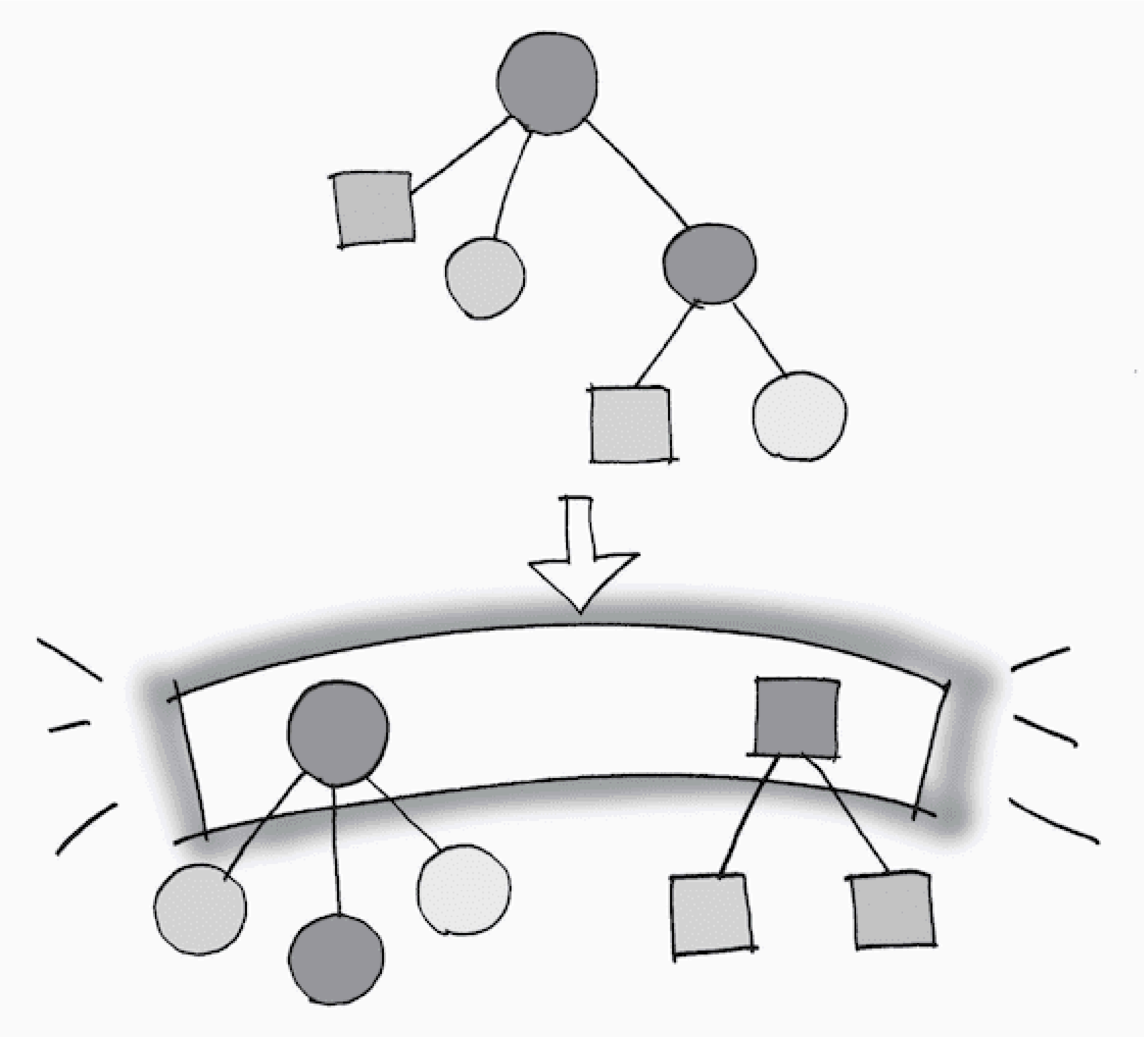
Bridge Pattern

Decoupling abstraction from its implementation so both can evolve independently



Bridge Pattern

Using the BLE (Bluetooth) protocol, we can use any mobile device to
control any BLE devices (IoT, car, TV, and others).

Decouple the **Abstraction** (mobile devices) from the **Implementation** (BLE devices).

⚠️ *Warning: it's not the abstraction(interface)/implementation that we have discussed.*

The Problem

- We have a **Display** class (shows text)
- We have **different Display types**
    - Normal Display
    - CountDisplay (repeats text)
- We also have **different ways to print**
    - Plain Print
    - Fancy Print

If we mix them, we need:

- NormalPlainDisplay
- NormalFancyDisplay
- CountPlainDisplay
- CountFancyDisplay

Too many classes! We call this "Explosion of classes (EoC)"

The challenge: how to **separate abstraction** from **implementation** so that both can **vary independently** without class explosion?

