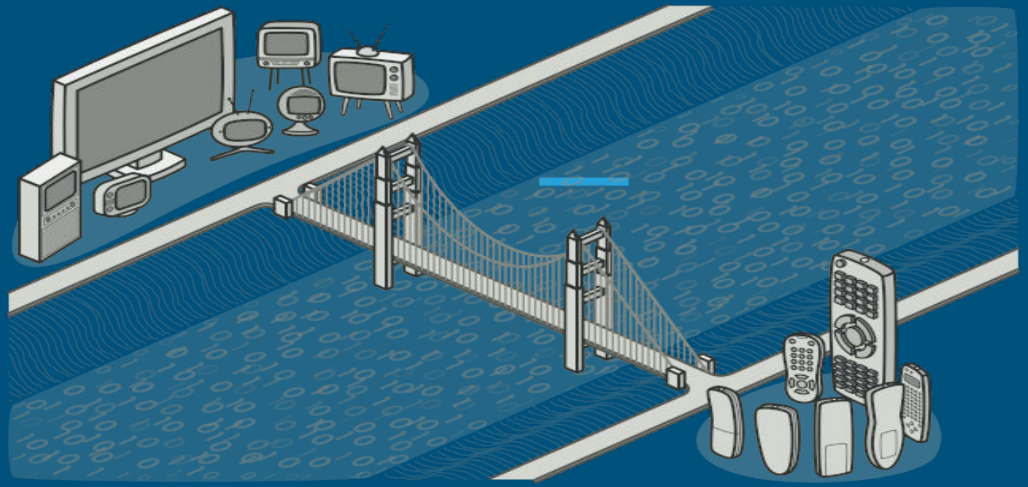
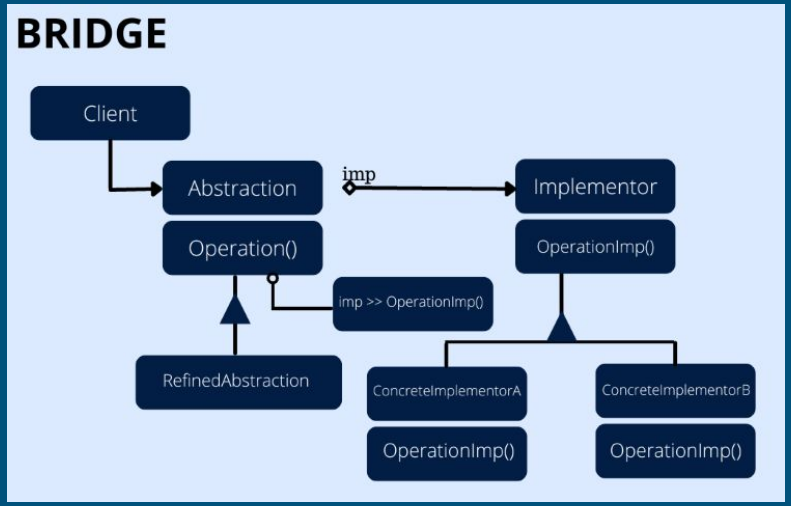
JOSE MIGUEL SANTOS ROSADO

