# SOLID

## ال**SOLD**

هي كلمه مختصره لخمس مبادئ اساسيه بطتبيق هذه المبادئ تجعل الكود قابل للتوسع والتحديث والاختبار

حرف ال **S** ==>**مبدا المسئوليه الواحده** **Single Responsibility Principle**

حرف ال **O** ==> **مبدا الفتح والاغلاق** **Open-Close Principle**

حرف ال **L** ==>**مبدا الاستبدال** **Liskov Substitution Principle**

حرف ال **I** ==>**مبدا فصل الواجهات** **Interface segregation Principle**

حرف ال **D** ==>**مبدا انعكاس التبعيه** **Dependency Inversion Principle**

*قبل ما اوضح الخمس مبادئ دول هوضح ليك الفرق بين ال **Design Pattern** &**Design Principle**

أولا :Design Principle

هي مجموعه مبادئ وقواعد توضح لنا شكل الكود علشان الكود في النهايه يكون

Extensible, Maintainable , Testable

هذه المبادئ بتجنبنا من العاده السيئه بتاعت التركيز علي الوصل للحل دون الاخد في الاعتبار جوده الحل

ثانيا :Design Pattern

هي حلول عمليه وفعليه تم بناؤها علشان تحل مشكله برمجيه معينه

الغرض الاساسي منها تقديم حلول برمجيه لمشاكل شائعه

***الفرق بين ال**Design Pattern** &**Design Principle**

ذي الفرق بين الحاجه ال بيقولها المهندس والحاجه ال بيفعلها البناء

المهندس بيعطي التعليمات والتوجهات علشان البناء يكون طبقا لمعايير هندسيه معينه Design principle

البناء بينفذ الحلول الفعليه علشان تتم عمليه البناءDesign Pattern

الاربع مصطلحات دول مهمين **Concrete** ,**Abstract** , **Cohesion** , **Coupling**

ال **Coupling**الارتباط

لما يكون في كلاس بيستعمل اوبجكت من كلاس اخر بنقول ان الكلاسين دول مترابطين ممكن الارتباط يكون ضعيفا:**Loose Coupling**

يكون قويا:**Tight Coupling**

*كلما كان الارتباط بين الكلاس وبعضها قوي كلما كان الكود صعب الصيانه والتعديل

ولو الارتباط ضعيف بيكون افضل لاننا بنقلل الاعتماديه

ال **Cohesion**التماسك

هو بيعكس العلاقه بين المكونات

يعني مينفعش الاقي كلاس للموظفين وفي نفس الكلاس بيانات الشركه وبيانات تسجيل الدخول كدا المكونات غير مترابطه ببعض ....لازم احط الخصائص ال شبه بعض في كلاس واحد

مثال:

الكلاس ده كلاس غير متماسك لانه اشتمل علي بيانات عير متماسكه مينفعش كلاس الموظف يحتوي علي بيانات الشركه وحاله المستخدم

Employee.cs:

```
public class Employee
{
    public string Name { get; set; }
    public string Company { get; set; }
    public DateTime DateOfBirth { get; set; }
    public bool IsLocked { get; set; }

    public string GetEmployeeName() => this.Name;

    public int Age() => DateTime.Now.Year - DateOfBirth.Year;

    public bool EmployeeLoginState() => this.IsLocked;
}
```

ممكن التعديل علي الكلاس ده ليصبح كلاس غير متماسك كالاتي:

Employee.cs:

```
public class Employee
{
    public string Name { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string GetEmployeeName() => this.Name;
    public int Age() => DateTime.Now.Year - DateOfBirth.Year;
}
```

Company.cs:

```
public class Company
{
    public string Name { get; set; }
    public string Address { get; set; }
}
```

User.cs:

```
public class User
{
    public string UserName { get; set; }
    public string Password { get; set; }
    public bool IsLocked { get; set; }
}
```

التجريد **Abstract** & الواقعيه **Concrete**

هوضح المفهومين دول بمثال لما اقول حيوان ده حاجه مجرده انت دلوقت مش تعرف اي حيوان اقصده كدا يعتبر Abstract

