Simple factory

Iterator

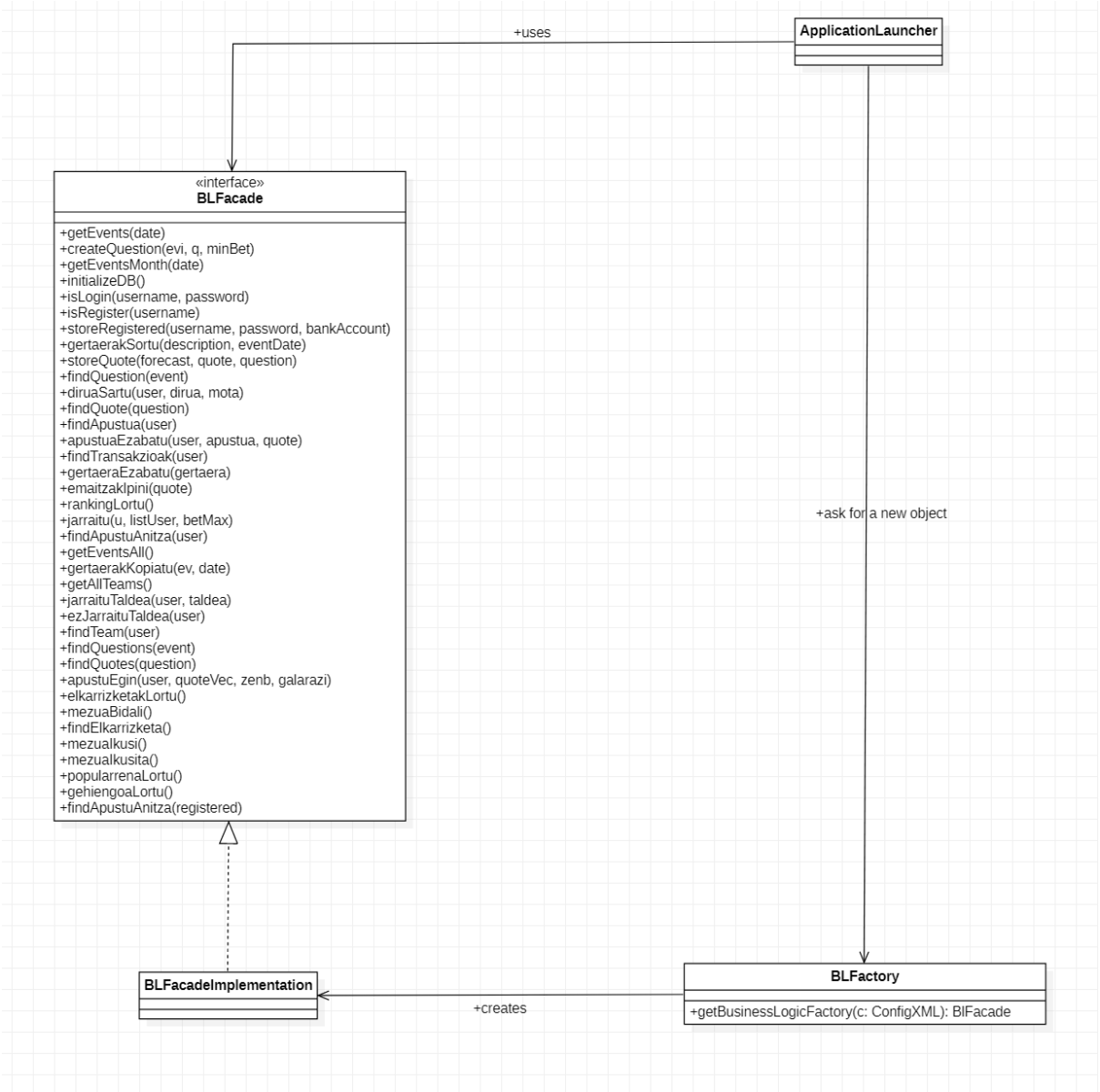
Adapter

Github

https://github.com/leirecael/BetsLiteIS2.git
Puedes encontrar en la raíz del proyecto un UML actualizado.

