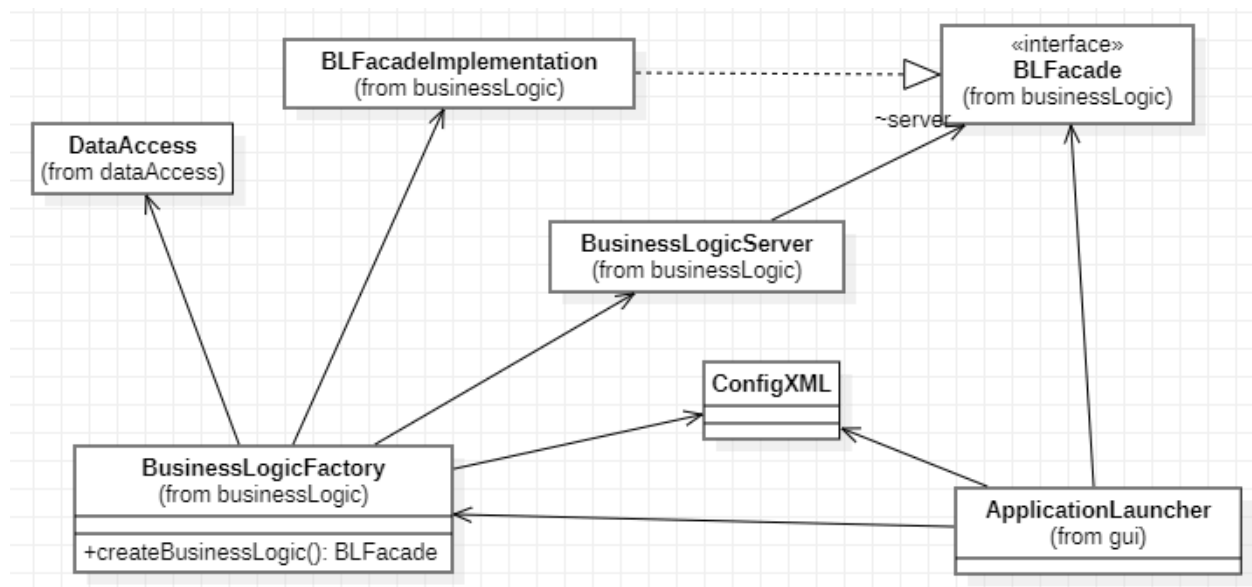


# PATRONES DE DISEÑO

## FactoryMethod:

UML extendido:



ApplicationLauncher llama a BusinessLogicFactory para crear un objeto de la interfaz BLFacade. Este va a crear y devolver un objeto BLFacadeImplementation si ConfigXML.getInstance().isBusinessLogicLocal() es true, o un objeto BusinessLogicServer si es false.

Al crear el objeto BLFacadeImplementation, BusinessLogic se encargará de crear también el objeto DataAccess necesario para esta acción.

En este caso, nuestro Creator es la clase BusinessLogicFactory porque es el encargado de crear el producto, que sería un objeto de una de las clases que implementan BLFacade.

El producto en sí sería la interfaz BLFacade, ya que es lo que el cliente (ApplicationLauncher) quiere recibir del creador.

El Concrete Product sería el objeto verdadero que va a recibir el cliente, es decir, cualquiera de las dos clases que implementan BLFacade: BLFacadeImplementation o BusinessLogicServer, dependiendo de la configuración del fichero config.xml.

## Código modificado:

En la implementación anterior a los cambios, ApplicationLauncher era el encargado de crear el objeto de lógica de negocios que el programa utilizaba después. Para utilizar el patrón Factory Method, se ha extraído esta funcionalidad a la nueva clase BusinessLogicFactory.