لاكن لما اقول قطه اوكلب كدا انت عرفت عن اي حيوان اتكلم ده يعتبر Concrete

* لما يكون هناك كلاس في الخصائص الخاصه بتاعته واقدر اخد منه اوبجكت كدا انا بتكلم عن حاجه واقعيهConcrete

*لما اتكلم عن حاجه مجرده لا يحتوي علي اي تفاصيل ذي ال Abstract &Interfaceكدا بتكلم عن

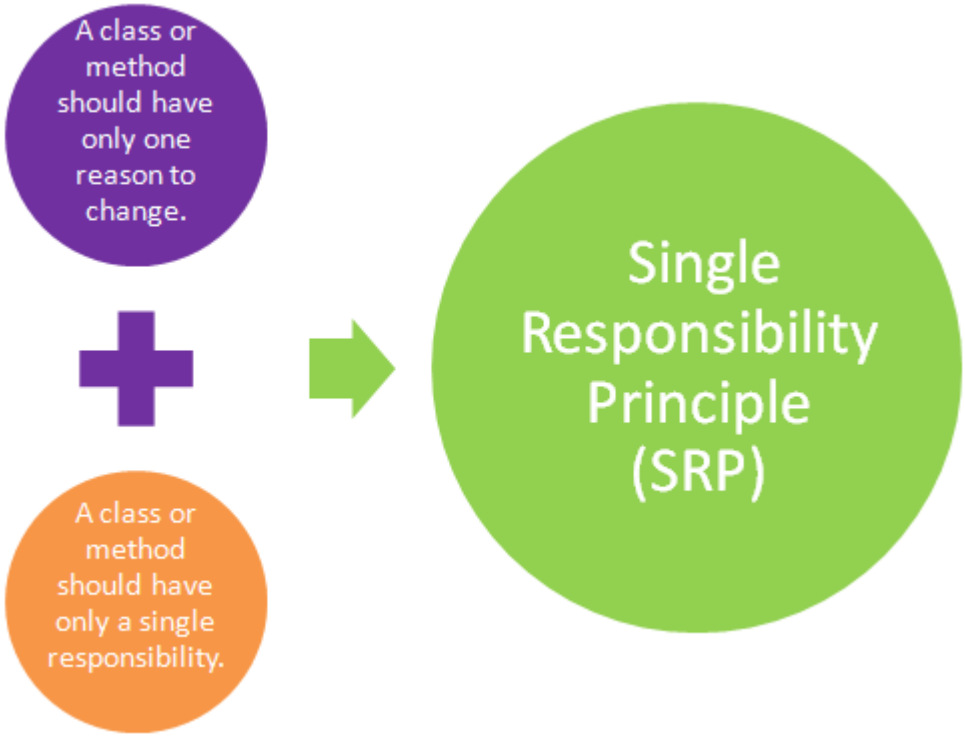
حاجه مجردهAbstract

## المبدأ الاول: Single Responsibility

معناها ان كل Function اوClassاو Modelله مسئوليه واحده فقط يقوم بيها وله سبب واحد فقط في التغيير

مثلا المحاسب هو ال بيقوم بعمليه الحسابات في الشركه (مسئوليه واحده )غير مسئول مثلا عن التسويق

مبدأ المسئوليه الواحده بيجعل الكلاس اكثر تماسك Cohesive




## متي نحتاج الي هذا المبدا (مبدا المسئوليه الواحده)؟

لما الاقي كلاس معين يقوم باكثر من مسئوليه في نفس الوقت.

ذي مثلا كلاس واحد يقوم بقراءه البيانات وحفظ البيانات في قاعده البيانات والتحقق من سلامه البيانات

ده عباره عن انذار لاستخدام هذا المبدا واقسم الكلاس ده الي كلاسات صغيره كل كلاس مسئول عن مسئوليه واحده فقط

