Documentación de Patterns

Jon Dorronsoro y Jhon Díaz

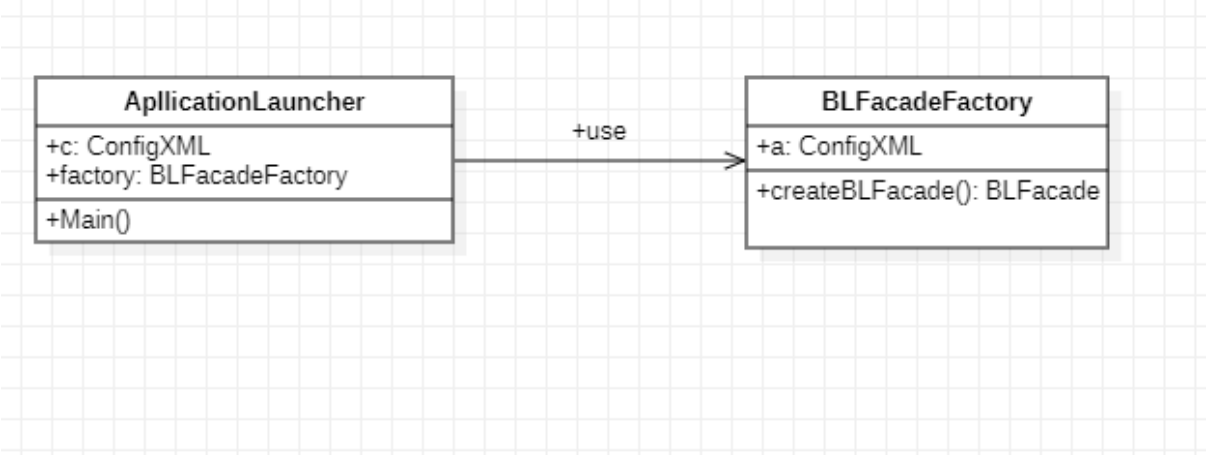
1. Patrón factory method

```java
package Factoria;

import java.net.MalformedURLException;

public class BLFacadeFactory {
    ConfigXML a;

    public BLFacadeFactory(ConfigXML c) {
        // TODO Auto-generated constructor stub
        this.a = c;
    }

    public BLFacade createBLFacade() {

        if (a.isBusinessLogicLocal()) {
            DataAccess da = new DataAccess(a.getDataBaseOpenMode().equals("initialize"));
            BLFacade appFacadeInterface = new BLFacadeImplementation(da);
            return appFacadeInterface;

        } else {
            // TODO Auto-generated method stub
            String serviceName = "http://" + a.getBusinessLogicNode() + ":" + a.getBusinessLogicPort() + "/ws/"
                    + a.getBusinessLogicName() + "?wsdl";

            // URL url = new URL("http://localhost:9999/ws/ruralHouses?wsdl");
            URL url = null;
            try {
                url = new URL(serviceName);
            } catch (MalformedURLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

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

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

            BLFacade appFacadeInterface = service.getPort(BLFacade.class);
            return appFacadeInterface;
        }
```

Para implementar el patrón Simple Factory hemos creado una nueva clase llamada BLFacadeFactory. En ella hemos creado su propio constructor y el método createBLFacade(), en el cual hemos adaptado el código que usaba ApplicationLauncher. Dicho código decidía cuál de las 2 implementaciones utilizar para nuestra aplicación, acceder a un objeto de la lógica de negocio ubicado de forma local o acceder a un objeto web de la lógica de negocio a través de un servidor.

```java
package gui;

import java.awt.Color;

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());

        MainGUI a = new MainGUI();
        a.setVisible(false);

        MainUserGUI b = new MainUserGUI();
        b.setVisible(true);

        try {

            BLFacade appFacadeInterface;
//          UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsClassicLookAndFeel");
//          UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
            UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
            BLFacadeFactory factory = new BLFacadeFactory(c);
            appFacadeInterface = factory.createBLFacade();
            MainGUI.setBussinessLogic(appFacadeInterface);

        } catch (Exception e) {
            a.jLabelSelectOption.setText("Error: " + e.toString());
            a.jLabelSelectOption.setForeground(Color.RED);

            System.out.println("Error in ApplicationLauncher: " + e.toString());
        }
        // a.pack();

    }

}
```

En la clase ApplicationLauncher hemos utilizado el BLFacadeFactory anteriormente creado y así poder ejecutar nuestra aplicación bien de forma local o bien mediante un servidor.



