PROYECTO RIDES
PATRONES DE DISEÑO

1. Patrón Factory Method

Situación actual

Como el enunciado indica, *ApplicationLaunches*.java decide directamente si crear una instancia local (*BLFacadeImplementation*) o remota vía web services o demás. Este mecanismo incumple con él principio SRP o principio de única responsabilidad.

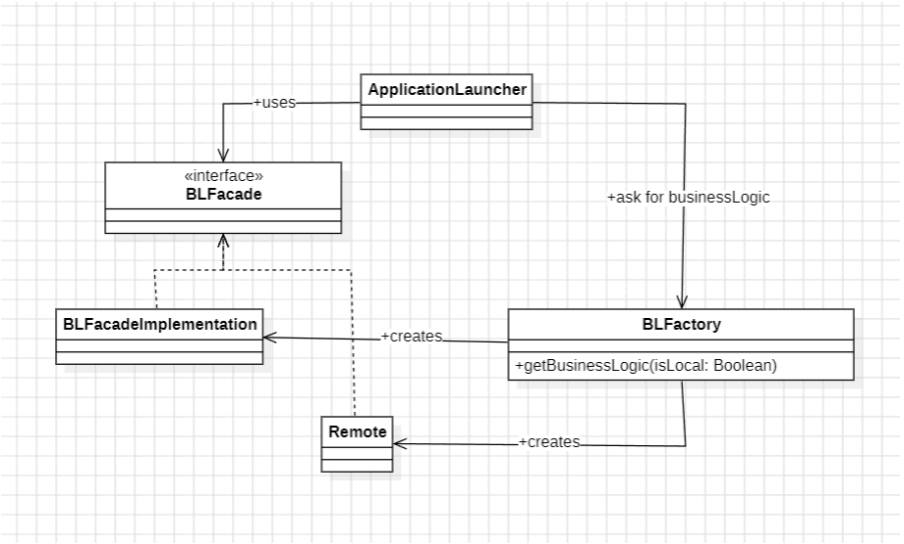
Objetivo

Centralizar la creación del objeto *BLFacade* en una clase factoría llamada *BLFactory*, para que la clase de presentación *ApplicationLauncher* decida si la implementación es local o no con la variable booleana isLocal.

Diseño

- **Roles**
  - **Client** *ApplicationLauncher* es la que invocará a la *BLFactory*
  - **Factory** *BLFactory* será la nueva clase a crear con la que se podrá obtener la lógica de negocio mediante su método.
  - **Product** *BLFacade*
    - Local (*BLFacadeImplementation* (creado con *DataAccess*)
    - Remoto
- **Variante** El método factoría decide basado en isLocal

UML



Cambios a realizar

1. **Crear la clase BLFactory** Esta clase encapsula toda la lógica de creación (mueve el código del if-else de *ApplicationLauncher* aquí).

```
public class BLFactory {
    public BLFacade getBusinessLogic(boolean isLocal) {
        ConfigXML c = ConfigXML.getInstance();
        try {
            if (isLocal) {
                // Implementación local: Usa DataAccess con modo initialize
                boolean initialize =
c.getDataBaseOpenMode().equals("initialize");
                DataAccess da = new DataAccess(initialize);
                return new BLFacadeImplementation(da);
            } else {
                // Implementación remota
                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);
                return service.getPort(BLFacade.class);
            }
        } catch (Exception e) {
            System.out.println("Error in BLFactory: " + e.toString());
            return null;
        }
    }
}
```

