Design Pattern Fabric

Шаблон проектирования Фабрика позволяет управлять созданием объектов. Все мы знаем, что для создания нового объекта необходимо использовать оператор new. Однако, такое объявление может спровоцировать осложнения, 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.

*(1)*

public class Coffee {

public void grindCoffee(){

}

public void makeCoffee(){

}

public void pourIntoCup(){

}

}

Предположим, мы хотим автоматизировать кофейню. Нам необходимо научиться готовить различные виды кофе

*(2)*

public class Americano extends Coffee {}

public class Cappuccino extends Coffee {} Наследники (ч/з extends)

public class CaffeLatte extends Coffee {}

public class Espresso extends Coffee {}

*(3)*

public enum CoffeeType {

ESPRESSO,

AMERICANO,

CAFFE\_LATTE,

CAPPUCCINO

}

*(4)*

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:

Клиенты будут заказывать какой-либо вид кофе, и это нужно передавать программе. Это можно сделать с использованием String, но лучше всего ч/з enum.

coffee = new Cappucсino();

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;

}

}

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 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.

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.

(5)

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 Cappucino();

break;

case CAFFE\_LATTE:

coffee = new CaffeLatte();

break;

}

return coffee;

}

}

(6)

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;

}

}

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.

Теперь поговорим о ***фабричном методе***. Во всех фабричных паттернах проектирования есть две группы участников — создатели (сами фабрики) и продукты (объекты, создаваемые фабриками). Наглядный пример можно привести, когда, скажем, некоторый концерн по производству автомобилей (условный **Volkswagen AG**) поглощает средних производителей (**[Škoda](https://ru.wikipedia.org/wiki/%C5%A0koda_Auto" \o ")**или [**Audi**](https://ru.wikipedia.org/wiki/Audi)**)*.*** И нам необходимо наладить изготовление тех типов кузовов, двигателей, моторов и т.д., которые раньше делались на поглощённых заводах. Шаблон проектирования *фабричный метод* решает проблему создания различных групп продуктов, каждая из которых обладает некоторой спецификой (т.е. инкапсулирует *изменчивость*).

Но вернёмся к кофейне. Пусть нам необходимо расширить её. Как предприимчивые ребята, мы не будем штамповать однообразные кофейни. Добавим национального колорита что ли. Поэтому для начала откроем две точки: в итальянском и американском стилях. Единственное что останется неизменным — это наша бизнес-модель.

(7)

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 {}

(8)

public class Simple(Italian/American)CoffeeFactory extends SimpleCoffeeFactory {

@Override

public Coffee createCoffee (CoffeeType type) {

Coffee coffee = null;

switch (type) {

case AMERICANO:

coffee = new (Italian/American)StyleAmericano();

break;

case ESPRESSO:

coffee = new (Italian/American)StyleEspresso();

break;

case CAPPUCCINO:

coffee = new (Italian/American)StyleCappuccino();

break;

case CAFFE\_LATTE:

coffee = new (Italian/American)StyleCaffeLatte();

break;

}

return coffee;

}

}

(9)

public class Main {

public static void main(String[] args) {

SimpleItalianCoffeeFactory italianCoffeeFactory = new SimpleItalianCoffeeFactory();

CoffeeShop italianCoffeeShop = new CoffeeShop(italianCoffeeFactory);

italianCoffeeShop.orderCoffee(CoffeeType.CAPPUCCINO);

SimpleAmericanCoffeeFactory americanCoffeeFactory = new SimpleAmericanCoffeeFactory();

CoffeeShop americanCoffeeShop = new CoffeeShop(americanCoffeeFactory);

americanCoffeeShop.orderCoffee(CoffeeType.CAPPUCCINO);

}

}

Раз мы желаем сохранить действующую бизнес-модель неизменной, нам хочется, чтобы метод orderCoffee(CoffeeType type) претерпел минимальное количество изменений (см. (7)).

Какие варианты у нас есть? 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 (см. (8)):

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(см. (9)):

Мы создали две различные кофейни, передав в каждую нужную фабрику. С одной стороны, мы достигли поставленной задачи, но с другой стороны... 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*, 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***.

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.

public abstract class Coffee {

public void makeCoffee(){

}

public void pourIntoCup(){

}

}

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);

}

public class (Italian/American)CoffeeShop extends CoffeeShop {

@Override

public Coffee createCoffee (CoffeeType type) {

Coffee coffee = null;

switch (type) {

case AMERICANO:

coffee = new (Italian/American)StyleAmericano();

break;

case ESPRESSO:

coffee = new (Italian/American)StyleEspresso();

break;

case CAPPUCCINO:

coffee = new (Italian/American)StyleCappuccino();

break;

case CAFFE\_LATTE:

coffee = new (Italian/American)StyleCaffeLatte();

break;

}

return coffee;

}

}

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);

}

}

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

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.
5. Все продукты должны реализовывать общий интерфейс — быть потомками общего класса-продукта. Это нужно, чтобы классы, использующие продукты, могли оперировать ими на уровне абстракций, а не конкретных реализаций.

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:
   1. Иерархия продуктов. Мы видим итальянских потомков и американских потомков
   2. 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.