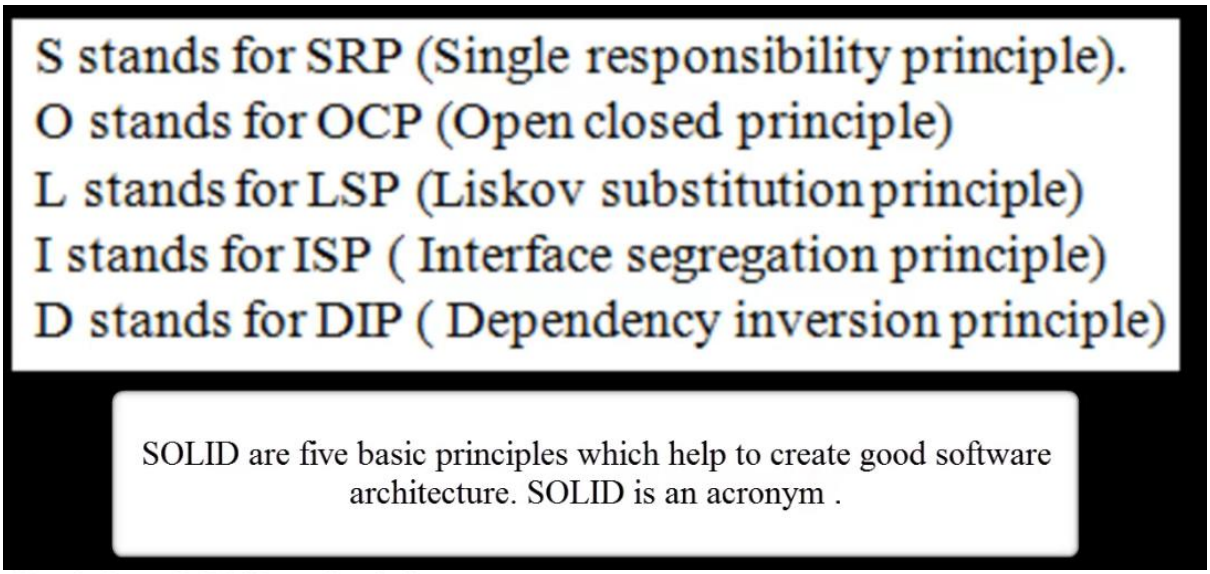

Revising SOLID principles

S stands for SRP (Single responsibility principle):- A class should take care of only one responsibility.

O stands for OCP (Open closed principle):- Extension should be preferred over modification.

L stands for LSP (Liskov substitution principle):- A parent class object should be able to refer child objects seamlessly during runtime polymorphism.

I stands for ISP (Interface segregation principle):- Client should not be forced to use a interface if it does not need it.

D stands for DIP (Dependency inversion principle) :- High level modules should not depend on low level modules but should depend on abstraction.

S — Single responsibility principle

In programming, the Single Responsibility Principle states that every module or class should have responsibility over a single part of the functionality provided by the software.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace SOLID_PRINCIPLES_DEMO
{

S STANDS FOR SRP  (Single responsibility principle)
   public class Buyer
    {
        public void Add()
        {

            try
            {
                //Adds the buyer to the database
            }
            catch (Exception ex)
            {
                ErrorHandler obj = new ErrorHandler();
                obj.HandleError(ex.ToString());
                //As per the SRP says that a class should have only one responsiblity
                //and not multiple
                //System.IO.File.WriteAllText(@"d:\Error.txt", ex.ToString());
            }

        }
    }
    public class ErrorHandler
    {
        public void HandleError(string err)
        {
            System.IO.File.WriteAllText(@"d:\Error.txt", err.ToString());

        }

    }

    class Program
    {
        static void Main(string[] args)
        {

            Buyer objBuyer = new Buyer();
            objBuyer.Add();


