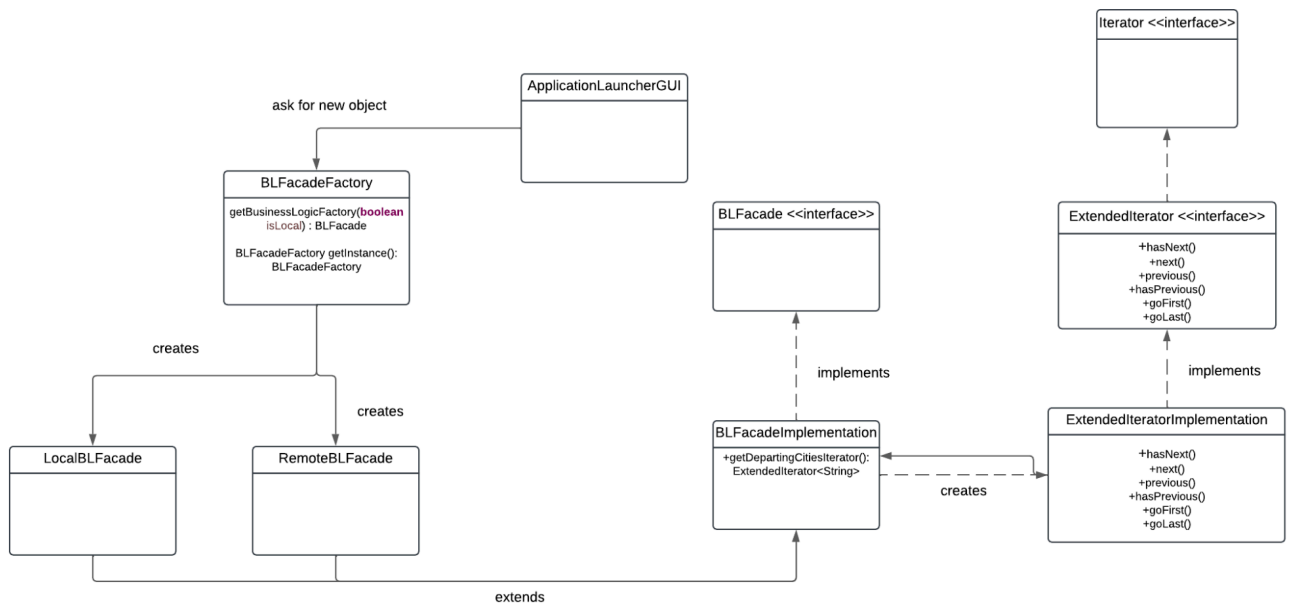


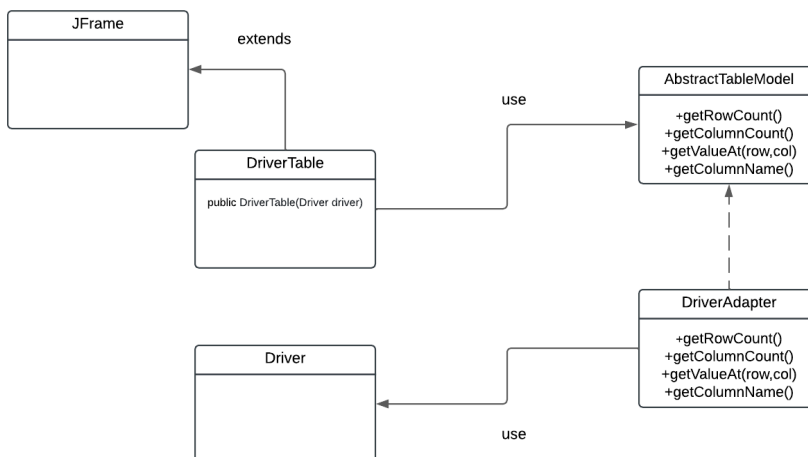
Patrones de diseño

UML extendido con los cambios realizados

Factory e Iterator:



Adapter:



Código modificado

Fix en DataAccess

Hemos deshecho un cambio realizado durante el proyecto de refactorización que causaba que ciertas queries con la base de datos resultara en un fallo de ejecución.

