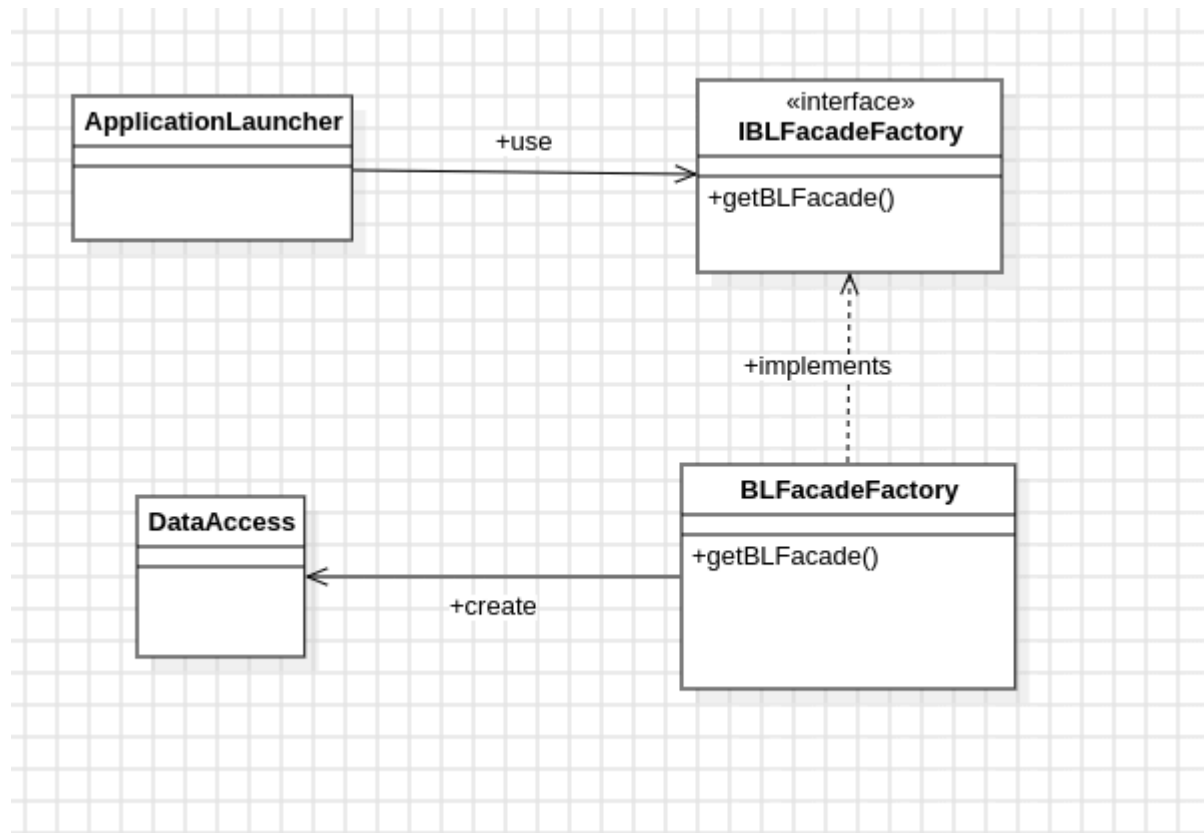


Factory Method Patroia:

UML-a:



ApplicationLauncher-ek berak BLFacade-a sortu beharrian, IBLFacadeFactory motako objektu bati deitu dio, eta berak sortuko du BLFacade hori. IBLFacadeFactory sortzea ez da beharrezko, baina etorkizunean BLFacadeFactory mota bat baino gehiago badago, nahikoa izango litzake interfaze hori implementatuko duen beste klase bat sortzea.

KODEA:

```
IBLFacadeFactory faktoria = new BLFacadeFactory();
BLFacade appFacadeInterface = faktoria.getBLFacade();
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");

MainGUI.setBussinessLogic(appFacadeInterface);
```

Aplication launcherren, BLFacadeFactory berri bat sortuko du, eta bere `.getBLFacade` metodoari deituko dio BLFacade motako objektua eskuratzeko.

```
package businessLogic;

public interface IBFacadeFactory {
    public BLFacade getBLFacade() throws Exception;
}
```

