**SOLID Design Principles Introduction**

**SOLID Introduction** 

1. SOLID principles are the design principles that enable us manage most of the software design problems
2. The term SOLID is an acronym for five design principles intended to make software designs more understandable, flexible and maintainable
3. The principles are a subset of many principles promoted by Robert C. Martin
4. The SOLID acronym was first introduced by Michael Feathers

**SOLID Acronym** 

* **S** : **S**ingle Responsibility Principle (SRP)
* **O** : **O**pen closed Principle (OSP)
* **L** :  **L**iskov substitution Principle (LSP)
* **I**  :  **I**nterface Segregation Principle (ISP)
* **D** : **D**ependency Inversion Principle (DIP)

**Single Responsibility Principle** 

* Robert C. Martin expresses the principle as, "A class should have only one reason to change”
* Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class

**Liskov Substitution Principle** 

* Introduced by Barbara Liskov state that “objects in a program should be replaceable with instances of their sub-types without altering the correctness of that program”
* If a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module
* We can also state that Derived types must be substitutable for their base types

**Open/Closed Principle** 

* “Software entities should be open for extension, but closed for modification”
* The design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code
* The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged

**Interface Segregation Principle** 

* “Many client-specific interfaces are better than one general-purpose interface”
* We should not enforce clients to implement interfaces that they don't use. Instead of creating one big interface we can break down it to smaller interfaces

**Dependency Inversion Principle** 

* One should “depend upon abstractions, [not] concretions"
* Abstractions should not depend on the details whereas the details should depend on abstractions
* High-level modules should not depend on low level modules

**If we don’t follow SOLID Principles we** 

* End up with tight or strong coupling of the code with many other modules/applications
* Tight coupling causes time to implement any new requirement, features or any bug fixes and some times it creates unknown issues
* End up with a code which is not testable
* End up with duplication of code
* End up creating new bugs by fixing another bug
* End up with many unknown issues in the application development cycle

**Following SOLID Principles helps us to** 

* Achieve reduction in complexity of code
* Increase readability, extensibility and maintenance
* Reduce error and implement Reusability
* Achieve Better testability
* Reduce tight coupling

**Solution to develop a successful application depends on** 

* Architecture : choosing an architecture is the first step in designing application based on the requirements. **Example :** MVC, WEBAPI, MVVM..etc
* Design Principles : Application development process need to follow the design principles
* Design Patterns : We need to choose correct design patterns to build the software

### Single Responsibility Principle

### As per the single responsibility principle

* A class should have only one reason to change
* Which means, every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.
* Encapsulation is one of the fundamentals of OOP. At this moment, understanding more about encapsulation is out of scope of this session.

Now you might be wondering what do we achieve with the Single Responsibility Principle or rather with the SOLID Design Principles.   
  
Let's first understand the **motivation behind the usage of SOLID Principles**  
  
In any enterprise software application development when we design and develop software systems, we need to account the below factors during the development cycle. 

* **Maintainability :**Maintainable systems are very important to the organisations.
* **Testability :** Test driven development (TDD) is required when we design and develop large scale systems
* **Flexibility and Extensibility :** Flexibility and extensibility is a very much desirable factor of enterprise applications.Hence we should design the application to make it flexible so that it can be adapt to work in different ways and extensible so that we can add new features easily.
* **Parallel Development :** It is one of the key features in the application development as it is not practical to have the entire development team working simultaneously on the same feature or component.
* **Loose Coupling :** We can address many of the requirements listed above by ensuring that our design results in an application that loosely couples many parts that makes up the application.

SOLID Principles and Design Patterns plays a key role in achieving all of the above points.  
  
**In Single Responsibility Principle** 

* Each class and module should focus on a single task at a time
* Everything in the class should be related to that single purpose
* There can be many members in the class as long as they related to the single responsibility
* With SRP, classes become smaller and cleaner
* Code is less fragile

Hence we can say that Single Responsibility Principle achieves the motivation points that we have just discussed.  
  
Below code demonstrates how we can achieve Single Responsibility Principle  
  
**Code before Single Responsibility Segregation**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

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);

    }

}

**Code after Single Responsibility Segregation**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

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);

    }

}

Now that we have segregated the single responsibility principle in these multiple interfaces the next step is to implement these interfaces with object creation mechanisms.

### Interface Segregation Principle

**we will discuss** 

* Interface Segregation Principle
* Will look at a Case Study of Interface Segregation Principle
* And will implement Interface Segregation Principle with a simple example

In the first session of SOLID Introduction we have understood that **"I"** in SOL**I**D is an acronym for **Interface Segregation Principle** 

* The interface-segregation principle (ISP) states that "no client should be forced to depend on methods it does not use".
* This means, instead of one fat interface many small interfaces are preferred based on groups of methods with each one serving one sub-module.
* The ISP was first used and formulated by Robert C. Martin while consulting for Xerox.

Let us now understand how the ISP was evolved with a case study.   
  
**Case Study**  
  
**Problem**

* As we all know Xerox Corporation manufactures printer systems. In their development process of new systems Xerox had created a new printer system that could perform a variety of tasks such as stapling and faxing along with the regular printing task.
* The software for this system was created from the ground up.
* As the software grew for Xerox, making modifications became more and more difficult so that even the smallest change would take a redeployment cycle of an hour, which made development nearly impossible.
* The design problem was that a single Job class was used by almost all of the tasks. Whenever a print job or a stapling job needed to be performed, a call was made to the Job class.
* This resulted in a 'fat' class with multitudes of methods specific to a variety of different clients.

Because of this design, a staple job would know about all the methods of the print job, even though there was no use for them.   
  
**Solution**

* To overcome this problem Robert C Martin suggested a solution which is called the Interface Segregation Principle.
* Which means, Instead of one fat interface many small interfaces are preferred based on groups of methods with each one serving one sub-module.

Below example demonstrates how we can achieve Single Responsibility Principle   
  
**Code before Interface Segregation Principle.** 

namespace ISPDemoConsole

{

    public interface IPrintTasks

    {

        bool PrintContent(string content);

        bool ScanContent(string content);

        bool FaxContent(string content);

        bool PhotoCopyContent(string content);

        bool PrintDuplexContent(string content);

    }

}

namespace ISPDemoConsole.Client

{

    class HPLaserJet : IPrintTasks

    {

        public bool FaxContent(string content)

        {

            Console.WriteLine("Fax Done"); return true;

        }

        public bool PhotoCopyContent(string content)

        {

            Console.WriteLine("PhotoCopy Done"); return true;

        }

        public bool PrintContent(string content)

        {

            Console.WriteLine("Print Done"); return true;

        }

        public bool PrintDuplexContent(string content)

        {

            Console.WriteLine("Print Duplex Done"); return true;

        }

        public bool ScanContent(string content)

        {

            Console.WriteLine("Scan Done"); return true;

        }

    }

}

namespace ISPDemoConsole.Client

{

    class CannonMG2470 : IPrintTasks

    {

        public bool PhotoCopyContent(string content)

        {

            Console.WriteLine("PhotoCopy Done"); return true;

        }

        public bool PrintContent(string content)

        {

            Console.WriteLine("Print Done"); return true;

        }

        public bool ScanContent(string content)

        {

            Console.WriteLine("Scan Done"); return true;

        }

        public bool PrintDuplexContent(string content)

        {

            return false;

        }

        public bool FaxContent(string content)

        {

            return false;

        }

     }

}

**Code after Interface Segregation Principle**

namespace ISPDemoConsole

{

    interface IPrintScanContent

    {

        bool PrintContent(string content);

        bool ScanContent(string content);

        bool PhotoCopyContent(string content);

    }

    interface IFaxContent

    {

        bool FaxContent(string content);

    }

    interface IPrintDuplex

    {

        bool PrintDuplexContent(string content);

    }

}

namespace ISPDemoConsole.Client

{

    class HPLaserJet : IPrintScanContent, IFaxContent, IPrintDuplex

    {

        public bool FaxContent(string content)

        {

            Console.WriteLine("Fax Done"); return true;

        }

        public bool PhotoCopyContent(string content)

        {

            Console.WriteLine("PhotoCopy Done"); return true;

        }

        public bool PrintContent(string content)

        {

            Console.WriteLine("Print Done"); return true;

        }

        public bool PrintDuplexContent(string content)

        {

            Console.WriteLine("Print Duplex Done"); return true;

        }

        public bool ScanContent(string content)

        {

            Console.WriteLine("Scan Done"); return true;

        }

    }

}

namespace ISPDemoConsole.Client

{

    class CannonMG2470 : IPrintScanContent

    {

        public bool PhotoCopyContent(string content)

        {

            Console.WriteLine("PhotoCopy Done");

            return true;

        }

        public bool PrintContent(string content)

        {

            Console.WriteLine("Print Done");

            return true;

        }

        public bool ScanContent(string content)

        {

            Console.WriteLine("Scan Done");

            return true;

        }

    }

}

### Open Closed Principle

we have understood that **O** in the S**O**LID is acronym for **"Open/Closed Principle"** also known as **OCP**.   
  
**Definition :** In object-oriented programming, the open/closed principle states that "software entities such as classes, modules, functions, etc. should be open for extension, but closed for modification" 

* Which means, any new functionality should be implemented by adding new classes, attributes and methods, instead of changing the current ones or existing ones.
* Bertrand Meyer is generally credited for having originated the term open/closed principle and This Principle is considered by Bob Martin as "the most important principle of object-oriented design".

**Implementation guidelines** 

* The simplest way to apply OCP is to implement the new functionality on new derived (sub) classes that inherit the original class implementation.
* Another way is to allow client to access the original class with an abstract interface.
* So, at any given point of time when there is a requirement change instead of touching the existing functionality it’s always suggested to create new classes and leave the original implementation untouched.

**Pit falls of Not following OCP :**What if I do not follow Open closed principle during a requirement enhancement in the development process. In that case, we end up with the following disadvantages

1. If a class or a function always allows the addition of new logic, as a developer we end up testing the entire functionality along with the requirement.
2. Also, as a developer we need to ensure to communicate the scope of the changes to the Quality Assurance team in advance so that they can gear up for enhanced regression testing along with the feature testing.
3. Step 2 above is a costly process to adapt for any organization
4. Not following the Open Closed Principle breaks the SRP since the class or function might end up doing multiple tasks.
5. Also, if the changes are implemented on the same class, Maintenance of the class becomes difficult since the code of the class increases by thousands of unorganised lines.

Hope the above counter facts help in understanding on why we need to follow the open closed principle.   
  
Below code demonstrates how we can achieve Open Closed Principle  
  
**Code before Open Closed Principle**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace OpenClosedDemo

{

    public class Employee

    {

        Employee() { }

        public Employee(int id, string name, string type)

        {

            this.ID = id;

            this.Name = name;

            this.EmployeeType = type;

        }

        public int ID { get; set; }

        public string EmployeeType { get; set; }

        public string Name { get; set; }

        public decimal CalculateBonus(decimal salary)

        {

            if (this.EmployeeType == "Permanent")

                return salary \* .1M;

            else

                return salary \* .05M;

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace OpenClosedDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            Employee empJohn = new Employee(1, "John", "Permanent" );

            Employee empJason = new Employee(2, "Jason", "Temp");

            Console.WriteLine(string.Format("Employee {0} Bonus: {1}",

                empJohn.ToString(),

                empJohn.CalculateBonus(100000).ToString()));

            Console.ReadLine();

        }

    }

}

**Code after Open Closed Principle Implementation**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace OpenClosedDemo

{

    public abstract class Employee

    {

        public int ID { get; set; }

        public string Name { get; set; }

        public Employee()

        {

        }

        public Employee(int id, string name )

        {

            this.ID = id; this.Name = name;

        }

        public abstract decimal CalculateBonus(decimal salary);

        public override string ToString()

        {

            return string.Format("ID : {0} Name : {1}", this.ID, this.Name);

        }

    }

    public class PermanentEmployee : Employee

    {

        public PermanentEmployee()

        { }

        public PermanentEmployee(int id, string name) : base(id, name)

        { }

        public override decimal CalculateBonus(decimal salary)

        {

            return salary \* .1M;

        }

    }

    public class TemporaryEmployee : Employee

    {

        public TemporaryEmployee()

        { }

        public TemporaryEmployee(int id, string name) : base(id, name)

        { }

        public override decimal CalculateBonus(decimal salary)

        {

            return salary \* .05M;

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace OpenClosedDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            Employee empJohn = new PermanentEmployee(1, "John"  );

            Employee empJason = new TemporaryEmployee(2, "Jason" );