        }
    }
}
```

O — Open/closed principle

In programming, the open/closed principle states that software entities (classes, modules, functions, etc.) should be open for extensions, but closed for modification.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SOLID_PRINCIPLES_DEMO
{


    //the open/closed principle states
    //"software entities (classes, modules, functions, etc.)
    //should be open for extension, but closed for modification”; [1] that is, such an
entity can allow its behaviour to be extended without modifying its source code.
    public class Buyer
    {

        private int _BuyerType;

        public int BuyerType
        {
            get { return _BuyerType; }
            set { _BuyerType = value; }
        }

        public double CalculateDiscount()
        {
            if (_BuyerType ==1)
            {
                return 10;
            }
            else
            {
                return 5;
            }

        }


        public void Add()
        {
            try
            {
                //Adds the buyer to the database
            }
            catch (Exception ex)
            {
                ErrorHandler obj = new ErrorHandler();
                obj.HandleError(ex.ToString());
                //As per the SRP says that a class should have only one responsiblity
                //and not multiple
```

```
                //System.IO.File.WriteAllText(@"d:\Error.txt", ex.ToString());
            }

        }
    }
    public class ErrorHandler
    {
        public void HandleError(string err)
        {
            System.IO.File.WriteAllText(@"d:\Error.txt", err.ToString());

        }

    }

    class Program
    {
        static void Main(string[] args)
        {

            Buyer objBuyer = new Buyer();
            objBuyer.Add();


        }
    }
}
```

SOLUTION:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SOLID_PRINCIPLES_DEMO
{

    //SOLID : Are five basic principles which helps to create good software architecture
    //SOLLID is an acronym

    //S STANDS FOR SRP  (Single responsibility principle)

    //the open/closed principle states
    //"software entities (classes, modules, functions, etc.)
    //should be open for extension, but closed for modification";[1] that is, such an
entity can allow its behaviour to be extended without modifying its source code.
    public class Buyer
    {

        private int _BuyerType;

        public int BuyerType
        {
            get { return _BuyerType; }
```

```
            set { _BuyerType = value; }
        }

        public virtual double CalculateDiscount()
        {
            return 0;

        }


        public virtual void Add()
        {
            try
            {
                //Adds the buyer to the database
            }
            catch (Exception ex)
            {
                ErrorHandler obj = new ErrorHandler();
                obj.HandleError(ex.ToString());
                //As per the SRP says that a class should have only one responsiblity
                //and not multiple
                //System.IO.File.WriteAllText(@"d:\Error.txt", ex.ToString());
            }

        }
    }
    public class GoldBuyer : Buyer
    {
        public override double CalculateDiscount()
        {
            return base.CalculateDiscount() + 10;

        }

    }
    public class SilverBuyer : Buyer
    {
        public override double CalculateDiscount()
        {
            return base.CalculateDiscount() + 5;

        }

    }
    public class EquiryBuyer : Buyer
    {
        public override double CalculateDiscount()
        {
            return 2;

        }
        public override void Add()
        {
            throw new NotImplementedException("For Enquiry type of buyer , not add into
database");
        }
    }
```

```
    public class ErrorHandler
    {
        public void HandleError(string err)
        {
            System.IO.File.WriteAllText(@"d:\Error.txt", err.ToString());

        }

    }

    class Program
    {
        static void Main(string[] args)
        {

            Buyer objBuyer = new Buyer();
            objBuyer.Add();


