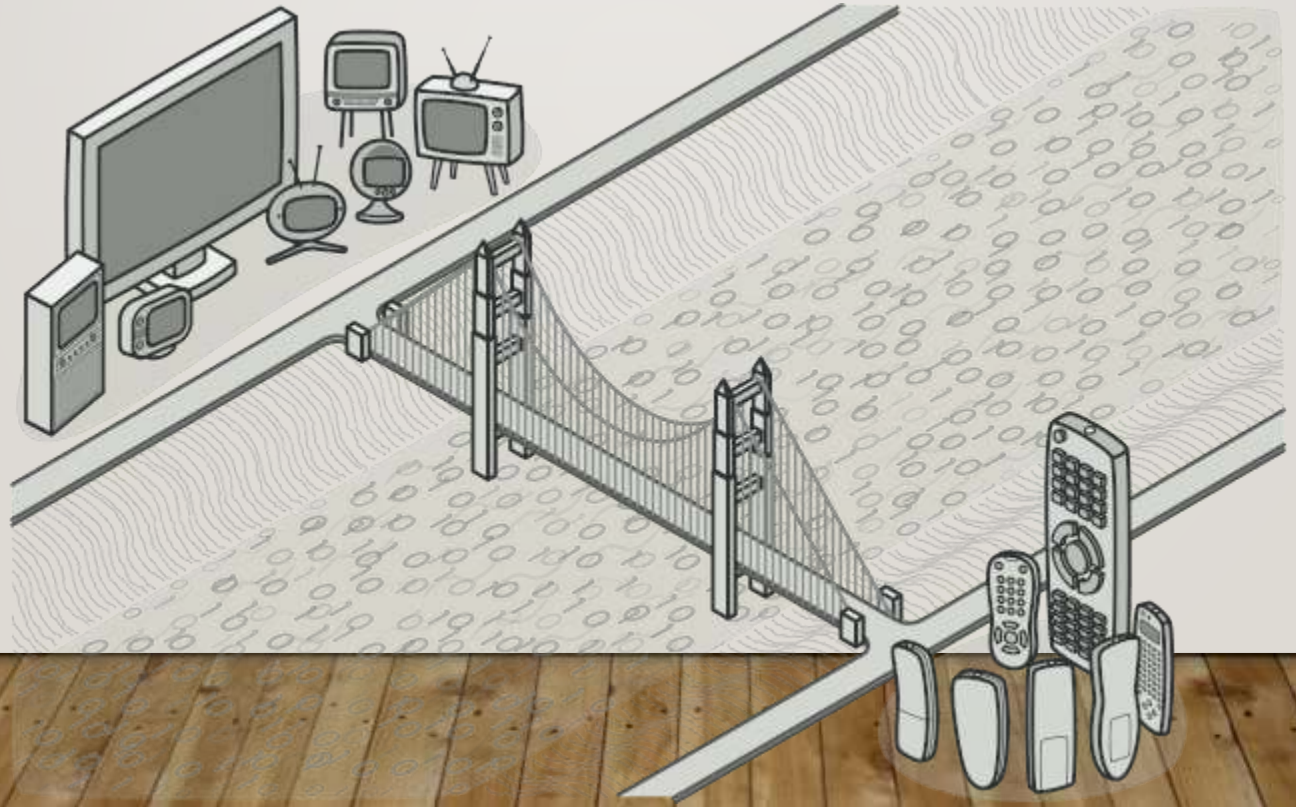
