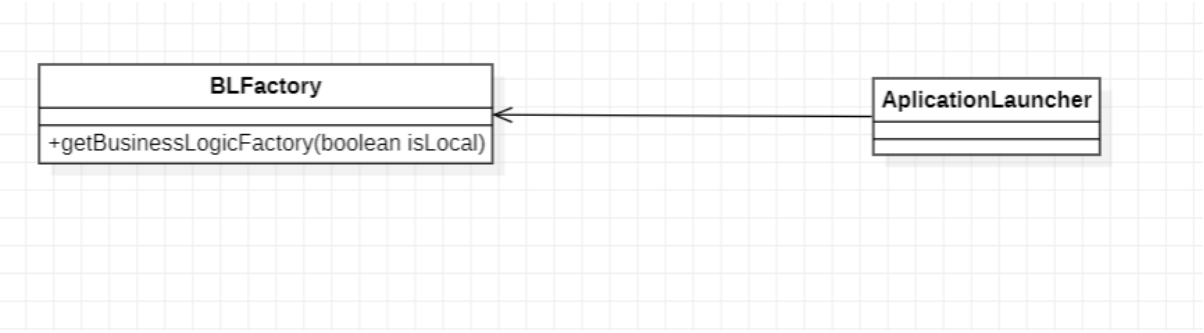
PROYECTO RIDES
PATRONES DE DISEÑO

Autores: David Murguialday, Haritz Gomez e Igor Da Silva

Factory

Diagrama UML

Este es el diagrama UML, donde la clase BLFactory es la que se encarga de crear la instancia de la lógica de negocio y AplicationLauncher se limita a crear una instancia de ella y pedirle que cree la lógica de negocio.



Código modificado

Hemos creado la clase BLFactory que a través del método getBussinessLogicFactory() se encarga de crear la instancia de la lógica de negocio de la aplicación dependiendo del valor que se le pasa como parámetro, que determina si debe ser local o de un servicio web.

Por tanto, en AplicationLauncher hemos modificado la parte del código donde era ella quien creaba la lógica de negocio, por una instancia de BLFactory y llamando a ésta para que cree la lógica de negocio.

Este es el código de AplicationLauncher:

```
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 {

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

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

Y este es el código de BLFactory:

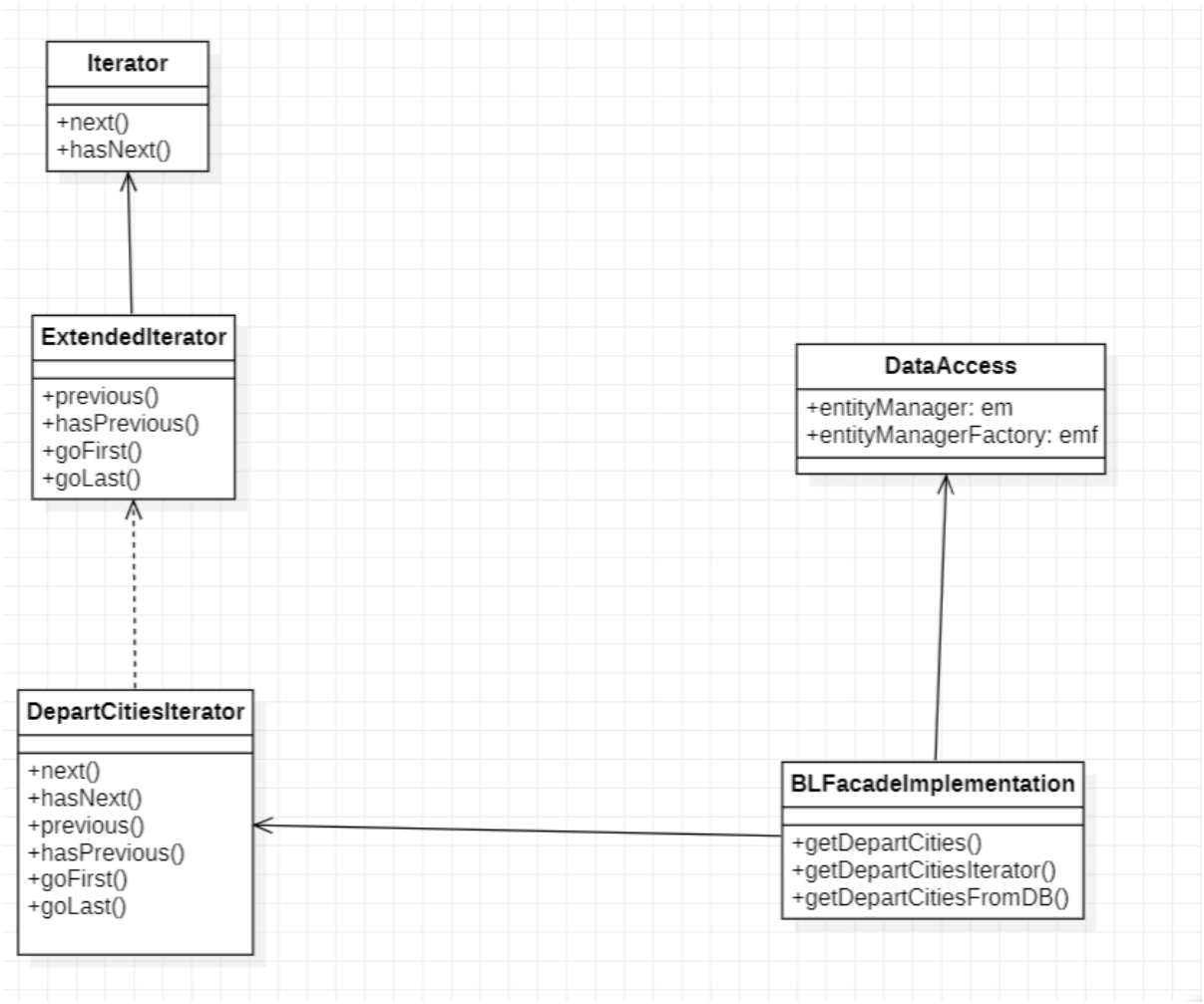
```java
public class BLFactory {

