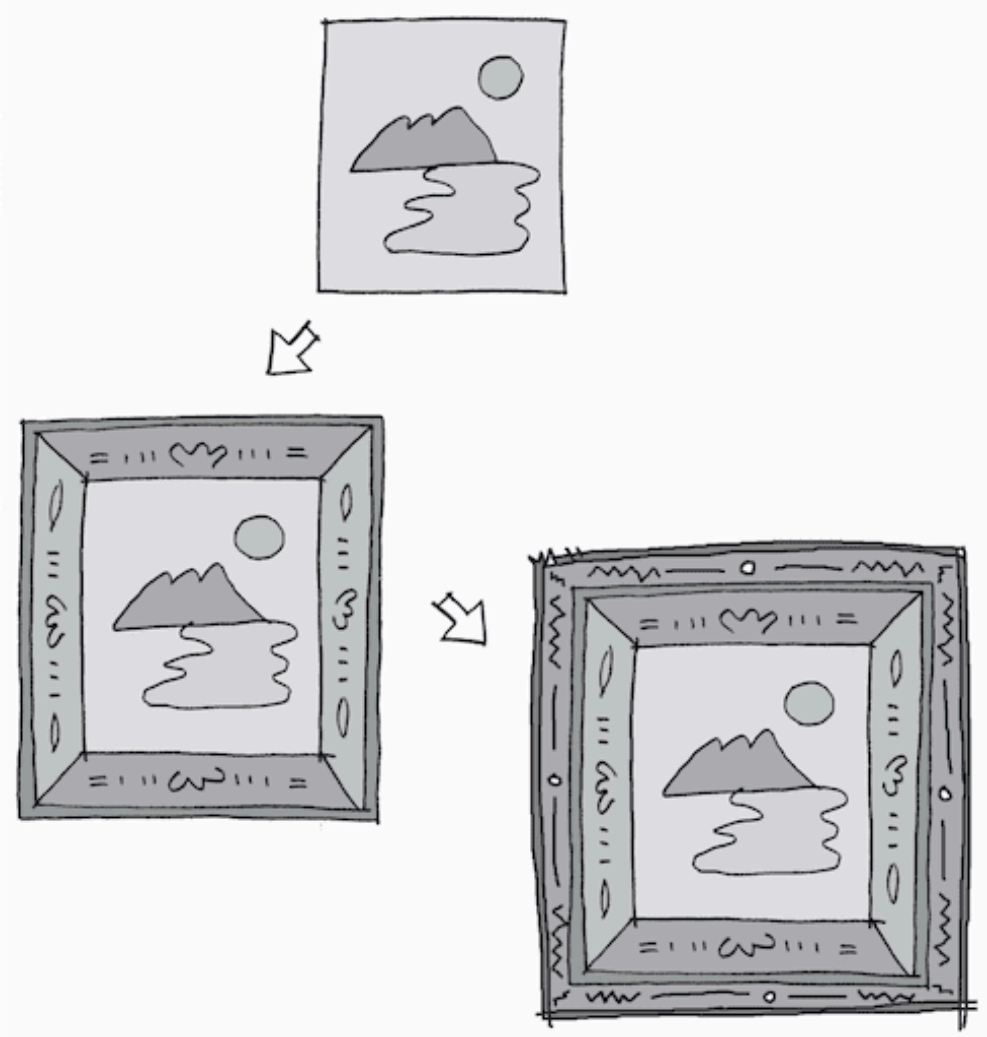
Decorator Pattern

Add Responsibilities to Objects Dynamically



Decorator Pattern

We start with something **basic**, and we add **extra features** step by step; each feature *decorates* the original:

- **Coffee shop**: Basic coffee + milk + sugar + whipped cream
- **Gift wrapping**: Present + box + wrapping paper + ribbon + bow
- **Car options**: Basic car + air conditioning + leather seats + navigation
- **Text formatting**: Plain text + bold + italic + underline

