

# Aplicar patrones de diseño al proyecto

Koldo Intxausti y Matt Ruiz

Enlace a repositorio github: <https://github.com/Nesket/Practica-IS2>

1. Patrón Factory Method .....	2
2. Patrón Iterator .....	4
3- Patrón Adapter .....	7

## 1. Patrón Factory Method

**Modifica la aplicación para que la obtención del objeto de la lógica de negocio se centralice en un objeto factoría, y sean las clases de la presentación las que decidan cuál de las implementaciones de la lógica de negocio utilizar. Diseña e implementa la solución indicando qué clases juegan el papel de Creator, Product y ConcreteProduct.**

Primero de todo he creado una clas llamada BLFactory. A esta función se le pasará por parámetro un booleano indicando si se quiere utilizar la configuración local (true) o remota (false).

Este Factory creará el BusinessLogic correspondiente a la configuración solicitada:

```
public class BLFactory {
    public BLFacade getBusinessLogicFactory(boolean isLocal) {
        BusinessLogicFactoryInterface blf;
        if(isLocal) blf = new BusinessLogicLocalFactory();
        else blf = new BusinessLogicRemoteFactory();

        return blf.getBusinessLogicFactory();
    }
}
```

Para optimizar el código he creado una interfaz para todos los Factories de BusinessLogic (en este caso tenemos BusinessLogicLocalFactory y BusinessLogicRemoteFactory). De esta manera aseguraremos que ambas clases implementan el método “getBusinessLogicFactory()” que utilizaremos para devolver el BLFacade correspondiente.

```
public interface BusinessLogicFactoryInterface {
    public BLFacade getBusinessLogicFactory();
}
```

```
public class BusinessLogicLocalFactory implements BusinessLogicFactoryInterface {
    @Override
    public BLFacade getBusinessLogicFactory() {
        System.out.println(">>>");
        DataAccess da = new DataAccess();
        return new BLFacadeImplementation(da);
    }
}
```

```

public class BusinessLogicRemoteFactory implements BusinessLogicFactoryInterface {
    public BLFacade getBusinessLogicFactory() {
        try {
            ConfigXML c = ConfigXML.getInstance();

            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 ApplicationLauncher: " + e.toString());
        }

        return null;
    }
}

```

Finalmente, como se pedía que fuese la “clase presentación” la que decidiese qué lógica de negocio utilizar, hemos adaptado la función principal de “ApplicationLauncher” para que decida qué lógica utilizar y llame al Factory correspondiente.

```

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());

        try {
            UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
        } catch (ClassNotFoundException | InstantiationException | IllegalAccessException
            | UnsupportedLookAndFeelException e) {
            e.printStackTrace();
        }

        boolean isLocal = c.isBusinessLogicLocal();
        BLFacade appFacadeInterface = new BLFactory().getBusinessLogicFactory(isLocal);

        MainGUI.setBusinessLogic(appFacadeInterface);
        MainGUI a = new MainGUI();
        a.setVisible(true);
    }
}

```

## 2. Patrón Iterator

En primer lugar hemos añadido la interfaz ExtendedIterator proporcionada:

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

Después, hemos creado nuestro propio Iterator para los departing cities, adaptando la lista de Strings para que funcione con los métodos esperados del ExtendedIterator:

```
public class DepartingCitiesIterator implements ExtendedIterator<Object> {  
    private List<String> departingCities;  
    int position;  
  
    public DepartingCitiesIterator(List<String> departingCities) {  
        this.departingCities = departingCities;  
        this.position = 0;  
    }  
  
    public boolean hasNext() {  
        return this.position < this.departingCities.size();  
    }  
  
    @Override  
    public String next() {  
        String dCity = this.departingCities.get(this.position);  
        this.position++;  
        return dCity;  
    }  
  
    @Override  
    public String previous() {  
        this.position--;  
        String dCity = this.departingCities.get(this.position);  
        return dCity;  
    }  
  
    @Override  
    public boolean hasPrevious() {  
        return position > 0;  
    }  
  
    @Override  
    public void goFirst() {  
        this.position = 0;  
    }  
  
    @Override  
    public void goLast() {  
        this.position = departingCities.size();  
    }  
}
```

En la interfaz BLFacade he añadido la declaración del nuevo método que se pide implementar:

```
public ExtendedIterator<Object> getDepartingCitiesIterator();
```

Después, he implementado la función para que devuelva el nuevo iterator que hemos creado:

```
public ExtendedIterator<Object> getDepartingCitiesIterator() {  
    List<String> departingCities = this.getDepartCities();  
    return new DepartingCitiesIterator(departingCities);  
}
```