    ConfigXML c = ConfigXML.getInstance();

    public BLFactory() {

    }

    public BLFacade getBusinessLogicFactory(boolean isLocal) {
        try {
            if (isLocal) {
                DataAccess da = new DataAccess();
                return new BLFacadeImplementation(da);
            }
            else {
                String serviceName = "http://" + c.getBusinessLogicNode() + ":" + c.getBusinessLogicPort() + "/ws/"
                        + c.getBusinessLogicName() + "?wsdl";

                URL url = new URL(serviceName);

                // 1st argument refers to wsdl document above
                // 2nd argument is service name, refer to wsdl document above
                QName qname = new QName("http://businessLogic/", "BLFacadeImplementationService");

                Service service = Service.create(url, qname);

                return service.getPort(BLFacade.class);
            }
        } catch(Exception e) {
            e.printStackTrace();
        }

        return null;

    }

}
```

Iterator

Diagrama UML

Hemos creado la interfaz ExtendedIterator, donde declaramos los métodos extra que debe implementar el iterador, que son: comprobar si tienen elemento previo y acceder al elemento previo (esto nos permite recorrer el elemento hacia atrás) y acceder tanto al primer como al último elemento directamente.
Después hemos creado la clase DepartCitiesIterator que implementa a ExtendedIterator e implementa todos los métodos descritos anteriormente.

En la clase BLFacadeImplementation hemos creado un método getDepartCitiesIterator(), para que en vez de devolver una lista de elementos devuelva un ExtendedIterator y asi se puedan recorrer sus elementos como un iterador. También hemos creado el método getDepartCitiesFromDB() para evitar la repetición de código entre el método original getDepartCities() y getDepartCitiesIterator().



Código modificado

Este es el código de la clase DepartCitiesIterator donde simplemente se implementan los métodos de la interfaz ExtendedIterator:

```java
public class DepartCitiesIterator implements ExtendedIterator {

    List<String> depCities;
    int position = 0;

    public DepartCitiesIterator(List<String> depCities) {
        this.depCities = depCities;
    }

    public Object next() {
        String city = depCities.get(position);
        position += 1;
        return city;
    }

    public boolean hasNext() {
        return position < depCities.size();
    }

    public Object previous() {
        String city = depCities.get(position);
        position -= 1;
        return city;
    }

    public boolean hasPrevious() {
        return position >= 0;
    }

    public void goFirst() {
        position = 0;
    }

    public void goLast() {
        position = depCities.size() - 1;
    }

}
```

Esta es la implementación de los métodos de la clase BLFacadeImplementation, donde simplemente creamos un DepartCitiesIterator tras haber recogio antes una lista

con las depart cities que están guardadas en la base de datos.

