Статья

Управление

АВТОР

Горковенко Андрей

Фронтенд-разработчик в **NFON AG**

22.02.2023 57861 32

Паттерн проектирования Factory

Статья из группы Java Developer

48675 участников

Вы в группе

Привет, друг! Сегодня мы продолжим изучать с тобой паттерны проектирования. В этой лекции будем говорить о Фабрике. Обсудим с тобой, какую проблему решают с помощью данного шаблона, посмотрим на примере, как фабрика помогает открывать кофейню. А еще я я дам тебе 5 простых шагов для создания фабрики.

Чтобы быть со всеми на одной волне и легко улавливать суть, тебе должны быть знакомы такие темы:

- Наследование в Java
- Сужение и расширение ссылочных типов в Java
- Взаимодействие между различными классами и объектами

Что такое Фабрика?

Шаблон проектирования Фабрика позволяет управлять созданием объектов.

Процесс создания нового объекта не то чтобы прост, но и не слишком сложен. Все мы знаем, что для создания нового объекта необходимо использовать оператор `new`. И может показаться, что здесь нечем управлять, однако это не так.

Сложности могут возникнуть, когда в нашем приложении есть некоторый класс, у которого есть множество наследников, и необходимо создавать экземпляр определенного класса в зависимости от некоторых условий.

Фабрика — это шаблон проектирования, который помогает решить проблему создания

различных объектов в зависимости от некоторых условий.

Абстрактно, не правда ли? Больше конкретики и ясности появится, когда мы рассмотрим пример ниже.

Создаем различные виды кофе

Предположим, мы хотим автоматизировать кофейню. Нам необходимо научиться готовить различные виды кофе. Для этого в нашем приложении мы создадим класс кофе и его производные: американо, капучино, эспрессо, латте — те виды кофе, которые мы будем готовить.

Начнем с общего кофейного класса:

```
1  public class Coffee {
2      public void grindCoffee(){
3          // перемалываем кофе
4      }
5      public void makeCoffee(){
6          // делаем кофе
7      }
8      public void pourIntoCup(){
9          // наливаем в чашку
10     }
11 }
```

Далее создадим его наследников:

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

Наши клиенты будут заказывать какой-либо вид кофе, и эту информацию нужно передавать программе. Это можно сделать разными способами, например использовать `String`. Но лучше всего для этих целей подойдет `enum`. Создадим `enum` и определим в

нем типы кофе, на которые мы принимаем заказы:

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

Отлично, теперь напишем код нашей кофейни:

```
public class CoffeeShop {

    public Coffee orderCoffee(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;
        }

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

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

```
27        }
28    }
```

Метод `orderCoffee` можно разделить на две составляющие:

1. Создание конкретного экземпляра кофе в блоке `switch-case`. Именно здесь происходит то, что делает Фабрика — создание конкретного типа в зависимости от условий.
2. Само приготовление — перемолка, приготовление и разлитие в чашку.

Что важно знать, если нужно будет вносить в метод изменения в будущем:

1. Сам алгоритм приготовления (перемолка, приготовление и разлитие в чашку) останется неизменным (по крайней мере мы на это рассчитываем).
2. А вот ассортимент кофе может измениться. Возможно, мы начнем готовить мока.. Мокка.. Моккачи… Господь с ним, новый вид кофе.

Мы уже сейчас можем предположить, что в будущем, с определенной долей вероятности, нам придется вносить изменения в метод, в блок `switch-case`.

Также возможно, в нашей кофейне метод `orderCoffee` будет не единственным местом, в котором мы будем создавать различные виды кофе. Следовательно, вносить изменения придется в нескольких местах.

Тебе уже наверняка понятно, к чему я клоню. Нам нужно рефакторить. Вынести блок, отвечающий за создание кофе, в отдельный класс по двум причинам:

1. Мы сможем переиспользовать логику создания кофе в других местах.
2. Если ассортимент изменится, нам не придется править код везде, где будет использоваться создание кофе. Достаточно будет изменить код только в одном месте.

Иными словами, пришло время запилить фабрику.

Пилим нашу первую фабрику

Для этого создадим новый класс, который будет отвечать только за создание нужных экземпляров классов кофе:

```
1  public class SimpleCoffeeFactory {
2      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 Cappucino();
                break;
            case CAFFE_LATTE:
                coffee = new CaffeLatte();
                break;
        }

