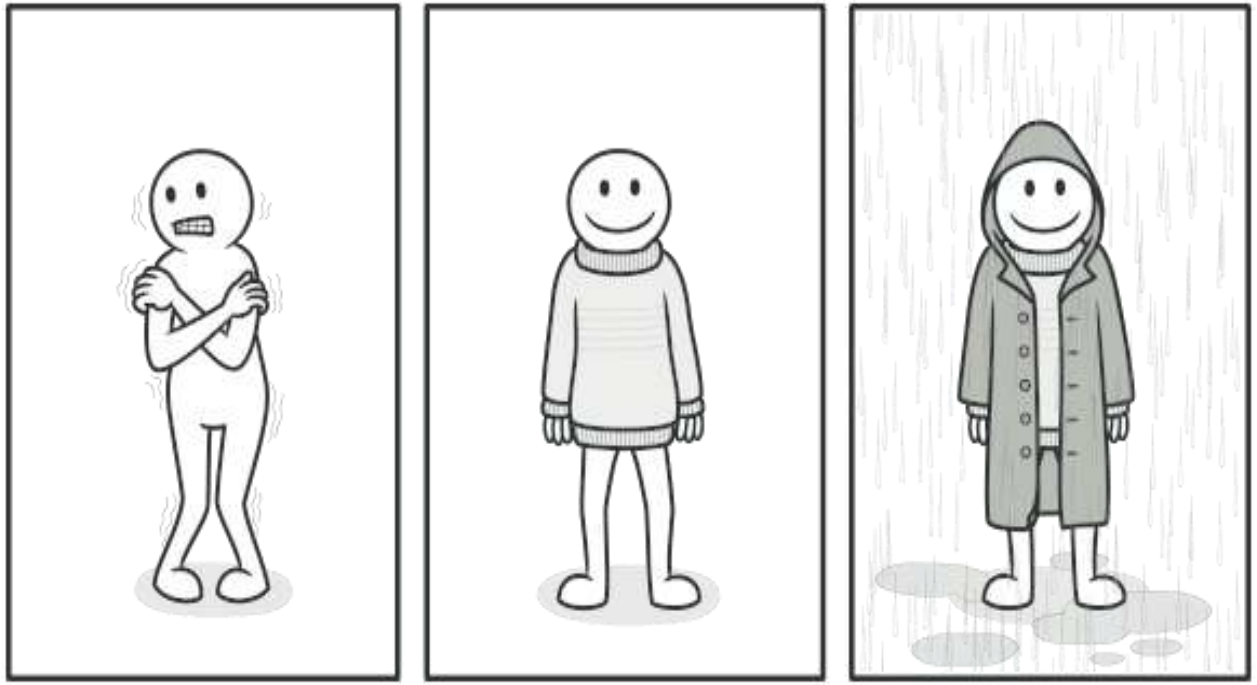
DECORATOR

Adam Budai

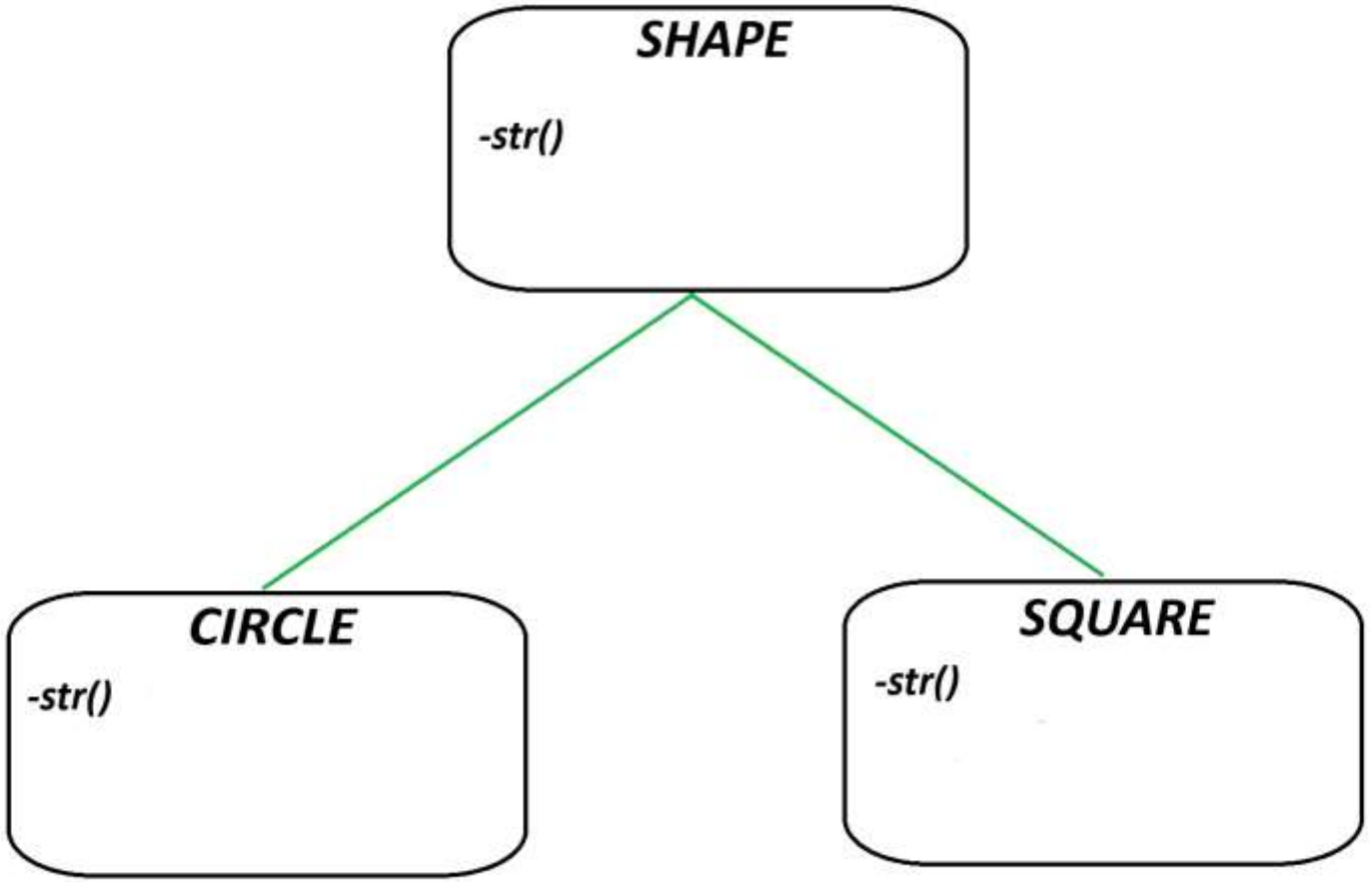
INTENT

- Attach additional responsibilities

