Proyecto Bets22

Patrones de Diseño

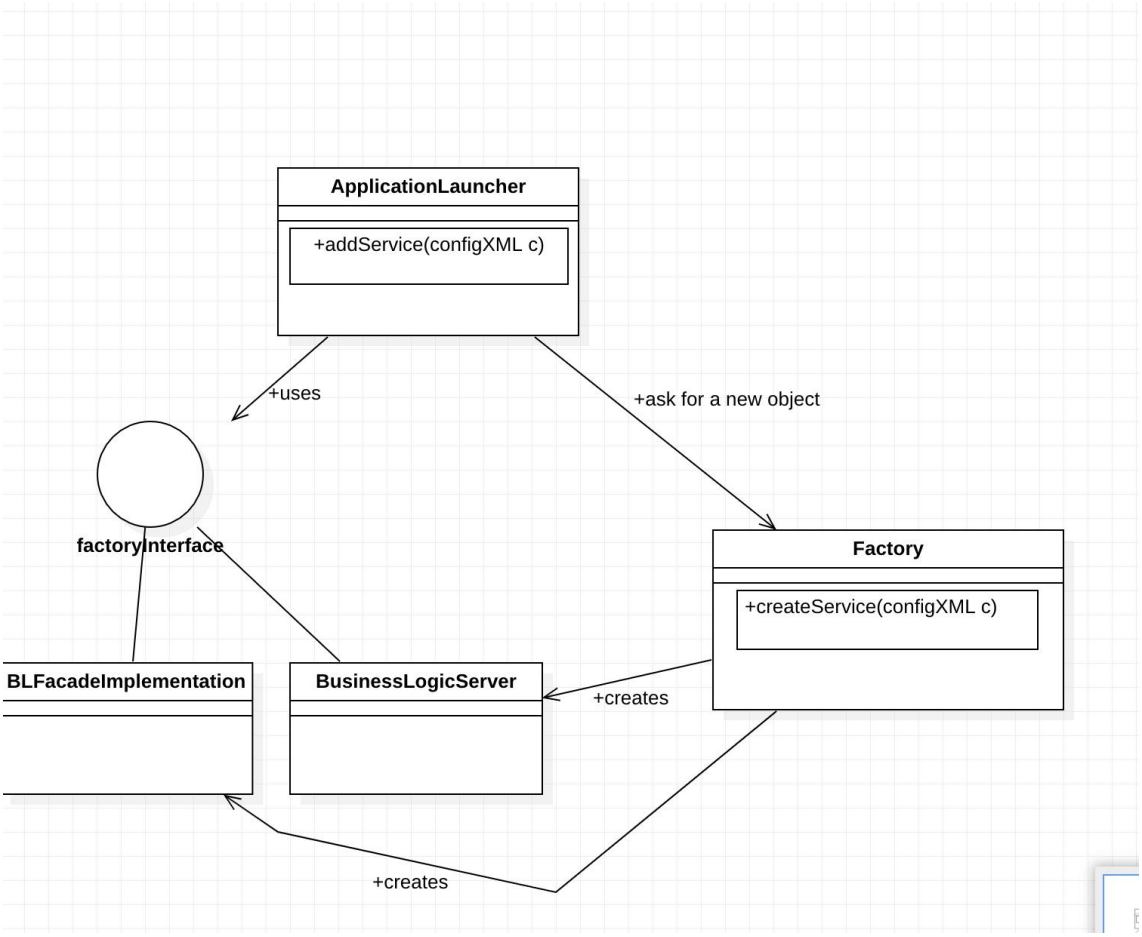
UPV-EHU Ingeniería del Software II

Anal Lucía Durán Lengo

Jon Ortega Goikoetxea

Patrón Factory Method

Diagrama UML:



El patrón Simple Factory es un patrón de diseño que sirve para crear objetos sin tener que especificar su clase exacta, es decir, que un objeto creado puede modificarse con facilidad. Para implementar el método, tenemos el Creator, que en este caso es la clase Factory, el Product que es la Interfaz creada: factoryInterfaz y el ConcreteProduct, que son las clases BLFacadeImplementation y BusinessLogicServer. El cliente en este caso es la clase ApplicationLauncher.

En la interfaz factoryInterfaz creamos services, e implementando esta misma interfaz en la clase factory, realizamos la prueba de si un servicio es local o remoto. Tras ello, con ayuda de un try catch, intentamos crear un service, y si algo falla, salta al catch.

Por último, en la clase ApplicationLauncher, omitimos el código utilizado en la clase factory y añadimos dentro del try una nueva instancia de factory. Tras ello, hemos aplicado el patrón Simple Factory.

