**Single Responsibility Principle in C#**

**Single Responsibility Principle in C# with Examples**

In this article, I am going to discuss the **Single Responsibility Principle in C#** with Examples. Please read our previous article before proceeding to this article where we discussed the basics of the [**SOLID Design Principle in C#**](https://dotnettutorials.net/lesson/solid-design-principles/)**.**The letter **S in S**OLID stands for the **Single Responsibility Principle**which is also known as SRP**.** As part of this article, we are going to discuss the following pointers in detail.

1. **What is the Single Responsibility Principle in C#?**
2. **How can we achieve the Single Responsibility Principle in C#?**
3. **Example without using the Single Responsibility Principle.**
4. **Problems of not following the Single Responsibility Principle.**
5. **Example using the SRP in C#.**

**What is the Single Responsibility Principle in C#?**

The Single Responsibility Principle in C# states that “**Each software module or class should have only one reason to change**“. In other words, we can say that each module or class should have only one responsibility to do.

So we need to design the software in such a way that everything in a class or module should be related to a single responsibility. That does not mean your class should contain only one method or property, you can have multiple members (methods or properties) as long as they are related to a single responsibility or functionality. So, with the help of SRP in C#, the classes become smaller and cleaner and thus easier to maintain.

**How can we achieve the Single Responsibility Principle in C#?**

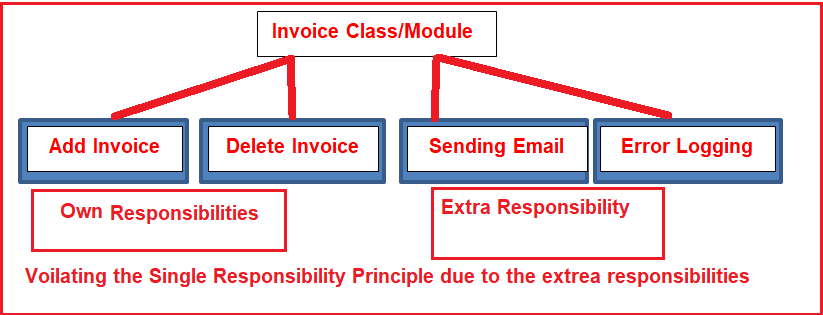
Let us understand the need for the Single Responsibility Principle in C# with an example. Suppose we need to design an Invoice class. As we know an Invoice class basically used to calculate various amounts based on its data. The Invoice class does not know how to retrieve the data, or how to format the data for display, print, logging, or sending an email, etc.

If we write the database logic, business logic as well as display logic in a single class, then our class performing multiple responsibilities. Then it becomes very difficult to change one responsibility without breaking the other responsibilities. So, by mixing multiple responsibilities into a single class, we are getting the following disadvantage,

1. **Difficult to understand**
2. **Difficult to test**
3. **Chance of duplicating the logic of other parts of the application**

**Example: Without using Single Responsibility Principle in C#:**

Please have a look at the following diagram that we want to implement in the following example.



As you can see in the above image, we are going to create an Invoice class with four functionalities as Adding and Deleting Invoices, Error Logging as well as Sending Emails. As we are putting all the above four functionalities into a single class or module, we are violating the Single Responsibility Principle in C#. This is because Sending Emails and Error Logging are not a part of the Invoice module. The following is the complete code and it is self-explained, so please go through the comments.

**namespace** *SOLID\_PRINCIPLES.SRP*

**{**

**public** **class** Invoice

**{**

**public** **long** InvAmount **{** **get**; **set**; **}**

**public** DateTime InvDate **{** **get**; **set**; **}**

**public** **void** AddInvoice**()**

**{**

**try**

**{**

// Here we need to write the Code for adding invoice

// Once the Invoice has been added, then send the mail

MailMessage mailMessage = new MailMessage**(**"EMailFrom", "EMailTo", "EMailSubject", "EMailBody"**)**;

this.SendInvoiceEmail**(**mailMessage**)**;

**}**

**catch** **(**Exception ex**)**

**{**

//Error Logging

System.IO.File.WriteAllText**(**@"c:\ErrorLog.txt", ex.ToString**())**;

**}**

**}**

**public** **void** DeleteInvoice**()**

**{**

**try**

**{**

//Here we need to write the Code for Deleting the already generated invoice

**}**

**catch** **(**Exception ex**)**

**{**

//Error Logging

System.IO.File.WriteAllText**(**@"c:\ErrorLog.txt", ex.ToString**())**;

**}**

**}**

**public** **void** SendInvoiceEmail**(**MailMessage mailMessage**)**

**{**

**try**

**{**

// Here we need to write the Code for Email setting and sending the invoice mail

**}**

**catch** **(**Exception ex**)**

**{**

//Error Logging

System.IO.File.WriteAllText**(**@"c:\ErrorLog.txt", ex.ToString**())**;

**}**

**}**

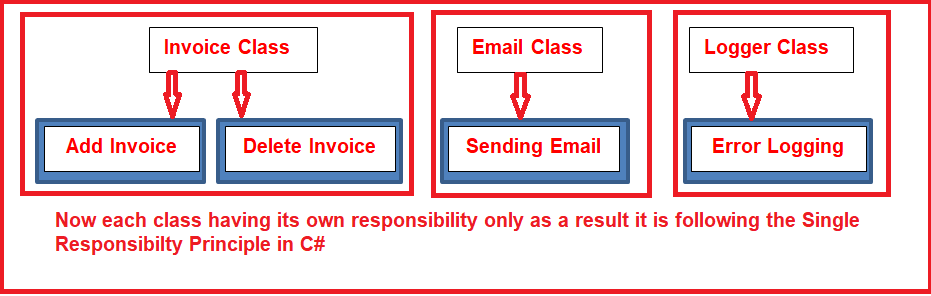
**}**

**}**

Now let us discuss how to implement the above functionalities in such a way that, it should follow the Single Responsibility Principle.

