Práctica 1
Arquitectura del Software

Marcos Galán Carrillo NIP: 874095
Mario Hernández Pereda NIP: 873094

21 de febrero de 2025

Implementación de un observer

1. Introducción

En esta práctica vamos a realizar la implementación del patrón observer. Para realizar esta tarea, hemos decidido implementar un sistema similar al de los canales de youtube. Para simular el patrón vamos a usar un sistema de suscripciones y notificaciones de los vídeos, directos o comunicados.

2. Patrón Observer

Para realizar el patrón observer, hemos puesto como sujeto los canales de youtube, y como observer los suscriptores de dichos canales. Además, hemos añadido los atributos y métodos necesarios para la implementación básica del patrón. Sin embargo, también hemos implementado algunos métodos adicionales que nos han sido de gran utilidad durante la ejecución del programa, y hemos realizado algunas modificaciones que hacían nuestra implementación más eficiente.

Está es nuestra implementación relacionada con la implementación base del patrón observer:

- **CanalYt.java (Subject)**: clase en la cual se implementa la clase "Sujeto", que se relacionara con una serie de suscriptores (Observers), avisandoles de si hay un nuevo video.
- **CanalYtConcreto.java (ConcreteSubject)**: clase hija de CanalYt, la cual adoptara una funcionalidad similar a la clase padre y estara relacionada con SuscriptorConcreto, para poder actualizar el estado de un suscriptor en concreto, cambiando su estado
- **Suscriptor.java (Observer)**: clase en la que se implementa la clase "observador", que se relaciona con una sola instancia de la clase CanalYt.java, la cual representa el canal de youtube al que está suscrito dicho usuario.
- **SuscriptorConcreto.java (ConcreteObserver)**: clase hija de Suscriptor, la cual estara relacionada con un canal de youtube en concreto teniendo que cambiar su estado cuando sea necesario (cuando haya un notify)

3. Diagrama de clases

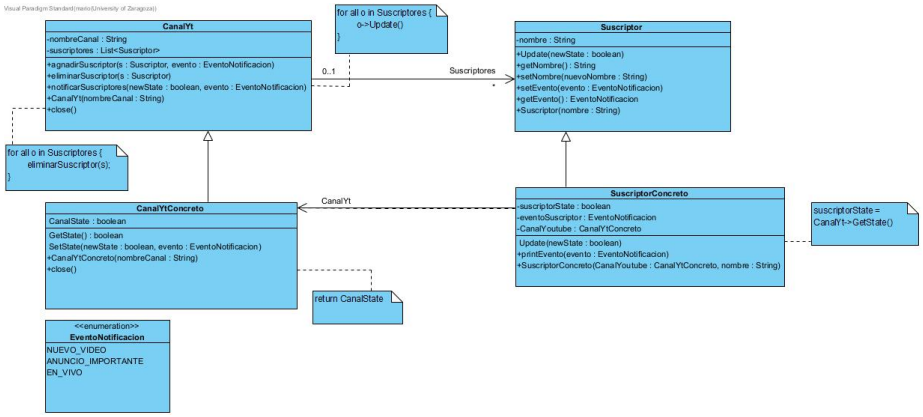

Figura 1: Diagrama de clases del patrón Observer

4. Justificación de decisiones tomadas

Podemos ver que en la figura anterior tenemos dos relaciones, una que relaciona el subject con uno o más observadores, y otra que relaciona, cada observer concreto con un subject concreto. Podemos observar esto en nuestro programa, para relacionar los subjects con los observers, lo que hemos hecho es que el subject tenga una lista donde puede almacenar los observers con los que se relaciona (array de suscriptores). Por otro lado, cada uno de los observers concretos (suscriptores concretos) tienen un atributo para identificar el subject concreto con el que están relacionados.

En cuanto al diseño, una de las decisiones más importantes que hemos tomado es el hacer uso del método push. Mediante este método, no es necesario que los observadores hagan uso de la función *getState()*, sino que en las propias funciones de *notificarSuscriptores()* y *update()*, ya pasamos el nuevo estado. Pensamos que en nuestro caso, sería mucho más eficiente usar esto al método pull, puesto que al ser todos los observadores del mismo tipo siempre esperan la misma información. De esta manera, no es necesario que cada observador solicite la información acerca del estado actual, siendo esta versión mucho más eficiente.

```java
// Se notifica a los suscriptores del canal de Youtube
@Override
public void notificarSuscriptores(boolean nuevoEstado, EventoNotificacion evento){
    boolean nadie = false;
    for(Suscriptor s : suscriptores){
        if (evento == s.getEvento()){
            s.update(nuevoEstado);
            nadie = true;
        }
    }
    if (! nadie) {System.out.println("En este canal no hay nadie interesado en ese tipo de notificaciones");}
}
```

