TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE TIJUANA

SUBDIRECCIÓN ACADÉMICA

DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN

Tarea

Patrón observer

Docente
José de Jesús Parra Galaviz

SEMESTRE agosto – diciembre 2021

INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN Y COMUNICACIONES

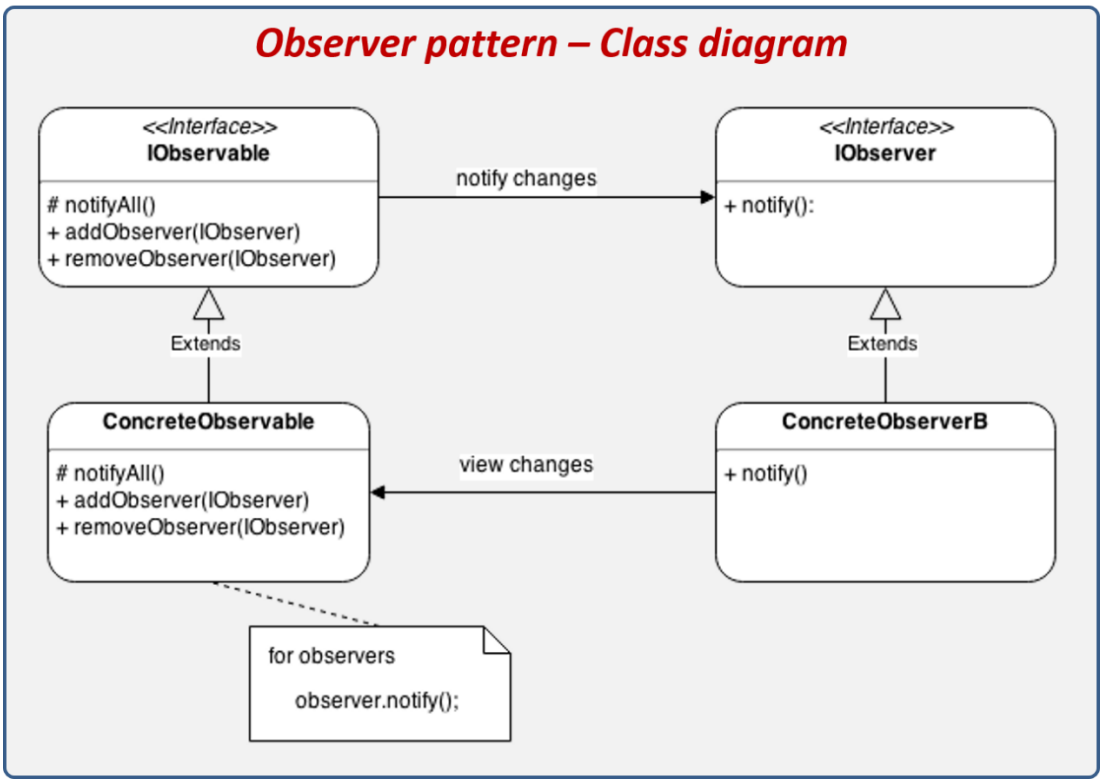
Patrones de diseño

Sergio Alberto Garza Aguilar 15211700

06 de octubre del 2021

Descripcion

El patrón de diseño Observer permite observar los cambios producidos por un objeto, de esta forma, cada cambio que afecte el estado del objeto observado lanzará una notificación a los observadores; a esto se le conoce como Publicador-Suscriptor.
Observer es uno de los principales patrones de diseño utilizados en interfaces gráficas de usuario (GUI), ya que permite desacoplar al componente gráfico de la acción a realizar.