IBLFacadeFactory interfazeaz, getBLFacade metodoa bakarrik definituko dugu.

```
public class BLFacadeFactory implements IBFacadeFactory{

    public BLFacade getBLFacade() throws Exception {
        ConfigXML c = ConfigXML.getInstance();

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

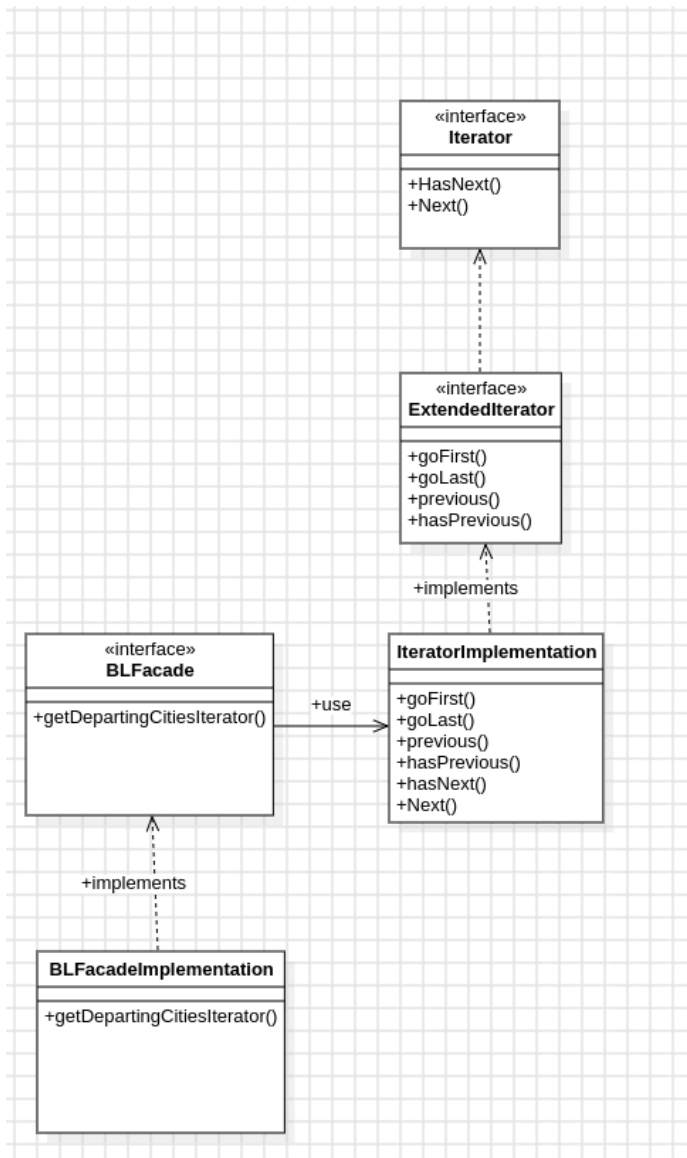
            URL url = new URL(serviceName);
            QName qname = new QName("http://businessLogic/", "BLFacadeImplementationService");

            Service service = Service.create(url, qname);
            return service.getPort(BLFacade.class);
        }
    }
}
```

BLFacadeFactory klasean, berriz, IBLFacadeFactory klasea inplementatuko du, eta beraz, bere metodoak ere inplementatu beharko dira. Bere metodoan, aurrez application launcher-en genuen kode bera jarriko dugu.

Iterator Patroia:

UML-a:



Iterator interfazeak ezagunak zaizkigun bi metodo ditu, `hasNext()` eta `Next()`. Gure kasuan beste hainbat metodo gehitu nahi ditugunez, iteratorren beste interfaze bat sortu dugu, **ExtendedIterator** izenekoa. Ondoren, interfaze horietako metodoak inplementatu dituen **IteratorImplementation** klasea sortu dugu. Gure `getDepartingCitiesIterator()` metodoak **IteratorImplementation** klaseko Objektu bat sortu eta erabiliko du.

Kodea:

IteratorImplementation klasearen kodea:

```
public class IteratorImplementation<String> implements ExtendedIterator<String>{
    private Vector<String> data;
    private int index;

    public IteratorImplementation(Vector<String> data) {
        this.data = data;
        this.index = 0;
    }

    @Override
    public boolean hasNext() {
        return index < data.size();
    }

    @Override
    public String next() {
        return data.get(index++);
    }

    @Override
    public boolean hasPrevious() {
        return index > 0;
    }

    @Override
    public String previous() {
        return data.get(--index);
    }

    @Override
    public void goFirst() {
        index = 0;
    }

    @Override
    public void goLast() {
        index = data.size();
    }
}
```

Eraikitzailean data izeneko bektore bat jasotzen du. Kodean, `Iterator<Object>` interfazearen metodoak inplementatzen dira (next eta hasNext besteak beste). Bestalde, interfazean sortu ditugun metodo berriak ere inplementatu dira.

