Structural Design Pattern

Hung Tran

Fpt software

October 4, 2022

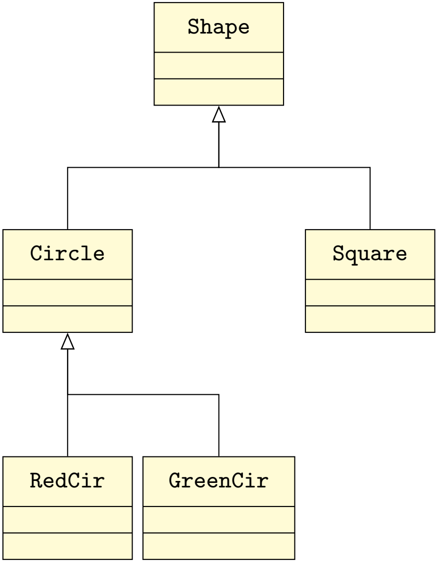
Outline

Structural Pattern Overview

How classes and objects are composed to form larger structure.

- **Adapter**: Convert the interface of a class into another interface.
- **Bridge**: Decouple an abstraction from its implementation.
- **Composite**: Compose objects into tree structure.
- **Decorator**: Attach additional responsibilities to an object dynamically.
- **Facade**: Provide a unified interface to a set of interfaces.
- **Flyweight**: Use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy**: Provide a surrogate or placeholder for another object to control access to it.

Problem Statement



- You have a geometric **Shape** class with a pair of subclasses: **Circle** and **Square**.
- You want to **extend** this class hierarchy to incorporate **colors (red, green)**.
- Adding new shape types and colors to the hierarchy will grow it exponentially.
- The total classes by combination?
- What if lack of virtual destructor?

Problem Statement

- Extending the class hierarchy by inheritance leads to explosion of number of derived classes.
- The **Decorator pattern** allows us to enhance existing types without either **modifying the original types** (Open-Closed Principle) or causing an **explosion of the number of derived types**.
- Inheritance alone does not offer us an efficient way to provide enhancements to shapes, so we must turn to **composition**.

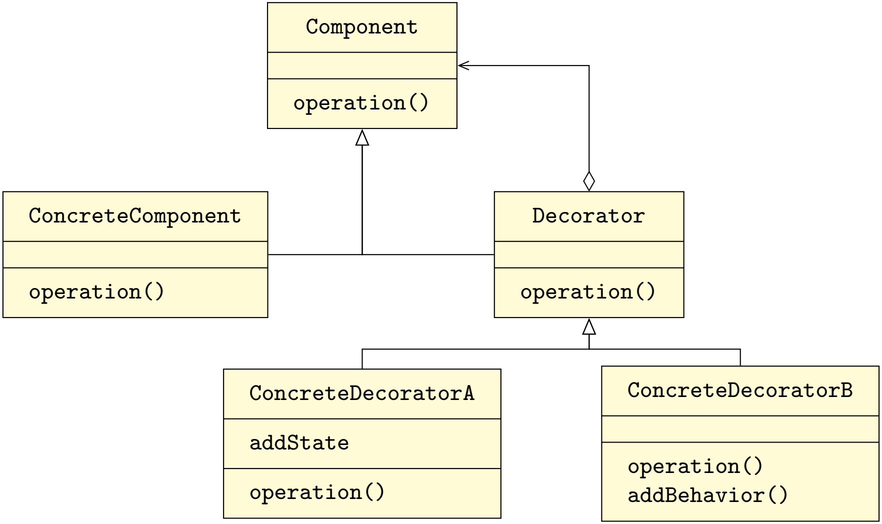
The Intent of Decorator Design Pattern

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

The Decorator Pattern is also known as Wrapper.

