عنوان: **چگونه یک الگوی طراحی را انتخاب و اعمال کنیم؟**
نویسنده: توسعه چابک
تاریخ: ۱۳۹۲/۰۹/۱۲ ۱۳:۵
آدرس: www.dotnettips.info
برچسب‌ها: Design Patternts

با توجه به اینکه الگوهای طراحی زیادی وجود دارند، چگونه می-توانید مناسب-ترین الگوی طراحی را برای حل مسئله خود انتخاب کنید و مهم-تر اینکه چگونه آن را اعمال نمایید؟ برای پاسخ به این سوال، رهنمودهای زیر را همیشه در نظر داشته باشید:

- شما نمی-توانید یک الگو را به کار بگیرید مگر آنکه آن را به خوبی فهمیده باشید. بنابراین در اولین گام باید اصول و الگوهای طراحی را هم به شکل انتزاعی و هم به شکل واقعی خوانده و تمرین کنید. دقت کنید که یک الگو را به شکل-های مختلفی می-توان پیاده سازی کرد. هر چه پیاده سازی-های بیشتری ببینید، به هدف و چگونگی استفاده از آن بهتر مسلط می-شوید.

- آیا می-خواهید با استفاده از یک الگوی طراحی، برنامه خود را پیچیده-تر کنید؟ این معمول است که توسعه دهندگان برای حل هر مسئله از الگوهای طراحی استفاده می-کنند. ابتدا هزینه و فایده پیاده سازی یک الگو را ارزیابی کرده، سپس اقدام به استفاده کنید. همیشه قاعده KISS را به خاطر داشته باشید.

- مسئله خود را تعمیم دهید. پیامدهای مسئله خود را با دید انتزاعی و سطح بالا بررسی کنید. به یاد داشته باشید که الگوهای طراحی، راه حل-های سطح بالا برای مسائل سطح بالا هستند. بنابراین روی پیامدهای جزئی یا وابسته به دامین مسئله خود تمرکز نکنید.

- به الگوهای مشابه و هم گروه نگاه کنید. اگر قبلا از یک الگو استفاده کرده-اید بدین معنی نیست برای هر مسئله-ای آن الگو درست است.

- هر چیزی که تغییر می-کند را بسته بندی کنید. ببینید که چه چیزی در برنامه کاربردی شما ممکن است تغییر کند. اگر شما می-دانید که یک الگوریتم اعمال تخفیف ممکن است به مرور زمان تغییر کند، به دنبال الگویی باشید که تغییرات در آن الگوریتم را بدون تاثیر بر سایر قسمت-های برنامه کاربردی انجام دهید.

- وقتی که یک الگو را انتخاب کردید، از زبان الگو در کنار زبان دامین برای نام گذاری کلاس-ها استفاده کنید. برای مثال اگر از الگوی Strategy استفاده می-کنید تا هزینه حمل و نقل کالا توسط شرکت FedEx را محاسبه کند، از نام FedExShippingCostStrategy استفاده کنید. با استفاده از زبان مشترک بین الگو طراحی و مدل دامین، کد برنامه برای شما و دیگران خواناتر و قابل فهم-تر می-گردد.

- همیشه منظور هر الگو را در ذهن خود مرور کنید و هنگام برخورد با یک مسئله به دنبال مناسب-ترین الگو بگردید. یک تمرین یادگیری عالی شناسایی الگوهای طراحی در فریم ورک .Net است. برای مثال، ASP.Net Cache از الگوی Singleton استفاده می-کند و کلاس Guid از الگوی Factory بهره می-برد.

تا به حال شما باید هدف و الگوریتم استفاده از الگوهای طراحی را درک کرده باشید. در ادامه با لایه بندی برنامه کاربردی آشنا می-شوید و سپس نحوه استفاده از این الگوها در لایه-های مختلف را فرا خواهید گرفت.

نظرات خوانندگان

نویسنده: سید مهدی فاطمی
تاریخ: ۱۳۹۲/۰۹/۱۲ ۱۹:۱۵

تشکر

مطالبی که گفتید رو من به عینه باهاش درگیر بودم و هستم اما تا حالا راه حلی براش پیدا نکردم مثلا الگویی که من استفاده می‌کنم به این صورت هست که برا هر موجودیتی یک فرم در نظر می‌گیرم و در این فرم 4 عمل (جستجو - اضافه - حذف - ویرایش) رو در اون تعبیه می‌کنم که در بعضی مواقع این احساس بهم دست میده که کدهام دارای پیچیدگی شده .

این مطالب شما به صورت نظری هستن اگه امکانش هست مثال هایی به صورت ملموس‌تری بزنید ممنون میشم.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۹/۱۲ ۲۲:۳

مثال اگر نیاز دارید سری بحث‌های معماری لایه بندی نرم افزار در سایت مفید است. همچنین اگر تمام متدها رو داخل یک فرم قرار دادید بهتره از مطلب آشنایی با Refactoring - قسمت 1 شروع کنید.

عنوان: **آشنایی با الگوی طراحی Builder**
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۲ ۲۰:۵
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

سناریوی زیر را در نظر بگیرید:
از شما خواسته شده است تا نحوه‌ی ساخت تلفن همراه را پیاده سازی نمایید. شما در گام اول 2 نوع تلفن همراه را شناسایی نموده‌اید (Android و Windows Phone). پس از شناسایی، احتمالا هر کدام از این انواع را یک کلاس در نظر می‌گیرید و به کمک یک واسط یا کلاس انتزاعی، شروع به ساخت کلاس می‌نمایید، تا در آینده اگر تلفن همراه جدیدی شناسایی شد، راحت‌تر بتوان آن را در پیاده سازی دخیل نمود.
اگر چنین فکر کرده اید باید گفت که 90% با الگوی طراحی Builder آشنا هستید و از آن نیز استفاده می‌کنید؛ بدون اینکه متوجه باشید از این الگو استفاده کرده‌اید. در کدهای زیر این الگو را قدم به قدم بررسی خواهیم نمود. **قدم 1:** تلفن همراه چه بخش هایی می‌تواند داشته باشد؟ (برای مثال یک OS دارند، یک Name دارند و یک Screen) همچنین برای اینکه تلفن همراهی بتواند ساخته شود ابتدا بایستی نام آن‌را بدانیم. کدهای زیر همین رویه را تصدیق می‌نمایند:

```
public class Product
{
        public Product(string name)
        {
            Name = name;
        }
        public string Name { get; set; }
        public string Screen { get; set; }
        public string OS { get; set; }
        public override string ToString()
        {
            return string.Format(Screen + "/" + OS + "/" + Name);
        }
}
```