**Example: Implementing the SRP in C#**

Please have a look at the following diagram.



As you can see in the above diagram, now we are going to create three classes. In the invoice class, only the invoice-related functionalities are going to be implemented. The Logger class is going to be used only for logging purposes. Similarly, the Email class is going to handle Email activities. Now each class having only its own responsibilities, as a result, it follows the Single Responsibility Principle in C#.

**Logger.cs**

Add a class file with the name **Logger.cs** and then copy and paste the following code in it.

**namespace** *SOLID\_PRINCIPLES.SRP*

**{**

**public** **interface** ILogger

**{**

**void** Info**(**string info**)**;

**void** Debug**(**string info**)**;

**void** Error**(**string message, Exception ex**)**;

**}**

**public** **class** Logger : ILogger

**{**

**public** Logger**()**

**{**

// here we need to write the Code for initialization

// that is Creating the Log file with necesssary details

**}**

**public** **void** Info**(**string info**)**

**{**

// here we need to write the Code for info information into the ErrorLog text file

**}**

**public** **void** Debug**(**string info**)**

**{**

// here we need to write the Code for Debug information into the ErrorLog text file

**}**

**public** **void** Error**(**string message, Exception ex**)**

**{**

// here we need to write the Code for Error information into the ErrorLog text file

**}**

**}**

**}**

As you can see in the above class file, we are defining all the logging activities i.e. info, debug and error.

**MailSender.cs**

Now, we need to add another class file with the name **MailSender.cs** and then copy and paste the following code.

**namespace** *SOLID\_PRINCIPLES.SRP*

**{**

**public** **class** MailSender

**{**

**public** string EMailFrom **{** **get**; **set**; **}**

**public** string EMailTo **{** **get**; **set**; **}**

**public** string EMailSubject **{** **get**; **set**; **}**

**public** string EMailBody **{** **get**; **set**; **}**

**public** **void** SendEmail**()**

**{**

// Here we need to write the Code for sending the mail

**}**

**}**

**}**

In the above MailSender class, we are defining the send mail activities.

**Modifying the Invoice class:**

Finally, modify the Invoice class as shown below.

**using** *System.Net.Mail;*

**namespace** *SOLID\_PRINCIPLES.SRP*

**{**

**public** **class** Invoice

**{**

**public** **long** InvAmount **{** **get**; **set**; **}**

**public** DateTime InvDate **{** **get**; **set**; **}**

**private** ILogger fileLogger;

**private** MailSender emailSender;

**public** Invoice**()**

**{**

fileLogger = new Logger**()**;

emailSender = new MailSender**()**;

**}**

**public** **void** AddInvoice**()**

**{**

**try**

**{**

fileLogger.Info**(**"Add method Start"**)**;

// Here we need to write the Code for adding invoice

// Once the Invoice has been added, then send the mail

emailSender.EMailFrom = "emailfrom@xyz.com";

emailSender.EMailTo = "emailto@xyz.com";

emailSender.EMailSubject = "Single Responsibility Princile";

emailSender.EMailBody = "A class should have only one reason to change";

emailSender.SendEmail**()**;

**}**

**catch** **(**Exception ex**)**

**{**

fileLogger.Error**(**"Error Occurred while Generating Invoice", ex.Message**)**;

**}**

**}**

**public** **void** DeleteInvoice**()**

**{**

**try**

**{**

//Here we need to write the Code for Deleting the already generated invoice

fileLogger.Info**(**"Delete Invoice Start at @" + DateTime.Now**)**;

**}**

**catch** **(**Exception ex**)**

**{**

fileLogger.Error**(**"Error Occurred while Deleting Invoice", ex**)**;

**}**

**}**

**}**

**}**

As you can see, the Invoice class delegating the logging activity to the “**Logger**” class. In the same way, delegate the Email Sending activity to the “**MailSender**” class. Now, the Invoice class now only concentrates on Invoice related activities.

# Open-Closed Principle in C#

## ****Open-Closed Principle in C# with Examples****

In this article, I am going to discuss the **Open-Closed Principle in C#** with Examples. Please read our previous article before proceeding to this article where we discussed the [**Single Responsibility Principle in C#**](https://dotnettutorials.net/lesson/single-responsibility-principle/) with one real-time example. The letter **O** in S**O**LID stands for the Open-Closed Principle which is also known as **OCP**. As part of this article, we are going to discuss the following pointers in detail.

1. **What is the Open-Closed Principle in C#?**
2. **Implementation Guidelines for the Open-Closed Principle in C#**
3. **What Problems you will get if you are not following the Open-Closed Principle?**
4. **Example without using the Open-Closed Principle.**
5. **Example using the Open-Closed Principle in C#.**

##### ****What is the Open-Closed Principle in C#?****

The Open-Closed Principle states that “**software entities such as modules, classes, functions, etc. should be open for extension, but closed for modification**“.