Código:

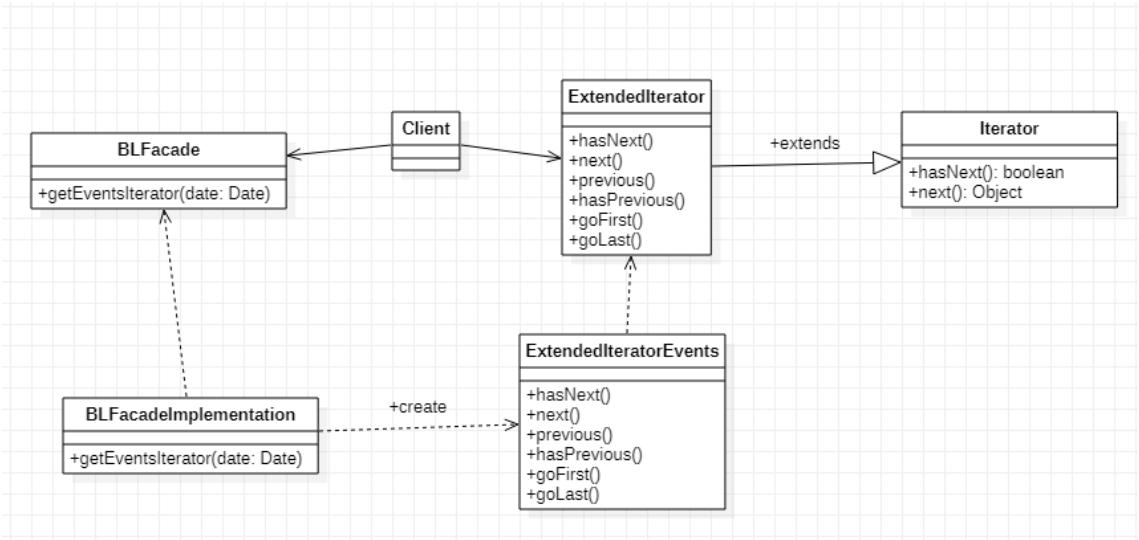
```
public class factory implements factoryInterface{

    @Override
    public  BLFacade services(ConfigXML c) {


    if (c.isBusinessLogicLocal()) {

        //In this option the DataAccess is created by FacadeImplementationWS
        //appFacadeInterface=new BLFacadeImplementation();

        //In this option, you can parameterize the DataAccess (e.g. a Mock DataAccess object)
        System.out.println("aqui probando que local");
        DataAccess da= new DataAccess(c.getDataBaseOpenMode().equals("initialize"));
        return new BLFacadeImplementation(da);


    } else { //If remote

        System.out.println("aqui probando que service sofroror");
        String serviceName= "http://"+c.getBusinessLogicNode() +":"+ c.getBusinessLogicPort()+"/ws/"+c.getBusinessLogicName()+"?wsdl";

        //URL url = new URL("http://localhost:9999/ws/ruralHouses?wsdl");
        URL url;
        try {
            url = new URL(serviceName);
            QName qname = new QName("http://businessLogic/", "BLFacadeImplementationService");
            Service service = Service.create(url, qname);
            return service.getPort(BLFacade.class );
        } catch (MalformedURLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            return null;
        }


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


    }
```

```
package Factory;

import businessLogic.BLFacade;
import configuration.ConfigXML;

public interface factoryInterface {


    BLFacade services(ConfigXML c);
}
```

```java
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 {
            factoryInterface fac = new factory();
            BLFacade appFacadeInterface = fac.services(c);
//          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");

            //if (c.isBusinessLogicLocal()) {
```

Patrón Iterator

Diagrama UML:



El patrón iterator permite recorrer una estructura de datos sin que sea necesario conocer la estructura interna de la misma. Es muy útil cuando estamos trabajando con estructuras de datos complejas.

El patrón nos permite recorrer sus elementos mediante un Iterador. El Iterador es una interfaz que proporciona los métodos necesarios para recorrer los elementos de la estructura de datos.

Código:

```java
public class mainIterator {

    public static void main(String[] args) {
        int isLocal = 1;
        BLFacade blFacade = new BLFacadeImplementation();
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        Date date;
        try {
            date = sdf.parse("17/12/2022");
            ExtendedIterator<Event> i = blFacade.getEventsIterator(date);
            Event e;
            System.out.println("____________________");
            System.out.println("RECORRIDO  HACIA  ATRÁS");
            i.goLast(); // Hacia atrás
            while (i.hasPrevious()) {
                e = 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 = i.next();
                System.out.println(e.toString());
            }
        } catch (ParseException e1) {
            System.out.println("Problems  with  date??  " + "17/12/2020");
        }

    }

}
```

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

    //Devuelve el elemento actual y va haca atras
    public Object previous();

    //true  si tiene un elemento anterior
    public boolean hasPrevious();

    //Se coloca en el primer elemento
    public void goFirst();

    //Se coloca en el último elemento
    public void goLast();
}
```

```java
public class ExtendedIteratorEvents implements ExtendedIterator {

    private Vector<Event> events;
    private int i;

    public ExtendedIteratorEvents(Vector<Event> events) {
        this.events = events;
        i = 0;
    }

    @Override
    public boolean hasNext() {
        try {
            if(events.get(i) != null)
                return true;
            return false;
        }catch(Exception e) {
            return false;
        }
    }

    @Override
    public Object next() {
        i++;
        return events.get(i-1);
    }