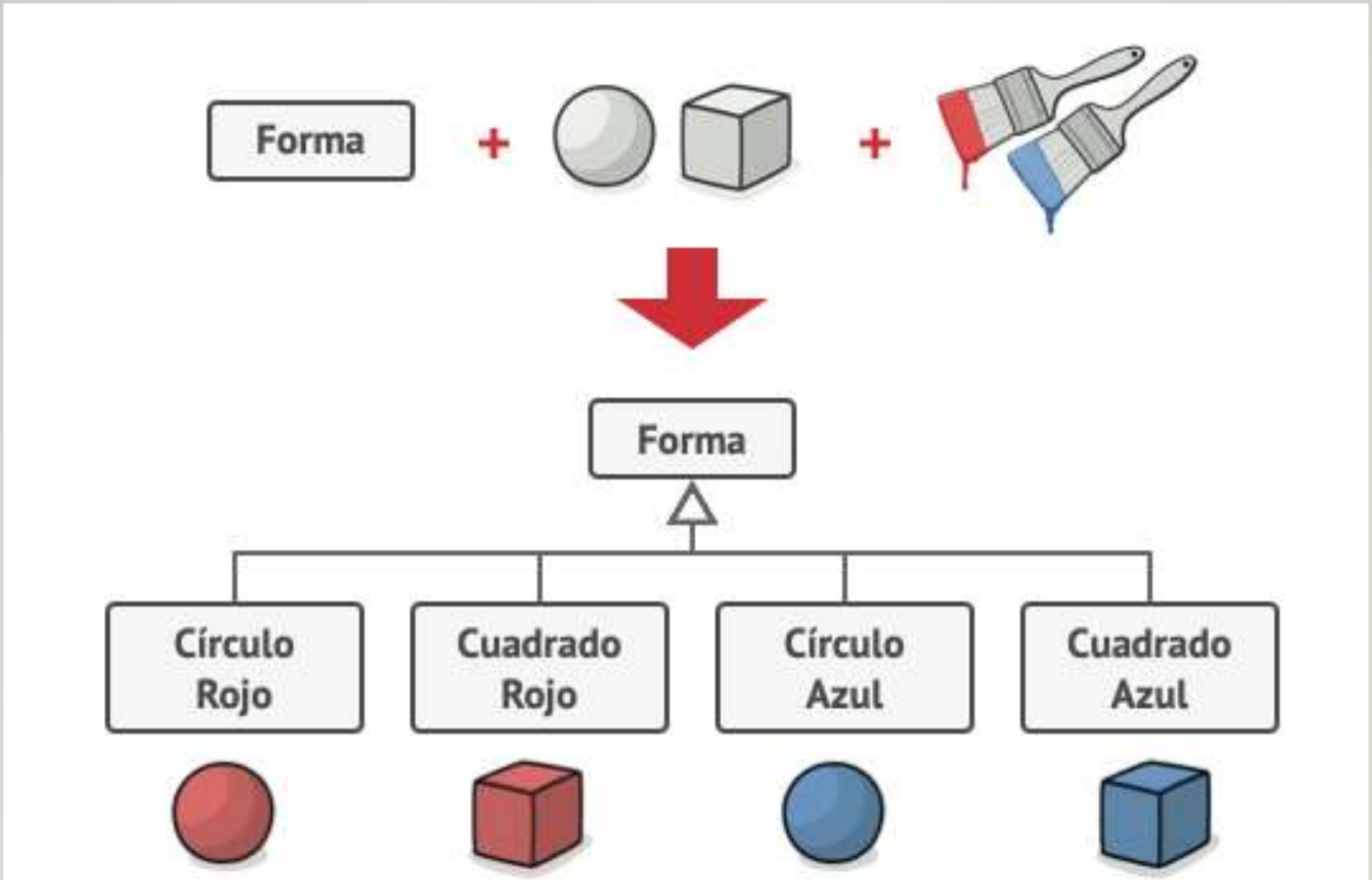
PATRONES DE DISEÑO