Simple factory

El cliente es el applicationLauncher, la fábrica es el BLFactory, el producto concreto es el BLFacadeImplementation y el producto es el BLFacade. A continuación está el UML extendido:



A continuación está el código modificado:

```
public class BLFactory {

	/**
	 * Method returns implementation of BLFacade. If is local, a DataAccess local data base will be implicitly created.
	 * @param c The ConfigXML file.
	 * @return The implementation of BLFacade.
	 * @throws MalformedURLException When reading the remote URL.
	 */
	public BLFacade getBusinessLogicFactory(ConfigXML c) throws MalformedURLException {
		BLFacade result;
		if (c.isBusinessLogicLocal()) {
			DataAccess da = new DataAccess(c.getDataBaseOpenMode().equals("initialize"));
			result = 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);
		result = service.getPort(BLFacade.class);
		}
		return result;
	}
}
```

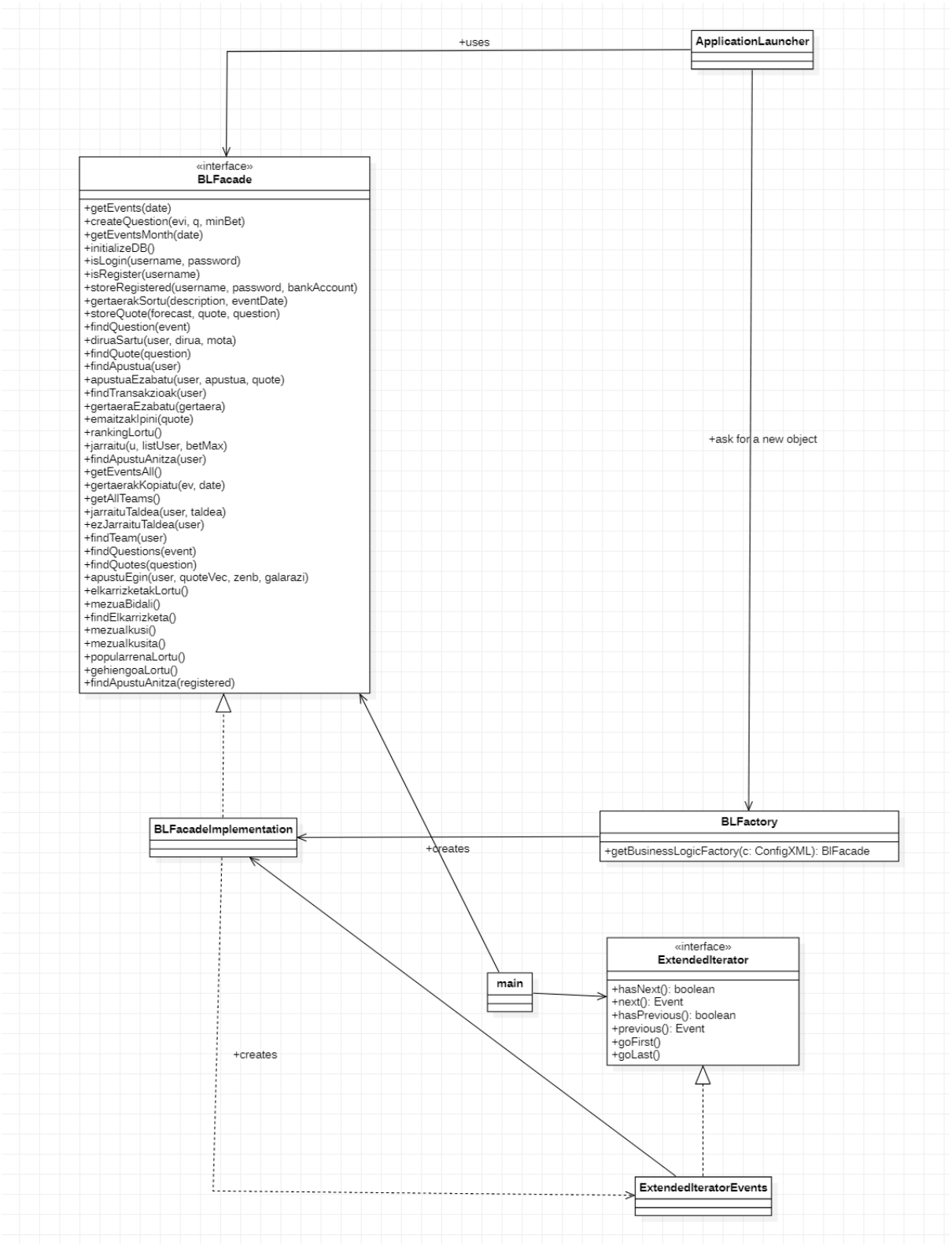
Hemos creado la clase BLFactory que es la fábrica que crea el producto BLFacade que nos piden en el application launcher.

```
try {

		BLFacade appFacadeInterface = new BLFactory().getBusinessLogicFactory(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.getDataBaseOpenMode().equals("initialize"))
			appFacadeInterface.initializeBD();
			*/
		MainGUI.setBussinessLogic(appFacadeInterface);
```

