Decorator pattern

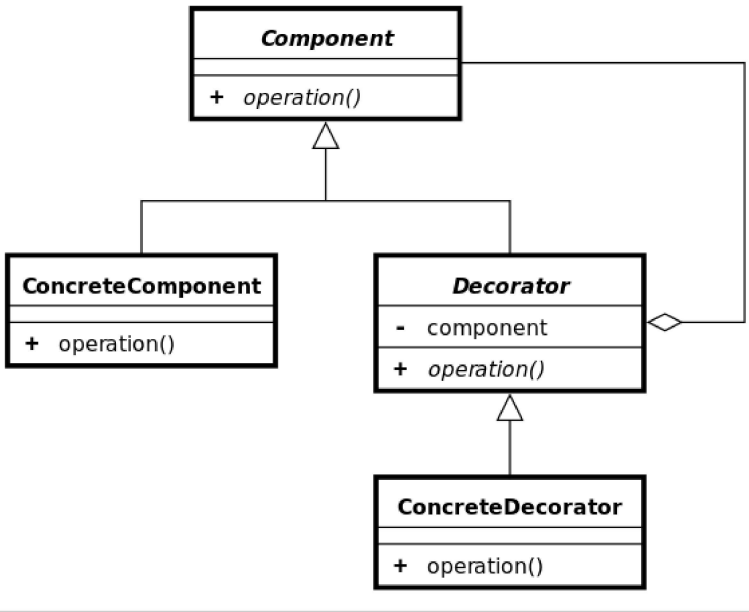
From Wikipedia, the free encyclopedia

In object-oriented programming, the **decorator pattern** (also known as Wrapper, an alternative naming shared with the Adapter pattern) is a design pattern that allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class.[1] The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern.[2]

Contents

Intent

The decorator pattern can be used to extend (decorate) the functionality of a certain object statically, or in some cases at run-time, independently of other instances of the same class, provided some groundwork is done at design time. This is achieved by designing a new *Decorator* class that wraps the original class. This wrapping could be achieved by the following sequence of steps:



Decorator UML class diagram

1. Subclass the original "Component" class into a *Decorator* class (see UML diagram);
2. In the *Decorator* class, add a Component pointer as a field;
3. Pass a Component to the *Decorator* constructor to initialize the Component pointer;
4. In the *Decorator* class, forward all "Component" methods to the "Component" pointer; and
5. In the ConcreteDecorator class, override any Component method(s) whose behavior needs to be

modified.

This pattern is designed so that multiple decorators can be stacked on top of each other, each time adding a new functionality to the overridden method(s).

Note that decorators and the original class object share a common set of features. In the previous diagram, the "operation()" method was available in both the decorated and undecorated versions.

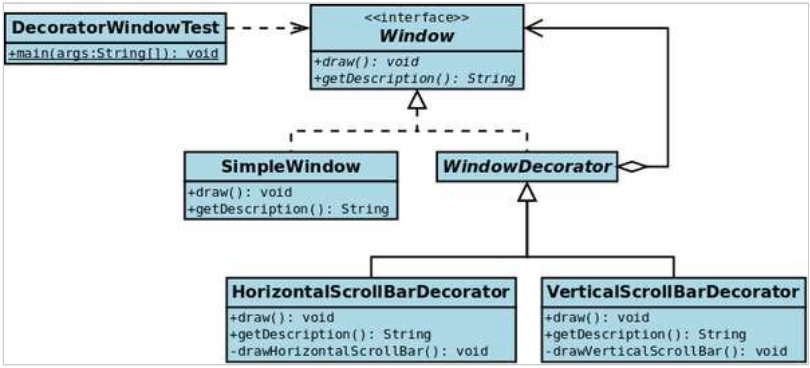
The decoration features (e.g., methods, properties, or other members) are usually defined by an interface, mixin (a.k.a. "trait") or class inheritance which is shared by the decorators and the decorated object. In the previous example the class "Component" is inherited by both the "ConcreteComponent" and the subclasses that descend from *Decorator*.

The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at run-time for selective objects.

This difference becomes most important when there are several *independent* ways of extending functionality. In some object-oriented programming languages, classes cannot be created at runtime, and it is typically not possible to predict, at design time, what combinations of extensions will be needed. This would mean that a new class would have to be made for every possible combination. By contrast, decorators are objects, created at runtime, and can be combined on a per-use basis. The I/O Streams implementations of both Java and the .NET Framework incorporate the decorator pattern.

Motivation

As an example, consider a window in a windowing system. To allow scrolling of the window's contents, one may wish to add horizontal or vertical scrollbars to it, as appropriate. Assume windows are represented by instances of the *Window* class, and assume this class has no functionality for adding scrollbars. One could create a subclass *ScrollingWindow* that provides them, or create a *ScrollingWindowDecorator* that adds this functionality to existing *Window* objects. At this point, either solution would be fine.