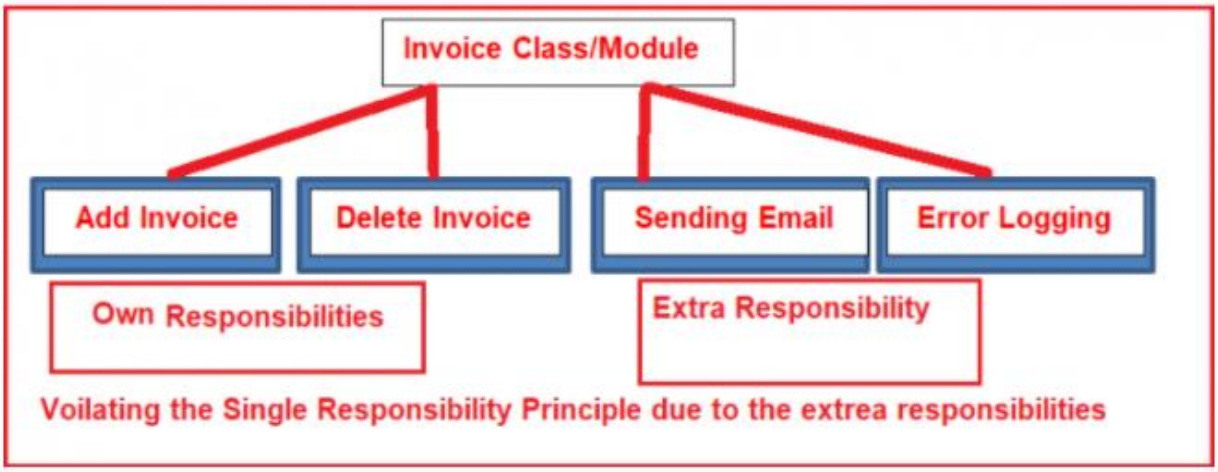
الحل

هعمل كلاس يكون مسئول عن قراءه البيانات....وكلاس مسئول عن حفظ البيانات في قاعده البيانات وكلاس مسئول عن التحقق من سلامه البيانات وكدا انا طبقت مبدا Single Responsibility

مثال:لوهتعمل فاتوره الكترونيه مثلا

هنا هتلاقيه بينتهك مبدا Single Responsibility Principleلان هتلاقي كلاس الفاتوره موجود فيه

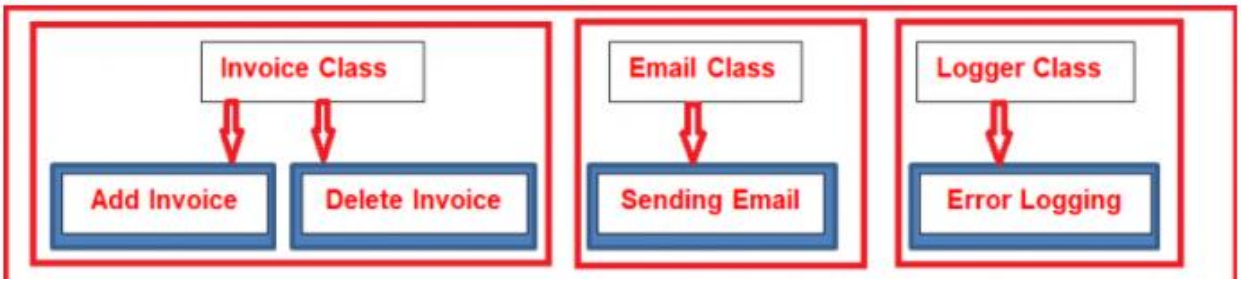
اربع وظائف وهي اضافه الفاتوره وحذف الفاتوره بالاضافه كمان الي ارسال الرسائل وتسجيل الاخطاء هنلاقي ارسال رسال البريد وتسجيل الاخطاء ليست من مسئوليه الفاتور




الحل علشان نطبق ال Single Responsibility

هنعمل تلاته كلاس كلاس الفاتوره هنحط فيه الحاجات ال متعلقه بالفاتوره

وكلاس لارسال الرسائل .....وكلاس لتصحيح الاخطاء




خلاص مبدأ المسئوليه الواحده Single Responsibility (SRP)

علي الكلاس الواحد القيام بمسئوليه واحده فقط

## المبدأ الثاني: Open-Close Principle

معناه ان بيسمح بالاضافه وغير مسموح بالتعديل

مسموح بالتوسع والاضافه Open→Extended

غير مسموح بالتعديل علي الكود الحالي Close→Modified

*ممكن استخدامه هذا المبدا باستخدام ال Interface & Abstract لانه بيسمحوا اننا نضيف علي الكود الموجود بداخله