Como se puede ver en la siguiente imagen, el programa principal crea un objeto BusinessLogicFactory() (línea 28) y llama a su único método createBusinessLogic() (línea 30) para crear el objeto BLFacade. Luego, el programa lanza el GUI principal, MainGUI, con el objeto BLFacade creado (líneas 32 a 34).

```
11 public class ApplicationLauncher {
12
13     public static void main(String[] args) {
14
15         ConfigXML c = ConfigXML.getInstance();
16
17         System.out.println(c.getLocale());
18
19         Locale.setDefault(new Locale(c.getLocale()));
20
21         System.out.println("Locale: " + Locale.getDefault());
22
23         try {
24
25             BLFacade appFacadeInterface;
26             UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
27
28             BusinessLogicFactory factory = new BusinessLogicFactory();
29
30             appFacadeInterface = factory.createBusinessLogic();
31
32             MainGUI.setBusinessLogic(appFacadeInterface);
33             MainGUI a = new MainGUI();
34             a.setVisible(true);
35
36         } catch (Exception e) {
37             // a.jLabelSelectOption.setText("Error: "+e.toString());
38             // a.jLabelSelectOption.setForeground(Color.RED);
39
40             System.out.println("Error in ApplicationLauncher: " + e.toString());
41         }
42         // a.pack();
43     }
44 }
```

BusinessLogicFactory contiene el método createBusinesslogic, que se encarga de crear el objeto BLFacade.

El programa lee el archivo config.xml, y comprueba si el valor de "local" es true para la configuración de la base de datos. En caso de que sea local, va a crear un objeto BLFacadeImplementation con un nuevo objeto DataAccess (líneas 20 y 21). En caso contrario, va a crear el URL del servidor a partir del fichero config.xml, y creará el objeto Service que hará de base de datos.

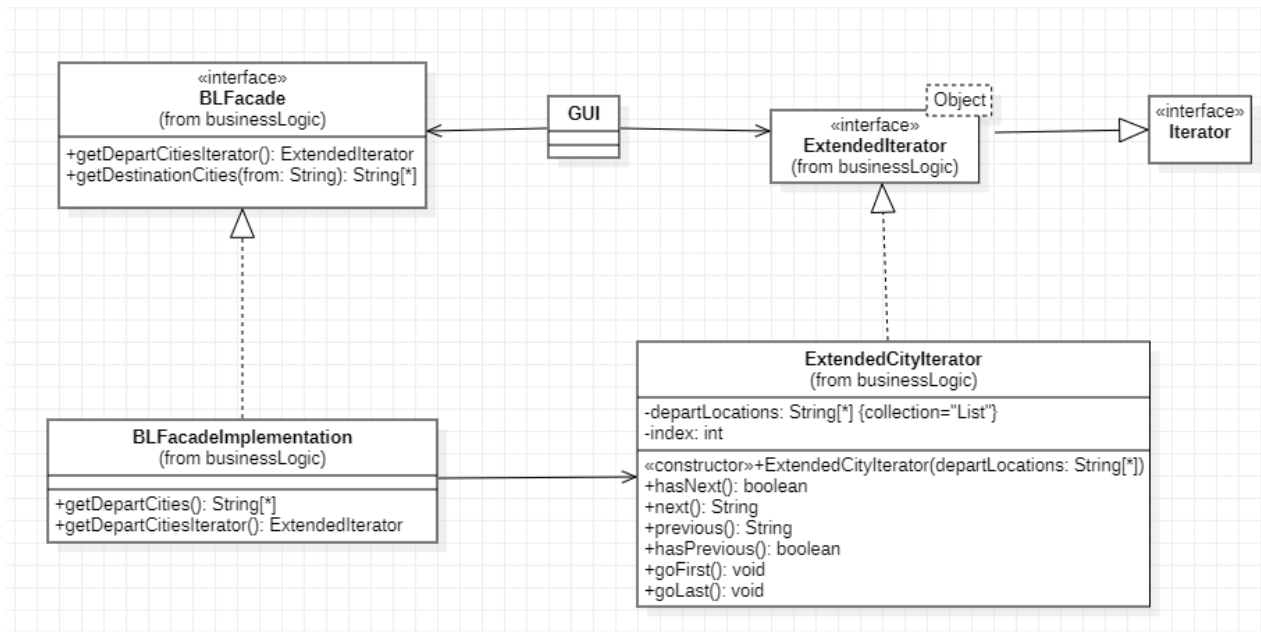
```

12 public class BusinessLogicFactory {
13
14     public BLFacade createBusinessLogic() throws MalformedURLException {
15
16         ConfigXML c = ConfigXML.getInstance();
17
18         if (c.isBusinessLogicLocal()) {
19
20             DataAccess da = new DataAccess();
21             return new BLFacadeImplementation(da);
22
23         }
24
25         else { // If remote
26
27             String serviceName = "http://" + c.getBusinessLogicNode() + ":" + c.getBusinessLogicPort() + "/ws/"
28                 + c.getBusinessLogicName() + "?wsdl";
29
30             URL url = new URL(serviceName);
31
32             // 1st argument refers to wsdl document above
33             // 2nd argument is service name, refer to wsdl document above
34             QName qname = new QName("http://businessLogic/", "BLFacadeImplementationService");
35
36             Service service = Service.create(url, qname);
37
38             return service.getPort(BLFacade.class);
39         }
40     }
41 }

