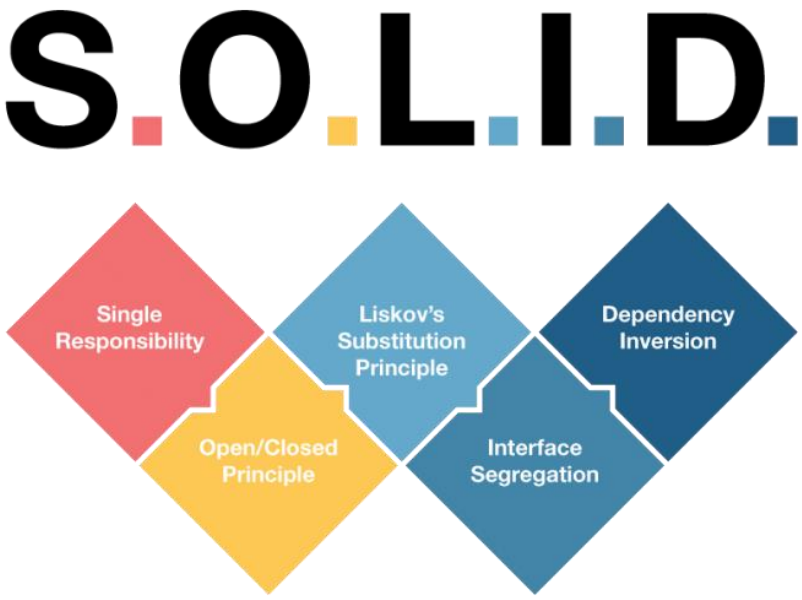
باسمه تعالی

اصول Solid:



اصول Solid از اصول مهم در توسعه نرم افزار است. Solid شامل 5 اصل است که اگر در برنامه نویسی لحاظ شود، باعث بالا رفتن کیفیت کد و ساده شدن دولوپ نرم افزار می شود. این اصول به ما کمک می کنند که کدی تمیز، قابل توسعه و قابل تست بنویسیم.

واژهٔ **SOLID** برگرفته شده از پنج اصلِ زیر است:

1. S - **S**ingle-responsibility Principle
2. O - **O**pen-closed Principle
3. L - **L**iskov Substitution Principle
4. I - **I**nterface Segregation Principle
5. D - **D**ependency Inversion Principle

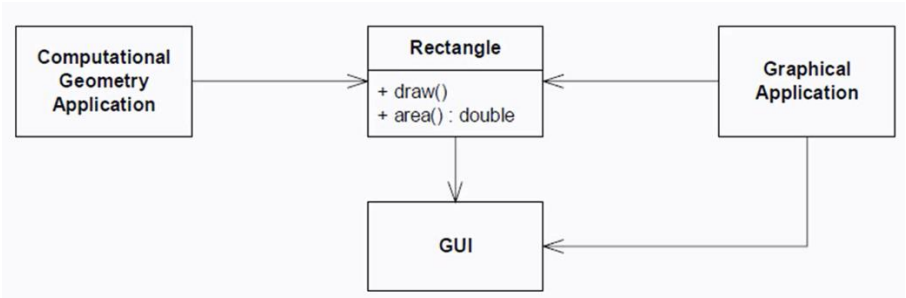
اصل 1 – Single Responsibility Principle یا اصل تک وظیفه ای (SRP):

هر کلاس باید فقط و فقط یک وظیفه و مسئولیت مشخص داشته باشد (و نه بیشتر) و فقط یک دلیل برای تغییر وجود داشته باشد. مثلا کلاسی که مسئول کار با دیتابیس است، نباید مسئول مدیریت لایه UI هم باشد!

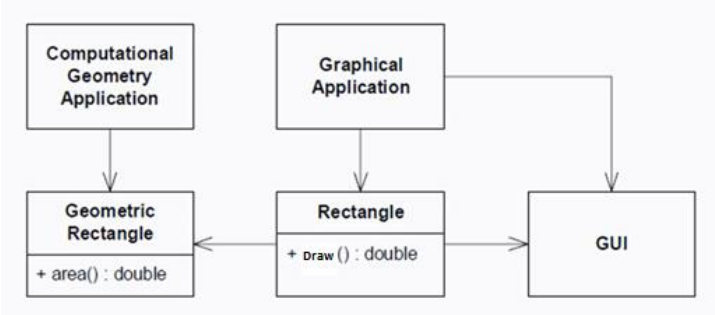
بر خلاف چاقو های سویسی چند کاره که هر تغییری در هر بخش نیاز به تغییر در بخش های دیگر دارد، باید بخش های مختلف را از هم جدا کنیم تا تغییر هر بخش، باعث تغییر در بخش های دیگر نشود و شسته رفته و تمیز باشد. در این ساختار هر بخش یک وظیفه مشخص دارد و متخصص همان وظیفه است.

این اصل می گوید، یک کلاس یا ماژول فقط باید یک دلیل برای تغییر داشته باشد و فقط یک وظیفه داشته باشد و نباید یک کلاس یا ماژول همه فن حریف داشته باشیم. در حقیقت، این اصل ما را به سمت زیر سیستم سازی جزئی هدایت می کند! باید بخش های مختلف برای انجام کارهای غیر مرتبط، از اشیای کلاسهای دیگر مرتبط با همان کار استفاده کنند! مثلا اگر خون داخل قلب باید تصفیه شود، از طریق سیاهرگها به سمت کلیه ها هدایت می شود و کلیه مسئول جدا کردن بخش های لازم خون و بخش های زاید خون از هم است!

سوال: اگر قلب، بخشی از کار تصفیه خون را انجام می داد، با از کار افتادنش، چند جای بدن از کار می افتاد؟ برای ساخت قلب مصنوعی باید چند مشکل دیگر مثل تصفیه خون را نیز پیاده می کردیم و این کار سخت یا نشدنی بود و حداقل تا این لحظه، قلب مصنوعی بوجود نیامده بود چون تا این لحظه کلیه مصنوعی تولید نشده است!
در مثال زیر، کلاس مستطیل هم مساحت را حساب می کند و هم مستطیل را رسم می کند. در صورتیکه باید این دو وظیفه را از هم تفکیک کرد تا یک کلاس مخصوص رسم مستطیل باشد و یک کلاس هم مساحت و محیط و مسایل دیگر آنرا محاسبه کند.



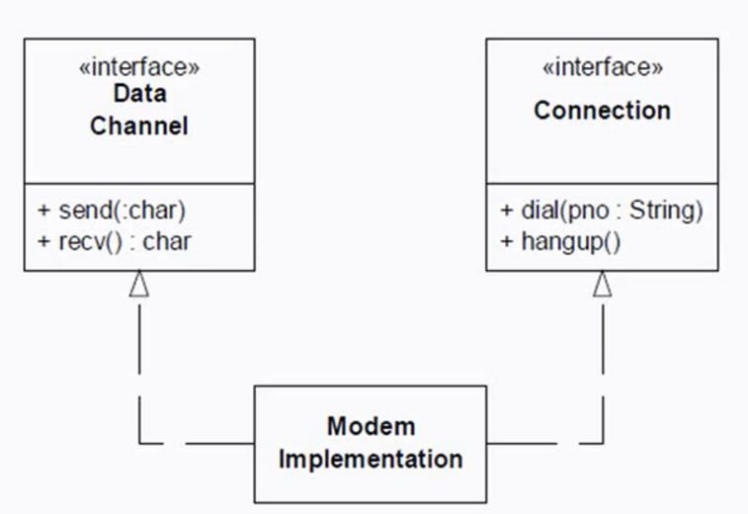
باید تبدیل به ساختار زیر شود:



در مثال دوم این اینترفیس مودم را ببینید:

```
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```

در این اینترفیس اصل تک وظیفه ای رعایت نشده است و می توان آنرا به دو اینترفیس شکست تا اتصال و قطع شدن ارتباط را به یکی و ارسال و دریافت اطلاعات را به دومی بسپاریم.



در مثال کد:

```
namespace SingleResponsibilityPrincipleDemo
{
    public class Customer
    {
        public void Add()
        {
            try
            {
                // Database code goes here
            }
            catch (Exception ex)
            {
                System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());
            }
        }
    }
}
```

اینکه لاگ خطای ارتباط با دیتابیس در همین کلاس پیاده سازی شود اشتباه است و اگر تغییری در مکانیسم لاگ شما اعمال شود، باید مجبور شوید این کلاس کاستومر را تغییر دهید و احتمالا نشر تغییر در تمام کلاسهایی که لاگ را در فایل ذخیره می کنند اعمال شود. برای جلوگیری از این کار می بایست کلاسی مخصوص لاگ کردن بوجود بیاورید و جزئیات لاگ شدن را در آن پیاده کنید و فقط در بقیه کلاسها متد مرتبط با لاگ کردن آنرا صدا بزنید. با این کار اگر مکانیسم لاک تغییر کند، شمای بیرونی آن که به صورت تابع در بقیه کلاسها صدا زده شده تغییر نمی کند و به این صورت تغییر فقط در کلاس لاگ اعمال می شود.

```
namespace SingleResponsibilityPrincipleDemo
{
    public class Logger
    {
        public void LogEvent(Exception ex)
        {
            System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());
        }
    }
}
```

```
namespace SingleResponsibilityPrincipleDemo
{
    public class Customer
    {
        Logger _logger = new Logger();
        public void Add()
        {
            try
            {
                // Database code goes here
            }
            catch (Exception ex)
            {
                _logger.LogEvent(ex);
            }
        }
```

پس کلاسها و متدهای مختص یک کار، داده جنرال را می گیرند و آنرا در درون خود هندل می کنند و به این روش، شمای بیرونی آنها تغییر نمی کند و دستورات درونی آنها در صورت نیاز یک بار تغییر می کند و این تغییرات به بیرون از آن سرایت نمی کند!

در مثال دیگری، کلاسی با 3 متد برای بررسی سطح اکسیژن آب وجود داشت که یکی از این متدها، سطح اکسیژن آب را از ورودی می خواند و میزان آنرا محاسبه می کرد و دیگری میزان اکسیژن آب را اندازه می گرفت و سومی نیز نتیجه چک کردن سطح اکسیژن را پرینت می کرد. این کلاس را به 3 کلاس شکاندیم. قبلا کلاس اکسیژن میتر می توانست به خاطر تفاوت در سخت افزار خواندن اطلاعات اکسیژن آب یا توسط محاسبه میزان اکسیژن آب و یا نحوه پرینت خروجی تغییر کند و 3 دلیل برای تغییر داشت.

