# **Single Responsibility Principle**

**In this video we will discuss**  
1. What is Single Responsibility  
2. Single Responsibility Example   
  
  
In our previous video we discussed S in the SOLID is acronym for **Single Responsibility Principle** (SRP)   
  
  
**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.
* However, We strongly recommend you to refer to the C# tutorial playlist for more details on Object oriented principles.

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.  GOF has defined many design patterns on object creations based on the requirements.  
  
Hence we strongly recommend you to refer to our design pattern tutorial for more details on creational design patterns.  
  
I believe this session has given you a good idea on how we can implement Single responsibility principle.  
  
In the next video we will discuss **Interface Segregation Principle.**

# Interface Segregation Principle

**Suggested Videos**  
[SOLID Design Principles Introduction](https://www.youtube.com/watch?v=HLFbeC78YlU) | [Text](http://csharp-video-tutorials.blogspot.com/2017/11/solid-design-principles-introduction.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2017/11/solid-design-principles-introduction_20.html)   
[Single Responsibility Principle](https://www.youtube.com/watch?v=hGf2upfDpdo) | [Text](http://csharp-video-tutorials.blogspot.com/2017/11/single-responsibility-principle.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2017/11/single-responsibility-principle-slides.html)  
  
**In this session 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;

        }

    }

}

Also, we strongly recommend you to refer to our [design pattern tutorial](https://www.youtube.com/watch?v=rI4kdGLaUiQ&list=PL6n9fhu94yhUbctIoxoVTrklN3LMwTCmd) for more details on creational design patterns   
  
I believe this session has given you a good idea on how we can implement Interface segregation principle.   
  
In our next video we will focus on Open closed principle.