**مثال** :هذا الكود فيه عندنا نوعين من انواع الفاتوره وهما ال**FinalInvoice , ProposedInvoice**

لواحتاجنا اننا نضيف نوع جديد من الفاتوره هنحتاج نضيف else if في هذا الكود وهذا يخترق مبدا ال OCP

```
public class Invoice
{

    public double GetInvoiceDiscount(double amount, InvoiceType invoiceType)
    {
        double finalAmount = 0;

        if (invoiceType == InvoiceType.FinalInvoice)
        {
            finalAmount = amount - 100;
        }
        else if (invoiceType == InvoiceType.ProposedInvoice)
        {
            finalAmount = amount - 50;
        }
        return finalAmount;
    }

}
public enum InvoiceType
{
    FinalInvoice,
    ProposedInvoice
};
```

If one more Invoice Type comes then we need to add another else if condition within the source code of the above GetInvoiceDiscount() method which violates the Open Closed Priciple in C#

**الحل**: اننا نعمل Interface or Abstract or Override ونحط فيه الداله ال عوزين نغير الBody بتاعها وبكدا قدرنا اننا تضيف علي الكود دون ان نغير في الكود

```
public class Invoice
{
    6 references
    public virtual double GetInvoiceDiscount(double amount)
    {
        return amount - 10;
    }
}

public class FinalInvoice : Invoice
{
    4 references
    public override double GetInvoiceDiscount(double amount)
    {
        return base.GetInvoiceDiscount(amount) - 50;
    }
}

0 rererences
public class ProposedInvoice : Invoice
{
    4 references
    public override double GetInvoiceDiscount(double amount)
    {
        return base.GetInvoiceDiscount(amount) - 40;
    }
}

public class RecurringInvoice : Invoice
{
    4 references
    public override double GetInvoiceDiscount(double amount)
    {
        return base.GetInvoiceDiscount(amount) - 30;
    }
}
```

## المبدأ الثالث:Liskov Substitution Principle

لو عند كلاس للاب وكلاس للابن اقدر اتبادل الادوار بنهم من غير ما ابوظ البرنامج

لو عندي اتنين كلاس S ,كلاس Tممكن اخلي الاوبجكت بتاع ال S يساوي الاوبجكت بتاع ال T

Ex:

T t=new T( );

S s=new S( );

ممكن اقول

T t=new T( );

Or

T t=new S( );

معناها لو الاب مش موجود ممكن الابن يقوم بنفس الدور

مثال:المربع هو عباره عن مسطيل متساوي الطول والعرض

مثال:الدائره عباره عن شكل بيضاوي اقطاره متساويه

```
public class Ellipse
{
    3 references
    public double Rx { get; set; }
    4 references
    public double Ry { get; set; }
    2 references
    public bool IsCircle => Rx == Ry;
    0 references
    public void SetRx(double value)
    {
        if (IsCircle)
            Ry = value;
        else

            Rx = value;
    }
0 references
public void SetRy(double value)
{
    if (IsCircle)
    {
        Ry = Rx;
    }
    Ry = value;
}
```

هنا هنلاقي IsCircle سهلت الموضوع بدل ما كنا نعمل داله تحسب مساحه الدائره لوحدها ودالله تانيه تحسب مساحه الشكل البيضاوي لواحده هنا عملناCheckواحده بس لو الاقطار متساويه هتكون دائره غير كدا هتكون شكل بيضاوي هنا مش احتاجنا نعمل داله علشان نثبت ان الشكل Ellipse

Program.cs:

```
Ellipse ellipse = new Ellipse();

ellipse.SetRx(5);
ellipse.SetRy(4);

AreaCalculator calculator = new AreaCalculator();

// Rx = 5; Ry = 4; Rx != Ry => IsCircle = False
double result = calculator.Area(ellipse);

ellipse.SetRx(5);
ellipse.SetRy(5);

// Rx = Ry = 5 => IsCircle = True
result = calculator.Area(ellipse);
```

