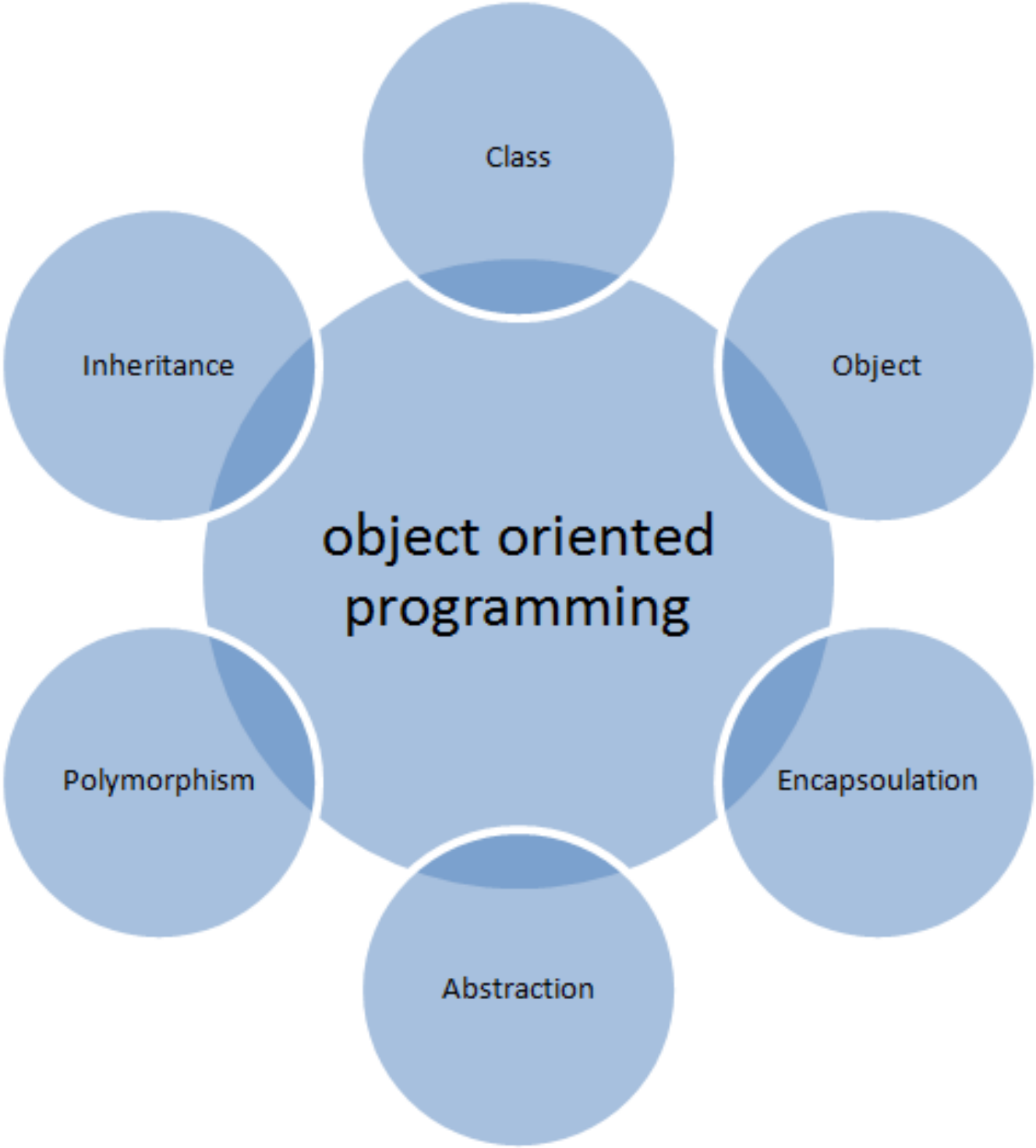
عنوان: **اصول طراحی شی گرا SOLID - #بخش اول اصل SRP**
نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۰۶ ۲۰:۳۰
آدرس: www.dotnettips.info
گروه‌ها: OOP, SOLID Principals, Object Oriented Design, Single Responsibility

مخاطب چه کسی است؟

این مقاله برای کسانی در نظر گرفته شده است که حداقل پیش زمینه ای در مورد برنامه نویسی شی گرا داشته باشند.کسانی که تفاوت بین کلاس‌ها و اشیاء را میدانند و میتوانند در مورد ارکان پایه ای برنامه نویسی شی گرایی نظیر : کپسوله سازی (Encapsulation) ، کلاس‌های انتزاعی (Abstraction) ، چند ریختی (Polymorphism) ، ارث بری (Inheritance) و... صحبت کنند.



مقدمه :

در جهان شی گرا ما فقط اشیاء را میبینیم که با یکدیگر در ارتباط هستند.کلاس ها، شی ها، ارث بری، کپسوله سازی ، کلاس‌های انتزاعی و ... کلماتی هستند که ما هر روز در حرفه‌ی خودمان بارها آنها را میشنویم.

در دنیای مدرن نرم افزار، بدون شک هر توسعه دهنده‌ی نرم افزار ،یکی از انواع زبان‌های شی گرا را برای استفاده انتخاب میکند. اما آیا او واقعا میداند که برنامه نویسی شی گرا به چه معنی است؟ آیا او واقعا از قدرت شی گرایی استفاده میکند؟

در این مقاله تصمیم گرفته ایم پای خود را فراتر از ارکان پایه ای برنامه نویسی شی گرا قرار دهیم و بیشتر در مورد طراحی شی گرا صحبت کنیم. **طراحی شی گرا :**

طراحی شی گرا یک فرایند از برنامه ریزی یک سیستم نرم افزاری است که در آن اشیاء برای حل مشکلات خاص با یکدیگر در ارتباط هستند. در حقیقت یک طراحی شی گرای مناسب ، کار توسعه دهنده را آسان میکند و یک طراحی نامناسب تبدیل به یک

فاجعه برای او میشود. **هر کسی چگونه شروع میکند؟**

وقتی کسانی شروع به ایجاد معماری نرم افزار میکنند، نشان میدهند که اهداف خوبی در سر دارند.آنها سعی میکنند از تجارب خود برای ساخت یک طراحی زیبا و تمیز استفاده کنند.

اما با گذشت زمان، نرم افزار کارایی خود را از دست میدهد و بلااستفاده میشود. با هر درخواست ایجاد ویژگی جدید در نرم افزار ، به تدریج نرم افزار شکل خود را از دست میدهد و درنهایت ساده‌ترین تغییرات در نرم افزار موجب تلاش و دقت زیاد، زمان طولانی و مهمتر از همه بالا رفتن تعداد باگها در نرم افزار میشود.