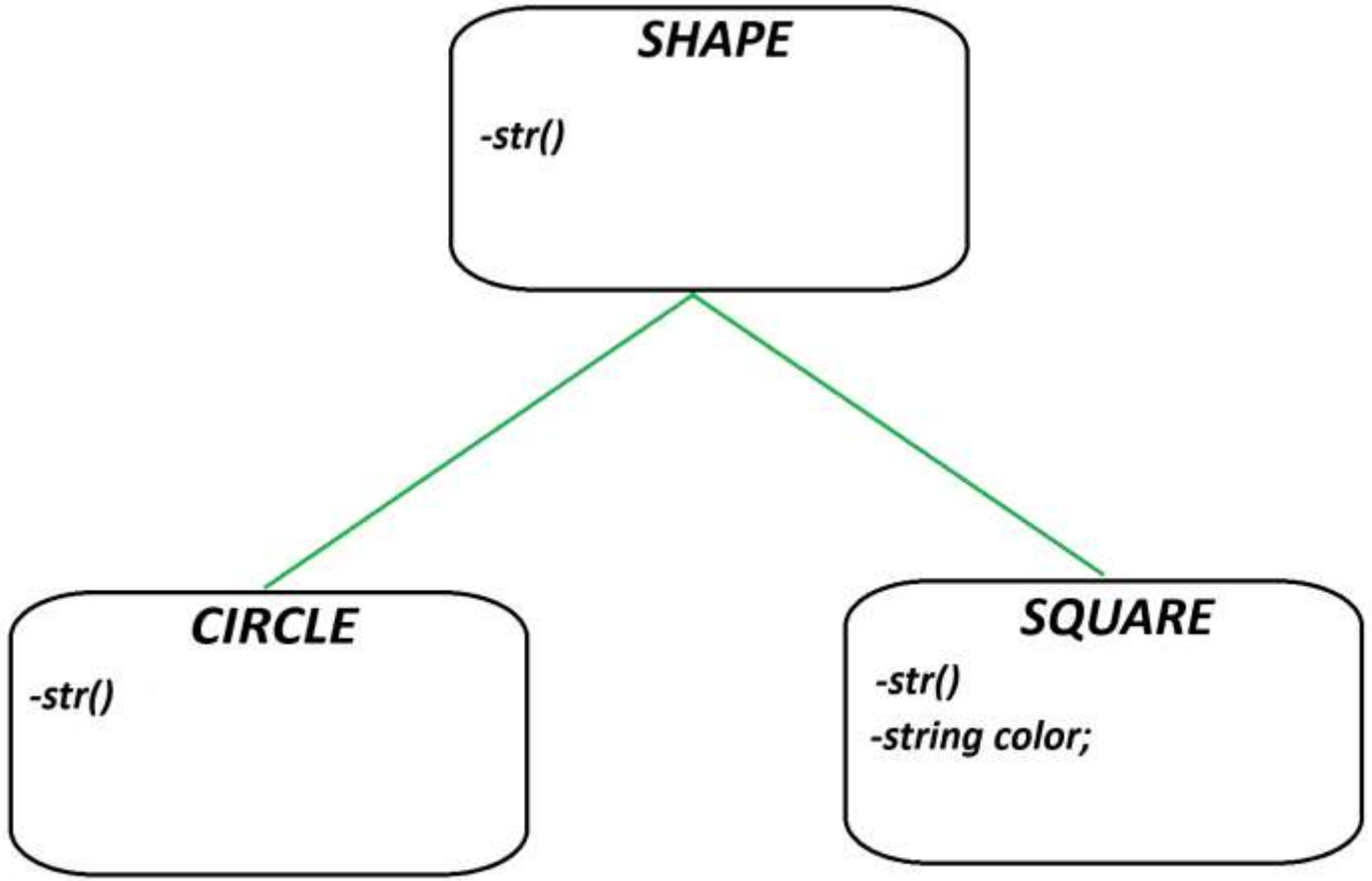
- Placing object inside special wrapper



EXAMPLE



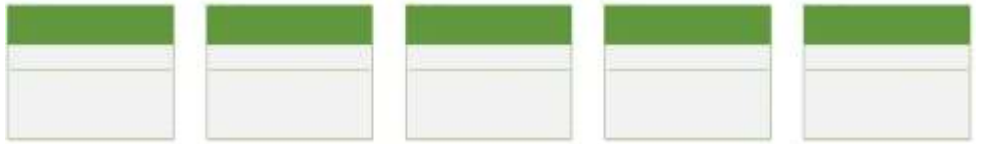
EXAMPLE



POSSIBLE SOLUTIONS

