https://unabedu-my.sharepoint.com/:v:/r/personal/karen_fandino_o365_unab_edu_co/Documents/Grabaciones/Reuni%C3%B3n%201-20211104_192530-Grabaci%C3%B3n%20de%20la%20reuni%C3%B3n.mp4?csf=1&web=1&e=7cVUwD

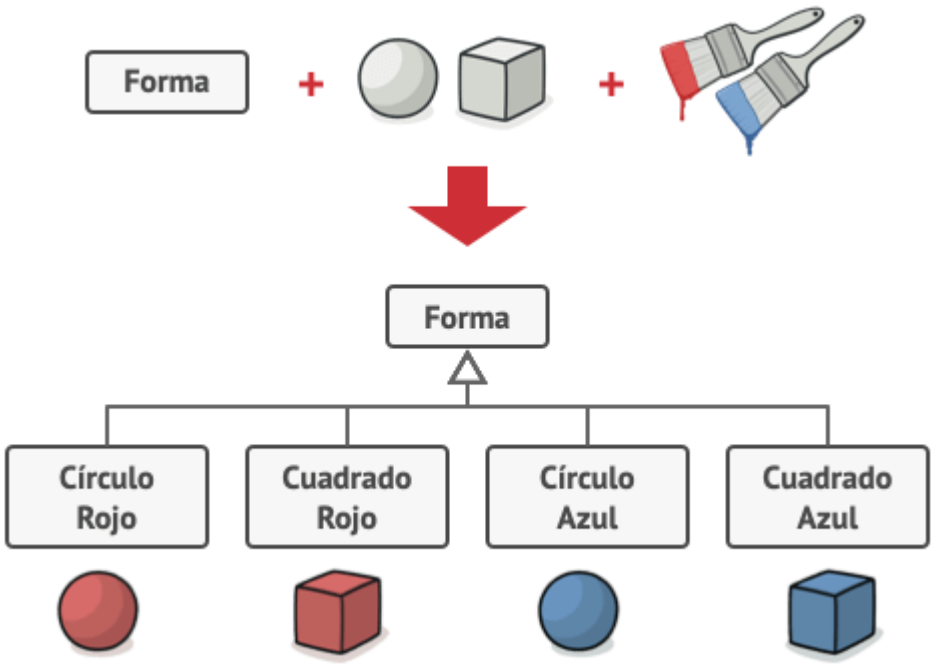
Patrón Bridge

Bridge es un patrón de diseño estructural que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.

EJEMPLO:

PROBLEMA

Digamos que tienes una clase geométrica `Forma` con un par de subclases: `Círculo` y `Cuadrado`. Deseas extender esta jerarquía de clase para que incorpore colores, por lo que planeas crear las subclases de forma `Rojo` y `Azul`. Sin embargo, como ya tienes dos subclases, tienes que crear cuatro combinaciones de clase, como `CírculoAzul` y `CuadradoRojo`.



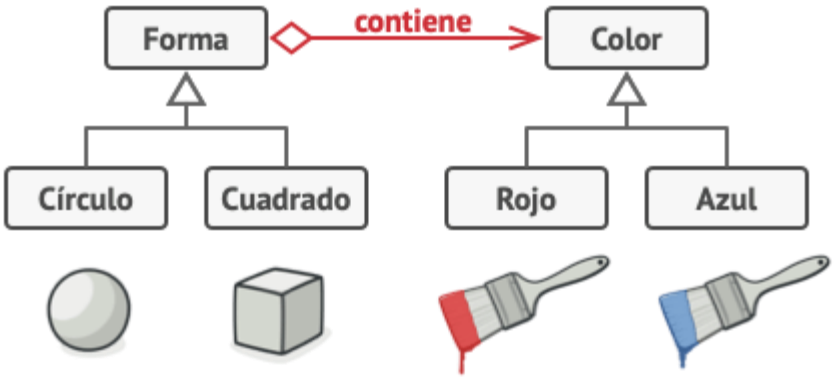
Añadir nuevos tipos de forma y color a la jerarquía hará que ésta crezca exponencialmente. Por ejemplo, para añadir una forma de triángulo deberás introducir dos

subclases, una para cada color. Y, después, para añadir un nuevo color habrá que crear tres subclases, una para cada tipo de forma. Cuanto más avancemos, peor será.