Figura 2: Notificación de los suscriptores

```java
// Actualiza el estado del suscriptor
@Override
public void update(boolean nuevoEstado){
    System.out.println(x:this.getNombre() + " ha recibido una notificación.");
    if (nuevoEstado) {
        System.out.println(x:this.getNombre() + " ha consultado: ¡ " + this.printEvento(e:this.getEvento()) + " disponible !");
    } else {
        System.out.println(x:this.getNombre() + " ha consultado: No hay " + this.printEvento(e:this.getEvento()) + ".");
    }
    suscriptorState = suscriptorState == nuevoEstado ? suscriptorState : nuevoEstado;
}
```

Figura 3: Update del estado de los suscriptores mediante el método push

Además de este cambio, también hemos introducido los aspectos de interés. En nuestro caso concreto, los suscriptores al suscribirse a un canal, especifican en que aspecto están interesados. Los aspectos posibles son:

- Hay un nuevo vídeo disponible
- Hay una emisión en directo en este momento
- Hay un anuncio importante disponible

La ventaja de este sistema es, que los diferentes suscriptores no tienen que revisar las notificaciones, para comprobar si la actualización corresponde a algo en lo que están interesados, sino que el propio canal se encargará de notificar solo a aquellas personas que estén interesadas en ese aspecto en concreto.

```java
// Tipo de dato enumerado que representa las posibles notificaciones
// que puede recibir un suscriptor segun sus intereses
public enum EventoNotificacion {
    NUEVO_VIDEO,
    EN_VIVO,
    ANUNCIO_IMPORTANTE;
}
```

Figura 4: Enumerado de los diferentes aspectos de interés

Por último, ante la posible eliminación de un canal de youtube, hemos implementado un método que elimina todos los suscriptores de dicho canal antes de eliminarlo, para que el suscriptor no permanezca con una referencia a un canal inexistente.

```java
// Método para liberar recursos de un Canal de Youtube
@Override
public void close() {
    System.out.println(x:"El canal de YouTube: " + nombreCanal + " ha sido eliminado.");
    System.out.println(x:"Liberando suscriptores...");
    suscriptores.clear();  // Elimina todos los suscriptores
}
```

Figura 5: Método close para el borrado de un canal de youtube

5. Diagrama de Secuencia

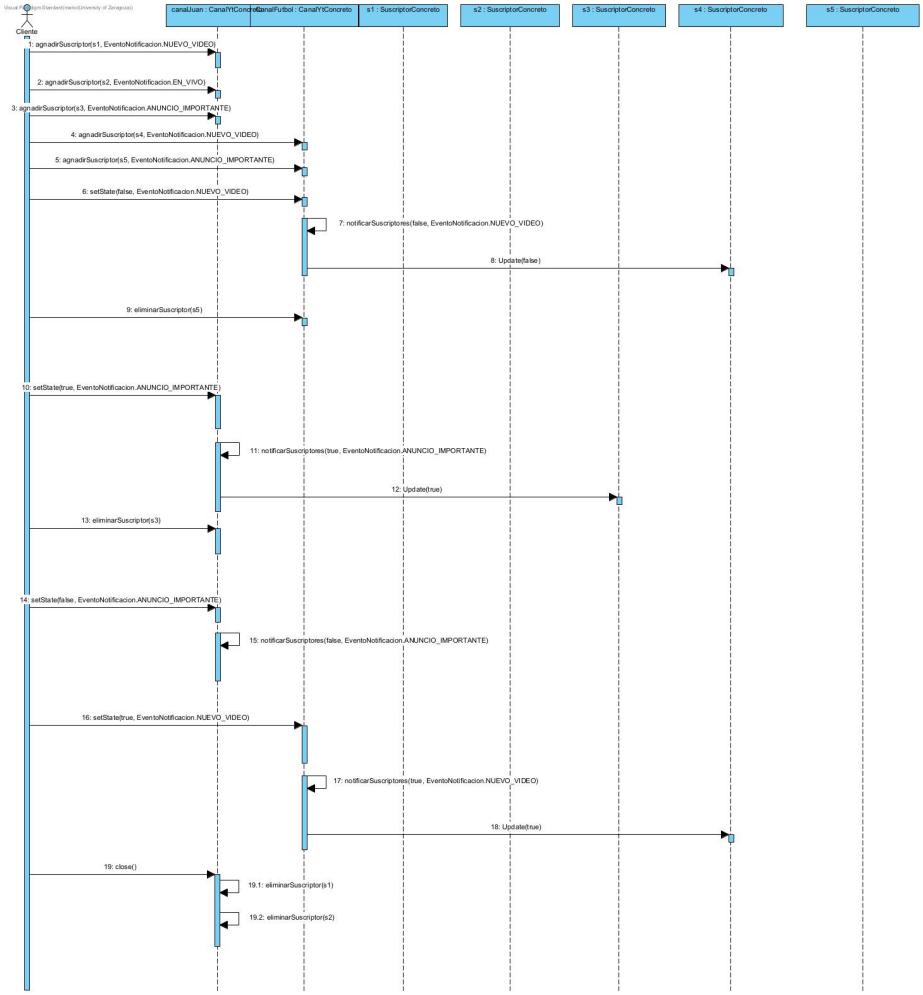

Figura 6: Diagrama de secuencia del patrón Observer