The *Bridge DP* as the Solution

- Split into **two worlds**:
  - **Abstraction:** Display, CountDisplay
  - **Implementation:** PlainDisplayImpl, FancyDisplayImpl

⚠️ You can use the names PlainPrint & FancyPrint

- Use a **bridge** (`impl`) to connect them.
- Now:
  - Display ↔️ PlainDisplayImpl (PlainPrint)
  - Display ↔️ FancyDisplayImpl (FancyPrint)
  - CountDisplay ↔️ PlainDisplayImpl (PlainPrint)
  - CountDisplay ↔️ FancyDisplayImpl (FancyPrint)

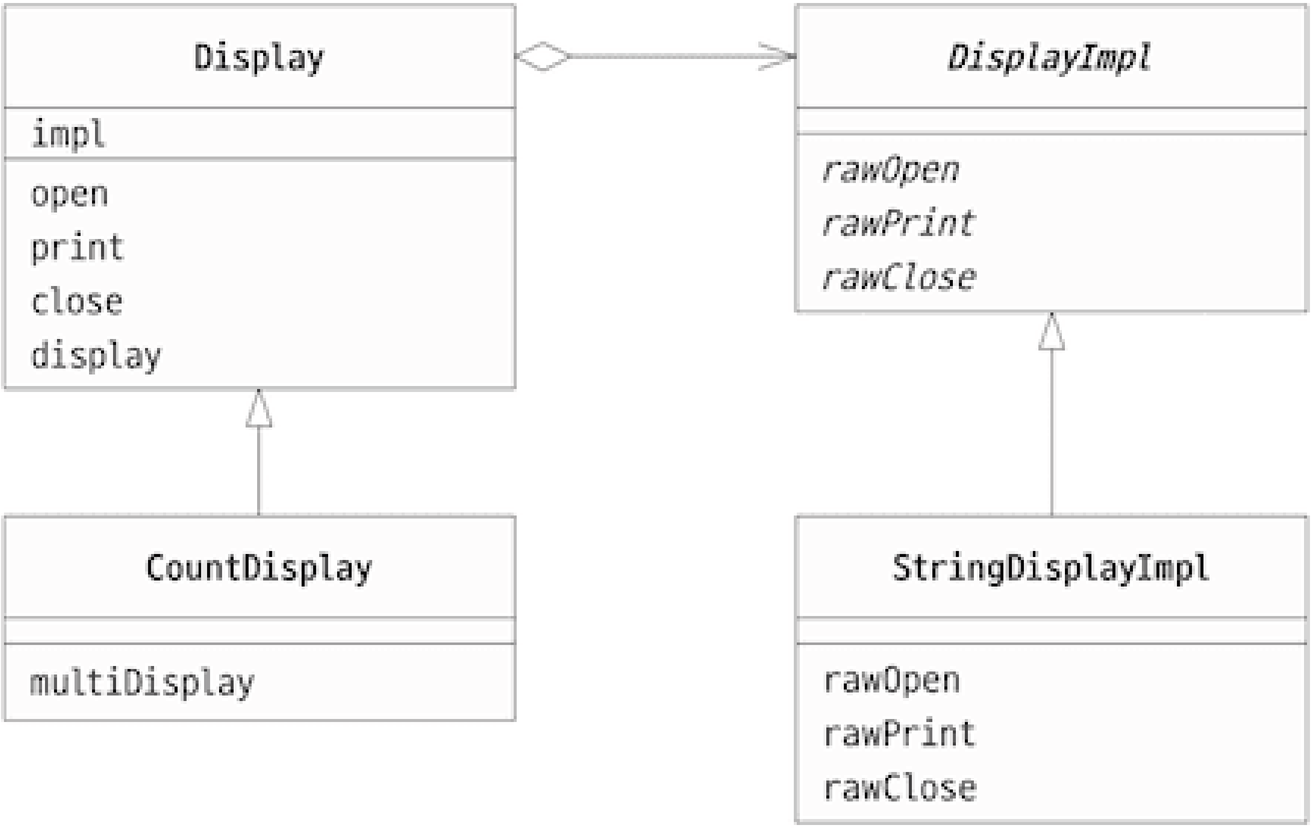
No explosion of classes (EoC)

The Solution

We use **aggregation** to show the connection between the abstraction (Display) and implementation (DisplayImpl)

- The **Abstraction** owns the **Implementation**.

The Design


Display
impl
open
print
close
display
DisplayImpl
rawOpen
rawPrint
rawClose
CountDisplay
multiDisplay
StringDisplayImpl
rawOpen
rawPrint
rawClose

Step 1: Understand the Players

In this design, we have players:

- *Abstraction* (Display)
    - **Refined Abstraction** (CountDisplay)
- *Implementor* (DisplayImpl)
    - **Concrete Implementor** (StringDisplayImpl, FancyDisplayImpl)

Step 2: Separation of *abstraction* (not Abstraction) and concretion

We have an abstraction of *Abstraction* that `uses` an *implementation* through a *bridge*.