Finalmente, he creado una clase BLMain para comprobar que se ejecuta correctamente.

```
public class BLMain {  
    public static void main(String[] args) {  
        // the BL is local  
        boolean isLocal = true;  
        BLFactory blf = new BLFactory();  
        BLFacade blFacade = blf.getBusinessLogicFactory(isLocal);  
  
        ExtendedIterator<Object> i = blFacade.getDepartingCitiesIterator();  
  
        String c;  
        System.out.println("FROM LAST TO FIRST");  
        i.goLast(); // Go to last element  
        while (i.hasPrevious()) {  
            c = (String) i.previous();  
            System.out.println(c);  
        }  
        System.out.println();  
        System.out.println("FROM FIRST TO LAST");  
        i.goFirst(); // Go to first element  
        while (i.hasNext()) {  
            c = (String) i.next();  
            System.out.println(c);  
        }  
    }  
}
```

```
Read from config.xml:      businessLogicLocal=true      databaseLocal=true
DataAccess opened => isDatabaseLocal: true
DataAccess created => isDatabaseLocal: true isDatabaseInitialized: false
DataAccess closed
nov 08, 2024 4:56:17 P. M. businessLogic.BLFacadeImplementation <init>
INFORMACIÓN: Creating BLFacadeImplementation instance with DataAccess parameter
DataAccess opened => isDatabaseLocal: true
DataAccess closed
```

---

FROM LAST TO FIRST

Madrid  
Irun  
Donostia  
Barcelona

---

FROM FIRST TO LAST

Barcelona  
Donostia  
Irun  
Madrid



### 3- Patrón Adapter

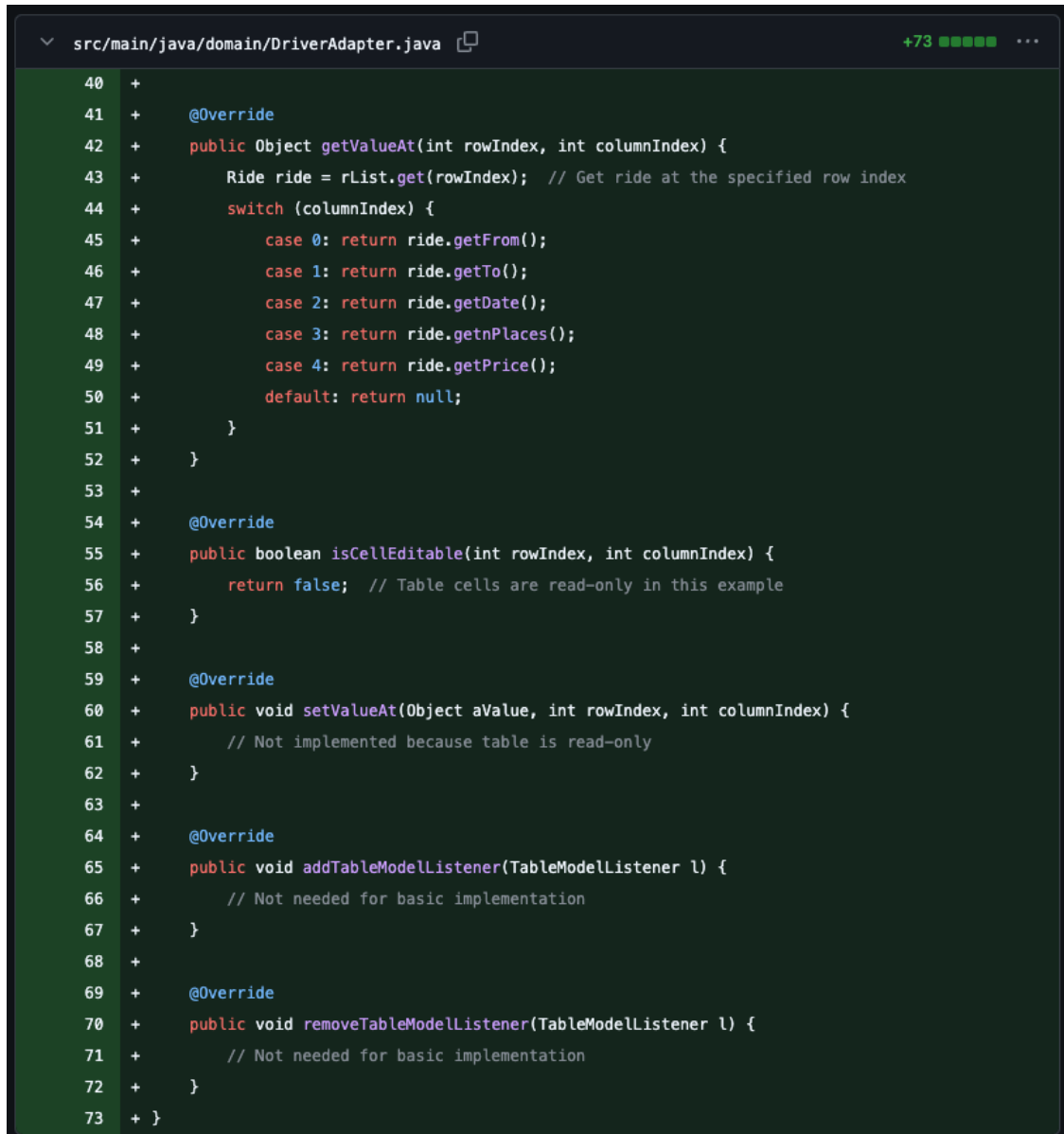
Para explicar el desarrollo del patrón adapter se muestra a continuación una serie de imágenes con los cambios realizados. Estas imágenes están acompañadas de una breve explicación. En esta primera imagen, se muestra la clase DriverAdapter realizada para poder mostrar la información de los objetos Ride de un objeto Driver concreto en una tabla. Lo más relevante de esta clase es la implementación de la interfaz TableModel.

En la clase DriverAdapter se ha escogido como atributos un objeto Driver, una lista con los objetos Ride, un array con los nombres de las columnas para la tabla y un array con los tipos de clases para facilitar la implementación de la interfaz.

```
src/main/java/domain/DriverAdapter.java +73

1  +
2  +
3  +
4  +
5  +
6  +
7  +
8  + public class DriverAdapter implements TableModel {
9  +     protected Driver driver;
10 +     protected List<Ride> rList;
11 +
12 +     // Define column headers
13 +     private final String[] columnNames = {"From", "To", "Date", "Places", "Price"};
14 +     private final Class<?>[] columnClasses = {String.class, String.class, Date.class,
15 +     Integer.class, Double.class};
16 +
17 +     public DriverAdapter(Driver d) {
18 +         this.driver = d;
19 +         this.rList = d.getCreatedRides(); // Populate ride list from the driver
20 +     }
21 +
22 +     @Override
23 +     public int getColumnCount() {
24 +         return columnNames.length; // Number of columns based on headers
25 +     }
26 +
27 +     @Override
28 +     public String getColumnName(int columnIndex) {
29 +         return columnNames[columnIndex]; // Return column header
30 +     }
31 +
32 +     @Override
33 +     public Class<?> getColumnClass(int columnIndex) {
34 +         return columnClasses[columnIndex]; // Return expected class for each column
35 +     }
36 +
37 +     @Override
38 +     public int getRowCount() {
39 +         return rList.size(); // Number of rides to display
40 +     }
41 +
42 + }
```

Esta segunda imagen es una continuación de la clase DriverAdapter anteriormente mencionada, ya que es un poco larga y no cabe en una sola página. Como se puede ver, sólo se han implementado los métodos realmente necesarios para lo que pedía el enunciado.



```
src/main/java/domain/DriverAdapter.java +73 ...
40 +
41 + @Override
42 + public Object getValueAt(int rowIndex, int columnIndex) {
43 +     Ride ride = rList.get(rowIndex); // Get ride at the specified row index
44 +     switch (columnIndex) {
45 +         case 0: return ride.getFrom();
46 +         case 1: return ride.getTo();
47 +         case 2: return ride.getDate();
48 +         case 3: return ride.getnPlaces();
49 +         case 4: return ride.getPrice();
50 +         default: return null;
51 +     }
52 + }
53 +
54 + @Override
55 + public boolean isCellEditable(int rowIndex, int columnIndex) {
56 +     return false; // Table cells are read-only in this example
57 + }
58 +
59 + @Override
60 + public void setValueAt(Object aValue, int rowIndex, int columnIndex) {
61 +     // Not implemented because table is read-only
62 + }
63 +
64 + @Override
65 + public void addTableModelListener(TableModelListener l) {
66 +     // Not needed for basic implementation
67 + }
68 +
69 + @Override
70 + public void removeTableModelListener(TableModelListener l) {
71 +     // Not needed for basic implementation
72 + }
73 + }
```



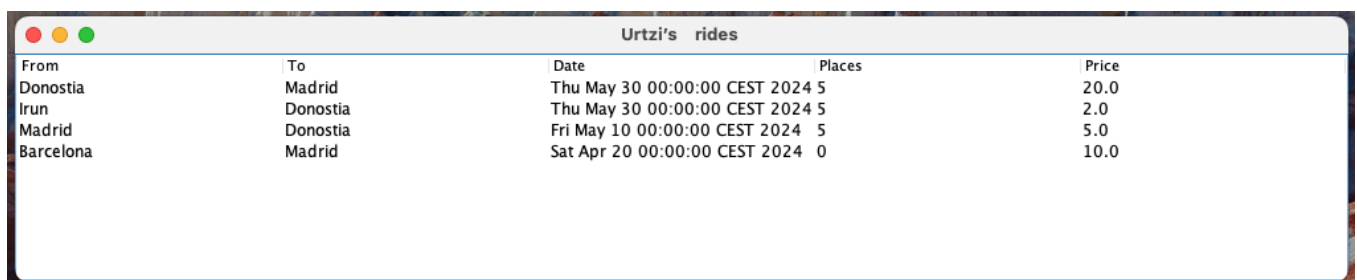
En la imagen que se muestra a continuación, está el código de la clase DriverTable. El código para esta clase ya se proporciona en el enunciado, pero se han añadido algunas líneas de código para asegurar que el contenido de cada columna en la tabla está alineado a la izquierda, al igual que se ve en la imagen de la tabla dada en el enunciado. Básicamente las líneas de código añadidas usan un render para alinear las columnas iterando las columnas y renderizándolas.

```
src/main/java/domain/DriverTable.java  +35  ...
...  @@ -0,0 +1,35 @@
1  + package domain;
2  +
3  + import java.awt.BorderLayout;
4  + import java.awt.Dimension;
5  +
6  + import javax.swing.JFrame;
7  + import javax.swing.JScrollPane;
8  + import javax.swing.JTable;
9  + import javax.swing.table.DefaultTableCellRenderer;
10 + import javax.swing.table.TableModel;
11 +
12 + public class DriverTable extends JFrame{
13 +     private Driver driver;
14 +     private JTable tabla;
15 +     public DriverTable(Driver driver){
16 +         super(driver.getUsername()+"'s rides ");
17 +         this.setBounds(100, 100, 700, 200);
18 +         this.driver = driver;
19 +         DriverAdapter adapt = new DriverAdapter(driver);
20 +         tabla = new JTable(adapt);
21 +         tabla.setPreferredScrollableViewportSize(new Dimension(500, 70));
22 +         // Create a left-aligned cell renderer
23 +         DefaultTableCellRenderer leftRenderer = new DefaultTableCellRenderer();
24 +         leftRenderer.setHorizontalAlignment(DefaultTableCellRenderer.LEFT);
25 +
26 +         // Apply the left-aligned renderer to all columns
27 +         for (int i = 0; i < tabla.getColumnCount(); i++) {
28 +             tabla.getColumnModel().getColumn(i).setCellRenderer(leftRenderer);
29 +         }
30 +         //Creamos un JScrollPane y le agregamos la JTable
31 +         JScrollPane scrollPane = new JScrollPane(tabla);
32 +         //Agregamos el JScrollPane al contenedor
33 +         getContentPane().add(scrollPane, BorderLayout.CENTER);
34 +     }
35 + }
```

En esta imagen se muestra el programa principal RidesMain para mostrar la tabla con la información de los objetos Ride del objeto Driver. En este caso no se ha tocado el código aportado en el enunciado.

```
src/main/java/main/RidesMain.java +19
... @@ -0,0 +1,19 @@
1 + package main;
2 +
3 + import businessLogic.BLFacade;
4 + import businessLogic.BLFactory;
5 + import domain.Driver;
6 + import domain.DriverTable;
7 +
8 + public class RidesMain {
9 +
10 +     public static void main(String[] args) {
11 +         // the BL is local
12 +         boolean isLocal = true;
13 +         BLFacade blFacade = new
14 +         BLFactory().getBusinessLogicFactory(isLocal);
15 +         Driver d= blFacade. getDriver("Urtzi");
16 +         DriverTable dt=new DriverTable(d);
17 +         dt.setVisible(true);
18 +     }
19 + }
```

En esta última imagen se muestra el programa en marcha. Tal y como se ve, funciona según lo esperado, cumpliendo así con lo que se pedía en el enunciado.



From	To	Date	Places	Price
Donostia	Madrid	Thu May 30 00:00:00 CEST 2024	5	20.0
Irun	Donostia	Thu May 30 00:00:00 CEST 2024	5	2.0
Madrid	Donostia	Fri May 10 00:00:00 CEST 2024	5	5.0
Barcelona	Madrid	Sat Apr 20 00:00:00 CEST 2024	0	10.0