Let us understand the above definition in simple words. Here we need to understand two things. The first thing is Open for extension and the second thing is Closed for modification. The Open for extension means we need to design the software modules/classes in such a way that the new responsibilities or functionalities should be added easily when new requirements come. On the other hand, Closed for modification means, we should not modify the class/module until we find some bugs.

The reason for this is, we have already developed a class/module and it has gone through the unit testing phase. So we should not have to change this as it affects the existing functionalities. In simple words, we can say that we should develop one module/class in such a way that it should allow its behavior to be extended without altering its source code.

##### ****Implementation Guidelines for the Open-Closed Principle (OCP) in C#****

1. The easiest way to implement the Open-Closed Principle in C# is to add the new functionalities by creating new derived classes which should be inherited from the original base class.
2. Another way is to allow the client to access the original class with an abstract interface.
3. So, at any given point of time when there is a change in requirement or any new requirement comes then instead of touching the existing functionality, it’s always better and suggested to create new derived classes and leave the original class implementation as it is.

##### ****Problems of Not following the Open-Closed Principle in C#:****

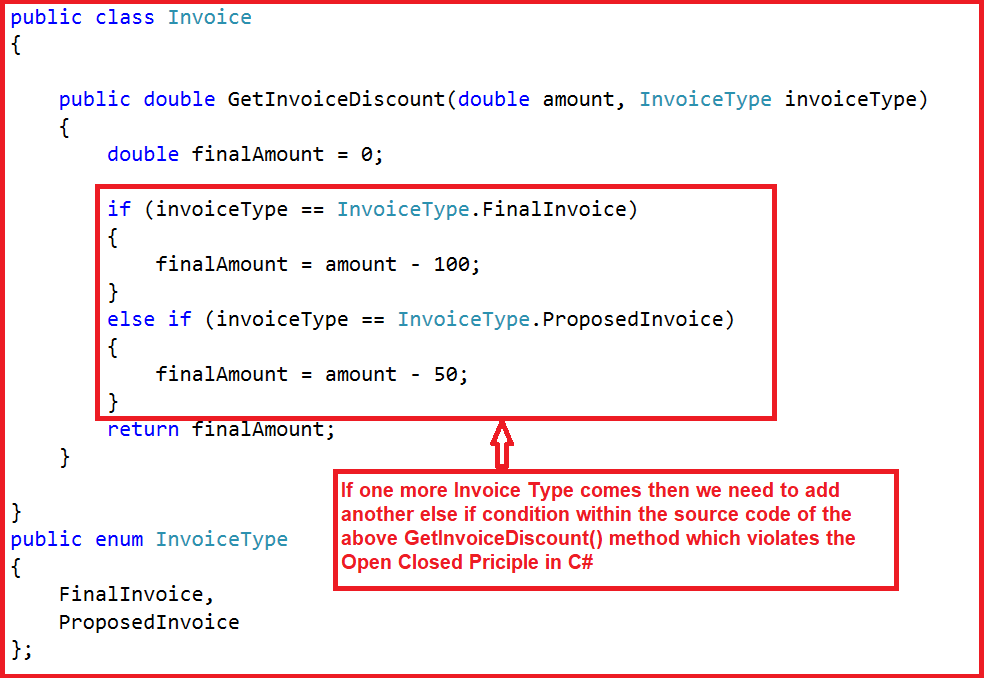
If you are not following the Open-Closed Principle during the application development process, then you may end up your application development with the following problems

1. If you allow a class or function to add new logic then as a developer you need to test the entire functionalities which include the old functionalities as well as new functionalities of the application.
2. As a developer, it is also your responsibility to tell the QA (Quality Assurance) team about the changes in advance so that they can prepare themselves in advance for regression testing along with the new feature testing.
3. If you are not following the Open-Closed Principle, then it also breaks the Single Responsibility Principle as the class or module is going to perform multiple responsibilities.
4. If you are implementing all the functionalities in a single class, then the maintenance of the class becomes very difficult.

Because of the above key points, we need to follow the open-closed principle in C# while developing the application.

##### ****Let us understand the Open-Closed Principle in C# with one example.****

Please have a look at the following diagram.



As you can see in the above image, within the Invoice class we have created the **GetInvoiceDiscount()** method. As part of that **GetInvoiceDiscount()** method, we are calculating the final amount based on the Invoice type. As of now, we have two Invoice Types as the Final Invoice and the Proposed Invoice. So we have implemented the logic using if-else. Tomorrow, if one more Invoice Type comes into the picture then we need to modify the **GetInvoiceDiscount()** method logic by adding another else if block to the source code. As we are changing the source code for the new requirement, we are violating the Open-Closed principle in C#.

##### ****Example: Without using the Open-Closed Principle in C#****

**namespace** *SOLID\_PRINCIPLES.OCP*

**{**

**public** **class** Invoice

**{**

**public** **double** GetInvoiceDiscount**(double** amount, InvoiceType invoiceType**)**

**{**

**double** finalAmount = 0;

**if** **(**invoiceType == InvoiceType.FinalInvoice**)**

**{**

finalAmount = amount - 100;

**}**

**else** **if** **(**invoiceType == InvoiceType.ProposedInvoice**)**

**{**

finalAmount = amount - 50;

**}**

**return** finalAmount;

**}**

**}**

**public** enum InvoiceType

**{**

FinalInvoice,

ProposedInvoice

**}**;

**}**

The problem with the above example is that if we want to add another new invoice type, then we need to add one more “else if” condition in the same “**GetInvoiceDiscount**” method, in other words, we need to modify the Invoice class. If we are changing the Invoice class again and again then we need to ensure that the previous functionalities along with the new functionalities are working properly by testing both the functionalities again. This is because we need to ensure that the existing clients, which are referencing this class are working properly as expected or not.