- The abstraction of *Abstraction*
    - Display
- The abstraction of *Implementation*
    - DisplayImpl

- The concretion of *Abstraction*
    - PlainDisplay
    - CountDisplay
- The concretion of *Implementation*
    - StringDisplayImpl
    - FancyDisplayImpl

- Notice that the *Abstraction* `delegates` to the *Implementation*.
  - It is as if we have a **remote control** that works with different **TV brands**.
  - We can add **iPhone** or **Android** to control any **TV brands**.
  - The *Abstractions* doesn't know about specific **Implementations**.

Code

- Main Method
- Abstraction Classes
- Implementation Classes

Main Method

```python
from display import Display
from count_display import CountDisplay
from string_display_impl import StringDisplayImpl
from fancy_display_impl import FancyDisplayImpl

def main():
  print("=== Bridge Pattern Demo ===\n")

  print("1. Plain Display + StringDisplayImpl:")
  d1 = PlainDisplay(StringDisplayImpl("Hello, USA."))
  d1.display()

  print("\n2. Plain Display + FancyDisplayImpl:")
  d2 = PlainDisplay(FancyDisplayImpl("Hello, USA."))
  d2.display()

  print("\n3. CountDisplay + StringDisplayImpl:")
  d3 = CountDisplay(StringDisplayImpl("Hello, World."))
  d3.display()
```

Step 1: Create Abstraction with Implementation

```
d1 = PlainDisplay(StringDisplayImpl("Hello, USA."))
d2 = PlainDisplay(FancyDisplayImpl("Hello, World."))
d3 = CountDisplay(StringDisplayImpl("Hello, World."))
d4 = CountDisplay(FancyDisplayImpl("Hello, World."))
```

- The **PlainDisplay/CountDisplay** (abstraction) is connected to **StringDisplayImpl/FancyDisplayImpl** (implementation) through the *bridge*.
- The abstraction doesn't know the specific implementation details.

Step 2: Use extended abstraction

```
d4 = CountDisplay(FancyDisplayImpl("Hello, World."))
d4.multi_display(5)
```

- **CountDisplay** extends the abstraction with additional functionality.
- It can work with any **implementation** (**StringDisplayImpl** or **FancyDisplayImpl**).

Step 3: Independence of hierarchies

```
# Any abstraction can work with any implementation
combinations = [
    PlainDisplay(StringDisplayImpl("Simple")),
    PlainDisplay(FancyDisplayImpl("Fancy")),
    CountDisplay(StringDisplayImpl("Count Simple")),
    CountDisplay(FancyDisplayImpl("Count Fancy"))
]
```

- We can `mix and match` any abstraction with any implementation.
- This gives us **n × m** combinations with only **n + m** classes.
- No EoC (we need n x m classes with Inheritance).

Discussion

Key Benefits

1. **Independence**: Abstraction and implementation can vary independently
2. **Runtime Binding**: Implementation can be selected at runtime
3. **Extensibility**: Easy to add new Abstractions or Implementations
4. **Hiding Details**: Implementation details are hidden from clients

When to Use Bridge

- When you want to avoid permanent binding between abstraction and implementation
- When both abstractions and implementations need to be extensible
- When implementation changes shouldn't affect clients
- When you want to share implementations among multiple objects

Related Patterns

- **Abstract Factory**: Can be used with Bridge to create and configure bridges
- **Adapter**: Bridge is designed upfront, Adapter is added to legacy code

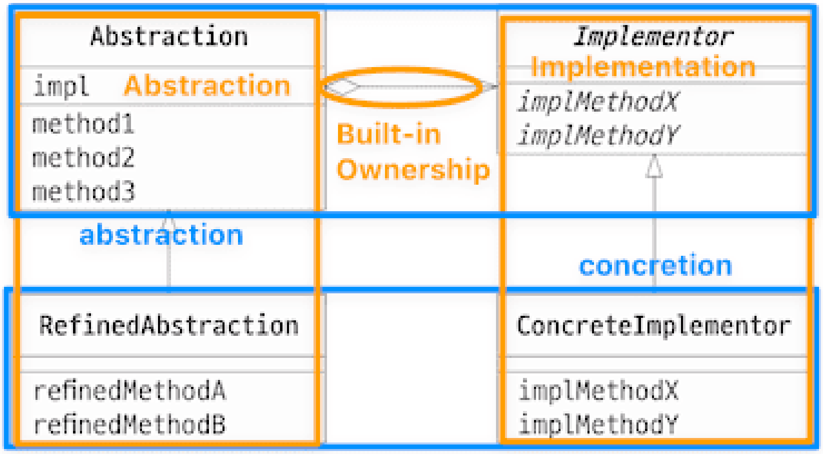
Bridge vs Abstract Factory

- **Bridge** = separates *what* vs. *how* and **Abstract Factory** = decides *which* implementation to plug into the bridge.
- **Abstract Factory** can **build the right DisplayImpl** for us.
  - Bridge then **uses that factory-made implementation**.

