Bridge

Abstracción

implementación

Descripción
La idea de este patrón es desacoplar una **abstracción** de su **implementación**, de manera que ambas puedan ser modificadas independientemente sin necesidad de que la alteración de una afecte a la otra.

Clasificación
Patrón estructural

Motivación
Cuando una abstracción puede tener varias posibles implementaciones, normalmente hacemos uso de la herencia para acomodar esta necesidad. Una clase abstracta define de la interfaz de la abstracción y las subclases concretas lo implementan. Esto no es los suficientemente flexible, es difícil de mantener y modificar y no permite reutilizar los componentes. Necesitamos un patrón que solucione esta problemática.

Implica crear dos jerarquías, una para la **abstracción** y otra para la **Implementación**. Una de estas jerarquías (a menudo denominada **Abstracción)** obtendrá una referencia a un objeto de la segunda jerarquía **(Implementación**). La abstracción podrá delegar algunas (en ocasiones, la mayoría) de sus llamadas al objeto de las implementaciones. Como todas las implementaciones tendrán una interfaz común, serán intercambiables dentro de la abstracción.

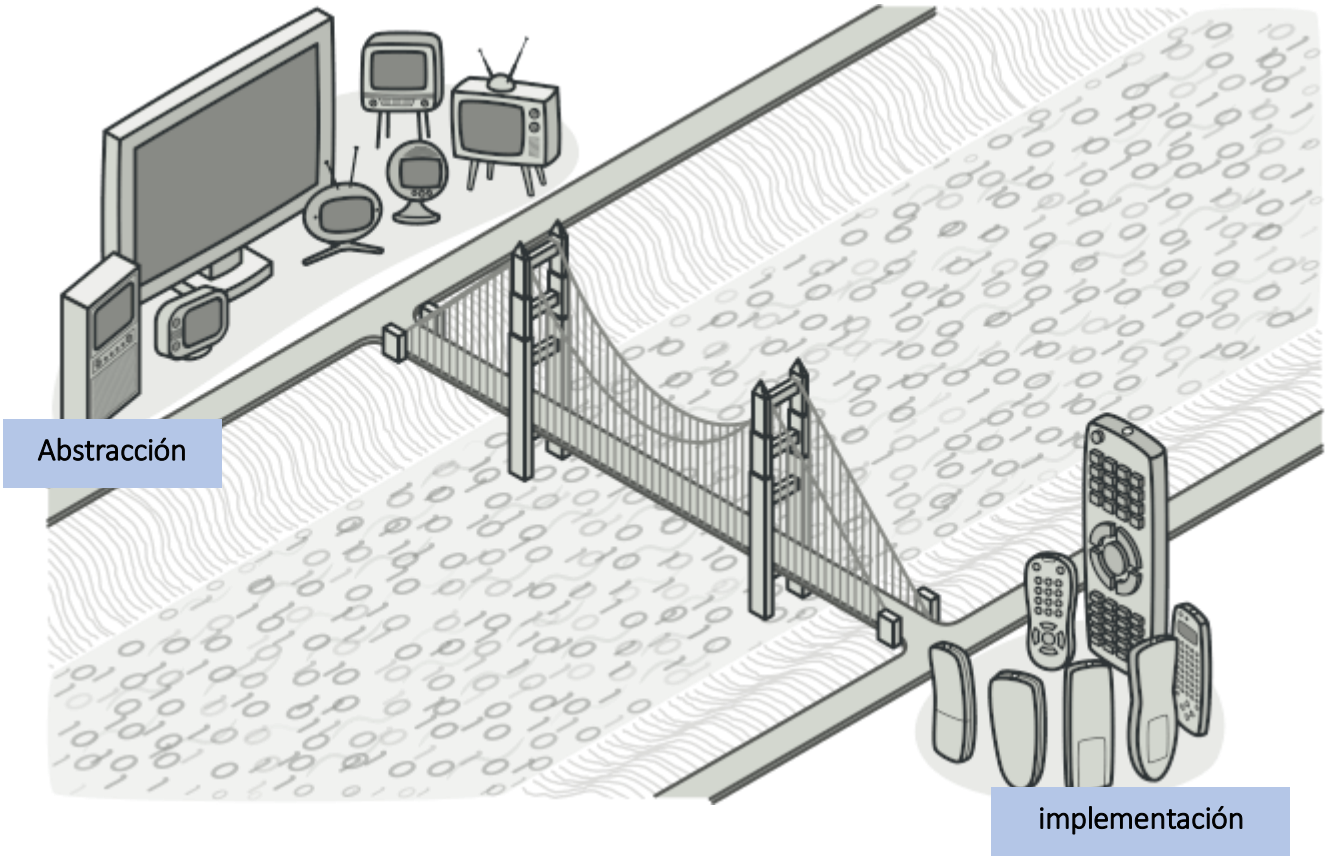
Bridge es un patrón de diseño estructural que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.

Problema y Contexto:
Cuando tenemos la necesidad que la implementación de una abstracción sea modificada en tiempo de ejecución o nuestro sistema requiere que la funcionalidad (parcial o total) de nuestra abstracción esté desacoplada de la implementación para poder modificar tanto una como otra sin que ello obligue a la cambiar las demás clases.

Se aplica cuando:

- Queremos evitar enlaces permanentes entre una **abstracción** y una **implementación.**
- Tanto las **abstracciones** como las **implementaciones** deben ser extensibles por medio de subclases.
- Queremos que los cambios en la implementación de una abstracción no afecten al cliente.

- Necesitamos que la implementación de una característica sea compartida entre múltiples objetos.



Solución y Estructura:
Partimos de una abstracción base (clase abstracta o interfaz) que tendrá como atributo un objeto que será el que realice las funciones a implementar y que denominaremos **implementador.** Nuestra **abstracción** contendrá todas las operaciones que nuestro sistema requiera.

Por otro lado, tendremos el **implementador,** que será una interfaz que defina las operaciones necesarias para cubrir la funcionalidad que ofrece nuestra **abstracción.** Para dotar de funcionalidad a las operaciones definidas podremos crear diferentes **implementadores** concretos que implementen dicha interfaz.

Debemos crear una clase que herede de nuestra abstracción para definir concretamente lo que hacen sus métodos, pero ésta deberá implementar la funcionalidad mediante el atributo que heredó del padre (que almacena un implementador).

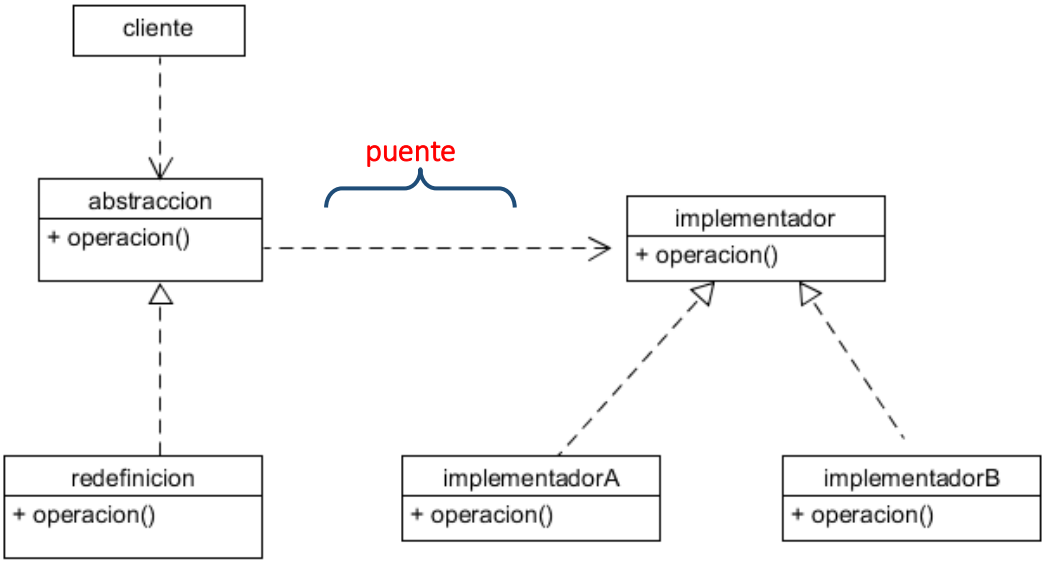
Este patrón de diseño permite desacoplar una **abstracción** de su **implementación**
de manera que ambas puedan variar de forma independiente.

Supongamos que tenemos una clase abstracta en la que se define un método que deberá implementar cada clase que herede de ella: ¿cómo haríamos si una clase hija necesitase

implementarlo de forma que realizase acciones diferentes dependiendo de determinadas circunstancias?.

En dichos casos nos resultaría útil el patrón **Bridge** (puente) ya que 'desacopla una abstracción' (un método abstracto) al permitir indicar (durante la ejecución del programa) a una clase qué 'implementación' del mismo debe utilizar (qué acciones ha de realizar).

El **diagrama UML** de este patrón es el siguiente:



Participantes

- **Abstracción:** define una interfaz abstracta, mantiene una referencia a un objeto de tipo ***Implementador***.
- **Redefinición:** extiende la interfaz definida por ***Abstracción***
- **Implementador:** define la interfaz para la implementación de clases. Esta interfaz no se tiene que corresponder exactamente con la interfaz de ***Abstracción***; de hecho, las dos interfaces pueden ser bastante diferente. Típicamente la interfaz ***Implementador*** provee solo operaciones específicas y *Abstraction* define operaciones de alto nivel basadas en estas específicas.
- **ImplementadorA, ImplementadorB:** implementa la interfaz de ***Implementador*** y define su implementación concreta.

```
//-------------------------implementador.h-------------------------------
```

```
#include <iostream>
using namespace std;

class  implementador {
public:
	virtual  void operacion() = 0;
};
//----------------------------implementaA.h---------------------------------
class implementadorA : public implementador {
public:
	void operacion() {
		cout << "Esta es la implementacion A" << endl;
	}
};
//----------------------------implementaB.h---------------------------------

class implementadorB : public implementador {
public:
	void operacion() {
		cout << "Esta es una implementacion de B" << endl;
	}
};

//------------------------ abstraccion.h-----------------------------------
#include "implementador.h"

class  abstraccion {
protected:
	implementador* imple;
public:
	virtual void operacion() = 0;
};
//------------------------ redefinición.h-----------------------------------


class redefinicion : public abstraccion {

public:
	redefinicion(implementador* p) {
		imple = p;
	}
	void operacion() {
		imple->operacion();
	}
};







//----------------------------main.cpp---------------------------------------

#include "abstraccion.h"
#include "implementador.h"

int main() {
	redefinicion* imple1 = new redefinicion(new implementadorA());
	redefinicion* imple2 = new redefinicion(new implementadorB());
	imple1->operacion();
	imple2->operacion();
	delete	imple1;
```

```
        delete     imple2;
        cin.get();
        return 0;
}
```

El patrón permite:

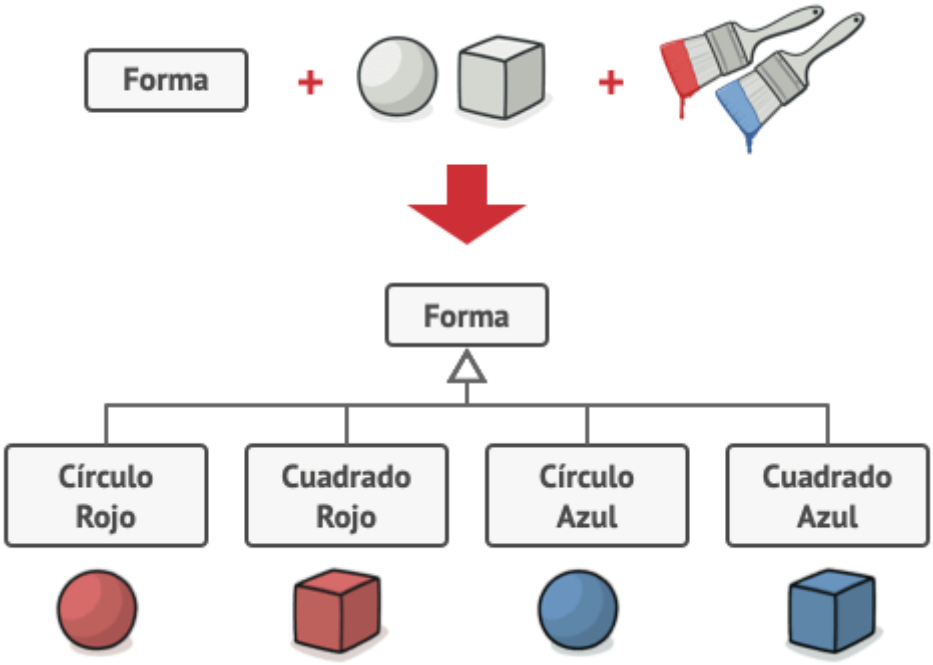
- ***Desacoplar interfaz e implementación*:** una implementación no es limitada permanentemente a una interfaz. La implementación de una abstracción puede ser configurada en tiempo de ejecución. Además le es posible a un objeto cambiar su implementación en tiempo de ejecución.
- Desacoplando ***Abstracción*** e ***Implementador*** también elimina las dependencias sobre la implementación en tiempo de compilación. Cambiar una clase de implementación no requiere recompilar la clase ***Abstraccion*** ni sus clientes.
- Mejora la extensibilidad: se puede extender las jerarquías de Abstraction e Implementor independientemente.
- Esconde los detalles de la implementación a los clientes.

Consecuencias:

- **POSITIVAS:**
  - Una implementación no se limita permanentemente a una interface.
  - La implementación de una abstracción puede ser configurada y/o cambiada en tiempo de ejecución.
  - Desacoplando **Abstraction** e **Implementor** también se eliminan las dependencias sobre la implementación en tiempo de compilación.
  - Cambiar una implementación no requiere recompilar la clase **Abstraction** ni sus clientes.
  - Las capas de alto nivel de un sistema sólo tiene que conocer Abstraction e Implementor.
  - Se pueden extender las jerarquías de Abstraction e Implementor sin que haya dependencias.
  - Oculta los detalles de implementación a los clientes.
- **NEGATIVAS:**
  - Puede ser complicado de entender al principio.
  - Añade complejidad.
  - Problemática al no entender bien su funcionamiento.

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`.



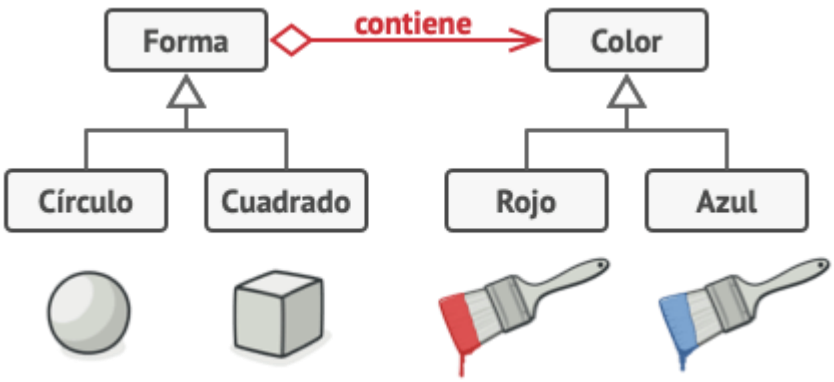
El número de combinaciones de clase crece exponencialmente.

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 mas 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.

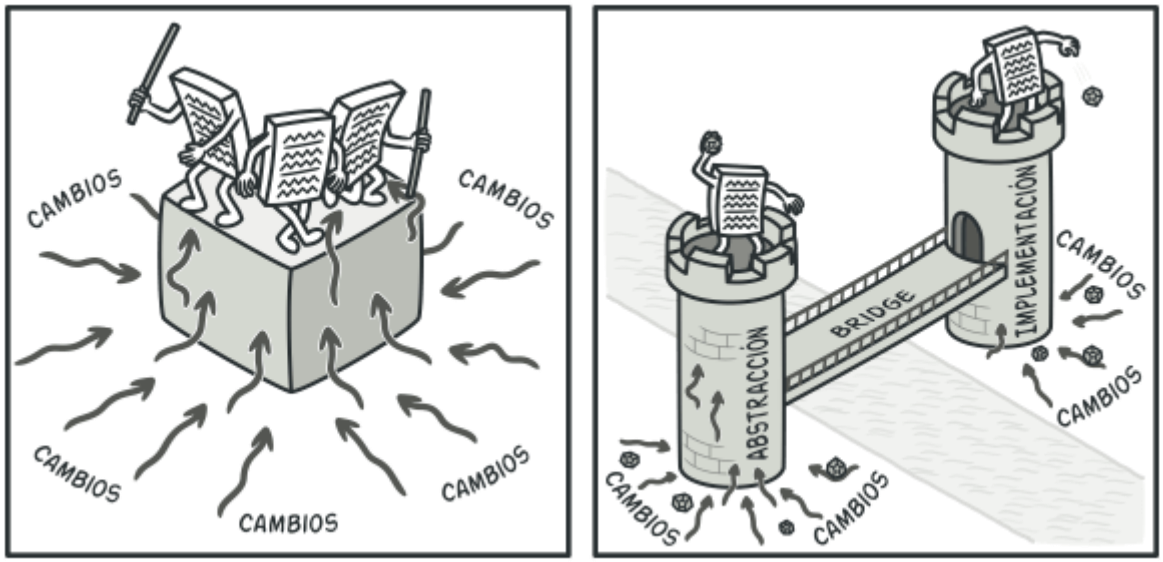


Con esta solución, podemos extraer el código relacionado con el color y colocarlo dentro de su propia jerarquía, con dos subclases: `Rojo` y `Azul`. La clase `Forma` obtiene entonces una referencia

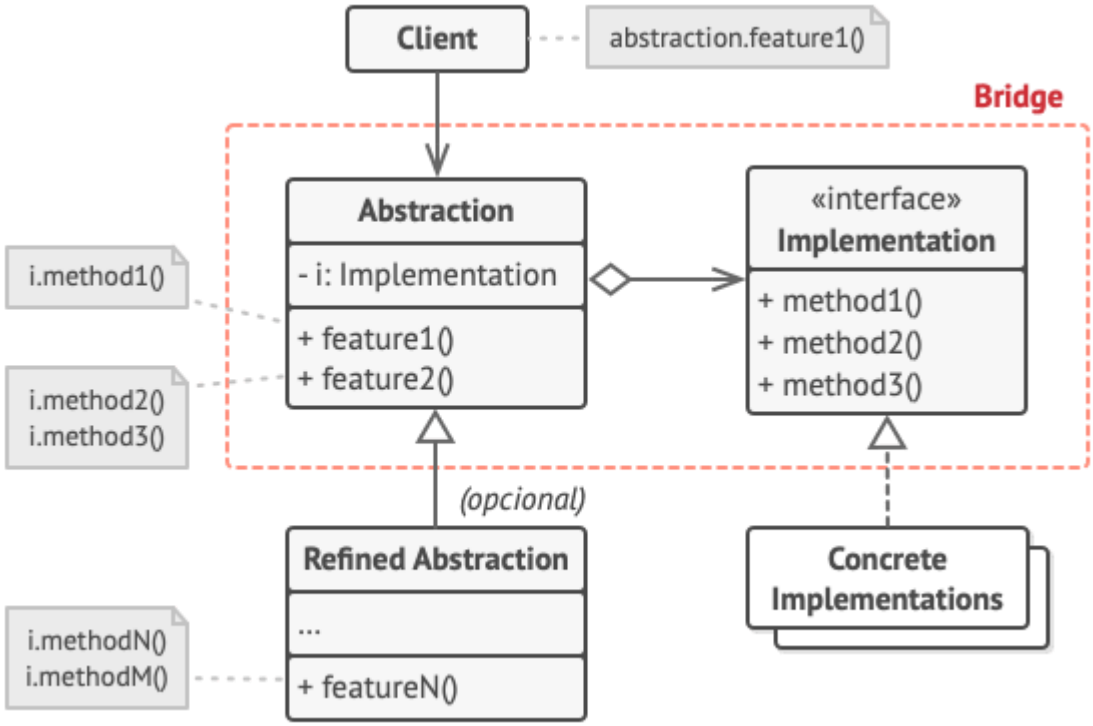
que apunta a uno de los objetos de color. Ahora la forma puede delegar cualquier trabajo relacionado con el color al objeto de color vinculado. Esa referencia actuará como un puente entre las clases `Forma` y `Color`. En adelante, añadir nuevos colores no exigirá cambiar la jerarquía de forma y viceversa.

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*** . Ten en cuenta que no estamos hablando de las *interfaces* o las *clases abstractas* de tu lenguaje de programación. Son cosas diferentes.



Estructura



1-La Abstracción ofrece lógica de control de alto nivel. Depende de que el objeto de la implementación haga el trabajo de bajo nivel.

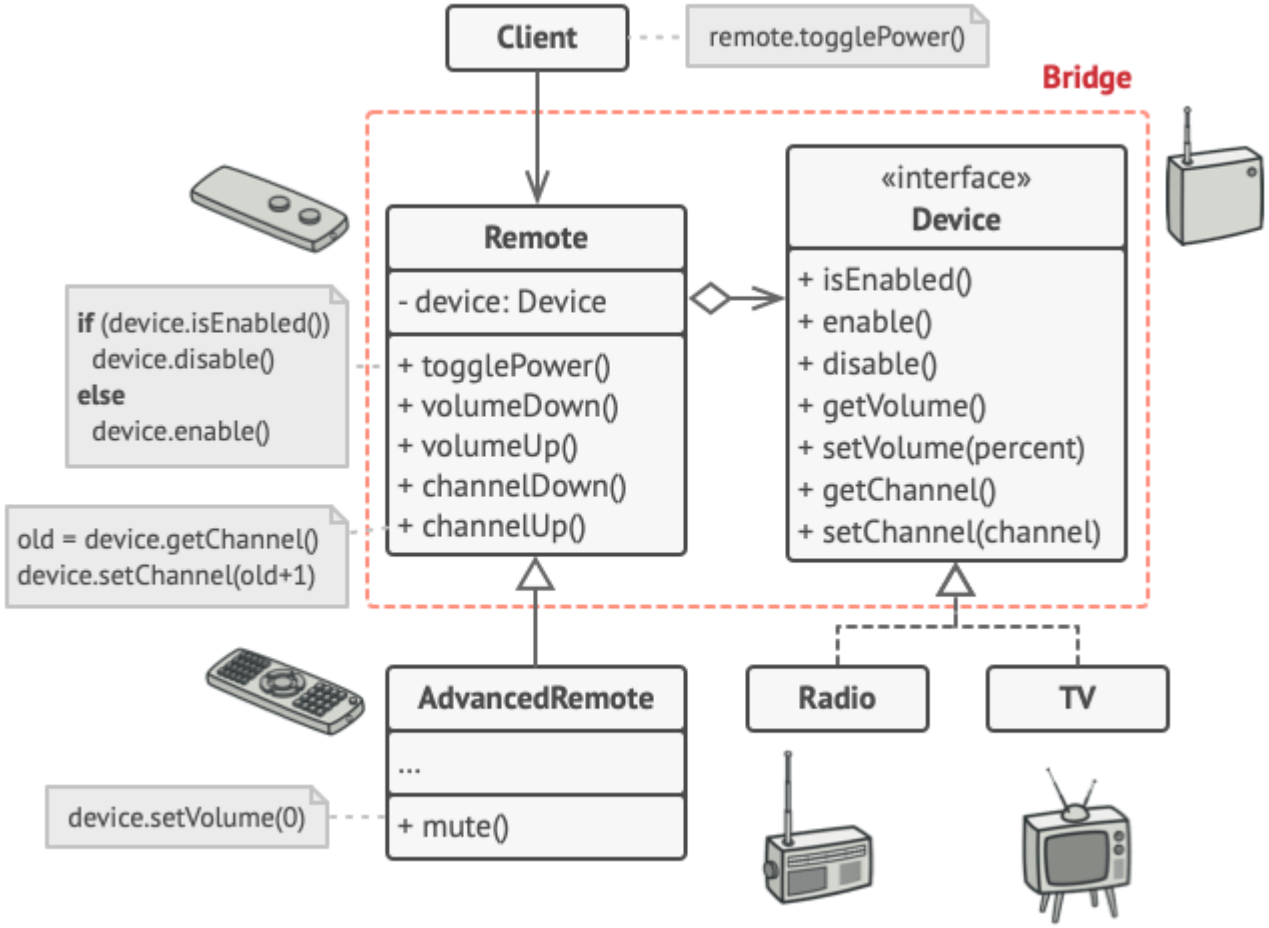
2-La Implementación declara la interfaz común a todas las implementaciones concretas. Una abstracción sólo se puede comunicar con un objeto de implementación a través de los métodos que se declaren aquí.

3-Las Implementaciones Concretas contienen código específico

4-Las Abstracciones Refinadas proporcionan variantes de lógica de control. Como sus padres, trabajan con distintas implementaciones a través de la interfaz general de implementación.

5-Normalmente, el Cliente sólo está interesado en trabajar con la abstracción. No obstante, el cliente tiene que vincular el objeto de la abstracción con uno de los objetos de la implementación.

Este ejemplo ilustra cómo puede ayudar el patrón **Bridge** a dividir el código de una aplicación que gestiona dispositivos y sus controles remotos. Las clases `Dispositivo` actúan como implementación, mientras que las clases `Remoto` actúan como abstracción.



La jerarquía de clase original se divide en dos partes: dispositivos y controles remotos.
La clase base de control remoto declara un campo de referencia que la vincula con un objeto de dispositivo. Todos los controles remotos funcionan con los dispositivos a través de la interfaz general de dispositivos, que permite al mismo remoto soportar varios tipos de dispositivos.

Puedes desarrollar las clases de control remoto independientemente de las clases de dispositivo. Lo único necesario es crear una nueva subclase de control remoto. Por ejemplo, puede ser que un control remoto básico cuente tan solo con dos botones, pero puedes extenderlo añadiéndole funciones, como una batería adicional o pantalla táctil.

El código cliente vincula el tipo deseado de control remoto con un objeto específico de dispositivo a través del constructor del control remoto.

Estrategia versus Puente

El patrón Puente es un patrón estructural (¿CÓMO SE CONSTRUYE UN COMPONENTE DE SOFTWARE?).
El patrón Estrategia es un patrón dinámico (¿CÓMO QUIERES EJECUTAR UN COMPORTAMIENTO EN EL SOFTWARE?).

El diagrama de clases UML para el patrón Estrategia es el mismo que el diagrama del patrón el patrón Puente. Sin embargo, estos dos patrones de diseño no son iguales en su intención. Mientras que el patrón Strategy está pensado para el comportamiento, el patrón Bridge está pensado para la estructura.

El acoplamiento entre el contexto y las estrategias es más estrecho que el acoplamiento entre la abstracción y la implementación en el patrón Bridge de la abstracción y la implementación.

Se utiliza el patrón de estrategia cuando se abstrae el comportamiento que podría ser proporcionado por una fuente externa (por ejemplo, la configuración podría especificar que se cargue algún ensamblaje de plugin), y se utiliza el patrón de puente cuando se utilizan las mismas construcciones para hacer el código un poco más ordenado. El código real será muy similar - sólo estás aplicando los patrones por razones ligeramente diferentes.

Fuentes:

https://refactoring.guru/es/design-patterns/bridge
https://www.youtube.com/watch?v=6bIHhzqMdgg