Extended Iterator interfazearen klasea:

```

package iterator;

import java.util.Iterator;

public interface ExtendedIterator<Object> extends Iterator<Object>{

    public Object previous();

    public boolean hasPrevious();

    public void goFirst();

    public void goLast();

}

```

Hemen Iterator objektuak ez dituen metodo batzuk definitzen dira. Kasu honetan 4 metodo definitu dira. Interfaze erabiltzen duen klasearen lana izango da metodoak implementatzea

Exekuzio Irudia:

```

terminated> IteratorProbatu [Java Application] /usr/lib/jvm/java-21-openjdk-amd64/bin/java (13 nov 2025, 18:58:18 - 18:58:18 elapsed: 0:00:00.467) [pid: 38297]
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
Eibar
Donostia
Bilbo

FROM      FIRST      TO      LAST
Bilbo
Donostia
Eibar

```

Horretarako 3 bidaia dituen gidari bat sortu dugu.

```

i.goLast(); // Go to last element
while (i.hasPrevious()) {
    c = i.previous();
    System.out.println(c);
}

```

Lehenik eta behin kode honen bidez, azkeneko elementura joan gara, eta atzetik aurrera elementu guztiak korritu ditugu.

```

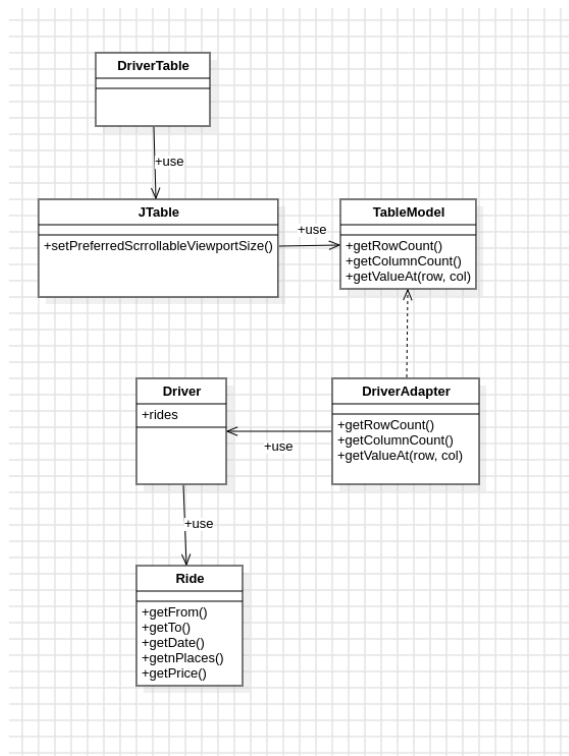
i.goFirst(); // Go to
while (i.hasNext()) {
    c = i.next();
    System.out.println(c);
}

```

Ondoren gauza bera egin dugu, baina alderantziz.

Adapter Patroia:

UML-a:



Zeregin hau egiteko bi klase sortu ditugu. Batean, Driver baten balioak taulan agertzeko balioak moldatu edo adaptatzen ditu. Klase honen izena DriverAdapter da. DriverTable klaseak, berriz, aurrez azaldutako klaseari driver objektu bat pasata, taula bat sortzen du.

KODEA:

DriverAdapter klasearen kodea:

```
public class DriverAdapter extends AbstractTableModel{

    private Driver driver;
    private List<Ride> rides;
    private String[] columnNames = {"From", "To", "Date", "Places", "Price"};
    public DriverAdapter(Driver driver) {
        this.driver = driver;
        this.rides = driver.getRides();
    }

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

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

    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        Ride ride = rides.get(rowIndex);
        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;
        }
    }

    @Override
    public String getColumnName(int column) {
        return columnNames[column];
    }
}
```

Kode honetan, driver bat jasotzen du eraikitzailean, eta bere bidaiak lortzen ditu. Horrez gain, getValueAt metodoan, parametro gisa pasatzen diguten rowIndex balioak driverraren zenbatgarren bidaiak nahi dugun esan nahi du. columnIndex parametroak, berriz, bidai baten zenbatbarren parametroa nahi dugun esan nahi du, 0=from, 1=to... izanik.

DriverTable klasearen kodea:

```
package adapter;

import java.awt.BorderLayout;

public class DriverTable extends JFrame {

    private Driver driver;
    private JTable tabla;

    public DriverTable(Driver driver){
        super(driver.getName()+"'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);
    }

}
```

Bertan, driver objektua lehioari izena emateko erabiltzen da. Horrez gain, DriverAdapter motako objektu bat sortzen da, eta objektu hori erabiltzen da taula sortzeko. Ondoren, taula hori bera erabiltzen da irratatzeko aukera ematen duen beste taula bat sortzeko.

Exekuzio Irudia:

Lur's rides				
From	To	Date	Places	Price
Bilbao	Donostia	Thu Nov 13 17:41:28 CET 2025	3.0	50.0
Gasteiz	Iruñea	Thu Nov 13 17:41:28 CET 2025	3.0	70.0
Donostia	Bilbao	Thu Nov 13 17:41:28 CET 2025	3.0	60.0

Kasu honetan, 3 bidaia ditu Lur gidariak, eta bidaia bakoitzean bere 5 parametroak agertzen dira.