Introduction To SOLID Principles

Kirtesh Shah

Updated date Sep 24, 2020 22.4k 0 12

Introduction

To design and develop high-quality software, well–established design principles need to be followed. Those five principles make the acronym S-O-L-I-D. That's what this article is all about.

I will explain SOLID principles in this article. Let's start with what the SOLID principle is.

What is SOLID?

SOLID stands for

1. S- Single Responsible Principle (SRP).
2. O- Open Closed Principle (OSP).
3. L- Liskov Substitute Principle (LSP).
4. I- Interface Segregation Principle (ISP).
5. D- Dependency Inversion Principle (DIP).

Michael Feathers introduced the SOLID acronym in the year 2000. SOLID is a design principle that plays a very important role during Object-Oriented design. Software development becomes easy, reusable, flexible, and maintainable using these design principles.

Robert C Martin has promoted SOLID Principles and now it's very famous. It has changed the development approach and dominated in software development industries.

Benefits of SOLID

3. Loosely coupled.
4. Parallel Development.
5. Testability.
6. Code becomes smaller and cleaner
7. Maintainability – Large Systems or Growing systems become complicated and difficult to maintain. This Principle helps us to create a maintainable system. A maintainable system is very important in industries.

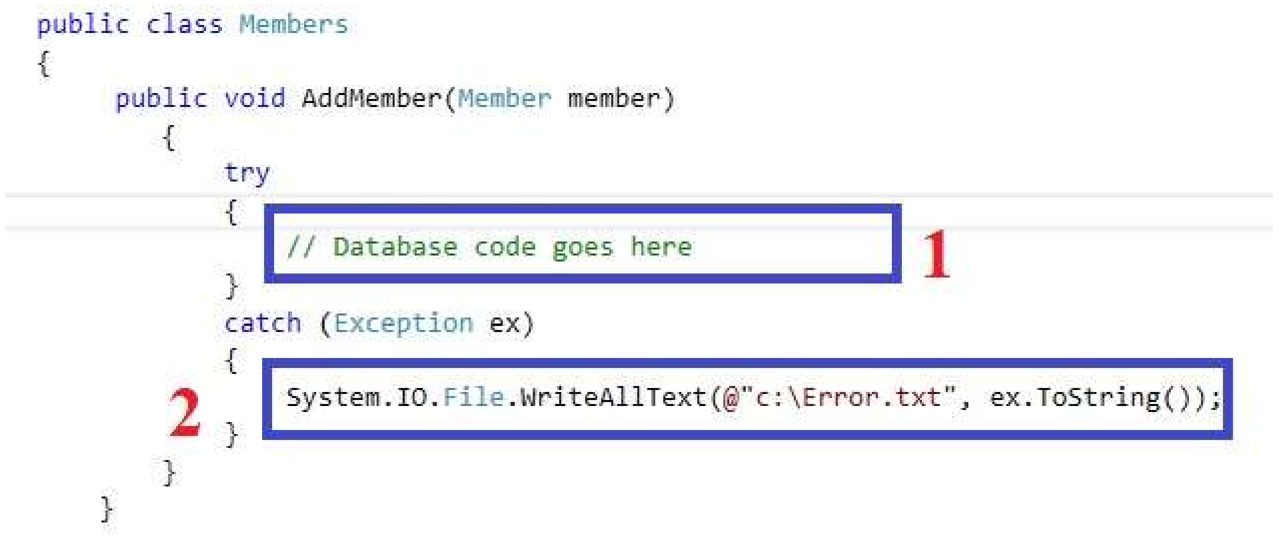
Single Responsible Principle (SRP)

Robert C Martin's original definition is, *"A class should have only one reason to change".*

As the SRP name says, each and every module/class should have single responsibility in software, and that responsibility is encapsulated by the class.

If a single class has multiple responsibilities it may cause problems. To avoid these problems we should separate responsibilities in multiple classes or modules.

Let’s start with a simple example. Suppose we have a company that creates new members and logs error messages.