Daniel Eduardo Ortiz Celis 2171469

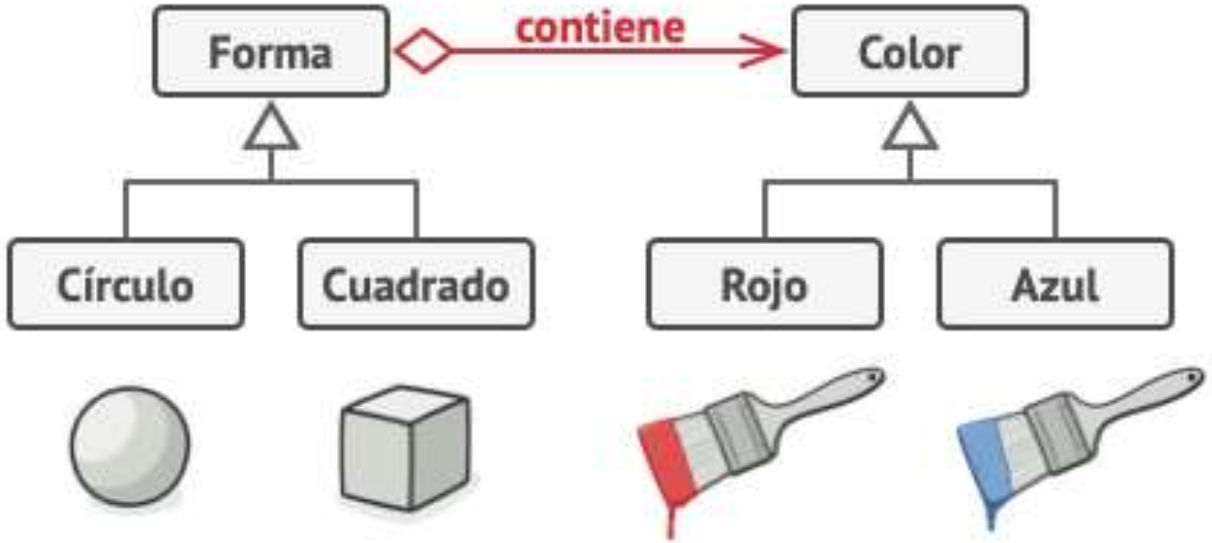
PATRÓN BRIDGE

- Patrón de diseño estructural
- 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.





El número de combinaciones de clase crece en progresión geométrica.



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

PROBLEMA

Tenemos la necesidad de 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:

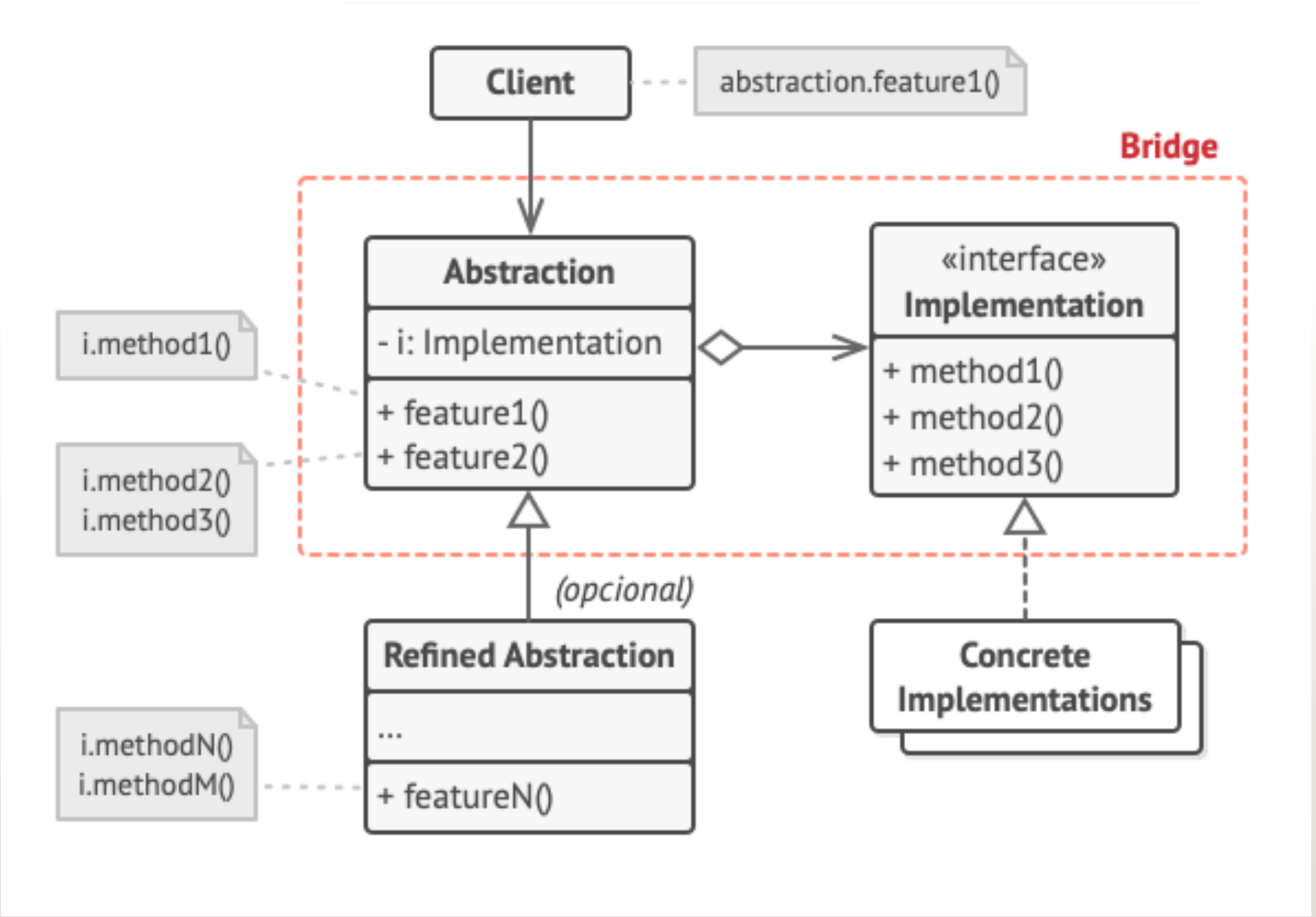
- Se quiere evitar enlaces permanentes entre una abstracción y una implementación.
- Tanto las abstracciones como las implementaciones deben ser extensibles por medio de subclases.
- Se quiere que los cambios en la implementación de una abstracción no afecten al cliente.
- Necesidad de que la implementación de una característica sea compartida entre múltiples objetos.

SOLUCIÓN

Se parte 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, se tiene 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.

Por último se debe 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).


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()

