

# Proyecto de patrones de diseño

Iñigo Alzugaray  
Aitor Gil  
Pablo Jurado

10/11/2024

# Proyecto patrones de diseño

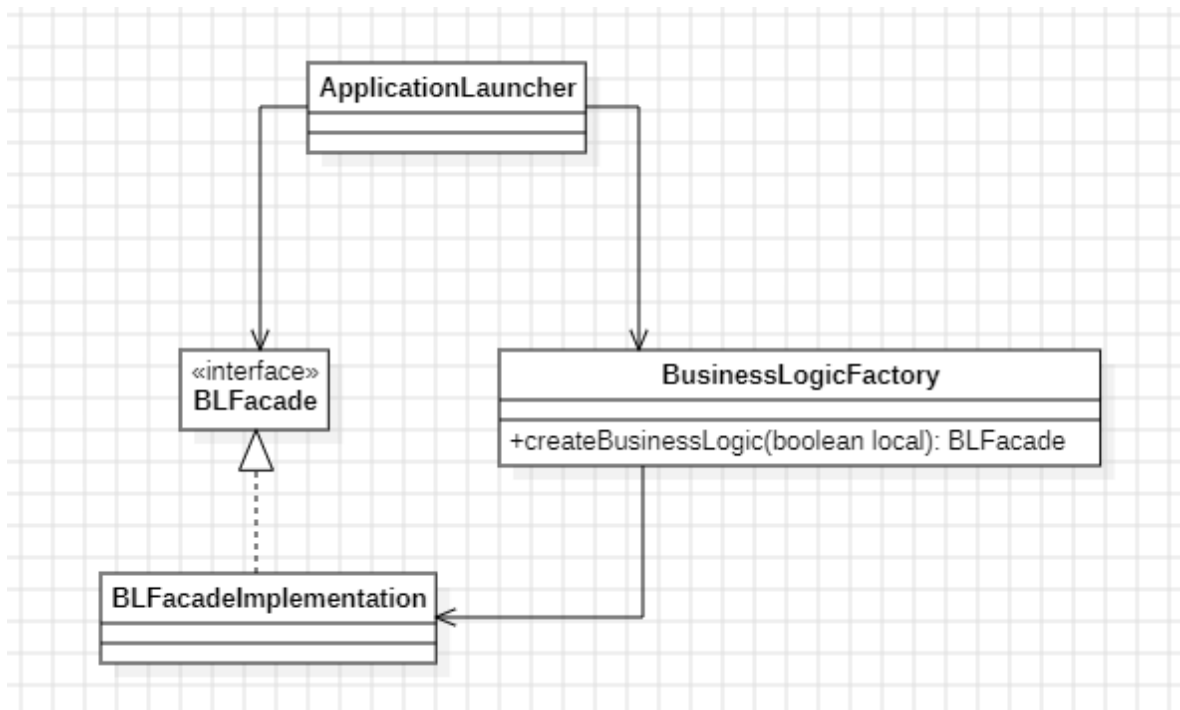
En este proyecto hemos hecho una serie de cambios en nuestro proyecto Rides utilizando los patrones de diseño factory, iterator y adapter.

Para probar el funcionamiento de los cambios hemos añadido código en el main() de nuestra aplicación, que habría que quitar más adelante pero sirve para los propósitos de este proyecto.

## Patrón factory

Para que la obtención del objeto de la lógica de negocio se haga siguiendo el patrón factory, hemos creado la clase BusinessLogicFactory. El método createBusinessLogic devuelve un objeto de lógica de negocio local o en la web dependiendo de lo que se le pida en la presentación.

La clase BusinessLogicFactory cumple el papel de Creator, BLFacade es el Product, y en este caso solo hay un ConcreteProduct (BLFacadeImplementation) aunque se crea de formas distintas.



uml-factory.png

```

public class BusinessLogicFactory {
    public static BLFacade createBusinessLogic(boolean local) throws
    MalformedURLException {
        if (local) return new BLFacadeImplementation(new DataAccess());
        else {
            ...
            return service.getPort(BLFacade.class);
        }
    }
}

```

En la presentación utilizamos la factory para obtener el objeto de la lógica de negocio.