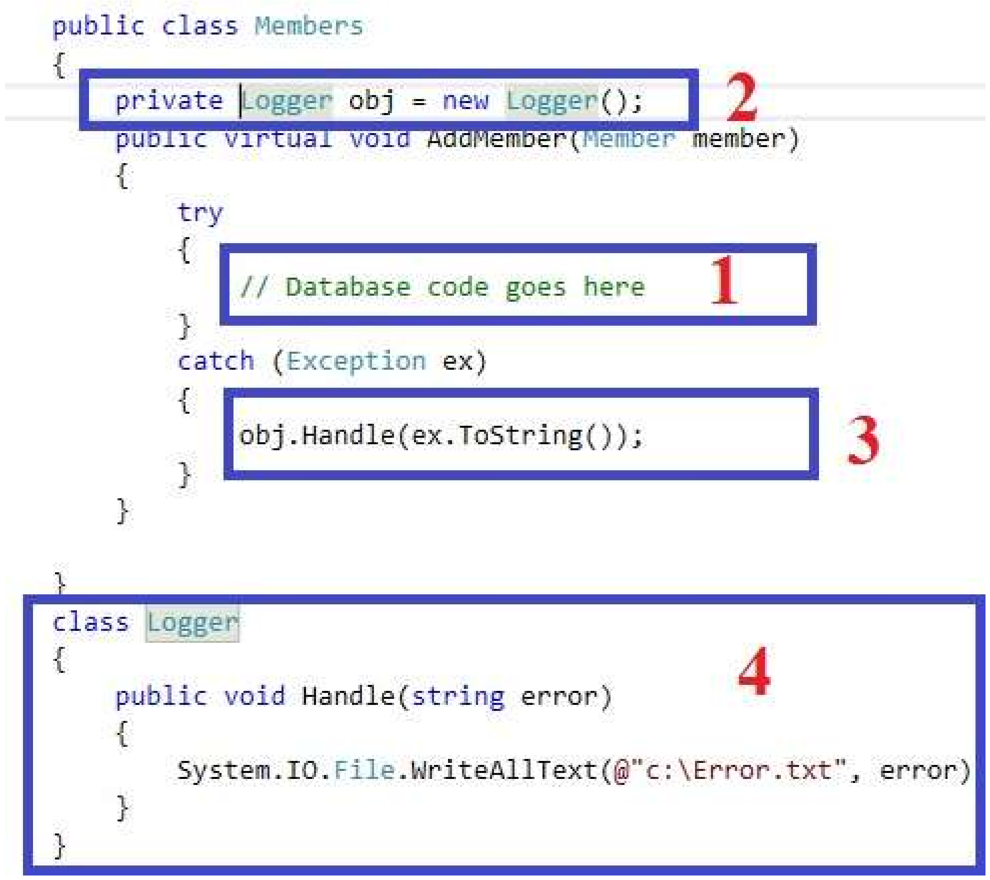
```
public class Members
{
     public void AddMember(Member member)
        {
            try
            {
             // Database code goes here
            }
            catch (Exception ex)
            {
             System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());
            }
  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

What is the code doing? This code has 2 responsibilities

1. Add newly added member in the database
2. Logging activities that log errors in a text file.

What will we be doing if we want to log errors in the event viewer or in the database? We h
to change the logging code in all the classes including member class. This means member

Now we will separate these two responsibilities in 2 separate classes,



```
public class Members
{
    private Logger obj = new Logger();
    public virtual void AddMember(Member member)
    {
        try
        {
            // Database code goes here
        }
        catch (Exception ex)
        {
            obj.Handle(ex.ToString());
        }
    }

}
class Logger
{
    public void Handle(string error)
    {
        System.IO.File.WriteAllText(@"c:\Error.txt", error)
    }
}
```

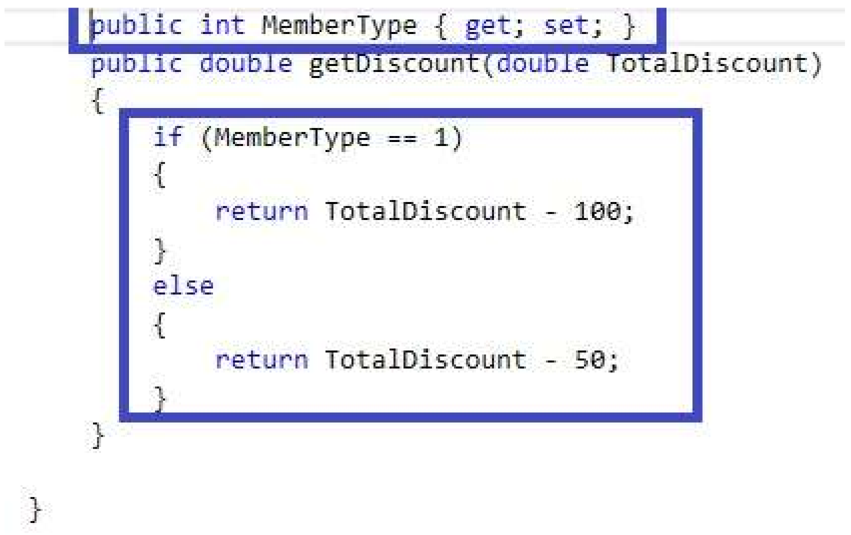
In the above code, two separate classes (Members and Logger) have their own responsibilities, which solved earlier problems in the code and fulfill the single responsibility Principle.

Open Close Principle (OCP)

Bertrand Meyer defines OCP in 1988, *“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”*

Let’s now discuss on what are the problems this principle solved,

Suppose we want to introduce member type and based on the type we want to give/calculate the discount.



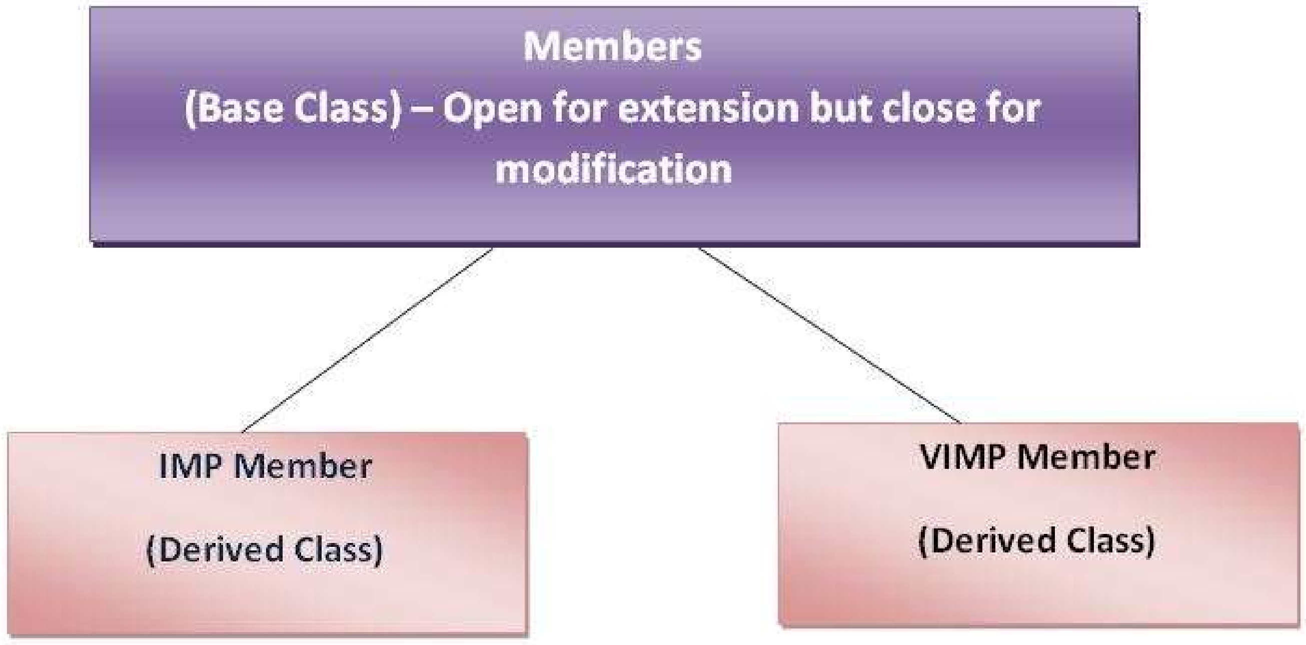
```
    public int MemberType { get; set; }
    public double getDiscount(double TotalDiscount)
    {
        if (MemberType == 1)
        {
            return TotalDiscount - 100;
        }
        else
        {
            return TotalDiscount - 50;
        }
    }

