

# SOLID Design Principles

The **SOLID Design Principles in C#** are the design principles that help us to solve most of the software design problems. These design principles provide us with multiple ways to remove the **tightly coupled** code between the software components which makes the software designs **more understandable, more flexible, and more maintainable**.

## Why Do We Need to Learn SOLID Design Principles?

As a developer, we start developing applications using our experience and knowledge. But over time, the applications might arise bugs. We need to alter the application design for every change request or for a new feature request. After some time, we might need to put in a lot of effort, even for simple tasks, it might require the full working knowledge of the entire system. But we cannot blame the change request or new feature requests as they are part of the software development. We cannot stop them and we cannot refuse them either. So, who is the culprit here? Obviously, it is the Design of the Application.

## Advantages of SOLID Design Principles in C#

As a developer, while you are designing and developing any software applications, then you need to consider the following points.

1. **Maintainability:** Nowadays maintaining the software is the biggest challenge for the industry. Day by day the business grows for the organization and as the business grows you need to enhance the software with new changes or with existing feature modifications or enhancements. So, you need to design the software in such a way that it should accept future changes and existing feature modifications or enhancements with minimum effort and without any problem.
2. **Testability:** Test-Driven Development (TDD) is one of the most important key aspects nowadays when you need to design and develop a large-scale application. We need to design the application in such a way that we should test each individual functionality.
3. **Parallel Development:** The Parallel Development of an application is one of the most important key aspects. As we know it is not possible to have the entire development team will work on the same module at the same time. So, we need to design the software in such a way that different teams can work on different modules of the project.

The **SOLID Design Principles** and **Design Patterns** play an important role in achieving all of the above key points.

## What are the Main Reasons Behind for Most Unsuccessful Applications?

The following are the Main Reasons Behind Most Unsuccessful Applications.

1. Putting More Functionalities on Classes. (In simple words a lot of functionalities we are putting into the class even though they are not related to that class).
2. Implementing Tight Coupling Between the Software Components (i.e. Between the Classes). If the classes are dependent on each other, then a change in one class will affect the other classes also.

## How to Overcome the Unsuccessful Application Development Problems?

1. We need to use the Correct Architecture (i.e. MVC, Layered, 3-Tier, MVP, and so on) as per the Project Requirements.
2. As a developer, we need to follow the Design Principles (i.e. SOLID Principles).
3. Again we need to choose the correct Design Patterns (**Creational Design Pattern**, **Structural Design Pattern**, **Behavioral Design Pattern**, **Dependency Injection Design Pattern**, **Repository Design Pattern**, etc.) as per the project requirements.

## What are SOLID Design Principles?

The **SOLID Design Principles** are the Design Principles that basically used to manage most of the Software Design Problems that generally we as a developer encountered in our day-to-day programming. These design principles are tested and proven mechanisms that will make the software designs more understandable, more flexible, and more maintainable. As a result, if we follow these principles while designing our application, then we can develop better applications.

**SOLID Design Principles** represent five Design Principles. The Five **SOLID** Design Principles are as follows:

1. **S** stands for the **Single Responsibility Principle** which is also known as **SRP**: The Single Responsibility Principle 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.
2. **O** stands for the **Open-Closed Principle** which is also known as **OSP**: The Open-Closed Principle states that **software entities such as modules, classes, functions, etc. should be open for extension, but closed for modification**.
3. **L** stands for the **Liskov Substitution Principle** which is also known as **LSP**: The Liskov Substitution Principle says **that the object of a derived class should be able to replace an object of the base class without bringing any errors in the system or modifying the behavior of the base class**. That means child class objects should be able to replace parent class objects without compromising application integrity.
4. **I** stand for the **Interface Segregation Principle** which is also known as **ISP**: 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**.
5. **D** stands for **Dependency Inversion Principle** which is also known as **DIP**: 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**.

## Single Responsibility Principle (SRP)

### What is the Single Responsibility Principle?

The Single Responsibility Principle 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. Before Proceeding further and understanding the Single Responsibility Principle, first we need to understand What is Responsibility.

### What is Responsibility?

An application can have many functionalities (features). For example, if you are developing an e-commerce application, then that application may have many features or functionalities such as Registering users, providing login functionality, displaying the product list, allowing the user to place an order, Providing Payment Functionality, Shipping the Order, Billing the Order, Logging the Order Information for Auditing and for Security purpose, sending the Order Invoice to the customer, etc. You can consider these functionalities or features as responsibilities. And another point that you need to remember is, changing the functionality means you need to change the class that is responsible for that functionality.

### How can we apply the Single Responsibility Principle in C#?

We can improve the application code readability, maintainability, and flexibility by applying the Single Responsibility Principle (SRP). SRP is one of the SOLID principles that states a class should have only one reason to change. That means a class should not mixed with multiple responsibilities. So, each class should be designed to perform a specific task only. Let us understand how we can apply the Single Responsibility Principle in C# while designing the classes:

- **Identify Responsibilities:** First, we need to Identify the different functionalities or responsibilities that a class should have, such as data access, validation, logging, caching, serialization, user authentication, etc.
- **Decompose into Smaller Classes:** Once we Identify the responsibilities, we must create separate classes for each responsibility. We should not mix multiple functionalities or responsibilities in a single class.
- **Separate Concerns:** Finally, we need to make sure that each class should only perform one specific task.

