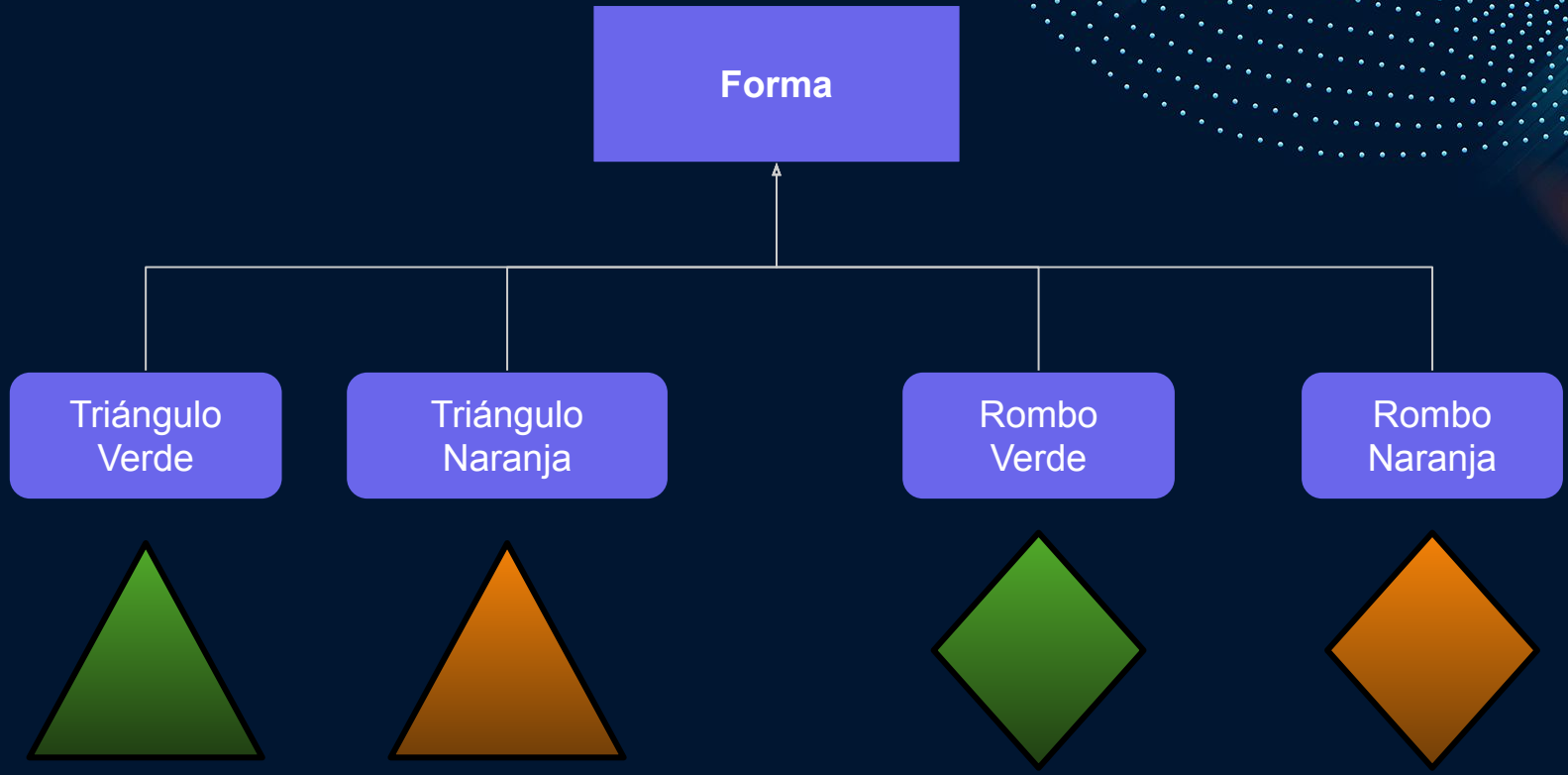
Bridge

Lizeth Corrales Cortés. C02428
Gabriel González Flores. C03376


Forma
Triángulo Verde
Triángulo Naranja
Rombo Verde
Rombo Naranja

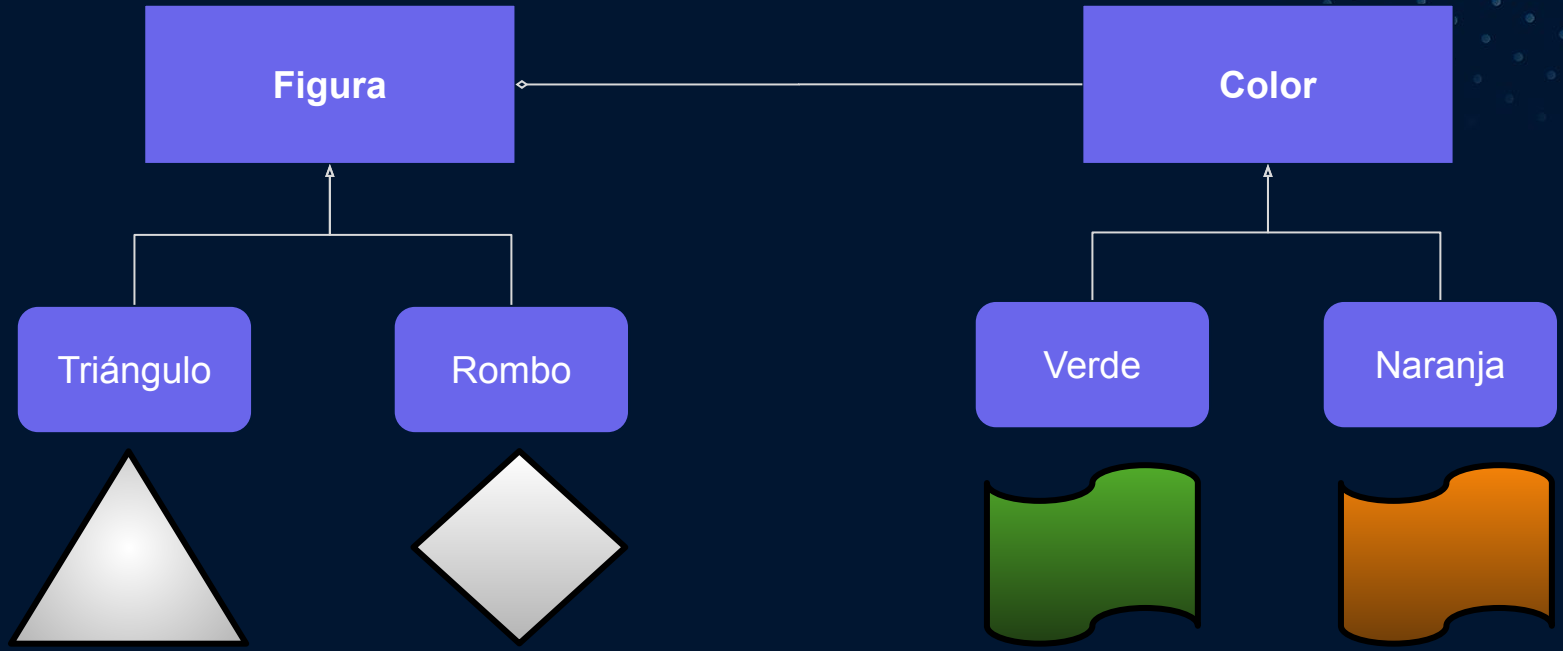


Figura
Color
Triángulo
Rombo
Verde
Naranja

- Concepto
- Problema
- Solución
- Código
- Consecuencias
- Implementación
- Relación con otros patrones
- Referencias
- Actividad