}
```

Let’s think, what is the problem with the above code? If we want to add another member type, we have to change the working member class code and here we use the Open close principle.

OCP says a class should be,

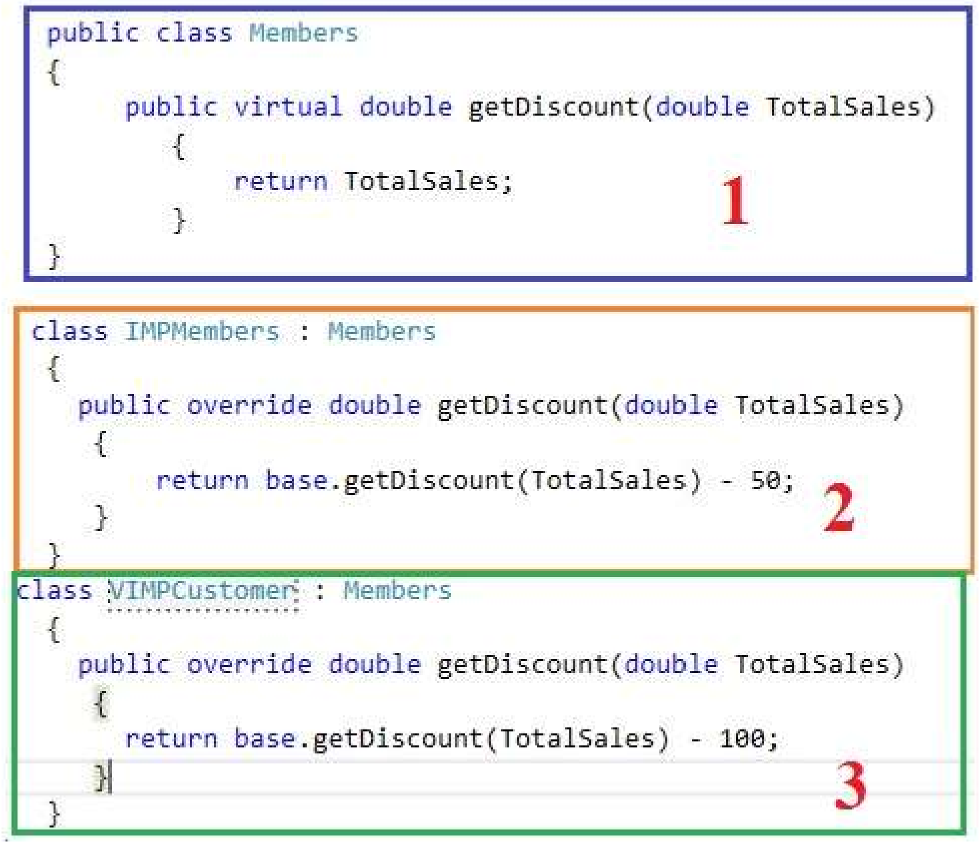
1. Open for extension – means we need to create a base class and that class will be available for extension. This class should have common functionality.
2. Close for Modification – Instead of changing the base class, we will extend the base class and add/modify type-specific coding in the derived class.

Assume that we have two types, if MeberType =1 (IMP Member) and MemberType =2 (VIMP Member) and a base class Members. Let's try to undestand using the below diagram.


Members
(Base Class) – Open for extension but close for modification
IMP Member
(Derived Class)
VIMP Member
(Derived Class)

another derived class.

Let's see the below code,



```
public class Members
{
     public virtual double getDiscount(double TotalSales)
        {
            return TotalSales;
        }
}
```