### Examples to Understand the Single Responsibility Principle using C#:

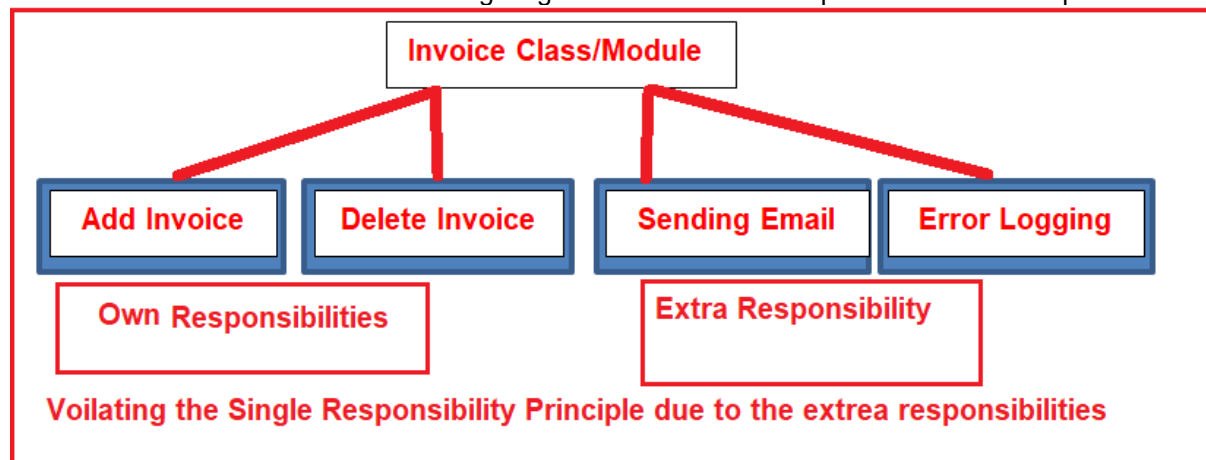
Let us understand the Single Responsibility Principle in C# with an Example. Suppose we need to design an Invoice class. As we know an Invoice class basically used to calculate various amounts based on its data. The Invoice class does not know how to retrieve the data, or how to format the data for displaying,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

If we write the database logic, and business logic as well as display logic in a single class, then our class performs 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#:

Let us first see the example without following the Single Responsibility Principle in C#, then we will see the problems if we are not following the Single Responsibility Principle and then we will see how we can overcome the problem using Single Responsibility Principle so that you will get a better idea of SRP. Please have a look at the following diagram that we want to implement in our example.



As you can see in the above image, we are going to create an Invoice class with four functionalities, i.e., 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 is not a part of the Invoice module. The following is the complete code and it is self-explained, so please go through the comments.

```

using System;
using System.Net.Mail;
namespace SOLID_PRINCIPLES.SRP
{
    public class Invoice
    {
        public long InvoiceAmount { get; set; }
        public DateTime InvoiceDate { 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());
    }
}

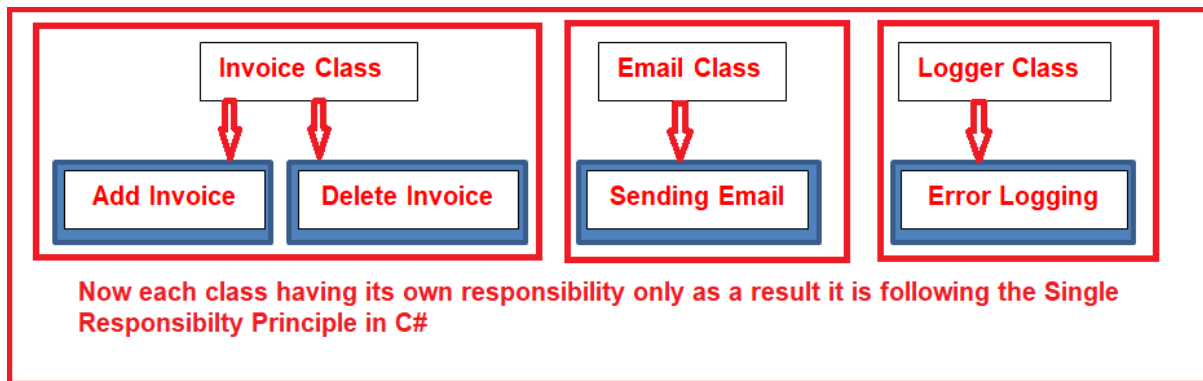
public void SendInvoiceEmail(MailMessage mailMessage)
{
    try
    {
        // Here we need to write the Code for Email setting and sending the invoice mail
    }
    catch (Exception ex)
    {
        //Error Logging
        System.IO.File.WriteAllText(@"c:\ErrorLog.txt", ex.ToString());
    }
}
}

```

With this design, if we want to modify the logging functionality or if we want to modify the sending email functionality, then we need to modify the Invoice class. This violates the Single Responsibility Principle as we are changing the Invoice Class for other functionality. If we do the changes, then we need to test the logging and email functionality along with the invoicing functionality. Now let us discuss how to implement the above functionalities in such a way that, it should follow the Single Responsibility Principle using C# Language.

### Example Implementing Single Responsibility Principle in C#

Let us implement the same example by following SRP. 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 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 has only its own responsibilities, as a result, it follows the Single Responsibility Principle in C#. Now, if you want to modify the email functionality, then you need to change the Email class only, not the Invoice and Logging class. Similarly, if you want to modify the Invoice functionalities, then you need to change the Invoice class only, not the Email and Logging class. Now, let us proceed and see how to implement this using C#.

### Logger.cs

Add a class file with the name **Logger.cs** and then copy and paste the following code into it. As you can see in the below class, we are defining all the logging activities, i.e., Info, Debug, and Error. The point that you need to remember is the class can contain multiple methods as long as all the methods belong to the same responsibility. Here, we are creating an Interface with the name **ILogger** with three abstract methods (by default interface methods are abstract). And then we implement the **ILogger** interface methods within the **Logger** class. The methods Info, Debug, and Error are going to perform different logging activities and hence putting all these methods within the **Logger** class.

```
using System;
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 necessary 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
        }
    }
}
```

## MailSender.cs

Now, we need to add another class file with the name **MailSender.cs** and then copy and paste the following code into it. In the below MailSender class, we are defining the send mail activities. So, here, you can see, for sending email whatever properties are required that we are putting here along with the SendEmail method. For simplicity, we have not provided the logic to send the email, but in real-time, you need to write the logic within the SendEmail method to send an email.

```
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
        }
    }
}
```

## Modifying Invoice Class:

Finally, modify the Invoice class as shown below. Within this class, we are only defining the Invoice related activities. As you can see, the Invoice class delegates the logging activity to the “**Logger**” class. In the same way, it also delegates the Email Sending activity to the “**MailSender**” class. Now, the Invoice class only concentrates on Invoice related activities.

```
using System;
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);
    }
}
}
}

```

This is how we need to design the application by following Single Responsibility Principle in C#. Now, each class has its own responsibilities. That means now **Each software module or class should have only one reason to change.**

### Advantages and Disadvantages of the Single Responsibility Principle:

The Single Responsibility Principle (SRP) is one of the SOLID principles of Object-Oriented Design to ensure well-structured, maintainable, and loosely coupled code. The Single Responsibility Principle (SRP) states that a class should only have one responsibility or reason for change. As a result, each class should be designed to perform a specific task. By dividing complex classes into smaller, more specialized ones where each class is responsible for a specific task, the Single Responsibility Principle (SRP) enhances code readability, maintainability, flexibility, and adaptability. Let's understand the advantages and disadvantages of following the Single Responsibility Principle (SRP) in C#:

#### Advantages:

- **Enhanced Readability and Maintainability:** When a class has a single responsibility, it becomes easier to understand its purpose and behavior. Changes related to a specific responsibility affect only one part of the code, making maintenance less error-prone and reducing the risk of introducing bugs.
- **Easier Testing:** Classes with one responsibility are easier to test. Tests focus on specific functionality, making unit tests simpler.
- **Increased Reusability:** Well-structured classes with a single responsibility are more reusable in different application parts, improving code organization and modularity.
- **Flexibility in Evolution:** Classes following the Single Responsibility Principle (SRP) are more flexible over time. You can modify or replace components without affecting other parts of the system.
- **Clearer Design:** Developers who follow the Single Responsibility Principle (SRP) must consider the design of their classes deeply and carefully. This principle breaks down a complex task into smaller, more manageable parts.
- **Reduce the Complexity:** It will reduce the Complexity of the application code. A code is based on its functionality. A class holds the logic for a single functionality. So, it reduces the complexity of the application code.
- **Reduce Tight Coupling:** It will reduce the tight coupling and dependency between software components. Changes in one method will not affect another.

#### Disadvantages:

- **Increased the Number of Classes:** Following the Single Responsibility Principle (SRP) may lead to a larger number of smaller classes, each responsible for a specific task. This can sometimes make the codebase feel more fragmented and complex.
- **Learning Curve for Developers:** Adhering to the Single Responsibility Principle (SRP) leads to better code quality. However, it may require developers to have a deeper understanding of the application's design and software engineering principles.
- **Initial Overhead:** Designing and structuring classes to follow the Single Responsibility Principle (SRP) may require extra effort initially, but it pays off in the long run with improved maintainability and reduced debugging.
- **Potential for Over-Splitting:** There's a risk of over-applying the Single Responsibility Principle (SRP), leading to too many small classes with minimal functionality. Finding the right



balance between having small, focused classes and maintaining a manageable number of classes.

### Use Cases of Single Responsibility Principle in C#:

The Single Responsibility Principle (SRP) encourages focusing classes on a single responsibility. This helps improve code readability, maintainability, and flexibility. Here are some common use cases where applying the SRP in C# is beneficial:

- **User Authentication and Authorization:** Split user authentication and authorization logic. Authentication handles login and token generation; authorization manages access based on roles and permissions.
- **Data Access and Business Logic:** It's beneficial to separate data access code (such as database queries) from business logic. This promotes the separation of concerns and simplifies switching between different data sources or storage technologies.
- **Input Validation and Processing:** Separate input validation from data processing logic. Validation enforces rules while processing performs actions.
- **Logging and Business Logic:** Keep logging code separate from core business logic. This lets you change the logging implementation without affecting the main application logic.
- **Serialization and Domain Logic:** Domain objects should not be tightly coupled to serialization. It is better to separate serialization (converting objects to/from JSON, XML, etc.) from domain logic.
- **UI Presentation and Data Retrieval:** Separate data retrieval logic from UI presentation, especially in web apps where rendering and fetching data are in separate components.
- **File I/O and Data Processing:** Separate file I/O operations (reading/writing files) from data processing. This makes it easier to adapt to different data sources or storage mechanisms.
- **Validation and Exception Handling: Keeping validation logic and exception handling separate is important.** Validation ensures input data correctness, while exception handling addresses unexpected errors.
- **Service Providers and Core Logic:** Isolate external dependencies by separating service providers (e.g., email sending and external API calls) from the core application logic. This facilitates changing providers without affecting the core logic.
- **Caching and Data Retrieval:** Separate data retrieval from caching logic to enable transparent caching.
- **Configuration and Application Logic:** Separate configuration-related code from core application logic to simplify configuration changes and updates.
- **Reporting and Data Processing:** Separate report generation from data processing for improved functionality.

For Multiple Real-Time Examples of the Single Responsibility Principle (SRP) using C#, please check the below link:

<https://dotnettutorials.net/lesson/real-time-examples-of-single-responsibility-principle-in-csharp/>



# Open-Closed Principle (OCP)

## What is the Open-Closed Principle?

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/functions** 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/function until we find some bugs.

The reason for this is, we have already developed a Class/Module/Function and it has already 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 Class/Module/Function in such a way that it should allow its behaviour to be extended without altering its source code. That means we should not edit the code of a method (until we find some bugs) instead we should use polymorphism or other techniques to add new functionality by writing new code.

## Implementation Guidelines for Open-Closed Principle (OCP) using C#

1. The easiest way to implement the Open-Closed Principle in C# is to add 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.

So, at any given point in time when there is a change in requirement or any new requirement comes then instead of touching the existing functionality, it is always better and suggested to create new derived classes and leave the original class implementation as it is. Let us understand this with an example, First, we will see the example by not following the Open-Close Principle, then we will understand the problems if we are not following the OCP and finally we will see the same example by following the Open-Close Principle using C# Language so that you will get a better idea of this Principle.

## Example to Understand Open-Closed Principle in C#.

Please have a look at the following class. As you can see in the below 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 i.e. Final Invoice and Proposed Invoice. So, we have implemented the logic using the if-else condition.

```

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
};

```

If one more Invoice Type comes then we need to add another else if condition within the source code of the above GetInvoiceDiscount() method which violates the Open Closed Principle in C#

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 GetInvoiceDiscount Method. If we are changing the Invoice class GetInvoiceDiscount Method again and again, then we need to ensure that the previous functionalities

along with the new functionalities are working properly by testing the existing 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.

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

So, if you are not following the Open-Closed Principle during the application development process, then you may end up with 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.

### 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 following Open-Closed Principle in C#

The following example shows Open Closed Principle (OCP) in C#. As you can see in the below 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 which is declared as Virtual in the Base Invoice class.

```
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;
        }
    }
}
```

```

    }
}

```

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 and if needed then we need to override the GetInvoiceDiscount() method. The point that you need to keep focus on is we are not changing the code of the Invoice class. Now, the Invoice class is **Closed for Modification**. But it is **Open for Extension** as it allows the creation of new classes deriving from the base Invoice class which clearly follows the Open-Closed Principle in C#.

