Professor Hans Noodles

41 уровень

12.11.2019 40131 16 + 1

Паттерны проектирования: FactoryMethod

Привет! Сегодня мы продолжим изучать паттерны проектирования и поговорим о фабричном методе (FactoryMethod).

Ты узнаешь, что это такое и для решения каких задач подходит данный шаблон. Мы рассмотрим этот паттерн проектирования на практике и изучим его структуру.

Чтобы все изложенное было тебе понятно, необходимо разбираться в следующих темах:

1. Наследование в Java.
2. Абстрактные методы и классы в Java.

Какую проблему решает фабричный метод

Во всех фабричных паттернах проектирования есть две группы участников — создатели (сами фабрики) и продукты (объекты, создаваемые фабриками).

Представь ситуацию: у нас есть фабрика, выпускающая автомобили под маркой AutoRush. Она умеет создавать модели автомобилей с различными видами кузовов:

- седаны
- универсалы
- купе

Как здравомыслящие управленцы, мы не хотим терять клиентов OneAuto, и перед нами стоит задача реструктурировать производство таким образом, чтобы мы могли выпускать:

- седаны AutoRush
- универсалы AutoRush
- купе AutoRush
- седаны OneAuto
- универсалы OneAuto
- купе OneAuto

Как видишь, вместо одной группы производных продуктов появилось две, которые различаются некоторыми деталями.

Шаблон проектирования **фабричный метод** решает проблему создания различных групп продуктов, каждая из которых обладает некоторой спецификой.

Принцип данного шаблона мы рассмотрим на практике, постепенно переходя от простого к сложному, на примере нашей кофейни, которую мы создали в одной из предыдущих лекций.

Немного о шаблоне фабрика

Напомню: мы построили с тобой небольшую виртуальную кофейню. В ней мы с помощью простой фабрики научились создавать различные виды кофе. Сегодня будем дорабатывать данный пример.

Давай вспомним, как выглядела наша кофейня с простой фабрикой.

У нас был класс кофе:

```
public class Coffee {
    public void grindCoffee(){
        // перемалываем кофе
    }
    public void makeCoffee(){
        // делаем кофе
    }
    public void pourIntoCup(){
        // наливаем в чашку
    }
}
```

А также несколько его наследников — конкретные виды кофе, которые могла производить наша фабрика:

```
public class Americano extends Coffee {}
public class Cappuccino extends Coffee {}
public class CaffeLatte extends Coffee {}
public class Espresso extends Coffee {}
```

Для удобства принятия заказов мы завели перечисления:

```
public enum CoffeeType {
    ESPRESSO,
    AMERICANO,
    CAFFE_LATTE,
    CAPPUCCINO
}
```

Сама фабрика по производству кофе выглядела следующим образом:

```
public class SimpleCoffeeFactory {
    public Coffee createCoffee (CoffeeType type) {
        Coffee coffee = null;

        switch (type) {
            case AMERICANO:
                coffee = new Americano();
                break;
            case ESPRESSO:
                coffee = new Espresso();
                break;
            case CAPPUCCINO:
                coffee = new Cappuccino();
                break;
            case CAFFE_LATTE:
                coffee = new CaffeLatte();
                break;
        }

        return coffee;
    }
}
```

Ну и, наконец, сама кофейня:

```
public class CoffeeShop {

    private final SimpleCoffeeFactory coffeeFactory;

    public CoffeeShop(SimpleCoffeeFactory coffeeFactory) {
        this.coffeeFactory = coffeeFactory;
    }

    public Coffee orderCoffee(CoffeeType type) {
        Coffee coffee = coffeeFactory.createCoffee(type);
        coffee.grindCoffee();
        coffee.makeCoffee();
        coffee.pourIntoCup();

        System.out.println("Вот ваш кофе! Спасибо, приходите еще!");
        return coffee;
    }
}
```

Модернизация простой фабрики

Наша кофейня работает хорошо. Настолько, что мы подумываем о расширении.