```
class IMPMembers : Members
 {
   public override double getDiscount(double TotalSales)
    {
        return base.getDiscount(TotalSales) - 50;
    }
 }
```

```
class VIMPCustomer : Members
 {
   public override double getDiscount(double TotalSales)
    {
      return base.getDiscount(TotalSales) - 100;
    }
 }
```

In the above code, 1 is Base Class and 2&3 are Derived classes. Derived class extended base class and modified discount as per type.

So in simple words, the Member base class is closed for modification but open for extensions.

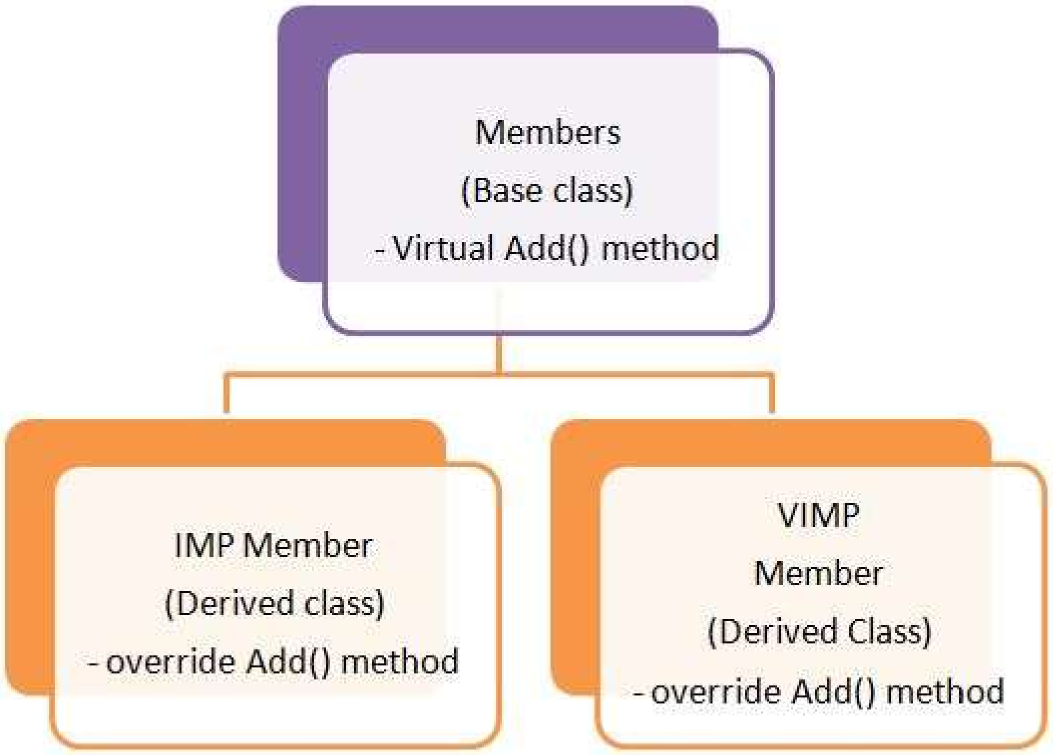
Liskov Substitution Principle (LSK)

Barbara Liskov introduced this principle in 1987 in the conference (Data abstraction and hierarchy) hence it is called the Liskov Substitution Principle (LSK). This principle is just an extension of the Open Close principle.

Robert C Martin also defines this principle. His definition of LSK is *“Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it”.*

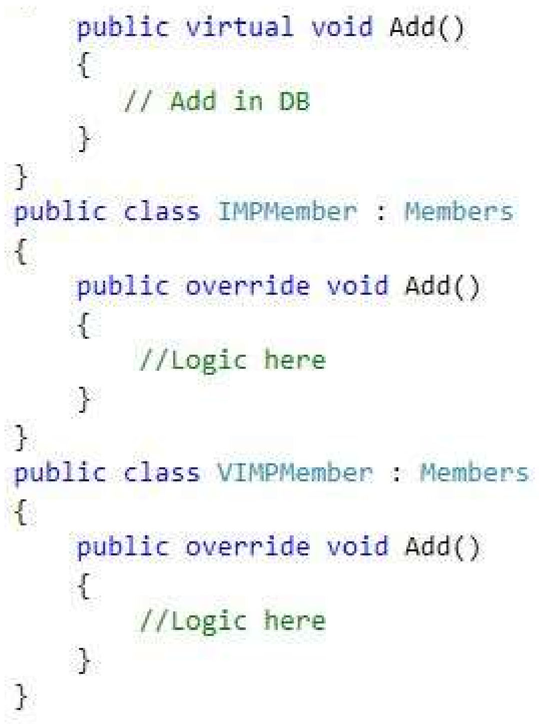
Before we start on LSK, we should have a basic understanding of polymorphism. The polymorphism rule says that during runtime, the parent/base class object can point to any

Suppose we have members class example as per below,



In the above example, we have

1. Base class member which has method Virtual - Add() method to add a member.
2. IMPMember derived class has override - Add() method to add IMPMember.
3. VIMPMember derived class has override- Add() method to add VIMPMember.

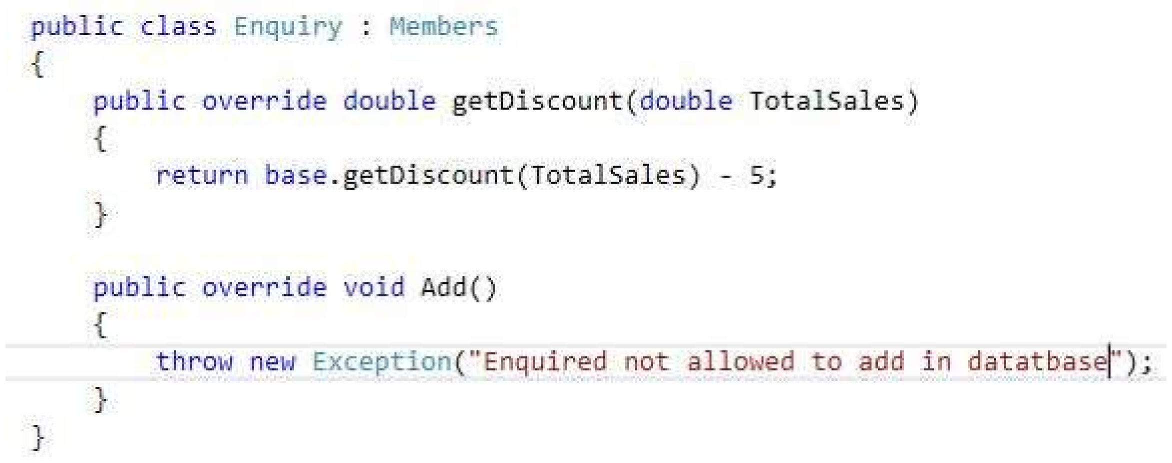


```
    public virtual void Add()
    {
        // Add in DB
    }
}
public class IMPMember : Members
{
    public override void Add()
    {
        //Logic here
    }
}
public class VIMPMember : Members
{
    public override void Add()
    {
        //Logic here
    }
}
```

We will use same class throughout the article now.

Now let’s assume that we need to introduce functionality for Enquiry, which means that anyone can enquire and become a member. A company decided to give a discount to anyone who came for an enquiry, so they can become a member.

Let’s now create a Member class,

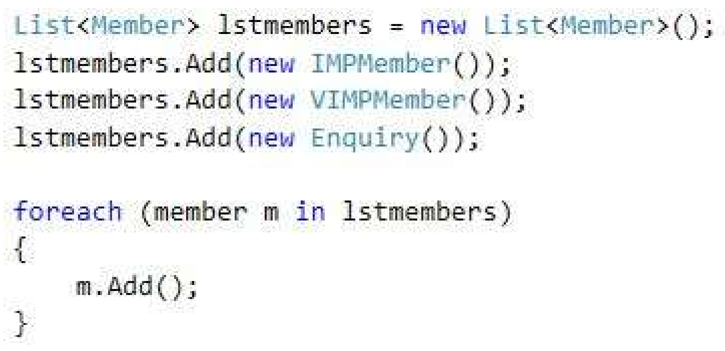


```
public class Enquiry : Members
{
    public override double getDiscount(double TotalSales)
    {
        return base.getDiscount(TotalSales) - 5;
    }

    public override void Add()
    {
        throw new Exception("Enquired not allowed to add in datatbase");
    }
}
```

In the above class example, we have 2 methods:

1. getDiscount
2. Add – Which override from parent member.

inheritance hierarchy.

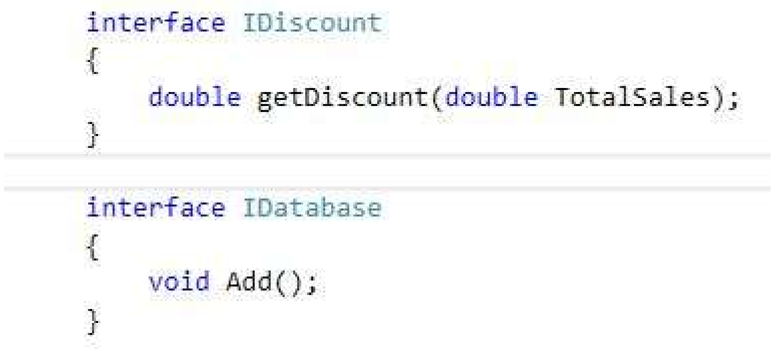


```
List<Member> lstmembers = new List<Member>();
lstmembers.Add(new IMPMember());
lstmembers.Add(new VIMPMember());
lstmembers.Add(new Enquiry());

foreach (member m in lstmembers)
{
    m.Add();
}
```

The above code throws an error as the Enquiry class doesn’t have an Add method.

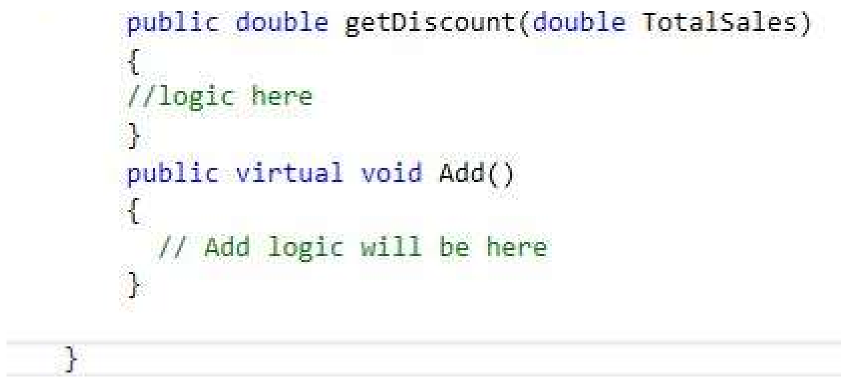
Let’s implement the Liskov principle. In this case Liskov principle says parent class object can easily replace child objects. So to implement the Liskov principle here, we will create two different interfaces :

1. IDatabase – Content Add method.
2. IDiscount – Content getDiscount method.

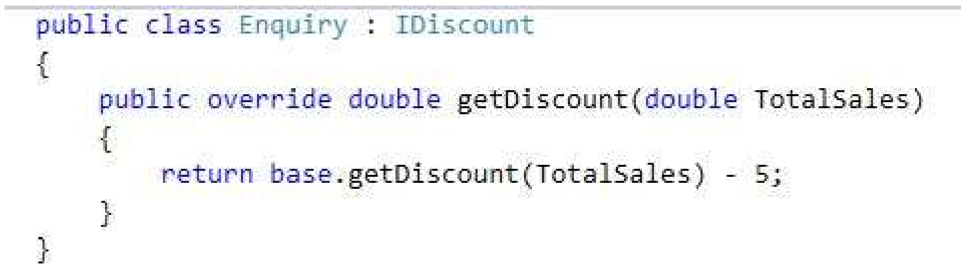