یک کلاس ساخته‌ایم و نام آن را Product گذاشتیم. بخش‌های مختلفی را نیز در آن تعریف نموده‌ایم. تابع ToString را برای استفاده‌های بعدی override کرده‌ایم (فعلا نیازی بدان نداریم). **قدم 2:** برای ساخت تلفن همراه چه کارهایی باید انجام شود؟ (برای مثال بایستی OS روی آن نصب شود، Screen آن مشخص شود. همچنین بایستی به طریقی بتوانم تلفن همراه ساخته شده‌ی خود را نیز پیدا کنم). کدهای زیر همین رویه را تصدیق می‌نمایند:

```
public interface IBuilder
    {
        void BuildScreen();
        void BuildOS();
        Product Product { get; }
    }
```

یک واسط تعریف کرده‌ایم تا به کمک آن هر تلفن همراهی را که خواستیم بسازیم. **قدم 3:** از آنجا که فقط دو نوع تلفن همراه را فعلا شناسایی کرده‌ایم (Android و Windows Phone) نیاز داریم تا این دو تا را بسازیم.
ابتدا تلفن همراه Android را می‌سازیم:

```
public class ConcreteBuilder1 : IBuilder
    {
        public Product p;
        public ConcreteBuilder1()
        {
            p = new Product("Android Cell Phone");
        }
        public void BuildScreen()
        {
            p.Screen = "Touch Screen 16 Inch!";
        }

        public void BuildOS()
        {
            p.OS = "Android 4.4";
        }
```

```
        public Product Product
        {
            get { return p; }
        }
    }
```

سپس تلفن همراه Windows Phone را می‌سازیم:

```
public class ConcreteBuilder2 : IBuilder
    {
        public Product p;

        public ConcreteBuilder2()
        {
            p = new Product("Windows Phone");
        }
        public void BuildScreen()
        {
            p.Screen = "Touch Screen 32 Inch!";
        }

        public void BuildOS()
        {
            p.OS = "Windows Phone 2014";
        }
        public Product Product
        {
            get { return p; }
        }
    }
```

قدم 4: اول باید OS نصب شود یا Screen مشخص شود؟ برای اینکه توالی کار را مشخص سازم نیاز به یک کلاس دیگر دارم تا اینکار را انجام دهد:

```
public class Director
    {
        public void Construct(IBuilder builder)
        {
            builder.BuildScreen();
            builder.BuildOS();
        }
    }
```

این کلاس در متد Construct خود یک ورودی از نوع IBuilder می‌گیرد و براساس توالی مورد نظر، شروع به ساخت آن می‌کند.
قدم 5: نهایتا میخواهم به برنامه‌ی خود بگویم که تلفن همراه Android را بسازد:

```
Director d = new Director();
ConcreteBuilder1 cb1 = new ConcreteBuilder1();
d.Construct(cb1);
Console.WriteLine(cb1.p.ToString());
```

و به این صورت تلفن همراه من آماده است!
متد ToString در اینجا، همان ToString ابتدای بحث است که آن را Override کردیم.
به این نکته توجه کنید که اگر یک تلفن همراه جدید شناسایی شود، چه مقدار تغییری در کدها نیاز دارید؟ برای مثال تلفن همراه BlackBerry شناسایی شده‌است. تنها کاری که لازم است این است که یک کلاس بصورت زیر ساخته شود:

```
public class BlackBerry: IBuilder
    {
        public Product p;

        public BlackBerry ()
        {
            p = new Product("BlackBerry");
        }
        public void BuildScreen()
        {
            p.Screen = "Touch Screen 8 Inch!";
        }
```

```
        public void BuildOS()
        {
            p.OS = "BlackBerry XXX";
        }
        public Product Product
        {
            get { return p; }
        }
    }
```

عنوان: **آشنایی با الگوی طراحی Bridge**
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۳ ۱۹:۱۵
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

سناریو زیر را در نظر بگیرید:
قصد دارید تا در برنامه‌ی خود ارسال پیام از طریق پیامک و ایمیل را راه اندازی کنید. هر کدام از این روش‌ها نیز برای خود راه‌های متفاوتی دارند. برای مثال ارسال پیامک از طریق وب سرویس یا یک API خارجی و غیره.
کاری را که می‌توان انجام داد، بشرح زیر نیز می‌توان بیان نمود:
ابتدا یک Interface ایجاد می‌کنیم (IBridge) و در آن متد Send را قرار می‌دهیم. این متد یک پارامتر ورودی از نوع رشته می‌گیرد و به کمک آن میتوان اقدام به ارسال پیامک یا ایمیل یا هر چیز دیگری نمود. کلاس‌هایی این واسط را پیاده سازی می‌کنند که یکی از روش‌های اجرای کار باشند (برای مثال کلاس WebService که یک روش ارسال پیامک یا ایمیل است).

```
public interface IBridge
    {
        string Send(string parameter);
    }
    public class WebService: IBridge
    {
        public string Send(string parameter)
        {
            return parameter + " sent by WebService";
        }
    }
    public class API: IBridge
    {
        public string Send(string parameter)
        {
            return parameter + " sent by API";
        }
    }
```

سپس در ادامه به مکانیزمی نیاز داریم تا بتوانیم از طریق آن پیامک یا ایمیل را ارسال کنیم. خوب می‌خواهیم ایمیل ارسال کنیم؛ اولین سوالی که مطرح می‌شود این است که چگونه ارسال کنیم؟ پس باید در مکانیزم خود زیرساختی برای پاسخ به این سوال آماده باشد.

```
public abstract class Abstraction
    {
        public IBridge Bridge;
        public abstract string SendData();
    }
    public class SendEmail : Abstraction
    {
        public override string SendData ()
        {
            return Bridge.Send("Email");
        }
    }
    public class SendSMS: Abstraction
    {
        public override string SendData ()
        {
            return Bridge.Send("SMS");
        }
    }
```

در کد فوق یک کلاس انتزاعی ایجاد کردیم و در آن یک object از نوع واسط خود قرار دادیم. این object به ما کمک می‌کند تا به طریق آن شیوه‌ی ارسال ایمیل یا پیامک را مشخص سازیم و به سوال خود پاسخ دهیم. سپس در ادامه متد SendData آورده شده است که به کمک آن اعلام می‌کنیم که قصد ارسال ایمیل یا پیامک را داریم و نهایتا هر یک از کلاس‌های ایمیل یا پیامک، این متد را برای خود پیاده سازی کرده‌اند.