Ejemplo de código modificado:

```
public User getUser(String erab) {
    TypedQuery<User> query = db.createQuery(
        "SELECT u FROM User u
        WHERE u.username = :username", User.class);
    setParam(query, erab);
    return query.getSingleResult();
}
```

Simple Factory

Hemos añadido una clase basada en el patrón Simple Factory

Código de la clase:

```
public class BLFacadeFactory {
    private static BLFacadeFactory instance = null;

    private BLFacadeFactory() {}

    public static BLFacadeFactory getInstance() {
        if (instance == null) {
            instance = new BLFacadeFactory();
        }
        return instance;
    }

    public BLFacade getBusinessLogicFactory(
        boolean isLocal) throws Exception {
        if (isLocal) {
            return new LocalBLFacade();
        } else {
            return new RemoteBLFacade();
        }
    }
}
```

```
}
```

La clase se ha creado utilizando el patrón Singleton de modo que no se crea públicamente un objeto de esta clase si no que se solicita la instancia de forma estática y si no existe es la propia clase la que crea la instancia. El método `getBusinessLogicFactory` se encarga de devolver la implementación de la lógica de negocios necesaria, local o remota.

Clase de la Remota:

```
public class RemoteBLFacade extends BLFacadeImplementation {
    private final BLFacade remoteFacade;

    public RemoteBLFacade() throws Exception {
        ConfigXML config = ConfigXML.getInstance();
        String serviceName = "http://" + config.getBusinessLogicNode() + ":" + config.getBusinessLogicPort() + "/ws/"
            + config.getBusinessLogicName() + "?wsdl";
        URL url = new URL(serviceName);
        QName qname = new QName("http://businessLogic/", "BLFacadeImplementationService");
        Service service = Service.create(url, qname);
        remoteFacade = service.getPort(BLFacade.class);
    }
}
```

Clase de la Local:

```
package businessLogic;

import dataAccess.DataAccess;

public class LocalBLFacade extends BLFacadeImplementation {
    private final BLFacadeImplementation localFacade;

    public LocalBLFacade() {
        DataAccess dataAccess = new DataAccess();
        this.localFacade = new BLFacadeImplementation(dataAccess);
    }
}
```

Esta clase se utiliza en la clase `ApplicationLauncher` sustituyendo el bloque `If Else` que existía anteriormente por la ejecución del `getInstance` de forma estática y el método `getBusinessLogicFactory` resaltados en azul.

```
public class ApplicationLauncher {
    public static void main(String[] args) {
        ConfigXML config = ConfigXML.getInstance();

        Locale.setDefault(new Locale(config.getLocale()));
        System.out.println("Locale: " + Locale.getDefault());
        try {
            boolean isLocal=config.isBusinessLogicLocal();
            UIManager.setLookAndFeel(
```

```

        "javax.swing.plaf.metal.MetalLookAndFeel");
        BLFacadeFactory factory =
            BLFacadeFactory.getInstance();
        BLFacade appFacadeInterface =
            factory.getBusinessLogicFactory(isLocal);
        MainGUI.setBussinessLogic(
            appFacadeInterface);
        MainGUI mainGUI = new MainGUI();
        mainGUI.setVisible(true);
    } catch (Exception e) {
        System.out.println("Error in
            ApplicationLauncher: " + e.toString());
    }
}
}

```

De modo que si cambia la obtención de la implementación de la lógica de negocio solo sea necesario cambiar el factory en lugar de tener que cambiar el lanzador de la aplicación y manteniendo así el principio OCP.

Iterator

Hemos creado la siguiente interfaz que extiende a iterador:

```

public interface ExtendedIterator<Object>
    extends Iterator<Object> {
    //return the actual element and go to the previous
    public Object 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();
}

```

A continuación hemos implementado este iterador extendido creando la siguiente clase:

```
public class ExtendedIteratorImplementation
    implements ExtendedIterator {
    List<String> list;
    Integer index;

    public ExtendedIteratorImplementation(List<String> list) {
        this.index = 0;
        this.list = list;
    }
    @Override
    public boolean hasNext() {
        return index < list.size();
    }
    @Override
    public String next() {
        String s = list.get(index);
        index++;
        return s;
    }
    @Override
    public String previous() {
        String s = list.get(index);
        index--;
        return s;
    }
    @Override
    public boolean hasPrevious() {
        return index >= 0;
    }
    @Override
    public void goFirst() {
```

```

        index = 0;
    }
    @Override
    public void goLast() {
        index = list.size() - 1;
    }
}

```

Por último hemos creado un nuevo método en la implementación de la lógica de negocio, `getDepartingCitiesIterator`:

```

@Override
public ExtendedIterator<String> getDepartingCitiesIterator() {
    dbManager.open();
    ExtendedIterator<String> list=
        new ExtendedIteratorImplementation(
            dbManager.getDepartCities());
    dbManager.close();
    return list;
}