El application launcher hemos cambiado el código para que directamente pida la creación del producto a la fábrica mencionada anteriormente. Se le pasa como parámetro el ConfigXML.

Iterator

A continuación está el UML extendido, hemos creado la interfaz de ExtendedIterators junto con su implementación. También hemos creado una nueva clase main que se utiliza para comprobar el funcionamiento del ExtendedIterator:



A continuación está el código modificado:

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

        // true if there 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();
}
```

Hemos creado la interfaz de ExtendedIterator.

```java
public class ExtendedIteratorEvents implements ExtendedIterator<Event>{
	private List<Event> events;
	private int index;

	public ExtendedIteratorEvents(List<Event> l) {
		events = l;
		goFirst();
	}

	@Override
	public Event previous() {
		Event e = events.get(index);
		index--;
		return e;
	}

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

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

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

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

	@Override
	public Event next() {
		Event e = events.get(index);
		index++;
		return e;
	}

}
```

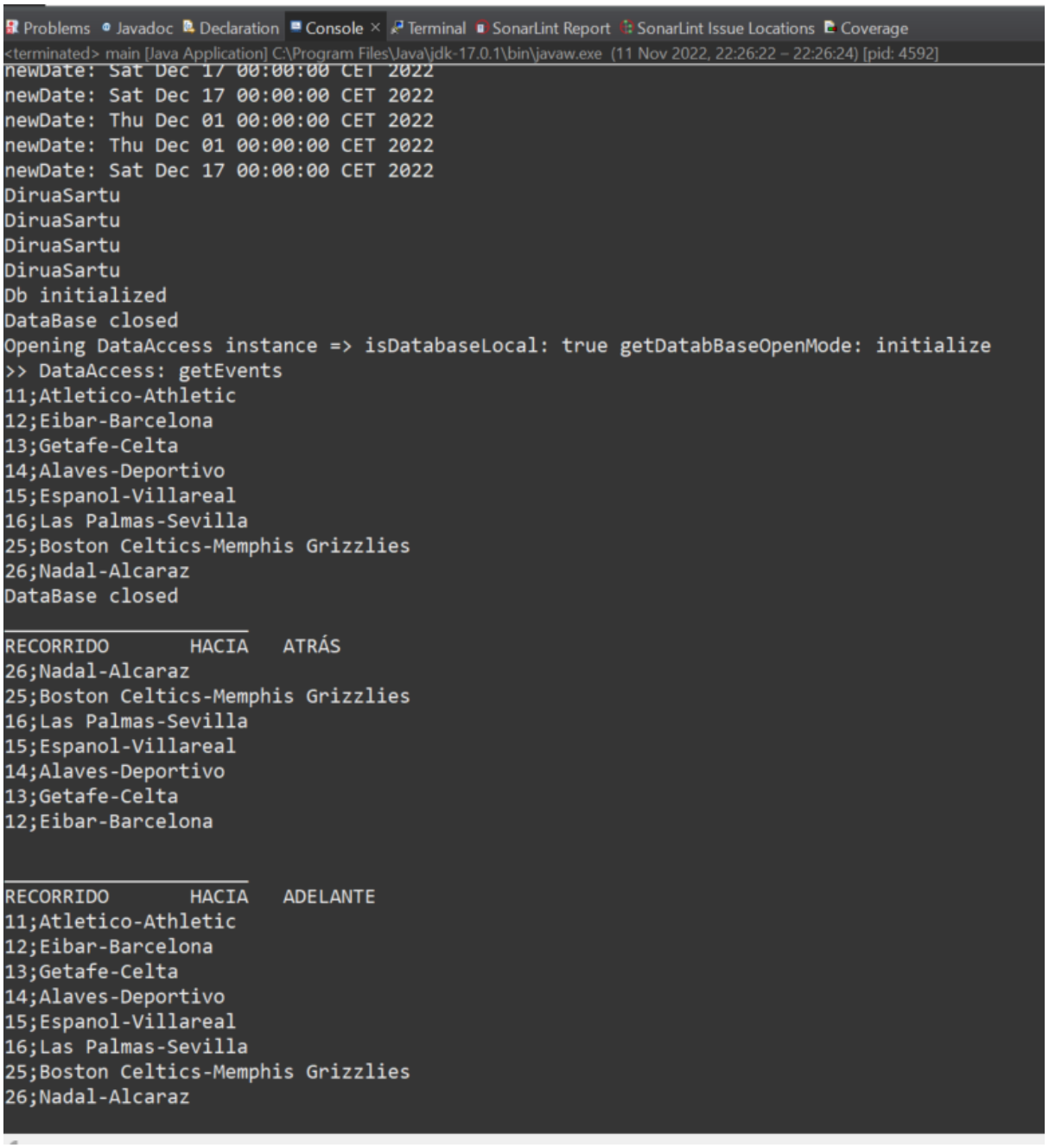
Aquí está la implementación de la interfaz anterior.

```java
public ExtendedIterator<Event> getEventsIterator(Date date) {
        return new ExtendedIteratorEvents(getEvents(date));
}
```

En BLFacadeImplementation hemos implementado el método getEventsIterator que hemos añadido en BLFacade y hemos reutilizado el código del método getEvents que usa un Vector.

```java
public class main {
	public static void main(String[] args) {
//		obtener el	objeto Facade	local
		int isLocal = 1;
		ConfigXML c = ConfigXML.getInstance();
		BLFacade blFacade;

		SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
		Date date;

		try {
			blFacade = new BLFactory().getBusinessLogicFactory(c);
			try {

				date = sdf.parse("01/12/2022"); // 17 del mes que viene
				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");
			}