The Problem

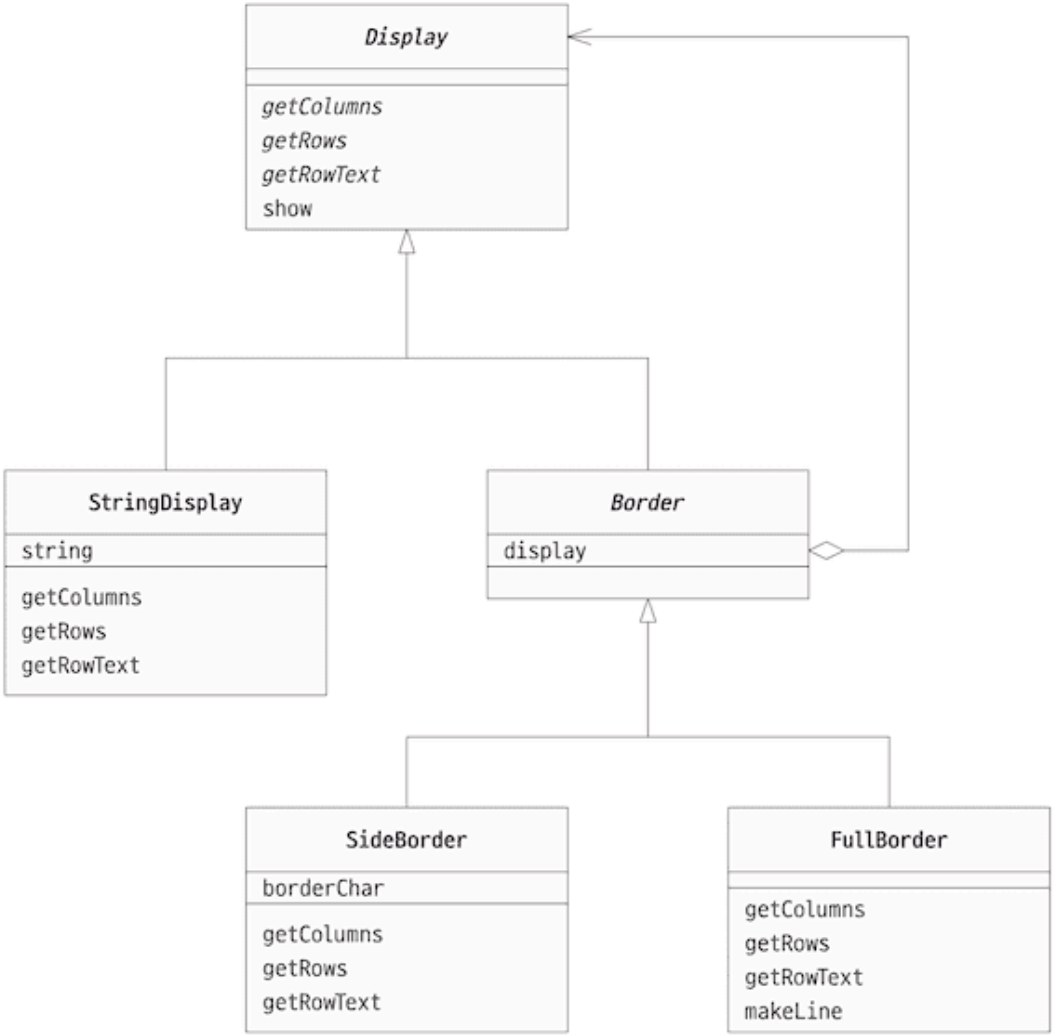
- We have **text display** that we want to enhance with different decorations.
- We want to add **borders**, **colors**, or other **enhancements** without changing the original **text**.

The challenge: how to add multiple decorations dynamically and stack them together?

The *Decorator* as the Solution

- We have an abstraction *Component* that both original **objects** and **decorators** implement.
- We do not need to `modify` the original object, we only need to `wrap` it with **decorators**.

The Solution (Design)



Step 1: Understand the Players

In this design, we have players:

- *Component* (Display)
    - **ConcreteComponent** (StringDisplay)
- *Decorator* (Border)
    - **ConcreteDecorator** (SideBorder, FullBorder)

Step 2: Same interface for all objects

- We need to see that both **Component** and **Decorator** implement the same *interface*.

Step 3: Understand abstractions (Component-Decorator)

- We have a *Component* that `defines` the interface for objects that can have responsibilities added.
    - *Component* (Display)
    - *Decorator* (Border) - maintains reference to Component
- In short, **decorators** `wrap` the original **component** while maintaining the same *interface*.

- Notice that **Decorator** `contains` a *Component* (composition).
    - It is as if we put **gift wrapping** around a **present** - the present is still there, just enhanced.
- Also notice that Component & Decorator have the same interfaces.
    - To do this, the Decorator should inherit from the Component.