```

## Iterator:

### UML extendido:



Para utilizar el patrón Iterator, hemos creado la clase `ExtendedCityIterator`. Esta clase implementa la interfaz `ExtendedIterator`, que a su vez extiende la Interfaz `Iterator` de java.

La lógica de negocios actualizada contiene el método `getDestinationCitiesIterator`, que utiliza `getDepartCities` para generar un objeto `ExtendedCityIterator` a partir de la lista de Strings recibida.

Cualquiera de las clases GUI puede llamar al método `getDepartCitiesIterator` mediante el `BLFacade` recibido por su constructor, y guardar el objeto en una variable del tipo `ExtendedIterator`.

### Código modificado:

Se ha añadido el método `getDepartCitiesIterator` en `BLFacade` y sus implementaciones. En `BLFacadeImplementation`, el método llama al antiguo `getDepartCities`, y crea un nuevo objeto `ExtendedCityIterator` utilizando la lista recibida.

En la siguiente imagen de la clase **BLFacadeImplementation**, se pueden ver las implementaciones de ambos métodos:

```
53  /**
54   * {@inheritDoc}
55   */
56  @WebMethod
57  public List<String> getDepartCities() {
58      dbManager.open();
59
60      List<String> departLocations = dbManager.getDepartCities();
61
62      dbManager.close();
63
64      return departLocations;
65  }
66
67  /**
68   * {@inheritDoc}
69   */
70  @SuppressWarnings("unchecked")
71  @WebMethod
72  public ExtendedIterator<String> getDepartCitiesIterator() {
73      List<String> departLocations = getDepartCities();
74
75      return (ExtendedIterator<String>) new ExtendedCityIterator(departLocations);
76  }
```

Nuestra clase Iterator recibe una lista de Strings como parámetro para el constructor. Los métodos que siguen son los forzados por la interfaz implementada. Siendo una implementación de ExtendedIterator, debe tener los métodos next(), hasNext(), previous(), hasPrevious(), goFirst() and goLast().

```
6  public class ExtendedCityIterator implements ExtendedIterator<String> {
7
8      private List<String> departLocations;
9      private int index;
10
11  public ExtendedCityIterator(List<String> departLocations) {
12
13      this.departLocations = departLocations;
14      this.index = 0;
15  }
16
17  @Override
18  public boolean hasNext() {
19      return index < departLocations.size();
20  }
21
22  @Override
23  public String next() {
24      if (!hasNext()) {
25          throw new NoSuchElementException("No next element available.");
26      }
27      String value = departLocations.get(index);
28      index++;
29      return value;
30  }
```

```

31
32  @Override
33  public String previous() {
34      if (!hasPrevious()) {
35          throw new NoSuchElementException("No previous element available.");
36      }
37      index--;
38      return departLocations.get(index);
39  }
40
41  @Override
42  public boolean hasPrevious() {
43      return index > 0;
44  }
45
46  @Override
47  public void goFirst() {
48      index = 0; // Move to the beginning of the list
49  }
50
51  @Override
52  public void goLast() {
53      index = departLocations.size(); // Move to one past the last element
54  }
55  }