		} catch (MalformedURLException e2) {
			e2.printStackTrace();
		}
	}
}
```

Por último hemos creado una clase main para probar el iterator. A continuación está la ejecución del programa:

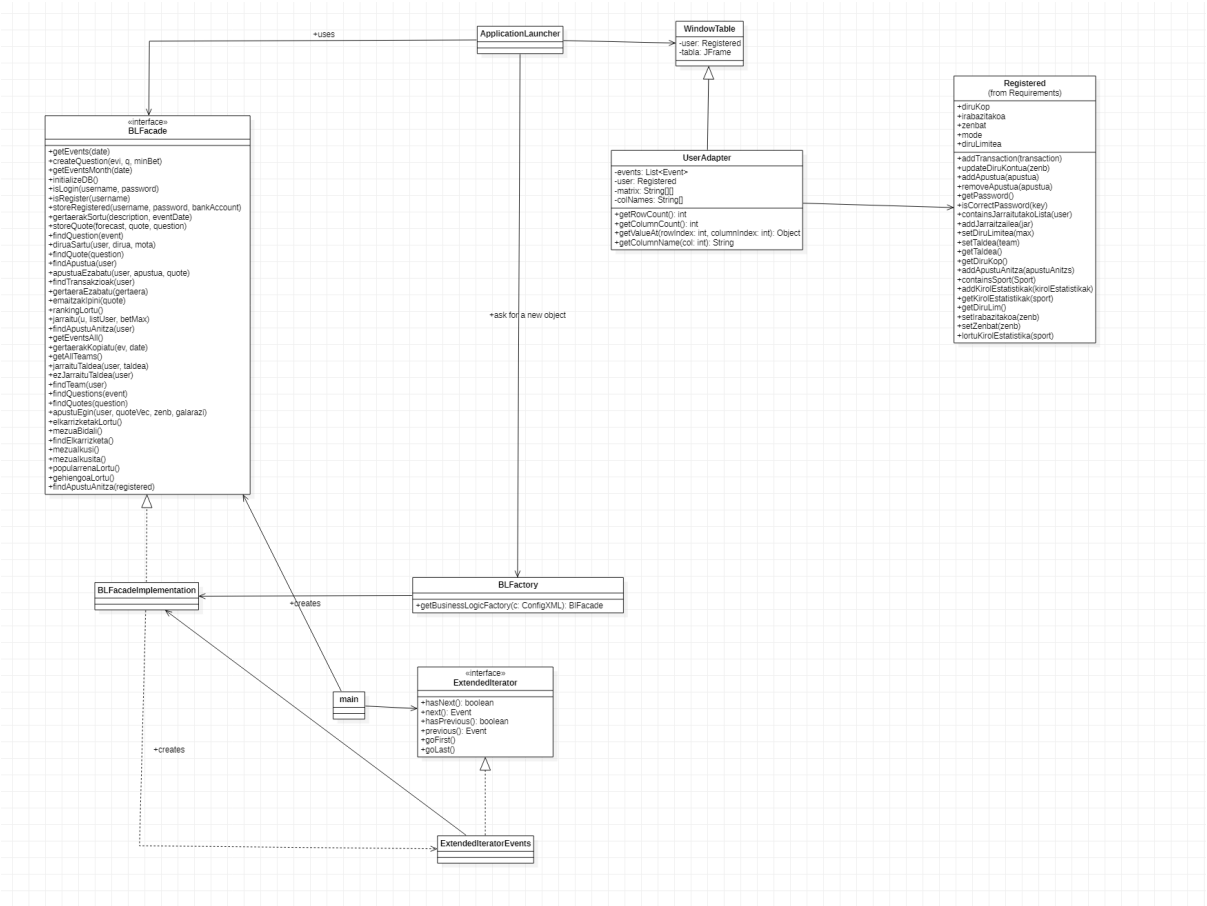

Problems Javadoc Declaration Console Terminal SonarLint Report SonarLint Issue Locations Coverage
<terminated> main [Java Application] C:\Program Files\Java\jdk-17.0.1\bin\javaw.exe (11 Nov 2022, 22:26:22 – 22:26:24) [pid: 4592]
newDate: Sat Dec 17 00:00:00 CET 2022
newDate: Sat Dec 17 00:00:00 CET 2022
newDate: Thu Dec 01 00:00:00 CET 2022
newDate: Thu Dec 01 00:00:00 CET 2022
newDate: Sat Dec 17 00:00:00 CET 2022
DiruaSartu
DiruaSartu
DiruaSartu
DiruaSartu
Db initialized
DataBase closed
Opening DataAccess instance => isDatabaseLocal: true getDatabBaseOpenMode: initialize
>> DataAccess: getEvents
11;Atletico-Athletic
12;Eibar-Barcelona
13;Getafe-Celta
14;Alaves-Deportivo
15;Espanol-Villareal