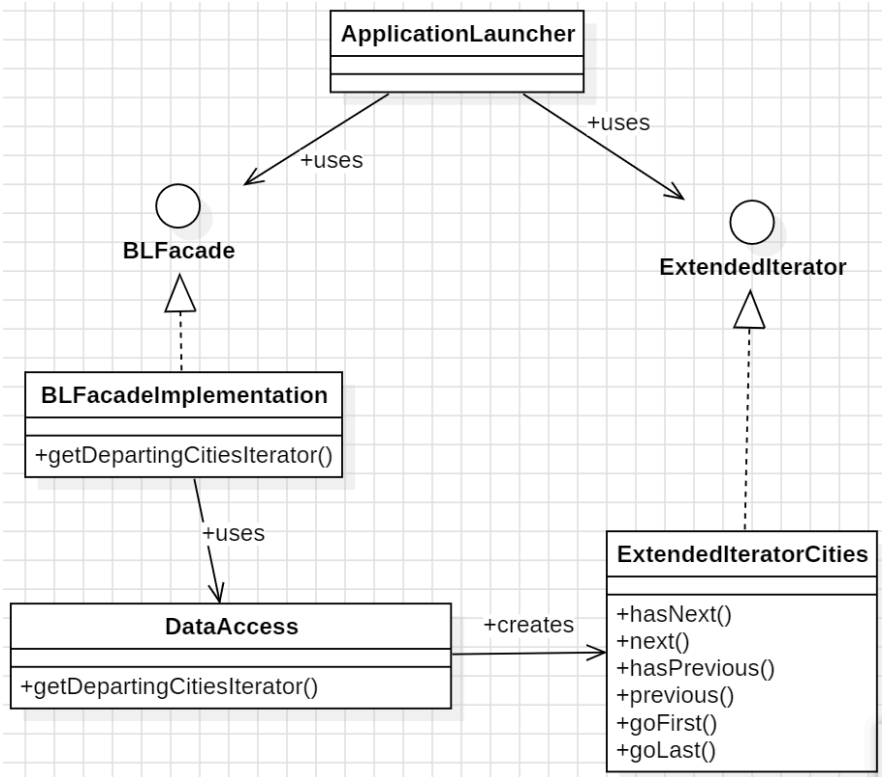
getBusinessLogic(boolean isLocal): Es el factory method. Decide el concrete product.

2. **Eliminar en ApplicationLauncher** el if-else para creación de BLFacade y reemplazarlo por una llamada a la factoría.

```
...

try {

     BLFacade appFacadeInterface;

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

    boolean isLocal = c.isBusinessLogicLocal();

     appFacadeInterface = new BLFactory().getBusinessLogic(isLocal);

     MainGUI.setBussinessLogic(appFacadeInterface);

}catch (Exception e) { . . .
```

2. Patrón Iterator

- Diagrama UML indicando los cambios realizados y la finalidad de cada clase/interfaz.



Se añade el nuevo método en la interfaz BLFacade y sigue el mismo camino que los métodos ya añadidos, finalmente se implementa el código en la clase DataAcces, que se encarga de crear el iterador.

`ExtendedIterator`: define qué debe hacer un iterador extendido, es como una versión mejorada del iterador normal ya que añade nuevas funciones: ir hacia atrás o saltar al inicio o final.
`ExtendedIteratorCities`: Proporciona la implementación de la interfaz `ExtendedIterator` de una lista de ciudades en String.

Así el programa no necesita saber cómo se implementa el iterador (porque trabaja con la interfaz). Solo necesita saber que puede moverse hacia adelante, hacia atrás, o saltar de posición. El programa no depende de una implementación concreta y facilita la extensión.

- Código modificado.
  Primero se ha creado una nueva carpeta llamada iterator en la que estará la interfaz:

```java
package iterator;

import java.util.Iterator;

public  interface   ExtendedIterator<String>    extends Iterator<Object>    {
    //return    the actual  element and go  to  the previous
    public  Object  previous();
    //true  if  ther    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();
}
```

y la clase:

```java
public class ExtendedIteratorCities implements ExtendedIterator<String>{
    List<String> cities;
    int posicion;

    public ExtendedIteratorCities(List<String> c) {
        this.cities = c;
        this.posicion = 0;
    }
```

En esta clase se tendrán 2 atributos: la lista de las ciudades y la posición que se usará tanto para avanzar como para retroceder. La posición, a diferencia de `Covid19PacientIterator`, se iniciará en la constructora.