        }
    }
}
```

L — Liskov substitution principle

This one is probably the hardest one to wrap your head around when being introduced for the first time.

In programming, the Liskov substitution principle states that if ***S*** is a subtype of ***T***, then objects of type ***T*** may be replaced (or substituted) with objects of type ***S***.

This can be formulated mathematically as

Let ***ϕ(x)*** *be a property provable about objects* ***x*** *of type* ***T****. Then* ***ϕ(y)*** *should be true for objects* ***y*** *of type* ***S****, where* ***S*** *is a subtype of* ***T****.*

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;

namespace SOLID_PRINCIPLES_DEMO
{

    //SOLID : Are five basic principles which helps to create good software architecture
    //SOLLID is an acronym

    //S STANDS FOR SRP  (Single responsibility principle)

    //the open/closed principle states
    //"software entities (classes, modules, functions, etc.)
    //should be open for extension, but closed for modification";[1] that is, such an
entity can allow its behaviour to be extended without modifying its source code.
    public class Buyer
    {

        private int _BuyerType;

        public int BuyerType
        {
            get { return _BuyerType; }
            set { _BuyerType = value; }
        }

        public virtual double CalculateDiscount()
        {
            return 0;

        }


        public virtual void Add()
        {
            try
            {
                //Adds the buyer to the database
            }
            catch (Exception ex)
            {
                ErrorHandler obj = new ErrorHandler();
                obj.HandleError(ex.ToString());
                //As per the SRP says that a class should have only one responsiblity
                //and not multiple
                //System.IO.File.WriteAllText(@"d:\Error.txt", ex.ToString());
            }

        }
    }
    public class GoldBuyer : Buyer
    {
        public override double CalculateDiscount()
        {
            return base.CalculateDiscount() + 10;

        }
```

```
    }
    public class SilverBuyer : Buyer
    {
        public override double CalculateDiscount()
        {
            return base.CalculateDiscount() + 5;

        }

    }
    public class EquiryBuyer : Buyer
    {
        public override double CalculateDiscount()
        {
            return 2;

        }
        public override void Add()
        {
            throw new NotImplementedException("For Enquiry type of buyer , not add into
database");
        }
    }
    public class ErrorHandler
    {
        public void HandleError(string err)
        {
            System.IO.File.WriteAllText(@"d:\Error.txt", err.ToString());

        }

    }

    class Program
    {
        static void Main(string[] args)
        {

            List<Buyer> ListOfBuyers = new List<Buyer>();
            ListOfBuyers.Add(new GoldBuyer());
            ListOfBuyers.Add(new SilverBuyer());
            ListOfBuyers.Add(new EquiryBuyer());

            foreach (Buyer b in ListOfBuyers)
            {
                b.Add();

            }


        }
    }
}
```

SOLUTION:

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SOLID_PRINCIPLES_DEMO
{

    //SOLID : Are five basic principles which helps to create good software architecture
    //SOLLID is an acronym

    //S STANDS FOR SRP  (Single responsibility principle)

    //the open/closed principle states
    //"software entities (classes, modules, functions, etc.)
    //should be open for extension, but closed for modification";[1] that is, such an
entity can allow its behaviour to be extended without modifying its source code.

    public interface IEnquiry
    {
        double CalculateDiscount();

    }

    public interface IBuyer
    {
        void Add();

    }

    public class Buyer : IBuyer
    {

        private int _BuyerType;

        public int BuyerType
        {
            get { return _BuyerType; }
            set { _BuyerType = value; }
        }

        public virtual double CalculateDiscount()
        {
            return 0;

        }


        public virtual void Add()
        {
            try
            {
                //Adds the buyer to the database
            }
            catch (Exception ex)
            {
                ErrorHandler obj = new ErrorHandler();
                obj.HandleError(ex.ToString());
```

```
                //As per the SRP says that a class should have only one responsiblity
                //and not multiple
                //System.IO.File.WriteAllText(@"d:\Error.txt", ex.ToString());
            }

        }
    }
    public class GoldBuyer : Buyer
    {
        public override double CalculateDiscount()
        {
            return base.CalculateDiscount() + 10;

        }

    }
    public class SilverBuyer : Buyer
    {
        public override double CalculateDiscount()
        {
            return base.CalculateDiscount() + 5;

        }

    }
    public class EquiryBuyer : IEnquiry
    {
        public  double CalculateDiscount()
        {
            return 2;

        }

    }
    public class ErrorHandler
    {
        public void HandleError(string err)
        {
            System.IO.File.WriteAllText(@"d:\Error.txt", err.ToString());

        }

    }

    class Program
    {
        static void Main(string[] args)
        {

            List<Buyer> ListOfBuyers = new List<Buyer>();
            ListOfBuyers.Add(new GoldBuyer());
            ListOfBuyers.Add(new SilverBuyer());
            //ListOfBuyers.Add(new EquiryBuyer());
            //So LISKOV Principle says that the parent should easily replace the child
object

            foreach (Buyer b in ListOfBuyers)
            {
```

```
                b.Add();