UML diagram for the window example

Now, assume one also desires the ability to add borders to windows. Again, the original *Window* class has no support. The *ScrollingWindow* subclass now poses a problem, because it has effectively created a new kind of window. If one wishes to add border support to many but not *all* windows, one must create subclasses *WindowWithBorder* and *ScrollingWindowWithBorder* etc. This problem gets worse with every new feature or window subtype to be added. For the decorator solution, we simply create a new *BorderedWindowDecorator*—at runtime, we can decorate existing windows with the *ScrollingWindowDecorator* or the *BorderedWindowDecorator* or both, as we see fit. Notice that if the functionality needs to be added to all Windows, you could modify the base class and that will do. On the other hand, sometimes (e.g., using external frameworks) it is not possible, legal, or convenient to modify the base class.

Note, in the previous example, that the "SimpleWindow" and "WindowDecorator" classes implement the "Window" interface, which defines the "draw()" method and the "getDescription()" method, that are required in this scenario, in order to decorate a window control.

Examples

C++

Two options are presented here, first a dynamic, runtime-composable decorator (has issues with calling decorated functions unless proxied explicitly) and a decorator that uses mixin inheritance.

Dynamic Decorator

```
struct Shape
{
  virtual string str() = 0;
};
struct Circle : Shape
{
  float radius;
  void resize(float factor) { radius *= factor; }
  string str() override
  {
    return string("A circle of radius") + lexical_cast<string>(radius);
  }
};
struct ColoredShape : Shape
{
  string color;
  Shape& shape;
  string str() override
  {
    return shape.str() + string(" which is ") + color;
  }
};

// usage:
Circle c{123};
ColoredShape cc{"red", c};
cout << cc.str() << endl;
// cannot call this:
cc.resize(1.2); // not part of ColoredShape
```

Static Decorator (Mixin Inheritance)

```
struct Circle
{
  float radius;
  void resize(float factor) { radius *= factor; }
  string str()
  {
    return string("A circle of radius") + lexical_cast<string>(radius);
  }
};

template <typename T> struct ColoredShape : T
{
  string color;
  string str()
  {
    return T::str() + " is colored " + color;
  }
};

// usage:
ColoredShape<Circle> red_circle{"red"};
cout << red_circle.str() << endl;
// and this is legal
red_circle.resize(1.2);
```

Java

First example (window/scrolling scenario)

The following Java example illustrates the use of decorators using the window/scrolling scenario.

```
// The Window interface class
public interface Window {
    public void draw(); // Draws the Window
    public String getDescription(); // Returns a description of the Window
}

// Implementation of a simple Window without any scrollbars
class SimpleWindow implements Window {
    public void draw() {
        // Draw window
    }

    public String getDescription() {
        return "simple window";
    }
}
```

The following classes contain the decorators for all `Window` classes, including the decorator classes themselves.

```
// abstract decorator class - note that it implements Window
abstract class WindowDecorator implements Window {
    protected Window windowToBeDecorated; // the Window being decorated

    public WindowDecorator (Window windowToBeDecorated) {
        this.windowToBeDecorated = windowToBeDecorated;
    }
    public void draw() {
        windowToBeDecorated.draw(); //Delegation
    }
    public String getDescription() {
        return windowToBeDecorated.getDescription(); //Delegation
    }
}

// The first concrete decorator which adds vertical scrollbar functionality
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window windowToBeDecorated) {
        super(windowToBeDecorated);
    }

    @Override
    public void draw() {
        super.draw();
        drawVerticalScrollBar();
    }

    private void drawVerticalScrollBar() {
        // Draw the vertical scrollbar
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", including vertical scrollbars";
    }
}

// The second concrete decorator which adds horizontal scrollbar functionality
class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator (Window windowToBeDecorated) {
        super(windowToBeDecorated);
    }

    @Override
    public void draw() {
        super.draw();
```

```
        drawHorizontalScrollBar();
    }

    private void drawHorizontalScrollBar() {
        // Draw the horizontal scrollbar
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", including horizontal scrollbars";
    }
}
```

Here's a test program that creates a `Window` instance which is fully decorated (i.e., with vertical and horizontal scrollbars), and prints its description:

```
public class DecoratedWindowTest {
    public static void main(String[] args) {
        // Create a decorated Window with horizontal and vertical scrollbars
        Window decoratedWindow = new HorizontalScrollBarDecorator (
                new VerticalScrollBarDecorator (new SimpleWindow()));

        // Print the Window's description
        System.out.println(decoratedWindow.getDescription());
    }
}
```

Below is the JUnit test class for the Test Driven Development

```
import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class WindowDecoratorTest {
    @Test
    public void testWindowDecoratorTest() {
        Window decoratedWindow = new HorizontalScrollBarDecorator(new VerticalScrollbarDecorator(new SimpleWi
            // assert that the description indeed includes horizontal + vertical scrollbars
            assertEquals("simple window, including vertical scrollbars, including horizontal scrollbars", dec
    }
}
```

The output of this program is "simple window, including vertical scrollbars, including horizontal scrollbars". Notice how the `getDescription` method of the two decorators first retrieve the decorated `Window`'s description and *decorates* it with a suffix.

