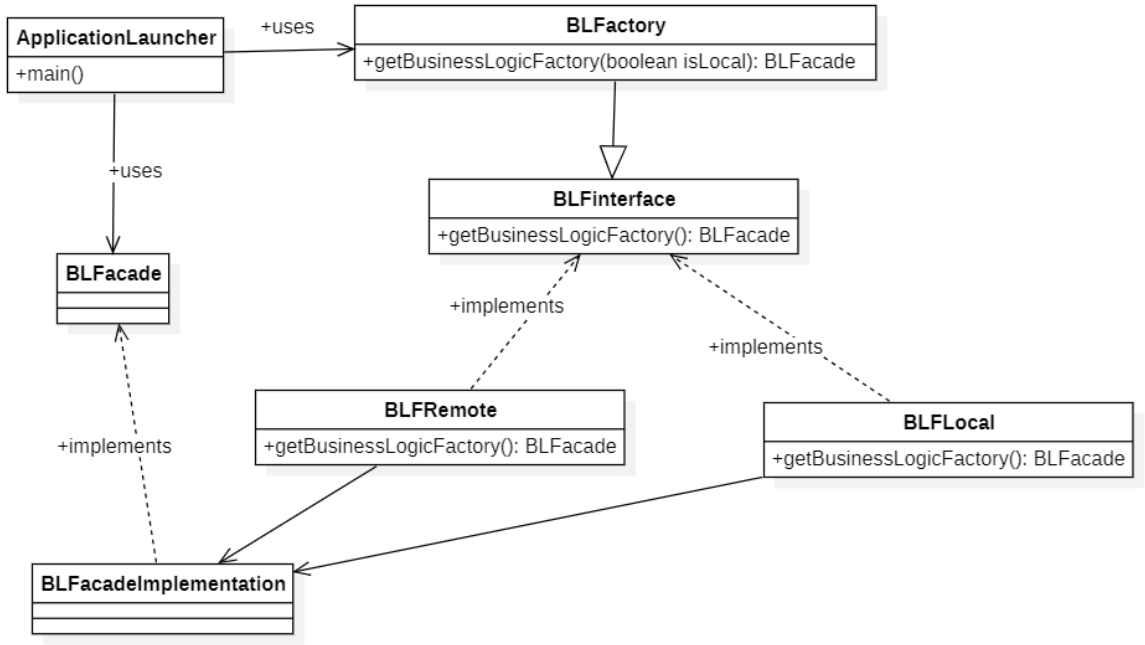
PROYECTO RIDES
PATRONES DE DISEÑO

Anne Baquedano, Mikel León, Mikel Bernal

Repositorio en Github: https://github.com/anneBaque/ridesFinal

Patrón Factory Method

Con la intención de entender mejor los cambios realizados para aplicar el patrón Factory Method hemos creado el siguiente UML con las nuevas clases creadas y a las clases afectadas.



A continuación, está las modificaciones del código con su correspondiente explicación.

Creación de la clase BLFactory:

Se ha añadido una clase intermedia que centraliza la decisión de qué tipo de fábrica usar:

```
public class BLFactory {
    public BLFacade getBusinessLogicFactory(boolean isLocal) {
        BLFInterface blf;
        if(isLocal) blf = new BLFLocal();
        else blf = new BLFRemote();

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

De esta menera se desacopla el código de ApplicationLauncher del proceso de creación. ApplicationLauncher solo indica si es local o remoto y recibe un objeto BLFacade, sin conocer los detalles.

Creación de la interfaz BLFInterface:

Gracias a esta interfaz se permite que existan múltiples fábricas concretas (BLFLocal, BLFRemote) que produzcan diferentes tipos de BLFacade. Contiene el método getBusinessLogicFactory de tipo BLFacade.

Creación de BLFLocal:

Implementa la creación local del BLFacade.