            Console.WriteLine(string.Format("Employee {0} Bonus: {1}",

                empJohn.ToString(),

                empJohn.CalculateBonus(100000).ToString()));

            Console.WriteLine(string.Format("Employee {0} Bonus: {1}",

              empJason.ToString(),

              empJason.CalculateBonus(150000).ToString()));

            Console.ReadLine();

        }

    }

}

### Liskov Substitution Principle

**L** in the SO**L**ID is acronym for **Liskov Substitution Principle** which is also known as **LSP**.   
  
  
**Definition :**Substitutability is a principle in object-oriented programming and it states that, in a computer program, if S is a Subtype of T, then objects of type T may be replaced with objects of type S 

* Which means, Derived types must be completely substitutable for their base types
* More formally, the Liskov substitution principle (LSP) is a particular definition of a subtyping relation, called (strong) behavioral subtyping
* This Principle is introduced by Barbara Liskov in 1987 during her conference address on Data abstraction and hierarchy
* This principle is just an extension of the Open Close Principle

The examples used in this session are related to the open closed principle.

**Implementation guidelines :**In the process of development we should ensure that  

* No new exceptions can be thrown by the subtype unless they are part of the existing exception hierarchy.
* We should also ensure that Clients should not know which specific subtype they are calling, nor should they need to know that. The client should behave the same regardless of the subtype instance that it is given.
* And last but not the least, New derived classes just extend without replacing the functionality of old classes

In the previous session as part of the [**Open closed Principle**](https://www.youtube.com/watch?v=wo06oCBuYYI) implementation we have created different employee classes to calculate bonus of the employee. From the employee perspective we have implemented the Open closed principle.   
  
Now if you take a look at the main program, we have created Employee objects which consists of both **permanent and contract employee**.   
  
If you take a closer look at this program the **Derived types** which are **Permanent**and **TemporaryEmployee**have completely substituted the base type employee class.  
  
So, based on the Liskov substitution principle we have achieved LSP by ensuring that **Derived types are completely substitutable for their base types.**  
  
Also, notice the main program, without using the subtypes we are calculating the bonus of the employee from the base class type itself. Hence, we are satisfying the Liskov substitution principle.   
  
That means along with the Open Closed Principle we have partially implemented the LSP.   
  
Also, I can state that this implementation is not adhering to guide lines of Liskov principle  
  
To understand why it’s not adhering to the Liskov Principle, Let’s assume that we need to have a Contract Employee as one of the employee category. A point to note here is a contract employee is not eligible for any bonus calculation and post implementing the Employee class we end up throwing exception at the runtime in the caclculatebonus() method. **This violates the Liskov Substitution Principle.**  
  
Hence, Please follow the below code which addresses this issue. Also, we recommend to watch our video tutorials for complete guidance and understanding of the code.  
  
**Code before Liskov Substitution Principle.** 

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace OpenClosedDemo

{

    public abstract class Employee

    {

        public int ID { get; set; }

        public string Name { get; set; }

        public Employee()

        {

        }

        public Employee(int id, string name )

        {

            this.ID = id; this.Name = name;

        }

        public abstract decimal CalculateBonus(decimal salary);

        public override string ToString()

        {

            return string.Format("ID : {0} Name : {1}", this.ID, this.Name);

        }

    }

    public class PermanentEmployee : Employee

    {

        public PermanentEmployee()

        { }

        public PermanentEmployee(int id, string name) : base(id, name)

        { }

        public override decimal CalculateBonus(decimal salary)

        {

            return salary \* .1M;

        }

    }

    public class TemporaryEmployee : Employee

    {

        public TemporaryEmployee()

        { }

        public TemporaryEmployee(int id, string name) : base(id, name)

        { }

        public override decimal CalculateBonus(decimal salary)

        {

            return salary \* .05M;

        }

    }

    public class ContractEmployee : Employee

    {

        public ContractEmployee()

        { }

        public ContractEmployee(int id, string name) : base(id, name)

        { }

        public override decimal CalculateBonus(decimal salary)

        {

            throw new NotImplementedException();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace OpenClosedDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            Employee empJohn = new PermanentEmployee(1, "John"  );

            Employee empJason = new TemporaryEmployee(2, "Jason" );

            Employee empMike = new ContractEmployee(3, "Mike" );

            Console.WriteLine(string.Format("Employee {0} Bonus: {1}",

                empJohn.ToString(),

                empJohn.CalculateBonus(100000).ToString()));

            Console.WriteLine(string.Format("Employee {0} Bonus: {1}",

              empJason.ToString(),

              empJason.CalculateBonus(150000).ToString()));

            Console.WriteLine(string.Format("Employee {0} Bonus: {1}",

              empMike.ToString(),

              empMike.CalculateBonus(150000).ToString()));

            Console.ReadLine();

        }

    }

}

Above code throws an error at empMike, as Bonus is not applicable to ContractEmployee. In that case LSP is violated and we have redefined the code to follow LSP below.