# PATRON DE DISEÑO “BRIDGE”



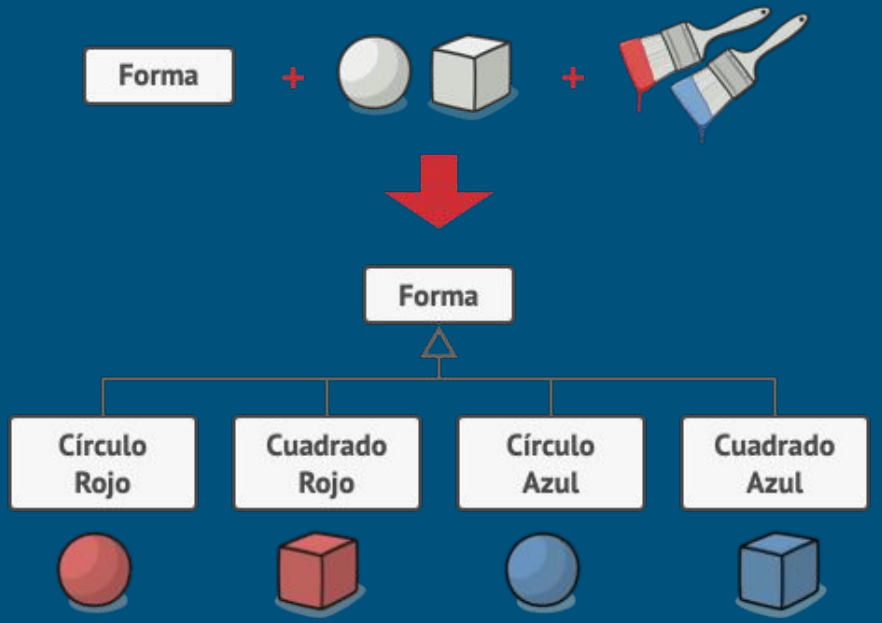
# PROPOSITO

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.



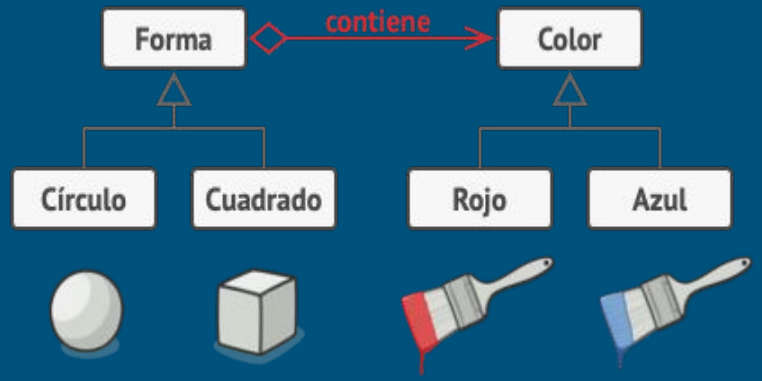
# PROBLEMA

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**.



# SOLUCION

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.



# Abstracción e implementación

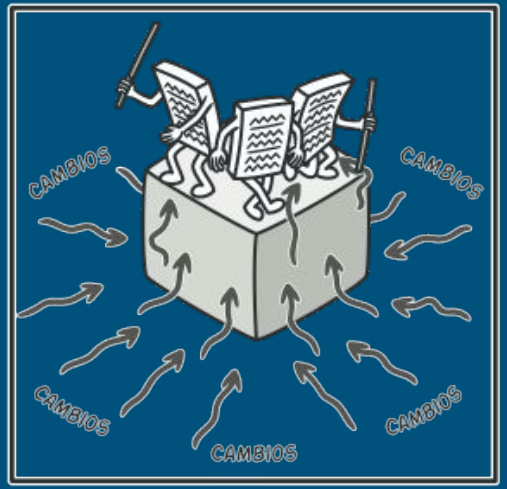
La Abstracción (también llamada interfaz) es una capa de control de alto nivel para una entidad. Esta capa no tiene que hacer ningún trabajo real por su cuenta, sino que debe delegar el trabajo a la capa de implementación (también llamada plataforma).

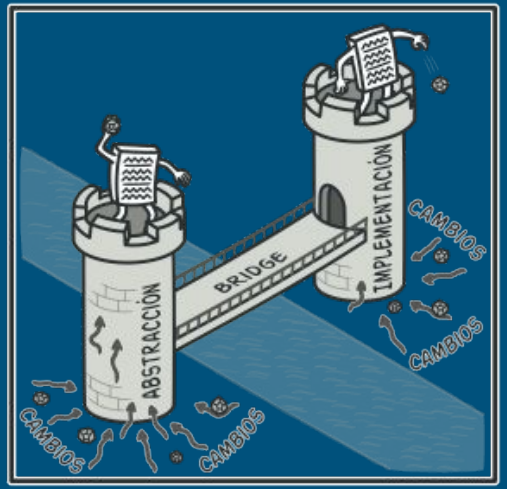
Cuando hablamos de aplicación reales, la abstracción puede representarse por una interfaz gráfica de usuario (GUI), y la implementación puede ser el código del sistema operativo subyacente (API) a la que la capa GUI llama en respuesta a las interacciones del usuario.