قبل از ادامه اجازه دهید کمی در مورد بدنه‌ی یکی از متدهای SendData صحبت کنیم. در این متد با کمک Bridge متد Send موجود

در واسط صدا زده شده است. از آنجا که این object از نظر سطح دسترسی عمومی می‌باشد، لذا از بیرون از کلاس قابل دسترسی است. این باعث می‌شود تا قبل از فراخوانی متد SendData موجود در کلاس ایمیل یا پیامک اعلام کنیم که Bridge از چه نوعی است (به چه روشی می‌خواهیم ارسال رخ دهد).

```
Abstraction ab1 = new Email();
  ab1.Bridge = new WebService();
  Console.WriteLine(ab1.SendData ());
```

```
ab1.Bridge = new API();
  Console.WriteLine(ab1.SendData ());
```

```
Abstraction ab2 = new SMS();
   ab2.Bridge = new WebService();
   Console.WriteLine(ab2.SendData ());
```

```
ab2.Bridge = new API();
    Console.WriteLine(ab2.SendData ());
```

نهایتا در کد فوق ابتدا بیان می‌کنیم که قصد ارسال ایمیل را داریم. سپس اعلام می‌داریم که این ارسال را به کمک WebService می‌خواهیم انجام دهی. و نهایتا ارسال را انجام می‌دهیم.

به کل این الگویی که ایجاد کردیم، الگوی Bridge گفته می‌شود.

حال فکر کنید قصد ارسال MMS دارید. در اینصورت فقط کافیست یک کلاس MMS ایجاد کنید و تمام؛ بدون اینکه کدی اضافی را بنویسید یا برنامه را تغییر دهید. یا فرض کنید روش ارسال جدیدی را می‌خواهید اضافه کنید. برای مثال ارسال به روش XYZ. در اینصورت فقط کافیست یک کلاس XYZ را ایجاد کنید که IBridge را پیاده سازی می‌کند.

عنوان: آشنایی با الگوی طراحی Decorator
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۴ ۱۴:۵
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

این بار مثال را با شیرینی و کیک پیش می‌بریم.
فرض کنید شما قصد پخت کیک و نان را دارید. طبیعی است که برای اینکار یک واسط را تعریف کرده و عمل «پختن» را در آن اعلام می‌کنید تا هر کلاسی که قصد پیاده سازی این واسط را داشت، «پختن» را انجام دهد. در ادامه یک کلاس بنام کیک ایجاد خواهید کرد و شروع به پخت آن می‌کنید.
خوب احتمالا الان کیک آماده‌است و می‌توانید آن‌را میل کنید! ولی یک سؤال. تکلیف شخصی که کیک با روکش کاکائو دوست دارد و شمایی که کیک با روکش میوه‌ای دوست دارید چیست؟ این را چطور در پخت اعمال کنیم؟ یا منی که نان کنجدی می‌خواهم و شمایی که نان برشته‌ی غیر کنجدی می‌خواهید چطور؟
احتمالا می‌خواهید سراغ ارث بری رفته و سناریوهای این چنینی را پیاده سازی کنید. ولی در مورد ارث بری، اگر کلاس sealed ((NotInheritable باشه چطور؟
احتمالا همین دو تا سؤال کافی‌است تا در پاسخ بگوئیم، گره‌ی کار، با الگوی Decorator باز می‌شود و همین دو تا سؤال کافی‌است تا اعلام کنیم که این الگو، از جمله الگوهای بسیار مهم و پرکاربرد است.
در ادامه سناریوی خود را با کد ذیل جلو می‌بریم:

```
public interface IBakery
    {
        string Bake();
        double GetPrice();
    }
    public class Cake: IBakery
    {
        public string Bake() { return "Cake baked"; }
        public double GetPrice() { return 2000; }
    }
    public class Bread: IBakery
    {
        public string Bake() { return "Bread baked"; }
        public double GetPrice() { return 100; }
    }
```

در کد فوق فرض کرده‌ام که شما می‌خواهید محصول خودتان را بفروشید و برای آن یک متد GetPrice نیز گذاشته‌ام. خوب در ابتدا واسطی تعریف شده و متدهای Bake و GetPrice اعلام شده‌اند. سپس کلاس‌های Cake و Bread پیاده سازی‌های خودشان را انجام دادند.
در ادامه باید مخلفاتی را که روی کیک و نان می‌توان اضافه کرد، پیاده نمود.

```
public abstract class Decorator : IBakery
    {
        private readonly IBakery _bakery;
        protected string bake = "N/A";
        protected double price = -1;

        protected Decorator(IBakery bakery) { _bakery= bakery; }
        public virtual string Bake() { return _bakery.Bake() + "/" + bake; }
        public double GetPrice() { return _bakery.GetPrice() + price; }
    }
    public class Type1 : Decorator
    {
        public Type1(IBakery bakery) : base(bakery) { bake= "Type 1"; price = 1; }
    }
    public class Type2 : Decorator
    {
        private const string bakeType = "special baked";
        public Type2(IBakery bakery) : base(bakery) { name = "Type 2"; price = 2; }
        public override string Bake() { return base.Bake() + bakeType ; }
    }
```

در کد فوق یک کلاس انتزاعی ایجاد و متدهای پختن و قیمت را پیاده سازی کردیم؛ همچنین کلاس‌های Type1 و Type2 را که من

فرض کردم کلاس‌هایی هستند برای اضافه کردن مخلفات به کیک و نان. در این کلاس‌ها در متد سازنده، یک شیء از نوع IBakery می‌گیریم که در واقع این شیء یا از نوع Cake هست یا از نوع Bread و مشخص می‌کند روی کیک می‌خواهیم مخلفاتی را اضافه کنیم یا بر روی نان. کلاس Type1 روش پخت و قیمت را از کلاس انتزاعی پیروی می‌کند، ولی کلاس Type2 روش پخت خودش را دارد.
با بررسی اجمالی در کدهای فوق مشخص می‌شود که هرگاه بخواهیم، می‌توانیم رفتارها و الحاقات جدیدی را به کلاس‌های Cake و Bread، اضافه کنیم؛ بدون آنکه کلاس اصلی آنها تغییر کند. حال شما شاید در پیاده سازی این الگو از کلاس انتزاعی Decorator هم استفاده نکنید.
با این حال شیوه‌ی استفاده از این کدها هم بصورت زیر خواهد بود:

```
Cake cc1 = new Cake();
 Console.WriteLine(cc1.Bake() + " ," + cc1.GetPrice());

 Type1 cd1 = new Type1 (cc1);
 Console.WriteLine(cd1.Bake() + " ," + cd1.GetPrice());

 Type2 cd2 = new Type2(cc1);
 Console.WriteLine(cd2.Bake() + " ," + cd2.GetPrice());
```

ابتدا یک کیک را پختیم در ادامه Type1 را به آن اضافه کردیم که این باعث می‌شود قیمتش هم زیاد شود و در نهایت Type2 را هم به کیک اضافه کردیم و حالا کیک ما آماده است.