##### ****Open-Closed Principle in C#****

As per the Open-Closed principle,**Instead of MODIFYING, we should go for EXTENSION.**If you want to follow the Open-Closed Principle in the above example, when a new invoice type needs to be added, then we need to add a new class. As a result, the current functionalities that are already implemented are going to be unchanged. The advantage is that we just only need to test and check the new classes.

###### **Example: With Open-Closed Principle in C#**

The following code example follows Open Closed Principle (OCP) in C#.

**namespace** *SOLID\_PRINCIPLES.OCP*

**{**

**public** **class** Invoice

**{**

**public** **virtual** **double** GetInvoiceDiscount**(double** amount**)**

**{**

**return** amount - 10;

**}**

**}**

**public** **class** FinalInvoice : Invoice

**{**

**public** **override** **double** GetInvoiceDiscount**(double** amount**)**

**{**

**return** **base**.GetInvoiceDiscount**(**amount**)** - 50;

**}**

**}**

**public** **class** ProposedInvoice : Invoice

**{**

**public** **override** **double** GetInvoiceDiscount**(double** amount**)**

**{**

**return** **base**.GetInvoiceDiscount**(**amount**)** - 40;

**}**

**}**

**public** **class** RecurringInvoice : Invoice

**{**

**public** **override** **double** GetInvoiceDiscount**(double** amount**)**

**{**

**return** **base**.GetInvoiceDiscount**(**amount**)** - 30;

**}**

**}**

**}**

As you can see in the above code, we have created three classes **FinalInvoice**, **ProposedInvoice**, and **RecurringInvoice**. All these three classes are inherited from the base class Invoice and if they want then they can override the **GetInvoiceDiscount()** method. Tomorrow if another Invoice Type needs to be added then we just need to create a new class by inheriting it from the Invoice class. The point that you need to keep the focus on is we are not changing the code of the Invoice class.

###### **Modify the Main method of the Program class as shown below to test the application.**

**using** *System;*

**namespace** *SOLID\_PRINCIPLES.OCP*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Invoice FInvoice = new FinalInvoice**()**;

Invoice PInvoice = new ProposedInvoice**()**;

Invoice RInvoice = new RecurringInvoice**()**;

**double** FInvoiceAmount = FInvoice.GetInvoiceDiscount**(**10000**)**;

**double** PInvoiceAmount = PInvoice.GetInvoiceDiscount**(**10000**)**;

**double** RInvoiceAmount = RInvoice.GetInvoiceDiscount**(**10000**)**;

Console.ReadKey**()**;

**}**

**}**

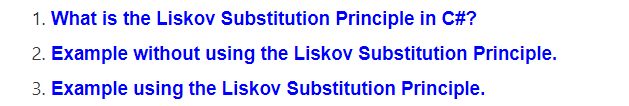
**}**

Now, the Invoice class is closed for modification. But it is open for the extension as it allows creating new classes deriving from the Invoice class which clearly follows the Open-Closed Principle in C#.

**Liskov Substitution Principle in C#**

**Liskov Substitution Principle in C# with Examples**

In this article, I am going to discuss the**Liskov Substitution Principle** **in C#** with Examples. Please read our previous article before proceeding to this article where we discussed the [**Open-Closed Principle in C#**](https://dotnettutorials.net/lesson/open-closed-principle/) with an example. The Letter **L** in **SOLID** stands for **Liskov Substitution Principle** which is also known as LSP. As part of this article, we are going to discuss the following pointers in detail.



**What is the Liskov Substitution Principle in C#?**

The **Liskov Substitution Principle**is a Substitutability principle in object-oriented programming Language. This principle states that, if **S** is a subtype of **T**, then objects of type **T** should be replaced with the objects of type **S**.

In simple words we can say that, when we have a base class and child class relationships i.e. inheritance relationships, then, if we can successfully replace the object/instance of a parent class with an object/instance of the child class, without affecting the behavior of the base class instance, then it is said to be in Liskov Substitution Principle. If you are not getting this point properly, don’t worry, we will see some real-time examples to understand this concept.

**For example,** a father is a teacher whereas his son is a doctor. So here, in this case, the son can’t simply replace his father even though both belong to the same family.

**Example: Without using the Liskov Substitution Principle in C#:**

Let us first understand one example without using the Liskov Substitution Principle in C#. In the following example, first, we create the Apple class with the method GetColor. Then we create the Orange class which inherits the Apple class as well as overrides the GetColor method of the Apple class. The point is that an Orange cannot be replaced by an Apple, which results in printing the color of the apple as Orange as shown in the below example.

**namespace** *SOLID\_PRINCIPLES.LSP*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Apple apple = new Orange**()**;

Console.WriteLine**(**apple.GetColor**())**;

**}**

**}**

**public** **class** Apple

**{**

**public** **virtual** string GetColor**()**

**{**

**return** "Red";

**}**

**}**

**public** **class** Orange : Apple

**{**

**public** **override** string GetColor**()**

**{**

**return** "Orange";

**}**

**}**

**}**

As you can see in the above example, Apple is the base class and Orange is the child class i.e. there is a Parent-Child relationship. So, we can store the child class object in the Parent Reference variable i.e. Apple apple = new Orange(); and when we call the GetColor i.e. apple.GetColor(), then we are getting the color of the Orange not the color of Apple. That means once the child object is replaced i.e. Apple storing the Orange object, the behavior is also changed. This is against the LSP Principle. The Liskov Substitution Principle in C# states that even the child object is replaced with the parent, the behavior should not be changed. So, in this case, if we are getting the color of Apple instead of Orange, then it follows the Liskov Substitution Principle. That means there is some issue with our software design. Let us see how to overcome the design issue and makes the application follow Liskov Substitution Principle.