Componentes

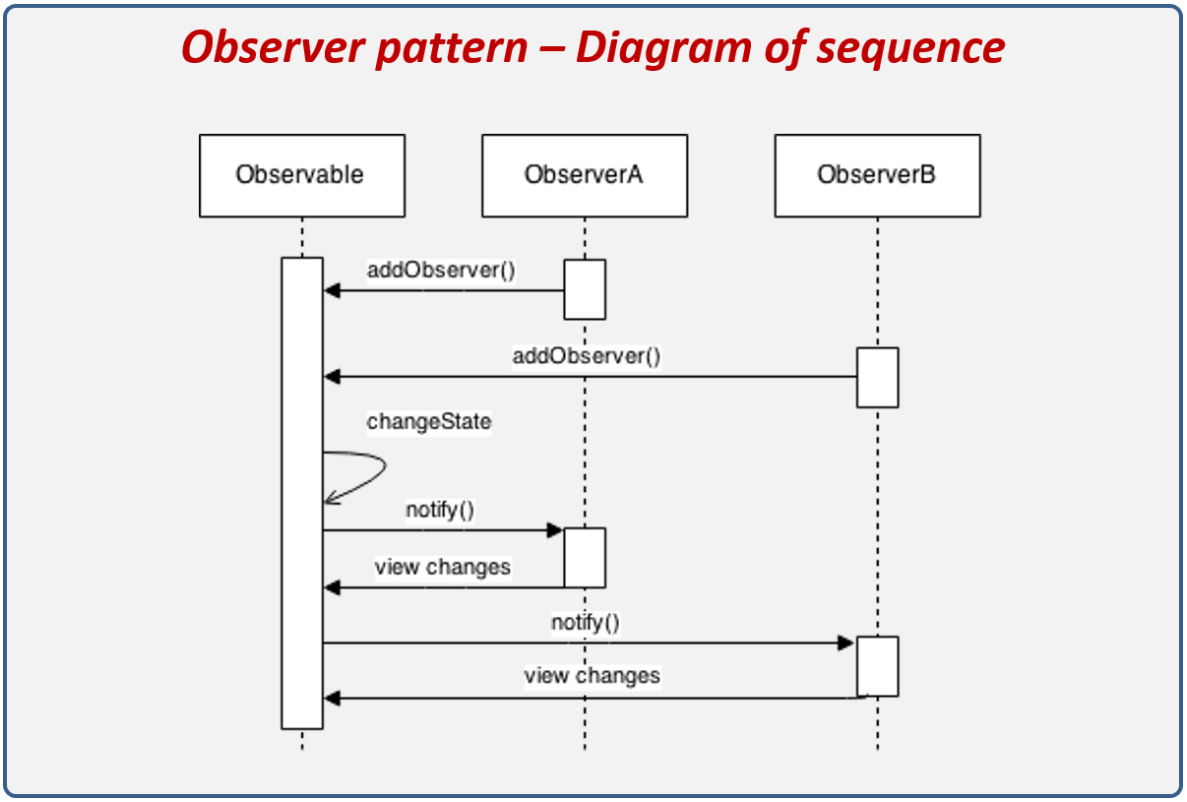
- **IObservable**: Interface que deben de implementar todos los objetos que quieren ser observados, en ella se definen los métodos mínimos que se deben implementar.

- **ConcreteObservable**: Clase que desea ser observada, ésta implementa IObservable y debe implementar sus métodos.

- **IObserver**: Interfaces que deben implementar todos los objetos que desean observar los cambios de IObservable.

- **ConcreteObserver**: Clase concreta que está atenta de los cambios de IObserver, esta clase hereda de IObserver y debe de implementar sus métodos.

Diagrama de secuencia



- El ObserverA se registra con el objeto Observable para ser notificado de algún cambio.
- El ObserverB se registra con el objeto Observable para ser notificado de algún cambio.
- Ocurre algún cambio en el estado del Observable.
- Todos los Observers son notificados con el cambio ocurrido.

Observaciones

Ventajas del modo observador

- El patrón de observador puede realizar la separación de la capa de presentación y la capa de lógica de datos, definir un mecanismo estable de transferencia de actualización de mensajes y abstraer la interfaz de actualización, de modo que una variedad de capas de presentación diferentes pueda actuar como roles de observador específicos.

- El modo observador establece un acoplamiento abstracto entre el objetivo de observación y el observador. El objetivo de observación solo necesita mantener una colección de observadores abstractos, sin conocer a sus observadores específicos. Dado que el objetivo de observación y el observador no están estrechamente unidos, pueden pertenecer a diferentes niveles de abstracción.

- El modo de observador admite la comunicación de difusión. El objetivo de observación enviará notificaciones a todos los objetos de observador registrados, lo que simplifica la dificultad del diseño del sistema de uno a muchos.

- El modo de observador cumple con los requisitos del "principio de apertura y cierre", agregar nuevos observadores específicos no necesita modificar el código del sistema original, y no hay correlación entre el observador específico y el objetivo de observación En el caso de las relaciones, también es muy conveniente agregar nuevos objetivos de observación.

Desventajas del patrón observador

- Si un objetivo de observación tiene muchos observadores directos e indirectos, llevará mucho tiempo notificar a todos los observadores.

- Si existe una dependencia circular entre el observador y el objetivo de observación, el objetivo de observación activará una llamada cíclica entre ellos, lo que puede provocar que el sistema se bloquee.

- El modo de observador no tiene un mecanismo correspondiente que le permita saber cómo ha cambiado el objetivo observado, pero solo sabe que el objetivo observado ha cambiado.

Aplicación

El patrón de diseño Observer se implementa sobre todo en aplicaciones basadas en componentes cuyo estado.

- por un lado, es muy observado por otros componentes
- y, por otro, es modificado regularmente.

Algunos de los casos de aplicación típicos de este patrón son las GUI, que actúan como interfaz de comunicación de manejo sencillo entre los usuarios y el programa. Cada vez que se modifican los datos, estos deben actualizarse en todos los componentes de la GUI. Esta situación es perfecta para la aplicación de la estructura sujeto-observador del patrón Observer.