Мы хотим открыть несколько новых точек. Как предприимчивые ребята, мы не будем штамповать однообразные кофейни. Хочется, чтобы у каждой была изюминка.

Изменения затронут не только интерьер, но и напитки:

- в итальянской кофейне мы будем использовать исключительно итальянские кофейные бренды, с особым помолом и прожаркой.
- в американской порции будут чуточку больше, и к каждому заказу будем подавать плавленный зефир — маршмеллоу.

Единственное что останется неизменным — это наша бизнес-модель, которая хорошо зарекомендовала себя.

Если говорить на языке кода, то вот что получается. У нас было 4 класса продуктов:

```
public class Americano extends Coffee {}
public class Cappuccino extends Coffee {}
public class CaffeLatte extends Coffee {}
public class Espresso extends Coffee {}
```

А станет 8:

```
public class ItalianStyleAmericano extends Coffee {}
public class ItalianStyleCappucino extends Coffee {}
public class ItalianStyleCaffeLatte extends Coffee {}
public class ItalianStyleEspresso extends Coffee {}

public class AmericanStyleAmericano extends Coffee {}
public class AmericanStyleCappucino extends Coffee {}
public class AmericanStyleCaffeLatte extends Coffee {}
public class AmericanStyleEspresso extends Coffee {}
```

Раз мы желаем сохранить действующую бизнес-модель неизменной, нам хочется, чтобы метод `orderCoffee(CoffeeType type)` претерпел минимальное количество изменений.

Взглянем на него:

```
public Coffee orderCoffee(CoffeeType type) {
    Coffee coffee = coffeeFactory.createCoffee(type);
    coffee.grindCoffee();
    coffee.makeCoffee();
    coffee.pourIntoCup();

    System.out.println("Вот ваш кофе! Спасибо, приходите еще!");
    return coffee;
}
```

Какие варианты у нас есть? Мы ведь уже умеем писать фабрику? Самое простое, что сходу приходит в голову — написать две аналогичные фабрики, а затем передавать нужную реализацию в нашу кофейню в конструкторе. Тогда класс кофейни не изменится.

Для начала нам нужно создать новый класс-фабрику, унаследоваться от нашей простой фабрики и переопределить метод `createCoffee (CoffeeType type)`. Напишем фабрики для создания кофе в итальянском и американском стилях:

```
public class SimpleItalianCoffeeFactory extends SimpleCoffeeFactory {

    @Override
```

```
        Coffee coffee = null;
        switch (type) {
            case AMERICANO:
                coffee = new ItalianStyleAmericano();
                break;
            case ESPRESSO:
                coffee = new ItalianStyleEspresso();
                break;
            case CAPPUCCINO:
                coffee = new ItalianStyleCappuccino();
                break;
            case CAFFE_LATTE:
                coffee = new ItalianStyleCaffeLatte();
                break;
        }
        return coffee;
    }
}

public class SimpleAmericanCoffeeFactory extends SimpleCoffeeFactory{

    @Override
    public Coffee createCoffee (CoffeeType type) {
        Coffee coffee = null;

        switch (type) {
            case AMERICANO:
                coffee = new AmericanStyleAmericano();
                break;
            case ESPRESSO:
                coffee = new AmericanStyleEspresso();
                break;
            case CAPPUCCINO:
                coffee = new AmericanStyleCappuccino();
                break;
            case CAFFE_LATTE:
                coffee = new AmericanStyleCaffeLatte();
                break;
        }

        return coffee;
    }

}
```