Structure of Dynamic Decorator Pattern



Basic implementation: shapes class

shape.h

```cpp
#ifndef _SHAPE_H_
#define _SHAPE_H_
#include <string>

class Shape {
public:
  virtual std::string str() const = 0;
};
#endif // _SHAPE_H_
```

circle.h

```cpp
#ifndef _CIRCLE_H_
#define _CIRCLE_H_

#include "shape.h"

class Circle : public Shape {
  float radius;
public:
  Circle(const float radius);
  void resize(float factor);
  std::string str() const override;
};
#endif // _CIRCLE_H_
```

circle.cpp

```cpp
#include "circle.h"
#include <sstream>

Circle::Circle(const float radius) :
     radius{radius}{

}

void Circle::resize(float factor) {
  radius *= factor;
}

std::string Circle::str() const {
  std::ostringstream oss;
  oss << "A circle of radius " << radius
      ;
  return oss.str();
}
```

Basic implementation: shapes class

coloredShape.h

```cpp
#ifndef _COLORED_SHAPE_H_
#define _COLORED_SHAPE_H_

#include "shape.h"

 class ColoredShape : public Shape {
   Shape& shape;
   std::string color;
 public:
   ColoredShape(Shape& shape, const std::
       string& color);
   std::string str() const override;
 };
#endif // _COLORED_SHAPE_H_
```

coloredShape.cpp

```cpp
#include "coloredShape.h"
#include <sstream>

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

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

Basic implementation: shapes class

transparentShape.h

```cpp
#ifndef _TRANSPARENT_SHAPE_H_
#define _TRANSPARENT_SHAPE_H_

#include "shape.h"

class TransparentShape : public Shape {
  Shape& shape;
  uint8_t transparency;
public:
  TransparentShape(Shape& shape, const
      uint8_t transparency);
  std::string str() const override;
};
#endif // _TRANSPARENT_SHAPE_H_
```

transparentShape.cpp

```cpp
#include "transparentShape.h"
#include <sstream>

TransparentShape::TransparentShape(Shape
     & shape, const uint8_t transparency
     )
  : shape{shape}, transparency{
      transparency} {
}

std::string TransparentShape::str()
      const {
  std::ostringstream oss;
  oss << shape.str() << " has " <<
      static_cast<float>(transparency) /
      \
    255.f * 100.f << "% transparency\n";
  return oss.str();
}
```

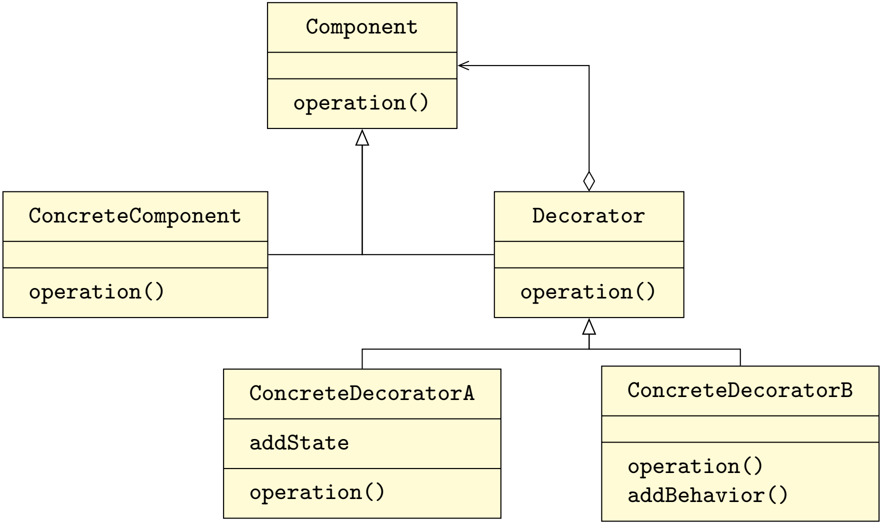
Basic implementation: shapes class

main.cpp.cpp

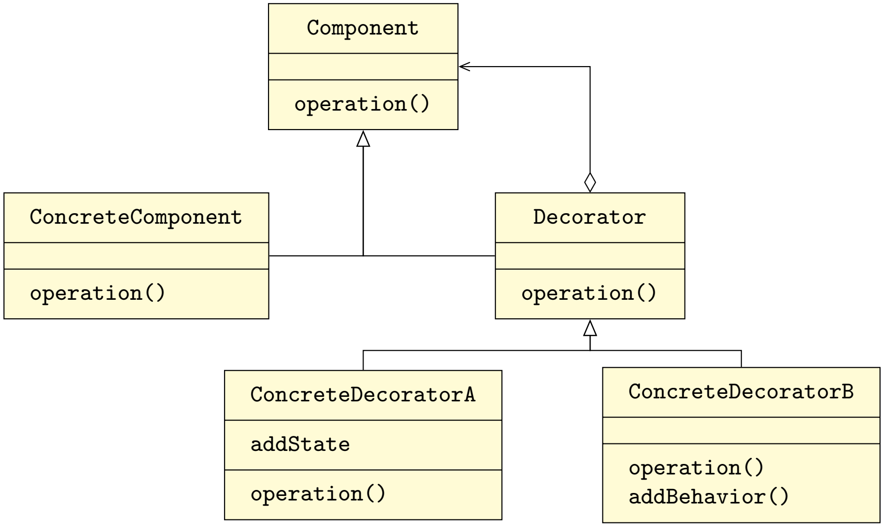
```cpp
#include <iostream>
#include "coloredShape.h"
#include "circle.h"
#include "transparentShape.h"