Now, modify the Main method of the Program class as shown below to test the application and see whether the different Invoice types are working as expected or not as per our business requirement.

```

using System;
namespace SOLID_PRINCIPLES.OCP
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Invoice Amount: 10000");

            Invoice FInvoice = new FinalInvoice();
            double FInvoiceAmount = FInvoice.GetInvoiceDiscount(10000);
            Console.WriteLine($"Final Invoice : {FInvoiceAmount}");

            Invoice PInvoice = new ProposedInvoice();
            double PInvoiceAmount = PInvoice.GetInvoiceDiscount(10000);
            Console.WriteLine($"Proposed Invoice : {PInvoiceAmount}");

            Invoice RInvoice = new RecurringInvoice();
            double RInvoiceAmount = RInvoice.GetInvoiceDiscount(10000);
            Console.WriteLine($"Recurring Invoice : {RInvoiceAmount}");
            Console.ReadKey();
        }
    }
}

```

## Advantages and Disadvantages of Open-Closed Principle in C#

The Open-Closed Principle (OCP) is one of the SOLID principles of object-oriented design. It states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. Let's see the advantages and disadvantages of following the Open-Closed Principle in C#:

### Advantages:

- **Maintainability:** Following the Open/Closed Principle (OCP) reduces the need to modify well-tested code when adding new features, thereby ensuring system stability and reliability.
- **Code Reusability:** The Open-Closed Principle (OCP) encourages extending existing classes or components to create new functionality, resulting in reusable and modular code.
- **Reduced Risk of Regression Bugs:** When adding new features, existing code remains unchanged, significantly reducing the risk of introducing bugs or breaking existing functionality.
- **Encourages Design Patterns:** The Open-Closed Principle (OCP) promotes using design patterns such as Strategy, Decorator, and Factory. These patterns help in the creation of software systems that are both flexible and extensible.
- **Scalability:** Software Systems following the Open-Closed Principle (OCP) can easily adapt to new requirements and features over time.

### Disadvantages:

- **Initial Complexity:** Considering the Open-Closed Principle requires additional effort and planning when designing systems. This involves creating abstraction layers and interfaces to allow for extensibility.
- **Learning Curve:** To implement open-closed behavior, developers must learn and apply appropriate design patterns, which might require a learning curve.
- **Limited Applicability:** Some system parts may not be compatible with the Open-Closed Principle (OCP). Certain changes could necessitate modifications to existing code depending on the nature of the modification.
- **Abstraction Overhead:** Creating abstractions for every possible extension point could introduce unnecessary abstraction overhead.
- **Performance Impact:** Adding abstractions and extra layers might decrease performance due to added indirection.

### Use Cases of Open-Closed Principle in C#:

The Open-Closed Principle (OCP) encourages software entities to be open for extension and closed for modification. You can add new functionality to a module or class without changing its code. Here are some common use cases where you can apply the Open-Closed Principle effectively in C#:

- **Plugin Systems:** Create a plugin system where new features can be added to an application without modifying the core code. Each plugin can extend the application's behavior by adhering to a common interface.
- **Strategy Pattern:** Use the Strategy pattern to encapsulate different algorithms or strategies in separate classes. The main class can switch between strategies without changing its code.
- **Decorator Pattern:** Apply the Decorator pattern to dynamically add or modify objects' behavior. Decorator classes can wrap existing objects and extend functionality without changing the original code.
- **Abstract Factories:** Implement abstract factories to create families of related objects. You can create new factories without changing existing code when introducing new object variations.
- **Template Method Pattern:** Define a template method with a fixed structure and let subclasses provide specific implementations. You can add new behavior by creating new subclasses without modifying the template method.
- **Event Handlers:** In C#, use events and delegates to enable external code to extend the behavior of a class. Subscribers can respond to events raised by the class without changing its code.
- **Extensions and Interfaces:** Create interfaces and extension methods to allow external classes to extend the behavior of existing classes without modifying their code.
- **Abstract Classes and Inheritance:** Use abstract base classes to provide common functionality and derive new classes from the base class to add specialized behavior. This way, you can introduce new classes without modifying the base class.
- **Customizable Components:** Develop components with customizable behavior by using configuration or parameterization. This enables users to extend or modify the component's behavior without changing its implementation.
- **Dependency Injection:** Use dependency injection to provide components or services to classes. This lets you swap implementations or extend behavior without changing the class's code.
- **Pluggable Algorithms:** Implement algorithms as separate classes and provide a way to plug in different algorithms. This allows you to introduce new algorithms without modifying existing code.
- **Command Pattern:** Use the Command pattern to encapsulate actions as objects. You can introduce new commands without altering the existing code that uses them.

Applying the Open-Closed Principle in these use cases creates more flexible and extensible systems. New features and functionalities can be introduced without altering existing code, promoting maintainability and reducing the risk of introducing bugs.

For Multiple Real-Time Examples of the Open-Closed Principle (OCP) using C#, please check the below link:

<https://dotnettutorials.net/lesson/real-time-examples-of-open-closed-principle-in-csharp/>

# Liskov Substitution Principle (LSP)

## 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 objects of type **S**.

So, the Liskov Substitution Principle says that the object of a derived class should be able to replace an object of the base class without bringing any errors in the system or modifying the behavior of the base class. That means child class objects should be able to replace parent class objects without compromising application integrity.

In simple words, we can say that when we have Parent-Child relationships, i.e., Inheritance Relationships between two classes, then if we 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, 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#, then we will see the problem if we are not following the Liskov Substitution Principle and then we will see how we can overcome such problems using Liskov Substitution Principle. 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.

```
using System;
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 class Reference variable i.e. **Apple apple = new Orange();** and when we call the GetColor i.e. **apple.GetColor()**, then

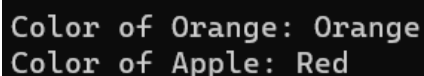
we are getting the color Orange, not the color of an Apple. That means once the child object is replaced i.e. Apple storing the Orange object, the behavior is also changed. This is against the LSP Principle. The Liskov Substitution Principle states that even if the child object is replaced with the parent, 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 using C# Language.

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

Let us modify the previous example to follow the Liskov Substitution Principle using C# Language. Here, first, we need a generic base Interface i.e. IFruit which is going to be the base class for both Apple and Orange classes. Now, you can replace the IFruit variable can be replaced with its subtypes either Apple or Orange and it will behave correctly. Now, you can see in the below code, we created the super IFruit as an interface with the GetColor method and then the Apple and Orange classes inherited from the Fruit class and implement the GetColor method.