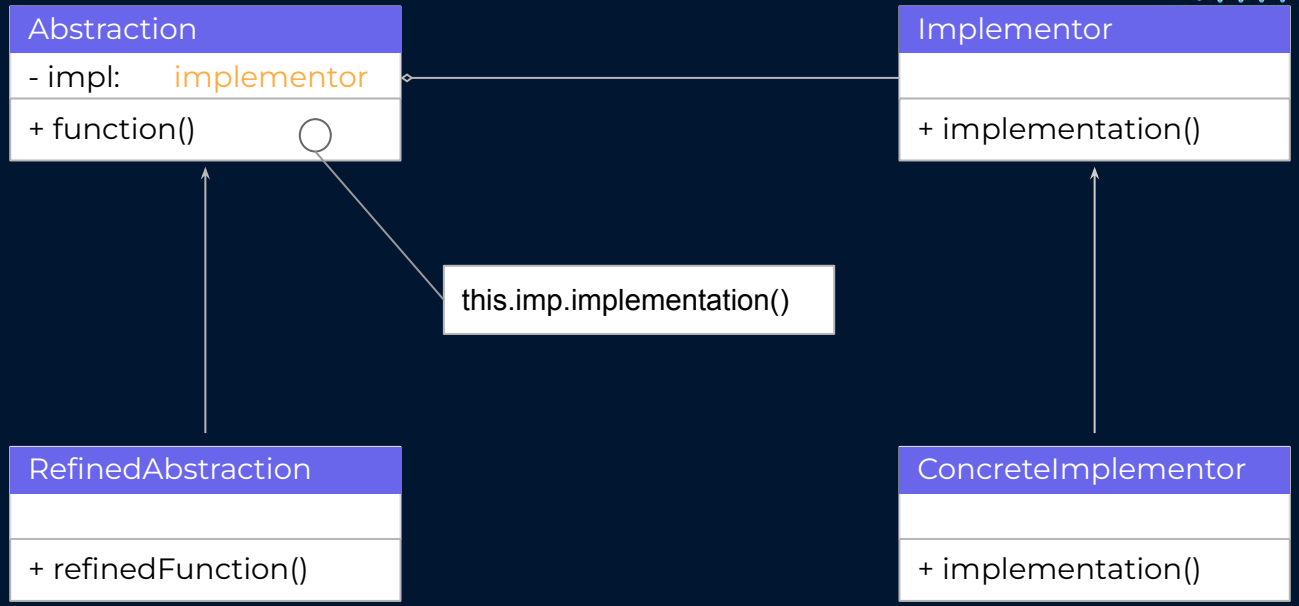
Bridge

Permite desacoplar una abstracción de su implementación, de forma que ambas puedan modificarse de manera independiente sin necesidad de alterar por ello la otra.

¿Cuándo usar el patrón Bridge?

- Para separar y ordenar una clase monolítica que tenga múltiples variaciones de una misma funcionalidad.
- Cuando se necesite extender las clases para que sean independientes entre sí.
- Cuando se vea la necesidad de cambiar las implementaciones durante el tiempo de ejecución.