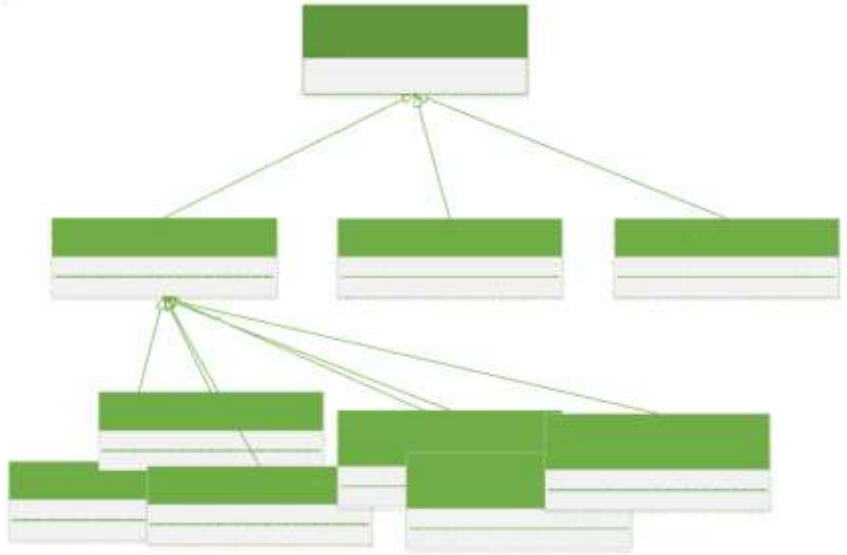
1. Restrict to a certain subset

-So many classes, change of common attribute



2. Inheritance