SOLUCIÓN

Este problema se presenta porque intentamos extender las clases de forma en dos dimensiones independientes: por forma y por color. Es un problema muy habitual en la herencia de clases.

El patrón Bridge intenta resolver este problema pasando de la herencia a la composición del objeto. Esto quiere decir que se extrae una de las dimensiones a una jerarquía de clases separada, de modo que las clases originales referencian un objeto de la nueva jerarquía, en lugar de tener todo su estado y sus funcionalidades dentro de una clase.



Puedes evitar la explosión de una jerarquía de clase transformándola en varias jerarquías relacionadas.

Con esta solución, podemos extraer el código relacionado con el color y colocarlo dentro de su propia clase, con dos subclases: `Rojo` y `Azul`. La clase `Forma` obtiene entonces un campo de referencia que apunta a uno de los objetos de color. Ahora la forma puede delegar cualquier trabajo relacionado con el color al objeto de color vinculado. Esa referencia actuará como un puente entre las clases `Forma` y `Color`. En adelante, añadir nuevos colores no exigirá cambiar la jerarquía de forma y viceversa.

```
// La "abstracción" define la interfaz para la parte de
// "control" de las dos jerarquías de clase. Mantiene una
// referencia a un objeto de la jerarquía de "implementación" y
// delega todo el trabajo real a este objeto.
class RemoteControl is
    protected field device: Device
    constructor RemoteControl(device: Device) is
        this.device = device
    method togglePower() is
        if (device.isEnabled()) then
            device.disable()
```

```
        else
            device.enable()
    method volumeDown() is
        device.setVolume(device.getVolume() - 10)
    method volumeUp() is
        device.setVolume(device.getVolume() + 10)
    method channelDown() is
        device.setChannel(device.getChannel() - 1)
    method channelUp() is
        device.setChannel(device.getChannel() + 1)


// Puedes extender clases de la jerarquía de abstracción
// independientemente de las clases de dispositivo.
class AdvancedRemoteControl extends RemoteControl is
    method mute() is
        device.setVolume(0)


// La interfaz de "implementación" declara métodos comunes a
// todas las clases concretas de implementación. No tiene por
// qué coincidir con la interfaz de la abstracción. De hecho,
// las dos interfaces pueden ser completamente diferentes.
// Normalmente, la interfaz de implementación únicamente
// proporciona operaciones primitivas, mientras que la
// abstracción define operaciones de más alto nivel con base en
// las primitivas.
interface Device is
    method isEnabled()
    method enable()
    method disable()
    method getVolume()
    method setVolume(percent)
    method getChannel()
    method setChannel(channel)


// Todos los dispositivos siguen la misma interfaz.
class Tv implements Device is
    // ...

class Radio implements Device is
    // ...


// En algún lugar del código cliente.
tv = new Tv()
remote = new RemoteControl(tv)
remote.togglePower()

radio = new Radio()
remote = new AdvancedRemoteControl(radio)
```

PATRON OBSERVER

Observer es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

PROBLEMA

Imagina que tienes dos tipos de objetos: un objeto `Cliente` y un objeto `Tienda`. El cliente está muy interesado en una marca particular de producto (digamos, un nuevo modelo de iPhone) que estará disponible en la tienda muy pronto.

El cliente puede visitar la tienda cada día para comprobar la disponibilidad del producto. Pero, mientras el producto está en camino, la mayoría de estos viajes serán en vano.

