

# Паттерн декоратор

Наумов Д.А., доц. каф. КТ

Основы программной инженерии, 2019

## 1 Паттерн Декторатор

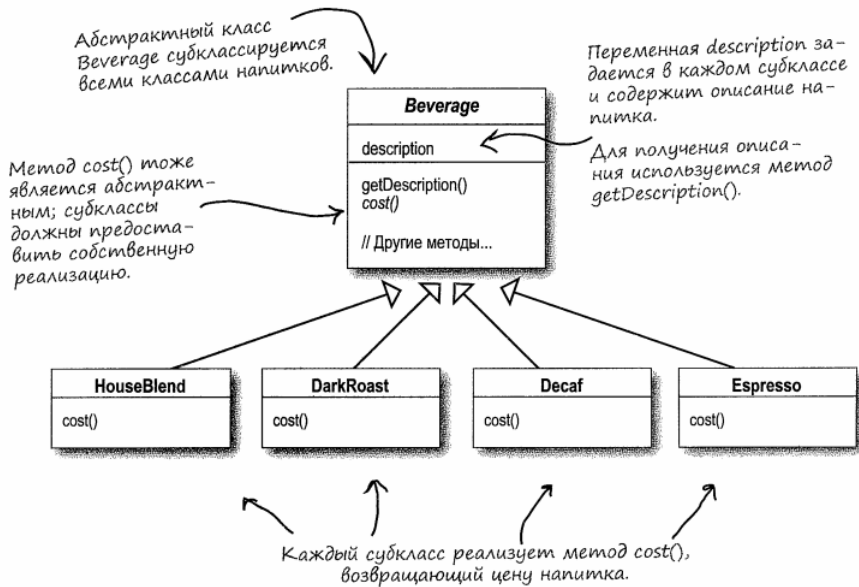
## Добро пожаловать в Starbuzz!

Сеть кофеен Starbuzz стремительно развивается. Если вы увидите одну из этих кофеен на углу, посмотрите через дорогу — и вы наверняка увидите другую.

Из-за бурного роста руководству Starbuzz никак не удастся привести свою систему заказов в соответствие с реальным ассортиментом.

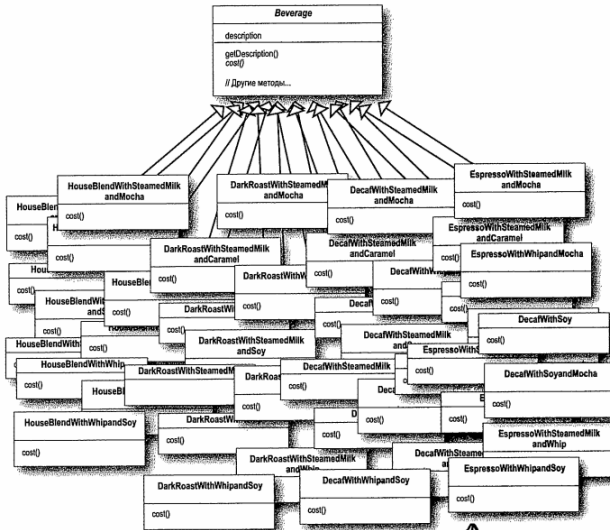
Когда бизнес только начинался, иерархия классов выглядела примерно так...





**К кофе можно заказать различные дополнения (пенка, шоколад и т. д.), да еще украсить все сверху взбитыми сливками. Дополнения не бесплатны, поэтому они должны быть встроены в систему оформления заказов.**

**Первая попытка...**

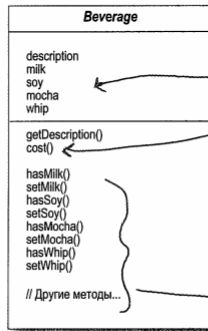


Каждый метод cost вычисляет стоимость кофе вместе со всеми дополнениями.



Глупо; зачем нужны все эти классы? Разве для отслеживания дополнений нельзя использовать переменные экземпляров в суперклассе и наследовании?

Давайте попробуем. Начнем с базового класса Beverage и добавим переменные, которые указывают, присутствует ли в кофе то или иное дополнение...



Логическая переменная для каждого дополнения.