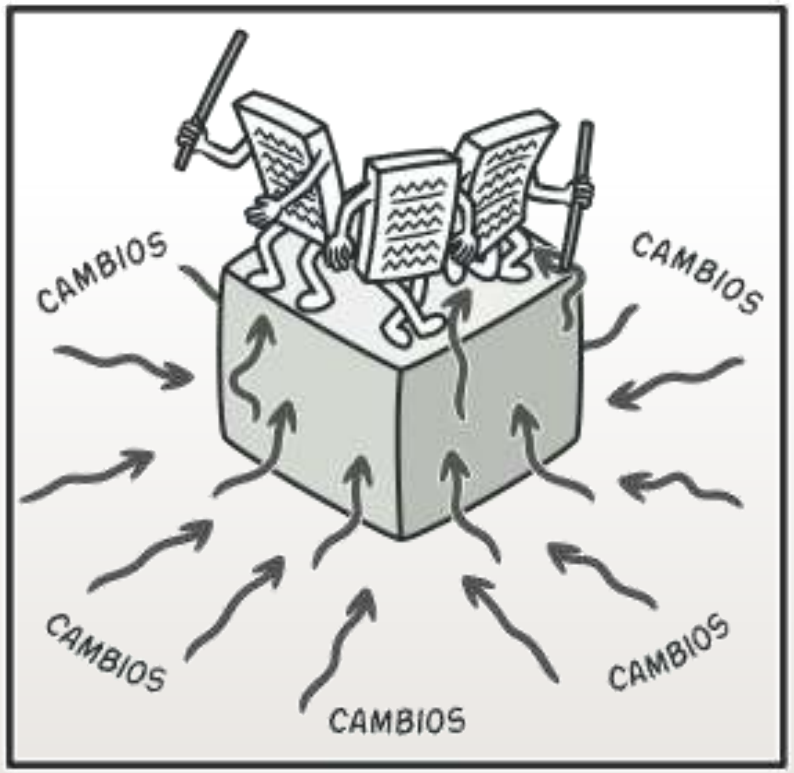
ABSTRACCIÓN E IMPLEMENTACIÓN

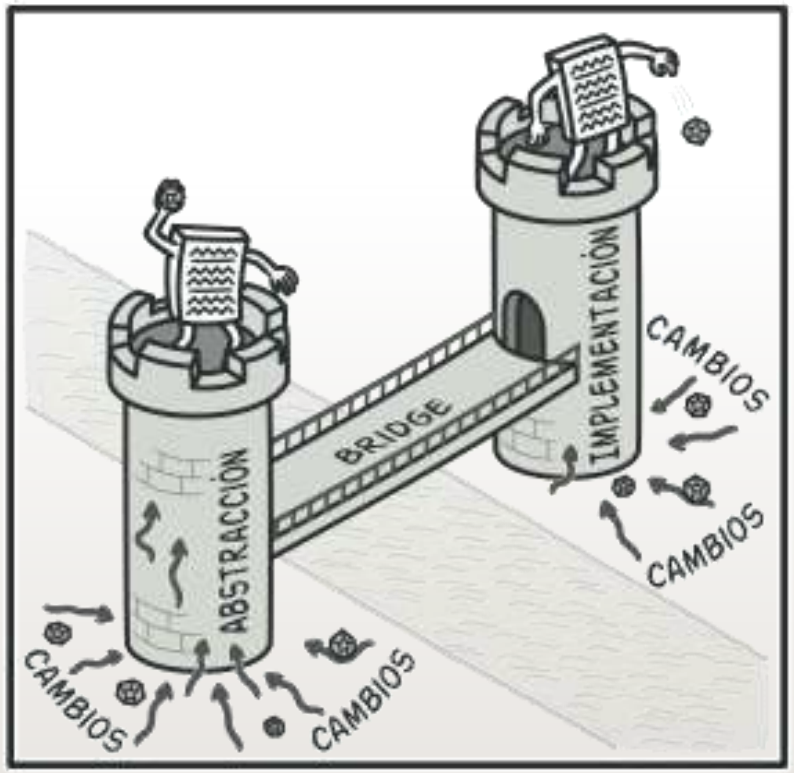
La *Abstracción* (también llamada *interfaz*) es una capa de control de alto nivel para una entidad. Esta capa no tiene que hacer ningún trabajo real por su cuenta, sino que debe delegar el trabajo a la capa de *implementación* (también llamada *plataforma*).

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.

Se puede extender esa aplicación en dos direcciones independientes:

- Tener varias GUI diferentes
- Soportar varias API





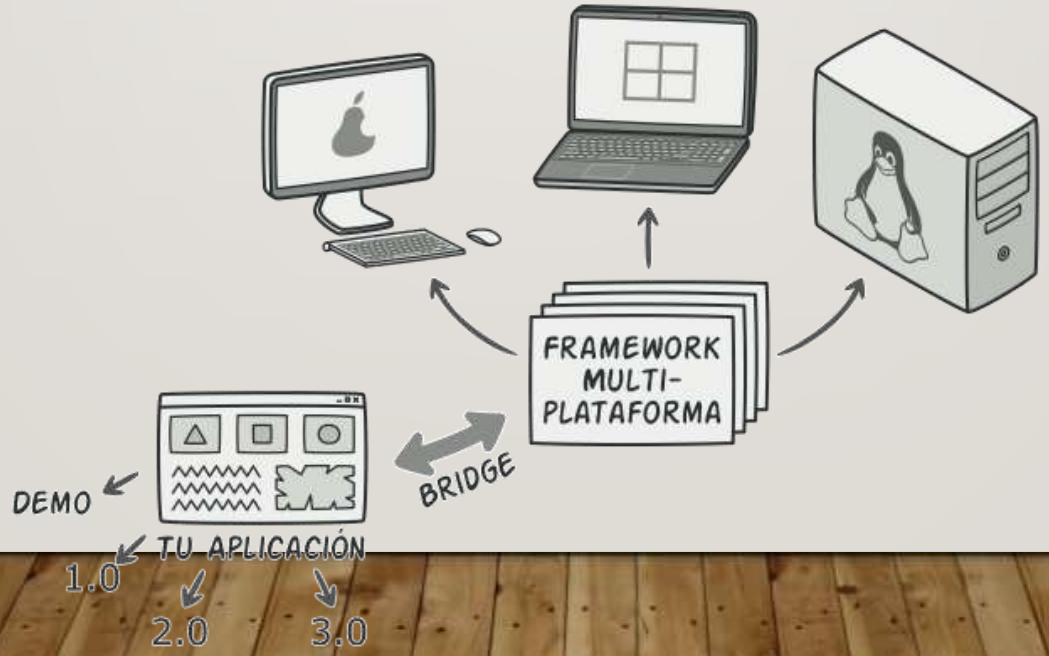
Realizar incluso un cambio sencillo en una base de código monolítica es bastante difícil porque se debe comprender todo el asunto muy bien. Es mucho más sencillo realizar cambios en módulos más pequeños y bien definidos.

La aplicación puede convertirse en un revoltijo de cosas, en el que cientos de condicionales conectan distintos tipos de GUI con varias API por todo el código. Se puede 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.

Con el patrón bridge podemos plantear una solución, donde dividimos las clases en dos jerarquías:

•Abstracción: la capa GUI de la aplicación.

•Implementación: las API de los sistemas operativos.



El objeto de la abstracción controla la apariencia de la aplicación, delegando el trabajo real al objeto de la implementación vinculado. Las distintas implementaciones son intercambiables siempre y cuando sigan una interfaz común, permitiendo a la misma GUI funcionar con Windows y Linux.