```
interface IDiscount
{
    double getDiscount(double TotalSales);
}

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

IDatabase & IDiscount interface implemented to Members, IMPMember, VIMPmember classes. As the inquiry class doesn’t have an Add method only IDiscount interface implemented.



```
        public double getDiscount(double TotalSales)
        {
        //logic here
        }
        public virtual void Add()
        {
          // Add logic will be here
        }

    }
```



```
public class Enquiry : IDiscount
{
    public override double getDiscount(double TotalSales)
    {
        return base.getDiscount(TotalSales) - 5;
    }
}
```

That’s all for LSK, now we can create a list of IDatabase and add relevant classes to it to get a list of all members.

Interface Segregation Principle (ISP)

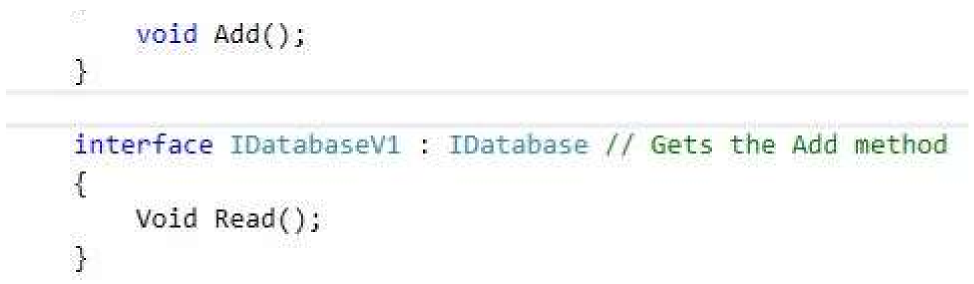
Robert C Martin's definition of ISP, *“Clients should not be forced to depend upon interfaces that they do not use.”*

Let’s try to understand with an example, suppose our member's class become so popular and thousands of companies are using it. Now a new company wants to implement members class with new functionality read members.

What will we do in this case? If we will change IDatabase interface and add a new function called ReadMemeber then thousands of existing companies will be impacted which will not use read functionality.

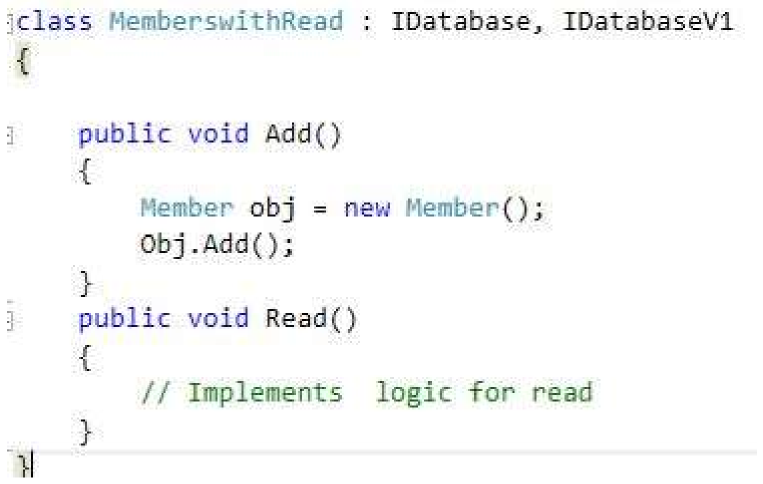
ISP principle helps us to solve this problem,

1. Instead of adding a new method in existing members-interfaces, will create a new interface.
2. IDatabase 1 – Inherit IDatabase and add a new method ReadMember() method in it.