نظرات خوانندگان

نویسنده: محمد اسکندری
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۰:۵۴

در استفاده از الگوی دکوراتور روش بهتر بهره گیری از آن بصورت سری است و نه ایجاد شیء جدید برای تایپ جدید.
مثلا:

```
Cake c = new Cake();

c = new Type1(c);
c = new SubType(c); //SubType derived from  Cake (e.g. CakeComponent like Cream)

//or: c = new Type1 (new SubType(c));

Console.WriteLine(c.Bake() + ", " + c.GetPrice());
```

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۱:۴۴

بستگی به هدف نهایی دارد. اگر هدف تولید کیک با روکش کاکائویی و روکش میوه‌ای به صورت همزمان است، نحوه‌ی تزئین آن با کیکی که فقط قرار هست روکش کاکائویی داشته باشه، فرق می‌کنه.

نویسنده: محمد اسکندری
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۲:۰

فرقی نمیکنه. اگر قرار بود فرق میکرد و نیاز به ایجاد تغییرات در کد بود که این الگوها ارائه نمی‌شدند.

```
// ساخت کیک معمولی با روکش کاکائویی
Cake c = new CommonCake();
c = new Chocolate(c);

// ساخت کیک معمولی با روکش میوه‌ای
Cake c = new CommonCake();
c = new Fruity(c);

// ساخت کیک معمولی مخلوط با روکش کاکائویی و روکش میوه‌ای به صورت همزمان
Cake c = new CommonCake();
c = new Chocolate(c);
c = new Fruity(c);

// ساخت کیک مخصوص مخلوط با روکش کاکائویی و روکش میوه‌ای به صورت همزمان
Cake c = new SpecialCake();
c = new Chocolate(c);
c = new Fruity(c);
```

برای هر c میتوان متدهای اینترفیسش را اجرا کرد.

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۲:۸

عنوان کردید «در استفاده از الگوی دکوراتور روش بهتر بهره گیری از آن بصورت **سری** است و نه ایجاد شیء **جدید** برای تایپ جدید»، بعد الان برای تهیه روکش فقط میوه‌ای از حالت سری استفاده نکردید و یک وهله جدید ایجاد شده. بحث بر سر سری بودن یا نبودن مراحل بود. بنابراین بسته به هدف، می‌تونه سری باشه یا نباشه و اگر نبود، مشکلی نداره، چون هدفش تولید یک روکش مخصوص بوده و نه ترکیبی.

نویسنده: محمد اسکندری
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۲:۲۳

فرض من این بود که کاربر نیازی به رفرنس گیری از هر آبجکت ندارد.
مثلا طبق مقاله:

```
// ساخت کیک مخصوص مخلوط با روکش کاکائویی و روکش میوه‌ای به صورت همزمان
Cake c = new SpecialCake();
Chocolate ch = new Chocolate(c);
Fruity f = new Fruity(ch);
```

همانطور که در مقاله گفته شده:

```
Cake cc1 = new Cake();
Type1 cd1 = new Type1 (cc1);
Type2 cd2 = new Type2(cc1);
```

کد فوق را میتوان اینگونه **هم** داشت:

```
// ساخت کیک مخصوص مخلوط با روکش کاکائویی و روکش میوه‌ای به صورت همزمان
Cake c = new SpecialCake();
c = new Chocolate(c);
c = new Fruity(c);
```

بدون اینکه *شیء جدید برای تایپ جدید* بسازیم.

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۲:۴۳

مهم این نیست که نام تمام متغیرها را c تعریف کردید، مهم این است که به ازای هر new یک شیء کاملا جدید ایجاد می‌شود که ریفرنس آن با ریفرنس قبلی یکی نیست.

عنوان: **آشنایی با الگوی طراحی Prototype**
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۴ ۱۹:۲۰
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

فرض کنید در حال پختن یک کیک هستید. ابتدا کیک را می‌پذید و سپس آن را تزیین می‌کنید. عملیات پختن کیک، فرآیند ثابتی است و تزیین کردن آن متفاوت. گاهی کیک را با کاکائو تزیین می‌کنید و گاهی با میوه و غیره.
پیش از اینکه سناریو را بیش از این جلو ببریم، وارد بحث کد می‌شویم. طبق سناریوی فوق، فرض کنید کلاسی بنام Prototype دارید که این کلاس هم از کلاس انتزاعی APrototype ارث برده است. در ادامه یک شیء از این کلاس می‌سازید و مقادیر مختلف آن را تنظیم کرده و کار را ادامه می‌دهید.

```
public abstract class APrototype : ICloneable     {
        public string Name { get; set; }
        public string Health { get; set; }
    }

    public class Prototype : APrototype
    {
        public override string ToString() { return string.Format("Player name: {0}, Health statuse:
{1}", Name, Health); }
    }
```

در ادامه از این کلاس نمونه‌گیری می‌کنیم:

```
Prototype p1 = new Prototype { Name = "Vahid", Health = "OK" };
Console.WriteLine(p1.ToString());
```

حالا فرض کنید به یک آبجکت دیگر نیاز دارید، ولی این آبجکت عینا مشابه p1 است؛ لذا نمونه‌گیری، از ابتدا کار مناسبی نیست.
برای اینکار کافیست کدها را بصورت زیر تغییر دهیم:

```
public abstract class APrototype : ICloneable
    {
        public string Name { get; set; }
        public string Health { get; set; }
        public abstract object Clone();
    }

    public class Prototype : APrototype
    {
        public override object Clone() { return this.MemberwiseClone() as APrototype; }
        public override string ToString() { return string.Format("Player name: {0}, Health statuse:
{1}", Name, Health); }
    }
```

در متد Clone از MemberwiseClone استفاده کرده‌ایم. خود Clone هم در داخل واسط ICloneable تعریف شده‌است و هدف از آن کپی نمودن آبجکت‌ها است. سپس کد فوق را بصورت زیر مورد استفاده قرار می‌دهیم:

```
Prototype p1 = new Prototype { Name = "Vahid", Health = "OK" };
Prototype p2 = p1.Clone() as Prototype;
Console.WriteLine(p1.ToString());
Console.WriteLine(p2.ToString());
```