```
using System;
namespace SOLID_PRINCIPLES.LSP
{
    class Program
    {
        static void Main(string[] args)
        {
            IFruit fruit = new Orange();
            Console.WriteLine($"Color of Orange: {fruit.GetColor()}");
            fruit = new Apple();
            Console.WriteLine($"Color of Apple: {fruit.GetColor()}");
            Console.ReadKey();
        }
    }
    public interface IFruit
    {
        string GetColor();
    }
    public class Apple : IFruit
    {
        public string GetColor()
        {
            return "Red";
        }
    }
    public class Orange : IFruit
    {
        public string GetColor()
        {
            return "Orange";
        }
    }
}
```

Now, run the application and it should give the output as expected as shown in the below image. Here we are following the LSP as we are now able to change the object with its subtype without affecting the behavior.

A screenshot of a terminal window showing the output of the C# program. The first line is "Color of Orange: Orange" and the second line is "Color of Apple: Red".

```
Color of Orange: Orange
Color of Apple: Red
```

So, now Fruit can be any type and any color, but orange cannot be the color red and an apple cannot be of the color orange, meaning we cannot replace orange with an apple but fruit can be replaced with an orange or an apple because they are both Fruits, an apple is not an orange and an orange is not an apple.



**Note:** The point that you need to remember is that, as we have the Inheritance concept, that does not mean we can create the relationship between classes randomly. We will not get any error or exception, but the behavior that we expect might not get. So, always make sure that the relationship and functionality that we are providing make sense. If make sense, then go for the inheritance relationship, if not, then do not go for the inheritance relationship.

## How to Use the Liskov Substitution Principle in C#?

Using the Liskov Substitution Principle (LSP) in C# involves designing your classes and inheritance hierarchies to ensure derived classes can be substituted for their base classes without causing unexpected behavior. Here's a step-by-step guide on how to use the Liskov Substitution Principle effectively in C#:

- **Identify the Base Class or Interface:** Identify the base class or interface that defines the common contract or behavior that derived classes should adhere to.
- **Follow the Contract:** Ensure derived classes implement the same contract or behavior defined by the base class or interface. This includes implementing methods, properties, and behavior as specified.
- **Override Methods Carefully:** When overriding methods in derived classes, make sure that the overridden methods adhere to the expected behavior defined by the base class. Avoid changing the semantics of the method.
- **No Weakening Preconditions:** Ensure that preconditions (parameters, constraints, etc.) required by the base class methods are not weakened in derived class methods. Derived class methods should meet or strengthen the preconditions.
- **No Strengthening Postconditions:** Derived class methods should not strengthen the base class methods' postconditions (return values, effects, etc.). The behavior should be consistent or more relaxed, but not more strict.
- **Avoid Empty Overrides:** Avoid overriding methods in derived classes with empty or no-op implementations. This violates the LSP because the derived class is not substitutable for the base class.
- **Use Polymorphism:** Utilize polymorphism to treat objects of derived classes as objects of the base class. This allows you to interchangeably use different implementations without affecting correctness.
- **Test Substitutability:** Test the substitutability of derived classes by using them in contexts where base class instances are expected. Ensure that the behavior remains consistent and expected.
- **Document Contracts:** Document base classes and interfaces' contracts (interfaces, abstract methods, behavior). This helps developers understand the expected behavior when implementing derived classes.
- **Refactor for Consistency:** If you encounter violations of the LSP, refactor your code to ensure that the derived classes adhere to the contract and behavior defined by the base class.

## Advantages and Disadvantages of the Liskov Substitution Principle in C#:

The Liskov Substitution Principle (LSP) is one of the SOLID principles of object-oriented design. It emphasizes that objects of a derived class must be able to replace objects of the base class without affecting the correctness of the program. In simpler terms, derived classes should be substitutable for their base classes without causing unexpected behavior. Let's explore the advantages and disadvantages of following the Liskov Substitution Principle in C#:

### Advantages:

- **Behavioral Consistency:** Following the LSP ensures that derived classes behave consistently with their base classes. This allows developers to reason about the behavior of objects more reliably.
- **Code Reusability:** Well-designed subclasses can reuse the functionality of their base classes, leading to more efficient and less duplicated code.
- **Flexibility and Extensibility:** New subclasses can be introduced without affecting the base class code. This makes the system more adaptable to changes.
- **Interchangeability:** LSP-compliant objects can be interchanged in a program without altering its correctness. This allows for easy substitution and testing.

- **Design by Contract:** LSP encourages defining clear contracts (interfaces or base classes) that specify the expected behavior of subclasses. This promotes better communication between developers.
- **Support for Polymorphism:** LSP enables the effective use of polymorphism, allowing a single interface to be implemented by multiple classes with different behavior.

#### Disadvantages:

- **Violations Can Be Subtle:** Violations of the Liskov Substitution Principle can sometimes be subtle and difficult to detect. Unexpected behavior might occur at runtime when replacing base class objects with derived ones.
- **Complexity in Inheritance Hierarchies:** Inheritance hierarchies that become too deep or complex can make it challenging to maintain and ensure compliance with LSP.
- **Potential Overhead:** The need to adhere to the contract specified by the base class can sometimes lead to additional overhead when implementing derived classes.
- **Design Challenges:** Designing base classes and interfaces that allow for proper substitution can sometimes be challenging and require careful planning.
- **Semantic Issues:** If base classes do not have well-defined contracts or behaviors, achieving proper LSP compliance cannot be easy.
- **Trade-offs with Performance:** In some cases, adhering strictly to LSP might require additional runtime checks, impacting performance.

The Liskov Substitution Principle provides several advantages by ensuring behavioral consistency, code reusability, and flexibility in object-oriented design. However, it requires careful consideration of the relationships between the base and derived classes to avoid subtle runtime issues. Adhering to the LSP leads to more robust and maintainable code, which might involve some trade-offs and design challenges.

For Multiple Real-Time Examples of the Liskov Substitution Principle (LSP) using C#, please check the below link:

<https://dotnettutorials.net/lesson/real-time-examples-of-liskov-substitution-principle-in-csharp/>

# Interface Segregation Principle (ISP)

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

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

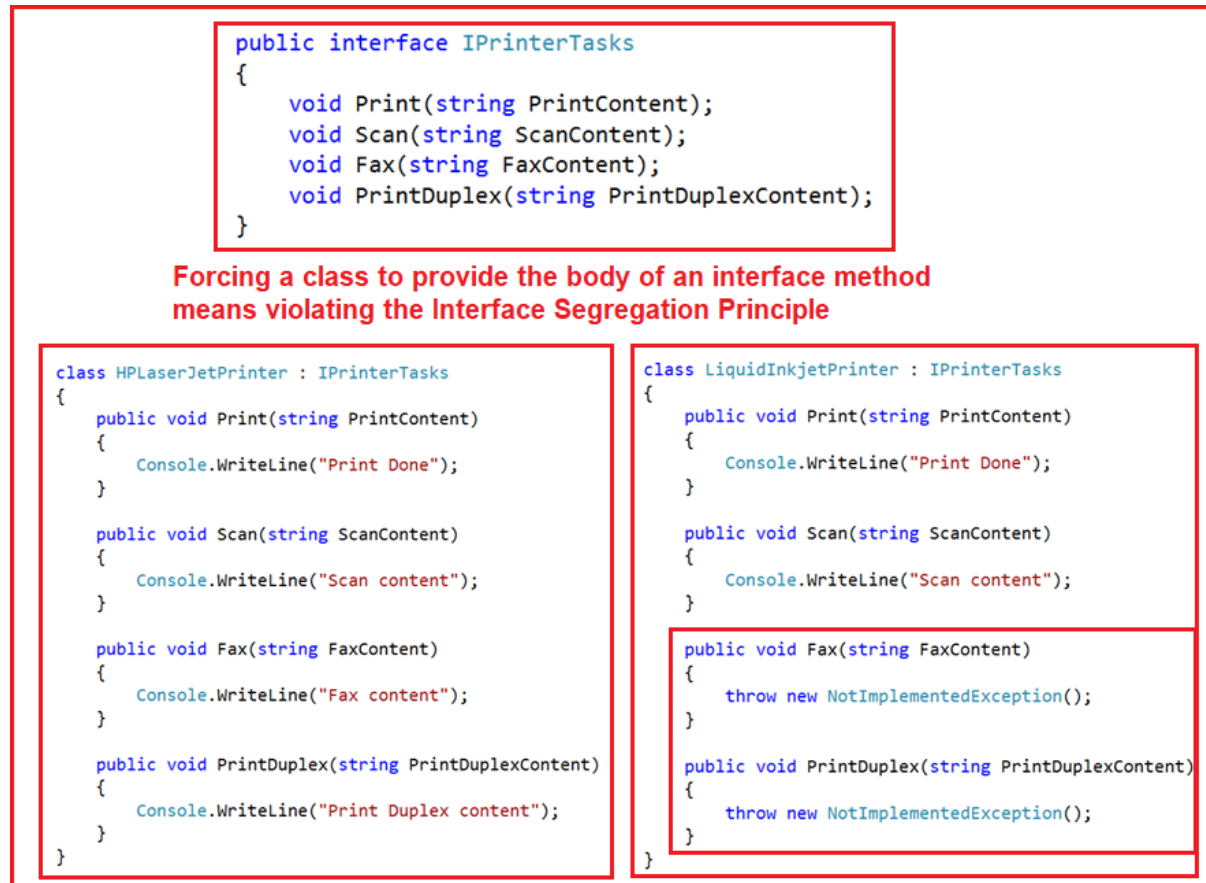
Let us break down the above definition into two parts.

1. First, no class should be forced to implement any method(s) of an interface they do not 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** of SOLID, like classes, interfaces also should have a single responsibility. That means we shouldn't force any class to implement any method(s) that they don't require.

Example to Understand Interface Segregation Principle in C#.

Let us understand Interface Segregation Principle in C# with an example. Please have a look at the following diagram. Here, you can see, we are having one interface and two classes implementing that Interface.



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 and must have to provide the implementation to all 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 PrintDuplex method which are not required by the class. This violates the Interface Segregation Principle in C# as are forcing the class to provide the implementation which they don't require.

### Example Without using the Interface Segregation Principle in C#:

Let us first see the example without following the Interface Segregation Principle and then we will see the problem. and finally, we will rewrite the same example by following Interface Segregation Principle using C# Language.

#### IPrinterTasks.cs

Create a class file with the name IPrinterTasks.cs and then copy and paste the following code into it. Here, IPrinterTasks is an interface and contains the declaration of four methods. By default, interface methods are public and abstract, so the child class of this interface needs to provide the implementation of all these methods.