Incluso los programas que trabajan con conjuntos de datos en formato visual (ya sean tablas clásicas o diagramas gráficos) pueden beneficiarse de la estructura de este patrón de diseño.
En lo que respecta al lenguaje de programación, el patrón Observer no conlleva, en principio, ninguna limitación específica. El único requisito es que el paradigma orientado a objetos sea compatible con el patrón. Algunos de los lenguajes más utilizados para implementar el patrón Observer son C#, C++, Java, JavaScript, Python y PHP.

Ejemplo de uso

Supongamos que tenemos una clase *Producto* con un precio y queremos emitir un mensaje en la terminal cuando un producto cambie de precio. Para implementar este patrón *Producto* debería extender de *Observable* y otra clase hacer que implemente la interfaz *Observer*, sin embargo, si no queremos o no podemos hacer que nuestra clase extienda de *Observable* para no limitarnos en nuestra jerarquía de clases o porque ya extiende de otra podemos usar composición, por otro lado, si no queremos registrar el observador en cada instancia de *Producto* sino observar cualquier instancia que se cree podemos implementar el *Observer* de forma estática en la clase *Producto*. El observable *ProductoObservable* amplia la visibilidad del método *setChanged* para poder hacer uso de él usando composición, deberemos invocarlo para que los observadores sean notificados.

Producto.java

```java
package io.github.picodotdev.blogbitix.patronobserver;

import java.math.BigDecimal;
import java.util.Observable;

public class Producto {

    public class PrecioEvent {

        private Producto producto;
        private BigDecimal precioAntiguo;
        private BigDecimal precioNuevo;

        public PrecioEvent(Producto producto,  BigDecimal precioAntiguo, BigDecimal precioNuevo) {
            this.producto = producto;
            this.precioAntiguo = precioAntiguo;
            this.precioNuevo = precioNuevo;
        }

        public Producto getProducto() {
            return producto;
        }

        public BigDecimal getPrecioAntiguo() {
            return precioAntiguo;
        }

        public BigDecimal getPrecioNuevo() {
            return precioNuevo;
        }
    }

    private static final ProductoObservable OBSERVABLE;

    private String nombre;
    private BigDecimal precio;

    static {
        OBSERVABLE = new ProductoObservable();
    }

    public static Observable getObservable() {
        return OBSERVABLE;
    }

    public Producto(String nombre, BigDecimal precio) {
        this.nombre = nombre;
        this.precio = precio;
    }

    public String getNombre() {
        return nombre;
    }

    public BigDecimal getPrecio() {
        return precio;
    }

    public void setPrecio(BigDecimal precio) {
        PrecioEvent event = new PrecioEvent(this, this.precio, precio);

        this.precio = precio;

        synchronized (OBSERVABLE) {
            OBSERVABLE.setChanged();
            OBSERVABLE.notifyObservers(event);
        }
    }

    private static class ProductoObservable extends Observable {
        @Override
        public synchronized void setChanged() {
            super.setChanged();
        }
    }
}
```

Main.java

```java
package io.github.picodotdev.blogbitix.patronobserver;

import java.math.BigDecimal;
import java.util.Observer;

public class Main {
    public static void main(String[] args) {
        Producto p1 = new Producto("Libro", new BigDecimal("3.99"));
        Producto p2 = new Producto("Lector libros electrónico", new BigDecimal("129"));

        Observer o1 = new ProductoObserver();
        Producto.getObservable().addObserver(o1);

        p1.setPrecio(new BigDecimal("4.99"));
        p2.setPrecio(new BigDecimal("119"));
    }
}
```

ProductoObserver.java

```java
package io.github.picodotdev.blogbitix.patronobserver;

import java.util.Observable;
import java.util.Observer;

import io.github.picodotdev.blogbitix.patronobserver.Producto.PrecioEvent;

public class ProductoObserver implements Observer {

    @Override
    @SuppressWarnings("unchecked")
    public void update(Observable observable, Object args) {
        if (args instanceof PrecioEvent) {
            PrecioEvent evento = (PrecioEvent) args;
            System.out.printf("El producto %s ha cambiado de precio de %s a %s%n", evento.getProducto().getNombre(), evento.getPrecioAntiguo(), evento.getPrecioNuevo());
        }
    }
}
```

System.out

```
El producto Libro ha cambiado de precio de 3,99 a 4,99
El producto Lector libros electrónico ha cambiado de precio de 129 a 119
```

System.out

Fuentes de consulta.

(19 de octubre del 2020), IONOS Digital guide, desarrollo web, ¿Que es el patrón observer?, (https://www.ionos.mx/digitalguide/paginas-web/desarrollo-web/que-es-el-patron-observer/)

(2020) Programmerclick, Patron observador de diseños en java (https://programmerclick.com/article/1940843368/)

Oscar Blancarte (08 marzo del 2021), introducción a los patrones de diseño un enfoque practico, Observer Patter, (https://reactiveprogramming.io/blog/es/patrones-de-diseno/observer)

Picodotdev (24 de octubre del 2015) El patrón de diseño Observer y una forma de implementarlo en Java, (https://picodotdev.github.io/blog-bitix/2015/10/el-patron-de-diseno-observer-y-una-forma-de-implementarlo-en-java/)