```
namespace SingleResponsibilityPrincipleDemo
{
    public class OxygenMeter
    {
        public double OxygenSaturation { get; set; }

        public void ReadOxygenLevel()
        {
            using (MeterStream ms = new MeterStream("O2"))
            {
                int raw = ms.ReadByte();
                OxygenSaturation = (double)raw / 255 * 100;
            }
        }

        public bool OxygenLow()
        {
            return OxygenSaturation <= 75;
        }

        public void ShowLowOxygenAlert()
        {
            Console.WriteLine("Oxygen low ({0:F1}%)", OxygenSaturation);
        }
    }
}
```

اینک که به 3 کلاس مختلف شکسته است، هر کلاس فقط یک دلیل برای تغییر دارند. حالا نتیجه شکستن هر سه فرآیند را در سه کلاس زیر می بینند!

کلاس تفکیک شده اکسیژن میتر:

```
namespace SingleResponsibilityPrincipleDemo
{
    public class OxygenMeter
    {
        public double OxygenSaturation { get; set; }

        public void ReadOxygenLevel()
        {
            using (MeterStream ms = new MeterStream("O2"))
            {
                int raw = ms.ReadByte();
                OxygenSaturation = (double)raw / 255 * 100;
            }
        }
    }
}
```

کلاس چک کردن سطح اکسیژن:

```
namespace SingleResponsibilityPrincipleDemo
{
    public class OxygenChecker
    {
        public bool OxygenLow(OxygenMeter _oxygenMeter)
        {
            return _oxygenMeter.OxygenSaturation <= 75;
        }
    }
}
```

کلاس پرینت نتیجه بررسی:

```
namespace SingleResponsibilityPrincipleDemo
{
    public class OxygenAlert
    {
        public void ShowLowOxygenAlert(OxygenMeter _oxygenMeter)
        {
            Console.WriteLine("Oxygen low ({0:F1}%)", _oxygenMeter.OxygenSaturation);
        }
    }
}
```

پس برای پیاده‌سازیSRP ، می‌توانید از کلاس‌های ساختاری (Structs) و کلاس‌ها و متدها که فقط یک وظیفه دارند، استفاده کنید.

بنابراین، اصلِ SRP در شیءگرایی به معنی این است که هر کلاس و متد باید فقط یک مسئولیت یا وظیفه را برعهده داشته باشد. این یعنی که هر کلاس فقط باید یک مورد از نرم‌افزار را انجام دهد و تغییر در آن، صرفا برای اعمال تغییرات در آن مورد خاص یا در راستای بهبود مسئولیتش باشد. با رعایت اصلSRP ، کدها به راحتی قابل نگهداری، توسعه و آزمایش خواهند بود! زیرا هر قطعه از کد فقط برای انجام کار خاص خود طراحی شده است و هیچ گونه وظیفه‌ای به کلاس اضافه نشده است. این نه تنها باعث بهبود خوانایی و قابلیتِ پیش‌بینیِ کد می‌شود، بلکه باعث کاهش پیچیدگی و احتمال خطا نیز می‌شود.

برای پیاده‌سازی SRP در سی‌پلاس‌پلاس مثالی خواهم زد؛ ابتدا باید کلاس را به شکلی طراحی کنید که فقط یک مسئولیت را در بر داشته باشد. به عنوان مثال، فرض کنید یک کلاس برای محاسبه مساحت یک شکل هندسی طراحی می‌کنید. با توجه بهSRP ، این کلاس فقط باید مسئول محاسبه مساحت باشد و هیچ وظیفه‌ای دیگر را نباید برعهده بگیرد.

```
class Shape
{
   public:
   virtual double area() const = 0;
};

class Rectangle : public Shape
{
   public:
   Rectangle(double h, double w) : height(h), width(w) {}
   double area() const override
   {
      return height * width;
   }
```

```
private:
   double height;
   double width;
};

class Circle : public Shape
{
public:
   Circle(double r) : radius(r) {}
   double area() const override
   {
      return 3.1415 * radius * radius;
   }
private:
    double radius;
};
```

در این مثال، کلاس Shape یک مسئولیت واحد دارد که محاسبه مساحت شکل هندسی است. همینطور کلاس‌های Rectangle و Circle نیز فقط مسئول محاسبه مساحت هر یک از شکل‌های هندسی خود هستند.

به این ترتیب، هر یک از این کلاس‌ها فقط یک مسئولیت دارند و تغییراتی که در آینده رخ می‌دهد، مربوط به تابعی است که این مسئولیت را پوشش می‌دهد و هرگونه تغییرات دیگری نباید شامل کلاس شود.

مثال در جاوا:

کلاس کتاب زیر اصل SRP را نقض می کند چون غیر از داشتن مشخصات اصلی کتاب، عمل سیو و عمل پرینت کتاب را هم انجام می دهد و بیش از 1 وظیفه و در نتیجه بیش از 1 دلیل برای ویرایش دارد!

Java

```
// Violates SRP
public class Book {
    private String name;
    private String author;
    private String text;

    public void printBook() {
        // print the book
    }

    public void saveBook() {
        // save the book to database
    }
}
```

می توان آنرا به سه کلاس زیر شکاند:

```
// Follows SRP
public class Book {
    private String name;
    private String author;
    private String text;

    // getters and setters
}

public class BookPrinter {
    public void printBook(Book book) {
        // print the book
    }
}

public class BookRepository {
    public void saveBook(Book book) {
        // save the book to database
    }
}
```

نتیجه ساختاری SRP:

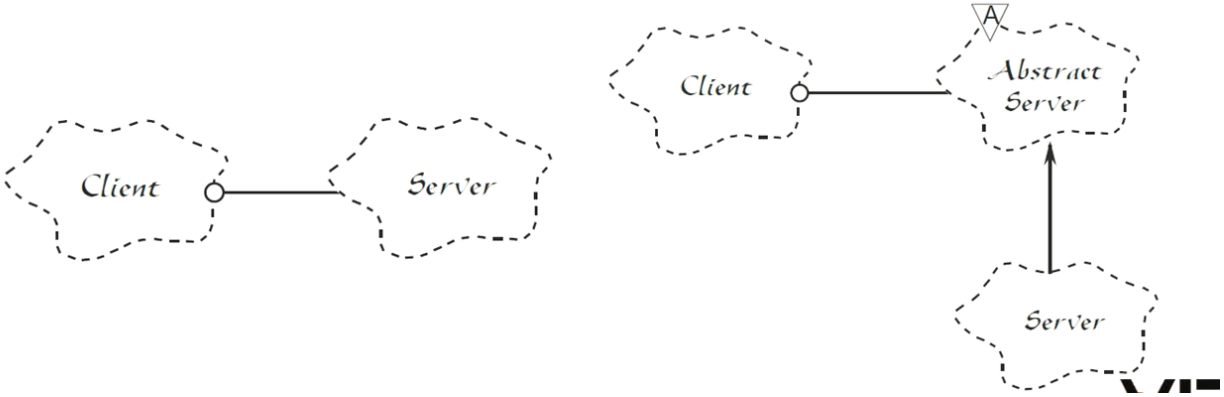
معمولا یک استراکت یا کلاس ظرف پراپرتی با/بدون setter ها و getter های پراپرتی ها داریم و عملکرد های مهم جدا شده، هر کدام به صورت یک کلاس جدید در می آیند و به کانسترکتور هر کدام از آنها، شیی از کلاس ظرف پراپرتی ها را می دهیم تا محاسبات اصلی را انجام دهند!

اصل 2: اصل (OCP) The Open-Close Principle :

هر کلاس باید برای توسعه باز و برای تغییر بسته باشد .یعنی بتوانیم رفتار کلاس را با ارث بری تغییر دهیم بدون اینکه کد اصلی کلاس را تغییر دهیم.

کلاس باز به کلاسی گفته می شود که final نباشد و بتوان آنرا توسعه داد و به آن، متد و پراپرتی اضافه کرد و یا از آن کلاس دیگری extend کرد .
کلاس بسته به کلاسی گفته می شود که final باشد و نه بتوان از آن extend کرد و نه توسعه داد و به قدری کامل و تست شده و بی نقص باشد که نیازی به توسعه نداشته باشد!
پس این الگو، ما را به سمت عدم تعریف کلاسهای final سوق می دهد ولی در عین حال باید برای ساخت انواع جدید ، کلاسهای جدید را زیر کلاس کلاس یا اینترفیس اصلی ساخت! با این کار انواع جدید کلاسها را به عنوان زیر کلاس کلاس abstract یا پیاده ساز اینترفیس خاص قابل گسترش کرده ایم ولی در عین حال، جلوی تغییر کدهایی را که قبلا درست کار می کردند را گرفته ایم.

برنامه باید برای توسعه و تغییر و اضافه کردن یا کم کردن امکانی باز باشد و امکان توسعه برنامه با کمترین یا بدون تغییر بخش های دیگر برنامه انجام گیرد! برای این کار، بخش های مختلف برنامه باید قابلیت استفاده مجدد داشته باشند! این عمل با Abstraction امکانپذیر است.



در مثال فوق اگر از کلاس سرور در کلاینت استفاده شود و تغییری در کلاس سرور رخ دهد، باید انتشار تغییر در خود سرور و در کلاینت مینیمم شود. برای این کار از کلاس ابسترکت و از مفهوم abstraction استفاده می شود به این صورت که مفهوم اولیه کلاس توسط یک کلاس abstract تعریف می شود و کلاس های سطوح پایینتر توسعه می یابند و با آمدن نیازمندی جدید، کلاس جدیدی بوجود آمده و ارتقا می یابد تا دیگر کلاسهای قبل به کمترین میزان ممکن دستکاری شوند و با توسعه کلاس جدید، بخش های مورد نیاز در کلاینت تغییر یابند تا از کلاس جدید استفاده کنند. معمولا با این روش، هر کلاس به حد بدون باگ و پایدار خود می رسد و دیگر دست نمی خورد و در صورت نیاز، یک کلاس دستکاری می شود و اگر تغییرات side effect داشته باشد، آنرا در بخش های مورد نیاز اعمال می کنند!

این اصل تلاش دارد به جای تغییر روی یک متد قابل گسترش همچون متد return کردن عبارت بر اساس کلید زبان خاص، تعداد کلاسها را (مثلا برای هر زبان یک کلاس جدید) بالا ببرد تا به جای تغییر متد، کلاسهای جدید داشته باشیم و همه این کلاسها زیر کلاس یک زبان abstract باشند!
البته بهتر می توان با interface ها آنها را پیش برد!
هر کلاس و هر متد فقط باید یک دلیل برای تغییر داشته باشد.
پس هر کلاس یا متدی که بوجود می آوریم فقط و فقط یک وظیفه اصلی دارد. حتی log کردن نیاز به کلاس مجزای خود دارد!

می توانید به جای اینکه برای تایپ های مختلف یک متد با type enum تعریف کنید تا بر اساس type enum چند رفتار متفاوت را انجام دهد، یک کلاس ابسترکت با انواع کلاسهای sub بوجود آورده و برای هر تایپ، کلاس جدیدی داشته باشید تا در هنگام تغییر، به جای اینکه یک متد یک کلاس همیشه تغییر کند یک کلاس جزئی تر تغییر کند!
همچنین می توان به جای Inheritance از Interface استفاده کرد و به جای یک enum در یک تابع همه فن حریف، متد های مختلف را در Interface به صورت پروتوتایپ تعریف کرد و زیر کلاس Implement شده همه متدهای جدا شده را پیاده کرده و در مواقع تغییر، فقط یک تابع تغییر کند به جای اینکه متد همه فن حریف همیشه تغییر یابد!

مثال: فرض کنید ما در اپلیکیشن فروشگاهی، انواع مختلف مشتریان داریم و می خواهیم برای هر نوع، تخفیفی را حساب کنیم. در حالت اولیه کد را اینگونه نوشته ایم که ایراداتی دارد.