    @Override
    public Object previous() {
        i--;
        return events.get(i+1);
    }
```

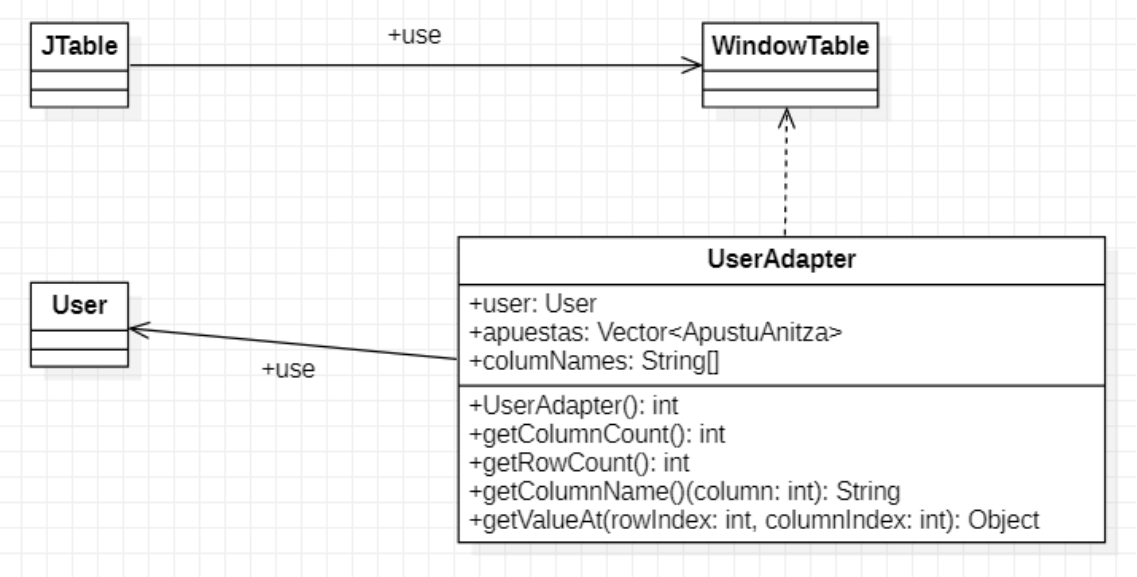
```java
    @Override
    public boolean hasPrevious() {
        return hasNext();
    }

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

    @Override
    public void goLast() {
        i = events.size()-1;
    }


}
```

En la clase BLFacade:

```java
@WebMethod public ExtendedIterator<Event> getEventsIterator(Date date);
```

Patrón Adapter

Diagrama UML:



El patrón Adapter se utiliza cuando tenemos interfaces de software incompatibles, las cuales a pesar de su incompatibilidad tiene una funcionalidad bastante similar.

Este patrón es implementado cuando se desea homogeneizar la forma de trabajar con estas interfaces incompatibles, para lo cual se crea una clase intermedia que funciona como un adaptador. Esta clase adaptador proporcionará los métodos para interactuar con la interfaz no compatible.

Código:

```java
package adapter;

import java.util.Vector;

public class UserAdapter extends AbstractTableModel {
    private User user;
    private Vector<ApustuAnitza> apuestas;
    private String[] columNames = new String[] {"Event", "Question", "Event Date", "Bet (€)"};

    public UserAdapter(User user) {
        apuestas = ((Registered) user).getApustuAnitzak();
        this.user = user;
    }

    public int getColumnCount() {
        return 4;
    }

    public int getRowCount() {
        return apuestas.size();
    }


    public String getColumnName(int colum) {
        return columNames[colum];
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        switch(columnIndex) {
            case 0: return (Object)apuestas.get(rowIndex).getApustuak();
            case 1: return (Object)apuestas.get(rowIndex);
            case 2: return (Object)apuestas.get(rowIndex).getData();
            case 3: return (Object)apuestas.get(rowIndex).getBalioa();
        }
        return null;
    }
}
```

```java
public class WindowTable extends JFrame{
    private User user;
    private JTable  tabla;
    public WindowTable(User user){
        super("Apuestas realizadas  por "+  user.getUsername()+":");
        this.setBounds(100, 100,    700,    200);
        this.user = user;
        UserAdapter adapt   =   new UserAdapter(user);
        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);
    }
}
```

```
JButton btnNewButton_1 = new JButton();
btnNewButton_1.setBackground(new Color(255, 128, 128));
btnNewButton_1.setFont(new Font("Tahoma", Font.PLAIN, 16));
btnNewButton_1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JFrame wt = new WindowTable(user);
        wt.setVisible(true);
    }
});
btnNewButton_1.setBounds(10, 391, 282, 68);
jContentPane.add(btnNewButton_1);
```

Ejecución:

No ha sido posible ejecutar el código, ya que al añadir la clase User al proyecto, cuando se ejecuta la aplicación, salta el siguiente error:

```
Exception in thread "AWT-EventQueue-0"
com.objectdb.o.UserException: Too many persistable types (>10) -
exceeds evaluation limit
```

Se ha intentado solucionarlo sin mucho éxito.

Repositorio GitHub:

https://github.com/anadurlen/IS2BetsBueno