-Still a lot of classes, not flexible



3. Superclass

-Wasteful, long…

DECORATOR

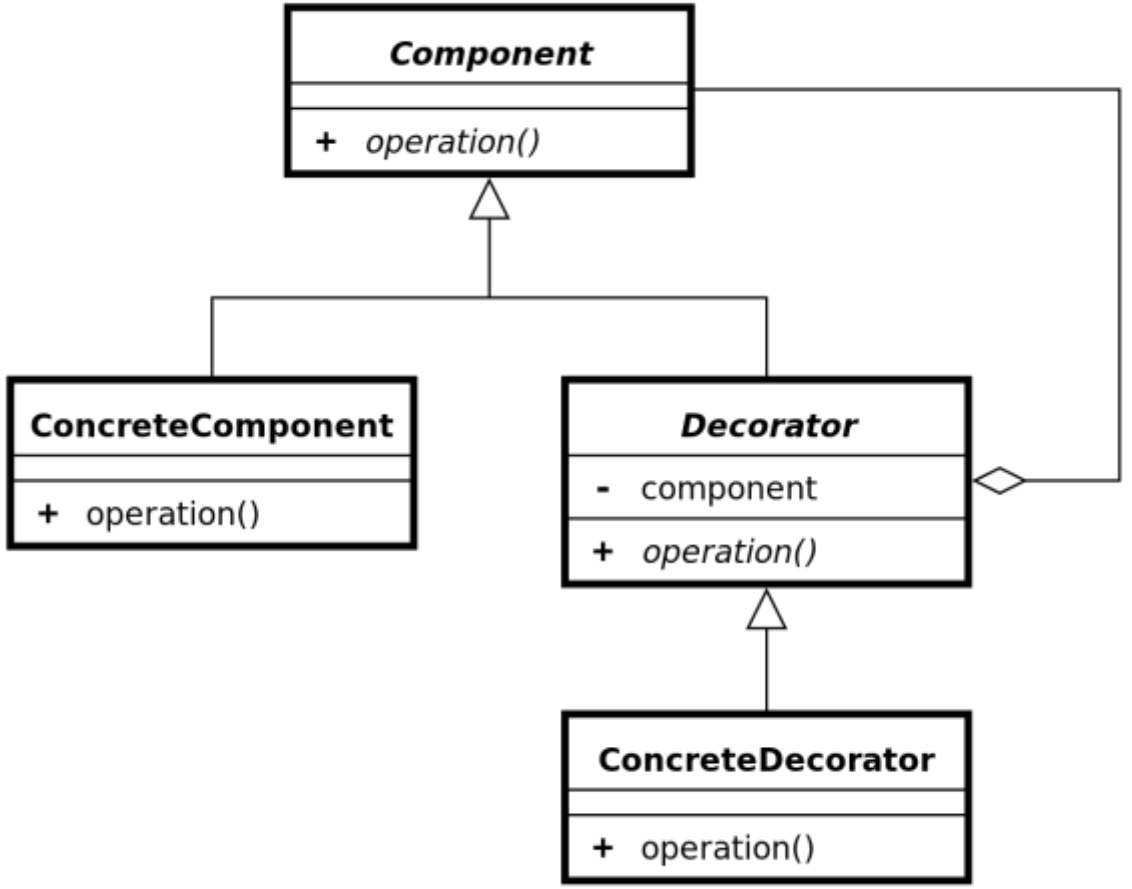
Dynamically adding/removing responsibilities