```
namespace OpenClosedPrincipleDemo
{
    public class Customer
    {
        public int CustomerType { get; set; }

        public double GetDiscount(double TotalSales)
        {
            if (CustomerType == 1)
            {
                return TotalSales - 100;
            }
            else
            {
                return TotalSales - 50;
            }
        }
    }
}
```

اگر در آینده، انواع دیگر مشتریان اضافه شوند باید این کلاس را دوباره ویرایش کنیم. ولی برای جلوگیری از این ویرایش، تابع مربوطه را ویرچوال می کنیم تا کلاس ابسترکت مشتری تعریف شود.

```
public class Customer
{
    public int CustomerType { get; set; }

    public virtual double GetDiscount(double TotalSales)
    {
        return TotalSales;
    }
}
```

سپس می توانیم کاربران نوع طلایی و نقره ای و برنزی را به صورت کلاسهای مجزای زیر کلاس کاستومر تعریف کنیم و برای هر کدامشان، تابع ابسترکت مربوطه را override کنیم.

```
class GoldCustomer:Customer
{
    public override double GetDiscount(double TotalSales)
    {
        return TotalSales - 100;
    }
}
```

```
class SilverCustomer:Customer
{
    public override double GetDiscount(double Total
    {
        return TotalSales - 50;
    }
}
```

```
class BronzeCustomer:Customer
{
    public override double GetDiscount(double TotalSales)
    {
        return base.GetDiscount(TotalSales);
    }
}
```

با این کار دیدیم که به جای اینکه کلاس مشتری را تغییر دهیم، مشتری به مشتریهایمان اضافه کردیم و آنها را گسترش دادیم. پس کلاسی اضافه شد و کلاسهای دیگر دست نخوردند. حالا فرض کنید یک نوع مشتری سوپر ویژه جدید به نام مشتری الماسی یا DiamondCustomer تعریف می کنیم و بالاترین تخفیف ممکن را به وی می دهیم. در این شرایط کافیست کلاس جدید مشتری الماسی را بسازیم و متد تخفیف را برایش Override کنیم و در شرایطی که بخواهیم تخفیف مشتری الماسی را در آینده ویرایش کنیم، تابع تخفیفات کلاس مشتری الماسی ویرایش می شود و دیگر کلاس اصلی customer ویرایش نمی شود!

ویرایش نشدن کلاسی که قبلا کار می کرده و انواع مشتریان را پشتیبانی مستقیم می کرده از آنجایی مهم است که هر ویرایش می تواند دسته ای از کدهای مرتبط با کلاسهای دیگر همچون مشتریان دیگر را نیز از کار بیندازد در صورتیکه ایرادات وارده به مشتری الماسی فقط برای مشتری الماسی باعث خطا می شود و خدشه ای به مشتریان قبلی وارد نمی شود!

مثال دیگر برای Open-Close Principle:

یک کلاس لاگر را در نظر بگیرید که به ازای هر نوع خطا یا لاگ مورد نیازی، پیام خاصی را چاپ می کند.

```
public class Logger
{
    public void Log(string message, LogType logType)
    {
        switch (logType)
        {
            case LogType.Console:
                Console.WriteLine(message);
                break;

            case LogType.File:
                // Code t (enum member) LogType OpenClosedPrincipleDemo.LogType.File
                break;
        }
    }
}
```

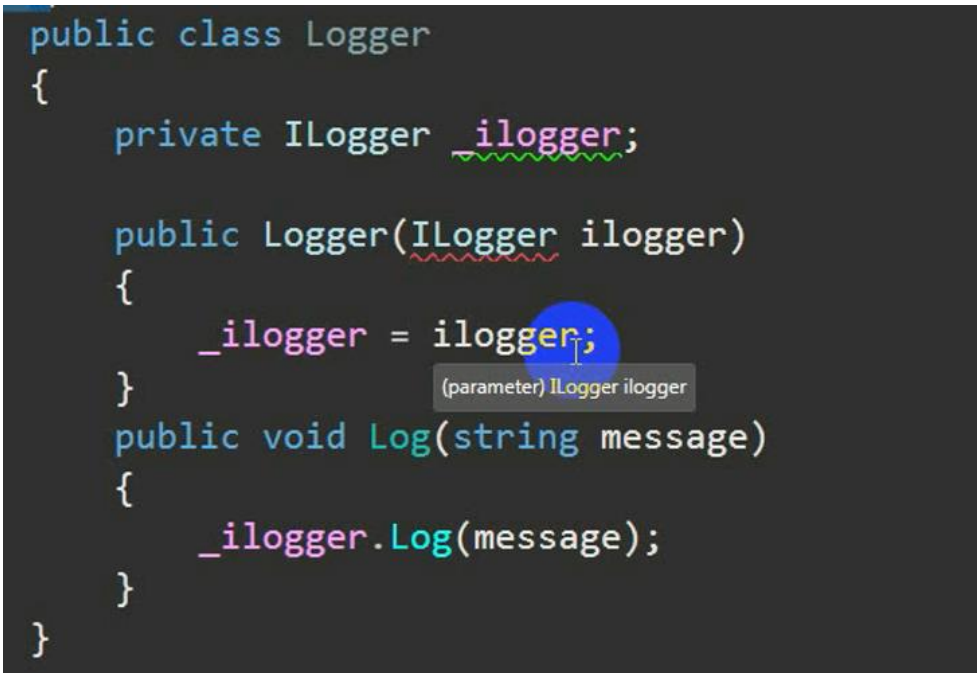
در این مثال، هر نوع لاگ دیگری که لازم باشد، باید این فایل لاگر را ادیت کنیم. ولی این بر خلاف منویات OCP است. برای جلوگیری از این مساله، آنرا به چند فایل می شکنیم. برای این کار یک اینترفیس برای لاگر تعریف کردیم.

```
namespace OpenClosedPrincipleDemo
{
    interface ILogger
    {
        void Log(string message);
    }
}
```

حال انواع کلاسهای لاگر را می سازیم که از این اینترفیس ایمپلیمنت می کند. پس هر کدامشان باید متد لاگ را داشته باشند که نوع void است و رشته ورودی را می-گیرد.

```
public class PrinterLogger : ILogger
{
    public void Log(string message)
    {
        //Write data to printer
    }
}

public class DatabaseLogger : ILogger
{
    public void Log(string message)
    {
        //Write data to database
    }
}
public class Logger
```

حالا کلاس اصلی لاگر را می سازیم:



این کلاس لاگر، یکی از انواع لاگر های بالا را که از اینترفیس مدنظر پیاده سازی می شوند را می گیرد و با ران کردن متد لاگ این کلاس، متد لاگ آنها ران می شود. پس دقت می شود که با این کار، نیاز به اصلاح را به حداقل می رسانیم و با اضافه شدن هر نوع لاگ جدیدی، یک کلاس جدید از اینترفیس Ilogger ساخته می شود.

برای پیاده‌سازیOCP ، می‌توانید از الگوی معماری Visitor و Strategy استفاده کنید. این باعث می‌شود که کلاس‌های شما قابلیت بسته‌بودن (Close) و در عین حال قابلیت گسترش (Open) را داشته باشند.

تا به اینجا فهمیدیم برای پیاده‌سازیOCP ، می‌توانید از ارث‌بری، پلی‌مورفیسم، استفاده از ابستراکت کلاس‌ها (کلاس‌های انتزاعی)، الگوی تزریق وابستگی و ... استفاده کنید. این اصل بهبود پذیری (extensibility) و بهره‌وری کد را بهبود می‌بخشد .همچنین برای پیاده‌سازیOCP ، می‌توانید از الگوی Visitor و Strategyاستفاده کنید. این باعث می‌شود که کلاس‌های شما قابلیت بسته‌بودن (Close) و در عین حال قابلیت گسترش (Open) را داشته باشند.

به عنوان مثال، فرض کنید یک برنامه داریم که می‌تواند اشکال هندسی مختلف را رسم کند. برای پیاده‌سازیOCP ، می‌توانید کلاس اشکال هندسی را به گونه‌ای طراحی کنید که بتوانید به راحتی از آن‌ها ارث‌بری کنید و اشکال جدیدی را به سیستم اضافه کنید بدون ایجاد تغییرات بیشتر در کد قبلی. به عنوان مثال، اینجا یک کد ساده برای این کار نوشته شده است:

```
#include <iostream>
#include <vector>

class Shape {
public:
    virtual void draw() = 0;
};

class Rectangle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a rectangle\n";
    }
};
```

```
class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle\n";
    }
};

class GraphicEditor {
public:
    void drawAllShapes(std::vector<Shape*> shapes) {
        for (auto shape : shapes) {
            shape->draw();
        }
    }
};

int main() {
    GraphicEditor editor;
    std::vector<Shape*> shapes = { new Rectangle(), new Circle() };
    editor.drawAllShapes(shapes);
    for (auto shape : shapes) {
        delete shape;
    }
    return 0;
}
```

در این مثال، کلاس Shape در واقع یک واسط برای هر یک از اشکال هندسی است. کلاس Rectangle و کلاس Circle از کلاس Shape ارث‌بری کرده‌اند و تابع draw آن را پیاده‌سازی کرده‌اند. به این ترتیب، در آینده می‌توانید اشکال هندسی جدیدی را به کد اضافه کنید! آن هم بدون نیاز به تغییر کد اصلی.

Java