Second example (coffee making scenario)

The next Java example illustrates the use of decorators using coffee making scenario. In this example, the scenario only includes cost and ingredients.

```
// The interface Coffee defines the functionality of Coffee implemented by decorator
public interface Coffee {
    public double getCost(); // Returns the cost of the coffee
    public String getIngredients(); // Returns the ingredients of the coffee
}

// Extension of a simple coffee without any extra ingredients
public class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 1;
```

```
    }

    @Override
    public String getIngredients() {
        return "Coffee";
    }
}
```

The following classes contain the decorators for all `Coffee` classes, including the decorator classes themselves..

```
// Abstract decorator class - note that it implements Coffee interface
public abstract class CoffeeDecorator implements Coffee {
    protected final Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee c) {
        this.decoratedCoffee = c;
    }

    public double getCost() { // Implementing methods of the interface
        return decoratedCoffee.getCost();
    }

    public String getIngredients() {
        return decoratedCoffee.getIngredients();
    }
}

// Decorator WithMilk mixes milk into coffee.
// Note it extends CoffeeDecorator.
class WithMilk extends CoffeeDecorator {
    public WithMilk(Coffee c) {
        super(c);
    }

    public double getCost() { // Overriding methods defined in the abstract superclass
        return super.getCost() + 0.5;
    }

    public String getIngredients() {
        return super.getIngredients() + ", Milk";
    }
}

// Decorator WithSprinkles mixes sprinkles onto coffee.
// Note it extends CoffeeDecorator.
class WithSprinkles extends CoffeeDecorator {
    public WithSprinkles(Coffee c) {
        super(c);
    }

    public double getCost() {
        return super.getCost() + 0.2;
    }

    public String getIngredients() {
        return super.getIngredients() + ", Sprinkles";
    }
}
```

Here's a test program that creates a `Coffee` instance which is fully decorated (with milk and sprinkles), and calculate cost of coffee and prints its ingredients:

```
public class Main {
    public static void printInfo(Coffee c) {
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());
    }
```

```
    public static void main(String[] args) {
        Coffee c = new SimpleCoffee();
        printInfo(c);

        c = new WithMilk(c);
        printInfo(c);

        c = new WithSprinkles(c);
        printInfo(c);
    }
}
```

The output of this program is given below:

```
Cost: 1.0; Ingredients: Coffee
Cost: 1.5; Ingredients: Coffee, Milk
Cost: 1.7; Ingredients: Coffee, Milk, Sprinkles
```

PHP

```
abstract class Component
{
    protected $data;
    protected $value;

    abstract public function getData();

    abstract public function getValue();
}

class ConcreteComponent extends Component
{
    public function __construct()
    {
        $this->value = 1000;
        $this->data = "Concrete Component:\t{$this->value}\n";
    }

    public function getData()
    {
        return $this->data;
    }

    public function getValue()
    {
        return $this->value;
    }
}

abstract class Decorator extends Component
{

}

class ConcreteDecorator1 extends Decorator
{
    public function __construct(Component $data)
    {
        $this->value = 500;
        $this->data = $data;
    }

    public function getData()
    {
        return $this->data->getData() . "Concrete Decorator 1:\t{$this->value}\n";
    }

    public function getValue()
    {
```

```
        return $this->value + $this->data->getValue();
    }
}

class ConcreteDecorator2 extends Decorator
{
    public function __construct(Component $data)
    {
        $this->value = 500;
        $this->data = $data;
    }

    public function getData()
    {
        return $this->data->getData() . "Concrete Decorator 2:\t{$this->value}\n";
    }

    public function getValue()
    {
        return $this->value + $this->data->getValue();
    }
}

class Client
{
    private $component;

    public function __construct()
    {
        $this->component = new ConcreteComponent();
        $this->component = $this->wrapComponent($this->component);

        echo $this->component->getData();
        echo "Client:\t\t\t";
        echo $this->component->getValue();
    }

    private function wrapComponent(Component $component)
    {
        $component1 = new ConcreteDecorator1($component);
        $component2 = new ConcreteDecorator2($component1);
        return $component2;
    }
}

$client = new Client();

// Result: #quanton81

//Concrete Component:   1000
//Concrete Decorator 1: 500
//Concrete Decorator 2: 500
//Client:               2000
```

See also

- Composite pattern
- Adapter pattern
- Abstract class
- Abstract factory
- Aspect-oriented programming
- Immutable object

References

1. Gamma, Erich; et al. (1995). *Design Patterns*. Reading, MA: Addison-Wesley Publishing Co, Inc. pp. 175ff. ISBN 0-201-63361-2.

2. "How to Implement a Decorator Pattern".

External links

- Decorator pattern description from the Portland Pattern Repository (http://c2.com/cgi/wiki?DecoratorPattern)
- Decorator pattern C++11 implementation example (http://patterns.pl/decorator.html)

The Wikibook *Computer Science Design Patterns* has a page on the topic of: ***Decorator implementations in various languages***

Retrieved from "https://en.wikipedia.org/w/index.php?title=Decorator_pattern&oldid=725547701"

Categories: Software design patterns