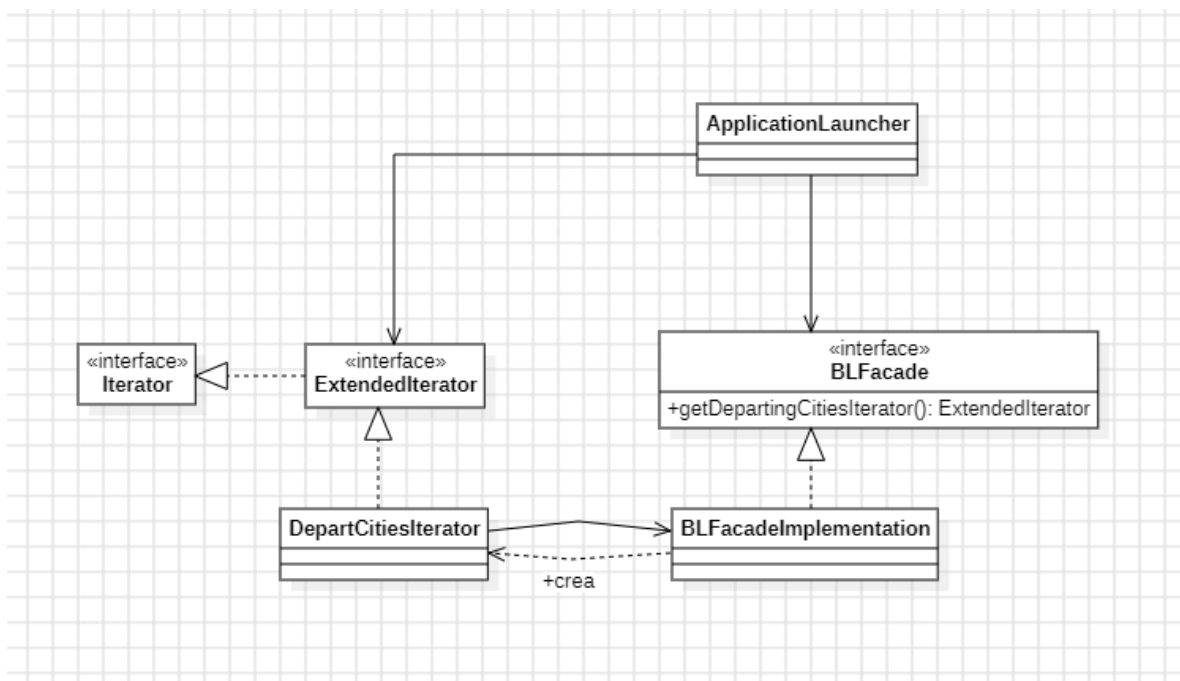
```

appFacadeInterface =
BusinessLogicFactory.createBusinessLogic(c.isBusinessLogicLocal());

```

## Patrón iterator

Para hacer uso del patrón iterator con la lista de ciudades, hemos creado la clase DepartingCitiesIterator que implementa la interfaz ExtendedIterator.



uml-iterator.png

```

public class DepartingCitiesIterator implements ExtendedIterator<String>
{
    public DepartingCitiesIterator(List<String> cities) {
        this.cities = cities;
        index = cities.size()-1;
    }

    private List<String> cities;
    private int index;

    @Override
    public String previous() {
        return cities.get(index--);
    }
    @Override
    public boolean hasPrevious() {
        return index >= 0;
    }
    @Override
    public void goFirst() {
        index = 0;
    }
    @Override
    public void goLast() {
        index = cities.size()-1;
    }
    @Override
    public boolean hasNext() {
        return index < cities.size();
    }
    @Override
    public String next() {
        return cities.get(index++);
    }
}

```

En la lógica de negocio creamos el método que devuelve el iterador, reutilizando el código de `getDepartCities()`, que devuelve el Set de ciudades.

```
public ExtendedIterator<String> getDepartingCitiesIterator() {  
    return new DepartingCitiesIterator(getDepartCities());  
}
```

Así podemos probar el iterador en el programa principal y obtener la siguiente salida:

```
ExtendedIterator<String> i =  
appFacadeInterface.getDepartingCitiesIterator();  
String city;  
System.out.println("_____");  
System.out.println("FROM    LAST    TO    FIRST");  
i.goLast(); // Go to last element  
while (i.hasPrevious()) {  
    city = i.previous();  
    System.out.println(city);  
}  
System.out.println();  
System.out.println("_____");  
System.out.println("FROM    FIRST    TO    LAST");  
i.goFirst(); // Go to first element  
while (i.hasNext()) {  
    city = i.next();  
    System.out.println(city);  
}
```

```

-----
FROM    LAST    TO    FIRST
Madrid
Irun
Donostia
Barcelona

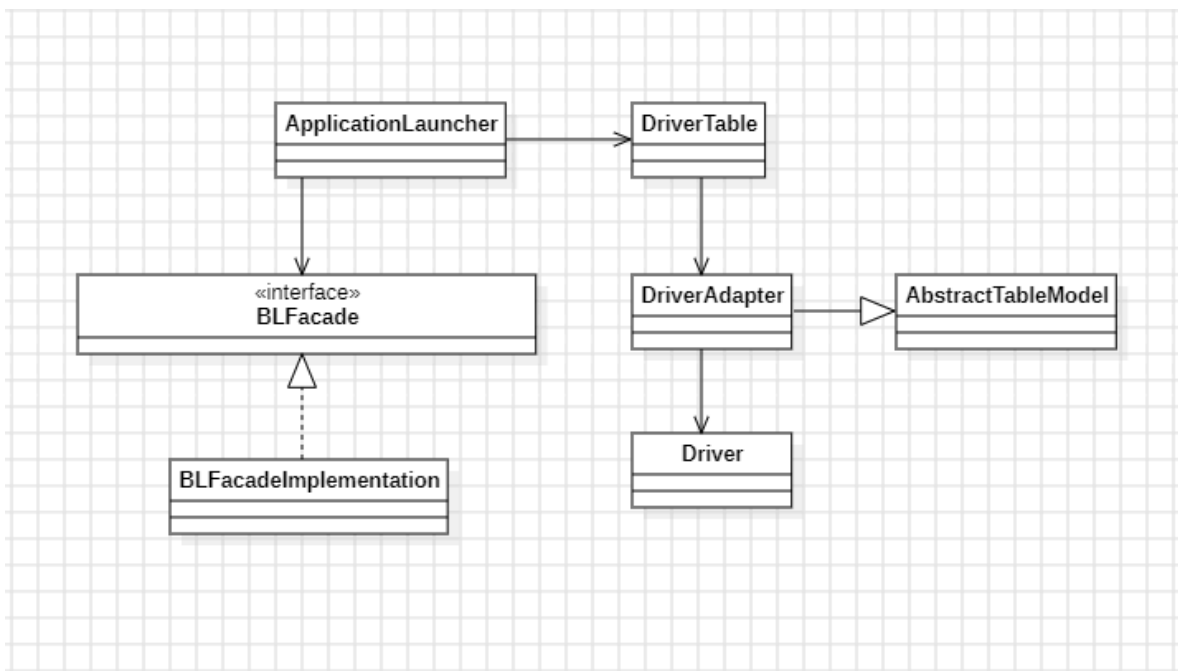
-----
FROM    FIRST    TO    LAST
Barcelona
Donostia
Irun
Madrid