```
// Violates OCP
public class AreaCalculator {
    public double calculateArea(Object[] shapes) {
        double area = 0;
        for (Object shape : shapes) {
            if (shape instanceof Rectangle) {
                Rectangle rectangle = (Rectangle) shape;
                area += rectangle.getLength() * rectangle.getWidth();
            } else if (shape instanceof Circle) {
                Circle circle = (Circle) shape;
                area += Math.PI * circle.getRadius() * circle.getRadius();
            }
            // What if we want to add more shapes?
        }
        return area;
    }
}
```

```
// Follows OCP
public interface Shape {
    double calculateArea();
}

public class Rectangle implements Shape {
    private double length;
    private double width;

    // constructor, getters and setters

    @Override
    public double calculateArea() {
        return length * width;
    }
}

public class Circle implements Shape {
    private double radius;

    // constructor, getters and setters

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

public class AreaCalculator {
    public double calculateArea(Shape[] shapes) {
        double area = 0;
        for (Shape shape : shapes) {
            area += shape.calculateArea();
        }
        return area;
    }
}
```

3- اصل Liskov Substitution Principle یا جایگزاری لیسکو یا LSP:

لیسکو نام کسی است که این اصل را مطرح کرده است.

هر زیر کلاس باید قابل جایگزینی با کلاس پدرش باشد .یعنی اگر یک کلاس از یک کلاس دیگر ارث بری کرده باشد، باید بتوانیم از آن کلاس به جای کلاس پدرش استفاده کنیم بدون اینکه منطق برنامه خراب شود! به بیان بهتر، کلاس پدر، کلاس فرزندی را ملزم به پیاده سازی متدی که سازگار با فرزند نیست نمی کند! در عوض فرزندان، متدهایی را که مختص خودشان است و جایی در نمونه خواهر و برادران یا پدرشان ندارند را در خودشان پیاده می کنند و آنها را از پدرانشان به ارث نمی برند.

به عبارت دیگر، هر جایگزین یا زیرکلاس باید بتواند عملکرد و ویژگی‌های کلاس والد را حفظ کند و بدون هیچ خطایی، به جای کلاس والد استفاده شود. در واقع، هدف این اصل این است که به جای ایجاد جایگزین‌هایی که منجر به عملکرد غیرقابل پیش‌بینی می‌شوند، جایگزین‌هایی داشته باشیم که با انجام کارهایی مانند جایگزینی هنوز هم به نحو شایسته عمل کنند.

فرض کنید که گربه و سگ، از یک کلاس حیوان وارث هستند. برای همه حیوانات منطقی است که بتوانند صدای حیوان را تولید کنند. در اینجا با توجه به اصلLSP ، باید بتوانیم به جای یک شیء گربه، یک شیء سگ یا شیء کلاس والدِ حیوان را به عنوان جایگزین استفاده کنیم و همچنین از آن‌ها بخواهیم که قابلیت تولید صدا را داشته باشند. به عبارت دیگر، صدای گربه و صدای سگ برای ما دقیقاً در یک رده‌بندی هستند و در همان حیطه معنایی قرار دارند، بنابراین باید بتوانیم هر یک از آن‌ها را به عنوان جایگزین برای دیگری استفاده کنیم.

برای پیاده سازی اصل لیسکو باید contravariance و covariance در کلاسهایمان اعمال شود. Contravariance می گوید در سلسله مراتب کلاسها، باید پارامترهای کلاس پدری با فرزندی برابر باشد مگر اینکه کلاسهای فرزندی آنها را محدودتر کنند. Covariance هم می گوید باید return type متدهای همنام کلاسهای پدری و فرزندی مثل هم باشد مگر اینکه کلاسهای فرزندی آنها را رستریکت تر کنند.

همچنین precondition ها و postcondition های کلاسهای پدر و فرزندی باید برابر باشد. مثلا اگر قرار است کلاس پدر قبل از خواندن داده ها، دیتابیس کانکشن را باز کند و بعد از ذخیره داده ها روی فایل، فایل را ببندد، همین ترتیب باید در کلاس های فرزندی هم لحاظ شود و نباید ترتیب آنها شدیدتر یا ضعیف تر باشد.

اصل بعدی در لیکسو، invariant ها است. یعنی چیزی که غیر قابل تغییر است. اصل لیسکو می گوید Invariant های کلاس پدری باید در فرزند هم عینا لحاظ شود. یعنی مثلا اگر قرار است در طول ران بودن یک کلاس پدری یا یک متد خاص، دیتابیس باز باشد، نباید کلاس یا متد فرزندی آنرا عوض کند.

موضوع آخر History Constraint است. فرض کنید سن کسی در کلاس پدر نباید از مقدار خاصی کمتر یا بیشتر باشد ولی در کلاس فرزندی عوض می شود. این باعث رعایت نشدن History Constraint در اصل لیکسو می شود.

در نتیجه تمام تلاش اصل Liskov آنست که کلاس فرزندی، فقط کلاس پدری را گسترش دهد و مشتق و تکامل یافته آن باشد! نه تغییر دهنده و منحرف کننده منطق پدری!

یک مثال ساده از اصل LSP درC++20 ، می‌تواند مربوط به کلاس‌های Shape و Rectangle باشد. فرض کنید کلاس Rectangle از کلاس Shape و از آن ارث‌بری کرده باشد. برای رعایت اصلLSP ، کلاس Rectangle باید تمام روش‌های کلاس Shape را پیاده‌سازی کند، به گونه‌ای که در هر جایی که در کد از کلاس Shape استفاده می‌شود، می‌توانیم جایگزین آن را با کلاس Rectangle استفاده کنیم.

کلاس Shape می‌تواند به صورت زیر باشد:

```
class Shape {
public:
    virtual double area() = 0;
    virtual double perimeter() = 0;
};
```

و کلاس Rectangle از شکل یک مستطیل با طول و عرض دلخواه پیاده‌سازی شده است:

```
class Rectangle : public Shape {
private:
    double length;
    double width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}
    double area() override {
        return length * width;
    }
    double perimeter() override {
        return 2 * (length + width);
    }
};
```

حال، با استفاده از این کلاس‌ها، می‌توانیم مستطیلی با طول و عرض دلخواه ایجاد کنیم و روش‌های کلاس Shape را روی آن صدا بزنیم:

```
Shape* shapePtr = new Rectangle(4, 6);
double area = shapePtr->area();
double perimeter = shapePtr->perimeter();
```

در اینجا، کلاس Rectangle با کلاس Shape جایگزین شده و اصل LSP رعایت شده است، به این معنی که هر جایی که از کلاس Shape استفاده شده، می‌توانیم جایگزین آن را با کلاس Rectangle استفاده کنیم، بدون هیچ تغییری در کد قبلی.

برای پیاده‌سازی LSP ، می‌توانید از وراثت، کلاس‌های پایه (Base classes) و کلاس‌های مشتق (Derived classes) استفاده کنید و مطمئن شوید که اصول وراثت رعایت شده است.

مثال جزئی تری در جاوا: در مثال زیر، اصل لیسکو نقض شده است:

Java

```
// Violates LSP
public class Rectangle {
    private double length;
    private double width;

    // constructor, getters and setters
    public double getArea() {
        return length * width;
    }
}

public class Square extends Rectangle {
    // constructor
    @Override
    public void setLength(double length) {
        super.setLength(length);
        super.setWidth(length);
    }
```

```
    @Override
    public void setWidth(double width) {
        super.setWidth(width);
        super.setLength(width);
    }
}

public class LSPDemo {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle();
        rectangle.setLength(5);
        rectangle.setWidth(4);
        System.out.println(rectangle.getArea()); // prints 20
        Rectangle square = new Square();
        square.setLength(5);
        square.setWidth(4);
        System.out.println(square.getArea()); // prints 16, violates LSP
    }
}
```

مثال بالا، LSP را نقض کرده است! چون هدف کانسترکتور و توابع setWidth و setLength صرفا تنظیم طول و عرض مستطیل بود که می توانستند برابر باشند و می توانستند برابر نباشند! در صورتیکه کلاس مربع که از مستطیل مشتق شده است، قاعده و منطق پدری را کنار زد و با تنظیم width، مقدار length را نیز دست کاری می کرد و بر عکس!

این اتفاق باعث می شود که وقتی نمونه شی کلاس مستطیل را در متغیر مستطیل می ریزید و وقتی نمونه شی مربع را نیز در متغیر مستطیل می ریزید و برای هر دو شی، متد تنظیم width و height را با مقادیر مشابه صدا می زنید، رفتار مربع در تنظیم مقدار محیط و مساحت خودش، با رفتار مستطیل در تنظیم مقدار محیط و مساحت خودش فرق داشته باشد! اینجا طراحی غلط پدری، همه فرزندان را ملزم کرده اند که طول و عرض داشته باشند و بر این اساس، مربع مجبور شده است قواعد پدرش را به زور تغییر دهد! در صورتیکه پدران باید قواعد مشترک را پیاده کنند به صورتیکه فرزندان مجبور نباشند آنها را از بیخ و ریشه به نفع هدف خودشان تغییر دهند و معنا و مفهوم را عوض کنند! بلکه باید آنها را بر اساس رفتارهای خود Override یا محدودتر کنند! اگر قرار است فرزندان هر کدام ساز مخالف و خاص خود را بزنند، آنها نباید ساز مخالفشان را در مقابل پدرشان بزنند، بلکه باید هر کدام، ساز مخالف و سازگار خودشان را، جدای از الزام پدران، در درون خودشان پیاده کنند!

چون وقتی محیط محاسبه شده مربع با محیط محاسبه شده توسط کلاس والد مستطیل با هم برابر نیست! یعنی اگر به جای شی مستطیل، اگر شی فرزندی مربع را استفاده کنیم، باید نتیجه حاصل شده توسط مربع با نتیجه حاصل از مستطیل برابر باشد ولی تغییر رفتار را شاهد هستیم و مستطیل، برای طول 4 و عرض 5، مساحت 20 را محاسبه کرده و مربع برای همان مقادیر، عدد 16 را محاسبه کرده است که با هم تفاوت دارند! 16 != 20

در اینجا برای اصلاح رعایت LSP به این روش عمل می کنیم (پیاده سازی تغییرات هر کلاس از همان کلاس فرزندی آغاز می شود و به کلاس پدر اتلاق نمی شود):