La implementación de cada método:

```java
@Override
public boolean hasNext() {
    return posicion < cities.size();
}

@Override
public Object next() {
    String city = cities.get(posicion);
    posicion++;
    return city;
}
```

```java
@Override
public Object previous() {
    posicion--;
    String city = cities.get(posicion);
    return city;
}

@Override
public boolean hasPrevious() {
    return posicion > 0;
}
```

En el método '`previous`' primero se resta de la posición y luego se devuelve la ciudad, a diferencia de la ciudad siguiente, que primero devuelve la ciudad y luego suma la posición.

```
@Override
public void goFirst() {
    posicion = 0;
}

@Override
public void goLast() {
    posicion = cities.size();

}
```

Esto es porque cuando la posición va hacia atrás el puntero se coloca una posición después del último, así cuando haga 'previous' la posición será la correcta, la última.

En la interfaz BLFacade se añade el nuevo método public ExtendedIterator<String>getDepartingCitiesIterator();

```
@WebService
public interface BLFacade extends Serializable{

    /**
     * This method returns all the cities where rides depart
     * @return collection of cities
     */
    @WebMethod public List<String> getDepartCities();

    @WebMethod public ExtendedIterator<String>getDepartingCitiesIterator();
```

Luego en BLFacadeImplementation:

```
@WebMethod
public ExtendedIterator<String> getDepartingCitiesIterator() {
    dbManager.open();
    ExtendedIterator<String> cities =dbManager.getDepartingCitiesIterator();
    dbManager.close();
    return cities;
}
```

y por último se ha implementado en DataAccess:

```
public ExtendedIterator<String> getDepartingCitiesIterator() {
    List<String> cities = getDepartCities();
    return new ExtendedIteratorCities(cities);
}
```

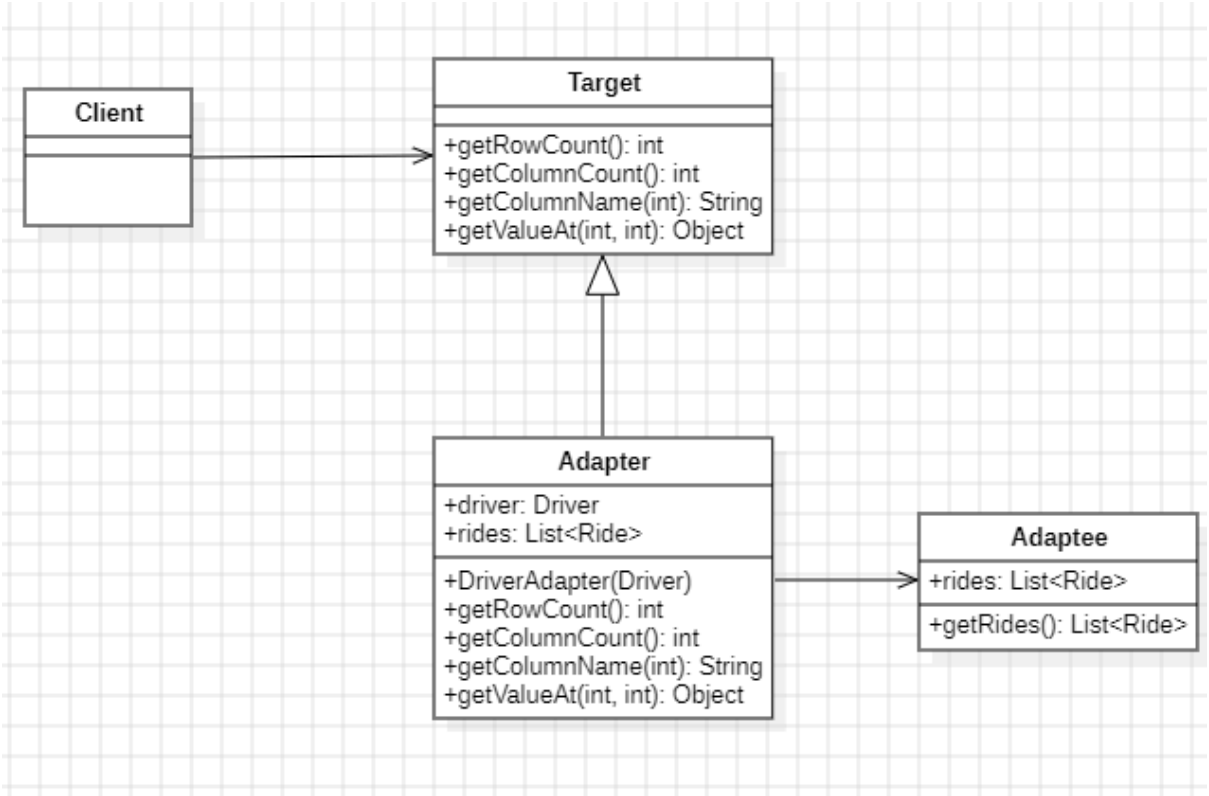
Para que no haya repetición de código llamamos al método getDepartCities() para que nos devuelva la lista de las ciudades y así crear el iterador para recorrerlo.