**Example Using the Liskov Substitution Principle in C#**

Let’s modify the previous example to follow the Liskov Substitution Principle. Here, first, we need a generic base class such as Fruit which is going to be the base class for both Apple and Orange. Now you can replace the Fruit class object with its subtypes either Apple and Orage and it will behave correctly. Now, you can see in the below code, we created the super Fruit class as an abstract class with the GetColor abstract method and then the Apple and Orange class inherited from the Fruit class and implement the GetColor method.

**namespace** *SOLID\_PRINCIPLES.LSP*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Fruit fruit = new Orange**()**;

Console.WriteLine**(**fruit.GetColor**())**;

fruit = new Apple**()**;

Console.WriteLine**(**fruit.GetColor**())**;

**}**

**}**

**public** **abstract** **class** Fruit

**{**

**public** **abstract** string GetColor**()**;

**}**

**public** **class** Apple : Fruit

**{**

**public** **override** string GetColor**()**

**{**

**return** "Red";

**}**

**}**

**public** **class** Orange : Fruit

**{**

**public** **override** string GetColor**()**

**{**

**return** "Orange";

**}**

**}**

**}**

Now, run the application and it should give the output as expected. Here we are following the LSP as we are now able to change the object with its subtype.

# Interface Segregation Principle in C#

## ****Interface Segregation Principle in C# with Examples****

In this article, I am going to discuss the**Interface Segregation Principle in C#** with Examples. Please read our previous article before proceeding to this article where we discussed the **[Liskov Substitution Principle](https://dotnettutorials.net/lesson/liskov-substitution-principle/)****[in C#](https://dotnettutorials.net/lesson/liskov-substitution-principle/)** with a real-time example. The letter **I** in the SOL**I**D Design Principle stands for **Interface Segregation Principle**which is also known as **ISP**. As part of this article, we are going to discuss the following pointers in detail.

1. **What is the Interface Segregation Principle in C#?**
2. **Example without using the Interface Segregation Principle in C#.**
3. **Example using the Interface Segregation Principle in C#.**

##### ****What is the Interface Segregation Principle in C#?****

The Interface Segregation Principle states that **“Clients should not be forced to implement any methods they don’t use. Rather than one fat interface, numerous little interfaces are preferred based on groups of methods with each interface serving one submodule“.**

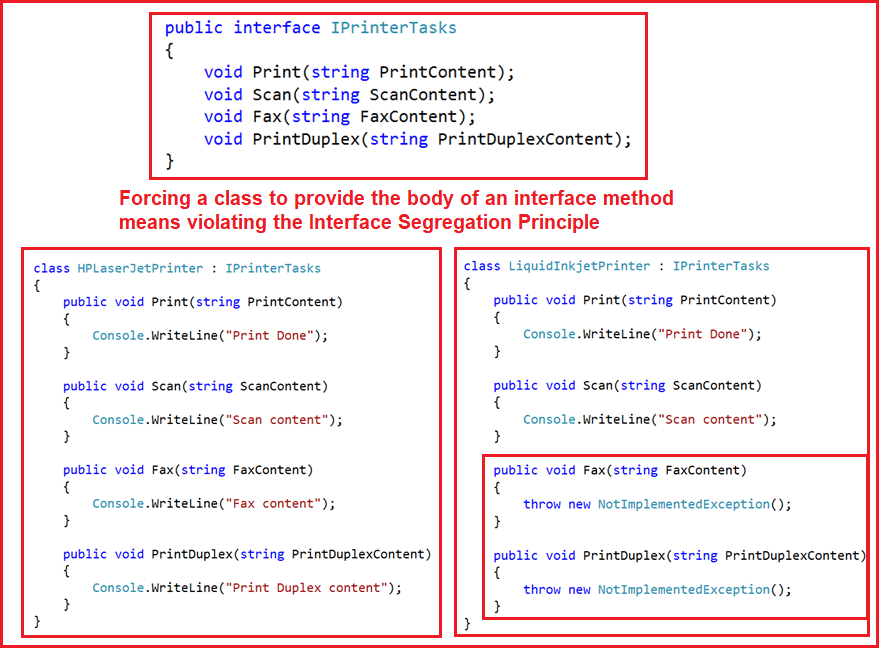
Let us break down the above definition into two parts.

1. First, no class should be forced to implement any method(s) of an interface they don’t use.
2. Secondly, instead of creating large or you can say fat interfaces, create multiple smaller interfaces with the aim that the clients should only think about the methods that are of interest to them.

As per the [**Single Responsibility Principle**](https://dotnettutorials.net/lesson/single-responsibility-principle/) of SOLID, like classes, interfaces also should have a single responsibility. That means we shouldn’t force any class to implement any method(s) which they don’t require.

##### ****Let us understand the Interface Segregation Principle in C# with an example.****

Please have a look at the following diagram.