```

ejecucion-iterator.png

## Patrón adapter

Queremos mostrar los datos de la clase Driver en una tabla, y para ello hemos utilizado el patrón adapter. La clase DriverAdapter implementa los métodos de AbstractTableModel con la información de Driver, de modo que adaptamos la clase Driver para poder ser utilizada en tablas. Es decir, DriverAdapter hace de adaptador, Driver es la clase adaptada y AbstractTableModel es la clase a la que queremos adaptar Driver.



uml-adapter.png

```

public class DriverAdapter extends AbstractTableModel {
    public DriverAdapter(Driver driver) {
        this.driver = driver;
    }
    private Driver driver;
    private String[] columnNames = {"from", "to", "date", "places",
"price"};
    @Override
    public int getRowCount() {
        return driver.getCreatedRides().size();
    }
    @Override
    public int getColumnCount() {
        return columnNames.length;
    }
    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        switch (columnIndex) {
            case 0:
                return driver.getCreatedRides().get(rowIndex).getFrom();
            case 1:
                return driver.getCreatedRides().get(rowIndex).getTo();
            case 2:
                return driver.getCreatedRides().get(rowIndex).getDate();
            case 3:
                return
driver.getCreatedRides().get(rowIndex).getnPlaces();
            case 4:
                return
driver.getCreatedRides().get(rowIndex).getPrice();
            default:
                return null;
        }
    }
}

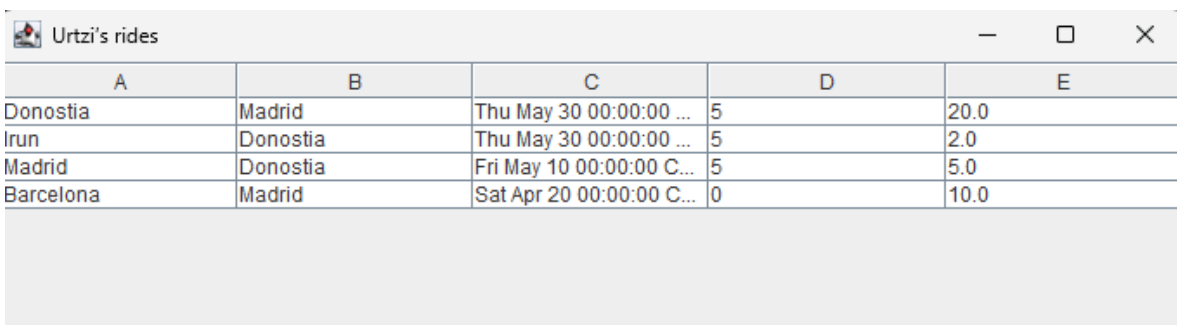
```

En la presentación tenemos la clase DriverTable que muestra una tabla con la información de un Driver utilizando la clase que hemos creado. Si probamos a ejecutar, el resultado que obtenemos es el siguiente:

```
public class DriverTable extends JFrame {
    private Driver driver;
    private JTable tabla;

    public DriverTable(Driver driver) {
        ...
        DriverAdapter adapt = new DriverAdapter(driver);
        tabla = new JTable(adapt);
        ...
    }
}
```

```
DriverTable dt = new DriverTable(appFacadeInterface.getDriver("Urtzi"));
dt.setVisible(true);
```



A	B	C	D	E
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

ejecucion-adapter.png