Теперь мы можем передавать нужную реализацию фабрики в CoffeeShop. Давай посмотрим, как бы выглядел код для заказа кофе из разных кофеен. Например, капучино в итальянском и американском стилях:

```
public class Main {
    public static void main(String[] args) {
        /*
            Закажем капучино в итальянском стиле:
            1. Создадим фабрику для приготовления итальянского кофе
```

```
        */
        SimpleItalianCoffeeFactory italianCoffeeFactory = new SimpleItalianCoffeeFactory();
        CoffeeShop italianCoffeeShop = new CoffeeShop(italianCoffeeFactory);
        italianCoffeeShop.orderCoffee(CoffeeType.CAPPUCCINO);


        /*
           Закажем капучино в американском стиле
           1. Создадим фабрику для приготовления американского кофе
           2. Создадим новую кофейню, передав ей в конструкторе фабрику американского кофе
           3. Закажем наш кофе
        */
        SimpleAmericanCoffeeFactory americanCoffeeFactory = new SimpleAmericanCoffeeFactory();
        CoffeeShop americanCoffeeShop = new CoffeeShop(americanCoffeeFactory);
        americanCoffeeShop.orderCoffee(CoffeeType.CAPPUCCINO);
    }
}
```

Мы создали две различные кофейни, передав в каждую нужную фабрику. С одной стороны, мы достигли поставленной задачи, но с другой стороны... Что-то скребет неуемную душу предпринимателя… Давай разбираться, что не так.

Во-первых, обилие фабрик. Это что, каждый раз теперь под новую точку свою фабрику создавать и вдобавок следить за тем, чтобы при создании кофейни в конструктор передавалась нужная фабрика?

Во-вторых, это все еще простая фабрика. Просто немного модернизированная. Мы тут все-таки новый паттерн изучаем.

В-третьих, а что, нельзя что ли по-другому? Вот было бы классно, если бы мы могли локализовать все вопросы по приготовлению кофе внутри класса `CoffeeShop`, связав процессы по созданию кофе и обслуживанию заказа, но при этом сохранив достаточную гибкость, чтобы делать кофе в различных стилях.

Ответ — да, можно. Это называется шаблон проектирования фабричный метод.

От простой фабрики к фабричному методу

Чтобы решить поставленную задачу максимально эффективно, мы:

1. Вернем метод `createCoffee(CoffeeType type)` в класс `CoffeeShop`.
2. Данный метод сделаем абстрактным.
3. Сам класс `CoffeeShop` станет абстрактным.
4. У класса `CoffeeShop` появятся наследники.

Да, друг. Итальянская кофейня, это ничто иное, как наследник класса `CoffeeShop`, реализующий метод `createCoffee(CoffeeType type)` в соответствии с лучшими традициями итальянских бариста.

Итак, по порядку.

Шаг 1. Сделаем класс `Coffee` абстрактным. У нас появилось целых два семейства различных продуктов. У итальянских и американских кофейных напитков по-прежнему есть общий предок — класс `Coffee`. Было бы правильно сделать его абстрактным:

```
public abstract class Coffee {
    public void makeCoffee(){
        // делаем кофе
    }
```

```
    }
}
```

Шаг 2. Делаем `CoffeeShop` абстрактным, с абстрактным методом `createCoffee(CoffeeType type)`

```
public abstract class CoffeeShop {

    public Coffee orderCoffee(CoffeeType type) {
        Coffee coffee = createCoffee(type);

        coffee.makeCoffee();
        coffee.pourIntoCup();

        System.out.println("Вот ваш кофе! Спасибо, приходите еще!");
        return coffee;
    }

    protected abstract Coffee createCoffee(CoffeeType type);
}
```

Шаг 3. Создадим итальянскую кофейню, класс-потомок абстрактной кофейни. В нем мы реализуем метод `createCoffee(CoffeeType type)` с учетом итальянской специфики.

```
public class ItalianCoffeeShop extends CoffeeShop {

    @Override
    public Coffee createCoffee (CoffeeType type) {
        Coffee coffee = null;
        switch (type) {
            case AMERICANO:
                coffee = new ItalianStyleAmericano();
                break;
            case ESPRESSO:
                coffee = new ItalianStyleEspresso();
                break;
            case CAPPUCCINO:
                coffee = new ItalianStyleCappuccino();
                break;
            case CAFFE_LATTE:
                coffee = new ItalianStyleCaffeLatte();
                break;
        }
        return coffee;
    }
}
```