Bridge vs Adapter

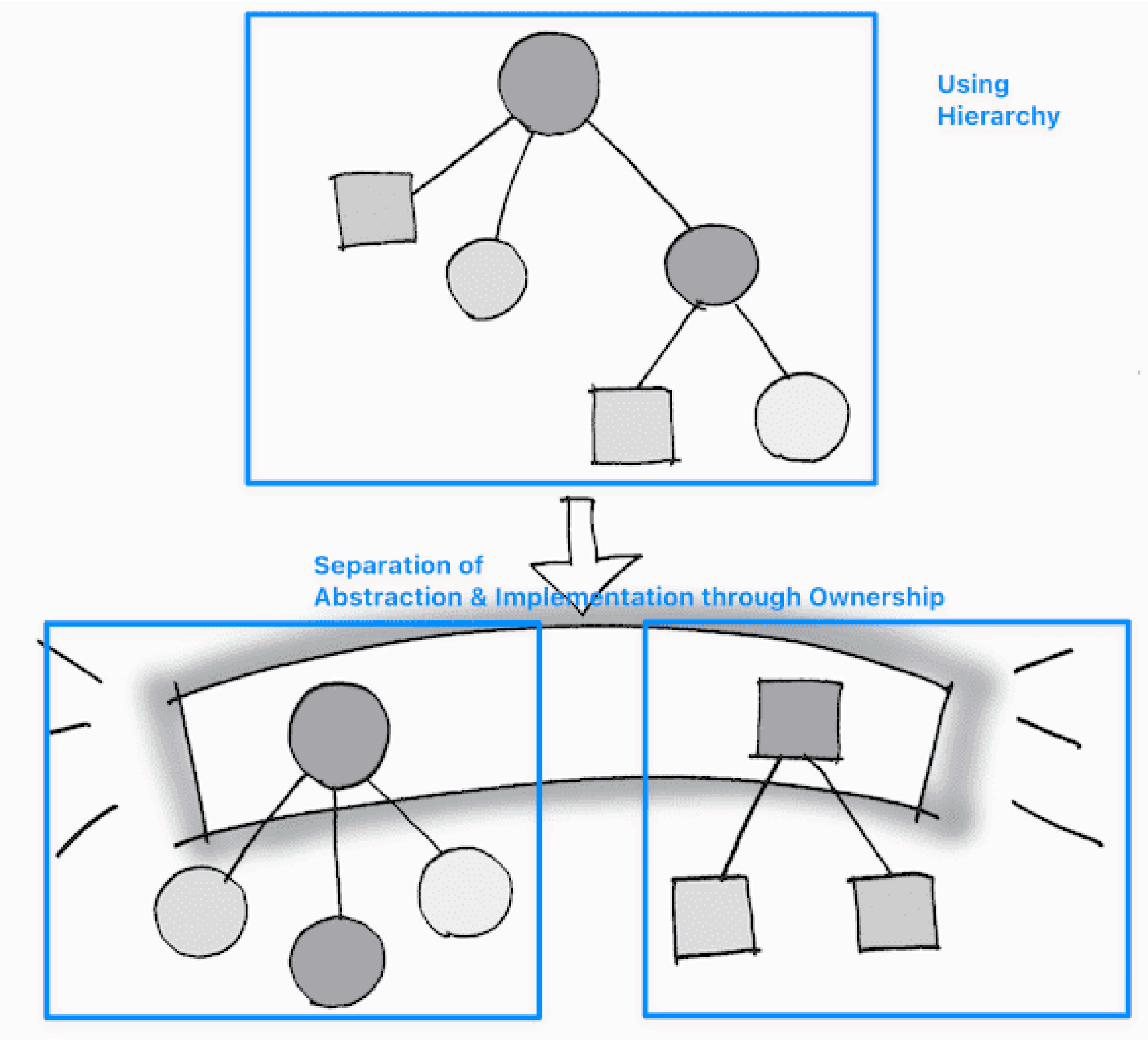
- **Bridge**:
  - Designed upfront to separate Abstraction from Implementation
  - Both Abstraction and Implementation can have their own hierarchies
- **Adapter**:
  - Added afterwards to make incompatible interfaces work together
  - Usually involves existing classes with incompatible interfaces

UML



Two divisions:

- The abstraction & concretion divide
- The Abstraction & Implementation divide
  - The Abstraction owns the Implementation


Using
Hierarchy
Separation of
Abstraction & Implementation through Ownership