**Decorator Pattern**

**Motivation**

Extending an object’s functionality can be done statically (at compile time) by using inheritance however it might be necessary to extend an object’s functionality dynamically (at runtime) as an object is used.

The **Decorator** pattern is an alternative to subclassing which favors object composition over class inheritance.

A common issue of subclassing is its rigidity: when different subclasses provide different features, it's hard to combine them into a unique object.

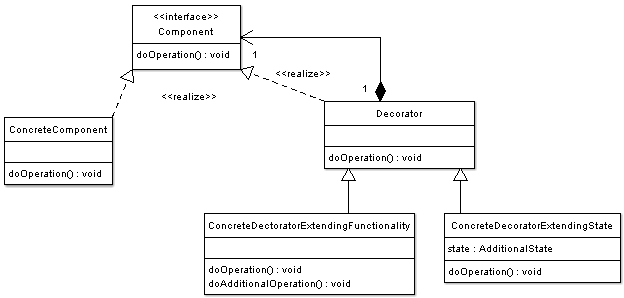
A classic example consists in considering a Ball class and two BouncingBall and ColoredBall subclasses. What should I instantiate if I want a ColoredBouncingBall? There are different solutions that fall in the subclassing realm:

* One more subclass is created; often multiple inheritance is not allowed and thus some code from BouncingBall or ColoredBall would be duplicated.
* If the desired ball traits are three or more, for instance because a RollingBall class exists, there would be a proliferation of subclasses: ColoredBouncingBall, ColoredRollingBall, BouncingRollingBall and ColoredBouncingRollingBall. If you add a business requirement (the client wants rolling balls) and the codebase explodes, this is a sign that something is wrong in the design: when two different balls are requested, we should *take the first bullet* and refactor into a Decorator pattern, where every other trait will require only one additional class instead of doubling the number of concrete classes.
* We can also prepare a God class that puts together the different Colored and Bouncing behaviors, but remember that a class should have only one responsibility. Why adding traits that sometimes will never be used to an object? This solution forces all Balls to be an instance of BouncingColoredRollingBorderedFlashingBall.

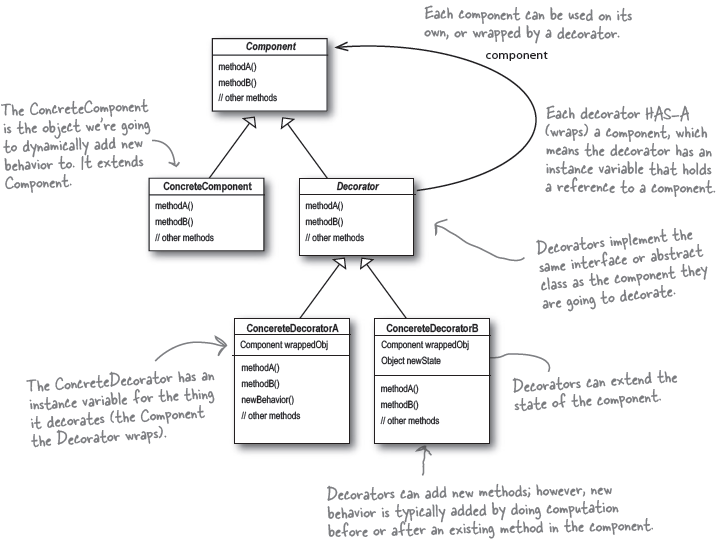
**Intent**

* The intent of this pattern is to add additional responsibilities dynamically to an object.

**Implementation**

The figure below shows a UML class diagram for the Decorator Pattern:  
The participants classes in the decorator pattern are:

* **Component** - Interface for objects that can have responsibilities added to them dynamically.
* **ConcreteComponent** - Defines an object to which additional responsibilities can be added.
* **Decorator** - Maintains a reference to a Component object and defines an interface that conforms to Component's interface.
* **Concrete Decorators** - Concrete Decorators extend the functionality of the component by adding state or adding behavior.



**Applicability & Examples**

The Decorator pattern is a standard response to a big subclassing tree. It can often be viewed as a particular Composite pattern, whose Uml diagram is very similar, when there is only one *Component* and Composites (*ConcreteDecorators*) adds functionalities instead of aggregating various objects. In fact, they compose only one instance of Component, instead of subclassing it.