```

Para probar que el Iterator funciona correctamente, hemos utilizado el siguiente programa principal:

```

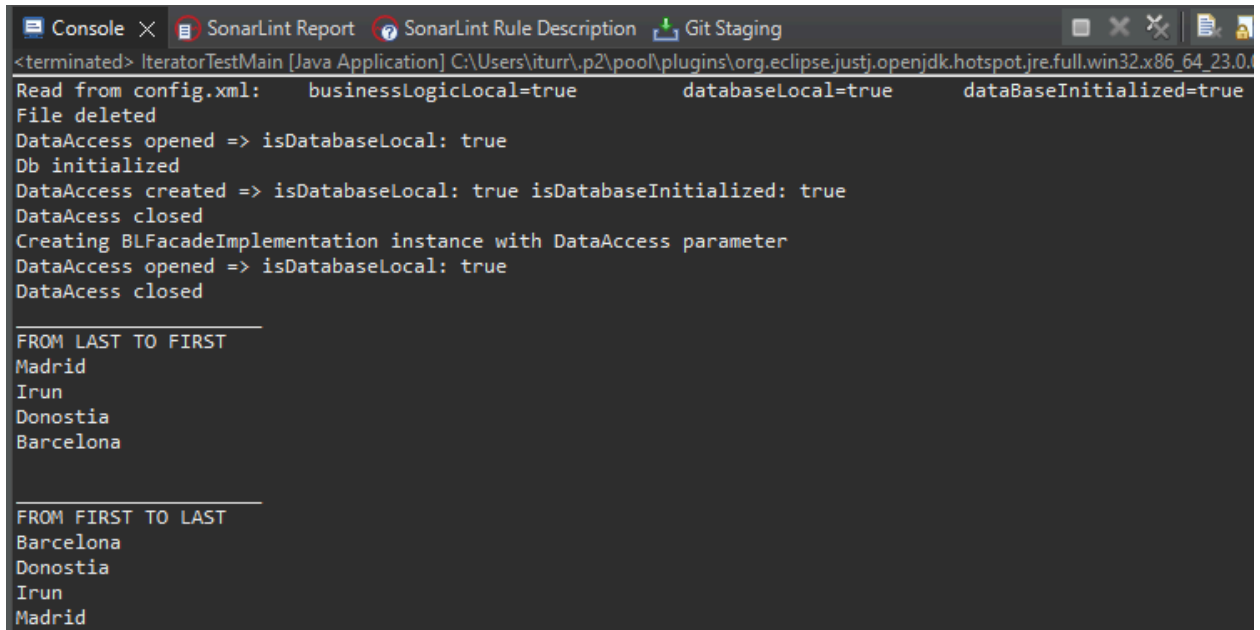
7  public class IteratorTestMain {
8
9      public static void main(String[] args) {
10         // the BL is local
11         ConfigXML config = ConfigXML.getInstance();
12         BLFacade blFacade;
13         try {
14             blFacade = new BusinessLogicFactory().createBusinessLogic(config);
15
16             ExtendedIterator<String> i = blFacade.getDepartCitiesIterator();
17             String c;
18             System.out.println("FROM LAST TO FIRST");
19             i.goLast(); // Go to last element
20
21             while (i.hasPrevious()) {
22                 c = i.previous();
23                 System.out.println(c);
24             }
25
26             System.out.println();
27             System.out.println("FROM FIRST TO LAST");
28             i.goFirst(); // Go to first element
29
30             while (i.hasNext()) {
31                 c = i.next();
32                 System.out.println(c);
33             }
34
35             } catch (MalformedURLException e) {
36                 // TODO Auto-generated catch block
37                 e.printStackTrace();
38             }
39         }
40     }
41 }
42

```

El programa crea un objeto BLFacade utilizando el BusinessLogicFactory que hemos cubierto en el ejercicio anterior. Luego, llama al método getDepartCitiesiterator() y guarda el iterador en una variable tipo ExtendedIterator<String>.