Шаг 4. Проделаем тоже самое, для кофейни в американском стиле

```
public class AmericanCoffeeShop extends CoffeeShop {
    @Override
    public Coffee createCoffee (CoffeeType type) {
        Coffee coffee = null;
```

```
            case AMERICANO:
                coffee = new AmericanStyleAmericano();
                break;
            case ESPRESSO:
                coffee = new AmericanStyleEspresso();
                break;
            case CAPPUCCINO:
                coffee = new AmericanStyleCappuccino();
                break;
            case CAFFE_LATTE:
                coffee = new AmericanStyleCaffeLatte();
                break;
        }

        return coffee;
    }
}
```

Шаг 5. Взглянем на то, как будет выглядеть заказ латте в американском и итальянском стиле:

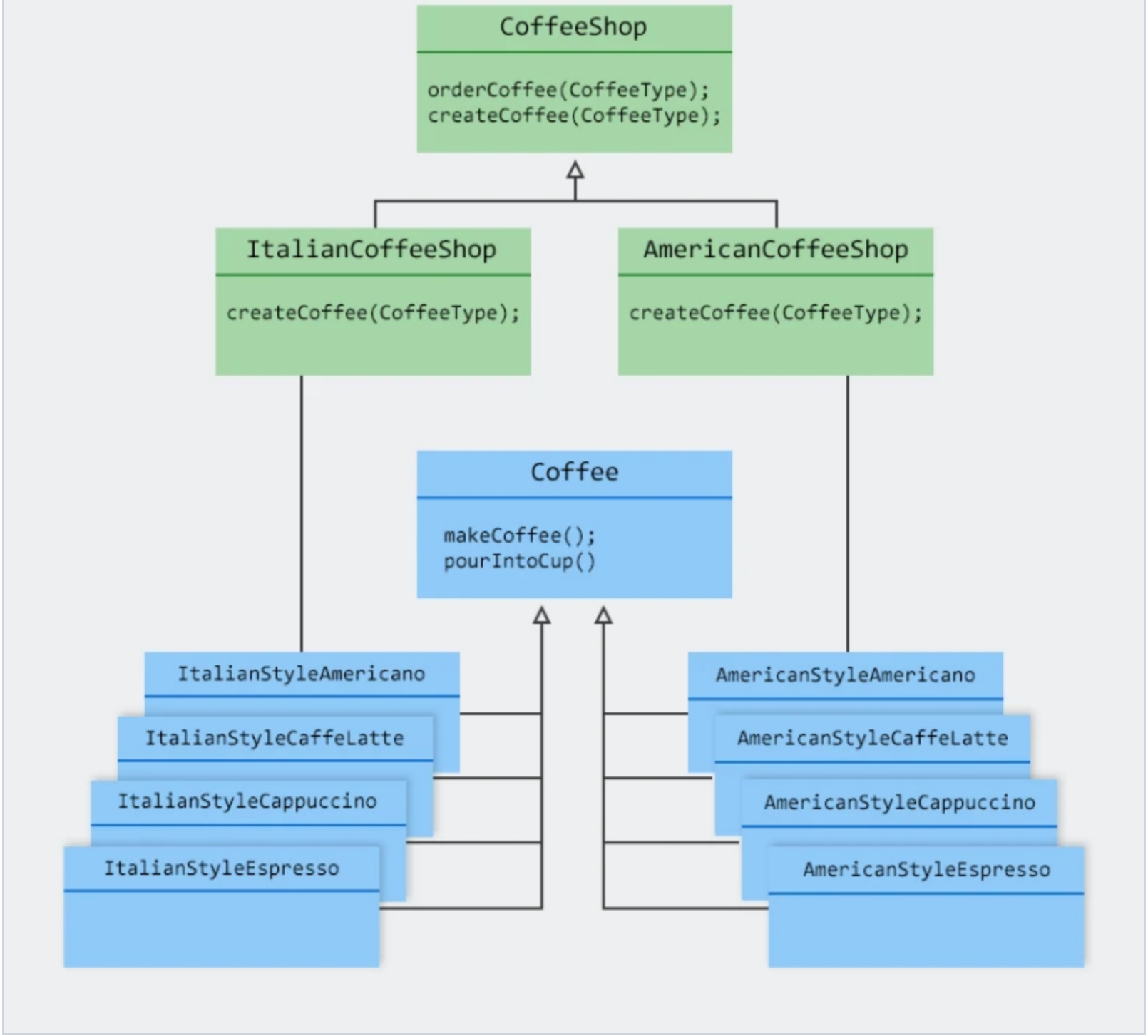
```
public class Main {
    public static void main(String[] args) {
        CoffeeShop italianCoffeeShop = new ItalianCoffeeShop();
        italianCoffeeShop.orderCoffee(CoffeeType.CAFFE_LATTE);

        CoffeeShop americanCoffeeShop = new AmericanCoffeeShop();
        americanCoffeeShop.orderCoffee(CoffeeType.CAFFE_LATTE);
    }
}
```