As you can see in the above diagram, we have an interface i.e. IPrinterTasks declared with four methods. Now if any class wants to implement this interface then that class should have to provide the implementation to all the four methods of the IPrinterTasks interface. As you can see in the above diagram, we have two classes HPLaserJetPrinter and LiquidInkjetPrinter who want the printer service.

But the requirement is the HPLaserJetPrinter wants all the services provided by the IPrinterTasks while the LiquidInkjetPrinter wants only the Print and Scan service of the printer. As we have declared all the methods within the IPrinterTasks interface, then it is mandatory for the LiquidInkjetPrinter class to provide implementation to Scan and Print methods along with the Fax and PrinctDulex method which are not required by the class.

##### ****Example without using the Interface Segregation Principle in C#:****

**IPrinterTasks.cs**

**namespace** *SOLID\_PRINCIPLES.ISP*

**{**

**public** **interface** IPrinterTasks

**{**

**void** Print**(**string PrintContent**)**;

**void** Scan**(**string ScanContent**)**;

**void** Fax**(**string FaxContent**)**;

**void** PrintDuplex**(**string PrintDuplexContent**)**;

**}**

**}**

###### **HPLaserJetPrinter.cs**

**namespace** *SOLID\_PRINCIPLES.ISP*

**{**

**public** **class** HPLaserJetPrinter : IPrinterTasks

**{**

**public** **void** Print**(**string PrintContent**)**

**{**

Console.WriteLine**(**"Print Done"**)**;

**}**

**public** **void** Scan**(**string ScanContent**)**

**{**

Console.WriteLine**(**"Scan content"**)**;

**}**

**public** **void** Fax**(**string FaxContent**)**

**{**

Console.WriteLine**(**"Fax content"**)**;

**}**

**public** **void** PrintDuplex**(**string PrintDuplexContent**)**

**{**

Console.WriteLine**(**"Print Duplex content"**)**;

**}**

**}**

**}**

###### **LiquidInkjetPrinter.cs**

**namespace** *SOLID\_PRINCIPLES.ISP*

**{**

**class** LiquidInkjetPrinter : IPrinterTasks

**{**

**public** **void** Print**(**string PrintContent**)**

**{**

Console.WriteLine**(**"Print Done"**)**;

**}**

**public** **void** Scan**(**string ScanContent**)**

**{**

Console.WriteLine**(**"Scan content"**)**;

**}**

**public** **void** Fax**(**string FaxContent**)**

**{**

**throw** new NotImplementedException**()**;

**}**

**public** **void** PrintDuplex**(**string PrintDuplexContent**)**

**{**

**throw** new NotImplementedException**()**;

**}**

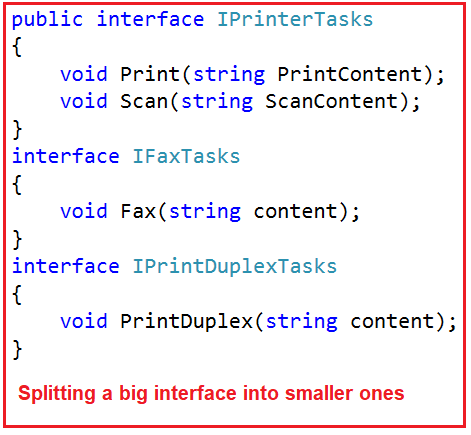
**}**

**}**

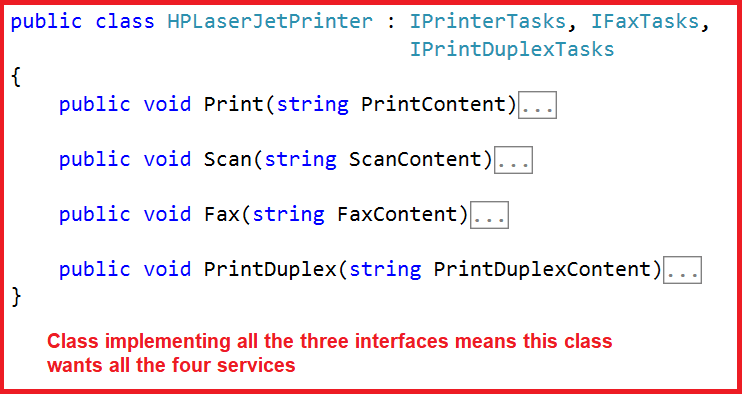
As you can see in the above LiquidInkjetPrinter class the Fax and PrintDuplex methods are not required by the class but, still, it is implementing these two methods. This is violating the Interface Segregation Principle in C# as we are forcing the class to implement two methods that they don’t require.

##### ****Example using the Interface Segregation Principle:****

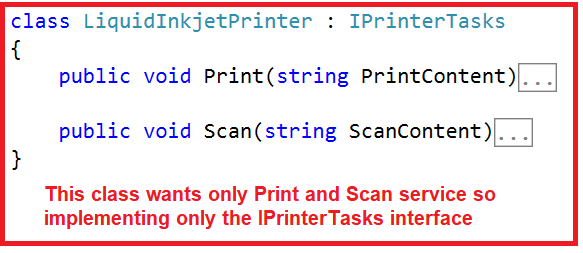
Please have a look at the following diagram.



As you can see in the above diagram, now we have split that big interface into three small interfaces. Each interface now having some specific purpose. Now if any class wants all the services then that class needs to implement all the three interfaces as shown below.



Now, if any class wants the Scan and Print service, then that class needs to implement only the IPrinterTasks interfaces as shown in the below image.