Supports open-closed principle

Supports singe-responsibility principle

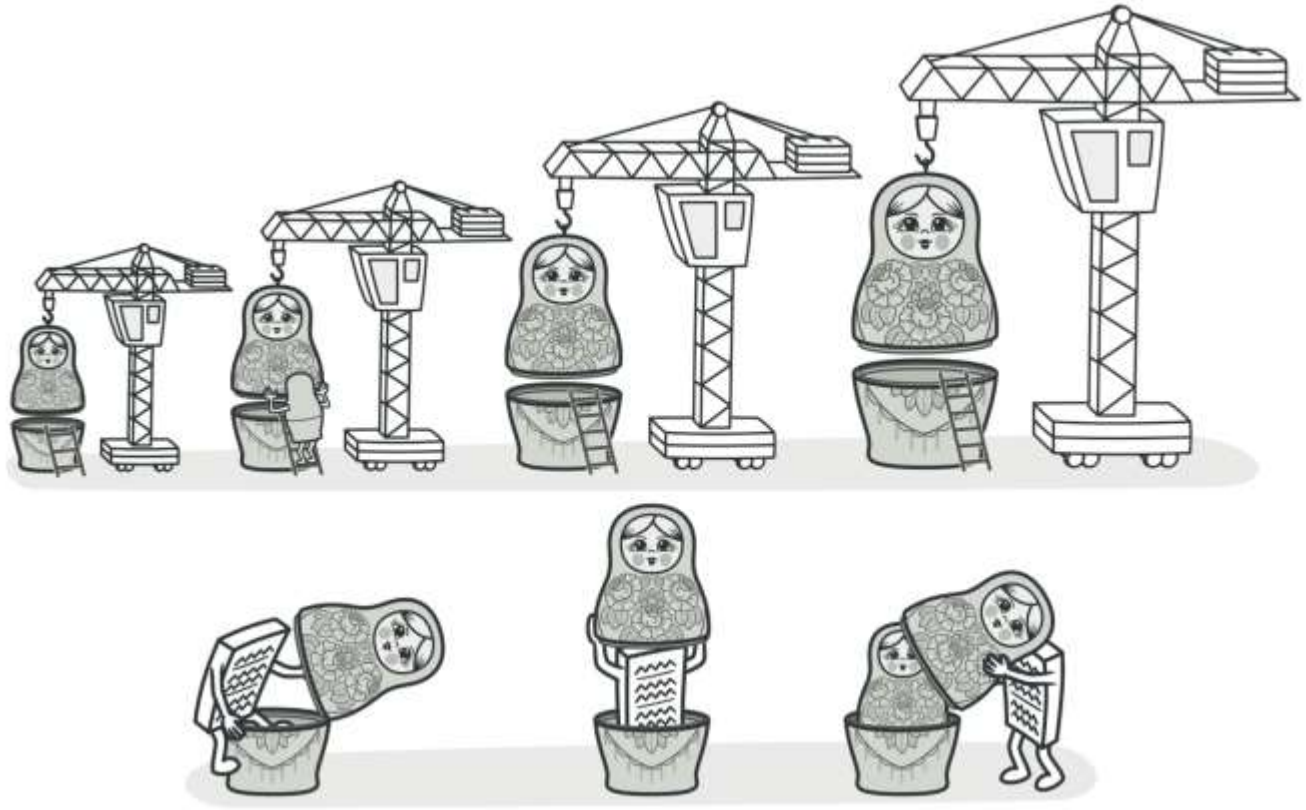
A flexible alternative to subclassing for extending functionality