Поздравляю тебя. Мы только что реализовали шаблон проектирования фабричный метод на примере нашей кофейни.

Принцип работы фабричного метода

Теперь рассмотрим подробнее, что же у нас получилось.

На диаграмме ниже — получившиеся классы. Зеленые блоки — классы создатели, голубые — классы продукты.



Какие выводы можно сделать?

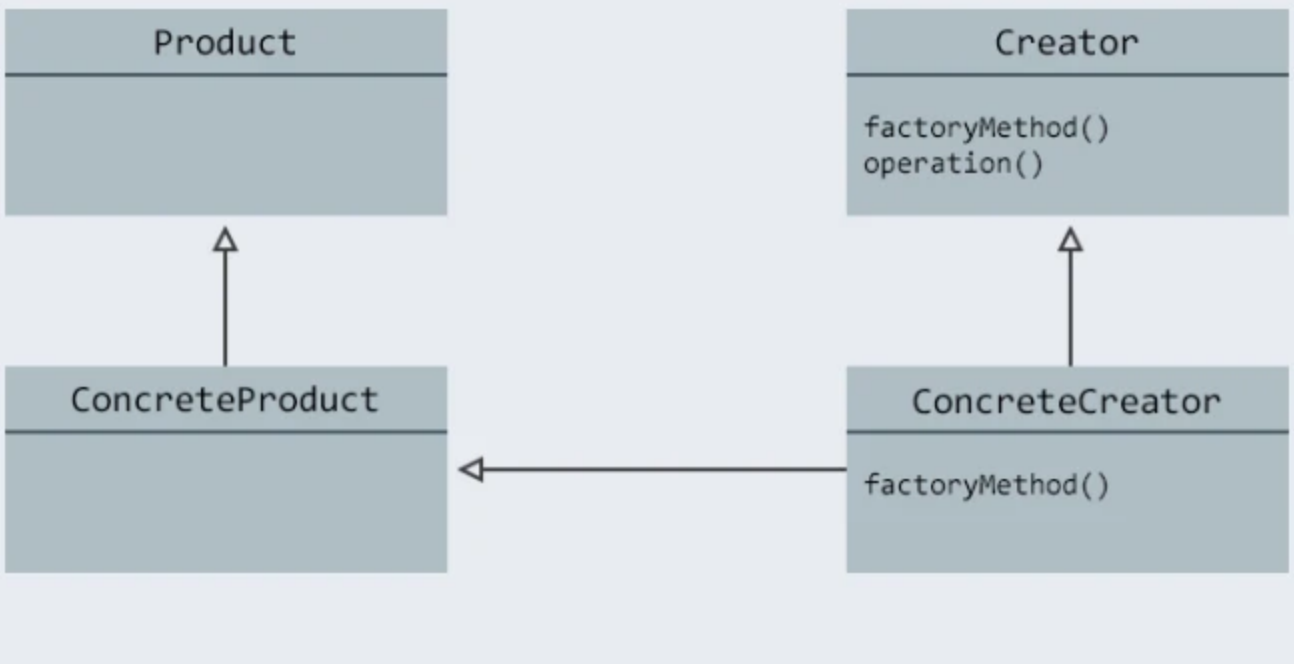
1. Все продукты — реализации абстрактного класса `Coffee`.
2. Все создатели — реализации абстрактного класса `CoffeeShop`.
3. Мы наблюдаем две параллельные иерархии классов:
    a. Иерархия продуктов. Мы видим итальянских потомков и американских потомков
    b. Иерархия создателей. Мы видим итальянских потомков и американских потомков
4. У суперкласса `CoffeeShop` нет информации о том, какая конкретно реализация продукта (`Coffee`) будет создана.
5. Суперкласс `CoffeeShop` делегирует создание конкретного продукта своим потомкам.
6. Каждый потомок класса `CoffeeShop` реализует фабричный метод `createCoffee()` в соответствии со своей спецификой. Иными словами, внутри реализаций классов-создателей принимается решение о приготовлении конкретного продукта, исходя из специфики класса создателя.