int main() {
  Circle c{0.5f};
  ColoredShape redC{c, "red"};
  std::cout << redC.str();

  TransparentShape myC{redC, 34};
  //TransparentShape myC{new
     ColoredShape{new Circle{23}, "green
     "}, 64};

  std::cout << myC.str();
  return 0;
}
```

- As we can see resize() method cant not call because it is not part of Shape class.

Static Decorator Pattern



Functional Decorator Pattern



Applicability

- Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.
- The Decorator lets you structure your business logic into layers, create a decorator for each layer and compose objects with various combinations of this logic at runtime. The client code can treat all these objects in the same way, since they all follow a common interface.
- Use the pattern when it's awkward or not possible to extend an object's behavior using inheritance.
- Many programming languages have the final keyword that can be used to prevent further extension of a class. For a final class, the only way to reuse the existing behavior would be to wrap the class with your own wrapper, using the Decorator pattern.

How to Implement

- Make sure your business domain can be represented as a primary component with multiple optional layers over it.
- Figure out what methods are common to both the primary component and the optional layers. Create a component interface and declare those methods there.
- Create a concrete component class and define the base behavior in it.
- Create a base decorator class. It should have a field for storing a reference to a wrapped object. The field should be declared with the component interface type to allow linking to concrete components as well as decorators. The base decorator must delegate all work to the wrapped object.
- Make sure all classes implement the component interface.

Pros and Cons

- You can extend an object's behavior without making a new subclass.
- You can add or remove responsibilities from an object at runtime.
- You can combine several behaviors by wrapping an object into multiple decorators.
- Single Responsibility Principle. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.

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

Relations with Other Patterns

- Adapter changes the interface of an existing object, while Decorator enhances an object without changing its interface. In addition, Decorator supports recursive composition, which isn't possible when you use Adapter.
- Adapter provides a different interface to the wrapped object, Proxy provides it with the same interface, and Decorator provides it with an enhanced interface.
- Chain of Responsibility and Decorator have very similar class structures. Both patterns rely on recursive composition to pass the execution through a series of objects. However, there are several crucial differences.

Relations with Other Patterns

- Composite and Decorator have similar structure diagrams since both rely on recursive composition to organize an open-ended number of objects.
- Designs that make heavy use of Composite and Decorator can often benefit from using Prototype. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.
- Decorator lets you change the skin of an object, while Strategy lets you change the guts.
- Decorator and Proxy have similar structures, but very different intents. Both patterns are built on the composition principle, where one object is supposed to delegate some of the work to another. The difference is that a Proxy usually manages the life cycle of its service object on its own, whereas the composition of Decorators is always controlled by the client.

Thank You!