En términos generales, puedes extender esa aplicación en dos direcciones independientes:

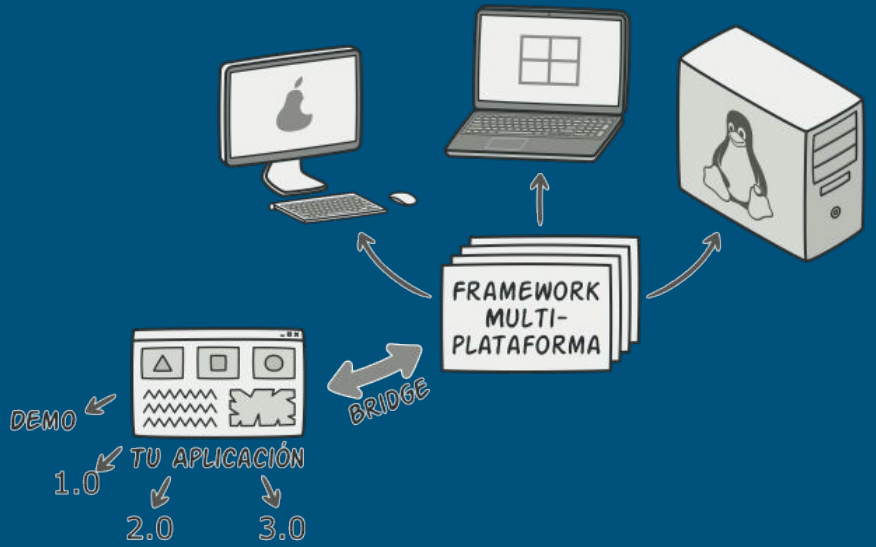
Tener varias GUI diferentes (por ejemplo, personalizadas para clientes regulares o administradores).

Soportar varias API diferentes (por ejemplo, para poder lanzar la aplicación con Windows, Linux y macOS).