- Imagen que muestra su ejecución

```
Read from config.xml:    businessLogicLocal=true         databaseLocal=true      dataBaseInitialized=true
en
Locale: en
File deleted
DataAccess opened => isDatabaseLocal: true
Db initialized
DataAccess created => isDatabaseLocal: true isDatabaseInitialized: true
DataAcess closed
Creating BLFacadeImplementation instance with DataAccess parameter
DataAccess opened => isDatabaseLocal: true
DataAcess closed
_____________________
FROM    LAST    TO      FIRST
Valencia
Madrid
Granada
Donostia
Barcelona

_____________________
FROM    FIRST   TO      LAST
Barcelona
Donostia
Granada
Madrid
Valencia
```

3. Patrón Adapter

- Diagrama UML indicando los cambios realizados y la finalidad de cada clase/interfaz.



Client: el componente para mostrar los datos en la tabla
Target: la interfaz que JTable debería recibir

Adapter (DriverAdapter): adapta la clase Driver para poder ser mostrado en el JTable
Adaptee (Driver): se utiliza para poder mostrar los datos de la clase en la tabla

- Código modificado.

  Lo primero se ha creado un nuevo paquete llamado adapter donde contiene 3 clases, DriverAdapter, DriverTable y la clase Main para ejecutar el programa

  Tanto la clase DriverTable como la clase Main está implementada por el propio ejercicio, haciendo un pequeño cambio en la clase DriverTable para que muestre el email en vez del nombre, ya que Driver guarda un objeto de tipo User en vez de su propia información, siendo el email la unica excepcion, y al intentar usar el objeto devuelve null.

  Main.java:

```java
package adapter;

import businessLogic.BLFacade;

public class Main {

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

  DriverTable.java

```java
package adapter;

import java.awt.BorderLayout;