16;Las Palmas-Sevilla
25;Boston Celtics-Memphis Grizzlies
26;Nadal-Alcaraz
DataBase closed
RECORRIDO HACIA ATRÁS
26;Nadal-Alcaraz
25;Boston Celtics-Memphis Grizzlies
16;Las Palmas-Sevilla
15;Espanol-Villareal
14;Alaves-Deportivo
13;Getafe-Celta
12;Eibar-Barcelona
RECORRIDO HACIA ADELANTE
11;Atletico-Athletic
12;Eibar-Barcelona
13;Getafe-Celta
14;Alaves-Deportivo
15;Espanol-Villareal
16;Las Palmas-Sevilla
25;Boston Celtics-Memphis Grizzlies
26;Nadal-Alcaraz

Adapter

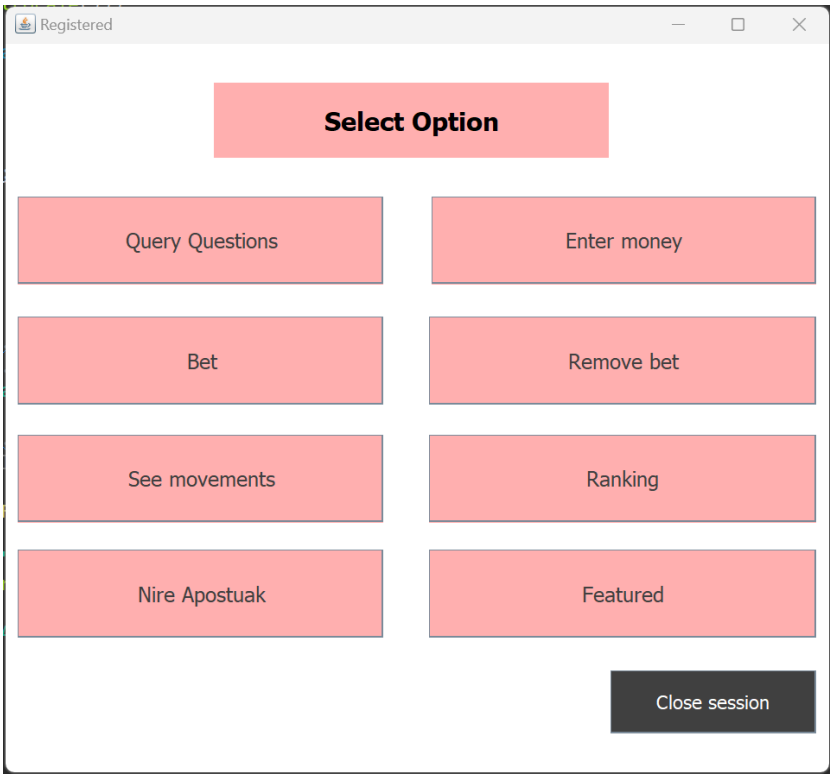
La clase intermediaria que hará la función de adaptadora entre “WindowTable” y “Registered” es “UserAdapter”.

De esta manera la clase que contiene la interfaz de usuario; la cual tiene la tabla que contiene los datos de las apuestas realizadas por un usuario concreto, no tiene que preocuparse por la estructura que esos datos representan en la clase “Registered” ni en todas las demás que también se han de consultar, como pueden ser: “Event”, “Quote”, “ApustuAnitza”, “Question” …