```java
/**
 * {@inheritDoc}
 */
@WebMethod
public List<String> getDepartCities() {
    List<String> departLocations = getDepartLocationsFromDB();
    return departLocations;

}

public ExtendedIterator<String> getDepartingCitiesIterator(){
    List<String> departLocations = getDepartLocationsFromDB();
    return new DepartCitiesIterator(departLocations);
}

public List<String> getDepartLocationsFromDB(){
    dbManager.open();

    List<String> departLocations = dbManager.getDepartCities();

    dbManager.close();

    return departLocations;
}
```

Ejecución

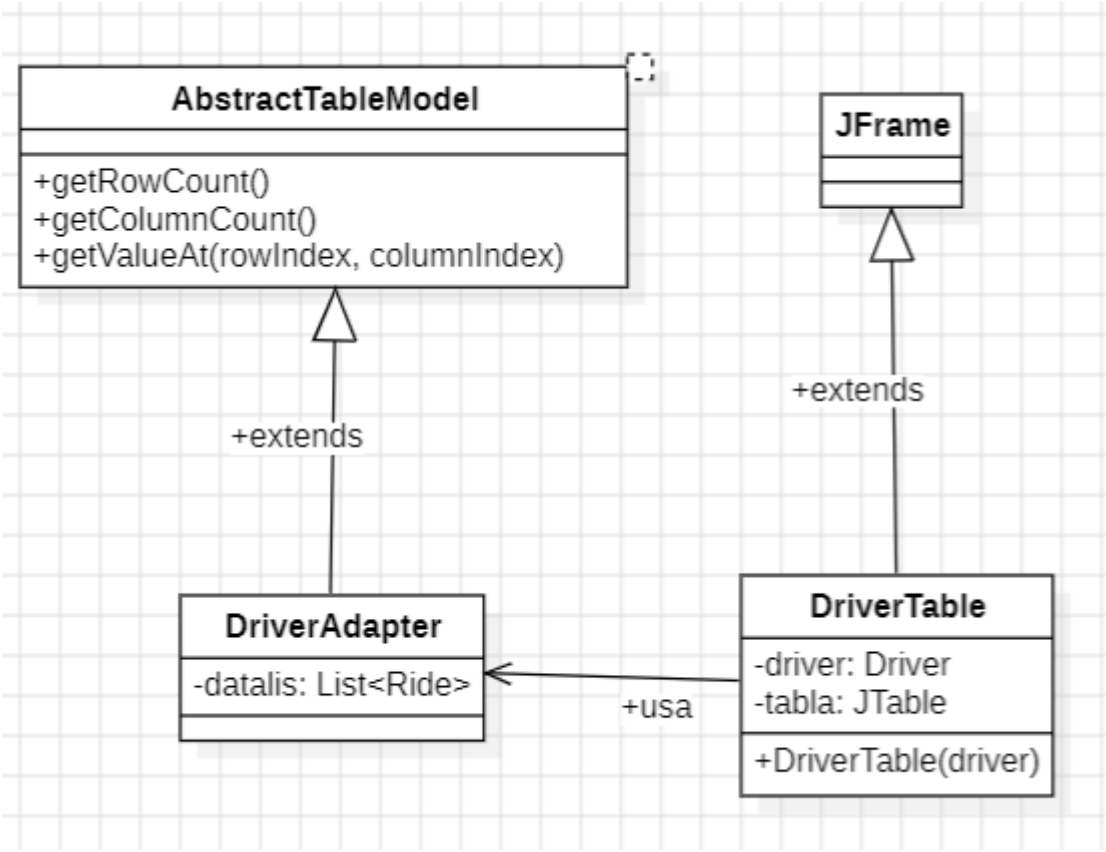
A continuación, adjuntamos las capturas del resultado de la ejecución del siguiente programa:

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

Y el resultado mostrado por consola es:

```
_____________________________
FROM     LAST     TO       FIRST
Madrid
Irun
Donostia
Barcelona