Utilizando los métodos implementados en nuestra clase Iterator, se recorre la lista de ciudades, primero hacia atrás y luego hacia delante, poniendo el nombre en la consola.

Ejecución del programa principal:



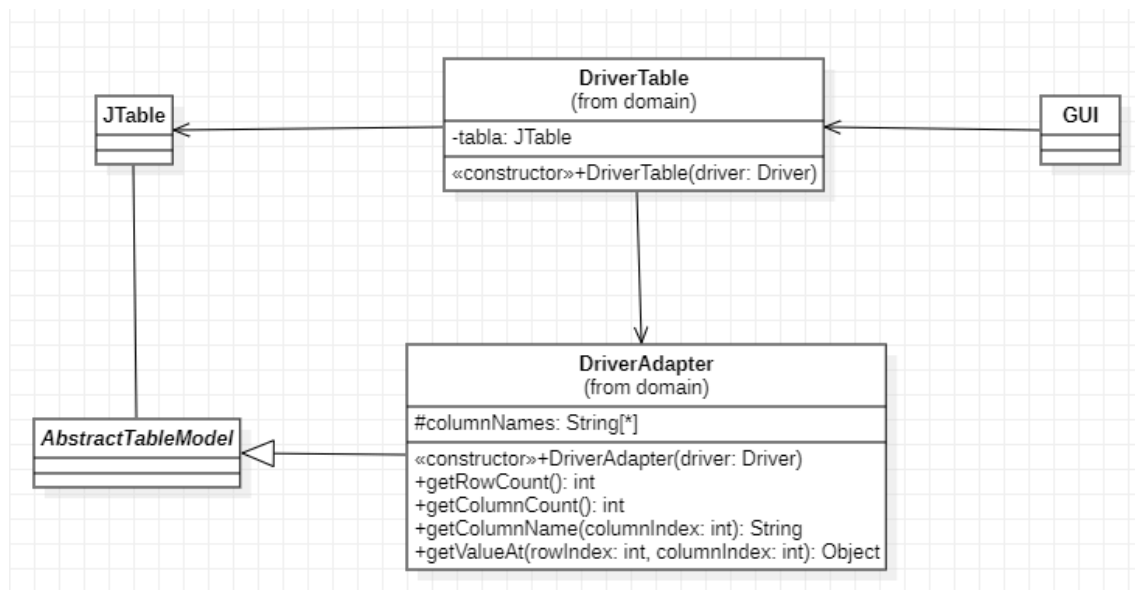
```
<terminated> IteratorTestMain [Java Application] C:\Users\iturr\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_23.0.0
Read from config.xml:    businessLogicLocal=true        databaseLocal=true        dataBaseInitialized=true
File deleted
DataAccess opened => isDatabaseLocal: true
Db initialized
DataAccess created => isDatabaseLocal: true isDatabaseInitialized: true
DataAccess closed
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
```

## Adapter:

UML extendido:



El constructor de un objeto `JTable` recibe como parámetro una tabla del tipo `TableModel`. Nuestra clase `DriverAdapter` extiende `AbstractTableModel`, que a su vez implementa `TableModel`. `DriverTable` crea un `DriverAdapter` para así crear su objeto `JTable`. Cualquiera de las clases de `GUI` puede utilizar `DriverTable` para crear la tabla de un conductor.