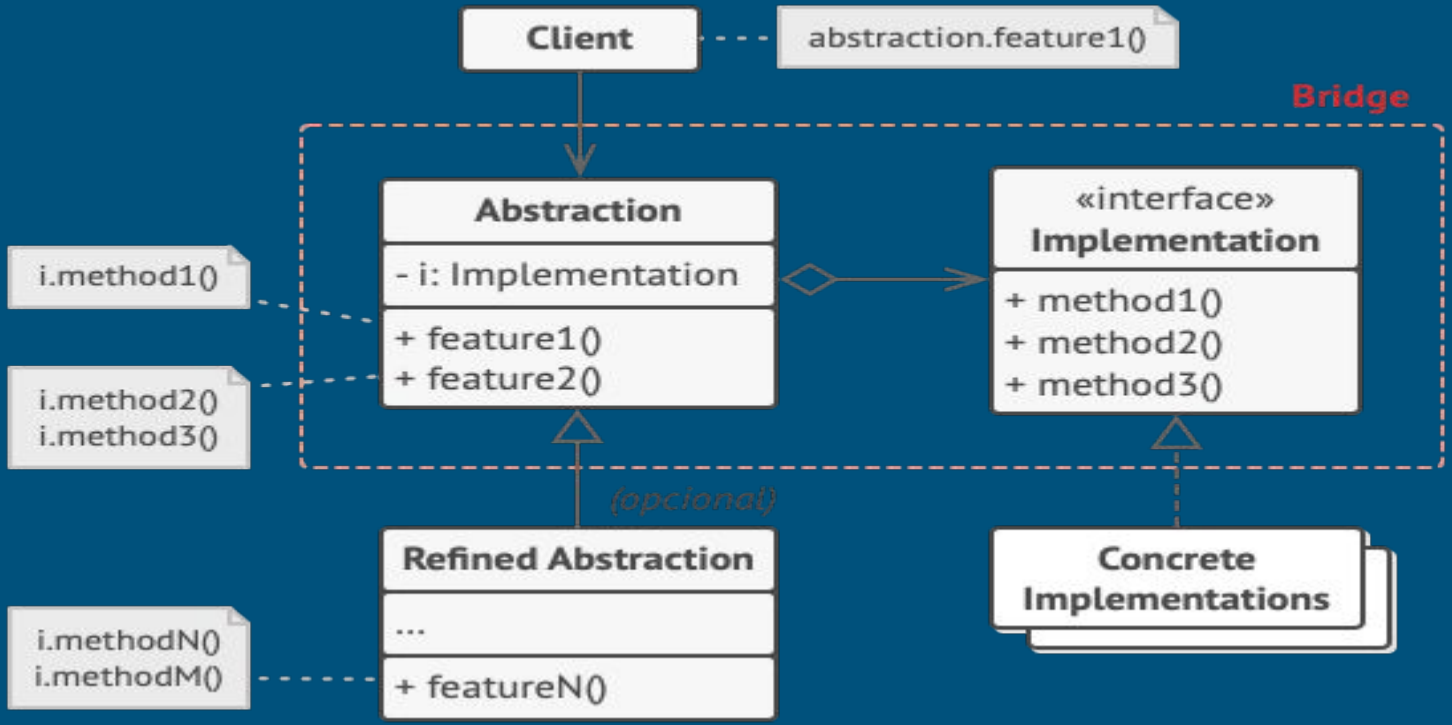


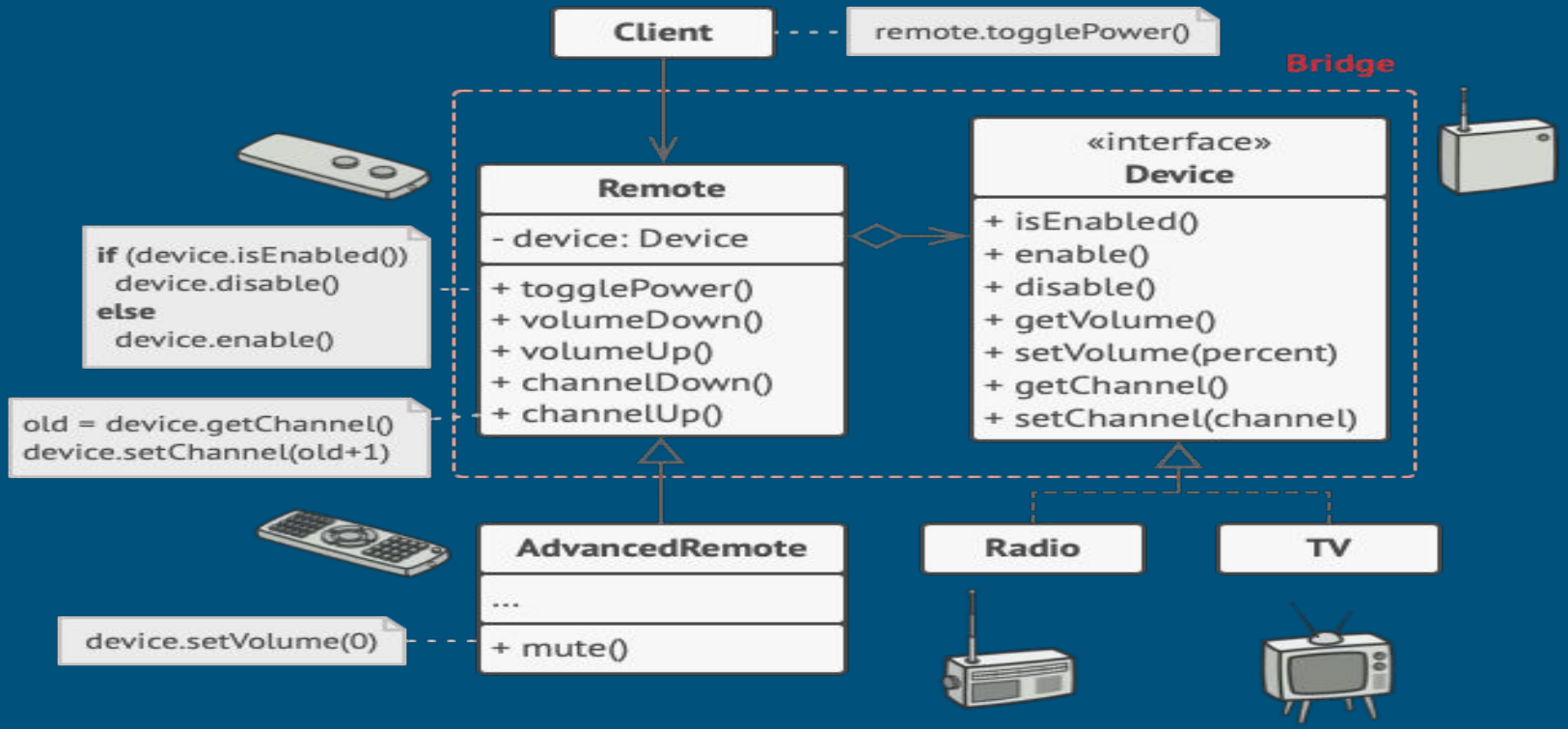



Puedes poner orden en este caos metiendo el código relacionado con combinaciones específicas interfaz-plataforma dentro de clases independientes. Sin embargo, pronto descubrirás que hay muchas de estas clases. La jerarquía de clase crecerá exponencialmente porque añadir una nueva GUI o soportar una API diferente exigirá que se creen más y más clases.



# ESTRUCTURA


Client
abstraction.feature1()
Bridge
Abstraction
- i: Implementation
+ feature1()
+ feature2()
«interface»
Implementation
+ method1()
+ method2()
+ method3()
i.method1()
i.method2()
i.method3()
(opcional)
Refined Abstraction
...
+ featureN()
Concrete
Implementations
i.methodN()
i.methodM()


Client
remote.togglePower()
Bridge
Remote
- device: Device
+ togglePower()
+ volumeDown()
+ volumeUp()
+ channelDown()
+ channelUp()
«interface»
Device
+ isEnabled()
+ enable()
+ disable()
+ getVolume()
+ setVolume(percent)
+ getChannel()
+ setChannel(channel)
if (device.isEnabled())
device.disable()
else
device.enable()
old = device.getChannel()
device.setChannel(old+1)
AdvancedRemote
...
+ mute()
device.setVolume(0)
Radio
TV

# APLICABILIDAD

Utiliza el patrón Bridge cuando quieras dividir y organizar una clase monolítica que tenga muchas variantes de una sola funcionalidad (por ejemplo, si la clase puede trabajar con diversos servidores de bases de datos).

# PROS Y CONTRAS

1. Puedes crear clases y aplicaciones independientes de plataforma.
2. El código cliente funciona con abstracciones de alto nivel. No está expuesto a los detalles de la plataforma.
3. Principio de abierto/cerrado. Puedes introducir nuevas abstracciones e implementaciones independientes entre sí.
4. Principio de responsabilidad única. Puedes centrarte en la lógica de alto nivel en la abstracción y en detalles de la plataforma en la implementación.

1. Puede ser que el código se complique si aplicas el patrón a una clase muy cohesionada.
2. Puede agregar complejidad inicial en el diseño.
3. Mayor número de clases y objetos.