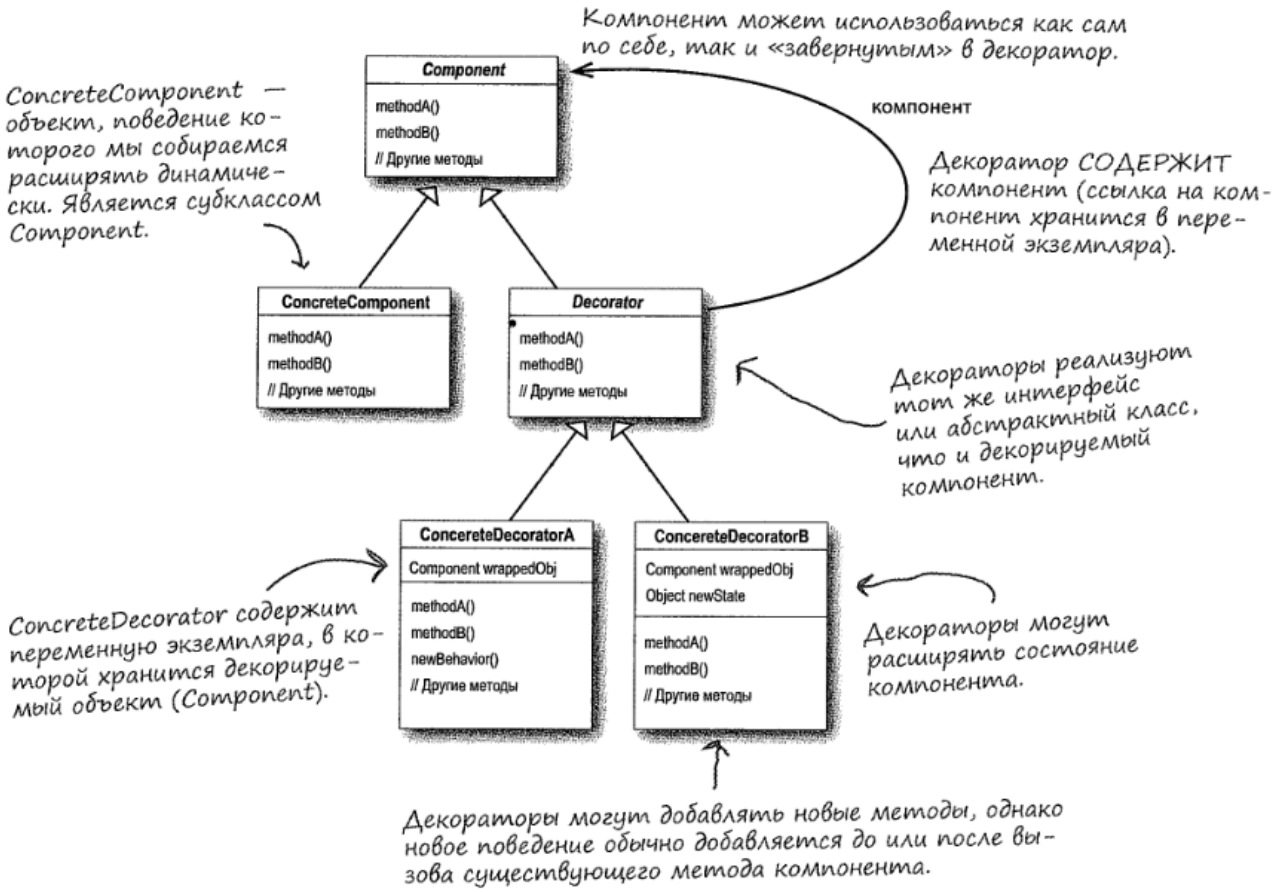
Декоратор

Декоратор — бинимически наделяет объект новыми возможностями и является гибкой альтернативой субклассированию в области расширения функциональности

- Типы декораторов соответвуют типам декорируемых компонентов(соответвие достигается посредством наследования или реализации интерфейса)
- Декоратры изменяют поведение компонентов, добавляя новую функциональность.
- Компонент  может декорироваться любым количесвом декораторов
- Декораторы обычно прозрачны для клиентов компонента.
- Классы должны быть открыты для расширения, но закрыты для измения

