Patrones de diseño

“Patrón Observer”

Empeño

Define una dependencia uno-a-varios entre objetos de manera que cuando un objeto cambia de estado, se notifica a todas sus dependencias y se actualizan automáticamente.

Básicamente debemos saber que el patrón **Observer** es un patrón de comportamiento.

También conocido como dependents, publish-subscribe

¿Cuando es aplicable?

Cuando un objeto cambia y esto requiere el cambio de varios objetos y se desconoce el número de objetos que necesitarán este cambio

Cuando una abstracción tiene dos aspectos, una dependiente de la otra, té permite variarlos y reusarlos independientemente

Cuando un objeto tiene que notificar a otros objetos sin hacer asunciones sobre su naturaleza.

Participantes

Subject (conoce sus observers que pueden ser uno, ninguno o varios y proporciona un interfaz para añadir y quitar objetos observadores)

Observer (define un interfaz para actualizar que debe ser llamado cuando el subject cambia de estado)

ConcreteSubject (almacena el estado de interés para los observadores y les envía notificaciones cuando su estado cambia)

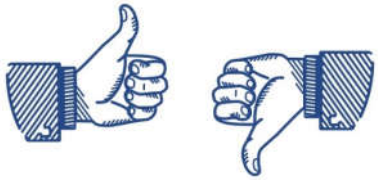
ConcreteObserver (mantiene una referencia a un ConcreteSubject almacena el estado del sujeto que le resulta de interesante implementa la interfaz de Observer para mantener su estado consistente con el del sujeto)

Ventajas y desventas

Al *sujeto* no le interesa los efectos o desenlaces de los **observadores**, él simplemente emite. El resultado es código reusable y flexible .

Desventaja importante, cuando un **observador** es demasiado grande. Eso puede traer conllevar daños en el uso de memoria.

Otra desventaja seria que si la implementación no es limpia, será muy difícil encontrar errores cuando sucedan.



Colaboraciones

El objeto observado notifica a sus **observadores** cada vez que ocurre un cambio.

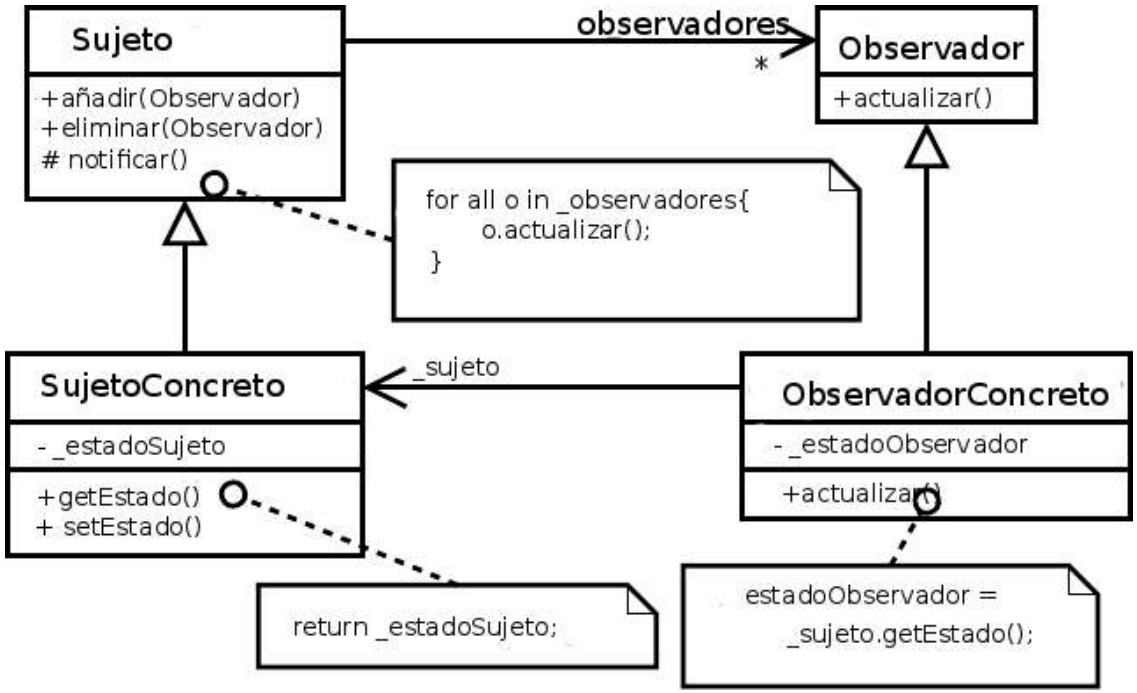
Después de ser informado de un cambio en el objeto observado, cada observador concreto puede pedirle la información que necesita para reconciliar su estado.