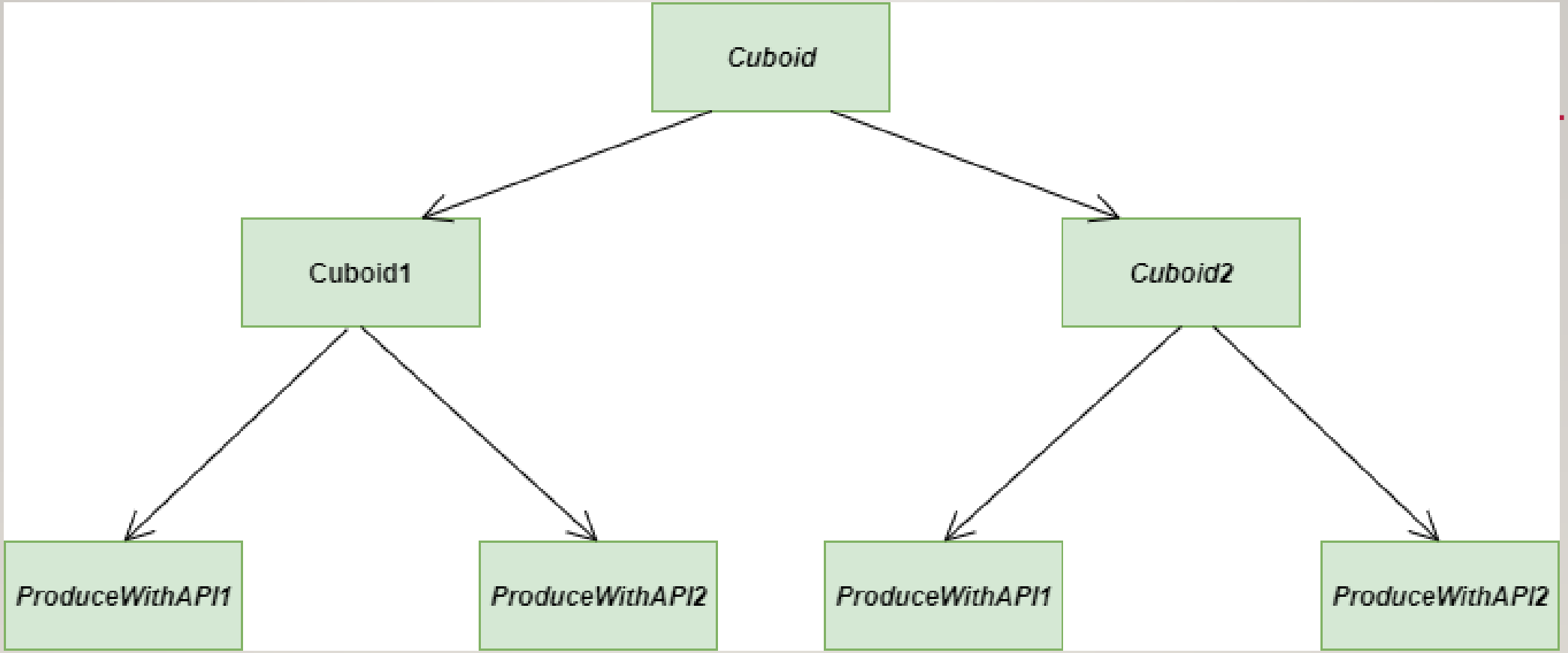
En consecuencia, se puede cambiar las clases de la GUI sin tocar las clases relacionadas con la API. Además, añadir soporte para otro sistema operativo sólo requiere crear una subclase en la jerarquía de implementación.

PROS DEL PATRÓN BRIDGE

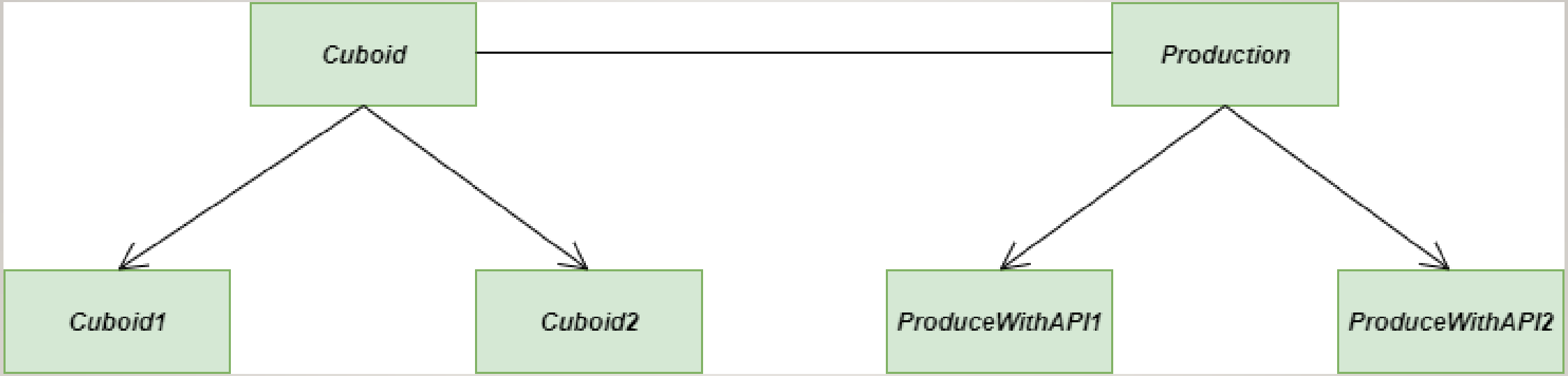
•Puedes crear clases y aplicaciones independientes de plataforma.
• El código cliente funciona con abstracciones de alto nivel. No está expuesto a los detalles de la plataforma.
• Principio de abierto/cerrado. Puedes introducir nuevas abstracciones e implementaciones independientes entre sí.
• Principio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

CONTRAS

- Puede ser complicado de entender al principio.
- Añade complejidad.
- Problemática al no entender bien su funcionamiento.
- Puede ser que el código se complique si aplicas el patrón a una clase muy cohesionada.


