Design Patterns

الگوهای طراحی

پروژه‌های بزرگ معمولا نیازمند نظم بالایی هستند. دلایل این امر واضح است: عیب‌یابی سریع، قابلیت توسعه‌پذیری بالا و سریع، و خوانا و قابل فهم بودن کد برای مهندسین جدید. اگر بنا باشد در هر پروژه استانداردهای متفاوتی برای برقرار کردن نظم اعمال شود، وفق پیدا کردن برای افراد جدید بسیار سخت و زمان‌بر خواهد بود؛ همچنین شکل‌گیری ارتباط مؤثر بین افراد نیز به دلیل متفاوت بودن این استانداردها دشوار خواهد بود. «دیزاین پترن»ها، یا الگوهای طراحی، اینجا به کمک ما می آیند. این الگوها، پاسخ‌هایی بهینه و آزموده، در برابر مسائلی که در برنامه‌نویسی شئ‌گرا بروز میکنند هستند. به همین دلیل، خیلی از شرکت‌ها به «دیزاین پترن»ها اهمیت زیادی می‌دهند، و حتی برخی دانستن آن‌ها را برای ورود به شرکت الزامی می‌دانند، چرا که نقش بسیار مؤثری در یکپارچه‌سازی و کاهش زمان توسعه ایفا می‌کنند. در ادامه به معرفی دو پترن ساده می‌پردازیم.

سید محمد جعفری
دانشجو مهندسی کامپیوتر
دانشکدگان فارابی دانشگاه تهران
Smjaffry@ut.ac.ir

اول: Factory Method

فرض کنید شما صاحب یک ساندویچی هستید. از ساعت ۱ بعد از ظهر که آغاز به کار می‌کنید، سفارش‌های متعددی برایتان ارسال می‌شود. در حال حاضر شما فقط ساندویچ فلافل و هات‌داگ عرضه می‌کنید. اما از آنجایی که شما خیلی خفن هستید، می‌خواهید در آینده ساندویچ‌های بیشتری در منو قرار بدهید. چون تازه کار هستید، هنوز پول کافی برای استخدام یک برنامه نویس جهت طراحی برنامه مدیریت سفارش ندارید؛ از طرفی شما برنامه‌نویس قدری هستید، پس تصمیم می‌گیرید که این سامانه را خود تهیه کنید. برای اینکه یک کد تمیز و خوشگل بنویسید که نوع ساندویچ سفارشی را بگیرد و یک شئ از آن نوع ساندویچ بسازد چه ایده‌ای دارید؟ آیا می‌توانید هنگام اجرای برنامه اشیای مختلفی بسازید؟ اگر قرار باشد ساندویچ‌های دیگری به منو اضافه شود، چقدر به کارتان اضافه می‌شود؟

مشخصا انجام دادن این کار سخت نیست، می‌توانیم با چند عبارت شرطی که ورودی را رصد میکند اشیای مختلفی بسازیم، اما این کار انعطاف برنامه را کاهش می‌دهد. بر خلاف روش ساده، پترن Factory Method یا روش کارخانه در واقع کار ساختن شئ‌های متعدد را ساده می‌کند. این پترن با اتکا بر مفاهیم ارث‌بری و کلاس‌های ابسترکت، تعیین نوع شئ را بر عهده زیرکلاس (کلاس فرزند) می‌گذارد. برای این مثال، ما یک کلاس ابسترکت با نام Sandwich می سازیم. حالا هر کدام از ساندویچ‌های ما این کلاس ابسترکت را پیاده‌سازی می‌کنند. اثر این کار موقعی بیشتر به چشم می‌آید که ساندویچی شما گسترش پیدا کند، و ساندویچ‌های بیشتری به منو اضافه شود.

روش کار این پترن ساده است. به موارد زیر توجه کنید:

یک کلاس ابسترکت (می‌توان از اینترفیس نیز با شرایط مختلف استفاده کرد) داریم که یک سری اتریبیوت‌ها و متدهای عمومی را دارا است.

```
public abstract class Sandwich {

    public String name;
    public int price;

    public String preparing(){
        return "One " + this.name + " heard!";
    }

    public String ready(){
        return "One " + this.name + " coming up!";
    }
}
```

چند کلاس کانْکرِت (concrete classes/ کلاس‌هایی که نوع‌شان مشخص است) را با اتریبیوت‌ها و متدهای خاص، به همراه سازنده داریم.

```
public class HotDog extends Sandwich {
    public HotDog(){
        this.name = "Hot Dog";
        this.price = 20;
    }
}

public class Falafel extends Sandwich{
    public Falafel(){
        this.name = "Falafel";
        this.price = 10;
    }
}
```

یک کلاس سازنده اشیا (کارخانه) وجود دارد که با توجه به پارامتر ارسال شده، نوع شئ درخواستی را تشخیص داده، و یک شئ جدید با آن نوع می‌سازد.

```
public class SandwichFactory{
    public Sandwich makeSandwich(String type){
        if (type.equals("Hot Dog")){
            return new HotDog();
        } else if (type.equals("Falafel")){
            return new Falafel();
        }
        else{
    System.out.println("Sorry, we don't have that sandwich.");
    return null;
        }
    }
}
```