        return coffee;
    }
}
```

Поздравляю тебя! Мы только что реализовали шаблон проектирования Фабрика в самом его простейшем виде.

Хотя все могло быть еще проще, если сделать метод `createCoffee` статичным. Но тогда мы потеряли бы две возможности:

1. Наследоваться от `SimpleCoffeeFactory` и переопределять метод `createCoffee`.
2. Внедрять нужную реализацию фабрики в наши классы.

Кстати о внедрении. Нам нужно вернуться в кофейню и внедрить нашу фабрику по созданию кофе.

Внедрение фабрики в кофейню

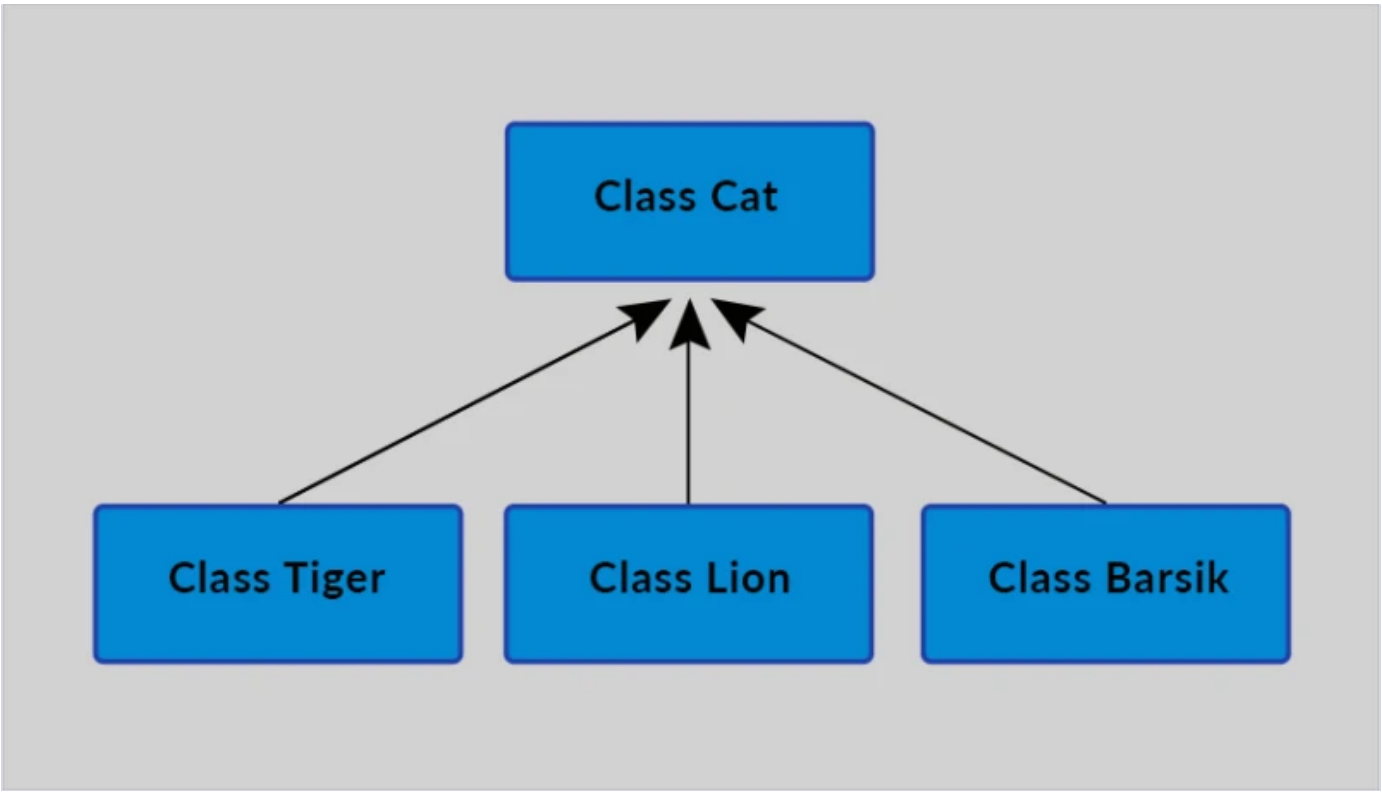
Перепишем класс нашей кофейни с использованием фабрики:

```
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;
    }
}
```

Отлично. Теперь схематично и лаконично попробуем описать структуру шаблона проектирования Фабрика.

5 шагов к открытию собственной фабрики

Шаг 1. У тебя в программе класс с несколькими потомками, как на картинке ниже:



Шаг 2. Ты создаешь `enum`, в котором определяешь enum-переменную для каждого класса-наследника:

```
enum CatType {
    LION,
    TIGER,
    BARSIK
}
```

Шаг 3. Ты строишь свою фабрику. Называешь её `MyClassFactory`, код ниже:

```
class CatFactory {}
```

Шаг 4. Ты создаешь в своей фабрике метод `createMyClass`, который принимает в себя переменную- `enum` `MyClassType`. Код ниже:

```
class CatFactory {
    public Cat createCat(CatType type) {


```

```
    }
}
```

Шаг 5. Ты пишешь в теле метода блок `switch-case`, в котором перебираешь все enum значения и создаешь экземпляр класса, соответствующий `enum` значению:

```
class CatFactory {
        public Cat createCat(CatType type) {
            Cat cat = null;

            switch (type) {
                case LION:
                    cat =  new Barsik();
                    break;
                case TIGER:
                    cat = new Tiger();
                    break;
                case BARSIK:
                    cat =  new Lion();
                    break;
            }

            return cat;
        }
    }
```

Like a boss.

Как тренироваться

Читать — хорошо, писать код —еще лучше. Если в твоем имени четное количество букв, попробуй создать свою виртуальную пиццерию.

Если в твоем имени нечетное количество букв, попробуй создать виртуальный суши-бар.

Если ты безымянный — тебе повезло. Сегодня можешь отдыхать.

Горковенко Андрей

Фронтенд-разработчик в NFON AG

В прошлом Андрей руководил собственной web-студией в Киеве и работал фронтенд-разработчиком в JavaRush, а сейчас пишет код для нем ...
[Читать полную биографию]

Я понимаю что в 1м примере с кофе нужно разделить создание экземпляров от приготовления кофе. Мне не понятно, а не проще ли просто было создать метод createCoffee в этом же классе CoffeeShop, а не городить дополнительный класс? Я не настаиваю что я прав, но хочу понять что бы разобраться.

```
public class CoffeeShop {

    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;
    }

    public Coffee orderCoffee(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;
        }*/
        Coffee coffee = createCoffee(type);

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

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

Ответить

0

Сирожиддин Уровень 33

18 января, 17:44

SRP! Кофейня не должна заниматься созданием кофе

Ответить 0

Anonymous #3090867 Уровень 1 2 августа 2022, 15:05

Вы на приколе case BARSIK:
cat = new Lion();
break;
}

Ответить 0

Art09 Уровень 35 25 мая 2022, 07:32

У данного примера с кофе, мы все также не решаем проблему маштабирования. Если у нас появляется новый вид кофе, то нам нужно вносить изменения в 2 класса SimpleCoffeeFactory и CoffeeType. "Если ассортимент изменится, нам не придется править код везде, где будет использоваться создание кофе. Достаточно будет изменить код только в одном месте." - остается проблемой!!!

Ответить 0

Сирожиддин Уровень 33 18 января, 17:47

Ну можете же enum запихнуть в класс фабрики. Тут скорее не про минимизацию изменений, а про контролируемые изменения

Ответить 0

Руслан Уровень 22 23 мая 2022, 08:04

А как это дело запустить то? В мэйне мы же не можем создать экземпляр кофешоп... Статик сделать тоже не получается.
Как вывести то в консоль что кофе готово?

Ответить 0

Art09 Уровень 35 25 мая 2022, 07:25

Создаем Магазин в кот. мы продаем кофе, также есть у нас фабрика по созданию разного кофе. Далее помещаем в ссылку созданный кофе и отдаем клиенту:
CoffeeShop coffeeShop = new CoffeeShop(new SimpleCoffeeFactory());
Coffee coffee = coffeeShop.orderCoffee(CoffeeType.AMERICANO);

Ответить 0

Ян Уровень 41 8 января 2022, 19:28

```
1 Хотя все могло быть еще проще, если сделать метод createCoffee статичным.
2 Но тогда мы потеряли бы две возможности:
3 1. Наследоваться от SimpleCoffeeFactory и переопределять метод createCoff
4 .......
```

Почему не будет возможности наследоваться и тем более переопределять метод, если createCoffe будет с модификатором static?

Ответить 0

Максим Караваев Java Developer в **Deutsche Telekom IT** 14 января 2022, 13:00

Потому что статические методы не переопределяются. Т.к. это метод класса, а не экземпляра.
Можно создать в наследнике статический метод с такой же сигнатурой, но это будет не переопределённый метод родителя, а просто метод другого класса с такой же сигнатурой.

Ответить +1

Ян Уровень 41 14 января 2022, 19:21

благодарю, стало понятнее. раньше думал, что переопределять методы нельзя только с модификатором final...

Ответить 0

alex Уровень 41 2 июня 2021, 11:21

Использовать switch тоже не совсем хорошая практика. Это претензия не к автору, почему то во всех примерах используют case. Но это как-то не красиво, да и в реальных проектах я такого не встречал

Ответить +1

Maks Panteleev Java Developer в **Bell Integrator** 26 июля 2021, 14:08

а что используют? иф?

Ответить +2

Денис Гарбуз Уровень 1 4 июня 2022, 14:46

Автор коммента не встречал в проектах свич, потому что он нее работал ни на каких проектах 😂

Ответить +1

Сирожиддин Уровень 33 18 января, 17:48

Ну и аргументы у вас

Ответить 0

Romanya Java Developer в **Продуктовая IT компа** 30 мая 2021, 12:20

Статья класс! Спасибо автору.

Ответить 0

Valua Sinicyn Уровень 41 23 февраля 2021, 20:41

Фабрика на человеческом.

Ответить +6

Azat Уровень 41 15 сентября 2020, 19:11

Интересно, как реализовать фабрику без свитча. Использовать свитч, насколько могу судить, не лучшая практика

Ответить +2

Сирожиддин Уровень 33 18 января, 17:48

Причина?

Ответить 0

Yaroslav Katrushka Уровень 18 7 августа 2020, 15:28

Хорошая статья. Доступно и полезно

Ответить +5

Показать еще комментарии