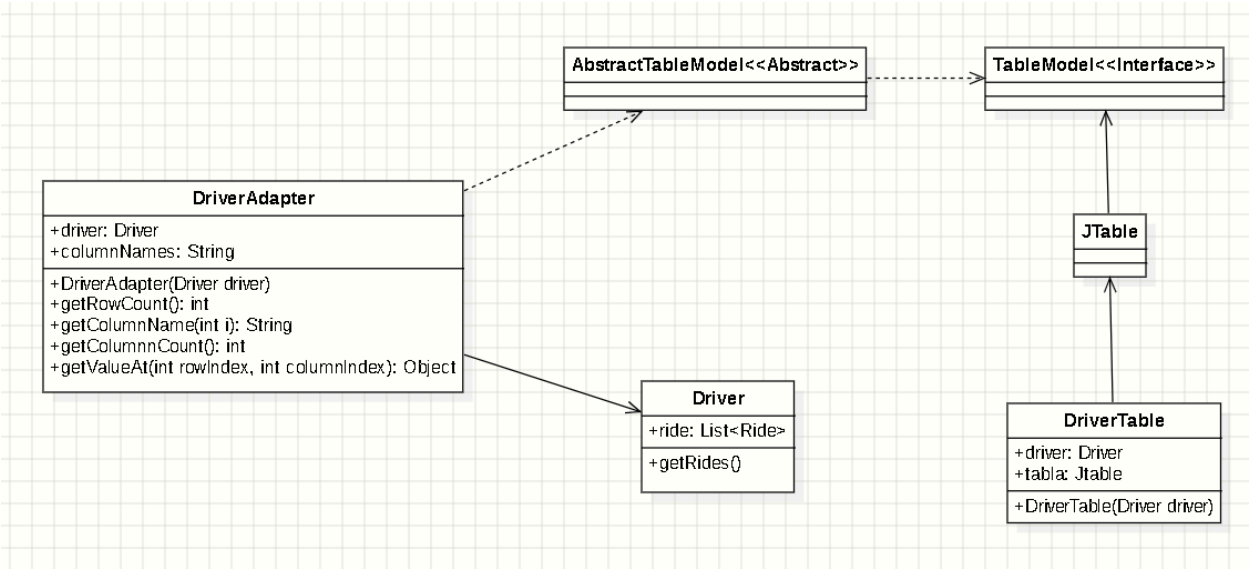
DriverTable.java

```
public class DriverTable extends JFrame{

    private Driver driver;
    private JTable tabla;

    public DriverTable(Driver driver){
        super(driver.getName()+"'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));
        //Creamos un JscrollPane y le agregamos la JTable
        JScrollPane scrollPane = new JScrollPane(tabla);
        //Agregamos el JScrollPane al contenedor
        getContentPane().add(scrollPane, BorderLayout.CENTER);
    }
```

Adapter.Main.java

```
public class Main {

    public static void main(String[] args) {
        //the BL is local
        boolean isLocal = true;
        BLFacade blFacade = new BLFactory().getBusinessLogicFactory(isLocal);
        Driver d = blFacade.getDriver("Mikel");
        DriverTable dt = new DriverTable(d);
        dt.setVisible(true);

    }
```

3.2. Diagrama UML



3.3. Ejecución de código implementado

La ejecución muestra el resultado en el componente JTable una vez pasados los Drivers por el adaptador creado.

From	To	Date	Places	Price
Gasteiz	Bilbo	Sat Nov 15 00:00:00 CET 2025	3	3.0
Iruña	Donostia	Tue Nov 25 00:00:00 CET 2025	2	5.0
Eibar	Gasteiz	Thu Nov 06 00:00:00 CET 2025	2	5.0
Donostia	Bilbo	Sat Nov 15 00:00:00 CET 2025	3	3.0
Bilbo	Donostia	Tue Nov 25 00:00:00 CET 2025	2	5.0
Eibar	Gasteiz	Thu Nov 06 00:00:00 CET 2025	2	5.0