Relaciones

Mediador: Define un objeto que encapsula cómo interactúa un conjunto de objetos.
El mediador promueve el acoplamiento flexible evitando que los objetos se refieran entre sí de forma explícita, y le permite variar su interacción de forma independiente

Siglenton :Consiste en garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella

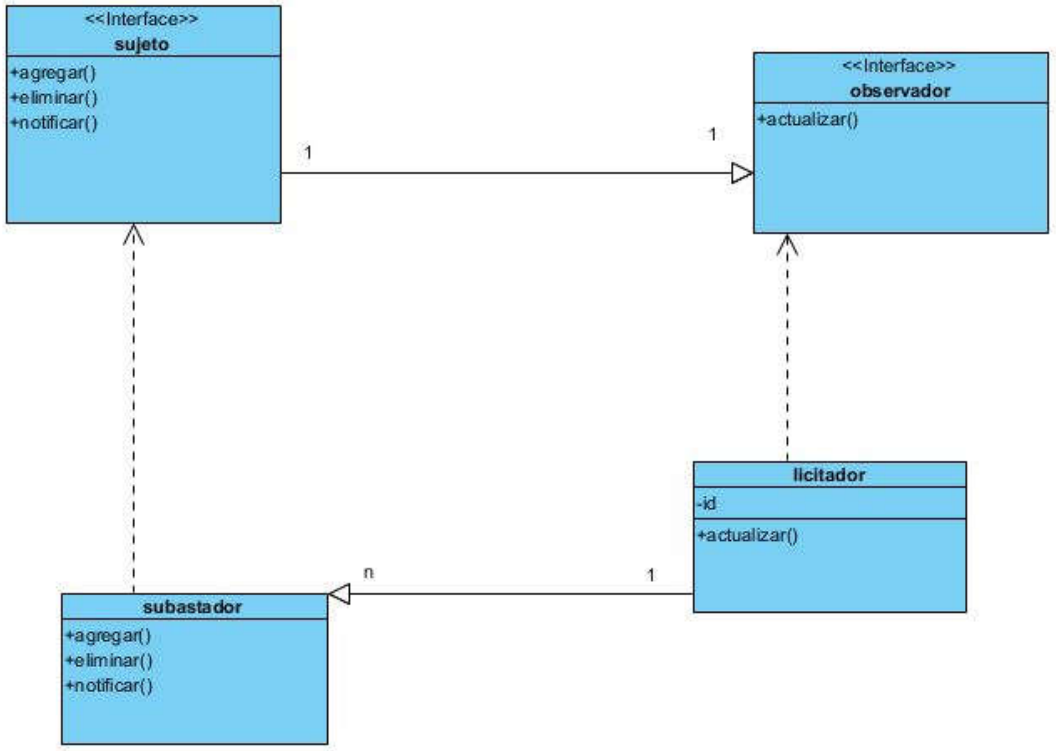
Estructura


Sujeto
+añadir(Observador)
+eliminar(Observador)
notificar()
observadores
*
Observador
+actualizar()
for all o in _observadores{
o.actualizar();
}
SujetoConcreto
- _estadoSujeto
+getEstado()
+ setEstado()
_sujeto
ObservadorConcreto
- _estadoObservador
+actualizar()
return _estadoSujeto;
estadoObservador =
_sujeto.getEstado();

Ejemplo

Mi practica como ejemplo será una empresa publica que saca una licitación para que en el concurso pujen otras empresas privadas.

Diagrama clases visión rápida…..



Estructura Java

CLASE
Licitador.java

```java
package codigo_patron;


import codigo_patron.patron.Evento;

/**
 * Jorge Corralo
 */
public class Licitador implements Observador {

    private static int ID = 0;

    @Override
    public void actualizar(Evento event) {
        System.out.println(
                "Identificador: " + (++ID) +
                        ", Evento actualizado: " +  event.getTipo() +
                        ", Descripciónn evento: " + event.getDescripcion());
    }
}

```

CLASE
PuebaSubasta.java

```java
package codigo_patron;


import codigo_patron.patron.Evento;

/**
 * Jorge Corralo
 */
public class PruebaSubasta {

    public static void main(String[] args) {

        Sujeto subastador = new Subastador();

        Observador licitador1 = new Licitador();
        Observador licitador2 = new Licitador();
        Observador licitador3 = new Licitador();

        subastador.agregar(0, licitador1);
        subastador.agregar(0, licitador2);
        subastador.agregar(0, licitador3);

        subastador.agregar(1, licitador3);

        Evento alta = new Evento(0, "Oferta alta");
        Evento baja = new Evento(1, "Oferta baja");

        subastador.notificar(0, alta);
        subastador.notificar(1, baja);

    }
}
```