##### ****The Complete Code is given below:****

**namespace** *SOLID\_PRINCIPLES.ISP*

**{**

**public** **interface** IPrinterTasks

**{**

**void** Print**(**string PrintContent**)**;

**void** Scan**(**string ScanContent**)**;

**}**

**interface** IFaxTasks

**{**

**void** Fax**(**string content**)**;

**}**

**interface** IPrintDuplexTasks

**{**

**void** PrintDuplex**(**string content**)**;

**}**

**}**

**namespace** *SOLID\_PRINCIPLES.ISP*

**{**

**public** **class** HPLaserJetPrinter : IPrinterTasks, IFaxTasks,

IPrintDuplexTasks

**{**

**public** **void** Print**(**string PrintContent**)**

**{**

Console.WriteLine**(**"Print Done"**)**;

**}**

**public** **void** Scan**(**string ScanContent**)**

**{**

Console.WriteLine**(**"Scan content"**)**;

**}**

**public** **void** Fax**(**string FaxContent**)**

**{**

Console.WriteLine**(**"Fax content"**)**;

**}**

**public** **void** PrintDuplex**(**string PrintDuplexContent**)**

**{**

Console.WriteLine**(**"Print Duplex content"**)**;

**}**

**}**

**}**

**namespace** *SOLID\_PRINCIPLES.ISP*

**{**

**class** LiquidInkjetPrinter : IPrinterTasks

**{**

**public** **void** Print**(**string PrintContent**)**

**{**

Console.WriteLine**(**"Print Done"**)**;

**}**

**public** **void** Scan**(**string ScanContent**)**

**{**

Console.WriteLine**(**"Scan content"**)**;

**}**

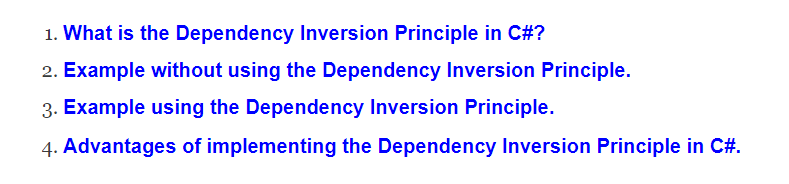
**}**

**}**

# Dependency Inversion Principle in C#

## ****Dependency Inversion Principle in C# with Examples****

In this article, I am going to discuss the **Dependency Inversion Principle in C#** with Examples. Please read our previous article before proceeding to this article where we discussed the [**Interface Segregation Principle in C#**](https://dotnettutorials.net/lesson/interface-segregation-principle/) with a real-time example. The Letter **D** in SOLI**D** stands for the **Dependency Inversion Principle** which is also known as **DIP**. At the end of this article, you will understand the following pointers in detail.



##### ****What is the Dependency Inversion Principle in C#?****

The **Dependency Inversion Principle** (DIP) states that **high-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions. Secondly, abstractions should not depend upon details. Details should depend upon abstractions**.

The most important point that you need to remember while developing real-time applications, always to try to keep the High-level module and Low-level module as loosely coupled as possible.

When a class knows about the design and implementation of another class, it raises the risk that if we do any changes to one class will break the other class. So we must keep these high-level and low-level modules/classes loosely coupled as much as possible. To do that, we need to make both of them dependent on abstractions instead of knowing each other.

##### ****Let us understand the Dependency Inversion Principle in C# with one example****

Let’s create one console application, Then create the following classes

##### ****Employee.cs****

Create a class file with the name Employee.cs and then copy and paste the following code into it. The following is a simple class having 4 properties.

**namespace** *SOLID\_PRINCIPLES.DIP*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Department **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

##### ****EmployeeBusinessLogic.cs****

Create a class file with the name EmployeeBusinessLogic.cs and then copy and paste the following code into it. The following class has one constructor that is used to create an instance of EmployeeDataAccess. Here, within the constructor we call the static GetEmployeeDataAccessObj() method on the DataAccessFactory class which will return an instance of EmployeeDataAccess and we initialize the \_EmployeeDataAccess property with the return instance. We have also one method i.e. GetEmployeeDetails which is used to call the GetEmployeeDetails method on the EmployeeDataAccess instance to get the employee detail by employee id.

**namespace** *SOLID\_PRINCIPLES.DIP*

**{**

**public** **class** EmployeeBusinessLogic

**{**

EmployeeDataAccess \_EmployeeDataAccess;

**public** EmployeeBusinessLogic**()**

**{**

\_EmployeeDataAccess = DataAccessFactory.GetEmployeeDataAccessObj**()**;

**}**

**public** Employee GetEmployeeDetails**(int** id**)**

**{**

**return** \_EmployeeDataAccess.GetEmployeeDetails**(**id**)**;

**}**

**}**

**}**

##### ****DataAccessFactory.cs****

Create a class file with the name DataAccessFactory.cs and then copy and paste the following code into it. The following class contains one static method which is returning an instance of the EmployeeDataAccess class.

**namespace** *SOLID\_PRINCIPLES.DIP*

**{**

**public** **class** DataAccessFactory

**{**

**public** **static** EmployeeDataAccess GetEmployeeDataAccessObj**()**

**{**

**return** new EmployeeDataAccess**()**;

**}**

**}**

**}**

###### **EmployeeDataAccessLogic.cs**

Create a class file with the name EmployeeDataAccess.cs and then copy and paste the following code into it. The following class contains one method which takes the employee id and returns that Employee information.