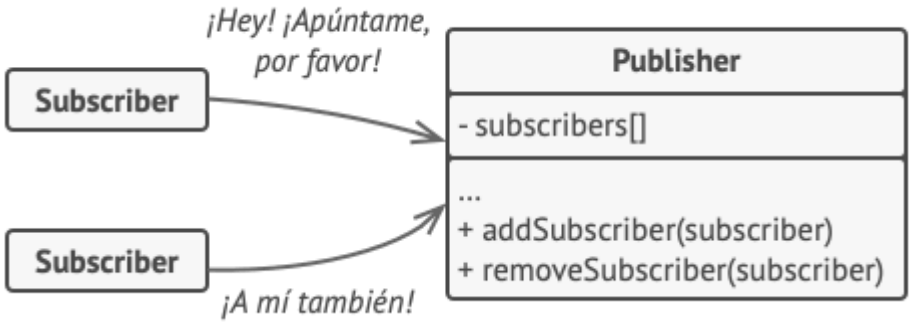
Por otro lado, la tienda podría enviar cientos de correos (lo cual se podría considerar spam) a todos los clientes cada vez que hay un nuevo producto disponible. Esto ahorraría a los clientes los interminables viajes a la tienda, pero, al mismo tiempo, molestaría a otros clientes que no están interesados en los nuevos productos.

Parece que nos encontramos ante un conflicto. O el cliente pierde tiempo comprobando la disponibilidad del producto, o bien la tienda desperdicia recursos notificando a los clientes equivocados.

SOLUCIÓN

El objeto que tiene un estado interesante suele denominarse *sujeto*, pero, como también va a notificar a otros objetos los cambios en su estado, le llamaremos *notificador* (en ocasiones también llamado *publicador*). El resto de los objetos que quieren conocer los cambios en el estado del notificador, se denominan *suscriptores*.

El patrón Observer sugiere que añadas un mecanismo de suscripción a la clase notificadora para que los objetos individuales puedan suscribirse o cancelar su suscripción a un flujo de eventos que proviene de esa notificadora. ¡No temas! No es tan complicado como parece. En realidad, este mecanismo consiste en: 1) un campo matriz para almacenar una lista de referencias a objetos suscriptores y 2) varios métodos públicos que permiten añadir suscriptores y eliminarlos de esa lista.

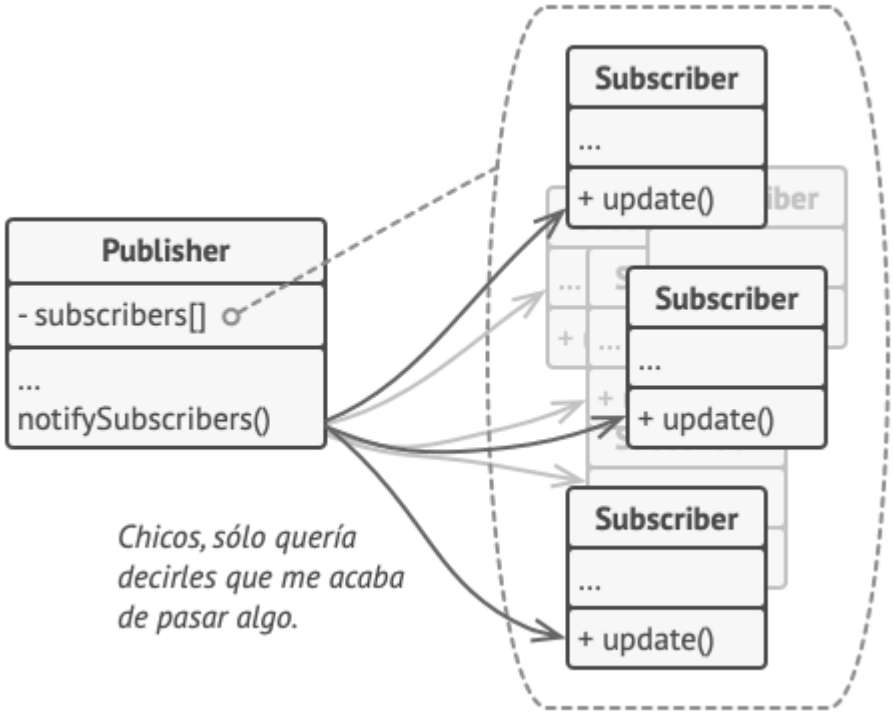


Un mecanismo de suscripción permite a los objetos individuales suscribirse a notificaciones de eventos.