Their job is to delegate all the methods of the Component interface to the wrapped Component instance, and to decorate some of them with additional behavior. Since the wrapping and delegation code is common to all the ConcreteDecorators of the same Component, it is usually shared in a base class.

Since Decorators conform to Component, the Client does not notice if a ConcreteComponent is substituted with a Decorator composing it, or with a Decorator composing a Decorator composing it.

**Example**

<?php

**abstract class** Beverage{

**protected** $description = "Unknown Beverage";

**public function** getDescription(){

**return** $this->description;

}

**public abstract function** cost();

}

**abstract class** CondimentDecorator **extends** Beverage{

**public function** getDescription(){

**throw new** Exception("getDescription must be overridden");

}

}

**class** DarkRoast **extends** Beverage{

**public function** \_\_construct(){

$this->description = "Dark Roast";

}

**public function** cost(){

**return** 0.99;

}

}

**class** Decaf **extends** Beverage{

**public function** \_\_construct(){

$this->description = "Decaf";

}

**public function** cost(){

**return** 1.05;

}

}

**class** Espresso **extends** Beverage{

**public function** \_\_construct(){

$this->description = "Espresso";

}

**public function** cost(){

**return** 1.99;

}

}

**class** HouseBlend **extends** Beverage{

**public function** \_\_construct(){

$this->description = "House Blend Coffee";

}

**public function** cost(){

**return** 0.89;

}

}

**class** Milk **extends** CondimentDecorator{

**private** $beverage;

**public function** \_\_construct(Beverage $beverage){

$this->beverage = $beverage;

}

**public function** getDescription(){

**return** $this->beverage->getDescription() . ", Steamed Milk";

}

**public function** cost(){

**return** 0.10 + $this->beverage->cost();

}

}

**class** Mocha **extends** CondimentDecorator{

**private** $beverage;

**public function** \_\_construct(Beverage $beverage){

$this->beverage = $beverage;

}

**public function** getDescription(){

**return** $this->beverage->getDescription() . ", Mocha";

}

**public function** cost(){

$cost = $this->beverage->cost();

**return** 0.20 + $this->beverage->cost();

}

}

**class** Soy **extends** CondimentDecorator{

**private** $beverage;

**public function** \_\_construct(Beverage $beverage){

$this->beverage = $beverage;

}

**public function** getDescription(){

**return** $this->beverage->getDescription() . ", Soy";

}

**public function** cost(){

**return** 0.15 + $this->beverage->cost();

}

}

**class** Whip **extends** CondimentDecorator{

**private** $beverage;

**public function** \_\_construct(Beverage $beverage){

$this->beverage = $beverage;

}

**public function** getDescription(){

**return** $this->beverage->getDescription() . ", Whip";

}

**public function** cost(){

**return** 0.10 + $this->beverage->cost();

}

}

**class** StarbuzzCoffee{

**public static function** *main*(){

$beverage = **new** Espresso();

**echo**($beverage->getDescription() . " $" . $beverage->cost());

$beverage2 = **new** DarkRoast();

$beverage2 = **new** Mocha($beverage2);

$beverage2 = **new** Mocha($beverage2);

$beverage2 = **new** Whip($beverage2);

**echo**("<br />" . $beverage2->getDescription() . " $" . $beverage2->cost());

$beverage3 = **new** DarkRoast();

$beverage3 = **new** Soy($beverage3);

$beverage3 = **new** Mocha($beverage3);

$beverage3 = **new** Whip($beverage3);

**echo**("<br />" . $beverage3->getDescription() . " $" . $beverage3->cost());

}

}

StarbuzzCoffee::*main*();

**Related Patterns**

* **Adapter Pattern** - A decorator is different from an adapter in that a decorator changes object's responsibilities, while an adapter changes an object interface.
* **Composite Pattern** - A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities.

**Consequences**

* Decoration is more convenient for adding functionalities to objects instead of entire classes at runtime. With decoration it is also possible to remove the added functionalities dynamically.
* Decoration adds functionality to objects at runtime which would make debugging system functionality harder.