با اجرای کد فوق مشاهده میشود p1 و p2 دقیقا عین هم کار می‌کنند. کل این فرآیند بیانگر الگوی Prototype می‌باشد. ولی تا اینجای کار درست است که الگو پیاده سازی شده است، ولی همچنین به نظر نقصی نیز در کد دیده می‌شود:
برای واضح نمودن نقص، یک کلاس بنام AdditionalDetails تعریف می‌کنیم. در واقع کد را بصورت زیر تغییر میدهیم:

```
public abstract class APrototype : ICloneable
    {
        public string Name { get; set; }
        public string Health { get; set; }
```

```
        public AdditionalDetails Detail { get; set; }
        public abstract object Clone();
    }
    public class AdditionalDetails { public string Height { get; set; } }

    public class Prototype : APrototype
    {
        public override object Clone() { return this.MemberwiseClone() as APrototype; }
        public override string ToString() { return string.Format("Player name: {0}, Health statuse:
{1}, Height: {2}", Name, Health, Detail.Height); }
    }
```

و از آن بصورت زیر استفاده می‌کنیم:

```
Prototype p1 = new Prototype { Name = "Vahid", Health = "OK", Detail = new AdditionalDetails { Height =
"100" } };
Prototype p2 = p1.Clone() as Prototype;
p2.Detail.Height = "200";
Console.WriteLine(p1.ToString());
Console.WriteLine(p2.ToString());
```

خروجی که نمایش داده می‌شود در بخش Height هم برای p1 و هم برای p2 عدد 200 را نمایش می‌دهد که می‌تواند اشتباه باشد. چراکه p1 دارای Height برابر با 100 است و p2 دارای Height برابر با 200. به این اتفاق ShallowCopy گفته میشود که ناشی از استفاده از MemberwiseClone است که در مورد ارجاعات با آدرس رخ می‌دهد. در این حالت بجای کپی نمودن مقدار، از کپی نمودن آدرس استفاده میشود (Ref Type چیست؟)

برای حل این مشکل باید DeepCopy انجام داد. لذا کد را بصورت زیر تغییر می‌دهیم:(DeepCopy و ShallowCopy چیست؟)

```
public abstract class APrototype : ICloneable
    {
        public string Name { get; set; }
        public string Health { get; set; }
        //This is a ref type
        public AdditionalDetails Detail { get; set; }
        public abstract APrototype ShallowClone();
        public abstract object Clone();
    }

    public class AdditionalDetails { public string Height { get; set; } }

    public class Prototype : APrototype
    {
        public override object Clone()
        {
            Prototype cloned = MemberwiseClone() as Prototype;
            //We need to deep copy each ref types in order to prevent shallow copy
            cloned.Detail = new AdditionalDetails { Height = this.Detail.Height };
            return cloned;
        }
        //Shallow copy will copy ref type's address instead of their value, so any changes in cloned
object or source object will take effect on both objects
        public override APrototype ShallowClone() { return this.MemberwiseClone() as APrototype; }
        public override string ToString() { return string.Format("Player name: {0}, Health statuse:
{1}, Height: {2}", Name, Health, Detail.Height); }
    }
```

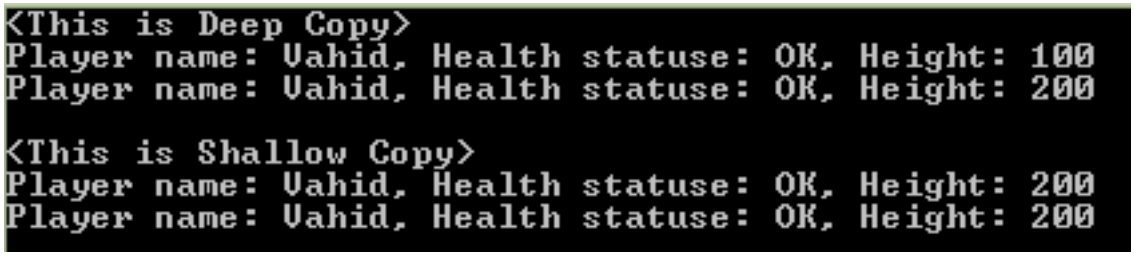
و سپس بصورت زیر از آن استفاده نمود:

```
Prototype p1 = new Prototype { Name = "Vahid", Health = "OK", Detail = new AdditionalDetails { Height =
"100" } };
            Prototype p2 = p1.Clone() as Prototype;
            p2.Detail.Height = "200";
            Console.WriteLine("<This is Deep Copy>");
            Console.WriteLine(p1.ToString());
            Console.WriteLine(p2.ToString());

            Prototype p3 = new Prototype { Name = "Vahid", Health = "OK", Detail = new
AdditionalDetails { Height = "100" } };
            Prototype p4 = p3.ShallowClone() as Prototype;
            p4.Detail.Height = "200";
            Console.WriteLine("\n<This is Shallow Copy>");
```

```
            Console.WriteLine(p3.ToString());
            Console.WriteLine(p4.ToString());
```

لذا خروجی بصورت زیر را می‌توان مشاهده نمود:



البته در این سناریو ShallowCopy باعث اشتباه شدن نتایج می‌شود. شاید شما در دامنه‌ی نیازمندیهای خود، اتفاقا به ShallowCopy نیاز داشته باشید و DeepCopy مرتفع کننده‌ی نیاز شما نباشد. لذا کاربرد هر کدام از آنها وابستگی مستقیمی به دامنه‌ی نیازمندی‌های شما دارد.

عنوان: **آشنایی با الگوی طراحی Fly Weight**
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۴:۲۰
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

سناریوی زیر را در نظر بگیرید:
فرض کنید از شما خواسته شده است تا یک پردازشگر متن را بنویسید. خوب در این پردازشگر با یک‌سری کاراکتر روبرو هستید که هر کاراکتر احتمالا آبجکتی از نوع کلاس خود می‌باشد؛ برای مثال آبجکت XYZ که آبجکتی از نوع کلاس A هست و برای نمایش کاراکتر A استفاده می‌شود. این آبجکت‌ها دارای دو دسته خصیصه هستند: (مطالعه بیشتر)
خصیصه‌های ثابت: یعنی همه کاراکترهای A دارای یک شکل مشخص هستند. در واقع مشخصات ذاتی آبجکت می‌باشند.

خصیصه‌های پویا: یعنی هر کاراکتر دارای فونت، سایز و رنگ خاص خود است. در واقع خصیصه‌هایی که از یک آبجکت به آبجکت دیگر متفاوت هستند .