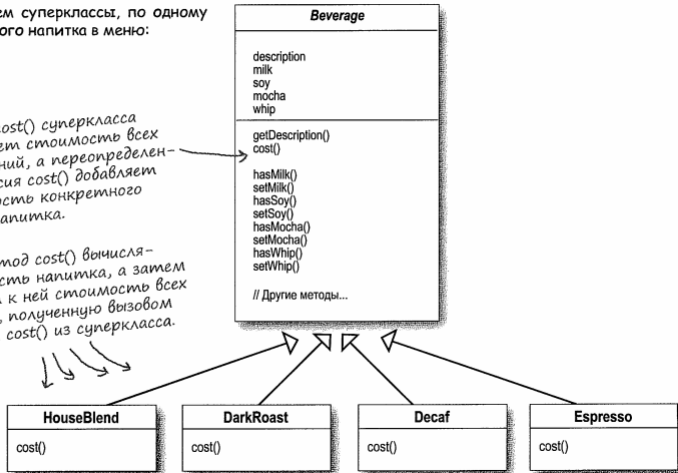
Теперь мы реализуем cost() в Beverage, чтобы метод вычислял суммарную стоимость напитка вместе со всеми дополнениями. Субклассы по-прежнему переопределяют cost(), но они также вызывают версию суперкласса для вычисления общей стоимости базового напитка со всеми дополнениями.

Эти методы читают и устанавливают флаги дополнений.

Добавляем суперклассы, по одному для каждого напитка в меню:

Версия `cost()` суперкласса вычисляет стоимость всех дополнений, а переопределенная версия `cost()` добавляет стоимость конкретного типа напитка.

Каждый метод `cost()` вычисляет стоимость напитка, а затем прибавляет к ней стоимость всех дополнений, полученную вызовом реализации `cost()` из суперкласса.





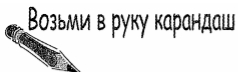
# Упражнение

Возьми в руку карандаш



Напишите реализации `cost()` для следующих классов : Beverage  
DarkRoast





Какие изменения требований или других факторов могут отразиться на работоспособности этой архитектуры?

---

*Изменение цены дополнений потребует модификации существующего кода.*

---

*При появлении новых дополнений нам придется добавлять новые методы и изменять реализацию `cost` в суперклассе.*

---

*Для некоторых новых напитков (холодный чай?) дополнения могут оказаться неуместными, но subclass `Tea` все равно будет наследовать `hasWhip()` и другие методы.*

---

*А если клиент захочет двойную порцию шоколада?*

---

*Ваша очередь:* \_\_\_\_\_

---

## Принцип открытости/закрытости



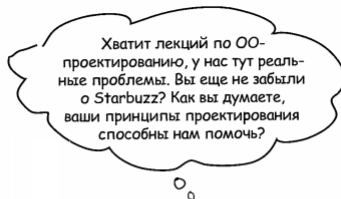
### Принцип проектирования

Классы должны быть открыты для расширения, но закрыты для изменения.



## Знакомство с паттерном Декоратор

- 1 берем объект DarkRoast;
- 2 декорируем его объектом Mocha;
- 3 декорируем его объектом Whip;
- 4 вызываем метод cost() и пользуемся делегированием для прибавления стоимости дополнений.



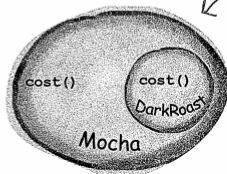
## Построение заказанного напитка

- 1 Начинаем с объекта DarkRoast.



Напоминаем, что DarkRoast наследует от Beverage и содержит метод `cost()` для вычисления стоимости напитка.

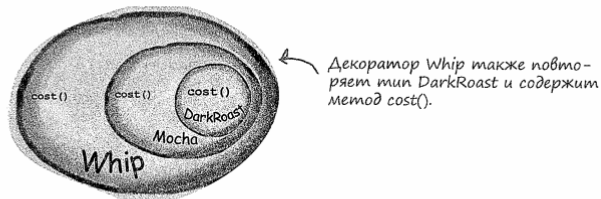
- 2 Клиент заказывает шоколад, поэтому мы создаем объект Mocha и «заворачиваем» в него DarkRoast.



Объект Mocha является декоратором. Его тип повторяет тип декорируемого объекта — в данном случае Beverage.

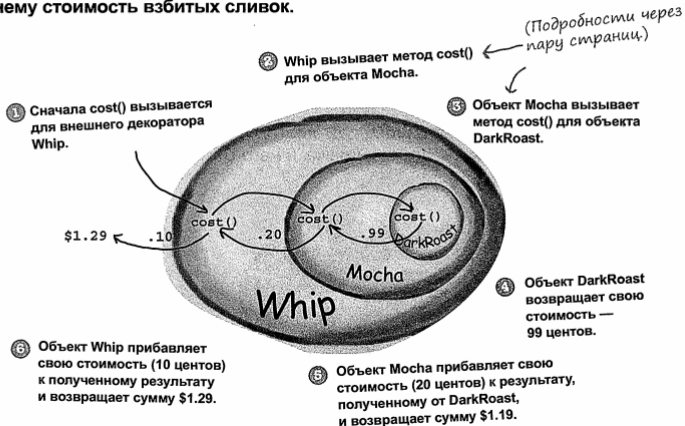
Объект Mocha тоже содержит метод `cost()`, а благодаря полиморфизму он может интерпретироваться как Beverage (так как Mocha является subclassом Beverage).