            }


        }
    }
}
```

I — Interface segregation principle

This principle is fairly easy to comprehend. In fact, if you're used to using interfaces, chances are that you're already applying this principle.

If not, it's time to start doing it!

In programming, the interface segregation principle states that no client should be forced to depend on methods it does not use. Put more simply: Do not add additional functionality to an existing interface by adding new methods.
Instead, create a new interface and let your class implement multiple interfaces if needed.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SOLID_PRINCIPLES_DEMO
{

    //SOLID : Are five basic principles which helps to create good software architecture
    //SOLLID is an acronym

    //S STANDS FOR SRP  (Single responsibility principle)

    //the open/closed principle states
    //"software entities (classes, modules, functions, etc.)
    //should be open for extension, but closed for modification";[1] that is, such an
entity can allow its behaviour to be extended without modifying its source code.

    public interface IEnquiry
    {
        double CalculateDiscount();
```

```
}


public interface IBuyer
{
    void Add();
    //void Read(); // That is not feasible way  to add new method


}
public interface IRead : IBuyer
{

    void Read();
}

public class Buyer : IBuyer,IRead
{

    private int _BuyerType;

    public int BuyerType
    {
        get { return _BuyerType; }
        set { _BuyerType = value; }
    }

    public virtual double CalculateDiscount()
    {
        return 0;

    }


    public virtual void Add()
    {
        try
        {
            //Adds the buyer to the database
        }
        catch (Exception ex)
        {
            ErrorHandler obj = new ErrorHandler();
            obj.HandleError(ex.ToString());
            //As per the SRP says that a class should have only one responsiblity
            //and not multiple
            //System.IO.File.WriteAllText(@"d:\Error.txt", ex.ToString());
        }

    }

    public void Read()
    {
        throw new NotImplementedException();
    }
}
public class GoldBuyer : Buyer
{
```

```
        public override double CalculateDiscount()
        {
            return base.CalculateDiscount() + 10;

        }

    }
    public class SilverBuyer : Buyer
    {
        public override double CalculateDiscount()
        {
            return base.CalculateDiscount() + 5;

        }

    }
    public class EquiryBuyer : IEnquiry
    {
        public  double CalculateDiscount()
        {
            return 2;

        }

    }
    public class ErrorHandler
    {
        public void HandleError(string err)
        {
            System.IO.File.WriteAllText(@"d:\Error.txt", err.ToString());

        }

    }

    class Program
    {
        static void Main(string[] args)
        {
            // 1000 of clients were using old buyer class
            IBuyer OldClients = new Buyer();
            OldClients.Add();
            // New clients

            IRead INewClients = new Buyer();
            INewClients.Add();
            INewClients.Read();


        }
    }
}
```

D – Dependency inversion principle

Finally, we got to D, the last of the 5 principles.

In programming, the dependency inversion principle is a way to decouple software modules.
This principle states that

- 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.

To comply with this principle, we need to use a design pattern known as a *dependency inversion pattern*, most often solved by using *dependency injection*.
Dependency injection is a huge topic and can be as complicated or simple as one might see the need for.

Typically, dependency injection is used simply by 'injecting' any dependencies of a class through the class' constructor as an input parameter.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SOLID_PRINCIPLES_DEMO
{

    //SOLID : Are five basic principles which helps to create good software architecture
    //SOLLID is an acronym

    //S STANDS FOR SRP  (Single responsibility principle)

    //the open/closed principle states
    //"software entities (classes, modules, functions, etc.)
    //should be open for extension, but closed for modification";[1] that is, such an
entity can allow its behaviour to be extended without modifying its source code.

    public interface IEnquiry
```

```
{
    double CalculateDiscount();

}


public interface IBuyer
{
    void Add();
    //void Read(); // That is not feasible way  to add new method


}
public interface IRead : IBuyer
{

    void Read();
}

public class Buyer : IBuyer,IRead
{

    private int _BuyerType;

    public int BuyerType
    {
        get { return _BuyerType; }
        set { _BuyerType = value; }
    }

    public virtual double CalculateDiscount()
    {
        return 0;

    }

    private IErrorHandler IErr = new EventErrorHandler();
    public virtual void Add()
    {
        try
        {
            //Adds the buyer to the database
        }
        catch (Exception ex)
        {

            IErr.HandleError(ex.ToString());
            //As per the SRP says that a class should have only one responsiblity
            //and not multiple
            //System.IO.File.WriteAllText(@"d:\Error.txt", ex.ToString());
        }

    }

    public void Read()
    {
        throw new NotImplementedException();
    }
```

```
}
public class GoldBuyer : Buyer
{
    public override double CalculateDiscount()
    {
        return base.CalculateDiscount() + 10;

    }

}
public class SilverBuyer : Buyer
{
    public override double CalculateDiscount()
    {
        return base.CalculateDiscount() + 5;

    }

}
public class EquiryBuyer : IEnquiry
{
    public  double CalculateDiscount()
    {
        return 2;

    }

}
public interface IErrorHandler
{
    void HandleError(string err);

}
public class FileErrorHandler : IErrorHandler
{
    public void HandleError(string err)
    {
        System.IO.File.WriteAllText(@"d:\Error.txt", err.ToString());

    }

}
public class EventErrorHandler : IErrorHandler
{
    public void HandleError(string err)
    {
        System.IO.File.WriteAllText(@"d:\Error.txt", err.ToString());

    }

}
class Program
{
    static void Main(string[] args)
    {
```

```
        }
    }
}
```

SOLUTION:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SOLID_PRINCIPLES_DEMO
{

    public interface IEnquiry
    {
        double CalculateDiscount();
    }
    public interface IBuyer
    {
        void Add();
        //void Read(); // That is not feasible way to add new method
    }
    public interface IRead : IBuyer
    {
        void Read();
    }
    public interface IErrorHandler
    {
        void HandleError(string err);

    }
    public class Buyer : IBuyer, IRead
    {
        private IErrorHandler IErr;

        public Buyer()
        {

        }
        public Buyer(IErrorHandler Err)
        {
            IErr = Err;
        }
        private int _BuyerType;
        public int BuyerType
        {
            get { return _BuyerType; }
            set { _BuyerType = value; }
        }
        public virtual double CalculateDiscount()
        {
            return 0;
        }
```

```
    public virtual void Add()
    {
        try
        {
            //Adds the buyer to the database
        }
        catch (Exception ex)
        {
            IErr.HandleError(ex.ToString());
            //As per the SRP says that a class should have only one responsiblity
            //and not multiple
            //System.IO.File.WriteAllText(@"d:\Error.txt", ex.ToString());
        }
    }
    public void Read()
    {
        throw new NotImplementedException();
    }
}
public class GoldBuyer : Buyer
{
    public override double CalculateDiscount()
    {
        return base.CalculateDiscount() + 10;
    }
}
public class SilverBuyer : Buyer
{
    public override double CalculateDiscount()
    {
        return base.CalculateDiscount() + 5;
    }
}
public class EquiryBuyer : IEnquiry
{
    public double CalculateDiscount()
    {
        return 2;
    }
}
public interface IErrorHandler
{
    void HandleError(string err);
}
public class FileErrorHandler : IErrorHandler
{
    public void HandleError(string err)
    {
        System.IO.File.WriteAllText(@"d:\Error.txt", err.ToString());
    }
}
public class EventErrorHandler : IErrorHandler
{
    public void HandleError(string err)
    {
        System.IO.File.WriteAllText(@"d:\Error.txt", err.ToString());
    }
```

```
    }
    public class StockErrorHandler : IErrorHandler
    {
        public void HandleError(string err)
        {
            System.IO.File.WriteAllText(@"d:\Error.txt", err.ToString());
        }
    }
    class DependencyInversionPrinciple_5
    {
        static void Main(string[] args)
        {
            // 1000 of clients were using old buyer class
            IBuyer OldClients = new Buyer(new FileErrorHandler());
            OldClients.Add();
            // New clients

            IReadV INewClients = new Buyer(new EventErrorHandler());
            INewClients.Add();
            INewClients.Read();

            IReadV INewNewClients = new Buyer(new StockErrorHandler());
            INewNewClients.Add();
            INewNewClients.Read();

        }
    }
}
```