```
public class BLFLocal implements BLFInterface {

    @Override
    public BLFacade getBusinessLogicFactory() {
        DataAccess da = new DataAccess();
        return new BLFacadeImplementation(da);
    }

}
```

Creación de BLFRemote:

Implementa la creación remota del BLFacade. Encapsula la creación de un objeto BLFacade remoto.

```java
public class BLFRemote implements BLFInterface {

    @Override
    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 al lanzar la aplicacion remota: " + e.toString());
        }
        return null;
    }
}
```

Adaptación de ApplicationLauncher:

Antes esaba toda la lógica de selección (local o remota) dentro de esta clase. Ahora se delega toda la responsabilidad de creación a la factoría (BLFactory), simplificando la clase ApplicationLauncher. Esta ahora solo usa el producto, sin conocer cómo se crea.

Así quedaría el método main:

```java
public class ApplicationLauncher {

    public static void main(String[] args) {

        ConfigXML c=ConfigXML.getInstance();

        setUpLocale(c);

        Driver driver=new Driver("driver3@gmail.com","Test Driver");
        MainGUI a=new MainGUI(driver);
        a.setVisible(true);

        try {
            setUILookAndFeel();
            boolean isLocal = c.isBusinessLogicLocal();
            BLFacade appFacadeInterface = new BLFactory().getBusinessLogicFactory(isLocal);
            MainGUI.setBussinessLogic(appFacadeInterface);

        } catch (Exception e) {
            handleInitializationError(a, e);
        }

    }
```

La clase BLFacade juega el papel de Producto, Es la interfaz común que define las operaciones de la lógica de negocio. Tanto la implementación local como la remota devuelven objetos que cumplen esta interfaz.

La clase BLFacadeImplementation es el producto concreto. Es la implementación concreta del producto (BLFacade).

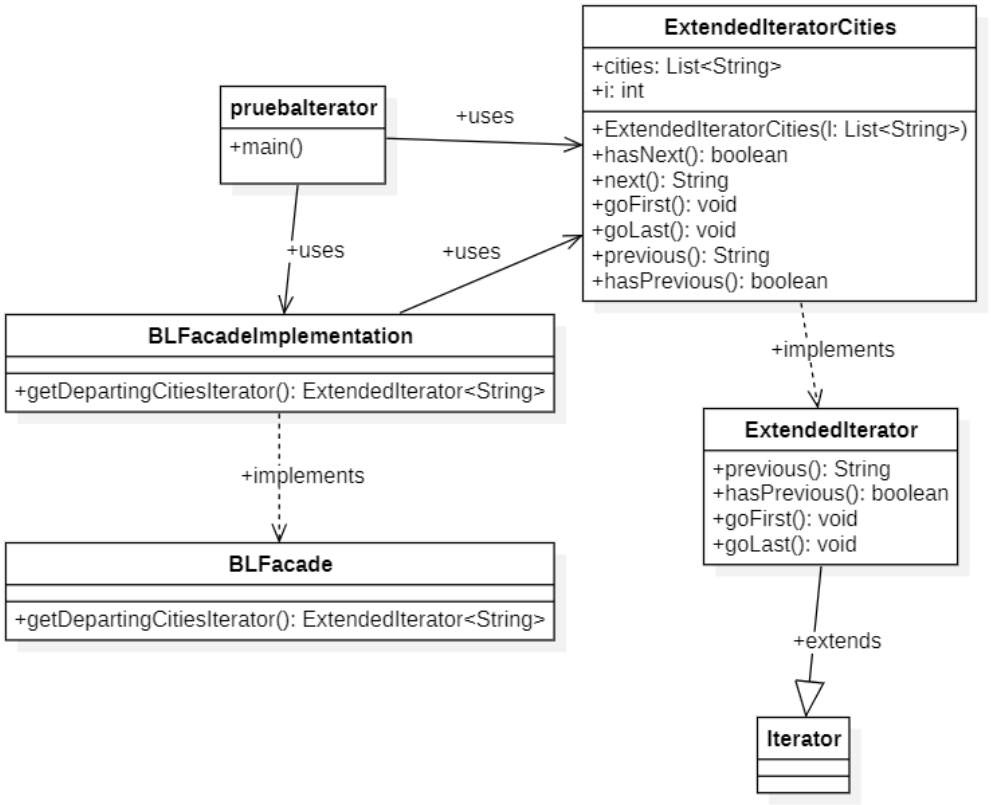
BLFInterface juega el papel de Creator. Es la interfaz de la factoría, que declara el método factory method getBusinessLogicFactory() encargado de crear el BLFacade.

Finalmente, BLFLocal y BLFRemote son los creadores concretos. Implementan el método de creación (getBusinessLogicFactory()), devolviendo el producto apropiado. Cada uno conoce los detalles de cómo construir su versión del BLFacade.

Patrón Iterator

Queremos crear un método que las tareas del método para conseguir las ciudades de embarque, pero que sea tipo ExtendedIterator, por lo que la lista se podrá leer en ambos sentidos.

El UML de la versión extendida es el siguiente:



Respecto al código, primero hemos creado una interfaz donde definiremos los métodos que nos interesa: ir al principio, al final, ver si hay anterior y ver si hay siguiente. Esta interfaz es una extensión de la clase Iterator. A continuación, se puede ver la interfaz creada:

```
package businessLogic;

import java.util.Iterator;

public interface ExtendedIterator<String> extends Iterator<String> {
    public String 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();
}
```

Luego hemos hecho la implementación de esta interfaz en la clase ExtendedIteratorCities que es la que usaremos en la BLFacade. Aquí tenemos dos atributos, uno donde se guardan las ciudades y otro donde tendremos el índice guardado. La implementación es la siguiente:

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

    public ExtendedIteratorCities(List<String> l) {
        this.cities = l;
        i = -1;
    }

    @Override
    public boolean hasNext() {
        return i < cities.size()-1;
    }

    @Override
    public String next() {
        i++;
        return cities.get(i);
    }

    public void goFirst() {
        i = -1;
    }

    public void goLast() {
        i = cities.size();
    }

    @Override
    public String previous() {
        i --;
        return cities.get(i);
    }
```

```java
@Override
public boolean hasPrevious() {
    return i!=0;
}
```

Además, tenemos que crear el método en la BLFacade y luego implementarlo en la clase BLFImplementation, en ese orden las fotos. En la implementación con el fin de evitar el código excesivo, hemos reutilizado el código de la función que queremos recrear llamando a la función.

```java
public ExtendedIterator<String> getDepartingCitiesIterator();
```

```java
public ExtendedIterator<String> getDepartingCitiesIterator(){
    List<String> lista = this.getDepartCities();
    return new ExtendedIteratorCities(lista);

}
```

Por último, hemos copiado la clase de prueba ofrecida en la actividad para comprar su correcto funcionamiento.

```java
package testIterator;

import businessLogic.BLFacade;

public class pruebaIterator {

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

}
```