هنا لما كان الX,Y متساويين كان الشكل دائره.غير كدا هيكون شكل بيضاوي

## المبدأ الرابع: Interface segregation Principle

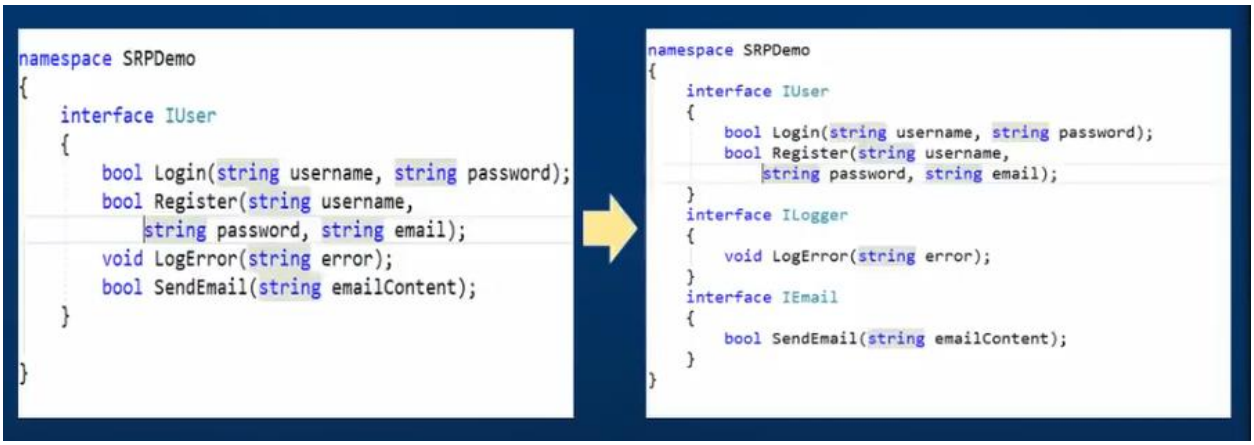
مخليش مثلا كلاس اضع فيه دوال Non Implemented هو مش المفروض يستخدمها

مثال: انا لما يكون عندي Iterfaceطبيعي لما اي كلاس مثلا يورث منه لازم يطبق كل الدوال ال داخله طيب ممكن الكلاس مش عاوز يطبق كل الدوال ال فيه في الحاله دي بنستخدم مبدا ISP

*وهي بنقوم بوضع الدوال ال محتاجها الكلاس في Interfaceلواحده

مثال:في المثال التالي مش لازم كلاس ال Userلما يورث من الIUser Interfaceمش لازم يطبق كل الدوال ال فيه ....علشان كدا قسمنا Interface IUserالي تلاته Interface

IUser , ILogger , IEmaiعلشان نتجنب الNo Implemented Function



```
namespace SRPDemo
{
    interface IUser
    {
        bool Login(string username, string password);
        bool Register(string username,
            string password, string email);
        void LogError(string error);
        bool SendEmail(string emailContent);
    }
}
```

```
namespace SRPDemo
{
    interface IUser
    {
        bool Login(string username, string password);
        bool Register(string username,
            string password, string email);
    }
    interface ILogger
    {
        void LogError(string error);
    }
    interface IEmail
    {
        bool SendEmail(string emailContent);
    }
}
```

## المبدأ الخامس:Dependency Inversion Principle

معناها ان ال High Level Moduleمينفعش يعتمد علي ال Low Level Module لازم يعتمدوا الاتنين علي ال Abstract

مينفعش ال Abstractionيعتمد علي تفاصيل ال Concreteوالعكس صحيح

ما الفرق بين الLow Level Module و ال High Level Module

الModuleال بيعتمد علي Module اخر == > هوHigh Level Module

ClassA.cs:

```
public class ClassA
{
    public void MethodA()
    {
        // Do something
    }
}
```

==>Low Level Module

ClassB.cs:

```
public class ClassB
{
    ClassA a;

    public void MethodB()
    {
        a = new ClassA();
        a.MethodA();
    }
}
```

==>High Level Module

هتلاقي ان كلاس **B**بيحتاج اوبجكت من كلاس **A**اذا **B**هو High Level Module

هنا هنعمل Interface علشان مش نجعل كلاس Bبيعتمد علي كلاس A

/https://www.linkedin.com/in/mona-abdelmonem-35453b216