Теперь ты готов к определению паттерна **фабричный метод**.

Паттерн фабричный метод определяет интерфейс создания объекта, но позволяет субклассам выбрать класс создаваемого экземпляра. Таким образом, Фабричный метод делегирует операцию создания экземпляра субклассам.

В общем, не столь важно помнить определение, как понимать, как все работает.

Структура фабричного метода



На схеме выше представлена общая структура паттерна фабричный метод.

Что еще здесь важно?

1. Класс Creator содержит реализации всех методов, взаимодействующих с продуктами, кроме фабричного метода.
2. Абстрактный метод `factoryMethod()` должен быть реализован всеми потомками класса `Creator`.
3. Класс `ConcreteCreator` реализует метод `factoryMethod()`, непосредственно производящий продукт.
4. Данный класс отвечает за создание конкретных продуктов. Это единственный класс с информацией о создании этих продуктов.

+97

Комментарии (16) + 1 — **популярные** **новые** **старые**

JavaCoder

Введите текст комментария

Сергей Java Developer в **Адвантум** — 17 августа 2021, 12:01

♂

Чем в конечном итоге SimpleItalianCoffeeFactory и SimpleAmericanCoffeeFactory отличаются от ItalianCoffeeShop и AmericanCoffeeShop?

"*Во-первых, обилие фабрик. Это что, каждый раз теперь под новую точку свою фабрику создавать и вдобавок следить за тем, чтобы при создании кофейни в конструктор передавалась нужная фабрика?* "

Хорошо! Внезапно я захотел открыть индусскую кофейню. Но вот проблема: мне придется создать очередной CoffeShop. В чем преимущество?

"*Во-вторых, это все еще простая фабрика. Просто немного модернизированная. Мы тут все-таки новый паттерн изучаем.*

В-третьих, а что, нельзя что ли по-другому? Вот было бы классно, если бы мы могли локализовать все вопросы по приготовлению кофе внутри класса CoffeeShop, связав процессы по созданию кофе и обслуживанию заказа, но при этом сохранив достаточную гибкость, чтобы делать кофе в различных стилях. "

Где гибкость? Все то же самое, с немного изменённой структурой? Что так - что так придется одинаковое количество кода писать/копипастить.

P.S. Ни фабрика ни фабричный метод не подходят, когда предполагается, что количество реализаций однотипных объектов будет увеличиваться. Автор статьи выбрал очень неудачный пример для данного паттерна.

Ответить

+1

Сергей Java Developer в **Адвантум** — 17 августа 2021, 16:12

Всем кто не разобрался и хочет понять суть и разницу рекомендую почитать:

Ответить +2

LuneFox инженер по сопровождению в **BIFIT** EXPERT 4 марта, 19:32

Если я правильно понял автора, то в случае с разными фабриками для приготовления кофе нужно сначала создать фабрику, на основе её создать магазин, а уже в магазине создавать кофе. Фабричный метод позволяет не создавать фабрику, а делать нужный кофе прямо в магазине.