```
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

Next, create a class file with the name HPLaserJetPrinter.cs and then copy and paste the following code into it. Here, the HPLaserJetPrinter implements the IPrinterTasks interface and provides implementations for all four methods. This class requires all printer services and hence, there is no issue in providing implementations for all interface methods.

```
using System;
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

Next, create a class file with the name LiquidInkjetPrinter.cs and then copy and paste the following code into it. Here, the LiquidInkjetPrinter implements the IPrinterTasks interface and provides

implementations for all four methods. But this class only requires the Print and Scan services. This class does not require Fax and PrintDuplex services, but, still, it is implementing these two methods. This is violating the Interface Segregation Principle as we are forcing the class to implement two methods that they do not require.

```
using System;
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();
        }
    }
}
```

### Why we are Facing this problem?

We are facing the above problem because we are declaring all the methods in a single class. And as per Inheritance Rules, the child class will take the responsibility to provide implementations for all interface methods. Because of this, LiquidInkjetPrinter class provides implementation to all IPrinterTasks Interface methods. We can overcome this problem by following the Interface Segregation Principle. Let us proceed and try to understand how we can rewrite the same example by following Interface Segregation Principle using C# Language.

### Example using Interface Segregation Principle in C#:

Please have a look at the following image. As you can see in the below image, now we have split that big interface into three small interfaces. Each interface now has some specific purpose.

```

public interface IPrinterTasks
{
    void Print(string PrintContent);
    void Scan(string ScanContent);
}
interface IFaxTasks
{
    void Fax(string content);
}
interface IPrintDuplexTasks
{
    void PrintDuplex(string content);
}

```

**Splitting a big interface into smaller ones**

Now if any class wants all the printer services then that class needs to implement all three interfaces. In our example, HPLaserJetPrinter wants all the printer services. So, the HPLaserJetPrinter class needs to implement all three interfaces and provide implementations of all interface methods as shown in the below image.

```

public class HPLaserJetPrinter : IPrinterTasks, IFaxTasks,
                                IPrintDuplexTasks
{
    public void Print(string PrintContent)...
    public void Scan(string ScanContent)...
    public void Fax(string FaxContent)...
    public void PrintDuplex(string PrintDuplexContent)...
}

```

**Class implementing all the three interfaces means this class wants all the four services**

Now, if any class wants the Scan and Print services, then that class needs to implement only the IPrinterTasks interfaces. In our example, LiquidInkjetPrinter wants only the Scan and Print services. So, the LiquidInkjetPrinter class needs to implement only the IPrinterTasks interfaces and needs to provide implementations for Print and Scan methods as shown in the below image.

```

class LiquidInkjetPrinter : IPrinterTasks
{
    public void Print(string PrintContent)...
    public void Scan(string ScanContent)...
}

```

**This class wants only Print and Scan service so implementing only the IPrinterTasks interface**

### The Complete Example Code is Given Below:

First, modify the IPrinterTasks.s class file as follows. Here, you can see, we are splitting the big interface into three interfaces. Now, each interface now has some specific purpose. And based on the requirement, the child class will implement 1, 2, or all the interfaces.

```
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);
    }
}
```

Next, modify the HPLaserJetPrinter.cs class file as follows. The HPLaserJetPrinter printer class requires all the printer services and hence implementing all three interfaces and providing implementation to all four methods.

```
using System;
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");
        }
    }
}
```

Next, modify the LiquidInkjetPrinter.cs class file as follows. The LiquidInkjetPrinter printer class requires only the Print and Scan Printer services and hence implements only the IPrinterTasks interface and provides implementation to Print and Scan methods.

```
using System;
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");
    }
}

```

Now, you can see the LiquidInkjetPrinter class is not providing implementation to the Fax and PrintDuplex method as these services are not required by the LiquidInkjetPrinter class. That means now our application design follows the Interface Segregation Principle. Now, you can test the functionality of the two classes by modifying the Program class code as follows.

```

using System;
namespace SOLID_PRINCIPLES.ISP
{
    public class Program
    {
        static void Main(string[] args)
        {
            //Using HPLaserJetPrinter we can access all Printer Services
            HPLaserJetPrinter hPLaserJetPrinter = new HPLaserJetPrinter();
            hPLaserJetPrinter.Print("Printing");
            hPLaserJetPrinter.Scan("Scanning");
            hPLaserJetPrinter.Fax("Faxing");
            hPLaserJetPrinter.PrintDuplex("PrintDuplex");

            //Using LiquidInkjetPrinter we can only Access Print and Scan Printer Services
            LiquidInkjetPrinter liquidInkjetPrinter = new LiquidInkjetPrinter();
            liquidInkjetPrinter.Print("Printing");
            liquidInkjetPrinter.Scan("Scanning");

            Console.ReadKey();
        }
    }
}

```

## How to Use Interface Segregation Principle in C#?

The Interface Segregation Principle (ISP) states that a class should not be forced to implement interfaces it does not use (a class should only implement interfaces it uses). In other words, interfaces should be designed to be specific to the needs of the implementing classes. Following this, the Interface Segregation Principle (ISP) avoids unnecessary coupling between classes and interfaces, resulting in more organized and easier-to-maintain code. This principle prevents unnecessary coupling between classes and interfaces, leading to more maintainable and cohesive code. Here's how to use the Interface Segregation Principle effectively in C#:

- **Identify Client Needs:** Determine the unique behaviors or methods each client class requires for an interface. It's best to avoid creating large, monolithic interfaces that multiple clients will use.
- **Design Fine-Grained Interfaces:** Create smaller, more focused interfaces with methods relevant to each client. This prevents classes from implementing methods they don't use.
- **Avoid Fat Interfaces:** Avoid combining many methods with different purposes when designing interfaces. Instead, break down the interface into smaller interfaces where each serves a specific purpose.
- **Create Specific Interfaces:** Design interfaces for implementing classes' context or domain. This ensures that classes are only required to implement the methods they want.
- **Prefer Composition:** Consider using composition to assemble smaller interfaces and components instead of creating a larger interface covering multiple aspects of a class's behavior.
- **Extract Common Behaviors:** If multiple classes have common methods, extract them into a separate interface to avoid code duplication and promote reuse.
- **Refactor Existing Interfaces:** If any existing interfaces violate the ISP, create smaller, more cohesive interfaces that align with the needs of implementing classes.
- **Use Default Implementations:** In C# 8 and later, default interface implementations can provide basic method implementations. This allows selective method overrides in classes.

- **Apply Dependency Injection:** Implement dependency injection by providing specific interfaces to a class for loose coupling and easy substitution of implementations.

## Advantages and Disadvantages of Interface Segregation Principle in C#

The Interface Segregation Principle (ISP) is one of the SOLID principles of object-oriented design, focusing on designing small, cohesive interfaces that cater to the specific needs of implementing classes. Let's explore the advantages and disadvantages of following the Interface Segregation Principle in C#:

### Advantages:

- **Reduced Coupling:** Interfaces designed with the ISP in mind are more focused and contain only the methods relevant to specific classes. This leads to lower coupling between classes and interfaces, promoting better separation of concerns.
- **Improved Maintainability:** Small, specialized interfaces are easier to understand, implement, and maintain. Changes to one interface are less likely to impact unrelated classes.
- **Flexibility in Implementations:** Implementing classes can choose which specific interfaces to implement based on their responsibilities. This allows for more flexible and modular code.
- **Enhanced Reusability:** Interfaces that align closely with the requirements of implementing classes lead to better code reuse. Implementing classes can easily fit into different contexts by selecting relevant interfaces.
- **Avoiding Fat Interfaces:** Following the ISP prevents the creation of "fat" interfaces with many methods that not all classes need. This keeps interfaces clean and focused.
- **Clearer Intent:** Specific interfaces convey the intent and responsibilities of implementing classes more clearly. This makes the codebase more self-documenting and easier to understand.

### Disadvantages:

- **Interface Proliferation:** Adhering to the ISP might lead to numerous small interfaces, making the codebase more complex to navigate and understand.
- **Design Effort:** Designing and maintaining a higher number of smaller interfaces might require additional effort compared to creating larger, more general interfaces.
- **Potential for Overhead:** In some cases, the overhead of implementing multiple small interfaces might be higher than implementing a larger one, especially if implementations share common methods.
- **Compatibility Issues:** If interfaces are broken down too much, providing default implementations for common methods might become difficult. This could lead to issues with backward compatibility.
- **Designing Contracts:** Creating precise and effective interfaces requires careful consideration of the needs of implementing classes, which might involve more upfront design effort.
- **Balancing Abstraction:** Striking the right balance between fine-grained interfaces and generalization can be challenging, as overly specific interfaces might not cater to all potential use cases.

For Multiple Real-Time Examples of the Interface Segregation Principle (ISP) using C#, please check the below link:

<https://dotnettutorials.net/lesson/real-time-examples-of-interface-segregation-principle-in-csharp/>

# Dependency Inversion Principle (DIP)

## What is the Dependency Inversion Principle?

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 is 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.

## Example to Understand Dependency Inversion Principle in C#

Let us understand Dependency Inversion Principle with one Example using C# Language. First, we will see the example without following the Dependency Inversion Principle and then we will identify the problems of not following the Dependency Inversion Principle, and then we will rewrite the same example using the Dependency Inversion Principle so that you will understand this concept easily. First create a Console Application and then add the following class files.

### 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. The following class is going to hold the employee data.