اکنون ما یک کلاس با نام SandwichFactory داریم که کار آن ساختن انواع مختلف ساندویچ است. می توان تصور کرد که این کلاس نقش آشپز را ایفا می‌کند. ما به او به فارسی سفارش‌مان را میگوییم، مثلا می گوییم:«هات‌داگ»، و او یک هات‌داگ واقعی به ما تحویل میدهد! حال برنامه‌ای که برای اجرا مینویسیم را ببینید:

```
public static void main(String[] args) {
    SandwichFactory sandwichFactory = new SandwichFactory();
    System.out.println("What kind of sandwich do you want?");
    Sandwich sandwich = sandwichFactory.makeSandwich(userInput());
    System.out.println(sandwich.preparing());
    System.out.println(sandwich.ready());
}
```

یکی از اهداف این روش، مشخص کردن نوع شئ در زمان اجرا است. میتوانستیم بصورت if-else در کلاس main هم کد این برنامه را بنویسیم، اما قواعد زیادی را باید زیرپا می‌گذاشتیم. در واقع ما ساخت اشیا را به کلاس SandwichFactory محول کردیم تا کاربر را درگیر جزییات ایجاد شئ نکنیم. علاوه بر این، دیگر لازم نیست برنامه اجرا را دستخوش تغییر کنیم، بلکه تغییرات را در کلاس SandwichFactory اعمال می‌کنیم. این تغییرات میتواند شامل ساختن کلاس کانکرت جدید، حذف کلاس، و یا تغییر پارامتر لازم برای ایجاد یک شئ باشد.

دوم: Adapter

ممکن است شما نیز هنگام خریدن وسایل برقی متوجه شده باشید که کابل برق هر وسیله‌ای با پریز خانه‌ی ما سازگار نیست. البته ما خیلی شگفت‌زده نمی‌شویم و برای حل مشکل، یک تبدیل خریداری می‌کنیم تا بتوانیم از وسیله استفاده کنیم. مشابه این مسأله در برنامه نویسی نیز ممکن است پیش آید. فرض کنید برنامه‌ای نوشته‌اید که در خروجی اطلاعات را بصورت یک فایل تکست ذخیره می‌کند. یک برنامه نیز از گیت‌هاب دانلود کرده‌اید که اطلاعات را پردازش می‌کند. مسأله این است که برنامه پردازش اطلاعات، فقط فایل اکسل می‌پذیرد. شما اکنون میتوانید کد خود را تغییر دهید، یا یک برنامه بنویسید که ورودی تکست گرفته و خروجی اکسل می‌دهد .

برای نشان دادن روش کار این الگو، که یک الگوی ساختاری محسوب می‌شود ، از مثال یک اسباب‌بازی استفاده می‌کنیم. یک تخته داریم که در آن حفره‌ی گردی وجود دارد که شعاع آن مشخص است. حال می‌خواهیم یک سری میخ با قاعده مربعی را در این حفره‌ها قرار دهیم. اندازه قاعده‌ی میخ‌ها متفاوت است، و ما باید تشخیص دهیم کدام میخ‌ها در این حفره قرار می‌گیرد. واضح است که هر میخ گردی که شعاع قاعده‌ی آن کمتر از شعاع حفره باشد در آن قرار می‌گیرد، اما درباره میخ‌های مربعی که برایشان شعاع تعریف نشده چه باید کرد؟ درست است که میتوان یک اتریبیوت با نام شعاع در کلاس میخ مربعی در نظر گرفت اما شاید ما فقط یک بار از این صفت استفاده کنیم، یا شاید لازم باشد برای اشکال مختلفی این تبدیل را انجام دهیم، این امور نشان می‌دهد که اتریبیوت نوشتن راه حل مناسبی نیست. الگوی Adapter در ورودی یک نوع متغیر دریافت میکند و در خروجی اطلاعاتی که از آن متغیر لازم داریم را با نوع مختلفی برمی‌گرداند.

میخواهیم یک مبدل برای میخ‌های مربعی بسازیم، تا به ما معادل آن میخ را بصورت میخ گرد برگرداند. اگر نصف قطر قاعده را محاسبه کنیم و آن را با شعاع حفره مقایسه کنیم، می توانیم درباره قرار گرفتن یا نگرفتن میخ در حفره اظهارنظر کنیم. نصف قطر را می‌توانیم با استفاده از رابطه فیثاغورث محاسبه کنیم که مقدار آن، به شرطی که اندازه ضلع a باشد برابر است با:

$$Radius = \frac{1}{2} * a * \sqrt{2}$$

ساختار برنامه را در ادامه مشاهده می‌کنید. توجه کنید که SquarePeg Adapter یک میخ مربعی دریافت می‌کند و آن را به یک میخ گرد تبدیل می‌کند. حالا می‌توانیم از این میخ گرد برای چک کردن قرارگیری آن در حفره استفاده کرد.

```
public class RoundPeg {
    public int radius;

    public RoundPeg(int radius){
        this.radius = radius;
    }
}

public class RoundHole {
    public int radius;

    public RoundHole(int radius){
        this.radius = radius;
    }
    public boolean fits(RoundPeg peg){
        return this.radius > peg.radius;
    }
}

public class SquarePeg {
    public int width;

    public SquarePeg(int width){
        this.width = width;
    }
}
```

```
public class SquarePegAdapter extends RoundPeg {
    public SquarePeg peg;

    public SquarePegAdapter(SquarePeg peg){
        super(peg.width);
        this.peg = peg;
    }

    public int getRadius(){
        return (int) (peg.width * Math.sqrt(2) / 2);
    }
}
```