Decorators can be recursively layered



TYPES OF DECORATOR

- Dynamic
  - Runtime flexibility
  - Dynamic composition
- Static
  - Compile-Time optimization
  - Code clarity



IMPLEMENTATION

```
struct Shape
{
    virtual std::string str() const = 0;
};
```

```
struct Square : Shape
{
    float side;

    Square(){}

    explicit Square(const float side)
            : side{side}
    {
    }

    std::string str() const override
    {
        std::ostringstream oss;
        oss << "A square of with side " << side;
        return oss.str();
    }
};
```

IMPLEMENTATION

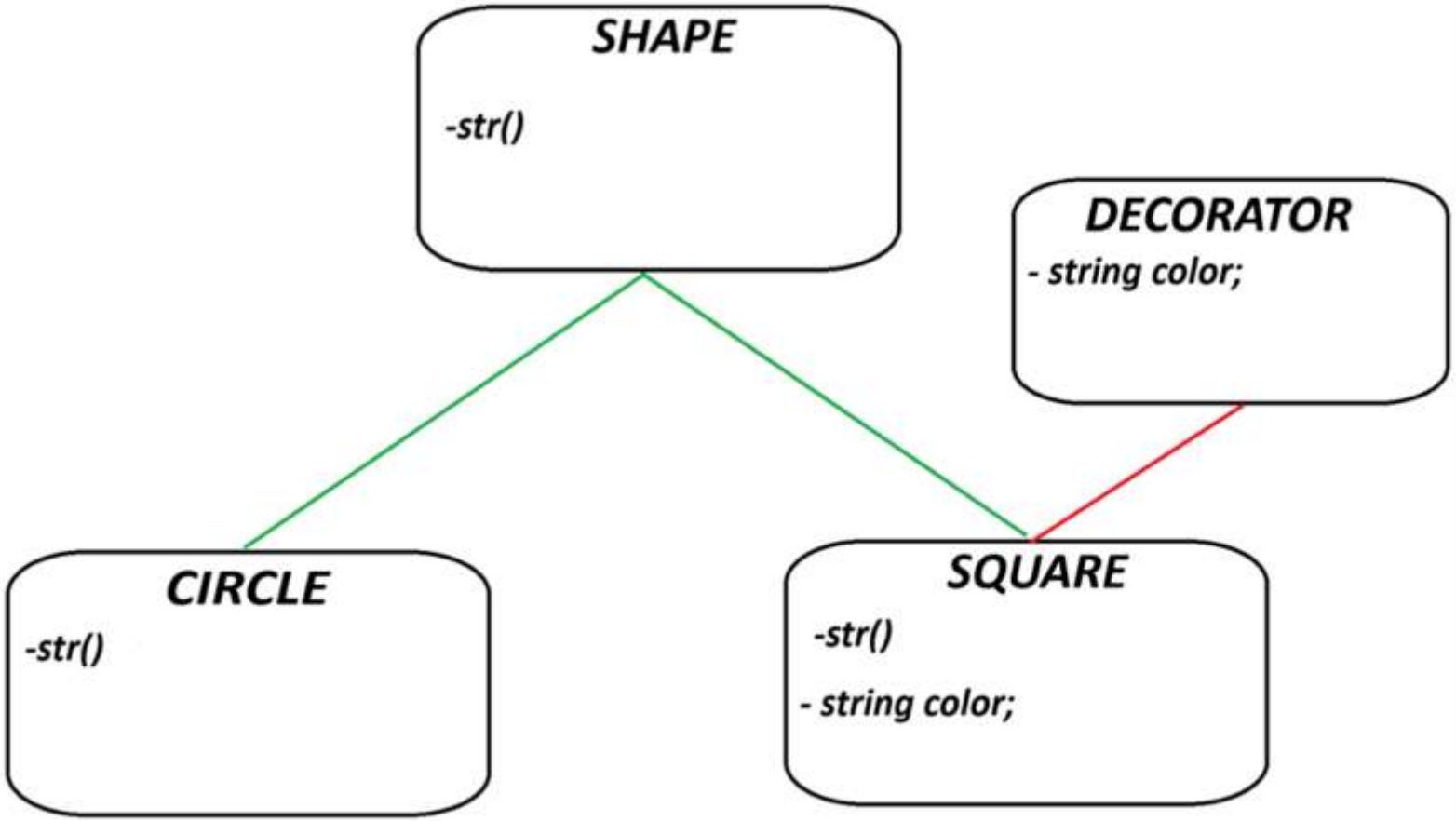
```
struct ColoredShape : Shape
{
    Shape& shape;
    std::string color;

    ColoredShape(Shape& shape, const string& color)
            : shape{shape},
              color{color}
    {
    }

    std::string str() const override
    {
        std::ostringstream oss;
        oss << shape.str() << " has the color " << color;
        return oss.str();
    }
};
```

```
int main() {
    Square square {5};
    ColoredShape red_square{square, "red"};
    std::cout << square.str() << std::endl
    << red_square.str() ;
    return 0;
}
```