خوب احتمالا در ساده‌ترین راه حل، به ازای تک تک کاراکترهایی که کاربر وارد می‌کند، یک آبجکت از نوع کلاس متناسب با آن ساخته می‌شود. ولی بحث مهم این است که با این همه آبجکت که هر یک مصرف خود را از حافظه دارند، می‌خواهید چکار کنید؟ احتمالا به مشکل حافظه برخورد خواهید کرد! پس باید یک سناریوی بهتر ایجاد کرد.
سناریوی پیشنهادی این است که برای هر نوع کاراکتر، یک کلاس داشته باشیم، همانند قبل(یک کلاس برای A یک کلاس برای B و غیره) و یک استخر پر از آبجکت داشته باشیم که آبجکت‌های ایجاد شده در آن ذخیره شوند.
سپس کاربر، کاراکتر A را درخواست می‌کند. ابتدا به این استخر نگاه می‌کنیم. اگر کاراکتر A موجود بود، آن را برمی‌گردانیم و اگر موجود نبود، یک آبجکت از نوع A می‌سازیم، سپس این آبجکت را در استخر ذخیره می‌کنیم و آبجکت را بر می‌گردانیم. در این صورت اگر کاربر دوباره درخواست A را کرد، دیگر نیازی به ساخت آبجکت جدید نیست و از آبجکت قبلی می‌توانیم استفاده نماییم. با این شرایط تکلیف خصایص ایستا مشخص است. ولی مشکل مهم با خصایص پویا این است که می‌توانند بین آبجکت‌ها متفاوت باشند که برای این هم یک متد در کلاس‌ها قرار می‌دهیم تا این خصایص را تنظیم نماید.
به کد زیر دقت نمایید:

```
public interface IAlphabet
    {
        void Render(string font);//Define Extrinsic and non-static states for each object
    }

    public class A : IAlphabet
    {
        public void Render(string font) { Console.WriteLine(GetType().Name + " has font of type " +
font); }
    }
    public class B : IAlphabet
    {
        public void Render(string font) { Console.WriteLine(GetType().Name + " has font of type " +
font); }
    }
```

از متد Render برای تنظیم نمودن خصایص پویا استفاده خواهد شد.
سپس در ادامه به یک موتور نیاز داریم که قبل از ساخت آبجکت، استخر را بررسی نماید:

```
public class FlyWeightFactory
    {
        private readonly Dictionary<string, IAlphabet> _dictionary = new Dictionary<string,
IAlphabet>();
        public int Count { get { return _dictionary.Count; } }
        public IAlphabet GetObject(string name)
        {
            if (!_dictionary.ContainsKey(name))
                switch (name)
                {
                    case "A":
                        _dictionary.Add(name, new A());
```

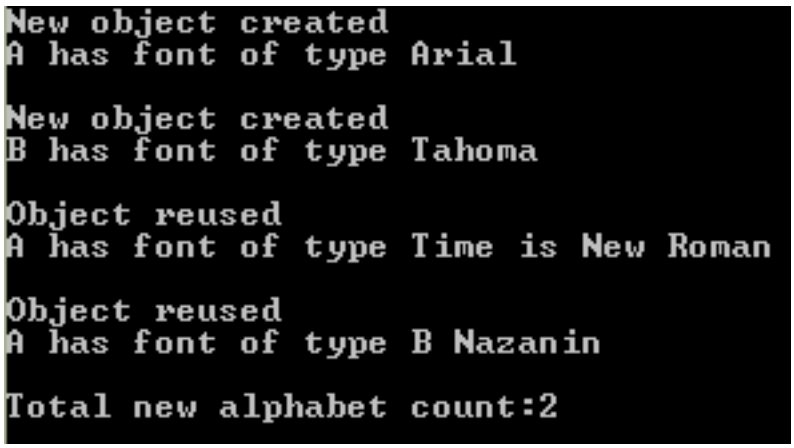
```
                    Console.WriteLine("New object created");
                    break;
                case "B":
                    _dictionary.Add(name, new B());
                    Console.WriteLine("New object created");
                    break;
                default:
                    throw new Exception("Factory can not create given object");
            }
        else
            Console.WriteLine("Object reused");
        return _dictionary[name];
    }
}
```

در اینجا _dictionaries همان استخر ما می‌باشد که قرار است آبجکت‌ها در آن ذخیره شوند. Count برای نمایش تعداد آبجکت‌های موجود در استخر استفاده می‌شود (حداکثر مقدار آن چقدر خواهد بود؟). GetObject نیز همان موتور اصلی کار است که در آن ابتدای استخر بررسی می‌شود. اگر آبجکت در استخر نبود، یک نمونه‌ی جدید از آن ساخته شده، به استخر اضافه گردیده و برگردانده می‌شود.

لذا برای استفاده‌ی از این کد داریم:

```
FlyWeightFactory flyWeightFactory = new FlyWeightFactory();
 IAlphabet alphabet = flyWeightFactory.GetObject(typeof(A).Name);
 alphabet.Render("Arial");
 Console.WriteLine();
 alphabet = flyWeightFactory.GetObject(typeof(B).Name);
 alphabet.Render("Tahoma");
 Console.WriteLine();
 alphabet = flyWeightFactory.GetObject(typeof(A).Name);
 alphabet.Render("Time is New Roman");
 Console.WriteLine();
 alphabet = flyWeightFactory.GetObject(typeof(A).Name);
 alphabet.Render("B Nazanin");
 Console.WriteLine();
 Console.WriteLine("Total new alphabet count:" + flyWeightFactory.Count);
```

با اجرای این کد خروجی زیر را مشاهده خواهید نمود:



نکته‌ی قابل توجه این است که این الگو بصورت داخلی از الگوی Factory Method استفاده می‌کند. با توجه بیشتر به پیاده سازی Flyweight Factory شباهت هایی بین آن و Singleton Pattern می‌بینیم. کلاس‌هایی از این دست را Multiton می نامند. در Multiton نمونه‌ها بصورت زوج کلیدهایی نگهداری می‌شوند و بر اساس Key دریافت شده نمونه‌ی متناظر بازگردانده می‌شود. همچنین در Singleton تضمین می‌شود که از کلاس مربوطه فقط یک نمونه در کل Application وجود دارد. در Multiton Pattern تضمین می‌شود که برای هر Key تنها یک Instance وجود دارد.