چه کسی مقصر است؟

"تغییرات" یک قسمت جدایی ناپذیر از جهان نرم افزار هستند بنابراین ما نمتوانیم "تغییر" را مقصر بدانیم و در حقیقت این طراحی ما است که مشکل دارد.

یکی از بزرگترین دلایل مخرب کننده‌ی نرم افزار، تعریف وابستگی‌های ناخواسته و بیخود در قسمت‌های مختلف سیستم است. در این گونه طراحی‌ها ، هر قسمت از سیستم وابسته به چندین قسمت دیگر است ، بنابراین تغییر یک قسمت، بر روی قسمت‌های دیگر نیز تاثیر میگذارد و باعث این چنین مشکلاتی میشود. ولی در صورتی که ما قادر به مدیریت این وابستگی باشیم در آینده خواهیم توانست از این سیستم نرم افزاری به آسانی نگهداری کنیم.

مثال:

راه حل : اصول ، الگوهای طراحی و معماری نرم افزار معماری نرم افزار به عنوان مثال MVC, MVP, 3-Tire به ما میگویند که پروژه‌ها از از چه ساختاری استفاده میکنند. **الگوهای طراحی** یک سری راه حل‌های قابل استفاده‌ی مجدد را برای مسائلی که به طور معمول اتفاق می‌افتند، فراهم میکند. یا به عبارتی دیگر الگوهای طراحی راه کارهایی را به ما معرفی میکنند که میتوانند برای حل مشکلات کد نویسی بارها مورد استفاده قرار بگیرند. **اصول به ما میگوید** اینها را انجام بده تا به آن دست پیدا کنی واینکه چطور انجامش میدهی به خودت بستگی دارد. هر کس یک سری اصول را در زندگی خود تعریف میکند مانند : "من هرگز دروغ نمیگویم" یا "من هرگز سیگار نمیکشم" و از این قبیل. او با دنبال کردن این اصول زندگی آسانی را برای خودش ایجاد میکند. به همین شکل، طراحی شی گرا هم مملو است از اصولی که به ما اجازه میدهد تا با طراحی مناسب مشکلاتمان را مدیریت کنیم.

آقای رابرت مارتین(Robert Martin) این موارد را به صورت زیر طبقه بندی کرده است :

1- اصول طراحی کلاس‌ها که SOLID نامیده می‌شوند.

2- اصول انسجام بسته بندی

3- اصول اتصال بسته بندی

در این مقاله ما در مورد اصول SOLID به همراه مثال‌های کاربردی صحبت خواهیم کرد.

SOLID مخففی از 5 اصول معرفی شده توسط آقای مارتین است:

```
S -> Single responsibility Principle
```

```
O-> Open Close Principle
```

```
L-> Liskov substitution principle
```

```
I -> Interface Segregation principle
```

```
D-> Dependency Inversion principle
```

اصل 1) S - SRP - Single responsibility Principle

به کد زیر توجه کنید :

```
public class Employee
{
    public string EmployeeName { get; set; }
    public int EmployeeNo { get; set; }

    public void Insert(Employee e)
    {
        //Database Logic written here
    }
    public void GenerateReport(Employee e)
    {
        //Set report formatting
    }
}
```

در کد بالا هر زمان تغییری در یک قسمت از کد ایجاد شود این احتمال وجود دارد که قسمت دیگری از آن مورد تاثیر این تغییر قرار بگیرد و به مشکل برخورد کنید. دلیل نیز مشخص است : هر دو در یک خانه‌ی مشابه و دارای یک والد یکسان هستند.

برای مثال با تغییر یک پراپرتی ممکن است متدهای هم خانه که از آن استفاده میکنند با مشکل مواجه شوند و باید این تغییرات را نیز در آنها انجام داد. در هر صورت خیلی مشکل است که همه چیز را کنترل کنیم. بنابراین تنها تغییر موجب دوبرابر شدن عملیات تست میشود و شاید بیشتر.

اصل **SRP** برای رفع این مشکل میگوید **"هر ماژول نرم افزاری میبایست تنها یک دلیل برای تغییر داشته باشد"** .

(منظور از ماژول نرم افزاری همان کلاس‌ها ، توابع و ... است و عبارت "دلیل برای تغییر" همان مسئولیت است.) به عبارتی هر شی باید یک مسئولیت بیشتر بر عهده نداشته باشد. هدف این قانون جدا سازی مسئولیت-های چسبیده به هم است. به عنوان مثال کلاسی که هم مسئول ذخیره سازی و هم مسئول ارتباط با واسط کاربر است، این اصل را نقض می-کند و باید به دو کلاس مجزا تقسیم شود.

برای رسیدن به این منظور میتوانیم مثال بالا را به صورت 3 کلاس مختلف ایجاد کنیم :

1- Employee : که حاوی خاصیت‌ها است.

2- EmployeeDB : عملیات دیتابیسی نظیر درج رکورد و واکشی رکوردها از دیتابیس را انجام میدهد.

3- EmployeeReport : وظایف مربوط به ایجاد گزارش‌ها را انجام میدهد.

کد حاصل :

```
public class Employee
{
    public string EmployeeName { get; set; }
    public int EmployeeNo { get; set; }
}

public class EmployeeDB
{
    public void Insert(Employee e)
    {
        //Database Logic written here
    }
    public Employee Select()
    {
        //Database Logic written here
    }
}

public class EmployeeReport
{
    public void GenerateReport(Employee e)
```

```
    {
        //Set report formatting
    }
}
```

این روش برای متدها نیز صدق میکند به طوری که هر متد باید مسئولیت واحدی داشته باشد.
برای مثال قطعه کد زیر اصل SRP را نقض میکند :

```
//Method with multiple responsibilities – violating SRP
public void Insert(Employee e)
{
    string StrConnectionString = "";
    SqlConnection objCon = new SqlConnection(StrConnectionString);
    SqlParameter[] SomeParameters=null;//Create Parameter array from values
    SqlCommand objCommand = new SqlCommand("InertQuery", objCon);
    objCommand.Parameters.AddRange(SomeParameters);
    ObjCommand.ExecuteNonQuery();
}
```

این متد وظایف مختلفی را انجام میدهد مانند اتصال به دیتابیس ، ایجاد پارامترها برای مقادیر، ایجاد کوئری و در نهایت اجرای آن بر روی دیتابیس.
اما با توجه به اصل SRP میتوان آن را به صورت زیر بازنویسی کرد :

```
//Method with single responsibility – follow SRP
public void Insert(Employee e)
{
    SqlConnection objCon = GetConnection();
    SqlParameter[] SomeParameters=GetParameters();
    SqlCommand ObjCommand = GetCommand(objCon,"InertQuery",SomeParameters);
    ObjCommand.ExecuteNonQuery();
}

private SqlCommand GetCommand(SqlConnection objCon, string InsertQuery, SqlParameter[] SomeParameters)
{
    SqlCommand objCommand = new SqlCommand(InsertQuery, objCon);
    objCommand.Parameters.AddRange(SomeParameters);
    return objCommand;
}

private SqlParameter[] GetParaeters()
{
    //Create Paramter array from values
}

private SqlConnection GetConnection()
{
    string StrConnectionString = "";
    return new SqlConnection(StrConnectionString);
}
```

نظرات خوانندگان

نویسنده: حسن دهیاری
تاریخ: ۱۳۹۲/۰۷/۲۳ ۲۰:۲۷

با سلام.ضمن تشکر.یک سوال داشتم.آیا با توجه به اصل SRP نباید کلاس EmployeeDB تفکیک شود.چون که دو کار را انجام میدهد.ممنون میشم توضیح دهید.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۷/۲۳ ۲۳:۲۶

تنها دلیل تغییر کلی این کلاس در آینده، تغییر خاصیت‌های شیء کارمند است. بنابراین اصل تک مسئولیتی را نقض نمی‌کند. اگر این کلاس برای مثال دو Select داشت که یکی لیست کارمندان و دیگری لیست نقش‌های سیستم را بازگشت می‌داد، در این حالت تک مسئولیتی نقض می‌شد. ضمنا این نوع طراحی تحت عنوان الگوی مخزن یا لایه سرویس و امثال آن، یک طراحی پذیرفته شده و عمومی است. اگر قصد دارید که کوئری‌های خاص آن‌را طبقه بندی کنید می‌شود مثلا از Specification pattern استفاده کرد.

نویسنده: فراز
تاریخ: ۱۳۹۳/۰۵/۱۹ ۰:۴۹

با سلام ممنون از مقاله ای که گذاشتین من 3 سال هست تقریبا به صورت دست پا شکسته دنبال OOAD می‌گردم که در عمل توضیح داده باشد که شما این کار رو انجام دادین با سپاس فراوان

عنوان: **اصول طراحی شی گرا SOLID - #بخش دوم اصل OCP**
نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۰۶ ۲۰:۳۵
آدرس: www.dotnettips.info
برچسب‌ها: OOP, SOLID Principals, Object Oriented Design, Open Closed

در قسمت قبل در مورد اصل Single responsibility Principle یا به اختصار SRP صحبت شد. در این قسمت قصد داریم اصل دوم از اصول SOLID را مورد بررسی قرار دهیم.

O - OCP – Open Close Principle (2 اصل

فرض میکنیم که شما میخواهید یک طبقه بین طبقه‌ی اول و دوم خانه‌ی 2 طبقه‌ی خود اضافه کنید. فکرمیکنید امکان پذیر است؟

راه حل هایی که ممکن است به ذهن شما خطور کنند :

-1 زمانی که برای اولین بار در حال ساخت خانه هستید آن را 3 طبقه بسازید و طبقه‌ی وسط را خالی نگه دارید.اینطوری هر زمان که شما بخواهید میتوانید از آن استفاده کنید. به هر حال این هم یک راه حل است.

-2 خراب کردن طبقه دوم و ساخت دو طبقه‌ی جدید که خوب اصلا معقول نیست.

کد زیر را مشاهده کنید :

```
public class EmployeeDB
{
    public void Insert(Employee e)
    {
        //Database Logic written here
    }
    public Employee Select()
    {
        //Database Logic written here
    }
}
```

متد Select در کلاس EmployeeDB توسط کاربران و مدیر مورد استفاده قرار میگیرد. در این بین ممکن است مدیر نیاز داشته باشد تغییراتی را در آن انجام دهد. اگر مدیر این کار را برای برآورده کردن نیاز خود انجام دهد ،روی دیگر قسمت‌ها نیز تاثیر میگذارد، به علاوه ایجاد تغییرات در راه حل‌های تست شده‌ی موجود ممکن است موجب خطاهای غیر منتظره ای شود. چگونه ممکن است که رفتار یک برنامه تغییر کند بدون اینکه کد آن ویرایش شود؟ چگونه می‌توانیم بدون تغییر یک موجودیت نرم افزاری کارکرد آن را تغییر دهیم؟

اصل **OCP** میگوید : **"ماژول‌های نرم افزار باید برای تغییرات بسته و برای توسعه باز باشند."**

راه حل هایی که OCP را نقض نمیکنند : **1- استفاده از وراثت (inheritance):**

ایجاد یک کلاس جدید به نام EmployeeManagerDB که از کلاس EmployeeDB ارث بری کند و متد Select آن را جهت نیاز خود بازنویسی کند.

```
public class EmployeeDB
{
    public virtual Employee Select()
    {
        //Old Select Method
    }
}
public class EmployeeManagerDB : EmployeeDB
{
    public override Employee Select()
    {
        //Select method as per Manager
        //UI requirement
    }
}
```

این انتخاب خیلی خوبی است در صورتی که این تغییرات در زمان طراحی اولیه پیش بینی شده باشد و همکنون قابل استفاده باشند.

کد UI هم به شکل زیر خواهد بود :

```
//Normal Screen
EmployeeDB objEmpDb = new EmployeeDB();
Employee objEmp = objEmpDb.Select();

//Manager Screen
EmployeeDB objEmpDb = new EmployeeManagerDB();
Employee objEmp = objEmpDb.Select();
```

2- متدهای الحاقی (Extension Method):

اگر شما از NET 3.5. یا بالاتر از آن استفاده میکنید، دومین راه استفاده از متدهای الحاقی است که به شما اجازه میدهد بدون هیچ دست زدنی به نوع‌های موجود، متدهای جدیدی را به آنها اضافه کنید.

```
Public static class MyExtensionMethod{
    public static Employee managerSelect(this EmployeeDB employeeDB) {
//Select method as per Manager
    }
}

//Manager Screen
Employee objEmp = EmployeeDB.managerSelect();
```

البته ممکن است راه‌های دیگری هم برای رسیدن به این منظور وجود داشته باشد.

درقسمت‌های بعدی قانون‌های دیگر را بررسی خواهیم کرد.

عنوان: **اصول طراحی شی گرا SOLID - #بخش سوم اصل LSP**
نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۰۶ ۲۰:۴۰
آدرس: www.dotnettips.info
برچسب‌ها: OOP, SOLID Principals, Object Oriented Design, Liskov substitution

بخش‌های پیشین : اصول طراحی شی گرا SOLID - #بخش اول اصل SRP
اصول طراحی شی گرا SOLID - #بخش دوم اصل OCP

اصل 3) L – LSP – Liskov substitution principle

اصل **LSP** میگوید : **"زیر کلاس‌ها باید بتوانند جایگزین نوع پایه‌ی خود باشند".**

مقایسه با جهان واقعی :

شغل یک پدر تجارت املاک است درحالی که پسرش دوست دارد فوتبالیست شود.
یک پسر هیچگاه نمیتواند جایگزین پدرش شود، با اینکه که آنها به یک سلسله مراتب خانوادگی تعلق دارند.
در یک مثال عمومی‌تر بررسی میکنیم :
به طور معمول زمانی که ما در مورد اشکال هندسی صحبت میکنیم ، مستطیل را یک کلاس پایه برای مربع میدانیم. به کد زیر توجه کنید :

```
public class Rectangle
{
    public int Width { get; set; }
    public int Height { get; set; }
}

public class Square:Rectangle
{
    //codes specific to
    //square will be added
}
```

و میتوان گفت :

```
Rectangle o = new Rectangle();
o.Width = 5;
o.Height = 6;
```

بسیار خوب، اما با توجه به LSP باید قادر باشیم مستطیل را با مربع جایگزین کنیم. سعی میکنیم این کار را انجام دهیم :

```
Rectangle o = new Square();
o.Width = 5;
o.Height = 6;
```

موضوع چیست؟ مگر مربع می‌تواند طول و عرض نا برابر داشته باشد؟! امکان ندارد.
خوب این به چه معنی است؟ به این معنی که ما نمیتوانیم کلاس پایه را با کلاس مشتق شده جایگزین کنیم و باز هم این معنی را میدهد که ما داریم اصل LSP را نقض میکنیم.
آیا ما میتوانیم طول و عرض را در کلاس Square طبق کد زیر دوباره نویسی کنیم؟

```
public class Square : Rectangle
{
    public override int Width
    {
        get{return base.Width;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
    public override int Height
    {
        get{return base.Height;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}
```

باز هم اصل LSP نقض میشود چون ما داریم رفتار خاصیت‌های طول و عرض در کلاس مشتق شده را تغییر میدهیم. ولی با توجه به کد بالا یک مستطیل نمیتواند طول و عرض برابر داشته باشد چون در صورت برابری دیگر مستطیل نیست.

اما راه حل چیست؟

یک کلاس انتزاعی (abstract) را به شکل زیر ایجاد و سپس دوکلاس Square و Rectangle را از آن مشتق میکنیم :

```
public abstract class Shape
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }
}
```

همکنون ما دو کلاس مستقل از هم داریم. یکی Square و دیگری Rectangle که هر دو از کلاس Shape مشتق شده اند.

حالا میتوانیم بنویسیم :

```
Shape o = new Rectangle();
o.Width = 5;
o.Height = 6;

Shape o = new Square();
o.Width = 5; //both height and width become 5
o.Height = 6; //both height and width become 6
```

زمانی که ما در مورد اشکال هندسی صحبت میکنیم ، هیچ قاعده‌ی خاصی جهت اندازه‌ی طول و عرض نیست. ممکن است برابر باشند یا نباشند.

در قسمت بعدی اصل ISP را مورد بررسی قرار خواهیم داد.

نظرات خوانندگان

نویسنده: فدورا
تاریخ: ۱۳۹۲/۰۷/۰۹ ۱۰:۳۸

سلام.
خیلی ممنون از بابت مقالات آموزشی خوبتون.
فقط سوالی برای من تو این بخش سوم پیش اومد و اون هم اینکه بعد از تعریف کلاس abstract تعریف کلاس‌های rectangle و square به چه شکل شد؟ لطفا کد اون کلاس‌ها رو هم اضافه کنید.
با تشکر

نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۰۹ ۱۳:۴۸

ممنون.
کلاس های Rectangle و Square هر دو به همون شکل باقی میمونند با این تفاوت که هر دو از کلاس Shape مشتق شده اند و میتوانند خاصیت‌های Width و Height را طبق نیاز خود دوباره نویسی کنند (override). کلاس Restangle:

```
public class Rectangle : Shape
{
    //شما میتوانید خاصیت‌ها طول و عرض در کلاس پایه را در صورت نیاز دوباره نویسی کنید
}
```

کلاس Square :

```
public class Square : Shape
{
    //دوباره نویسی کردن خاصیت‌های طول و عرض در کلاس پایه جهت برابر کردن طول و عرض مربع
    public override int Width
    {
        get{return base.Width;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
    public override int Height
    {
        get{return base.Height;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}
```

که با توجه به کدهای بالا ، کلاسهای مشتق شده‌ی Square و Restangle میتوانند جایگزین کلاس پایه خود یعنی Shape شوند :

```
Shape o = new Rectangle();
o.Width = 5;
o.Height = 6;

Shape o = new Square();
o.Width = 5; //طول و عرض هر دو برابر 5 میشوند
o.Height = 6; //عرض و طول هر دو برابر 6 میشوند
```

عنوان: **اصول طراحی شی گرا SOLID - #بخش چهارم اصل ISP**
نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۰۶ ۲۰:۴۵
آدرس: www.dotnettips.info
برچسب‌ها: OOP, SOLID Principals, Object Oriented Design, Interface Segregation

بخش‌های پیشین : اصول طراحی شی گرا SOLID - #بخش اول اصل SRP
اصول طراحی شی گرا SOLID - #بخش دوم اصل OCP اصول طراحی شی گرا SOLID - #بخش سوم اصل LSP

اصل 4) I – ISP- Interface Segregation principle

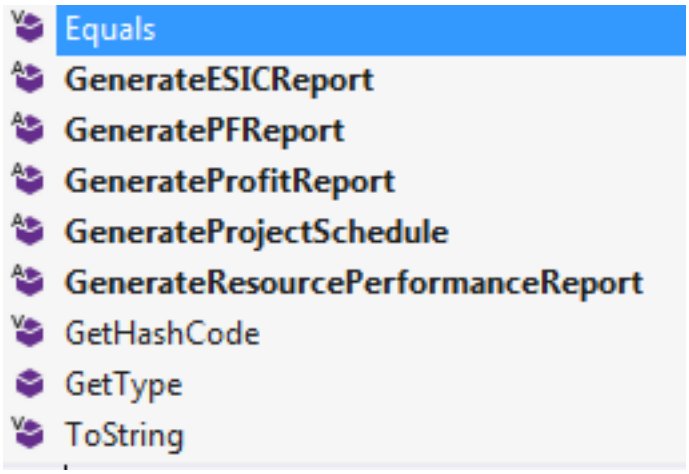
مقایسه با دنیای واقعی:

بیایید فکر کنیم شما یک کامپیوتر دسکتاپ جدید خریداری کرده اید.شما یک زوج پورت USP، چند پورت سریال، یک پورت VGA و ... را پیدا میکنید. اگر شما بدنه‌ی کیس خود را باز کنید، تعدادی اسلات بر روی مادربرد خود که برای اتصال قطعات مختلف با یکدیگر هستند را مشاهده خواهید کرد که عمدتا مهندسان سخت افزار در زمان منتاژ از آنها استفاده میکنند.
این اسلات‌ها تا زمانی که بدنه کیس را باز نکنید قابل مشاهده نخواهند بود. به طور خلاصه تنها رابطه‌های مورد نیازی که برای شما ساخته شده اند، قابل مشاهده خواهند بود.
و فرض کنید شما به یک توپ فوتبال نیاز دارید. به یک فروشگاه وسایل ورزشی میروید و فروشنده شروع به نشان دادن انواع توپ ها، کفش ها،لباس و گرم کن‌های ورزشی،لباس شنا و کاراته ،زانوبند و ... کرده است. در نهایت ممکن است شما چیزی را خریداری کنید که اصلا مورد نیازتان نبوده است یا حتی ممکن است فراموش کنیم که ما چرا اینجا هستیم.! **ارائه مشکل:**
ما میخواهیم یک سیستم مدیریت گزارشات را توسعه بدهیم. اولین کاری که انجام می‌دهیم ایجاد یک لایه منطقی که توسط سه UI مختلف دیگر مورد استفاده قرار میگیرد.
1- EmployeeUI : نمایش گزارش‌های مربوط با کارمند وارد شده‌ی جاری در سایت.
2- ManagerUI : نمایش گزارش‌های مربوط به خود و تیمی که به او تعلق دارد.
3- AdminUI : نمایش گزارش‌های مربوط به کارمندان، مربوط به تیم و مربوط به شرکت مانند گزارش سود و زیان و ...

```
public interface IReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();

    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();

    void GenerateProfitReport();
}
```

```
public class ReportBAL : IReportBAL
{
    public void GeneratePFReport()
    {/*..............*/}

    public void GenerateESICReport()
    {/*..............*/}

    public void GenerateResourcePerformanceReport()
    {/*..............*/}

    public void GenerateProjectSchedule()
    {/*..............*/}

    public void GenerateProfitReport()
    {/*..............*/}
}
public class EmployeeUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}
public class ManagerUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport ();
        objBal.GenerateProjectSchedule ();
    }
}
public class AdminUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport();
        objBal.GenerateProjectSchedule();
        objBal.GenerateProfitReport();
    }
}
```

حال زمانی که توسعه دهنده در هر UI عبارت objBal را تایپ کند، در لیست بازشوی آن (intellisense) به صورت زیر نمایش داده خواهد شد :

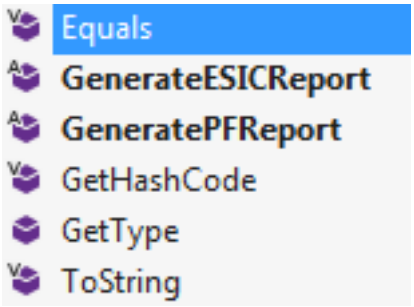


اما مشکل چیست؟ مشکل این است شخصی که روی لایه‌ی EmployeeUI کار میکند به تمام متدهاها به خوبی دسترسی دارد و این متدهای غیرضروری ممکن است باعث سردرگمی او شود.
این مشکل به این دلیل اتفاق می‌افتد که وقتی کلاسی یک واسط را پیاده سازی می‌کند، باید همه متدهای آن را نیز پیاده سازی کند. حالا اگر خیلی از این متدها توسط این کلاس استفاده نشود، می‌گوییم که این کلاس از مشکل واسط چاق رنج می‌برد.
اصل **ISP** میگوید : **" کلاینت‌ها نباید وابسته به متدهایی باشند که آنها را پیاده سازی نمی‌کنند. "**
برای رسیدن به این امر در مثال بالا باید آن واسط را به واسطهای کوچکتر تقسیم کرد. این تقسیم بندی باید بر اساس استفاده کنندگان از واسطها صورت گیرد.
ویرایش مثال بالا با در نظر گرفتن اصل ISP :

```
public interface IEmployeeReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();
}
public interface IManagerReportBAL : IEmployeeReportBAL
{
    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();
}
public interface IAdminReportBAL : IManagerReportBAL
{
    void GenerateProfitReport();
}
public class ReportBAL : IAdminReportBAL
{
    public void GeneratePFReport()
    {/*..............*/}

    public void GenerateESICReport()
    {/*..............*/}

    public void GenerateResourcePerformanceReport()
    {/*..............*/}

    public void GenerateProjectSchedule()
    {/*..............*/}

    public void GenerateProfitReport()
    {/*..............*/}
}
```

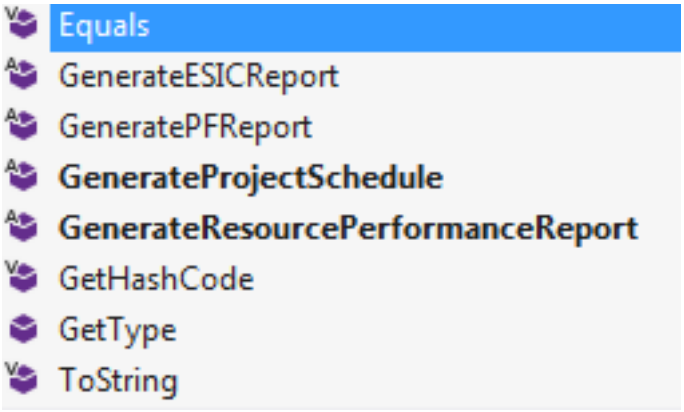
وضعیت نمایش :

```
public class EmployeeUI
{
    public void DisplayUI()
    {
        IEmployeeReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}
```

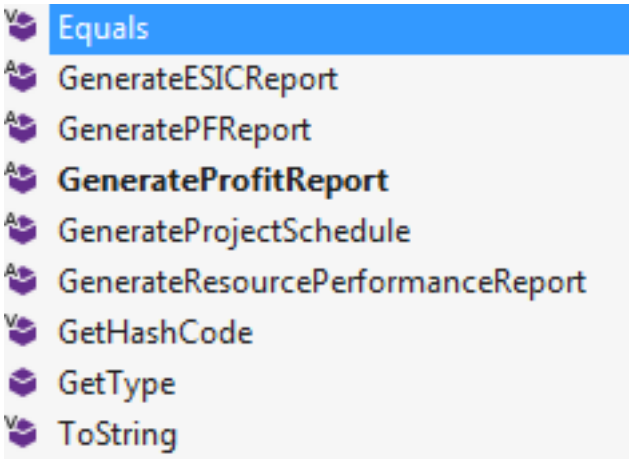



```
public class ManagerUI
{
    public void DisplayUI()
```

```
    {
        IManagerReportBAL  objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport ();
        objBal.GenerateProjectSchedule ();
    }
}
```




```
public class AdminUI
{
    public void DisplayUI()
    {
        IAdminReportBAL  objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport();
        objBal.GenerateProjectSchedule();
        objBal.GenerateProfitReport();
    }
}
```




```
DIP   :
```

در قسمت بعدی آخرین اصل

را بررسی خواهیم کرد

عنوان: **اصول طراحی شی گرا SOLID - #بخش پنجم اصل DIP**
نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۰۶ ۲۰:۵۰
آدرس: www.dotnettips.info
گروه‌ها: OOP, SOLID Principals, Dependency Inversion, Object Oriented Design

بخش‌های پیشین : اصول طراحی شی گرا SOLID - #بخش اول اصل SRP اصول طراحی شی گرا SOLID - #بخش دوم اصل OCP
اصول طراحی شی گرا SOLID - #بخش سوم اصل LSP
اصول طراحی شی گرا SOLID - #بخش چهارم اصل ISP

اصل 5) D – DIP– Dependency Inversion principle

مقایسه با دنیای واقعی:

همان مثال کامپیوتر را دوباره در نظر بگیرید.این کامپیوتر دارای قطعات مختلفی مانند RAM ، هارد دیسک، CD ROM و ... است که هر کدام به صورت مستقل به مادربرد متصل شده اند. این به این معنی است که اگر قسمتی از کار بیفتد میتوان آن را با یک قطعه‌ی جدید به آسانی تعویض کرد . حالا فقط تصور کنید که تمامی قطعات شدیداً به یکدیگر متصل شده اند آنوقت دیگر نمیتوانستیم قطعه ای را از مادربرد برداریم و به همین خاطر اگر مثلا RAM از کار بیفتد ، باید یک مادربرد جدید خریداری کنید که برای شما گران تمام می‌شود.

به مثال زیر توجه کنید :

```
public class CustomerBAL
{
    public void Insert(Customer c)
    {
        try
        {
            //Insert logic
        }
        catch (Exception e)
        {
            FileLogger f = new FileLogger();
            f.LogError(e);
        }
    }
}

public class FileLogger
{
    public void LogError(Exception e)
    {
        //Log Error in a physical file
    }
}
```

در کد بالا کلاسCustomerBAL مستقیما به کلاس FileLogger وابسته است که استثناءهای رخ داده را بر روی یک فایل فیزیکی لاگ میکند. حالا فرض کنید که چند روز بعد مدیریت تصمیم میگیرد که از این به بعد استثناءها بر روی یک Event Viewer لاگ شوند.
اکنون چه میکنید؟ با تغییر کدها ممکن است با خطاهای زیادی روبرو شوید(درصورت تعداد بالای کلاسهایی که از کلاس FileLogger استفاده میکنند و فقط تعداد محدودی از آنها نیاز دارند که بر روی Event Viewer لاگ کنند.) **DIP** به ما میگوید : **"ماژول‌های سطح بالا نباید به ماژولهای سطح پایین وابسته باشند، هر دو باید به انتزاعات وابسته باشند. انتزاعات نباید وابسته به جزئیات باشند، بلکه جزئیات باید وابسته به انتزاعات باشند. "**.

در طراحی ساخت یافته، ماژولهای سطح بالا به ماژولهای سطح پایین وابسته بودند. این مسئله دو مشکل ایجاد می‌کرد:
1- ماژول‌های سطح بالا (سیاست گذار) به ماژول‌های سطح پایین (مجری) وابسته هستند. در نتیجه هر تغییری در ماژول‌های سطح پایین ممکن است باعث اشکال در ماژول‌های سطح بالا گردد.
2- استفاده مجدد از ماژول‌های سطح بالا در جاهای دیگر مشکل است، زیرا وابستگی مستقیم به ماژول‌های سطح پایین دارند. **راه حل با توجه به اصل DIP :**

```
public interface ILogger
{
    void LogError(Exception e);
}

public class FileLogger:ILogger
```

```
{
    public void LogError(Exception e)
    {
        //Log Error in a physical file
    }
}
public class EventViewerLogger : ILogger
{
    public void LogError(Exception e)
    {
        //Log Error in a Event Viewer
    }
}
public class CustomerBAL
{
    private ILogger _objLogger;
    public CustomerBAL(ILogger objLogger)
    {
        _objLogger = objLogger;
    }

    public void Insert(Customer c)
    {
        try
        {
            //Insert logic
        }
        catch (Exception e)
        {
            _objLogger.LogError(e);
        }
    }
}
```

در اینجا وابستگی‌های کلاس CustomerBAL از طریق سازنده آن در اختیارش قرار گرفته است. یک اینترفیس ILogger تعریف شده‌است به همراه دو پیاده سازی مختلف از آن مانند FileLogger و EventViewerLogger.
یکی از انواع فراخوانی آن نیز می‌تواند به شکل زیر باشد:

```
var customerBAL = new CustomerBAL (new EventViewerLogger());
customerBAL.LogError();
```

اطلاعات بیشتر در دوره آموزشی " بررسی مفاهیم معکوس سازی وابستگی‌ها و ابزارهای مرتبط با آن ".

نظرات خوانندگان

نویسنده: سعید سلیمانی فر
تاریخ: ۱۳۹۲/۰۷/۰۹ ۹:۳۱

خیلی مطلب خوبی بود! لذت بردیم متشکرم :)

نویسنده: بهزاد علی محمدزاده
تاریخ: ۱۳۹۲/۰۷/۱۹ ۱۶:۲۲

اقای طاهری با تشکر . امکان داره منبع رو معرفی کنید . به دنبال یه کتاب یا منبع آموزشی خوب در این زمینه هستم که البته نمونه‌ها رو با C# انجام داده باشه .

نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۱۹ ۱۷:۴۱

چند مقاله ای که من اونها رو مطالعه کردم : اصول طراحی SOLID SOLIDify your software design concepts through SOLID Articles by Christian Vos Object Oriented Design Principles SOLID by example SOLID Agile Development Articles Of SOLID SOLID Principles in C# - An Overview

عنوان: **Design Pattern: Factory**
نویسنده: مهسا حسن کاشی
تاریخ: ۱۳۹۲/۱۱/۲۶ ۲۲:۱۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Design patterns, SOLID, Object Oriented Design, ServiceFactory, Patterns, Software architecture

الگوهای طراحی، سندها و راه حلهای از پیش تعریف شده و تست شده‌ای برای مسائل و مشکلات روزمره‌ی برنامه نویسی می‌باشند که هر روزه ما را درگیر خودشان می‌کنند. هر چقدر مقیاس پروژه وسیعتر و تعداد کلاسها و اشیاء بزرگتر باشند، درگیری برنامه نویس و چالش برای مرتب سازی و خوانایی برنامه و همچنین بالا بردن کارآیی و امنیت افزون‌تر می‌شود. از همین رو استفاده از ساختارهایی تست شده برای سناریوهای یکسان، امری واجب تلقی می‌شود.

الگوهای طراحی از لحاظ سناریو، به سه گروه عمده تقسیم می‌شوند:

-1 تکوینی: هر چقدر تعداد کلاسها در یک پروژه زیاد شود، به مراتب تعداد اشیاء ساخته شده از آن نیز افزوده شده و پیچیدگی و درگیری نیز افزایش می‌یابد. راه حل‌هایی از این دست، تمرکز بر روی مرکزیت دادن به کلاسها با استفاده از رابطها و کپسوله نمودن (پنهان سازی) اشیاء دارد.

-2 ساختاری: گاهی در پروژه‌ها پیش می‌آید که می‌خواهیم ارتباط بین دو کلاس را تغییر دهیم. از این رو امکان از هم پاشی اجزایِ دیگر پروژه پیش می‌آید. راه حلهای ساختاری، سعی در حفظ انسجام پروژه در برابر این دست از تغییرات را دارند.

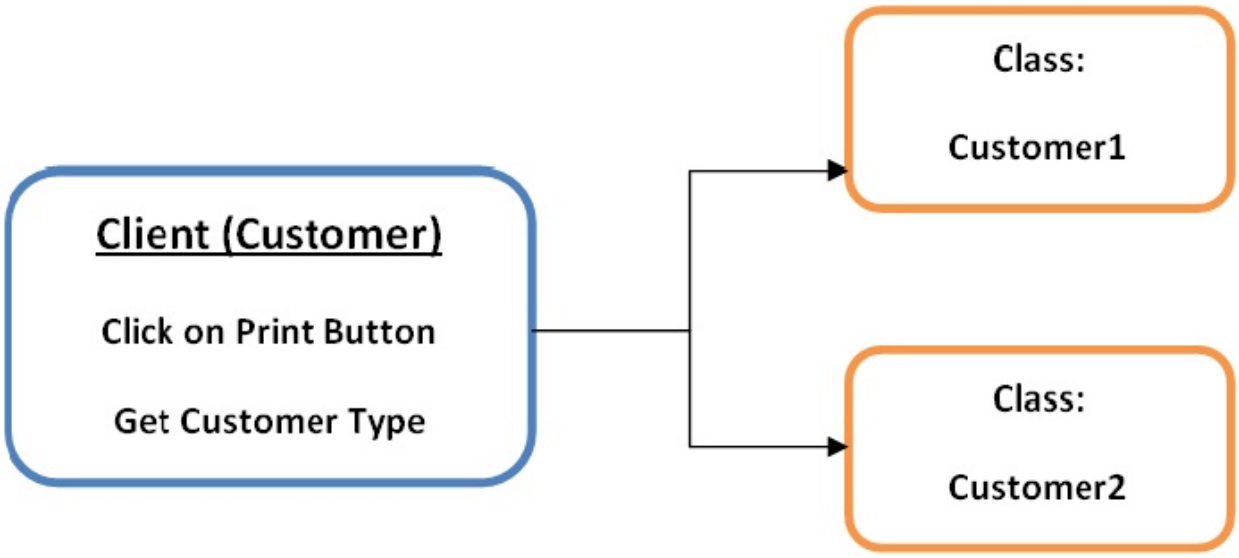
-3 رفتاری: گاهی بنا به مصلحت و نیاز مشتری، رفتار یک کلاس می‌بایستی تغییر نماید. مثلا چنانچه کلاسی برای ارائه صورتحساب داریم و در آن میزان مالیات 30% لحاظ شده است، حال این درصد باید به عددی دیگر تغییر کند و یا پایگاه داده به جای مشاهده‌ی تعدادِ معدودی گره از درخت، حال می‌بایست تمام گره‌ها را ارائه نماید.

الگوی فکتوری:

الگوی فکتوری در دستهء اول قرار می‌گیرد. من در اینجا به نمونه‌ای از مشکلاتی که این الگو حل می‌نماید، اشاره می‌کنم:

فرض کنید یک شرکت بزرگ قصد دارد تا جزییات کامل خرید هر مشتری را با زدن دکمه چاپ ارسال نماید. چنین شرکت بزرگی بر اساس سیاستهای داخلی، بر حسب میزان خرید، مشتریان را به چند گروه مشتری معمولی و مشتری ممتاز تقسیم می‌نماید. در نتیجه نمایش جزییات برای آنها با احتساب میزان تخفیف و به عنوان مثال تعداد فیلدهایی که برای آنها در نظر گرفته شده است، تفاوت دارد. بنابراین برای هر نوع مشتری یک کلاس وجود دارد.

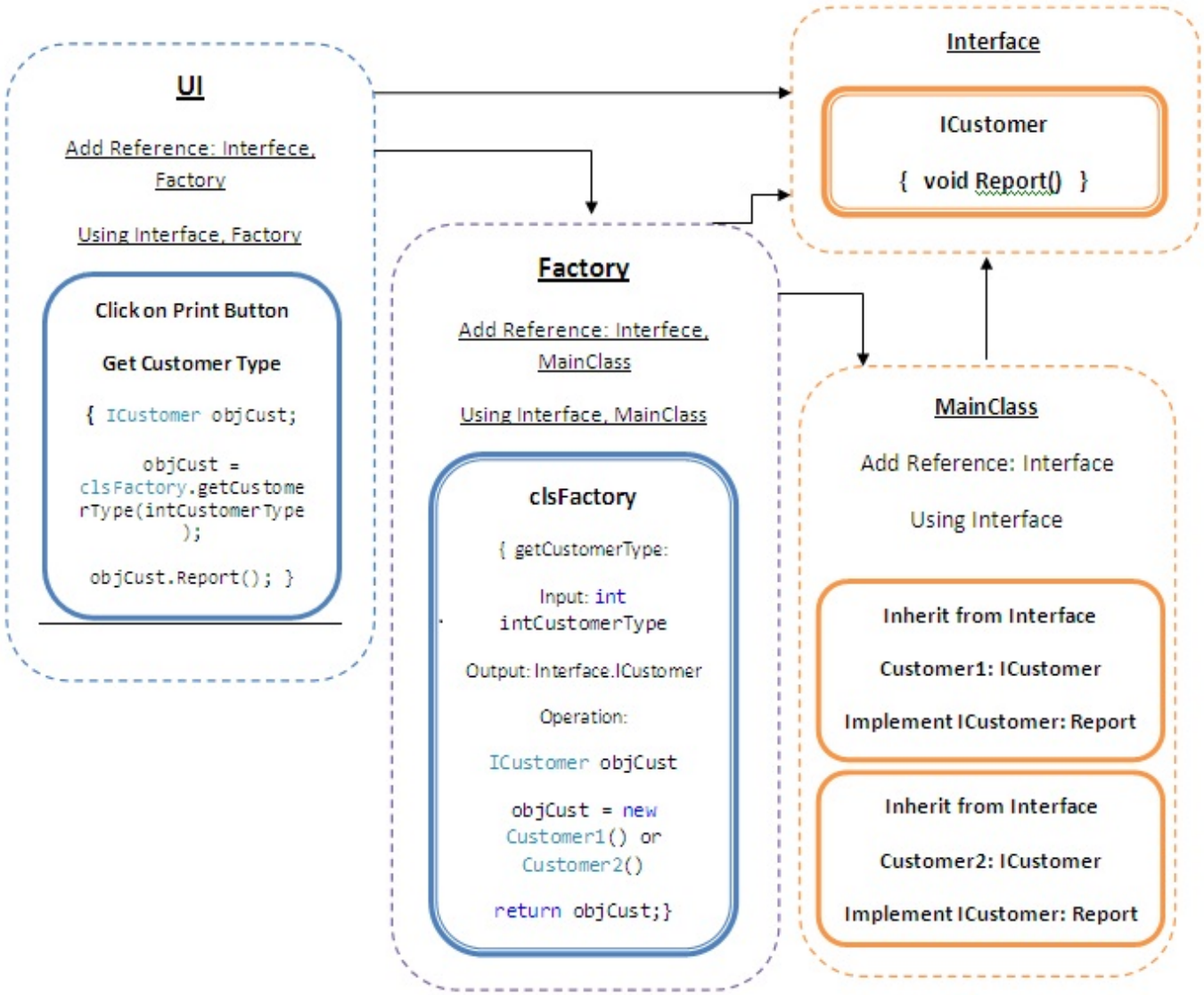
یک راه این است که با کلیک روی دکمه‌ی چاپ، نوع مشتری تشخیص داده شود و به ازای نوع مشتری، یک شیء از کلاس مشخص شده برای همان نوع ساخته شود.




```
// Get Customer Type from Customer click on Print Button
            int customerType = 0;

            // Create Object without instantiation
            object obj;


            //Instantiate obj according to customer Type
            if (customerType == 1)
            {
                obj = new Customer1();
            }
            else if (customerType == 2)
            {
                obj = new Customer2();
            }
            // Problem:
            //          1: Scattered New Keywords
            //          2: Client side is aware of Customer Type
```

همانگونه که مشاهده می‌نمایید در این سبک کدنویسی غیرحرفه‌ای، مشکلاتی مشهود است که قابل اغماض نیستند. در ابتدا سمت کلاینت دسترسی مستقیم به کلاسها دارد و همانگونه که در شکل بالا قابل مشاهده است کلاینت مستقیما به کلاس وصل است. مشکل دوم عدم پنهان سازی کلاس از دید مشتری است.

راه حل: این مشکل با استفاده از الگوی فکتوری قابل حل است. با استناد به الگوی فکتوری، کلاینت تنها به کلاس فکتوری و یک اینترفیس دسترسی دارد و کلاسهای فکتوری و اینترفیس، حق دسترسی به کلاسهای اصلی برنامه را دارند.



گام نخست: در ابتدا یک class library به نام Interface ساخته و در آن یک کلاس با نام ICustomer می سازیم که متد Report() را معرفی می‌نماید.

Interface//

```
namespace Interface
{
    public interface ICustomer
    {
        void Report();
    }
}
```

گام دوم: یک class library به نام MainClass ساخته و با Add Reference کلاس Interface را اضافه نموده، در آن دو کلاس با نام Customer1, Customer2 می‌سازیم و using Interface را Import می‌نماییم. هر دو کلاس از ICustomer ارث می‌برند و سپس متد Report() را در هر دو کلاس Implement می‌نماییم.

```
// Customer1
using System;
```

```
using Interface;

namespace MainClass
{
    public class Customer1 : ICustomer
    {
        public void Report()
        {
            Console.WriteLine("این گزارش مخصوص مشتری نوع اول است");
        }
    }
}

//Customer2
using System;
using Interface;

namespace MainClass
{
   public class Customer2 : ICustomer
    {
        public void Report()
        {
            Console.WriteLine("این گزارش مخصوص مشتری نوع دوم است");
        }
    }
}
```

گام سوم: یک class library به نام FactoryClass ساخته و با Add Reference کلاس Interface, MainClass را اضافه نموده، در آن یک کلاس با نام clsFactory می سازیم و using Interface, using MainClass را Import می‌نماییم. پس از آن یک متد با نام getCustomerType ساخته که ورودی آن نوع مشتری از نوع int است و خروجی آن از نوع Interface-ICustomer و بر اساس کد نوع مشتری object را از کلاس Customer1 و یا Customer2 می‌سازیم و آن را return می نماییم.

```
//Factory
using System;
using Interface;
using MainClass;

namespace FactoryClass
{
    public class clsFactory
    {
        static public ICustomer getCustomerType(int intCustomerType)
        {
            ICustomer objCust;
            if (intCustomerType == 1)
            {
                objCust = new Customer1();
            }
            else if (intCustomerType == 2)
            {
                objCust = new Customer2();
            }
            else
            {
                return null;
            }
            return objCust;
        }
    }
}
```

گام چهارم (آخر): در قسمت UI Client، کد نوع مشتری را از کاربر دریافت کرده و با Add Reference کلاس Interface, FactoryClass را اضافه نموده (دقت نمایید هیچ دسترسی به کلاس‌های اصلی وجود ندارد)، و using Interface, using FactoryClass را Import می‌نماییم. از clsFactory تابع getCustomerType را فراخوانی نموده (به آن کد نوع مشتری را پاس می‌دهیم) و خروجی آن را که از نوع اینترفیس است به یک object از نوع ICustomer نسبت می‌دهیم. سپس از این object متد Report را فراخوانی می‌نماییم. همانطور که از شکل و کدها مشخص است، هیچ رابطه ای بین UI(Client) و کلاسهای اصلی برقرار نیست.

```
//UI (Client)
using System;
using FactoryClass;
using Interface;

namespace DesignPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            int intCustomerType = 0;
            ICustomer objCust;
            Console.WriteLine("نوع مشتری را وارد نمایید");
            intCustomerType = Convert.ToInt16(Console.ReadLine());
            objCust = clsFactory.getCustomerType(intCustomerType);
            objCust.Report();
            Console.ReadLine();
        }
    }
}
```