**Code after Implementing Liskov Substitution Principle.**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace LSPDemoConsole

{

    interface IEmployee

    {

        int ID { get; set; }

        string Name { get; set; }

        decimal GetMinimumSalary();

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace LSPDemoConsole

{

    interface IEmployeeBonus

    {

        decimal CalculateBonus(decimal salary);

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace LSPDemoConsole

{

    public abstract class Employee : IEmployee, IEmployeeBonus

    {

        public int ID { get; set; }

        public string Name { get; set; }

        public Employee()

        { }

        public Employee(int id, string name)

        {

            this.ID = id;

            this.Name = name;

        }

        public abstract decimal CalculateBonus(decimal salary);

        public override string ToString()

        {

            return string.Format("ID : {0} Name : {1}", this.ID, this.Name);

        }

        public abstract decimal GetMinimumSalary();

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace LSPDemoConsole.Implementation

{

    public class PermanentEmployee : Employee

    {

        public PermanentEmployee()

        { }

        public PermanentEmployee(int id, string name) : base(id, name)

        { }

        public override decimal CalculateBonus(decimal salary)

        {

            return (salary \* .1M);

        }

        public override decimal GetMinimumSalary()

        {

            return 15000;

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace LSPDemoConsole.Implementation

{

    public class TemporaryEmployee : Employee

    {

        public TemporaryEmployee()

        { }

        public TemporaryEmployee(int id, string name) : base(id, name)

        { }

        public override decimal CalculateBonus(decimal salary)

        {

            return (salary \* .05M);

        }

        public override decimal GetMinimumSalary()

        {

            return 5000;

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace LSPDemoConsole.Implementation

{

    public class ContractEmployee : IEmployee

    {

        public int ID { get; set; }

        public string Name { get; set; }

        public ContractEmployee()

        { }

        public ContractEmployee(int id, string name)

        {

            this.ID = id;

            this.Name = name;

        }

        public decimal GetMinimumSalary()

        {

            return 5000;

        }

        public override string ToString()

        {

            return string.Format("ID : {0} Name : {1}", this.ID, this.Name);

        }

    }

}

using LSPDemoConsole.Implementation;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace LSPDemoConsole

{

    class Program

    {

        static void Main(string[] args)

        {

            List<Employee> employees = new List<Employee>();

            employees.Add(new PermanentEmployee(1, "John"));

            employees.Add(new TemporaryEmployee(2, "Jason"));

            //Un Comment to see the error

            //employees.Add(new ContractEmployee(3, "Mike"));

            foreach (var employee in employees)

            {

                Console.WriteLine(string.Format("Employee {0} Bonus: {1} MinSalary: {2}",

                employee.ToString(),

                employee.CalculateBonus(100000).ToString(),

                employee.GetMinimumSalary().ToString()));

            }

            Console.WriteLine();

            List <IEmployee> employeesOnly = new List<IEmployee>();

            employeesOnly.Add(new PermanentEmployee(1, "John"));

            employeesOnly.Add(new TemporaryEmployee(2, "Jason"));

            employeesOnly.Add(new ContractEmployee(3, "Mike"));

            foreach (var employee in employeesOnly)

            {

                Console.WriteLine(string.Format("Employee {0}  MinSalary: {1}",

                employee.ToString(),

                employee.GetMinimumSalary().ToString()));

            }

            Console.ReadLine();

        }

    }

}

### Dependency Inversion Principle Introduction

D in SOLID stands for Dependency Inversion Principle which is also known as DIP  
  
**The Dependency Inversion Principle introduced by Robert C Martin states that** 

* High-level modules should not depend on low-level modules. Both should depend on abstractions.  
    