عنوان: **آشنایی با الگوی طراحی Abstract Factory**
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۹:۱۰
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

قبل از مطالعه‌ی این مطلب، حتما الگوی طراحی Factory Method را مطالعه نمایید.
همانطور که در الگوی طراحی Factory Method مشاهده شد، این الگو یک عیب دارد، آن هم این است که از کدام Creator باید استفاده شود و مستقیما در کد بایستی ذکر شود.

```
class ConcreteCreator : Creator
{
     public override IProduct FactoryMethod(string type)
    {
            switch (type)
           {
                case "A": return  new ConcreteProductA();
                case "B": return  new ConcreteProductB();
                default: throw new ArgumentException("Invalid type", "type");
           }
     }
}
```

برای حل این مشکل می‌توانیم سراغ الگوی طراحی دیگری برویم که Abstract Factory نام دارد. این الگوی طراحی 4 بخش اصلی دارد که هر کدام از این بخش‌ها را طی مثالی توضیح می‌دهم:
1. Abstract Factory: در کشور، صنعت خودروسازی داریم که خودروها را در دو دسته‌ی دیزلی و سواری تولید می‌کنند :

```
public interface IVehicleFactory     {
        IDiesel GetDiesel();
        IMotorCar GetMotorCar();
    }
```

2. Concrete Factory: دو کارخانه‌ی تولید خودرو داریم که در صنعت خودرو سازی فعالیت دارند و عبارتند از ایران خودرو و سایپا که هر کدام خودروهای خود را تولید می‌کنند. ولی هر خودرویی که تولید می‌کنند یا دیزلی است یا سواری. شرکت ایران خودرو، خودروی آرنا را بعنوان دیزلی تولید می‌کند و پژو 206 را بعنوان سواری. همچنین شرکت سایپا خودروی فوتون را بعنوان خودروی دیزلی تولید می‌کند و خودروی پراید را بعنوان خودروی سواری.

```
public class IranKhodro : IVehicleFactory
    {
        public IDiesel GetDiesel() { return new Arena(); }
        public IMotorCar GetMotorCar() { return new Peugeot206(); }
    }
    public class Saipa : IVehicleFactory
    {
        public IDiesel GetDiesel() { return new Foton(); }
        public IMotorCar GetMotorCar() { return new Peride(); }
    }
```

3. Abstract Product: خودروهای تولیدی همانطور که گفته شد یا دیزلی هستند یا سواری که هر کدام از این خودروها ویژگی‌های خاص خود را دارند (در این مثال هر دو دسته خودرو برای خود نام دارند)

```
public interface IDiesel { string GetName();}
    public interface IMotorCar { string GetName();}
```

4. Concrete Product: در بین این خودروها، خودروی پژو 206 و پراید یک خودروی سواری هستند و خودروی فوتون و آرنا، خودروهای دیزلی.

```
public class Foton : IDiesel { public string GetName() { return "This is Foton"; } }
    public class Arena : IDiesel { public string GetName() { return "This is Arena"; } }
    public class Peugeot206 : IMotorCar { public string GetName() { return "This is Peugeot206"; } }
```

```
    public class Peride : IMotorCar { public string GetName() { return "This is Peride"; } }
```

حال که 4 دسته اصلی این الگوی طراحی را آموختیم می‌توان از آن بصورت زیر استفاده نمود:

```
IVehicleFactory factory = new IranKhodro();
 Console.WriteLine("***" + factory.GetType().Name + "***");
 IDiesel diesel = factory.GetDiesel();
 Console.WriteLine(diesel.GetName());
 IMotorCar motorCar = factory.GetMotorCar();
 Console.WriteLine(motorCar.GetName());

 factory = new Saipa();
 Console.WriteLine("***" + factory.GetType().Name + "***");
 diesel = factory.GetDiesel();
 Console.WriteLine(diesel.GetName());
 motorCar = factory.GetMotorCar();
 Console.WriteLine(motorCar.GetName());
```

همانطور که در کد فوق مشاهده میشود، ایراد موجود در الگوی Factory Method اینجا از بین رفته است و برای ساخت آبجکت‌های مختلف از Innterfaceها یا Abstract Classها استفاده می‌کنیم.

کلا Abstract Factory مزایای زیر را دارد:

پیاده سازی و نامگذاری Product در Factory مربوطه متمرکز می‌شود و بدین ترتیب Client به نام و نحوه پیاده سازی Typeهای مختلف Product وابستگی نخواهد داشت.

به راحتی می‌توان Concrete Factory مورد استفاده در برنامه را تغییر داد، بدون اینکه تاثیری در عملکرد سایر بخش‌ها داشته باشد.

در مواردی که بیش از یک محصول برای هر خانواده وجود داشته باشد، استفاده از Abstract Factory تضمین می‌کند که Productهای هر خانواده همه در کنار هم قرار دارند و با هم فعال و غیر فعال می‌شوند. (یا همه، یا هیچکدام)

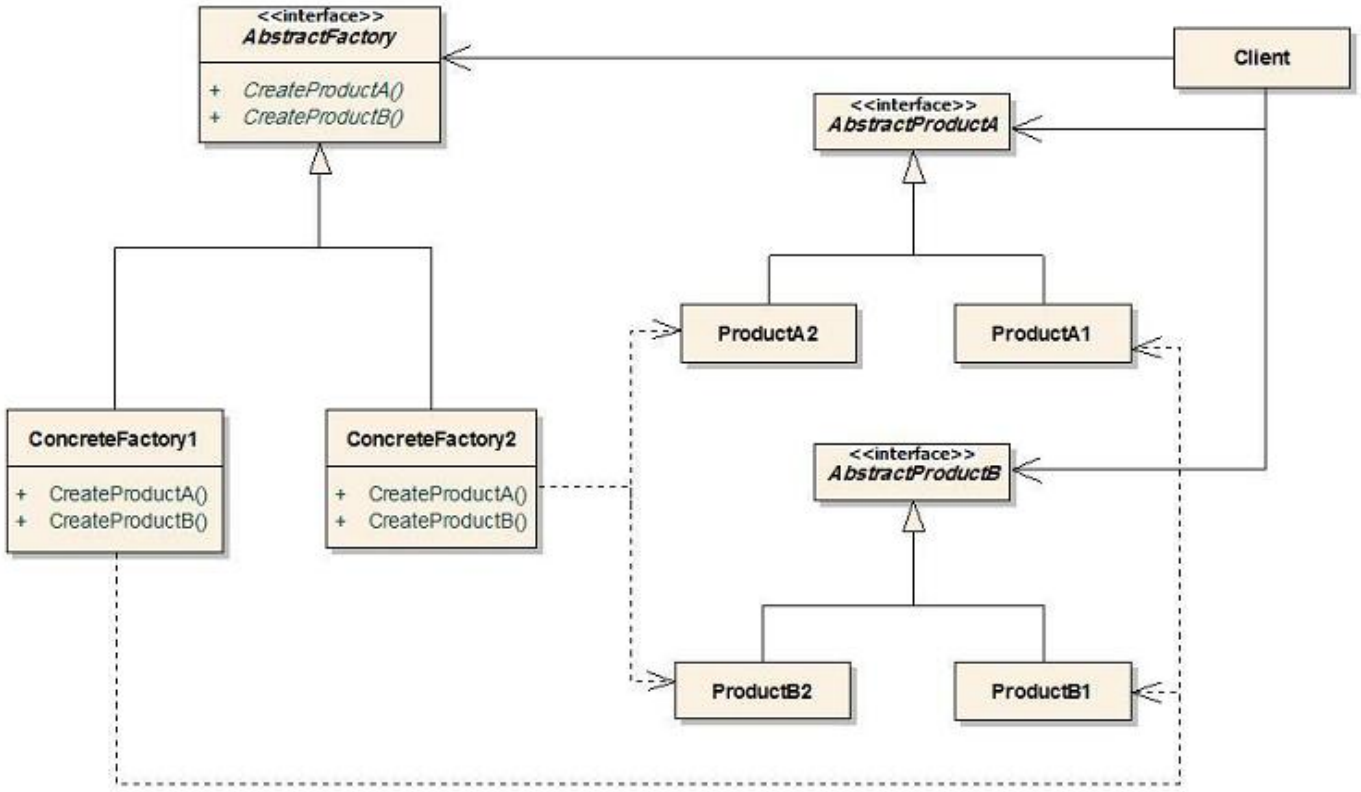
بزرگترین عیبی که این الگوی طراحی دارد این است که با اضافه شدن فقط یک Product تازه، Abstract Factory باید تغییر کند که این مساله منجر به تغییر همه Concrete Factoryها می‌شود.