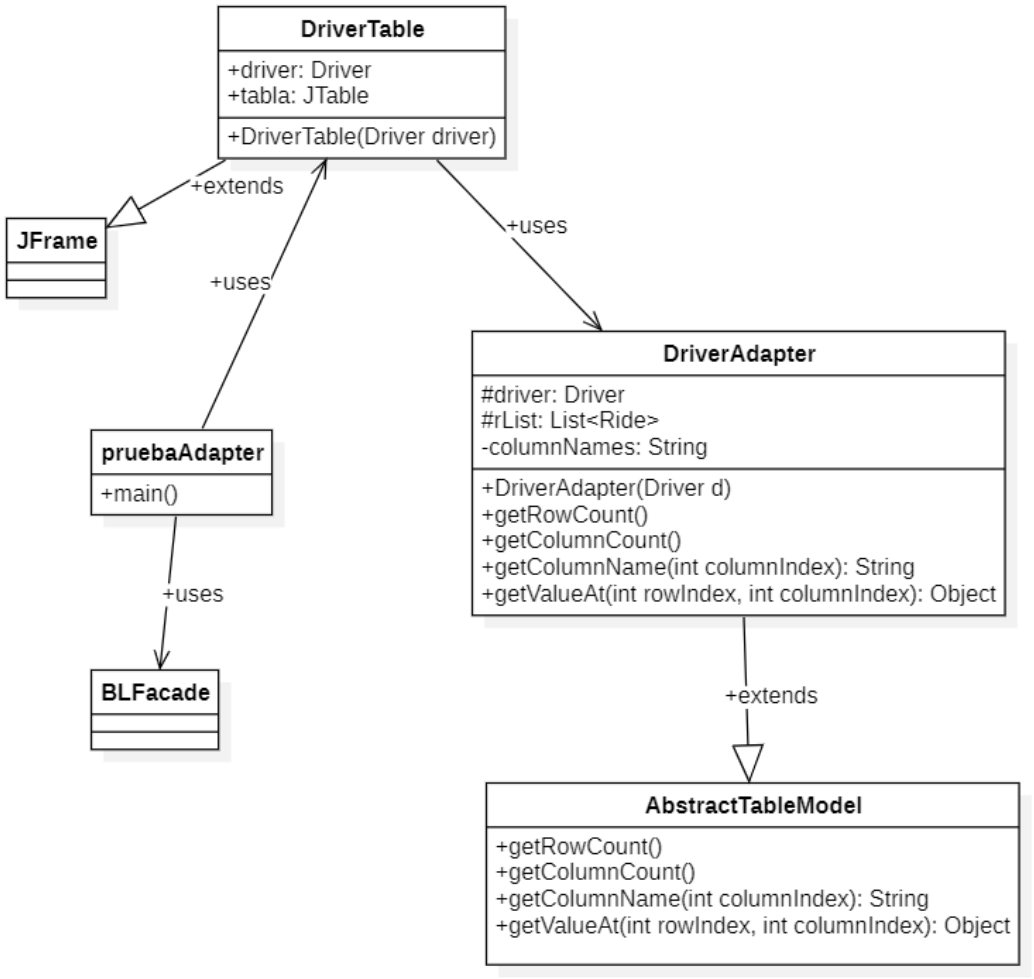
Y aquí tenemos una foto de la ejecución del programa.

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


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

Patrón Adapter

Como en los patrones anteriores tenemos un UML para entender los cambios hechos en el código. Tenemos las clases nuevas añadidas con sus métodos y patrones junto con las clases afectadas por los cambios.



Hemos creado una nueva clase DriverAdapter para poder mostrar la información de los viajes asociados a un driver concreto. Principalmente se definen las columnas de la tabla y se implementan los métodos de la interfaz. Además, hemos implementado el método getColumnName para mostrar los nombres de las columnas de las tablas con el texto que hemos definido en columnNames.

```java
public class DriverAdapter extends AbstractTableModel {

    protected Driver driver;
    protected List<Ride> rList;
    private final String[] columnNames = {"From", "To", "Date", "Places", "Price"};

    public DriverAdapter(Driver d) {
        this.driver = d;
        this.rList = d.getRides();
    }

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

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

    @Override
    public String getColumnName(int columnIndex) {
        return columnNames[columnIndex];  // Return column header
     }

    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        Ride ride = rList.get(rowIndex);  // Get ride at the specified row index
        switch (columnIndex) {
            case 0: return ride.getFrom();
            case 1: return ride.getTo();
            case 2: return ride.getDate();
            case 3: return ride.getnPlaces();
            case 4: return ride.getPrice();
            default: return null;
        }
    }
}
```

Se ha creado la nueva clase DriverTable con el código proporcionado en el enunciado.

Y finalmente se ha creado una clase en src/test/java/testAdapter/pruebaAdapter.java para probar el funcionamiento del patrón. Se ha utilizado el método main proporcionado en el enunciado.

Hemos creado el driver Urtzi y sus rides similar a los del ejemplo del enunciado para almacenarlo en la bd.