2. Patrón Iterator

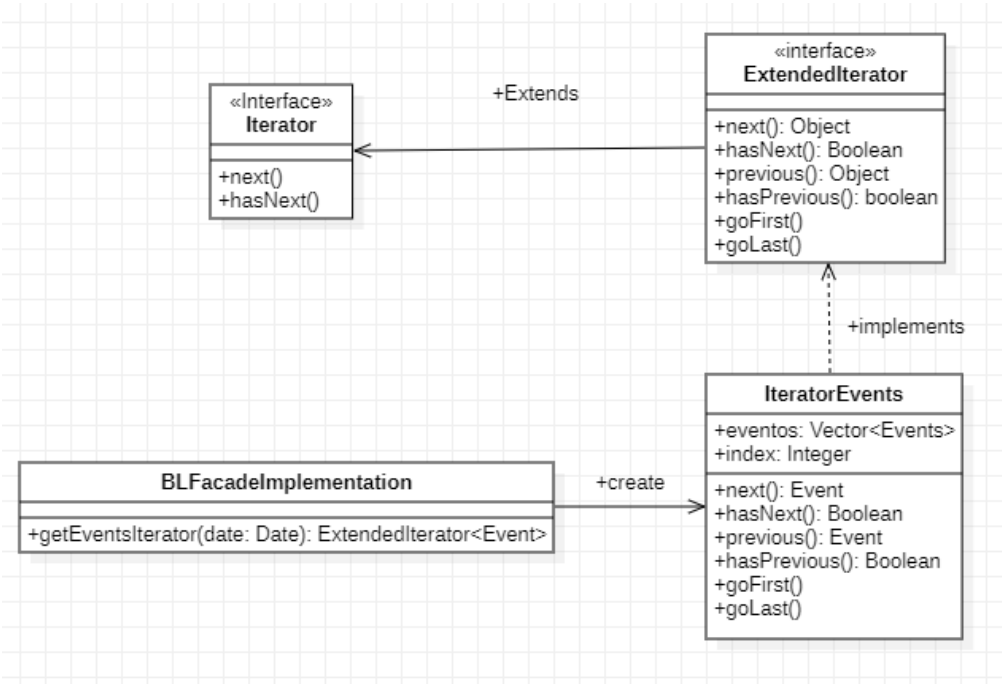
```java
public class IteratorEvents implements ExtendedIterator {
    private Vector<Event> eventos;
    private int index;

    public IteratorEvents(Vector<Event> eventos) {
        this.eventos = eventos;
        index = 0;
    }

    @Override
    public boolean hasNext() {
        // TODO Auto-generated method stub

        if (index < eventos.size()-1) {
            return true;
        }
        return false;
    }

    @Override
    public Object next() {
        // TODO Auto-generated method stub

        Event e = eventos.get(index);
        index++;
        return e;
    }

    @Override
    public Object previous() {
        // TODO Auto-generated method stub

        Event e = eventos.get(index);
        index--;
        return e;

    }

    @Override
    public boolean hasPrevious() {
        // TODO Auto-generated method stub

        if (index > 0 -1) {
            return true;
        }
        return false;
    }

    @Override
    public void goFirst() {
        // TODO Auto-generated method stub
        index = 0;

    }

    @Override
    public void goLast() {
        // TODO Auto-generated method stub
        index = eventos.size() - 1;
```

Para implementar el Patrón Iterator hemos creado una clase llamada IteratorEvents la cual implementa la interfaz ExtendedIterator que se nos muestra en el enunciado.

En la clase IteratorEvents hemos cruzado un constructor en el cual recoge como parámetro un vector de todos los eventos que ocurren en un cierto día.
También hemos completado los métodos que aparecen en la interfaz implementada para poder utilizarlos en el main.

```java
package Iterator;

import java.text.ParseException;

public class iteratorProba{

    public static void main(String[] args) {


        ConfigXML c = ConfigXML.getInstance();
        boolean isLocal =  c.isBusinessLogicLocal();
        BLFacadeFactory fakotori = new BLFacadeFactory(c);
        BLFacade blFacade = fakotori.createBLFacade();
        SimpleDateFormat    sdf =   new SimpleDateFormat("dd/MM/yyyy");
        Date date;
        try {

            date =  sdf.parse("17/12/2020"); // 17  del mes que viene
            ExtendedIterator i = blFacade.getEventsIterator(date);
            Event   e;
            System.out.println("____________________");
            System.out.println("RECORRIDO  HACIA  ATRÁS");
            i.goLast(); //  Hacia atrás
            while (i.hasPrevious()) {
                e = (Event) i.previous();
                System.out.println(e.toString());
            }
            System.out.println();
            System.out.println("____________________");
            System.out.println("RECORRIDO  HACIA  ADELANTE");
            i.goFirst();    //  Hacia adelante
            while (i.hasNext()) {
                e = (Event) i.next();
                System.out.println(e.toString());
            }
        }   catch (ParseException   e1) {
            System.out.println("Problems  with  date??  " + "17/12/2020");

        }
    }
}
```

La clase main simplemente la hemos adaptado para que se pueda utilizar con nuestro código creado anteriormente.