**namespace** SOLID\_PRINCIPLES.DIP

**{**

**public** **class** EmployeeDataAccess

**{**

**public** Employee GetEmployeeDetails**(int** id**)**

**{**

// In real time get the employee details from db

//but here we are hard coded the employee details

Employee emp = new Employee**()**

**{**

ID = id,

Name = "Pranaya",

Department = "IT",

Salary = 10000

**}**;

**return** emp;

**}**

**}**

**}**

##### ****Let us compare the above example with the Dependency Inversion Principle in C#****

As per the **Dependency Inversion Principle**definition**, “a high-level module should not depend on low-level modules. Both should depend on the abstraction”.**

So, first, we need to figure out which one is the high-level module (class) and which one is the low-level module (class) in our example. A high-level module is a module that always depends on other modules. So, in our example, the **EmployeeBusinessLogic** class depends on **EmployeeDataAccess**class, so here the **EmployeeBusinessLogic**class is the high-level module and the **EmployeeDataAccess**class is the low-level module.

So, as per the first rule of the Dependency Inversion Principle in C#, the**EmployeeBusinessLogic**class/module should not depend on the concrete **EmployeeDataAccess**class/module, instead, both the classes should depend on abstraction.

The second rule of the **Dependency Inversion Principle**state that **“Abstractions should not depend on details. Details should depend on abstractions”.**

Before understanding this let us first understand what is an abstraction.

##### ****What is Abstraction?****

In simple words, we can say that Abstraction means something which is non-concrete. So, abstraction in programming means we need to create either an interface or abstract class which is non-concrete so that we can not create an instance of it. In our example, the EmployeeBusinessLogic and EmployeeDataAccess are concrete classes that mean we can create objects of it.

As per the Dependency Inversion Principle in C#, the EmployeeBusinessLogic (high-level module) should not depend on the concrete EmployeeDataAccess (low-level module) class. Both classes should depend on abstractions, meaning both classes should depend on either an interface or an abstract class.

##### ****What should be in the interface (or in the abstract class)?****

As you can see in the above example, the EmployeeBusinessLogic uses the GetEmployeeDetails() method of EmployeeDataAccess class. In real-time, there will be many employee-related methods in the EmployeeDataAccess class. So, we need to declare the GetEmployeeDetails(int id) method within the interface. Add one interface with the name IDataAccess and then copy and paste the following codes.

###### **IDataAccess.cs**

Create a class file with the name IEmployeeDataAccess.cs and then copy and paste the following code into it. As you can see, here we created the interface with one abstract method i.e. GetEmployeeDetails.

**namespace** *SOLID\_PRINCIPLES.DIP*

**{**

**public** **interface** IEmployeeDataAccess

**{**

Employee GetEmployeeDetails**(int** id**)**;

**}**

**}**

Now, we need to implement the **IEmployeeDataAccess** in **EmployeeDataAccess** class. So, modify the **EmployeeDataAccess**class as shown below

###### **EmployeeDataAccess.cs**

**namespace** *SOLID\_PRINCIPLES.DIP*

**{**

**public** **class** EmployeeDataAccess : IEmployeeDataAccess

**{**

**public** Employee GetEmployeeDetails**(int** id**)**

**{**

// In real time get the employee details from db

//but here we are hardcoded the employee details

Employee emp = new Employee**()**

**{**

ID = id,

Name = "Pranaya",

Department = "IT",

Salary = 10000

**}**;

**return** emp;

**}**

**}**

**}**

Now, we need to change the factory class which will return the IEmployeeDataAccess instead of the concrete EmployeeDataAccess class as shown below.

###### **DataAccessFactory.cs**

Modify the DataAccessFactory class below. Please notice here we have changed the return type of the GetEmployeeDataAccessObj method from EmployeeDataAccess to IEmployeeDataAccess.

**namespace** *SOLID\_PRINCIPLES.DIP*

**{**

**public** **class** DataAccessFactory

**{**

**public** **static** IEmployeeDataAccess GetEmployeeDataAccessObj**()**

**{**

**return** new EmployeeDataAccess**()**;

**}**

**}**

**}**

Now, we need to change the EmployeeBusinessLogic class which will use the IEmployeeDataAccess instead of the concrete EmployeeDataAccess class as shown below.

**namespace** *SOLID\_PRINCIPLES.DIP*

**{**

**public** **class** EmployeeBusinessLogic

**{**

IEmployeeDataAccess \_EmployeeDataAccess;

**public** EmployeeBusinessLogic**()**

**{**

\_EmployeeDataAccess = DataAccessFactory.GetEmployeeDataAccessObj**()**;

**}**

**public** Employee GetEmployeeDetails**(int** id**)**

**{**

**return** \_EmployeeDataAccess.GetEmployeeDetails**(**id**)**;

**}**

**}**

**}**

That’s it. We have implemented the Dependency Inversion Principle in our example where the high-level module (EmployeeBusinessLogic) and low-level module (EmployeeDataAccess) depend on abstraction (IEmployeeDataAccess). Also, abstraction (IEmployeeDataAccess) does not depend on details (EmployeeDataAccess) but details depend on abstraction.

##### ****Advantages of implementing the Dependency Inversion Principle in C#:****

Now, the EmployeeBusinessLogic and EmployeeDataAccess classes are loosely coupled classes because EmployeeBusinessLogic does not depend on concrete EmployeeDataAccess class, instead, it includes a reference of IEmployeeDataAccess interface. So now, we can easily use another class that implements IEmployeeDataAccess with a different implementation.