- 3 Клиент также хочет взбитые сливки, поэтому мы создаем объект Whip и «заворачиваем» в него Mocha.



Таким образом, объект DarkRoast, «завернутый» в Mocha и Whip, сохраняет признаки Beverage, и с ним можно делать все, что можно делать с DarkRoast, включая вызов метода cost().

- 4 Пришло время вычислить общую стоимость напитка. Для этого мы вызываем метод `cost()` внешнего декоратора `Whip`, а последний делегирует вычисление декорируемым объектам. Получив результат, он прибавляет к нему стоимость взбитых сливок.



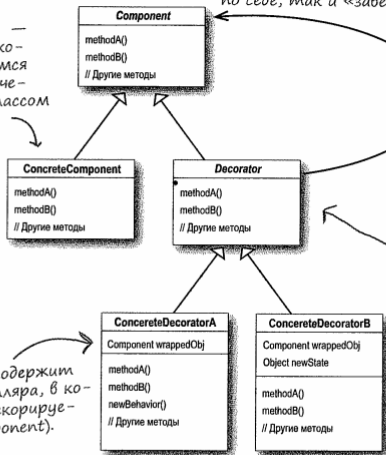


## Что нам уже известно...

- Декораторы имеют тот же супертип, что и декорируемые объекты.
- Объект можно «завернуть» в один или несколько декораторов.
- Так как декоратор относится к тому же супертипу, что и декорируемый объект, мы можем передать декорированный объект вместо исходного.
- Декоратор добавляет свое поведение до и (или) после делегирования операций декорируемому объекту, выполняющему остальную работу.
- Объект может быть декорирован в любой момент времени, так что мы можем декорировать объекты динамически и с произвольным количеством декораторов.

*Очень  
важно!*

*ConcreteComponent — объект, поведение которого мы собираемся расширять динамически. Является subclasses Component.*



*Компонент может использоваться как сам по себе, так и «завернутым» в декоратор.*

компонент

*Декоратор СОДЕРЖИТ компонент (ссылка на компонент хранится в переменной экземпляра).*

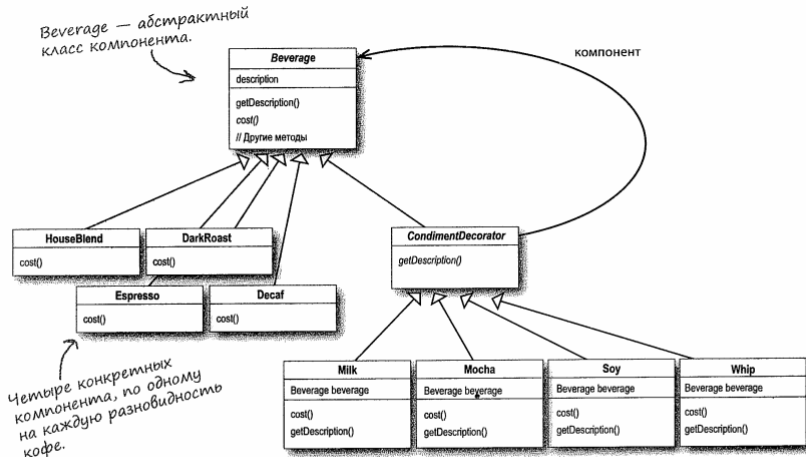
*Декораторы реализуют тот же интерфейс или абстрактный класс, что и декорируемый компонент.*

*ConcreteDecorator содержит переменную экземпляра, в которой хранится декорируемый объект (Component).*

*Декораторы могут расширять состояние компонента.*

*Декораторы могут добавлять новые методы, однако новое поведение обычно добавляется до или после вызова существующего метода компонента.*

Давайте преобразуем иерархию напитков Starbuzz к этой структуре...



Декораторы представляют собой дополнения к кофе. Обратите внимание: они должны реализовать не только `cost()`, но и `getDescription()`. Вскоре мы увидим, почему это необходимо...

## Пишем *kog* для Starbuzz

Пора воплотить наши замыслы в реальном коде.