public class DriverTable extends JFrame{
    private Driver driver;
    private JTable  tabla;
    public DriverTable(Driver driver){
        super(driver.getEmail()+"'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);
    }
}
```

Para utilizar DriverTable creamos la clase DriverAdapter, que será la que adapte Driver. Extendemos la clase de AbstractTableModel e implementamos y sobreescribimos los métodos.

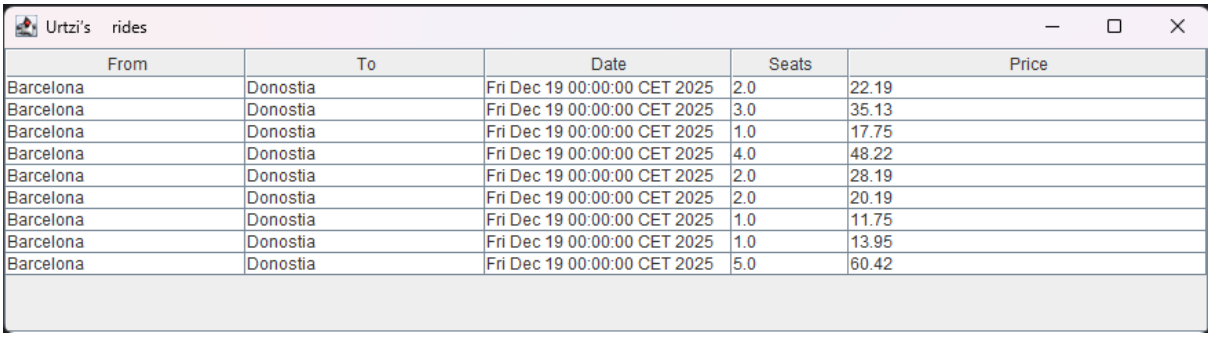
DriverAdapter.java:

```java
public class DriverAdapter extends AbstractTableModel {

    private Driver driver;
    private List<Ride> rides;

    private final String[] columnNames = {
        "From",
        "To",
        "Date",
        "Seats",
        "Price"
    };

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

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

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

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

    @Override
    public Object getValueAt(int i, int j) {
        Ride r = rides.get(i);

        switch (j) {
            case 0:
                return r.getFrom();
            case 1:
                return r.getTo();
            case 2:
                return r.getDate();
            case 3:
                return r.getnSeats();
            case 4:
                return r.getPrice();
            default:
                return null;
        }
    }
```

Por último, para que ver su funcionamiento en un Driver llamado “Urtzi”, añadimos un nuevo Driver en DataAccess donde se añadirán todos sus viajes

```java
private void createUsersAndCustomRides() {
    User userT = createUser("emailT", "T");
    User userD = createUser("emailD", "D");
    User urtzi = createUser("Urtzi", "D");

    float[] prices = {22.19f, 35.13f, 17.75f, 48.22f, 28.19f, 20.19f, 11.75f, 13.95f, 60.42f};
    int[] seats = {2, 3, 1, 4, 2, 2, 1, 1, 5};
    String[][] stops = {
        {"Pamplona"}, {"Pamplona", "Zaragoza"}, {"Pamplona", "Tarragona"}, {},
        {"Pamplona", "Zaragona", "Tarragona"}, {}, {}, {"Tarragona"}, {"Zaragoza"}
    };

    for (int i = 0; i < prices.length; i++) {
        addRide(userD, stops[i], seats[i], prices[i]);
        addRide(urtzi, stops[i], seats[i], prices[i]);
    }

    db.persist(userT);
    db.persist(userD);
    db.persist(urtzi);
}
```

- Imagen que muestra su ejecución



From	To	Date	Seats	Price
Barcelona	Donostia	Fri Dec 19 00:00:00 CET 2025	2.0	22.19
Barcelona	Donostia	Fri Dec 19 00:00:00 CET 2025	3.0	35.13
Barcelona	Donostia	Fri Dec 19 00:00:00 CET 2025	1.0	17.75
Barcelona	Donostia	Fri Dec 19 00:00:00 CET 2025	4.0	48.22
Barcelona	Donostia	Fri Dec 19 00:00:00 CET 2025	2.0	28.19
Barcelona	Donostia	Fri Dec 19 00:00:00 CET 2025	2.0	20.19
Barcelona	Donostia	Fri Dec 19 00:00:00 CET 2025	1.0	11.75
Barcelona	Donostia	Fri Dec 19 00:00:00 CET 2025	1.0	13.95
Barcelona	Donostia	Fri Dec 19 00:00:00 CET 2025	5.0	60.42