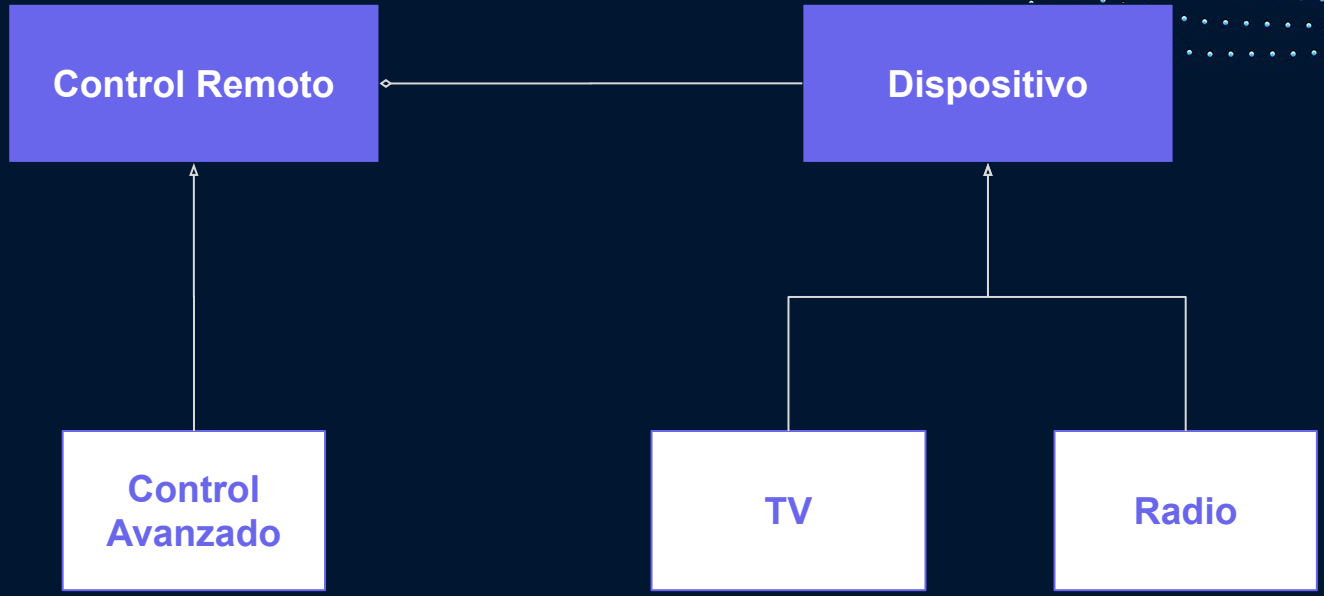
Componentes



Problema


Dispositivo
TV con Control Avanzado
Radio con Control Básico

Solución



Implementor

```python
from abc import ABC, abstractmethod

class Device(ABC):

    @abstractmethod
    def isEnabled(self) -> str:
        pass

    @abstractmethod
    def enable(self) -> None:
        pass

    @abstractmethod
    def disable(self) -> None:
        pass

    @abstractmethod
    def getVolume(self) -> int:
        pass

    @abstractmethod
    def setVolume(self, percent) -> None:
        pass

    @abstractmethod
    def getChannel(self) -> int:
        pass

    @abstractmethod
    def setChannel(self, channel) -> None:
        pass
```

ConcreteImplementor

```python
class Radio(Device):
    def __init__(self):
        self.is_enabled = False
        self.volume = 0
        self.channel = 0

    def isEnabled(self) -> bool:
        return self.is_enabled

    def enable(self) -> None:
        self.is_enabled = True
        print("Radio is Enabled! ✅")

    def disable(self) -> None:
        self.is_enabled = False
        print("Radio is Disabled! ❌")

    def getVolume(self) -> int:
        return self.volume

    def setVolume(self, percent) -> None:
        self.volume += percent

    def getChannel(self) -> int:
        return self.channel

    def setChannel(self, channel) -> None:
        self.channel += channel
```

ConcreteImplementor

```python
class TV(Device):
    def __init__(self):
        self.is_enabled = False
        self.volume = 0
        self.channel = 0

    def isEnabled(self) -> bool:
        return self.is_enabled

    def enable(self) -> None:
        self.is_enabled = True
        print("¡TV is Enabled! ✅")

    def disable(self) -> None:
        self.is_enabled = False
        print("¡TV is Disabled! ❌")

    def getVolume(self) -> int:
        return self.volume

    def setVolume(self, percent) -> None:
        self.volume += percent

    def getChannel(self) -> int:
        return self.channel

    def setChannel(self, channel) -> None:
        self.channel += channel
```

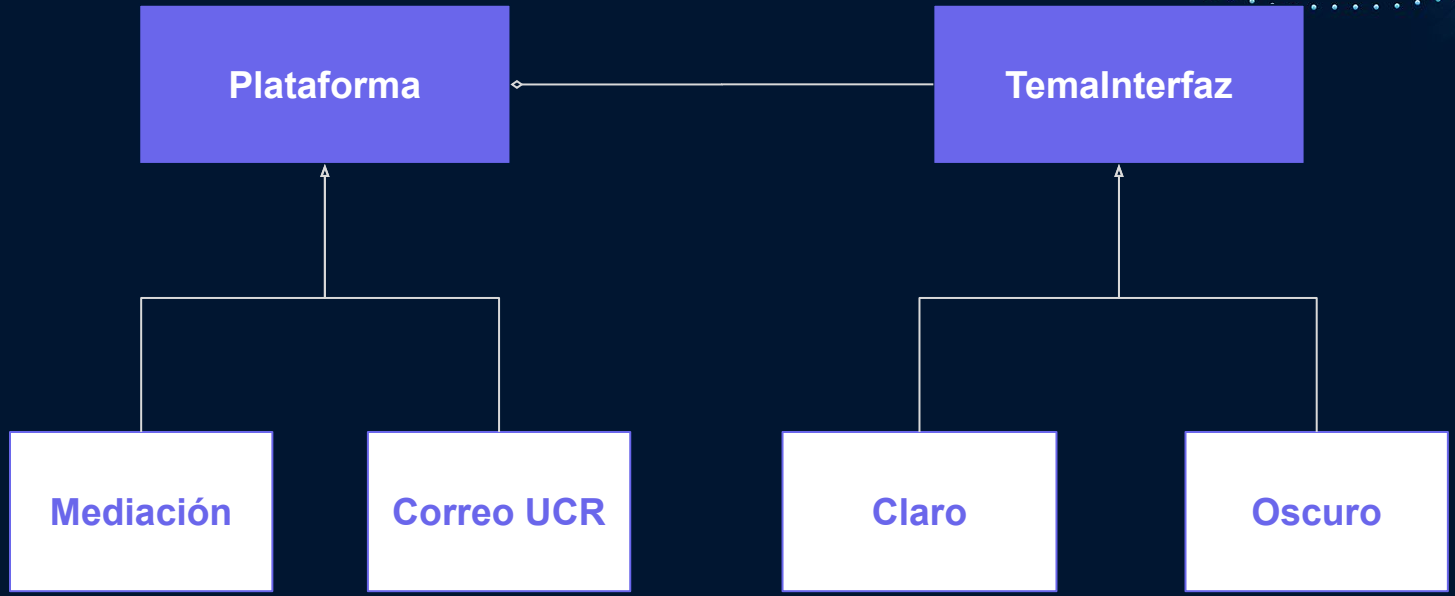
Abstraction

```python
class Remote:
    def __init__(self, device: Device) -> None:
        self.device = device  # ESTE ES EL BRIDGE

    def togglePower(self) -> None:
        if(self.device.isEnabled()):
            self.device.disable()
        else:
            self.device.enable()

    def volumeDown(self) -> None:
        self.device.setVolume(-1)

    def volumeUp(self) -> None:
        self.device.setVolume(1)

    def channelDown(self) -> None:
        self.device.setChannel(-1)

    def channelUp(self) -> None:
        self.device.setChannel(1)
```