Начнем с класса `Beverage`, который достался нам из исходной архитектуры Starbuzz. Он выглядит так:

```
public abstract class Beverage {  
    String description = "Unknown Beverage";  
  
    public String getDescription() {  
        return description;  
    }  
  
    public abstract double cost();  
}
```

*Beverage — абстрактный класс с двумя методами: `getDescription()` и `cost()`.*

*Метод `getDescription` уже реализован, а метод `cost()` необходимо реализовать в subclasses.*

Как видите, класс `Beverage` достаточно прост. Давайте реализуем абстрактный класс для дополнений:

```
public abstract class CondimentDecorator extends Beverage {  
    public abstract String getDescription();  
}
```

Объекты должны быть взаимозаменяемы с `Beverage`, поэтому расширяем класс `Beverage`.

Также все декораторы должны заново реализовать метод `getDescription()`. Зачем? Скоро узнаете...

Разобравшись с базовыми классами, переходим к реализации некоторых напитков. Начнем с эспрессо. Как говорилось ранее, мы должны задать описание конкретного напитка в методе `getDescription()` и реализовать метод `cost()`.

```
public class Espresso extends Beverage {  
  
    public Espresso() {  
        description = "Espresso";  
    }  
  
    public double cost() {  
        return 1.99;  
    }  
}
```

Все классы конкретных напитков расширяют `Beverage`.

Описание задается в конструкторе класса. Стоит напомнить, что переменная `description` наследуется от `Beverage`.

Остается вычислить стоимость напитка. В этом классе беспокоиться о дополнениях не нужно, поэтому мы просто возвращаем стоимость «базового» эспрессо: \$1.99.

```

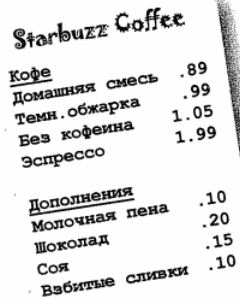
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "House Blend Coffee";
    }

    public double cost() {
        return .89;
    }
}

```

↑ Другой класс напитка. От нас требуется лишь назначить подходящее описание и вернуть правильную стоимость.

Два других класса напитков (DarkRoast и Decaf) создаются аналогично.



A photograph of a white menu card for Starbuzz Coffee. The card is tilted slightly to the right. It has a black header with the text 'Starbuzz Coffee'. Below the header, there are two sections. The first section is titled 'Кофе' (Coffee) and lists four items with their prices: 'Домашняя смесь' (House Blend) for .89, 'Темн. обжарка' (Dark Roast) for .99, 'Без кофеина' (Decaf) for 1.05, and 'Эспрессо' (Espresso) for 1.99. The second section is titled 'Дополнения' (Add-ons) and lists three items with their prices: 'Молочная пена' (Milk foam) for .10, 'Шоколад' (Chocolate) for .15, and 'Взбитые сливки' (Whipped cream) for .10.

Starbuzz Coffee	
<u>Кофе</u>	
Домашняя смесь	.89
Темн. обжарка	.99
Без кофеина	1.05
Эспрессо	1.99
<u>Дополнения</u>	
Молочная пена	.10
Шоколад	.15
Соя	.15
Взбитые сливки	.10

## Программирование дополнений

Взглянув на диаграмму классов паттерна Декоратор, вы увидите, что мы написали абстрактный компонент (Beverage), конкретные компоненты (HouseBlend) и абстрактный декоратор (CondimentDecorator). Пришло время реализации конкретных декораторов. Код декоратора Mocha:

Класс декоратора расширяет CondimentDecorator.

Не забудьте, что CondimentDecorator расширяет Beverage.

Чтобы в объекте Mocha хранилась ссылка на Beverage, нам понадобятся:

```

public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    public double cost() {
        return .20 + beverage.cost();
    }
}

```

(1) Переменная для хранения ссылки.

(2) Способ присваивания переменной ссылки на объект. Мы будем передавать ссылку при вызове конструктора.

В описании должны содержаться не только название напитка (допустим, «Dark Roast»), но и все дополнения — например, «Dark Roast, Mocha». Таким образом, мы сначала получаем описание, делегируя вызов декорируемому объекту, а затем присоединяем к нему строку «, Mocha».

Теперь необходимо вычислить стоимость напитка с шоколадом. Сначала вызов делегируется декорируемому объекту, а затем стоимость шоколада прибавляется к результату.



## Готовим кофе

Поздравляем. Можно устроиться поудобнее, заказать кофе и полюбоваться гибкой архитектурой, построенной на основе паттерна Декоратор.

### Тестовый код для оформления заказов:

```
public class StarbuzzCoffee {

    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
            + " $" + beverage.cost());

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription()
            + " $" + beverage2.cost());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription()
            + " $" + beverage3.cost());
    }
}
```

Заказываем эспрессо без дополнений, выводим описание и стоимость.

Создаем объект DarkRoast.

«Заворачиваем» в объект Mocha...

...Потом во второй...

... И еще в объект Whip.

Напоследок заказываем «домашнюю смесь» с соей, шоколадом и взбитыми сливками.