Ahora, cuando le sucede un evento importante al notificador, recorre sus suscriptores y llama al método de notificación específico de sus objetos.

Las aplicaciones reales pueden tener decenas de clases suscriptoras diferentes interesadas en seguir los eventos de la misma clase notificadora. No querrás acoplar la notificadora a todas esas clases. Además, puede que no conozcas algunas de ellas de antemano si se supone que otras personas pueden utilizar tu clase notificadora.

Por eso es fundamental que todos los suscriptores implementen la misma interfaz y que el notificador únicamente se comunique con ellos a través de esa interfaz. Esta interfaz debe declarar el método de notificación junto con un grupo de parámetros que el notificador puede utilizar para pasar cierta información contextual con la notificación.



El notificador notifica a los suscriptores invocando el método de notificación específico de sus objetos.

Si tu aplicación tiene varios tipos diferentes de notificadores y quieres hacer a tus suscriptores compatibles con todos ellos, puedes ir más allá y hacer que todos los notificadores sigan la misma interfaz. Esta interfaz sólo tendrá que describir algunos métodos de suscripción. La interfaz permitirá a los suscriptores observar los estados de los notificadores sin acoplarse a sus clases concretas.

```
// La clase notificadora base incluye código de gestión de
// suscripciones y métodos de notificación.
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        listeners.remove(eventType, listener)

    method notify(eventType, data) is
        foreach (listener in listeners.of(eventType)) do
            listener.update(data)

// El notificador concreto contiene lógica de negocio real, de
// interés para algunos suscriptores. Podemos derivar esta clase
// de la notificadora base, pero esto no siempre es posible en
// el mundo real porque puede que la notificadora concreta sea
// ya una subclase. En este caso, puedes modificar la lógica de
// la suscripción con composición, como hicimos aquí.
class Editor is
    public field events: EventManager
    private field file: File

    constructor Editor() is
        events = new EventManager()

    // Los métodos de la lógica de negocio pueden notificar los
    // cambios a los suscriptores.
    method openFile(path) is
        this.file = new File(path)
        events.notify("open", file.name)

    method saveFile() is
        file.write()
        events.notify("save", file.name)

    // ...


// Aquí está la interfaz suscriptora. Si tu lenguaje de
// programación soporta tipos funcionales, puedes sustituir toda
```

```
// la jerarquía suscriptora por un grupo de funciones.


interface EventListener is
    method update(filename)

// Los suscriptores concretos reaccionan a las actualizaciones
// emitidas por el notificador al que están unidos.
class LoggingListener implements EventListener is
    private field log: File
    private field message

    constructor LoggingListener(log_filename, message) is
        this.log = new File(log_filename)
        this.message = message

    method update(filename) is
        log.write(replace('%s',filename,message))

class EmailAlertsListener implements EventListener is
    private field email: string

    constructor EmailAlertsListener(email, message) is
        this.email = email
        this.message = message

    method update(filename) is
        system.email(email, replace('%s',filename,message))


// Una  aplicación puede configurar notificadores y suscriptores
// durante el tiempo de ejecución.
class Application is
    method config() is
        editor = new Editor()

        logger = new LoggingListener(
            "/path/to/log.txt",
            "Someone has opened the file: %s")
        editor.events.subscribe("open", logger)

        emailAlerts = new EmailAlertsListener(
            "admin@example.com",
            "Someone has changed the file: %s")
        editor.events.subscribe("save", emailAlerts)
```