```

Iterador y Adaptador

Para poder utilizar la lista de Rides en un `JTable` hemos creado una nueva clase que sigue el patrón adaptador. Adicionalmente como se trata de una lista de Rides hemos decidido que podría implementarse también mediante un iterador.

```

public class DriverAdapter extends AbstractTableModel {
    protected Driver d;
    protected String[] columnNames =
        new String[] { "from", "to", "date", "places", "price" };
    public DriverAdapter(Driver d) {
        this.d = d;
    }
    public int getColumnCount() {
        return this.columnNames.length;
    }
}

```

```

    }
    public String getColumnName(int i) {
        if (i >= 0 && i < columnNames.length) {
            return columnNames[i];
        }
        return "";
    }
    public int getRowCount() {
        return this.d.getCreatedRides().size();
    }
    public Object getValueAt(int row, int col) {
        Ride r = null;
        Iterator<Ride> it = this.d.getCreatedRides().iterator();
        for (int i = 0; i <= row && it.hasNext(); i++) {
            r = it.next();
        }
        if (r != null) {
            if (col == 0)
                return r.getFrom();
            else if (col == 1)
                return r.getTo();
            else if (col == 2)
                return r.getDate();
            else if (col == 3)
                return r.getnPlaces();
            else if (col == 4)
                return r.getPrice();
            else
                return null;
        }
        return null;
    }
}

```

El mayor reto de esta clase ha sido gestionar la obtención de los valores de la lista para que puedan ser mostrados en la tabla correctamente. Para ello hay que prestar atención a la fila y columna solicitadas y devolver lo necesario en cada caso. Por otro lado Driver ya tenía implementado un método que nos devuelve un iterador que nos ha sido de ayuda para ciclar hasta la fila solicitada.

Este adaptador lo utilizamos en una nueva clase DriverTable que es la encargada de crear el JTable que se añade a continuación al JFrame. El código es el siguiente:

```
public class DriverTable extends JFrame {
    private Driver driver;
    private JTable tabla;
    public DriverTable(Driver driver) {
        super(driver.getUsername() + "s rides ");
        this.setBounds(100, 100, 700, 200);
        this.driver = driver;
        DriverAdapter adapt = new DriverAdapter(driver);
        tabla = new JTable(adapt);
        tabla.setPreferredScrollableViewportSize(
            new Dimension(500, 70));
        // Creamos un JScrollPane y le agregamos la JTable
        JScrollPane scrollPane = new JScrollPane(tabla);
        // Agregamos el JScrollPane al contenedor
        getContentPane().add(
            scrollPane, BorderLayout.CENTER);
    }
}
```

Para utilizar este JFrame hemos creado el siguiente programa principal.

```
public class testDriverAdapter {
    public static void main(String[] args) {
        boolean isLocal = true;
        BLFacade blFacade;
        try {
            blFacade = BLFacadeFactory.getInstance().
```



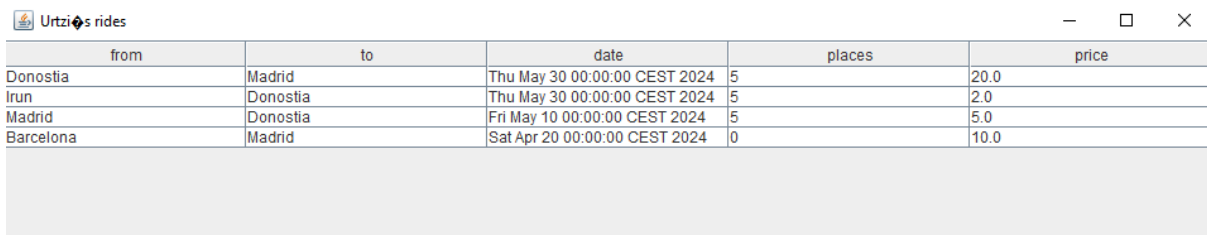
```

        getBusinessLogicFactory(isLocal);
        Driver d = blFacade.getDriver("Urtzi");
        DriverTable dt = new DriverTable(d);
        dt.setVisible(true);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Capturas de Iterador y Adaptador

En este apartado se presentan los resultados de la ejecución del programa principal del adaptador y iterador.



The screenshot shows a Java Swing window titled "Urtzi's rides". Inside the window is a table with five columns: "from", "to", "date", "places", and "price". The table contains four rows of data. The window has standard Mac OS X window controls (red, yellow, and green buttons) in the top-left corner.

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

Cabe destacar que para que se visualice completamente las fechas de la columna data es necesario aumentar con el ratón el tamaño de la ventana.