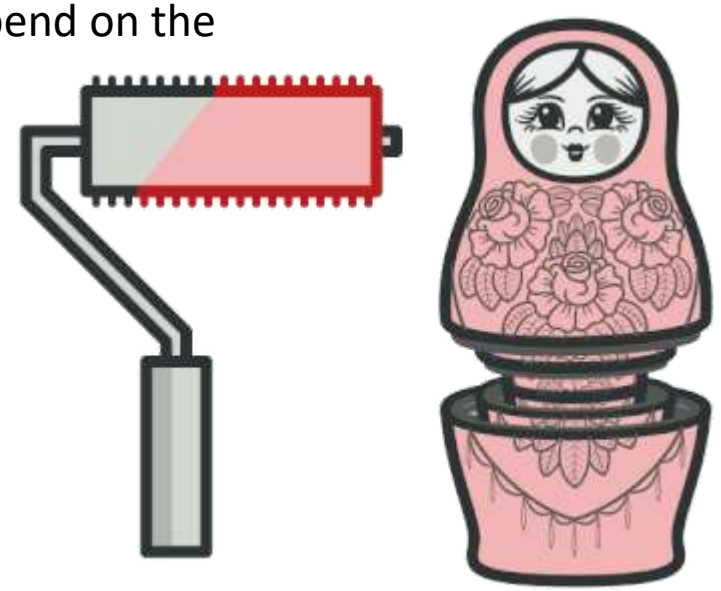
SOLUTION


SHAPE
-str()
DECORATOR
- string color;
CIRCLE
-str()
SQUARE
-str()
- string color;

PROS AND CONS

+ extending an object’s behavior without making a new subclass
+ combining several behaviors by wrapping an object into multiple decorators
+ Single Responsibility Principle.

- hard to remove a specific wrapper from the wrappers stack.
- hard to implement a decorator in such a way that its behavior doesn’t depend on the order in the decorator's stack.
- the initial configuration code of layers might look pretty ugly.



Related design patterns

- Composite
  - Decorator is a modified version of Composite(only one component)
  - Decorator doesn’t aggregate objects
- Strategy
  - Decorator changes the “outside” / Strategy changes the “inside” - of an object
  - Component in Strategy knows about its extensions
- Adapter
  - Decorator only changes responsibilities of an object, not its interface
  - Adapter gives the object a new interface

Thank you for your attention!