Ответить 0

Anonymous #2997315 Уровень 4, Тула, Russian Federation 27 мая, 17:36

Не работают ссылки, к сожалению

Ответить +1

Валера Калиновский Java Developer 19 августа, 21:05

потому что заблокирован этот сайт в рф. юзай впн

Ответить 0

Maks Panteleev Java Developer в **Bell Integrator** 26 июля 2021, 14:18

То ли пример гавно, то ли паттерн, самый просто вариант - добавить в енам еще 4 вида кофе и 4 кейс блока и не городить гавна. Если хотим сохранить абстракцию - то достаточно было создать два енама - итальянский и американский, абстрактный класс, две его реализации и внизу уже ниче не трогать

Ответить +2

Игорь Full Stack Developer в **IgorApplications** 9 августа 2021, 17:57

Код стал лучше выглядить и понятнее с помощью фабричного метода. Хочу отметить, что сначало в лекции показали, как сделать субфабрики, а потом только фабричный метод, а всё что связанно с фабриками было удаленно в конце.

Ответить 0

Сергей Java Developer в **Адвантум** 17 августа 2021, 13:19

Пример крайне неудачно выбран. По сути автор статьи поменял шило на мыло и пишет еще про какую-то мнимую гибкость. Про Enum полностью поддерживаю.

Ответить +1

On Girame Уровень 20, Москва 9 июня 2021, 07:41

Если у нас появляется американская кофейня, итальянская кофейня, то нам уже нужна "Абстрактная фабрика"

Ответить 0

Valua Sinicyn Уровень 41, Харьков, Украина 24 февраля 2021, 09:37

В общем, как я понял, ФМ - расширенная реализация Фабрики.

Ответить 0

Anonymous #2297535 Уровень 22, Северодвинск, Россия 18 февраля 2021, 23:07

Вот, отдохни чутка.
https://www.youtube.com/watch?v=t-Yf-c4FoXg

Ответить +3

Soros Уровень 39, Харьков, Украина 24 апреля 2020, 11:08

Если говорить на языке кода, то вот что получается. У нас было 4 класса продуктов:

```
1  public class Americano extends Coffee {}
2  public class Cappuccino extends Coffee {}
3  public class CaffeLatte extends Coffee {}
4  public class Espresso extends Coffee {}
```

А станет 8:

```
1  public class ItalianStyleAmericano extends Coffee {}
2  public class ItalianStyleCappucino extends Coffee {}
3  public class ItalianStyleCaffeLatte extends Coffee {}
4  public class ItalianStyleEspresso extends Coffee {}
5
6  public class AmericanStyleAmericano extends Coffee {}
7  public class AmericanStyleCappucino extends Coffee {}
8  public class AmericanStyleCaffeLatte extends Coffee {}
9  public class AmericanStyleEspresso extends Coffee {}
```

```
1  public class AmericanStyleAmericano extends Americano{}
```

Ответить

+3

hidden #2212451 Уровень 35

18 июня 2020, 11:01

Мне тоже кажется это логичнее

Ответить

0

Soros Уровень 39, Харьков, Украина

24 апреля 2020, 10:07

```
1  /*
2              Закажем капучино в американском стиле
3              1. Создадим фабрику для приготовления американского кофе
4              2. Создадим новую кофейню, передав ей в конструкторе фабрику американс
5              3. Закажем наш кофе
6           */
7          SimpleItalianCoffeeFactory americanCoffeeFactory = new SimpleItalianCoffee
```

необходимо исправить на **SimpleAmericanCoffeeFactory**

Ответить

+4

hidden #2109277 Уровень 22

10 апреля 2020, 15:20

без указания всем классам static не работает ваш код

Ответить

0

Sergey Уровень 1, Алматы

19 мая 2020, 17:52

у меня все работает

Ответить

+1