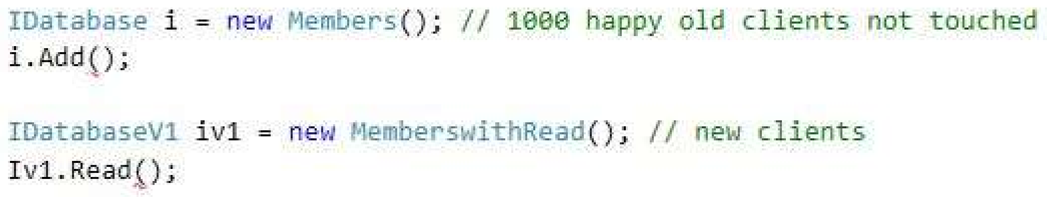
```
    void Add();
}

interface IDatabaseV1 : IDatabase // Gets the Add method
{
    Void Read();
}
```

Now new member would be as per below,



```
class MemberswithRead : IDatabase, IDatabaseV1
{

    public void Add()
    {
        Member obj = new Member();
        Obj.Add();
    }
    public void Read()
    {
        // Implements  logic for read
    }
}
```

So old company members continue using “IDatabase” interface and new company members can start using the new interface “IDatabase1”.



```
IDatabase i = new Members(); // 1000 happy old clients not touched
i.Add();

IDatabaseV1 iv1 = new MemberswithRead(); // new clients
Iv1.Read();
```

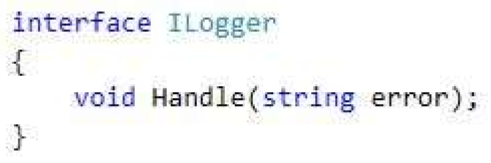
Dependency Inversion principle (DIP)

Robert C Martin has defined DIP as, *“High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions”.*

Let’s try to understand this principle with an example,

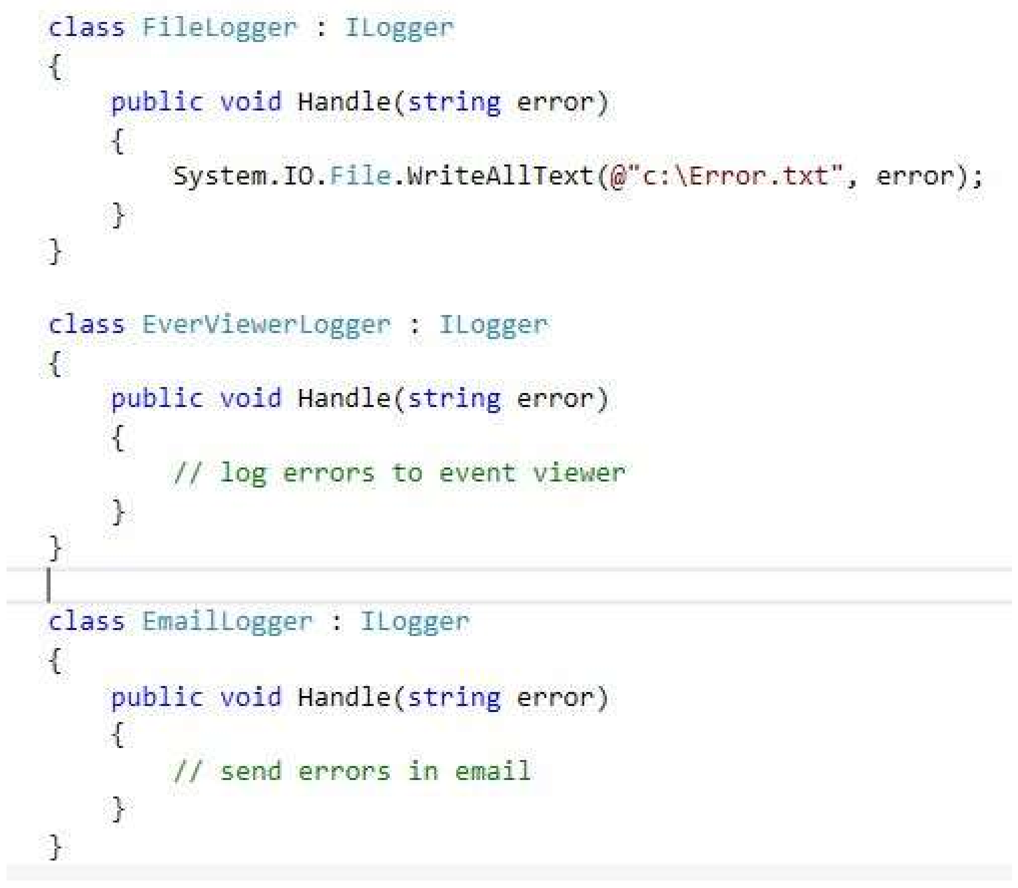
In the first principle, we have created a logger class that is used in members class. Let’s imagine, now we have a requirement to log details in EvernViwer, Database or in the file. T

1. Interface called 'ILogger'. This interface has a method called Handle.



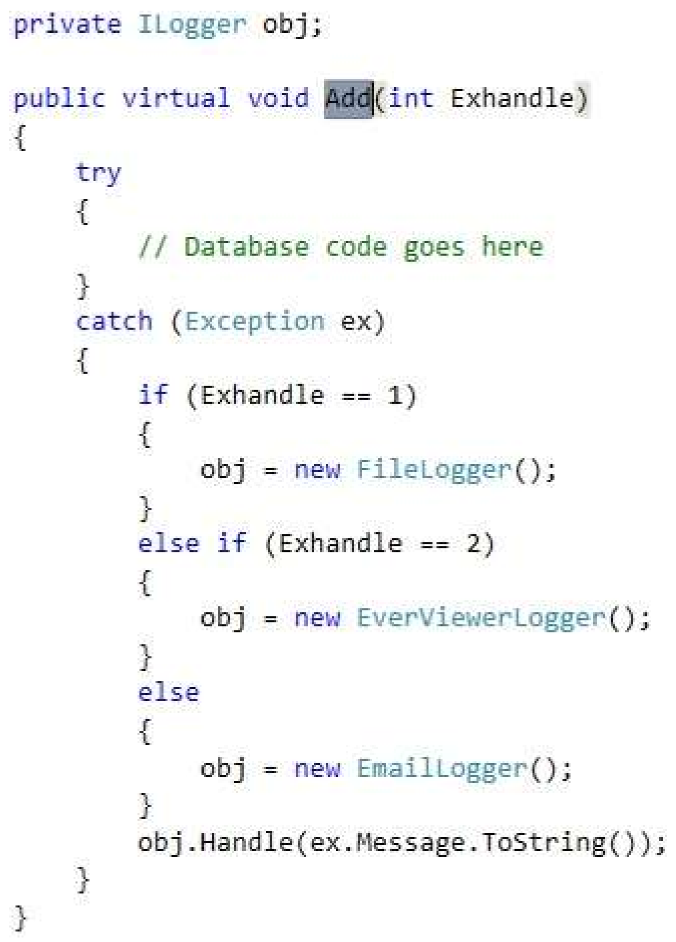
```
interface ILogger
{
    void Handle(string error);
}
```