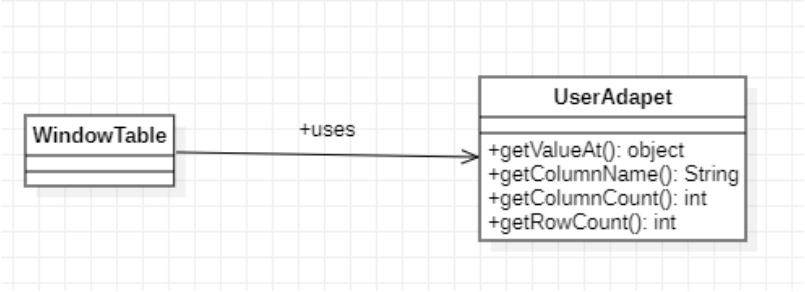


```
Opening DataAccess instance => isDatabaseLocal: true getDatabBaseOpenMode: initialize
>> DataAccess: getEvents
1;Atletico-Athletic
2;Eibar-Barcelona
3;Getafe-Celta
4;Alaves-Deportivo
5;Espanol-Villareal
6;Las Palmas-Sevilla
7;Malaga-Valencia
8;Girona-Leganes
9;Real Sociedad-Levante
10;Betis-Real Madrid
22;LA Lakers-Phoenix Suns
23;Atlanta Hawks-Houston Rockets
24;Miami Heat-Chicago Bulls
27;Djokovic-Federer
DataBase closed

_____________________
RECORRIDO       HACIA   ATRÁS
27;Djokovic-Federer
24;Miami Heat-Chicago Bulls
23;Atlanta Hawks-Houston Rockets
22;LA Lakers-Phoenix Suns
10;Betis-Real Madrid
9;Real Sociedad-Levante
8;Girona-Leganes
7;Malaga-Valencia
6;Las Palmas-Sevilla
5;Espanol-Villareal
4;Alaves-Deportivo
3;Getafe-Celta
2;Eibar-Barcelona
1;Atletico-Athletic


_____________________
RECORRIDO       HACIA   ADELANTE
1;Atletico-Athletic
2;Eibar-Barcelona
3;Getafe-Celta
4;Alaves-Deportivo
5;Espanol-Villareal
6;Las Palmas-Sevilla
7;Malaga-Valencia
8;Girona-Leganes
9;Real Sociedad-Levante
10;Betis-Real Madrid
22;LA Lakers-Phoenix Suns
23;Atlanta Hawks-Houston Rockets
24;Miami Heat-Chicago Bulls
```

3. Patrón Adapter

```java
public class UserAdapter extends AbstractTableModel {

    private final List<ApustuAnitza> apustuak;
    private Registered register;
    private String[] colNames = new String[] { "Event", "Question", "Event Date", "Bet (€)" };

    public UserAdapter(Registered r) {
        // copy the HashMap data to a sequential data structure
        apustuak = new ArrayList<ApustuAnitza>(r.getApustuAnitzak());
        this.register = r;
    }

    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        Vector<Apustua> bektorea = new Vector<Apustua>();
        for (int i = 0; i < register.getApustuAnitzak().size(); i++) {
            for (int j = 0; j < register.getApustuAnitzak().get(i).getApustuak().size(); j++)
                bektorea.add(register.getApustuAnitzak().get(i).getApustuak().get(j));
        }
        switch (columnIndex) {
        case 0:
            return ((Object) bektorea.get(rowIndex).getKuota().getQuestion().getEvent());
        case 1:
            return ((Object) bektorea.get(rowIndex).getKuota().getQuestion());
        case 2:
            return ((Object) bektorea.get(rowIndex).getKuota().getQuestion().getEvent().getEventDate());
        case 3:
            return ((Object) bektorea.get(rowIndex).getApustuAnitza().getBalioa());
        }
        return null;
    }

    @Override
    public String getColumnName(int col) {
        return colNames[col];
    }

    @Override
    public int getColumnCount() {
        return 4;
    }

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

Para finalizar, en este caso el Patrón Adapter lo hemos implementado creando la clase UserAdapter. En esta, hemos asignado los nombres de las columnas a la tabla y hemos creado un constructor en el usuario registrado como parámetro y guardando sus apuestas. También hemos creado distintos métodos para modificar y consultar la tabla. Entre ellos getValueAt() el cual nos servirá para consultar cualquier parámetro de la tabla indicando la fila y columna correspondientes.



https://github.com/JonDorronsoroo/bets