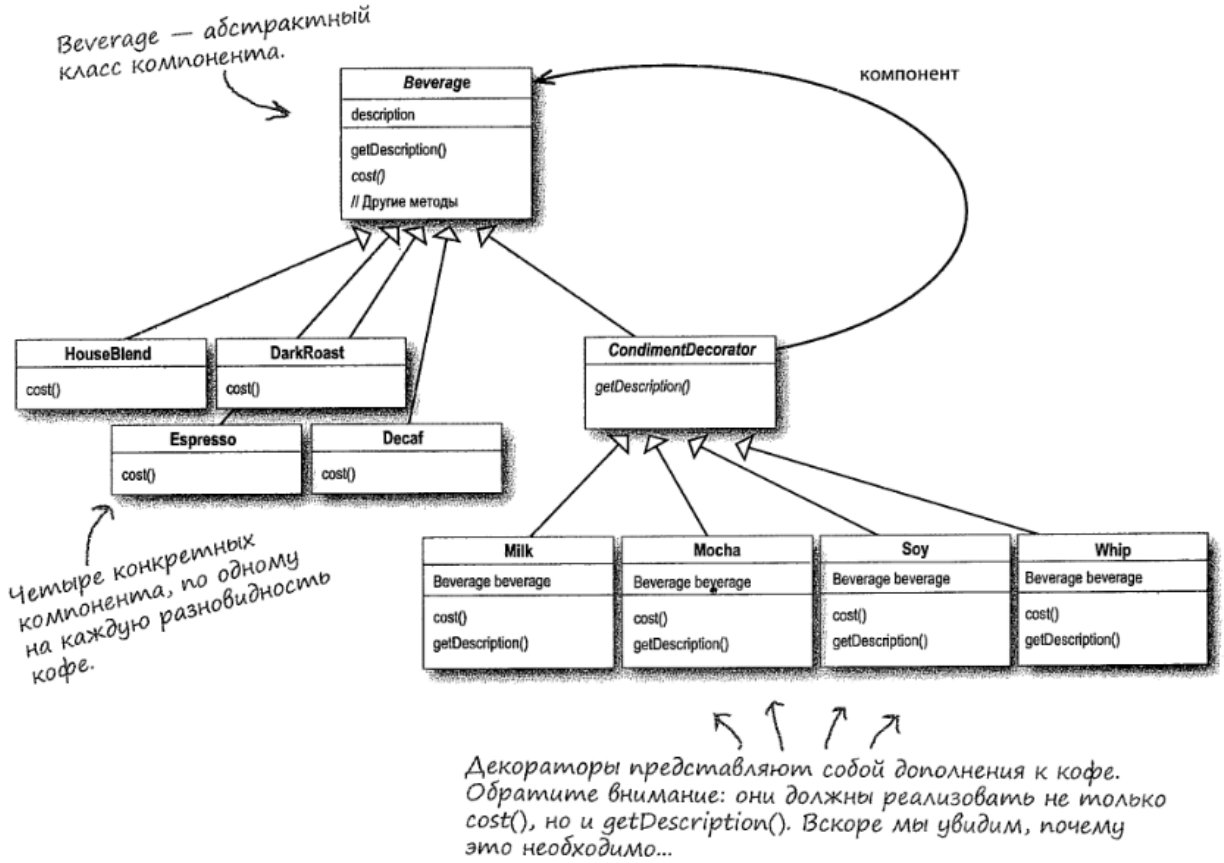



Применимость
Используйте паттерн декоратор:

- Для динамического, прозрачного для клиентов добавления обязанностей объектам;
- Для реализации обязанностей, которые могут быть сняты с объекта;
- Когда расширение путем порождения подклассов по каким-то причинам неудобно или невозможно.Иногда приходится реализовывать много независимых расширений, так что порождение подклассов для поддержки всех возможных комбинаций приведет к комбинаторному росту их числа. В других случаях определение класса может быть скрыто или почему-либо еще недоступно, так что породить от него подкласс нельзя.

Результаты
У паттерна декоратор есть, по крайней мере, два плюса и два минуса:

- Большая гибкость, нежели у статического наследования.
- Позволяет избежать перегруженных функциями классов на верхних уровнях иерархии.
- Декоратор и его компонент не идентичны.
- Множество мелких объектов. (При использовании в проекте паттерна декоратор нередко получается система, составленная из большого числа мелких объектов, которые похожи друг на друга различаются только способом взаимосвязи, а не классом и не значениями своих внутренних переменных. Хотя проектировщик, разбирающийся в устройстве такой системы, может легко настроить ее, но изучать и отлаживать ее очень тяжело.)




```
#include <iostream>
#include <string.h>
using  namespace std;
//абстрактный напиток
class  Beverage
{
protected:
    string decription="Unknown Beverage";
public:
    virtual string getDescriprion()
    {
        return decription;
    }
    virtual double cost()=0;
```

```
};
class CondimentDecorator:public Beverage
{
public:
    virtual string getDescriprion()=0;
};
class Espresso:public Beverage
{
public:
    Espresso()
    {
        decription="Espresso";
    }
    double  cost()
    {
        return 1.99;
    }
};
class HouseBlend:public Beverage
{
public:
    HouseBlend()
    {
        decription="House Blend";
    }
    double  cost()
    {
        return 1.09;
    }
};
class Milk:public CondimentDecorator
{
private:
    Beverage* beverage;
public:
    Milk(Beverage &bg):beverage(&bg) {}
    string getDescriprion()
    {
        return  beverage->getDescriprion()+" ,Milk";
    }
    double cost()
    {
        return 0.50+beverage->cost();
    }
};
class Whip:public CondimentDecorator
{
private:
    Beverage* beverage;
public:
    Whip(Beverage &bg):beverage(&bg) {}
    string getDescriprion()
    {
        return  beverage->getDescriprion()+" ,Whip";
```

```
    }
    double cost()
    {
        return 0.50+beverage->cost();
    }
};
int main() {
    Beverage* br2=new HouseBlend();
    br2=new Milk(*br2);
    br2=new Whip(*br2);
    br2->cost();
    std::cout << br2->cost() <<" "<< br2->getDescriprion() <<
std::endl;
    return 0;
}
```