**Software Design principle:**

1. it represents set of guidelines that help us to avoid bad design
2. it is less about specific language
3. its focus on DRY (Don’t REPEAT YOUSELF);

**SOLID:**

S: Single responsibility Principle

O: Open closed Principle

L: Liskov Substitution Principle

I: Interface Segregation Principle

D: Dependency Inversion principle

**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 organizations.
* **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:

**1 SRP: 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.

* **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.

In this example we have created multiple interface and for implement SRP. We can take same example with Class.

**2 .Open closed Principle:**

**"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?

**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 :**

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.

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.

Step 2 above is a costly process to adapt for any organization

Not following the Open Closed Principle breaks the SRP since the class or function might end up doing multiple tasks.

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 unorganized lines.

**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 Permanent Employee : Employee

    {

        public Permanent Employee()

        { }

        public Permanent Employee(int id, string name) : base(id, name)

        { }

        public override decimal Calculate Bonus(decimal salary)

        {

            return salary \* .1M;

        }

    }

    public class Temporary Employee : Employee

    {

        public Temporary Employee()

        { }

        public Temporary Employee(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();

        }

    }

}

**Interface Segregation Principle:**

* “Clients should not be forced to depend on methods that they do not use.”
* 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.

**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?

**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;

        }

    }

}

1. **Liskov Substitution Principle**

 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

**“We should be able to use any derived class instead of a parent class and it should behave in the same manner without modification”.**

“**Objects in an application should be replaceable with the instances of their subtypes without modifying the correctness of that application**”

* This principle is just an extension of the Open Close Principle

The examples used in this session are related to the open closed principle. Hence we request you to watch the [Open Closed Principle](https://www.youtube.com/watch?v=CWrRwC8iB30) tutorial before proceeding.   
  
**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:

**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.
* Abstractions should not depend on details. Details should depend on abstractions.
* <https://www.codeproject.com/Articles/1135745/Dependency-Inversion-Principle-DIP-2>
* https://www.journaldev.com/1392/factory-design-pattern-in-java

In short the higher-level module defines an interface and lower-level module implements that interface:

**Code Before DIP**

public class HighLevelModule

{

private readonly LowLevelModule \_lowLowelModule;

public HighLevelModule()

{

\_lowLevelModule = new LowLevelModule();

}

public void Call()

{

\_lowLevelModule.Initiate();

\_lowLevelModule.Send();

}

}

public class LowLevelModule

{

public void Initiate()

{

*//do initiation before sending*

}

public void Send()

{

*//perform sending operation*

}

}

In the above codes, *HighLevelModule* depends directly on *LowLevelModule* and this does not follow the first point of DIP. Why does this matters? The direct and tightly coupled relationship between the two makes it harder to create unit tests on *HighLevelModule* in isolation from *LowLevelModule*. You are forced to test *HighLevelModule*and *LowLevelModule* at the same time because they are tightly coupled.

#### How do we apply DIP?

The first point of DIP suggest us to apply two things at the same time to the codes:

* Abstraction,
* Dependency Inversion or Inversion of Control
* public interface IOperation
* {
* void Send();
* }
* public class LowLevelModule: IOperation
* {
* public LowLevelModule()
* {
* Initiate();
* }
* private void Initiate()
* {
* *//do initiation before sending*
* }
* public void Send()
* {
* *//perform sending operation*
* }
* }

public class HighLevelModule

{

private readonly IOperation \_operation;

public HighLevelModule(IOperation operation)

{

\_operation = operation;

}

public void Call()

{

\_operation.Send();

}

}

**The object is creation of LowLevelModule is passed to someone else call**

**Inversion of control and it is implemented by one of the way called depedency injector using**

**Construcor injectiona nd using unity container.**

var container = new UnityContainer();

container.RegisterType<IOperation, LowLevelModule>

();

HighLevelModule p = container.Resolve<HighLevelModule>();

p.Call();

1. **IOC is the design pattern by which we can practically implement DIP in software development.**
2. The IoC is inverting the control of something switching control
3. an outside module or class is responsible for creating an object of the class instead of directly creating the object of the low-level module class in the high-level module class so we can say that an IoC is an abstraction on which both high-level and low-level modules depend and it inverts the control flow.
4. IoC is used to invert the control flow of the application and an application module interacts with another module via interface and application classes object are created from one class.
5. Dependency Injection is a pattern that makes objects loosely coupled instead of tightly coupled. Generally we create a concrete class object in the class we require the object and bind it in the dependent class but DI is a pattern where we create a concrete class object outside this high-level module or dependent class.
6. The Dependency Injection Container is a framework to create dependencies and inject them automatically when required. It automatically creates objects based on requests and injects them when required.

Interface Injection

Setter Injection

Constructor Injection

Dependecy Injector(DI)

Service Locator

Inversion of control

Dependency Inversion Principle