## Código modificado:

La clase `DriverTable` solo tiene un constructor, que recibe un `Driver` como parámetro, donde se definen las características de la tabla y se crea el objeto `DriverAdapter` con el `Driver` recibido. Este adapter se utiliza para crear la tabla `JTable`.

```
10 public class DriverTable extends JFrame {
11     private Driver driver;
12     private JTable tabla;
13
14     public DriverTable(Driver driver) {
15         super(driver.getUsername() + "'s rides ");
16         this.setBounds(100, 100, 700, 200);
17         this.driver = driver;
18         DriverAdapter adapt = new DriverAdapter(driver);
19         tabla = new JTable(adapt);
20         tabla.setPreferredScrollableViewportSize(new Dimension(500, 70));
21         //Creamos un JScrollPane y le agregamos la JTable
22         JScrollPane scrollPane = new JScrollPane(tabla);
23         //Agregamos el JScrollPane al contenedor
24         getContentPane().add(scrollPane, BorderLayout.CENTER);
25     }
26 }
```



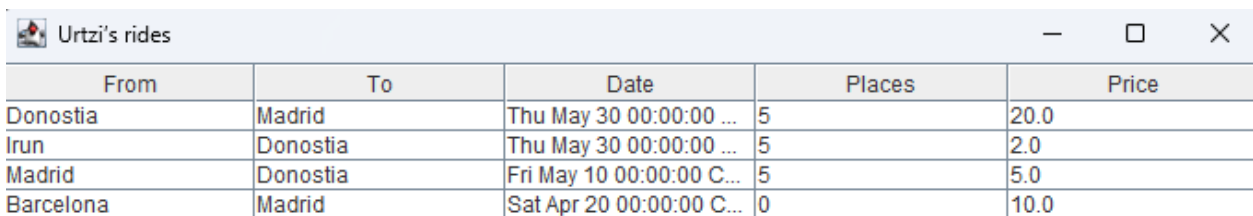
La clase `DriverAdapter` define los nombres de las columnas de la tabla, y recibe un `Driver` mediante su constructor. Los métodos implementados definen las dimensiones de la tabla a crear (`getRowCount()` y `getColumnCount()`), la forma de acceder a los valores (`getValueAt()`) y los nombres de las columnas (`getColumnName()`).

```
7 public class DriverAdapter extends AbstractTableModel {
8
9     protected Driver driver;
10
11     protected String[] columnNames = new String[] { "From", "To", "Date", "Places", "Price" };
12
13     public DriverAdapter(Driver driver) {
14         this.driver = driver;
15     }
16
17     @Override
18     public int getRowCount() {
19         return driver.getCreatedRides().size();
20     }
21
22     @Override
23     public int getColumnCount() {
24         return columnNames.length;
25     }
26
27     @Override
28     public String getColumnName(int columnIndex) {
29         return columnNames[columnIndex];
30     }
31
32     @Override
33     public Object getValueAt(int rowIndex, int columnIndex) {
34         List<Ride> rideList = driver.getCreatedRides();
35         Ride currentRide = rideList.get(rowIndex);
36
37         switch(columnIndex){
38             case 0:
39                 return currentRide.getFrom();
40             case 1:
41                 return currentRide.getTo();
42             case 2:
43                 return currentRide.getDate();
44             case 3:
45                 return currentRide.getnPlaces();
46             case 4:
47                 return currentRide.getPrice();
48         }
49
50         return null;
51     }
52 }
```

Para probar que nuestro nuevo Adapter está bien implementado, se utiliza el siguiente programa principal:

```
11 public class AdapterTestMain {
12
13     public static void main(String[] args) {
14         // the BL is local
15         // boolean isLocal = true;
16
17         ConfigXML config = ConfigXML.getInstance();
18
19         BLFacade blFacade;
20         try {
21             blFacade = new BusinessLogicFactory().createBusinessLogic(config);
22             Driver d= blFacade. getDriver("Urtzi");
23             DriverTable dt=new DriverTable(d);
24             dt.setVisible(true);
25         } catch (MalformedURLException e) {
26             e.printStackTrace();
27         }
28
29     }
30
31 }
```

Ejecución del programa principal:



The screenshot shows a Java Swing window titled "Urtzi's rides" with standard window controls (minimize, maximize, close). Inside the window is a table with five columns: "From", "To", "Date", "Places", and "Price". The table contains four rows of data representing different rides.

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