Step 4: Understand concretion (Component-Decorator)

- We have **StringDisplay** (basic component) and **SideBorder**, **FullBorder** (decorators).
    - **ConcreteComponent** (StringDisplay): core functionality
    - **ConcreteDecorator** (SideBorder, FullBorder): add specific enhancements

Code

- Main Method
- Component Classes
- Decorator Classes

Main Method

```
from string_display import StringDisplay
from side_border import SideBorder
from full_border import FullBorder

def main():
    print("=== Decorator Pattern Example ===\n")

    # Basic component
    text = StringDisplay("Hello, world.")
    text.show()

    # Add decorations
    bordered = SideBorder(text, '#')
    bordered.show()

    # Stack decorations
    fully_bordered = FullBorder(bordered)
    fully_bordered.show()
```

Step 1: Create a basic component

```
text = StringDisplay("Hello, world.")
text.show()
```

- **StringDisplay** is the basic **component** that provides core functionality.
- It implements the *Display* interface.

Step 2: Add decorations

```
bordered = SideBorder(StringDisplay("Hello, world."), '#')
bordered.show()
```

- **SideBorder** wraps the original **component** and adds side characters.
- It maintains the same *interface* as the original component.

Step 3: Stack multiple decorations

```
fully_bordered = FullBorder(SideBorder(StringDisplay("Hello, world."), '#'))
fully_bordered.show()
```

- **Decorators** can be stacked infinitely.
- Each decorator adds its enhancement while preserving previous decorations.

Discussion

Decorator @ in Python

```
def FullBorder(f):
    def wrapped(*args, **kwargs):
        s = f(*args, **kwargs)
        line = "+" + "-" * len(s) + "+"
        return f"{line}\n|{s}|\n{line}"
    return wrapped

def SideBorder(ch="#"):
    def deco(f):
        def wrapped(*args, **kwargs):
            s = f(*args, **kwargs)
            return f"{ch}{s}{ch}"
        return wrapped
    return deco

@FullBorder
@SideBorder("#")
def make_text():
```

Key Benefits

1. **Flexibility**: Add responsibilities dynamically at runtime
2. **Single Responsibility**: Each decorator has one clear purpose
3. **Open/Closed**: Open for extension, closed for modification
4. **Composition**: Alternative to inheritance for extending behavior

When to Use Decorator

- When you want to add responsibilities to objects dynamically
- When extension by subclassing is impractical
- When you want to add features that can be withdrawn
- When you have many optional features that can be combined

Related Patterns

- **Adapter** (Different interface): Adaptor changes interface, Decorator enhances behavior (implementation/algorithm)
- **Composite** (Same interface): Both use composition, but Composite focuses on tree structures
- **Strategy** (Different algorithm update): Decorator changes behaviors (algorithms) from outside, Strategy changes behaviors (algorithms) from inside

Decorator vs Inheritance

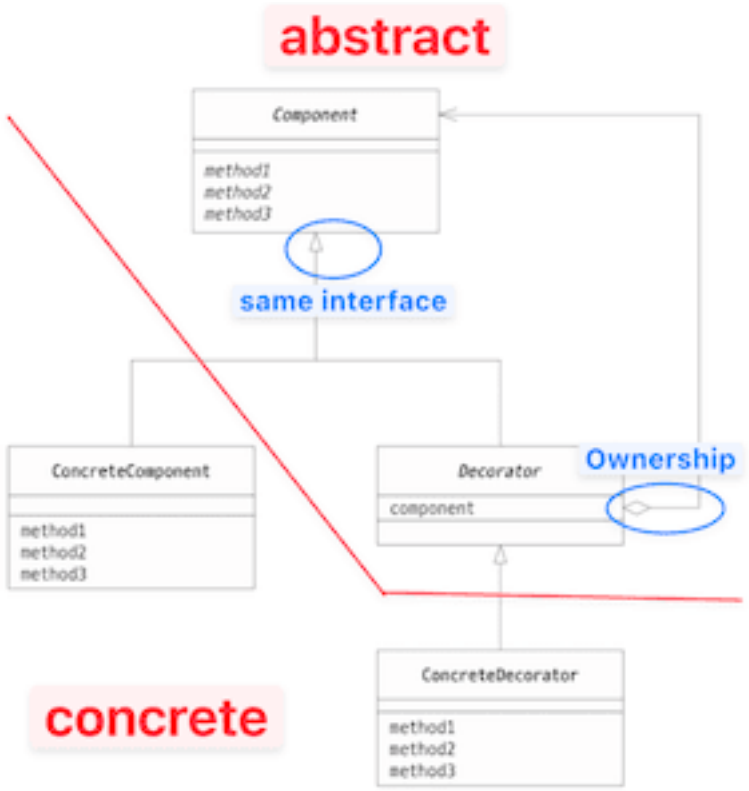
Inheritance:

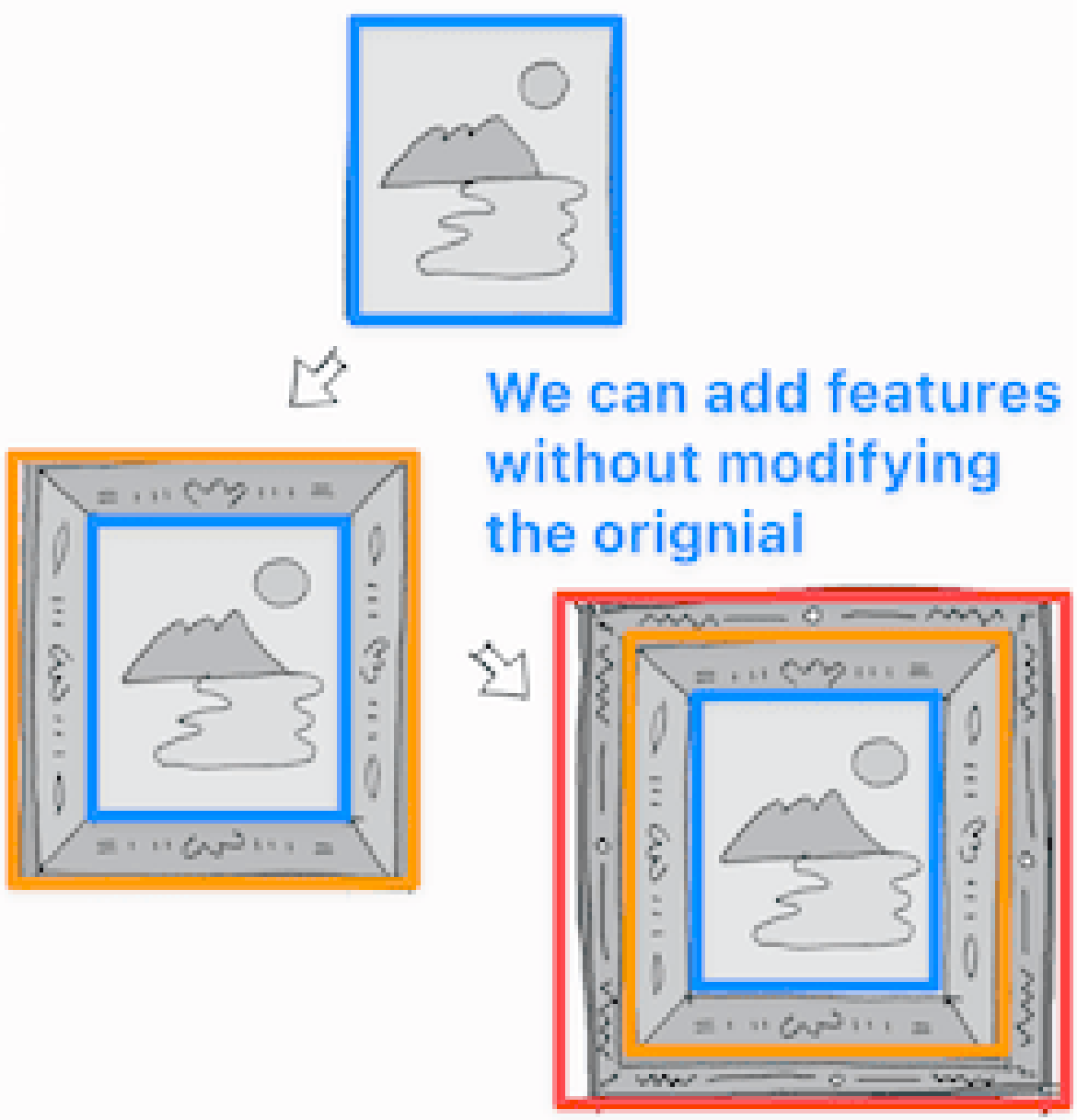
- Static behavior extension at compile time
- Single inheritance limitation

Decorator:

- Dynamic behavior extension at runtime
- Multiple decorations are possible through composition

UML


abstract
Component
method1
method2
method3
same interface
ConcreteComponent
method1
method2
method3
Decorator
component
Ownership
ConcreteDecorator
method1
method2
method3
concrete


We can add features
without modifying
the orignial