```
// Follows LSP
public interface Shape {
    double getArea();
}
```

```
public class Rectangle implements Shape {
    private double length;
    private double width;

    // constructor, getters and setters
    @Override
    public double getArea() {
        return length * width;
    }
}

public class Square implements Shape {
    private double side;
    // constructor, getters and setters
    @Override
    public double getArea() {
        return side * side;
    }
}

public class LSPDemo {
    public static void main(String[] args) {
        Shape rectangle = new Rectangle();
        rectangle.setLength(5);
        rectangle.setWidth(4);
        System.out.println(rectangle.getArea()); // prints 20
        Shape square = new Square();
        square.setSide(5);
        System.out.println(square.getArea()); // prints 25, follows LSP
    }
}
```

حالا پیاده سازی طول و عرض مستطیل و ضلع مربع به صورت جداگانه در هر کلاس پیاده سازی شده و پدری یک نسخه مشترک به هر دوی آنها تجویز نکرده و لذا مساحت هر دوی آنها به صورت صحیح کال می شود!

البته در C++ نمی توان بدون cast کردن پدر به فرزند، از طریق اشاره گر پدر به متدهای فرزندی دسترسی داشت و در آنجا باید با dynamic_cast، کلاس پدر را به کلاس فرزندی dynamic_cast کرد تا بتوان از طریق اشاره گر پدری، متدهای مختص فرزند را صدا کرد. علل الخصوص که با استفاده از virtual متدها، خاصیت polymorphismic را نیز به این کلاسها داده ایم و dynamic_cast می تواند روی سلسله کلاسهای polymorphysmic عمل کند!

```
dynamic_cast< Rectangle*>(rectangle)-> setWidth(4);
dynamic_cast< Square*>( square)-> setSide(5);
```

4- اصل interface segregation principle:

هر کلاس باید به جای اینکه از اینترفیس های بزرگ و کلی استفاده کند، از اینترفیس های کوچک و مخصوص خودش استفاده کند. یعنی باید اینترفیس ها را به گونه ای طراحی کنیم که کلاس هایی که از آن ها پیاده سازی می کنند، فقط متدهایی را پیاده سازی کنند که به آن ها نیاز دارند.

Segregation کردن بر خلاف aggregation کردن است. این اصل یعنی جداسازی اینترفیس ها برای اینکه اینترفیس ها اصطلاحا چاق و چله نباشند و اصطلاحا باید کوهسیو باشند و کلاسهای فرزندی که اینترفیس ها را پیاده سازی می کنند نباید مجبور باشند متدهای پرت و به درد نخورشان را پیاده سازی کنند:

فرض کنید می خواهید کلاس درب را بسازید که می تواند قفل شود یا از قفل باز شود و وضعیت قفل بودن را نشان دهد. می خواهید یک زیر کلاس بسازید که اگر بیش از اندازه باز بماند، مثلا آژیر بزند. اینجا به کلاس تایمر هم نیاز دارید.

Interface Segregation Principle

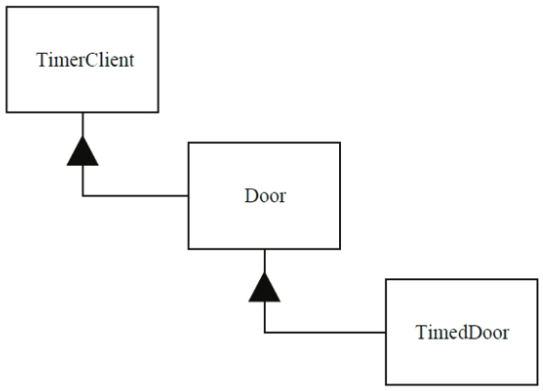
Security Door

```
class Door
{
  public:
    virtual void Lock()   = 0;
    virtual void Unlock() = 0;
    virtual bool IsDoorOpen() = 0;
};
```

Timer

```
class Timer
{
  public:
    void Regsiter(int timeout, TimerClient* client);
};

class TimerClient
{
  public:
    virtual void TimeOut() = 0;
};
```

اگر ساختار کدتان را به این شکل اشتباه بسازید:



کلاسهایی که نیازی به اینترفیس تایمر ندارند هم باید متدهای فورس شده توسط اینترفیس تایمر را پیاده سازی کنند! مثلا یک درب ساده بدون نیاز به تایمر هم آنرا پیاده سازی نماید حال اینکه هیچ نیازی به آن ندارد.

اصل چهارم به ما می گوید که کلاسها نباید مجبور باشند متد هایی را که به آنها احتیاج ندارند را پیاده کنند!
یعنی هر interface فقط باید یک کار را انجام می دهد! تفاوت این اصل با اصل اول آنست که اصل اول در مورد همه کلاسها و متدها صحبت می کند ولی اصل چهارم فقط در مورد اینترفیس ها صحبت می کند!
وقتی یک انسان فقط تشنه است مجبور نیست چیزی بخورد و وقتی فقط گرسنه است مجبور نیست آب هم بنوشد!

اگر قرار است نوع دستگاه کپی حرفه ای (پرینتر – اسکنر- فکس) و نوع پرینتر ساده که فقط پرینت می کند را هم گروه بدانیم و برایشان Interface تعریف کنیم، اگر Interface هر سه وظیفه print , fax , scan را به همراه داشته باشد و فرزندان implement کرده آن اینترفیس بخواهند هر سه متد را پیاده سازی کنند تا کامپایلر گیر ندهد، دستگاه ساده به دو متد اسکن و فکس نیازی نخواهد داشت با وجودی که مجبور است به صورت خالی آنها را پیاده کند در صورتیکه نیازی به آنها ندارد و این مخالف اصل ISP سالید است!

Java// Violates ISP

```
public interface Printer {
    void print();
    void fax();
    void scan();
}

public class SimplePrinter implements Printer {
    @Override
    public void print() {
        // print
    }

    @Override
    public void fax() {
        // do nothing, not supported
    }

    @Override
    public void scan() {
        // do nothing, not supported
    }
}
```

پس اینترفیس های بزرگ را به اینترفیس های کوچک بشکنید!

```
// Follows ISP
public interface Printer {
    void print();
}

public interface Fax {
    void fax();
}

public interface Scanner {
    void scan();
}

public class SimplePrinter implements Printer {
    @Override
    public void print() {
        // print
    }
}

public class MultiFunctionPrinter implements Printer, Fax, Scanner {
    @Override
    public void print() {
        // print
    }
```

```
    @Override
    public void fax() {
        // fax
    }

    @Override
    public void scan() {
        // scan
    }
}
```

تعریف کردید، در حین نوشتن کلاسی که آنرا پیاده سازی کند مثل دلفین، همچنان که دلفین ها پرواز نمی کنند، متد `run` و `eat` و `fly` را ساختید و برایش `animal`اگر اینترفیس تعریف می کنیم تا دلفین که پرواز نمی کند مجبور نباشد آنرا پیاده کند. `FlyableAnimals` را در اینترفیس جداگانه ای برای `fly` برایش غیر قابل استفاده خواهد بود پس `fly` در نتیجه این اصل تلاش دارد اینترفیس ها را از هم جدا کند!

فرض کنید که شما کلاس ابسترکت Customer را به شکل زیر ساخته اید:

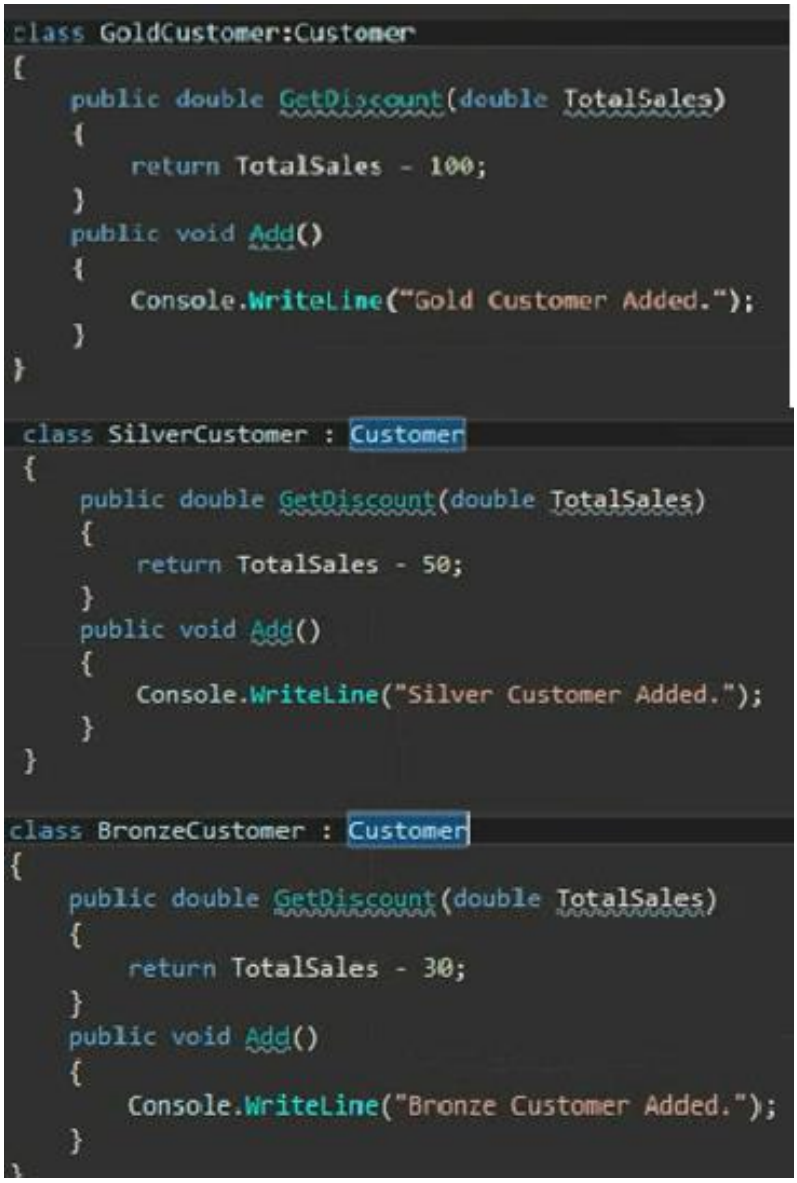
```
public class Customer:IDatabase, IDiscount
{
    public int CustomerType { get; set; }

    public virtual double GetDiscount(double TotalSales)
    {
        return TotalSales;
    }
    public virtual void Add()
    {
        Console.WriteLine("Added");
    }
}
```

این کلاس ابسترکت از دو اینترفیس زیر implement می کند:

```
public interface IDiscount
{
    double GetDiscount(double totalSales);
}
```

```
public interface IDatabase
{
    void Add();
}
```

همانطوری که کلاس ابسترکت Customer از هر دو اینترفیس تبعیت می کند و باید هر دو متد را پیاده سازی کند، فرزندان Customer یعنی مشتری طلایی و نقره ای و برنزی هم هر دوی این متدها را از پدر به ارث برده اند.



هر سه این کلاسهای فرزندی اختیار این را داشتند که هر دو متد پدر خود را پیاده کنند. در این مثال پدر لزوما ابسترکت صرف نیست بلکه خاصیت override کردن را در فرزندان گذاشته ولی می تواند ابسترکت صرف هم باشد. (pure virtual)

حال فرض کنید کلاس دیگری نوشته اید که قرار است یکی از دو متد بالا را داشته باشد و ذاتا متد دوم را ندارد. مثلا کلاس Lead!

می توانید آنرا زیر کلاس Customer تعریف کنید و متد نالازمش را throw exception کنید تا کال نشود.

سپس در لیستی از نوع کلاس پدری وقتی همه انواع 4 فرزند را new کردید و خواستید متد نالازم را کال کنید، شی ساخته شده از کلاس Lead به خطای اکسکپشن می خورد و این با اصل interface segregation همخوانی ندارد. اصل چهارم می گوید شما نباید کلاسی را زیر کلاس کلاس دیگری کنید که یک یا چند متد از متدهای آنرا نیاز ندارد و ذاتا نباید داشته باشد. مثل اینکه کلاس انسان را زیر کلاس Animal کرده باشید و این Animal قابلیت پرواز هم دارد ولی در مورد انسان آنرا با throw exception کور کرده باشید. پس در این مورد ما می خواهیم کاری کنیم تا امکان کال کردن متد به ارث رفته کور شده را از انسان برداریم.

در این ساختار به این روش عمل می کنیم:

لیست مد نظر را از نوع اینترفیسی که همگی اشیای داخلی آن متد های اصلی لازم را دارند تعریف می کنیم. مثلا همه Animal ها Run دارند و اینترفیس IRunner را تعریف می کنیم که متد Run را دارد و Animal ها از آن پیاده سازی و تبعیت می کنند. در عین حال کلاس ربات هم ساخته ایم که Run دارد ولی لزوما Animal نیست. پس لیستی از نوع IRunner می سازیم و همه حیوانات ساخته شده و ربات های ساخته شده را از آن پیاده سازی می کنیم و هر موقع خواستیم، متد Run همه

آنها را کال می کنیم. در اینجا، اینترفیس مد نظر همانند فیلتر عمل می کند که Run را دارد و به ما امکان استفاده از Run را می دهد. ولی هیچ کدام از متدهای غیر مشابه را ندارد!

مثال خود جلسه: با این روش ما می توانیم مساله Customer ها و کلاس Lead را حل کنیم:

```
class Lead:IDiscount
{

    public double GetDiscount(double TotalSales)
    {
        return TotalSales - 10;
    }
}
```

حالا کلاس Lead به جای اینکه زیرکلاس Customer باشد، از اینترفیس IDiscount تبعیت می کند و مجبور است همان متد تخفیف را پیاده سازی کند. حالا لیست اشیا را به جای اینکه از نوع کلاس Customer تعریف کنیم از نوع IDiscount تعریف می کنیم:

```
List<IDiscount> _customerCollection = new List<IDiscount>();
_customerCollection.Add(new GoldCustomer());
_customerCollection.Add(new SilverCustomer());
_customerCollection.Add(new BronzeCustomer());
_customerCollection.Add(new Lead());
foreach (IDiscount cust in _customerCollection)
{
    Console.WriteLine( cust.GetDiscount(200));

}
Console.ReadKey();
```

حالا مشاهده می کنید که می توانیم متد تخفیف را برای کل اشیای لیست کال کنیم و کلاس Lead مجبور نیست متد اضافی Add را که ذاتا نباید داشته باشد (همانند متد پرواز برای انسان) را پیاده کند!

مثال: فرض کنید ما کلاس ProjectFile را به عنوان کلاس پدر داریم:

```
public class ProjectFile
{
    public string FilePath { get; set; }

    public byte[] FileData { get; set; }

    public void LoadFileData()
    {
        // Retrieve FileData from disk
    }

    public virtual void SaveFileData()
    {
        // Write FileData to disk
    }
}
```

حال کلاس ReadInlyProjectFile را به این صورت داریم:

```
public class ReadOnlyFile : ProjectFile
{
    public override void SaveFileData()
    {
        throw new InvalidOperationException();
    }
}
```

همانطوری که می بینید در وهله اول اصلLSP و در وهله دوم ISP نقض شده چون کلاس ReadOnlyFile متد سیو دیتا از کلاس پدری را با اکسکپشن غیر فعال کرده و کلاس پدر نمی تواند برای این فرزند، متد سیو را ران کند.

```
public class Project
{
    public List<ProjectFile> ProjectFiles { get; set; }

    public void LoadAllFiles()
    {
        foreach (ProjectFile file in ProjectFiles)
        {
            file.LoadFileData();
        }
    }

    public void SaveAllFiles()
    {
        foreach (ProjectFile file in ProjectFiles)
        {
            if (file as ReadOnlyFile == null)
                file.SaveFileData();
        }
    }
```

در اینجا لیستی از فرزندان کلاس پدر داریم و در لود همگی دستور لود کردن را صادر می کنند و در سیو برای اینکه به اکسکپشن نخوریم، شرط گذاشته ایم که اگر نوع فرزند از نوع ReadOnlyFile نباشد، متد سیو را ران کنیم. این کاملا بر خلاف اصل لیسکو و اصل اینترفیس سگریشن است. برای رفع مشکل:

کلاس پدر اینک فقط شامل لود فایل است:

```
public class ProjectFile
{
    public string FilePath { get; set; }

    public byte[] FileData { get; set; }

    public virtual void LoadFileData()
    {
        Console.WriteLine("File Loaded.");
    }
```

کلاس WritableFile را ساختیم که زیر کلاس ProjectFile است منتها قابلیت Write را هم اضافه کرده ایم:

```
public class WritableFile : ProjectFile
{
    public void SaveFileData()
    {
        Console.WriteLine("Writable File Saved.");
    }

    public override void LoadFileData()
    {
        Console.WriteLine("Writable File Loaded.");
    }
}
```

حال در اسکریپت اصلی، دو گروه لیست داریم. لیست کلاسهای فقط قابل خواندن و لیست همه کلاسهای خواندن.

```
public class Project
{
    public List<ProjectFile> AllFiles { get; set; }
    public List<WritableFile> WritableFiles { get; set; }
    public void LoadAllFiles()
    {
        foreach (ProjectFile file in AllFiles)
        {
            file.LoadFileData();
        }
    }

    public void SaveAllFiles()
    {
        foreach (WritableFile file in WritableFiles)
        {
            file.SaveFileData();
        }
    }
}
```

حال دیگر بررسی اینکه آیا کلاس مد نظر کلاس ReadOnlyFile است یا نیست را دیگر نداریم و صرفا به سادگی با دو گروه شدن انواع فایل ها، به سادگی متد Read را برای همه و متد Write را برای فقط آنهایی که مجاز به Write هستند کال کردیم.

نکته مهم و مشترک Liskov Subsitution و Interface Segregation: همواره باید کلاسهای فرزندی، کلاسهای پدری را توسعه دهند و به جای اینکه پدران، کل متدهای ممکن را پیاده کنند و فرزندان فقط متدهای لازمشان را از پدرها گسترش دهند و متدهای غیر مجاز و نالازم را با اکسکپشن کور کنند، باید کلاسهای پدر حداقل های مشترک و لازم برای همه فرزندان را پیاده کنند و کلاسهای فرزند متدهای اضافی لازم را اضافه کنند. در واقع باید برعکس اینکه کلیات توسط پدران پیاده شوند و فرزندان، گروهی از متدهای اضافی را حذف کنند، باید پدرها، مشترک های اصلی را پیاده کنند و فرزندان بر اساس ذاتشان، تعداد متدهای مورد نیاز را افزایش دهند. نهایتا با تعریف انواع لیست ها از نوع پدر (مشترک) یا فرزند (خاص منظوره) می توان از نوشتن شروط وابسته ساز جلوگیری کرد.

فرض کنید در مثال کاستومرها بود، می خواهیم متد Delete را هم به اینترفیس IDatabase اضافه کنیم:

```
namespace InterfaceSegregationPrincipleDemo
{
    public interface IDatabase
    {
        void Add();
        void Delete();
    }
}
```

خوب از این به بعد هر کلاسی که بخواهد از این اینترفیس پیاده سازی داشته باشد باید متد Delete را نیز پیاده کند در صورتیکه شاید کلاسهایی باشند که نیازی به Delete کردن نداشته باشند. پس این کار اصلا کار جالبی نیست چون همه کلاسهای پیاده سازی شده از IDatabase را مجبور کرده ایم متد Delete را هم پیاده کنند در صورتیکه شاید بعضی از آنها نیازی به این متد نداشته باشند یا اصلا نباید آنرا پیاده کنند.

در این حالت اینترفیس دیگری را تعریف می کنیم و آنرا زیر کلاس یا مشتقی از اینترفیس IDatabase می کنیم:

```
namespace InterfaceSegregationPrincipleDemo
{
    public interface IDatabase
    {
        void Add();
    }

    public interface IDatabaseDelete : IDatabase
    {
        void Delete();
    }
}
```

به این شکل هر کلاس پیاده سازی شده از IDatabase حتما باید متد Add را پیاده سازی کند ولی اگر کلاس هایی باشند که بخواهند Delete را نیز پیاده سازی کنند، می توانند از اینترفیس IDatabaseDelete پیاده سازی شوند.

در شکل زیر ما یک شی از نوع کلاس کاستومر ساختیم و آنرا در اشاره گر اینترفیس کاستومر ریختیم. کلاس دیگری که از IDatabaseDelete پیاده سازی می شد به نام CustomerWithDelete ساختیم و نمونه ساخته شده از آنرا در متغیر IDatabaseDelete ریختیم. حالا مشاهده می کنید که کاستومر فقط متد Add را دارد ولی شی دوم که در اشاره گر اینترفیس IDatabaseDelete ریخته شده، هم Add را دارد هم Delete را.