نهایتا اینکه در استفاده از این الگوی طراحی به این تکنیک‌ها توجه داشته باشید:

Factoryها معمولا Singleton هستند. زیرا هر Application **بطور معمول** فقط به یک instance از هر Concrete Factory نیاز دارد.

انتخاب Concrete Factory مناسب معمولا توسط پارامترهایی انجام می‌شود.

نمودار کلاسی این الگو نیز بصورت زیر میباشد:



و کلام آخر در مورد این الگو:

Abstract Factory یک interface یا کلاس abstract است که signature متدهای ساخت Objectها در آن تعریف شده است و Concrete Factoryها آن‌ها را implement می‌نمایند.

در Abstract Factory Pattern همه Productهای هم خانواده در Concrete Factory مربوط به آن خانواده پیاده سازی و مجتمع می‌گردند.

در کدهای برنامه تنها با Abstract Factory و Abstract Productها سر و کار داریم و به هیچ وجه درگیر این مساله که کدام یک از Concrete Classها در برنامه مورد استفاده قرار می‌گیرند، نمی‌شویم.

عنوان: **آشنایی با الگوی طراحی Template Method**
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۹ ۱۸:۲۵
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

سناریویی وجود دارد که در آن شما می‌خواهید تنها یک کار را انجام دهید، ولی برای انجام آن n روش وجود دارد. برای مثال قصد مرتب سازی دارید و برای اینکار روش‌های مختلفی وجود دارند. برای حل این مساله پیشتر از الگوی طراحی استراتژی استفاده نمودیم.(مطالعه بیشتر در مورد الگوی طراحی استراتژی)
حال به سناریویی برخورد کردیم که بصورت زیر است:
می‌خواهیم یک کار را انجام دهیم ولی برای انجام این کار تنها برخی بخش‌های کار با هم متفاوت هستند. برای مثال قصد تولید گزارش و چاپ آن را داریم. در این سناریو خواندن اطلاعات و پردازش آن‌ها رخدادهایی ثابت هستند. ولی اگر بخواهیم گزارش را چاپ کنیم به مشکل می‌خوریم؛ چرا که چاپ گزارش به فرمت اکسل، فرمت و روش خود را دارد و چاپ به فرمت PDF شرایط خود را دارد.
در این سناریو دیگر الگوی طراحی استراتژی جواب نخواهد داد و نیاز داریم با یک الگوی طراحی جدید آشنا بشویم. این الگوی طراحی Template Method نام دارد.
در این الگو یک کلاس انتزاعی داریم به صورت زیر:

```
public abstract class DataExporter
    {
        public void ReadData()
        {
            Console.WriteLine("Data is reading from SQL Server Database");
        }

        public void ProcessData()
        {
            Console.WriteLine("Data is processing...!");
        }

        public abstract void PrintData();

        public void GetReport()
        {
            ReadData();
            ProcessData();
            PrintData();
        }
    }
```

این کلاس abstract، یک متد بنام GetReport دارد که نحوه‌ی انجام کار را مشخص می‌کند. متدهای ReadData و ProcessData نشان می‌دهند که انجام این دو عمل همیشه ثابت هستند (منظور در این سناریو همیشه ثابت هستند). متد PrintData همانطور که مشاهده می‌شود بصورت انتزاعی تعریف شده است، چرا که چاپ عملی است که در هر فرمت دارای خروجی متفاوتی می‌باشد.
لذا در ادامه داریم:

```
public class ExcelExporter : DataExporter
    {
        public override void PrintData()
        {
            Console.WriteLine("Data exported to Microsoft Excel!");
        }
    }

    public class PDFExporter : DataExporter
    {
        public override void PrintData()
        {
            Console.WriteLine("Data exported to PDF!");
        }
    }
```

کلاس ExcelExporter برای چاپ به فرمت اکسل می‌باشد. همانطور که مشاهده می‌شود این کلاس از کلاس انتزاعی DataExporter ارث بری کرده است. این بدین معنا است که کلاس ExcelExporter کارهای ReadData و ProcessData را از کلاس

پدر خود می‌گیرد و در ادامه نحوه‌ی چاپ مختص به خود را پیاده می‌کند. همین توضیحات در مورد PDFExporter نیز صادق است.
حال برای استفاده‌ی از این کدها داریم:

```
DataExporter dataExporter = new ExcelExporter();
            dataExporter.GetReport();
            Console.WriteLine("**************************");
            dataExporter = new PDFExporter();
            dataExporter.GetReport();
```

شما شاید بخواهید متدهای ReadData و ExportData و ProcessData را با سطح دسترسی متفاوتی از public تعریف نمایید که در این مقاله به این دلیل که خارج از بحث بود به آنها اشاره نشد و بصورت پیش فرض public در نظر گرفته شد.