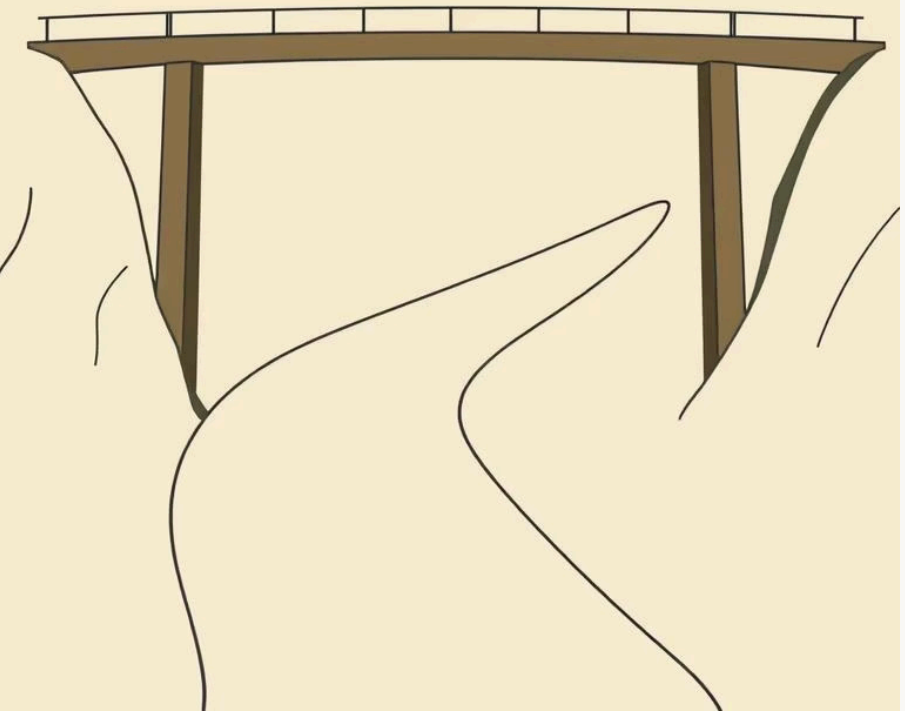

Patrón de Diseño Bridge: Separando Abstracción e Implementación

El patrón Bridge es un patrón estructural que divide una clase compleja en dos jerarquías independientes: la Abstracción y la Implementación. Este enfoque permite que ambas jerarquías varíen de forma independiente, facilitando la extensión y el mantenimiento del código a largo plazo. Es especialmente útil en sistemas que necesitan soportar múltiples plataformas, variantes o APIs, como diferentes bases de datos o servicios externos.

¿Cómo funciona el patrón Bridge?

Abstracción

Define la interfaz de alto nivel para el cliente. Mantiene una referencia a un objeto de la jerarquía de Implementación. Delega las operaciones a este objeto de implementación.

Implementación

Declara una interfaz común para todas las implementaciones concretas. Estas implementaciones pueden variar según la plataforma, el entorno o las necesidades específicas.

Delegación

La Abstracción delega sus operaciones a la Implementación. Esto permite cambiar la implementación subyacente sin alterar la abstracción ni el código cliente que la utiliza.

Beneficio Clave

Evita la "explosión de clases" que ocurre cuando se intentan combinar múltiples variantes de abstracciones y implementaciones en una única jerarquía de herencia.

La flexibilidad del patrón Bridge radica en su capacidad para desacoplar completamente la interfaz (Abstracción) de su implementación concreta. Esto significa que podemos añadir nuevas abstracciones o nuevas implementaciones sin afectar a las existentes, fomentando un diseño más modular y adaptable.

Ejemplo: Bridge con Abstracción e Implementaciones

Definiendo la Implementación

Comenzamos con la interfaz de Implementación y sus clases concretas. Aquí, la Implementación define cómo se dibuja una forma, adaptándose a diferentes APIs de dibujo (por ejemplo, una API de consola y una API gráfica).

```
# Interfaz de Implementación
class DibujoAPI:
    def dibuja_circulo(self, x, y, radio):
        pass

# Implementaciones Concretas
class DibujoAPIConsole(DibujoAPI):
    def dibuja_circulo(self, x, y, radio):
        print(f"Dibujando círculo en consola: ({x},{y}), radio
{radio}")

class DibujoAPIGrafica(DibujoAPI):
    def dibuja_circulo(self, x, y, radio):
        print(f"Dibujando círculo gráfico: ({x},{y}), radio
{radio}")
```

Definiendo la Abstracción

A continuación, creamos la Abstracción, que es la interfaz de alto nivel que los clientes utilizarán. Esta abstracción contiene una referencia a un objeto de la implementación y delega las llamadas a este.

```
# Abstracción
class Forma:
    def __init__(self, dibujo_api):
        self._dibujo_api = dibujo_api

    def dibuja(self):
        pass

# Abstracciones Refinadas
class Circulo(Forma):
    def __init__(self, x, y, radio, dibujo_api):
        super().__init__(dibujo_api)
        self._x = x
        self._y = y
        self._radio = radio

    def dibuja(self):
        self._dibujo_api.dibuja_circulo(self._x, self._y,
self._radio)

# Uso del patrón Bridge
api_console = DibujoAPIConsole()
api_grafica = DibujoAPIGrafica()

circulo1 = Circulo(1, 2, 3, api_console)
circulo1.dibuja()

circulo2 = Circulo(5, 7, 10, api_grafica)
circulo2.dibuja()
```

En este ejemplo, la clase `Forma` (y `Circulo`) es la Abstracción, mientras que `DibujoAPI` y sus subclases son la Implementación. Podemos cambiar entre dibujar en consola o gráficamente sin modificar la clase `Circulo`.