Cuboid
Cuboid1
Cuboid2
ProduceWithAPI1
ProduceWithAPI2
ProduceWithAPI1
ProduceWithAPI2

```
    class ProducingAPI1:
        """Implementation Specific Implementation"""
        def produceCuboid(self, length, breadth, height):
            print(f'API1 is producing Cuboid with length = {length}, '
                  f' Breadth = {breadth} and Height = {height}')

    class ProducingAPI2:
        """Implementation Specific Implementation"""
        def produceCuboid(self, length, breadth, height):
            print(f'API2 is producing Cuboid with length = {length}, '
                  f' Breadth = {breadth} and Height = {height}')


    def __init__(self, length, breadth, height):
        """Initialize the necessary attributes"""
        self._length = length
        self._breadth = breadth
        self._height = height

    def produceWithAPI1(self):
        """Implementation specific Abstraction"""
        objectAPIone = self.ProducingAPI1()
        objectAPIone.produceCuboid(self._length, self._breadth, self._height)

    def producewithAPI2(self):
        """Implementation specific Abstraction"""
        objectAPItwo = self.ProducingAPI2()
        objectAPItwo.produceCuboid(self._length, self._breadth, self._height)

    def expand(self, times):
        """Implementation independent Abstraction"""
        self._length = self._length * times
        self._breadth = self._breadth * times
        self._height = self._height * times

# Instantiate a Cubiod
cuboid1 = Cuboid(1, 2, 3)
# Draw it using APIone
cuboid1.produceWithAPI1()
# Instantiate another Cuboid
cuboid2 = Cuboid(19, 20, 21)
# Draw it using APItwo
cuboid2.producewithAPI2()
```


Cuboid
Production
Cuboid1
Cuboid2
ProduceWithAPI1
ProduceWithAPI2

```python
""" Código implementado con Bridge Method.
    Tenemos una clase Cuboid que tiene tres atributos.
    nombrados como largo, ancho y alto y tres
    métodos denominados como ProduceWithAPIOne(), ProduceWithAPItwo(),
    y expand(). Nuestro propósito es separar la abstracción especifica de la implementación de la abstracción independiente de la implementación."""

class ProducingAPI1:
    """Implementation specific Abstraction"""
    def produceCuboid(self, length, breadth, height):
        print(f'API1 is producing Cuboid with length = {length}, '
              f' Breadth = {breadth} and Height = {height}')

class ProducingAPI2:
    """Implementation specific Abstraction"""
    def produceCuboid(self, length, breadth, height):
        print(f'API2 is producing Cuboid with length = {length}, '
              f' Breadth = {breadth} and Height = {height}')

class Cuboid:
    def __init__(self, length, breadth, height, producingAPI):
        """Implementation independent Abstraction"""
        self._length = length
        self._breadth = breadth
        self._height = height
        self._producingAPI = producingAPI

    def produce(self):
        """Implementation specific Abstraction"""
        self._producingAPI.produceCuboid(self._length, self._breadth, self._height)

    def expand(self, times):
        """Implementation independent Abstraction"""
        self._length = self._length * times
        self._breadth = self._breadth * times
        self._height = self._height * times


cuboid1 = Cuboid(1, 2, 3, ProducingAPI1())
cuboid1.produce()
cuboid2 = Cuboid(19, 19, 19, ProducingAPI2())
cuboid2.produce()
```

```
API1 is producing Cuboid with length = 1,  Breadth = 2 and Height = 3
API2 is producing Cuboid with length = 19,  Breadth = 19 and Height = 19
```