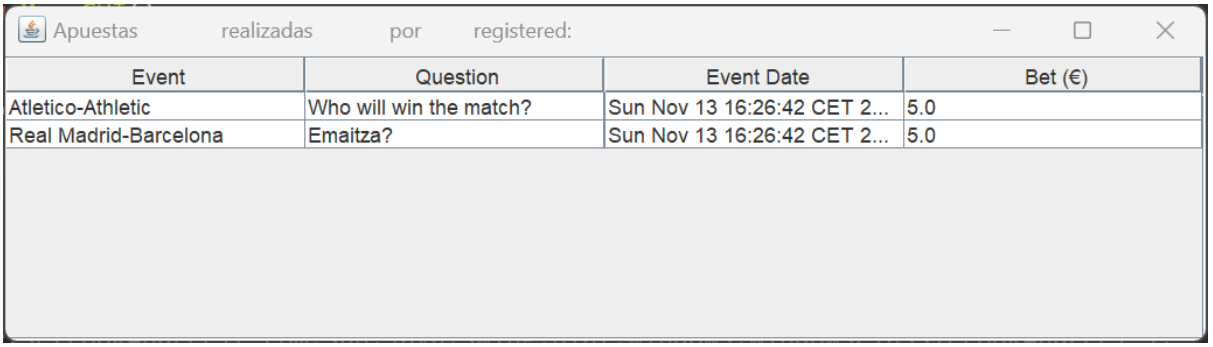
Así, en el siguiente diagrama podemos ver el patrón Adapter en la relación entre las clases “Aplication Launcher”; que hace la función de cliente, la clase “WindowTable”; que es el target dónde se encuentra la tabla UI, la clase “UserAdapter”; que representa la clase adaptadora, y la clase Register; que ejerce la función de Adaptee, ya que contiene los datos a representar en la tabla de “WindowTable” (junto con las demás clases que forman una apuesta y contienen la información de la misma).



En la siguiente imagen se puede ver el botón que se ha añadido para que el usuario registrado pueda acceder a la tabla que contiene la información sobre sus apuestas, la clase “WindowTable”.



Este es el resultado final para el usuario registered, dónde le aparecen todos sus resultados de apuestas gracias al patrón Adapter:



```java
package domain;

import java.util.List;

public class UserAdapter extends AbstractTableModel{
    private List<Event> events;
    private Registered user;
    private String[][] matrix;
    private String[] colNames = { "Event", "Question", "Event Date", "Bet (€)" };

    public UserAdapter(Registered user) {
        this.user = user;
        matrix = new String[getRowCount()][getColumnCount()];
        int i = 0;

        System.out.println(getRowCount() + " " + getColumnCount());

        for(ApustuAnitza aA : user.getApustuAnitzak()) {
            for(Apustua a: aA.getApustuak()) {
                matrix[i][0] = a.getKuota().getQuestion().getEvent().getDescription();
                matrix[i][1] = a.getKuota().getQuestion().getQuestion();
                matrix[i][2] = aA.getData().toString();
                matrix[i][3] = aA.getBalioa().toString();
                i++;
            }
        }
    }

    @Override
    public int getRowCount() {
        int result = 0;
        for(ApustuAnitza aA : user.getApustuAnitzak()) {
            for(Apustua a: aA.getApustuak()) {
                result++;
            }
        }
        return result;
    }

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

    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        return matrix[rowIndex][columnIndex];
    }

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

}
```

En la imagen anterior se puede ver la clase “UserAdapter”, clase adaptadora del patrón Adapter para la UI “nire apustuak”.

En la clase constructora se inicializa la matriz que va a almacenar los datos de las apuestas de un usuario.

El resto de clases se deben implementar ya que se extiende la clase AbstractTableModel para que el Target pueda pedir (request) la información que va a ser mostrada en la tabla de apuestas.

Nótese que también hay un atributo llamado “colNames” que contiene los nombres que tomarán las columnas de la tabla de apuestas. La clase target podrá obtener dicha información utilizando la función getColumnName, dónde se puede pasar por parámetro el número de columna y obtener el nombre de esa columna.