2. Create three classes FileLogger, EverViewerLogger, EmailLogger implementing ILogger interface.



```
class FileLogger : ILogger
{
    public void Handle(string error)
    {
        System.IO.File.WriteAllText(@"c:\Error.txt", error);
    }
}

class EverViewerLogger : ILogger
{
    public void Handle(string error)
    {
        // log errors to event viewer
    }
}

class EmailLogger : ILogger
{
    public void Handle(string error)
    {
        // send errors in email
    }
}
```

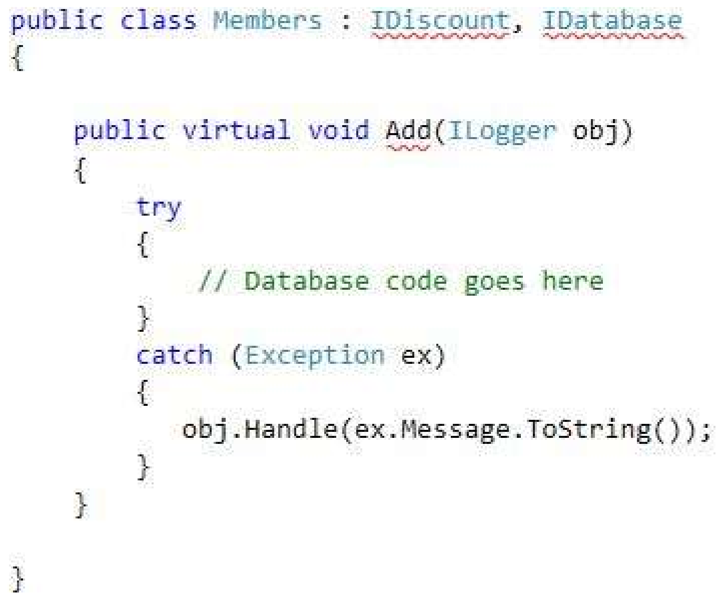
3. Create Members base class as per below



```
private ILogger obj;

public virtual void Add(int Exhandle)
{
    try
    {
        // Database code goes here
    }
    catch (Exception ex)
    {
        if (Exhandle == 1)
        {
            obj = new FileLogger();
        }
        else if (Exhandle == 2)
        {
            obj = new EverViewerLogger();
        }
        else
        {
            obj = new EmailLogger();
        }
        obj.Handle(ex.Message.ToString());
    }
}
```

In the first view, all looks good. Right? But if you see closely, you will notice that we violate single responsibility principle. We are creating logger objects in Members class.

To solve this we will apply DIP here,



```
public class Members : IDiscount, IDatabase
{

    public virtual void Add(ILogger obj)
    {
        try
        {
            // Database code goes here
        }
        catch (Exception ex)
        {
           obj.Handle(ex.Message.ToString());
        }
    }

}
```

conditions. We have removed logger dependency and that's the last principle. We can use different methodologies to remove the dependency, and we will discuss that incoming article.

I hope you enjoy this article and that it helped you to understand SOLID principles.

design Principles | Object Oriented design principles | SOLID Principles

Next Recommended Reading
SOLID Principles In C#

Kirtesh Shah TOP 500

C#.NET | .Net Core | MVC | ASP.NET | WCF | WEB API | MicroServices | React Js | Angular Js | SQL | WIN FORMS |Scrum Master | Mendix | Agile | Azure

172 769.8k 2

12 0

Type your comment here and press Enter Key (Minimum 10 characters)

FEATURED ARTICLES

OAuth2.0 Authorization With The Azure AD Client Credentials FLow To Secure APIs Of Azure API Management

What's New In C# 10?

AJAX In .NET Core

Challenges Software Developers Face Today

Intent And Its Types In Android

TRENDING UP

01 Azure Duration Functions - How To Use And Implement It

02 Creating Search Feature In Blazor Server Grid

03 Log Correlation In Microservices

04 Create A PowerApps Component Framework (PCFx) Using Custom Code In PowerAPPs

05 Import PowerApps Component Framework (PCFx) Into Model Driven PowerApps

06 Fundamentals Of Software Architecture

07 Safest Way To Convert String To Int In C#

08 How To Migrate From .NET Core 3.1 To .NET 6.0

09 Top Three VS Code Extensions Worth The Money

10 Web3 Developer Salary Soars In 2022

Learn C# 8.0

CHALLENGE YOURSELF

GET CERTIFIED

Microsoft Azure