CLASE
Subastador.java

```java
package codigo_patron;

import codigo_patron.patron.Evento;
import codigo_patron.patron.Observador;
import codigo_patron.patron.Sujeto;

import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;


public class Subastador extends Sujeto{

    private final HashMap<Integer, LinkedList<Observador>> observers;

    public Subastador(){
        observers = new HashMap<Integer, LinkedList<Observador>>();
    }

    private LinkedList<Observador> getList(int type) {
        if (!observers.containsKey(type)) {
            observers.put(type, new LinkedList<Observador>());
        }
        return observers.get(type);
    }

    @Override
    public void agregar(int eventTpye, Observador newObserver) {
        getList(eventTpye).add(newObserver);
    }

    @Override
    public void eliminar(int eventTpye, Observador observer) {
        getList(eventTpye).remove(observer);
    }

    @Override
    public void notificar(int eventTpye, Evento event) {
        if (observers.containsKey(eventTpye)){
            Iterator<Observador> iterator = observers.get(eventTpye).iterator();
            while(iterator.hasNext()){
                iterator.next().actualizar(event);
            }
        }
    }

}
```

CLASE
Evento.java

```java
package codigo_patron.patron;

import java.text.DateFormat;


public class Evento {

    private int tipo;
    private String descripcion;
    private Date fecha;

    public Evento(){}

    public Evento(int tipo, String descripcion){
        this.setTipo(tipo);
        this.setDescripcion(descripcion);
        this.fecha = new Date();
    }

    public int getTipo() {
        return tipo;
    }

    public void setTipo(int tipo) {
        this.tipo = tipo;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

    public String getFecha() {
        DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
        return dateFormat.format(fecha);
    }
}
```

INTERFACE
Obervador.java

```java
package codigo_patron.patron;


public interface Observador {
    public void actualizar(Evento event);
}

```

CLASE
ObservadorConcreto.java

```java
package codigo_patron.patron;


public class ObservadorConcreto implements Observador {

    private static int ID = 0;

    @Override
    public void actualizar(Evento event) {
        System.out.println(
                "Identificador: " + (++ID) +
                    ", Actualizando evento: " +  event.getTipo() +
                        ", Descripción evento: " + event.getDescripcion());
    }
}

```

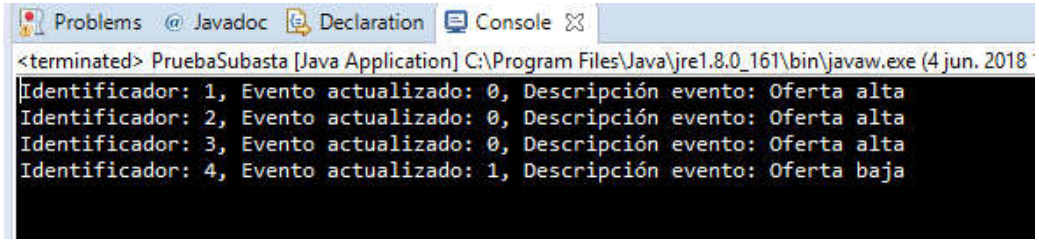
CLASE
(abstracta)
Sujeto.java

```java
package codigo_patron.patron;


public abstract class Sujeto {

    public abstract void agregar(int eventTpye, Observador observer);
    public abstract void eliminar(int eventTpye, Observador observer);
    public abstract void notificar(int eventTpye, Evento event);

}

```

CLASE
SujetoConcreto.java

```java
package codigo_patron.patron;

import java.util.HashMap;


public class SujetoConcreto extends Sujeto {

    private final HashMap<Integer, LinkedList<Observador>> observadores;

    public SujetoConcreto(){
        observadores = new HashMap();
    }

    private LinkedList<Observador> getList(int type) {
        if (!observadores.containsKey(type)) {
            observadores.put(type, new LinkedList<Observador>());
        }
        return observadores.get(type);
    }

    @Override
    public void agregar(int eventTpye, Observador newObserver) {
        getList(eventTpye).add(newObserver);
    }

    @Override
    public void eliminar(int eventTpye, Observador observer) {
        getList(eventTpye).remove(observer);
    }

    @Override
    public void notificar(int eventTpye, Evento event) {
        if (observadores.containsKey(eventTpye)){
            Iterator<Observador> iterator = observadores.get(eventTpye).iterator();
            while(iterator.hasNext()){
                iterator.next().actualizar(event);
            }
        }
    }

}
```

SALIDA


Problems
Javadoc
Declaration
Console
<terminated> PruebaSubasta [Java Application] C:\Program Files\Java\jre1.8.0_161\bin\javaw.exe (4 jun. 2018
Identificador: 1, Evento actualizado: 0, Descripción evento: Oferta alta
Identificador: 2, Evento actualizado: 0, Descripción evento: Oferta alta
Identificador: 3, Evento actualizado: 0, Descripción evento: Oferta alta
Identificador: 4, Evento actualizado: 1, Descripción evento: Oferta baja