  AND
* Abstractions should not depend on details. Details should depend on abstractions.

To simplify this we can state that while designing the interaction between a high-level module and a low-level one, the interaction should be thought of as an abstract interaction between them.    
  
**Usage Intention :**Before understanding the intention of usage, let’s try to understand a traditional application architecture implementation.   
  
During the process of the application design, lower-level components are designed to be consumed by higher-level components which enable increasingly complex systems to be built. In this Process of Composition, higher-level components depend directly upon lower-level components to achieve some task.    
  
This dependency upon lower-level components limits the reuse opportunities of the higher-level components and ends up in a bad design.   
  
   
  
From the illustrated diagram, **High-level Modules depends directly on Low-level Modules** and this does not follow the first point of DIP.   
  
Allowing the dependency causes many issues and we have discussed many of them in our first session of this tutorial. For now, we are not going to get into details of those as it’s out of scope of this session. However, we strongly recommend you to refer to the first part of this tutorial before proceeding.   
  
   
  
Now let’s take a look at **button click event example**from the Presentation layer that calls directly the **Employee Save Method**of **Business logic**which does some validation checks before saving the employee details to DB.   
  
This flow looks absolutely fine however we are coupling different layers and as said earlier, any further changes are complicated and cumbersome.  
  
Now, Let’s see this problem with a sample code. 

public class BusinessLogicLayer

{

    private readonly DataAccessLayer DAL;

    public BusinessLogicLayer()

    {

        DAL = new DataAccessLayer();

    }

    public void Save(Object details)

    {

        DAL.Save(details);

    }

}

public class DataAccessLayer

{

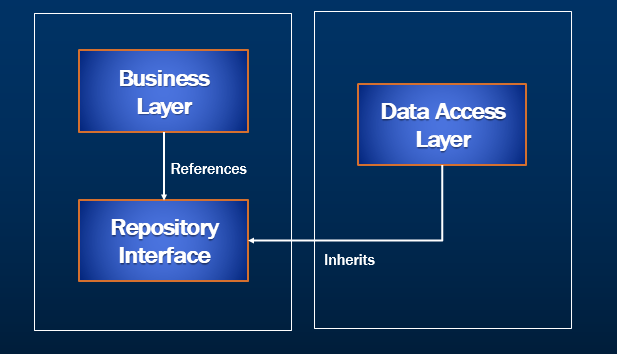
    public void Save(Object details)

    {

        //perform save

    }

}

Notice from the above code the BLL is directly dependent on the low level Data Access Layer and it’s hard to perform any unit tests on this code as both are coupled. Of course, we can do some amount of testing on the DAL but imagine if the DAL needs to be further extended to SQL and XML layers. If we need to that, implementation to extend it, becomes tedious and much more complicated.   
  
Hence based on the DIP we apply an Abstraction to decouple these layers.  
  
In order to achieve that we introduce interface that acts as abstraction so that both are decoupled.  
  
This decoupling is demonstrated in the representation diagram and further we can re-write our code as shown on the screen.   
  
  
  
If you take a look at this code based on the pictorial representation diagram we have introduced the Interface which is extended by the Data Access Layer and referred in the BLL. Hence we can say that Irepository layer acts as abstraction between these modules. 

public class BusinessLogicLayer

{

    private readonly IRepositoryLayer DAL;

    public BusinessLogicLayer(IRepositoryLayer repositoryLayer)

    {

        DAL = repositoryLayer;

    }

    public void Save(Object details)

    {

        DAL.Save(details);

    }

}

public interface IRepositoryLayer

{

    void Save(Object details);

}

public class DataAccessLayer : IRepositoryLayer

{

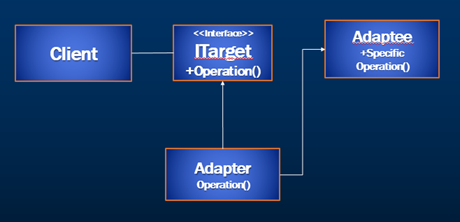
    public void Save(Object details)

    {

        //perform save

    }

}

Adapter Design pattern can be seen as an example which is applying the dependency inversion principle.   
  
   
  
The high-level class defines its **own adapter interface**which is the abstraction that the other high-level classes depend on. The Adaptee implementation also depends on the adapter interface abstraction.