```
IDatabase customer = new Customer();
IDatabaseDelete newCustomer=new CustomerWithDelete()

customer.Add();
newCustomer.Delete();
newCustomer.Add();
Console.ReadKey();
```

مثال دیگر Interface segregation:

کلاس Contact را مشاهده می کنید که مشخصات تماسی کاربران را دارد:

```
namespace InterfaceSegregationPrincipleDemo
{
    public class Contact
    {
        public string Name { get; set; }
        public string Address { get; set; }
        public string EmailAddress { get; set; }
        public string Telephone { get; set; }
    }
}
```

کلاس دیگری به نام Emailer داریم که مشخصات ایمیل را می گیرد تا ایمیل ارسال کند:

```
blic class Emailer

  public void SendMessage(Contact contact, string subject, string body)
  {
      // Code to send email, using contact's email address and name
  }
```

کلاس سوم ما شیی از نوع Contact می گیرد تا تماس تلفنی بگیرد:

```
namespace InterfaceSegregationPrincipleDemo
{
    public class Dialler
    {
        public void MakeCall(Contact contact)
        {
            // Code to dial telephone number of contact
        }
    }
}
```

ولی این ساختار از نظر interface segregation ایرادات فاحش دارد. چون کلاس Contact شامل ایمیل و تلفن است در صورتیکه شاید بعضی افراد باشند که ایمیل یا شماره تلفن نداشته باشند و این موارد نباید مجبور باشند هر دو را در کنار هم داشته باشند. از همه مهمتر ما یک داده کامل را به دو متد پاس می دهیم که هر کدام فقط بخشی از داده را لازم دارند و نه تمام آنرا! برای حل این مشکل ما داده را به دو نوع مختلف می شکنیم! یکی برای مشخصات ایمیل و یکی برای مشخصات تلفن!

برای حل این مشکل، این اینترفیس ها را هم جدا کردیم.

```
public interface IEmailable
{
        string Name { get; set; }
        string EmailAddress { get; set; }
}
```

```
public interface IDialable
{
        string Telephone { get; set; }
}
```

حال کلاس Contact ما از هر دو اینترفیس پیاده سازی می شود:

```
public class Contact:IDialable,IEmailable
{
    public string Name { get; set; }
    public string Address { get; set; }
    public string EmailAddress { get; set; }
    public string Telephone { get; set; }
}
```

حال می توانید کلاسی مثل کلاس مهندس را تعریف کنید که از اینترفیس تماس تلفنی پیاده سازی شود!

```
namespace InterfaceSegregationPrincipleDemo
{
    class Engineer : IDialable
    {
        public string Name { get; set; }
        public string Telephone { get; set; }
    }
}
```

نهایتا ورودی هر دو کلاس Emailer و Dialler را تغییر می دهیم:

```
public class Dialler
{
    public void MakeCall(IDialable contact)
    {
        // Code to dial telephone number of contact
    }
}
```

```
blic class Emailer

  public void SendMessage(    IEmailable Contact    , string subject, string body)
  {
      // Code to send email, using contact's email address and name
  }
```

حال مشاهده می فرمایید که به ورودی هر دو کلاس Emailer و Dialler می توان هم ورودی از نوع Contact داد و هم از نوع تخصصی خودشان. این امکان باعث خواهد شد حتما مجبور نباشیم داده سنگین تر را به هر دو کلاس بدهیم.

```
Dialler d=new Dialler();
d.MakeCall(new Contact());
d.MakeCall(new Engineer());
```

اصل 5 – Dependency Inversion Principle یا DIP (اصل وارونه کردن وابستگی)

هر کلاس باید به انتزاعات وابسته باشد نه به جزئیات .یعنی باید از وابستگی مستقیم به کلاس های دیگر خودداری کنیم و به جای آن از رابط ها یا کلاس های انتزاعی استفاده کنیم. با این کار، تمام وابستگی‌های بین کلاس‌ها جدا شده و به‌صورت جداگانه به هم پیوند می خورند، به‌طوری‌که تغییر در یک کلاس اثرات اصلی بر سایر کلاس‌ها نداشته باشد! این اصل به Inversion of Control (IOC) هم معروف است. این اصل تلاش می کند کاری کند برنامه بر اساس قرارداد کار کند نه کد پیاده‌سازی.
اگر قرار است یک کلاس وابسته به یک کلاس دیگر شود و شامل آن باشد، وقتی در کلاس مستقل تغییری ایجاد شود، کلاس وابسته هم مجبورند در بسیاری شرایط همان تغییرات را ایجاد کنند و برای رفع این مشکل بهتر است برای اصل کار، اینترفیسی تعریف شود و کلاسهای مستقل از آن اینترفیس implement یا پیاده سازی داشته باشند!

فرض کنید ما کلاس ارتباط با دیتابیس ساختیم و متد های Insert و delete و select را تعریف کردیم. اگر قرار باشد به جای دیتابیس sql از sqlite استفاده شود مجبوریم کدهای را در کلاس وابسته و مستقل تغییر دهیم. ولی با استفاده از این اصل کافیست برای ارتباط با دیتابیس یک interface تعریف کرده و کلاس مستقل را از آن پیاده سازی کرده و در کلاس وابسته به جای اینکه یک نمونه از کلاس مستقل بسازیم، یک نمونه از اینترفیس می سازیم! سپس شی ساخته شده از نوع مستقل را از بیرون به کانسترکتور کلاس وابسته پاس می دهیم و ورودی این کانسترکتور چیزی جز یک Interface نیست! لذا کلاس وابسته، وابسته به اینترفیس است نه وابسته به اصل کلاس! در اصل اینترفیس تغییر نمی کند ولی می توان هزاران کلاس برای آن اینترفیس بر اساس نیاز بوجود آورد و کد کلاس وابسته این بار وابسته به اینترفیس است نه اصل کلاس مستقل! با این روند هر تغییری در کلاس مستقل اعمال کنید، کلاس وابسته نیازی به منتشر کردن تغییرات ندارند!

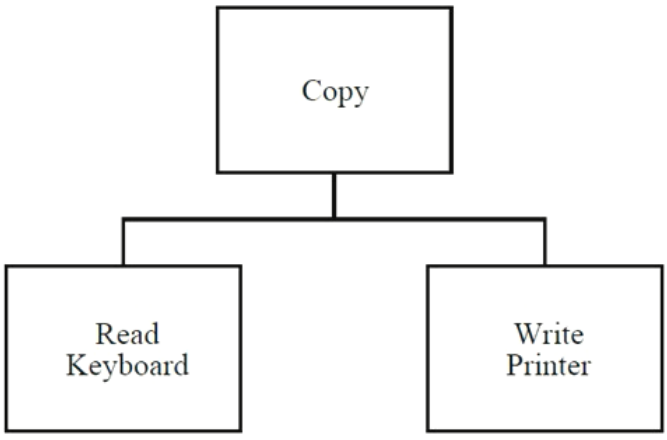
طراحی بد یک برنامه هر چند اگر در نسخه نهایی و پایدار خاصش به درستی کار کند باعث می شود با افزودن امکانات جدید، به هم بریزد. برای جلوگیری از به هم ریختگی و برای طراحی خوب نرم افزار، باید چند اصل را در نظر بگیریم و از آنها دوری کنیم:

الف- Rigidity: شما نتوانید در برنامه به سادگی تغییر ایجاد کنید و ایجاد تغییر نیاز به تغییر در بسیاری از بخش های برنامه داشته باشد.

ب- Fragility: سست بودن برنامه یعنی با کمترین تغییر کل برنامه به هم بریزد.

ج- Immobility: یعنی برنامه قابل استفاده در برنامه های دیگر نباشد و نتوانیم به سادگی بخش های مختلف برنامه را به پروژه های دیگر منتقل کنیم.

فرض کنید یک اپلیکیشن طراحی می کنید که به سادگی از کیبورد داده را می گیرد و مستقیما در پرینتر روی کاغذ مستقیما چاپ می کند.




```
void Copy()
{
  int c;
  while ((c = ReadKeyboard()) != EOF)
    WritePrinter(c);
}
```

این کد ایرادات زیادی دارد. این کد قابل حمل به پروژه های دیگر نیست چون تابع Copy وابسته به دو تابع خواندن از کیبورد و پرینت روی کاغذ است.

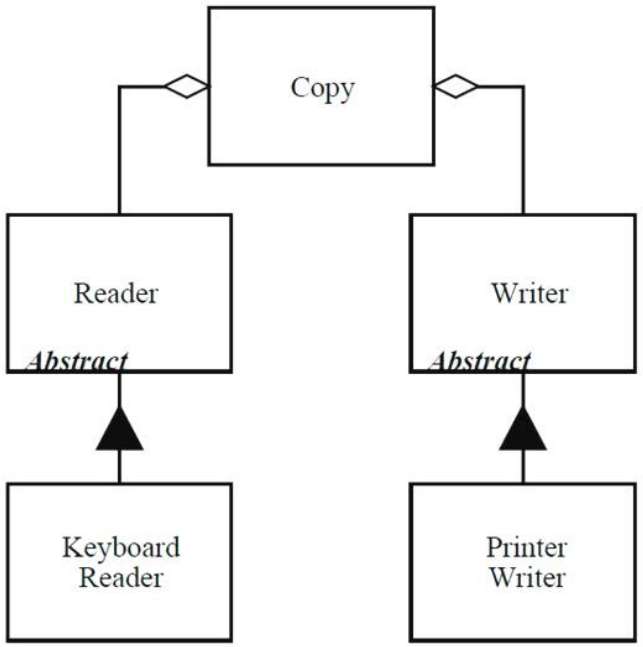
روش های دیگری هم می توان پیشنهاد داد:

```
enum OutputDevice {printer, disk};
void Copy(outputDevice dev)
{
  int c;
  while ((c = ReadKeyboard()) != EOF)
    if (dev == printer)
      WritePrinter(c);
    else
      WriteDisk(c);
}
```

در این نمونه اصل Open-Close به درستی رعایت نشده و با هر اضافه شدن outputDevice باید دوباره این متد تغییر کند و حالات دیگر به آن اضافه شود.

روش صحیح حل آن این روش است:

```
class Reader
{
  public:
    virtual int Read() = 0;
};

class Writer
{
  public:
    virtual void Write(char) = 0;
};

void Copy(Reader& r, Writer& w)
{
  int c;
  while((c=r.Read()) != EOF)
    w.Write(c);
}
```



خوب با این روش ما آمدیم و ذات کیبورد و ذات پرینتر را ورودی و خروجی دیدیم. پس یک کلاس ابسترکت برای Reader و یک کلاس ابسترکت برای Writer تعریف کردیم. پرینتر را زیرکلاس Writer و کیبورد را زیرکلاس Reader تعریف کردیم. متد کپی هم یک ورودی Reader و یک ورودی Writer می گیرد و تا وقتی که Reader چیزی برای ورود داشته باشد و داده های خوانده شده اش تمام نشود، از آن می خواند و در Writer آنرا با متد write می نویسد. به این صورت وابستگی برعکس شد. قبلا هر اضافه شدن یا تغییر در read و write و ... نیاز به تغییر در متد Copy داشت. اینک دیگر اگر تغییراتی در خواندن و نوشتن ایجاد شود، در متد کپی ایجاد نمی شود. پس اگر قبلا کپی وابسته به نحوه خواندن و نوشتن بود، اینک متد copy و Printer writer و keyboard reader وابسته به ابسترکشن های Read و Write شده اند پس وابستگی معکوس شده و همه رفته سمت ابسترکت های وسط.

در مثال بعدی کلاس کاستومر می خواهد عدد i را اد کند و اگر به اکسپشن بخورد باید لاگ مناسب را چاپ کند. می توان به روش های مختلف خطا را پرینت کرد ولی بهترین روش برای اعمال سالید آنست که یک اینترفیس ILogger تعریف کنیم که متد handle دارد با دریافت مسیج خطا، آنرا به روش مطلوبش نمایش دهد. پس اینجا هم وابستگی را معکوس کردیم تا متد Add وابسته به ابسترکت ILogger باشد نه برعکس/

```
private ILogger obj;

public Customer(ILogger ilogger)
{
    obj = ilogger;
}
public virtual void Add(int i)
{
    try
    {
        // Database code goes here
    }
    catch (Exception ex)
    {
        obj.Handle(ex.ToString());
    }
}
```

در مثال آخر از DIP فرض کنید کلاس اکانت با داشتن شماره حساب و میزان دارایی و متد های واریز و برداشت را داریم:

```
public class BankAccount
{
    public string AccountNumber { get; set; }

    public decimal Balance { get; set; }

    public void AddFunds(decimal value)
    {
        Balance += value;
    }

    public void RemoveFunds(decimal value)
    {
        Balance -= value;
    }
```

کلاس دیگری داریم که قرار است دو شماره حساب را داشته باشید تا موقع واریز از یک حساب برداشت و به حساب دیگر واریز کند:

```
public class TransferManager
{
    public BankAccount Source { get; set; }

    public BankAccount Destination { get; set; }

    public decimal Value { get; set; }

    public void Transfer()
    {
        Source.RemoveFunds(Value);
        Destination.AddFunds(Value);
    }
}
```

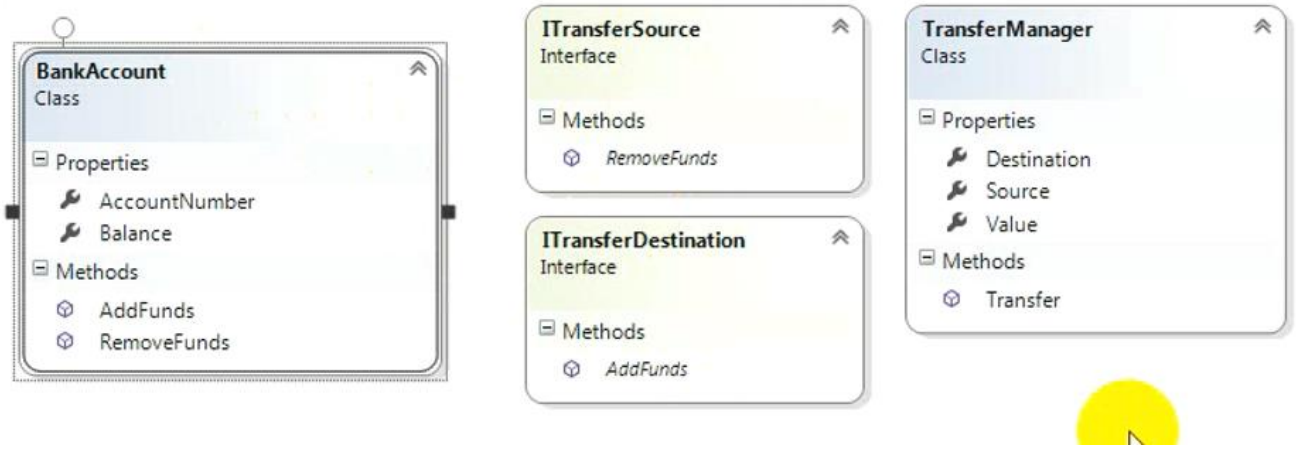
حالا فرض کنید یکی از انواع حساب های جدید، همانند حساب های یک طرفه دریافت قسط و وام، گزینه پرداخت یا واریز ندارد. اینجا باید دو اینترفیس بوجود بیاوریم یکی برای مبدا و منبع برداشت و دیگری برای مقصد واریز. کلاس BankAccount را از هر دو implement کردیم:

```
public interface ITransferSource
{
    void RemoveFunds(decimal value);
}

public interface ITransferDestination
{
    void AddFunds(decimal value);
}
public class BankAccount:ITransferDestination,ITransferSource
{
    public string AccountNumber { get; set; }

    public decimal Balance { get; set; }

    public void AddFunds(decimal value)
    {
        Balance += value;
    }

    public void RemoveFunds(decimal value)
    {
        Balance -= value;
    }
```

در نهایت در TransferManager همیشه حسابی که واریز می کند و از آن برداشت می شود را از اینترفیس جدید اول یعنی مبدا و دیگری را از اینترفیس جدید دوم یعنی مقصد طراحی می کنیم:

```
public class TransferManager
{
    public ITransferSource Source { get; set; }

    public ITransferDestination Destination { get; set; }

    public decimal Value { get; set; }

    public void Transfer()
    {
        Source.RemoveFunds(Value);
        Destination.AddFunds(Value);
    }
}
```

حالا همینطوری که می بینید اگر پرداخت وام یا هر حساب یک طرفه واریز کننده و پرداخت کننده هم که اضافه شود کافیست از اینترفیس مناسب پیاده سازی کند تا اصول مختلف سالید را حفظ کند! در اینجا مشاهده می کنید که دیگر کلاسهای پرداخت و اکانت به هم وابسته نیستند بلکه به اینترفیس ها وابسته هستند! لذا اگر بخواهید کلاس اکانت یا کلاس پرداخت را به تنهایی جای دیگری ببرید باید با اینترفیس های مورد نیازشان ببرید و دیگر کد کلاسها به هم وابسته نیستند!



یک مثال ساده از پیاده‌سازی DIP در C++ به صورت زیر است:

فرض کنید یک برنامه ساده را برای محاسبه جدول ضرب در نظر بگیرید. برای پیاده‌سازی این برنامه، یک کلاس MatrixCalculator وجود دارد که دارای دو متد است که برای محاسبه جدول ضرب به کار می‌روند:

print و multiply

ابتدا یک واسط مشترک بین MatrixCalculator و کلاس‌های مختلف جدول تعریف می‌کنیم:

```
class IMatrix {
public:
          virtual void multiply() = 0;
          virtual void print() = 0;
};
```

سپس دو کلاس Matrix2x2 و Matrix3x3 را برای پیاده‌سازی واسط IMatrix تعریف می‌کنیم:

```
class Matrix2x2 : public IMatrix {
public:
        void multiply() {
              // implementation for 2x2 matrix multiplication
        }
        void print() {
              // implementation for 2x2 matrix printing
        }
};
```

```
class Matrix3x3 : public IMatrix {
public:
         void multiply() {
                // implementation for 3x3 matrix multiplication
         }
         void print() {
                // implementation for 3x3 matrix printing
         }
};
```

حالا برای پیاده سازی صحیح DIP از Solid، کلاس MatrixCalculator را به گونه‌ای تغییر می‌دهیم که به جای استفاده از پیاده‌سازی خاص هر کلاس، از واسط IMatrix استفاده کند:

```
class MatrixCalculator {
private:
             IMatrix* matrix;
public:
            MatrixCalculator(IMatrix* m) : matrix(m) {}

            void multiply() {
                     matrix->multiply();
            }

            void print() {
                     matrix->print();
            }
};
```

بنابراین حالا می‌توانیم کلاس MatrixCalculator را به صورت زیر استفاده کنیم:

```
IMatrix* m2x2 = new Matrix2x2();
MatrixCalculator calculator2x2(m2x2);
calculator2x2.multiply();
calculator2x2.print();
IMatrix* m3x3 = new Matrix3x3();
MatrixCalculator calculator3x3(m3x3);
calculator3x3.multiply();
calculator3x3.print();
```

با این رویکرد، هر زمان که یک کلاس جدید با پیاده‌سازی واسط IMatrix ایجاد شود، می‌توانیم آن را به کلاس MatrixCalculator اضافه کرده و آن را بدون تغییر در کد MatrixCalculator استفاده کنیم. همچنین وابستگی MatrixCalculator به کلاس‌های Matrix2x2 و Matrix3x3 برای انجام کارش کاهش یافته است.

در مثال کد جاوای زیر، اصل Dependency Inversion را رعایت نکرده ایم چون کلاس UserService را وابسته به کلاس FileLogger کرده ایم:

Java

```
// Violates DIP
public class FileLogger {
    public void log(String message) {
        // write message to a file
    }
}

public class UserService {
    private FileLogger logger;

    public UserService() {
        this.logger = new FileLogger(); // depends on a concrete class
    }

    public void createUser(String name) {
        // create a user
        logger.log("User created: " + name); // tight coupling
    }
}
```

برای رفع مشکل و رعایت Dependency Inversion می توانیم در ابتدا یک اینترفیس تعریف کنیم تا کلاس UserService را وابسته به آن کنیم و کلاس FileLogger را نیز از آن Implement کنیم!

```
// Follows DIP
public interface Logger {
    void log(String message);
}

public class FileLogger implements Logger {
    @Override
    public void log(String message) {
        // write message to a file
    }
}

public class ConsoleLogger implements Logger {
    @Override
    public void log(String message) {
        // write message to the console
    }
}
```

```
public class UserService {
    private Logger logger;

    public UserService(Logger logger) {
        this.logger = logger; // depends on an abstraction
    }

    public void createUser(String name) {
        // create a user
        logger.log("User created: " + name); // loose coupling
    }
}
```

پایان

هادی عباسی

Hadi.abbasi.programmer@gmail.com