_____________________________
FROM     FIRST    TO       LAST
Barcelona
Donostia
Irun
Madrid
```

Adapter

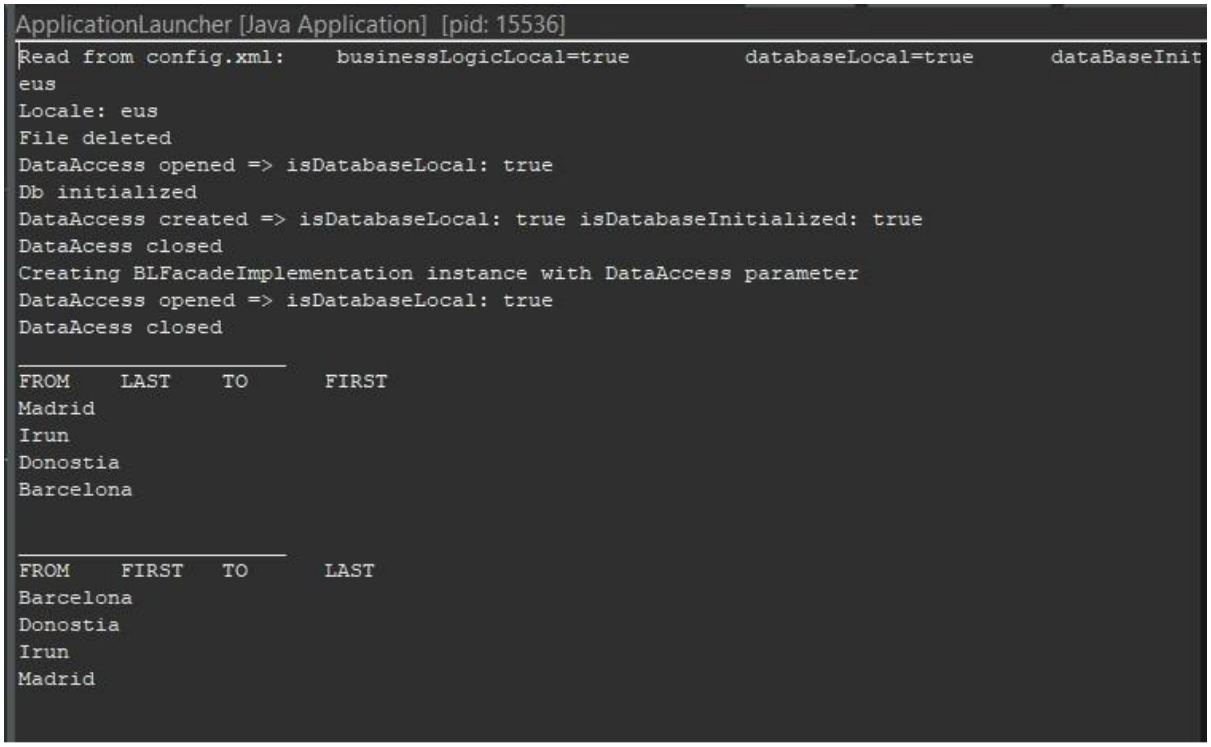


```java
package adapter;

import javax.swing.table.AbstractTableModel;

public class DriverAdapter extends AbstractTableModel{

    private List<Ride> datalist;

    public DriverAdapter(Driver d) {
        datalist = d.getCreatedRides();

    }

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

    @Override
    public int getColumnCount() {
        return 5;
    }

    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        Ride fila = datalist.get(rowIndex);
        switch (columnIndex) {
            case 0:
                return fila.getFrom();
            case 1:
                return fila.getTo();
            case 2:
                return fila.getDate();
            case 3:
                return fila.getnPlaces();
            case 4:
                return fila.getPrice();
            default:
                return null;
        }
    }

}
```

La clase extiende la clase abstracta 'AbstractTableModel' e implementa sus tres métodos. Se necesita una estructura que guarde los viajes del conductor, la cantidad de filas es la misma que el tamaño de la estructura (la cantidad de viajes del conductor), la cantidad de columnas depende de la información que quieras mostrar (en este caso hay 5). Para conseguir un valor concreto se coge el valor (fila, columna) de la estructura.

Ejecuciones

Iterator:



Adapter: