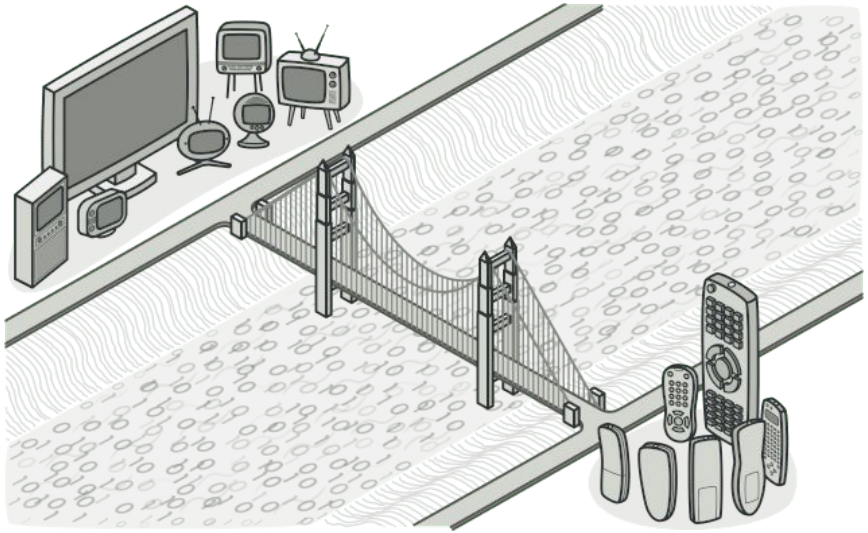
Patrón de Diseño Bridge

Que es?

El patrón de diseño Bridge es un patrón estructural que permite separar la implementación de una clase de su interfaz, de modo que ambos puedan variar de forma independiente.

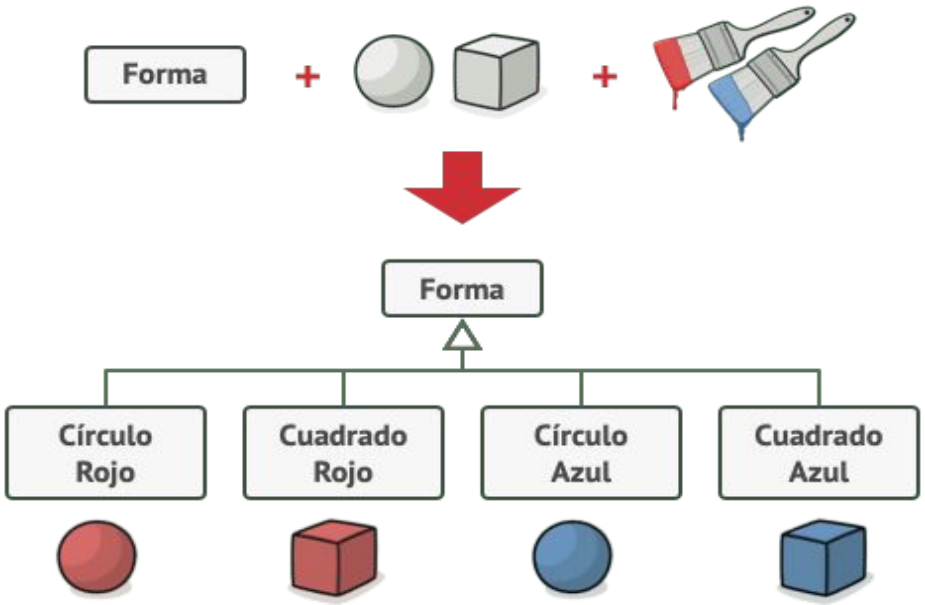
Esto permite una mayor flexibilidad y reutilización del código, y es muy útil cuando se trabaja con clases que tienen muchas variantes y que pueden ser combinadas de diferentes maneras.

En Kotlin, el patrón Bridge se implementa mediante la creación de una interfaz que define la funcionalidad de la clase, y una clase abstracta que implementa dicha interfaz y que contiene una referencia a un objeto de otra clase que implementa la implementación concreta de la funcionalidad.



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.

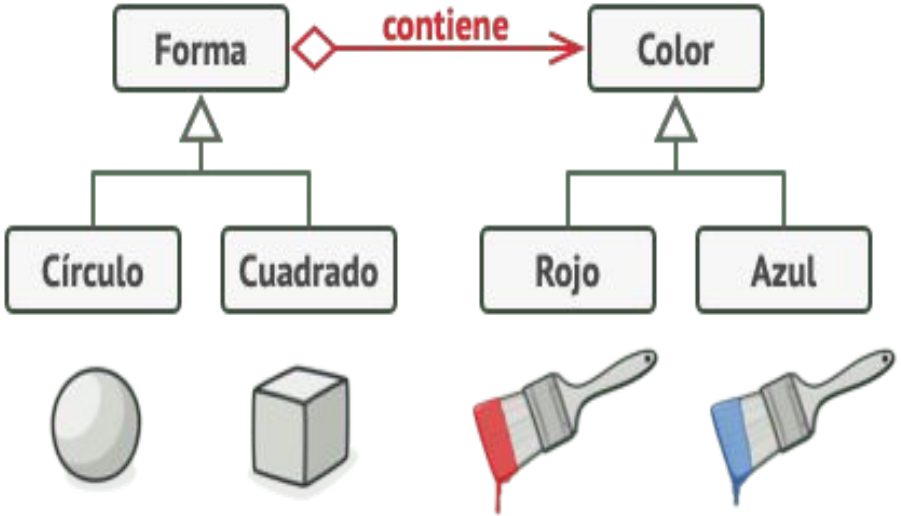


Añadir nuevos tipos de forma y color a la jerarquía hará que ésta crezca exponencialmente. Por ejemplo, para añadir una forma de triángulo deberás introducir dos subclases, una para cada color. Y, después, para añadir un nuevo color habrá que crear tres subclases, una para cada tipo de forma. Cuanto más avancemos, peor será.

Solución

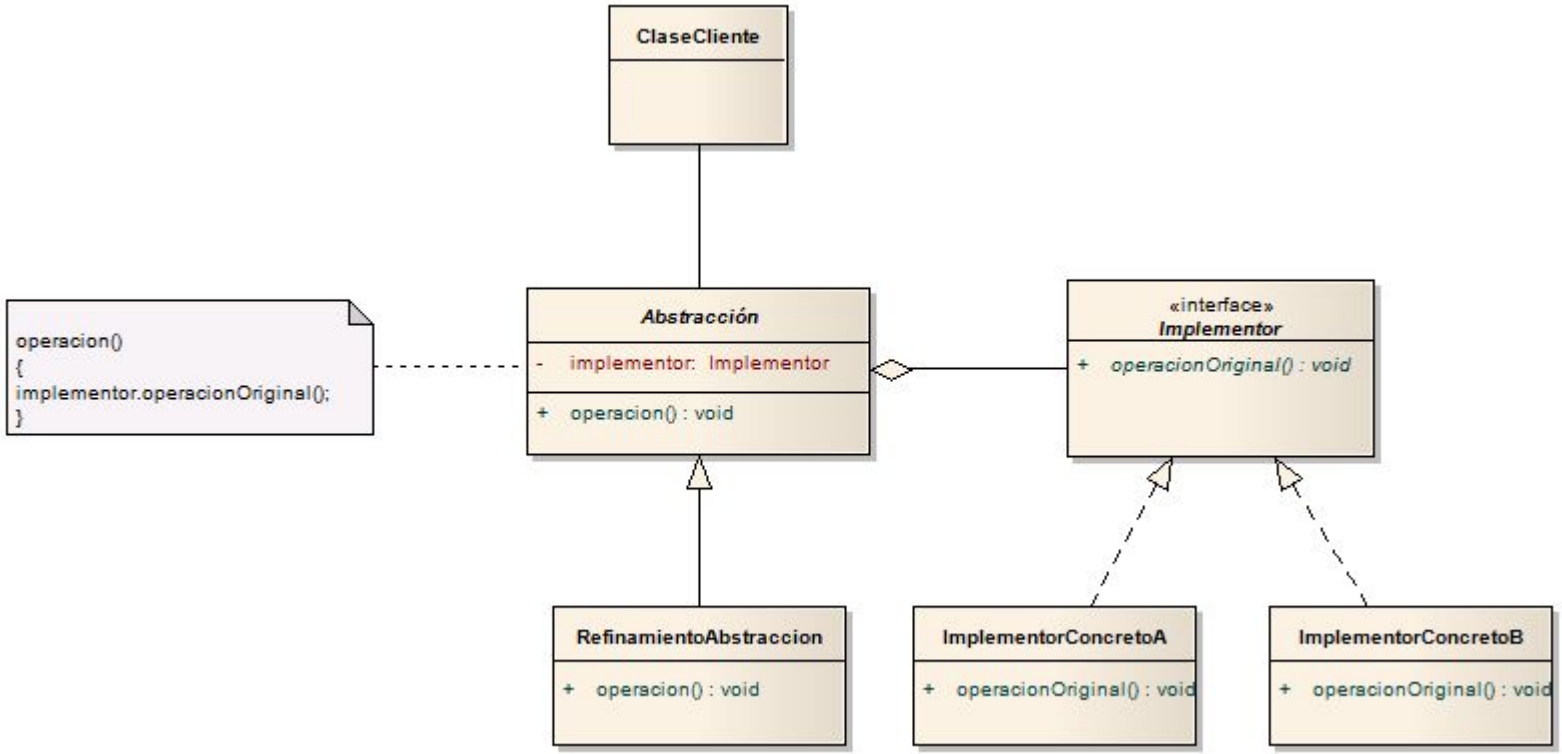
Este problema se presenta porque intentamos extender las clases de forma en dos dimensiones independientes: por forma y por color. Es un problema muy habitual en la herencia de clases.

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.



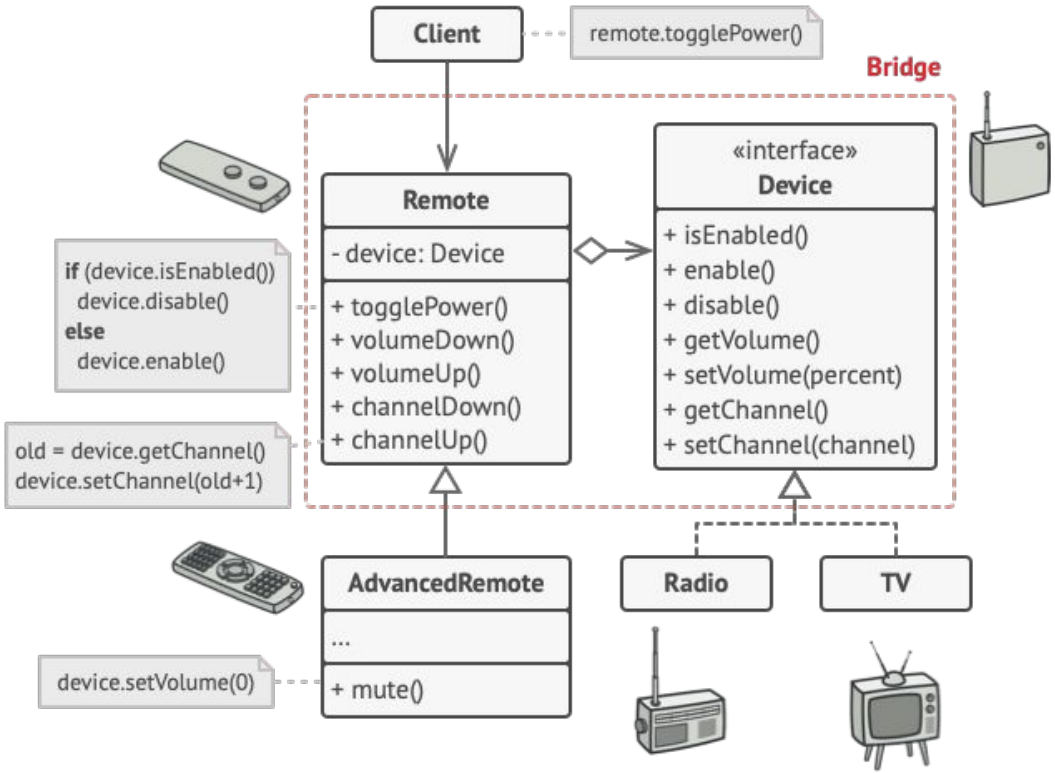
Con esta solución, podemos extraer el código relacionado con el color y colocarlo dentro de su propia clase, con dos subclases: Rojo y Azul. La clase Forma obtiene entonces un campo de referencia que apunta a uno de los objetos de color. Ahora la forma puede delegar cualquier trabajo relacionado con 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.

Estructura


ClaseCliente
operacion()
{
implementor.operacionOriginal();
}
Abstracción
- implementor: Implementor
+ operacion() : void
«interface»
Implementor
+ operacionOriginal() : void
RefinamientoAbstraccion
+ operacion() : void
ImplementorConcretoA
+ operacionOriginal() : void
ImplementorConcretoB
+ operacionOriginal() : void

Ejemplo

Este ejemplo ilustra cómo puede ayudar el patrón Bridge a dividir el código monolítico 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 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 añadiendo 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.

Ejemplo

devices/Device.java: Interfaz común de todos los dispositivos

```
package refactoring_guru.bridge.example.devices;

public interface Device {
    boolean isEnabled();

    void enable();

    void disable();

    int getVolume();

    void setVolume(int percent);

    int getChannel();

    void setChannel(int channel);

    void printStatus();
}
```

devices/Radio.java: Radio

```
package refactoring_guru.bridge.example.devices;

public class Radio implements Device {
    private boolean on = false;
    private int volume = 30;
    private int channel = 1;

    @Override
    public boolean isEnabled() {
        return on;
    }

    @Override
    public void enable() {
        on = true;
    }

    @Override
    public void disable() {
        on = false;
    }

    @Override
    public int getVolume() {
        return volume;
    }

    @Override
    public void setVolume(int volume) {
        if (volume > 100) {
            this.volume = 100;
        } else if (volume < 0) {
            this.volume = 0;
        } else {
            this.volume = volume;
        }
    }

    @Override
    public int getChannel() {
        return channel;
    }

    @Override
    public void setChannel(int channel) {
        this.channel = channel;
    }

    @Override
    public void printStatus() {
        System.out.println("--------------------------------------");
        System.out.println("| I'm radio.");
        System.out.println("| I'm " + (on ? "enabled" : "disabled"));
        System.out.println("| Current volume is " + volume + "%");
        System.out.println("| Current channel is " + channel);
        System.out.println("--------------------------------------\n");
    }
}
```

Ejemplo

devices/Tv.java: TV

```
package refactoring_guru.bridge.example.devices;

public class Tv implements Device {
    private boolean on = false;
    private int volume = 30;
    private int channel = 1;

    @Override
    public boolean isEnabled() {
        return on;
    }

    @Override
    public void enable() {
        on = true;
    }

    @Override
    public void disable() {
        on = false;
    }

    @Override
    public int getVolume() {
        return volume;
    }

    @Override
    public void setVolume(int volume) {
        if (volume > 100) {
            this.volume = 100;
        } else if (volume < 0) {
            this.volume = 0;
        } else {
            this.volume = volume;
        }
    }

    @Override
    public int getChannel() {
        return channel;
    }

    @Override
    public void setChannel(int channel) {
        this.channel = channel;
    }

    @Override
    public void printStatus() {
        System.out.println("------------------------------------");
        System.out.println("| I'm TV set.");
        System.out.println("| I'm " + (on ? "enabled" : "disabled"));
        System.out.println("| Current volume is " + volume + "%");
        System.out.println("| Current channel is " + channel);
        System.out.println("------------------------------------\n");
    }
}
```

remotes/Remote.java: Interfaz común de todos los remotos

```
package refactoring_guru.bridge.example.remotes;

public interface Remote {
    void power();

    void volumeDown();

    void volumeUp();

    void channelDown();

    void channelUp();
}
```

Ejemplo

remotes/BasicRemote.java: Control remoto básico

```
package refactoring_guru.bridge.example.remotes;

import refactoring_guru.bridge.example.devices.Device;

public class BasicRemote implements Remote {
    protected Device device;

    public BasicRemote() {}

    public BasicRemote(Device device) {
        this.device = device;
    }

    @Override
    public void power() {
        System.out.println( "Remote: power toggle" );
        if (device.isEnabled()) {
            device.disable();
        } else {
            device.enable();
        }
    }

    @Override
    public void volumeDown() {
        System.out.println( "Remote: volume down" );
        device.setVolume(device.getVolume() -  10);
    }

    @Override
    public void volumeUp() {
        System.out.println( "Remote: volume up" );
        device.setVolume(device.getVolume() +  10);
    }

    @Override
    public void channelDown() {
        System.out.println( "Remote: channel down" );
        device.setChannel(device.getChannel() -  1);
    }

    @Override
    public void channelUp() {
        System.out.println( "Remote: channel up" );
        device.setChannel(device.getChannel() +  1);
    }
}
```

remotes/AdvancedRemote.java: Control remoto avanzado

```
package refactoring_guru.bridge.example.remotes;

import refactoring_guru.bridge.example.devices.Device;

public class AdvancedRemote extends BasicRemote {

    public AdvancedRemote(Device device) {

        super.device = device;

    }

    public void mute() {

        System.out.println("Remote: mute");

        device.setVolume(0);

    }
}
```

Ejemplo

Demo.java: Código cliente

```
package refactoring_guru.bridge.example;

import refactoring_guru.bridge.example.devices.Device;
import refactoring_guru.bridge.example.devices.Radio;
import refactoring_guru.bridge.example.devices.Tv;
import
refactoring_guru.bridge.example.remotes.AdvancedRemote;
import
refactoring_guru.bridge.example.remotes.BasicRemote;

public class Demo {
    public static void main(String[] args) {
        testDevice(new Tv());
        testDevice(new Radio());
    }

    public static void testDevice(Device device) {
        System.out.println("Tests with basic remote.");
        BasicRemote basicRemote =new BasicRemote(device);
        basicRemote.power();
        device.printStatus();

        System.out.println("Tests with advanced remote.");
        AdvancedRemote advancedRemote =new
AdvancedRemote(device);
        advancedRemote.power();
        advancedRemote.mute();
        device.printStatus();
    }
}
```

OutputDemo.txt: Resultado de la ejecución

```
Tests with basic remote.
Remote: power toggle
--------------------------------------
| I'm TV set.
| I'm enabled
| Current volume is 30%
| Current channel is 1
--------------------------------------

Tests with advanced remote.
Remote: power toggle
Remote: mute
--------------------------------------
| I'm TV set.
| I'm disabled
| Current volume is 0%
| Current channel is 1
--------------------------------------

Tests with basic remote.
Remote: power toggle
--------------------------------------
| I'm radio.
| I'm enabled
| Current volume is 30%
| Current channel is 1
--------------------------------------

Tests with advanced remote.
Remote: power toggle
Remote: mute
--------------------------------------
| I'm radio.
| I'm disabled
| Current volume is 0%
| Current channel is 1
--------------------------------------
```

Ejemplo

El patrón de diseño Bridge se utiliza para separar una abstracción de su implementación, lo que permite que ambas evolucionen de forma independiente. En este caso, la abstracción se representa mediante las interfaces Devicey Remote, y la implementación se encuentra en las clases concretas Tv, Radio, BasicRemote y AdvancedRemote.

Explicación del código:

Device(Dispositivo): Es una interfaz común que define las operaciones básicas que un dispositivo puede realizar, como habilitar/deshabilitar, ajustar el volumen y cambiar de canal.

Tv(Televisión) y Radio: Son clases concretas que implementan la interfaz Device. Cada una de ellas tiene su propia implementación de las operaciones definidas en Device, incluyendo el estado (encendido/apagado, volumen, canal) y cómo se imprimen los detalles del estado.

Remote(Control remoto): Es otra interfaz común que define las operaciones básicas que un control remoto puede realizar, como encender/apagar, ajustar el volumen y cambiar de canal.

BasicRemote (Control remoto básico): Es una clase que implementa la interfaz Remote y tiene un campo que almacena una referencia al dispositivo (Device) con el que está emparejado. Implemente las operaciones del control remoto básico, como encender/apagar el dispositivo y ajustar el volumen y el canal.

AdvancedRemote (Control remoto avanzado): Es una clase que extiende BasicRemote y agrega una operación adicional para silenciar el dispositivo.

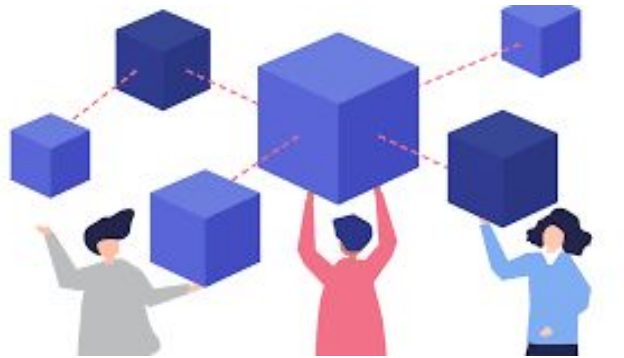
Demo.java: Este es el código cliente que demuestra cómo se pueden usar los dispositivos y los controles remotos. Crea instancias de Tv y Radio, y luego prueba el control remoto básico y el control remoto avanzado con estos dispositivos. El resultado de la prueba se imprime en la consola.

OutputDemo.txt: Este archivo muestra la salida de la ejecución del programa, que demuestra cómo funcionan los controles remotos con los dispositivos. En la salida, se puede ver cómo se encienden y apagan los dispositivos, se ajusta el volumen y se cambian los canales.

El patrón Bridge permite que las clases Device y Remote evolucionen de manera independiente. Si necesitas agregar nuevos tipos de dispositivos o controles remotos en el futuro, puedes hacerlo sin modificar las clases existentes. El patrón Bridge también facilita la reutilización de las implementaciones, ya que puedes emparejar diferentes tipos de dispositivos con diferentes controles remotos de manera flexible.

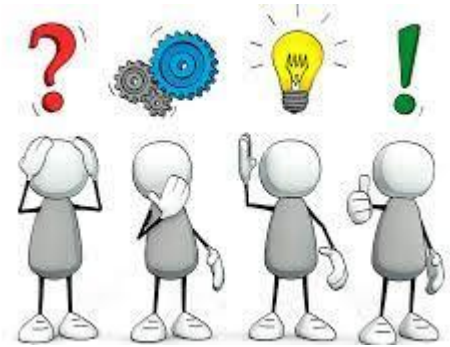
Aplicabilidad

- 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).
- Conforme más crece una clase, más difícil resulta entender cómo funciona y más tiempo se tarda en realizar un cambio. Cambiar una de las variaciones de funcionalidad puede exigir realizar muchos cambios a toda la clase, lo que a menudo provoca que se cometan errores o no se aborden algunos de los efectos colaterales críticos.
- El patrón Bridge te permite dividir la clase monolítica en varias jerarquías de clase. Después, puedes cambiar las clases de cada jerarquía independientemente de las clases de las otras. Esta solución simplifica el mantenimiento del código y minimiza el riesgo de descomponer el código existente.
- Utiliza el patrón cuando necesites extender una clase en varias dimensiones ortogonales (independientes).
- El patrón Bridge sugiere que extraigas una jerarquía de clase separada para cada una de las dimensiones. La clase original delega el trabajo relacionado a los objetos pertenecientes a dichas jerarquías, en lugar de hacerlo todo por su cuenta.
- Utiliza el patrón Bridge cuando necesites poder cambiar implementaciones durante el tiempo de ejecución.
- Aunque es opcional, el patrón Bridge te permite sustituir el objeto de implementación dentro de la abstracción. Es tan sencillo como asignar un nuevo valor a un campo.
- Por cierto, este último punto es la razón principal por la que tanta gente confunde el patrón Bridge con el patrón Strategy. Recuerda que un patrón es algo más que un cierto modo de estructurar tus clases. También puede comunicar intención y el tipo de problema que se está abordando.



Cómo implementarlo

1. Identifica las dimensiones ortogonales de tus clases. Estos conceptos independientes pueden ser: abstracción/plataforma, dominio/infraestructura, *front end*/*back end*, o interfaz/implementación.
2. Comprueba qué operaciones necesita el cliente y defínelas en la clase base de abstracción.
3. Determina las operaciones disponibles en todas las plataformas. Declara aquellas que necesite la abstracción en la interfaz general de implementación.
4. Crea clases concretas de implementación para todas las plataformas de tu dominio, pero asegúrate de que todas sigan la interfaz de implementación.
5. Dentro de la clase de abstracción añade un campo de referencia para el tipo de implementación. La abstracción delega la mayor parte del trabajo al objeto de la implementación referenciado en ese campo.
6. Si tienes muchas variantes de lógica de alto nivel, crea abstracciones refinadas para cada variante extendiendo la clase base de abstracción.
7. El código cliente debe pasar un objeto de implementación al constructor de la abstracción para asociar el uno con el otro. Después, el cliente puede ignorar la implementación y trabajar solo con el objeto de la abstracción.



Pros y contras

- ✓ Puedes crear clases y aplicaciones independientes de plataforma.
- ✓ El código cliente funciona con abstracciones de alto nivel. No está expuesto a los detalles de la plataforma.
- ✓ *Principio de abierto/cerrado*. Puedes introducir nuevas abstracciones e implementaciones independientes entre sí.
- ✓ *Principio de responsabilidad única*. Puedes centrarte en la lógica de alto nivel en la abstracción y en detalles de la plataforma en la implementación.
- ✗ Puede ser que el código se complique si aplicas el patrón a una clase muy cohesionada.