RefinedAbstraction

```
116   class AdvancedRemote(Remote):
117       def mute(self) -> None:
118           self.device.setVolume(-self.device.getVolume())
```

Ejemplo UCR


Plataforma
TemaInterfaz
Mediación
Correo UCR
Claro
Oscuro

Consecuencias

Ventajas

Separa interfaz e implementación

Una implementación de una abstracción no está permanentemente confinada a una interfaz, ya que puede ser configurada en tiempo de ejecución.

Desacopla las clases Abstraction e Implementor

Provoca que se elimine en tiempo de compilación las dependencias sobre una clase de implementación.

Mejora la extensibilidad

Se puede extender en forma independiente las jerarquías de Abstraction e Implementor.

Encapsulamiento de detalles

Se puede proteger a los clientes de los detalles de implementación.

Desventajas

Aparece un nivel adicional de direccionamiento indirecto

El patrón logra flexibilidad al introducir un nivel adicional de direccionamiento indirecto, lo que hace que el Abstraction dependa de un objeto Implementor.

Implementación

- Identificar las diferentes dimensiones independientes en nuestra clase por anticipado.
- Determinar las operaciones que va a necesitar el cliente y definirlas en la interfaz Abstraction.
- Establecer las operaciones abstractas que estarán disponibles para todas las plataformas en la interfaz Implementor.
- Crear las clases ConcreteImplementor asegurándonos de seguir la interfaz Implementor.
- Dentro de la clase Abstraction agregar un campo que referencia al Implementor.

Relación con otros patrones

Abstract Factory:

- Crea y configura un Bridge particular.
- Sin embargo, las clases concretas no implementan la abstracción.

Builder:

- La clase Director juega el papel de la abstracción.
- Los builders actúan como implementaciones.

Adapter :

- Client y Abstraction: delega a una interfaz.
- Target e Implementor: define una interfaz a la que adherirse.
- Adapter y Refined Abstraction: implementa la interfaz y cumple con los requisitos.

Referencias

- Almeira, A.y Perez, Vanina (2007) *Arquitectura de Software: Estilos y Patrones.* (pp.60-61) https://silo.tips/download/arquitectura-de-software-estilos-y-patrones
- Doeken.org (2021) *Adapter Pattern vs. Bridge Pattern.* https://doeken.org/blog/adapter-vs-bridge-pattern
- Gamma, E. et all (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (pp.151-161). Addison-Wesley. http://www.javier8a.com/itc/bd1/articulo.pdf
- Refactoring.guru. (s.f.) *Bridge.* https://refactoring.guru/es/design-patterns/bridge
- W3sdesign (s.f) *Bridge.* http://w3sdesign.com/?gr=s02&ugr=struct#gf

Actividad