```
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; }
    }
}
```

### EmployeeDataAccessLogic.cs

Create a class file with the name EmployeeDataAccessLogic.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. In a real-time application, you need to write the logic to get the employee details from the database but for simplicity here we have hard-coded the employee details. Also, in a real-time application, you might have more methods like getting all employee information, creating, and updating employees, deleting employees, etc. Here, we have created only one method.

```
namespace SOLID_PRINCIPLES.DIP
{
    public class EmployeeDataAccessLogic
    {
        public Employee GetEmployeeDetails(int id)
        {
            //In real time get the employee details from db
            //but here we have hard coded the employee details
            Employee emp = new Employee()
            {
                ID = id,
                Name = "Pranaya",
                Department = "IT",
                Salary = 10000
            };
        }
    }
}
```

```

        return emp;
    }
}

```

### 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 EmployeeDataAccessLogic class. If you want to consume any method of the EmployeeDataAccessLogic class, then you need to create an instance of that class. In our example, the following class, GetEmployeeDataAccessObj() static method is going to return an instance of the EmployeeDataAccessLogic class, and using that instance we can access the GetEmployeeDetails(int id) method. So, this is the class that is going to return an instance of the EmployeeDataAccessLogic class using which we can do the database operations.

```

namespace SOLID_PRINCIPLES.DIP
{
    public class DataAccessFactory
    {
        public static EmployeeDataAccessLogic GetEmployeeDataAccessObj()
        {
            return new EmployeeDataAccessLogic();
        }
    }
}

```

### 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 EmployeeDataAccessLogic class. Here, within the constructor we call the static GetEmployeeDataAccessObj() method on the DataAccessFactory class which will return an instance of EmployeeDataAccessLogic and we initialize the \_EmployeeDataAccessLogic property with the return instance. We have also one method i.e. GetEmployeeDetails which is used to call the GetEmployeeDetails method on the EmployeeDataAccessLogic instance to get the employee detail by employee id.

```

namespace SOLID_PRINCIPLES.DIP
{
    public class EmployeeBusinessLogic
    {
        EmployeeDataAccessLogic _EmployeeDataAccessLogic;
        public EmployeeBusinessLogic()
        {
            _EmployeeDataAccessLogic = DataAccessFactory.GetEmployeeDataAccessObj();
        }
        public Employee GetEmployeeDetails(int id)
        {
            return _EmployeeDataAccessLogic.GetEmployeeDetails(id);
        }
    }
}

```

### Comparing the Above Example with 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, EmployeeBusinessLogic class depends on EmployeeDataAccessLogic class, so here EmployeeBusinessLogic class is the high-level module and EmployeeDataAccessLogic class is the low-level module.

So, as per first rule of the Dependency Inversion Principle, the EmployeeBusinessLogic class/module should not depend on the concrete EmployeeDataAccessLogic class/module, instead, both classes should depend on abstraction. But, in our example, the way we have implemented the code, the EmployeeBusinessLogic depending on the EmployeeDataAccessLogic class, means the first rule we are not following. Later part of this article, I will modify the example to follow Dependency Inversion Principle.

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 an **Abstract Class** which is Non-Concrete so that we cannot create an instance of it. In our example, the **EmployeeBusinessLogic** and **EmployeeDataAccessLogic** are concrete classes which means we can create objects of them. That means we are also not following the second rule of the Dependency Inversion Principle.

As per the Dependency Inversion Principle in C#, the EmployeeBusinessLogic (**High-Level Module**) should not depend on the concrete EmployeeDataAccessLogic (**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 Interface (or in Abstract Class)?

As you can see in the above example, EmployeeBusinessLogic uses the **GetEmployeeDetails()** method of the EmployeeDataAccessLogic class. In real-time, there will be many employee-related methods in the EmployeeDataAccessLogic class. So, we need to declare the GetEmployeeDetails(int id) method or any employee-related methods within the interface or abstract class. By default, interface methods are going to be abstract. But if you create an abstract class, then you need to declare the methods as abstract explicitly by using the abstract keyword. I am going with Interface as it makes the code more loosely coupled as well as we can also achieve multiple inheritances.

### IEmployeeDataAccessLogic.cs

Create a class file with the name IEmployeeDataAccessLogic.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. If you have multiple employee-related methods, then you need to use declare those methods here.

```
namespace SOLID_PRINCIPLES.DIP
{
    public interface IEmployeeDataAccessLogic
    {
        Employee GetEmployeeDetails(int id);
        //Any Other Employee Related Method Declarations
    }
}
```

Next, we need to implement the **IEmployeeDataAccessLogic** in **EmployeeDataAccessLogic** class. So, modify the **EmployeeDataAccessLogic** class as shown below. Here, you can see, the EmployeeDataAccessLogic class implementing the IEmployeeDataAccessLogic class and providing implementations for the GetEmployeeDetails method.

```
namespace SOLID_PRINCIPLES.DIP
{
    public class EmployeeDataAccessLogic : IEmployeeDataAccessLogic
    {
        public Employee GetEmployeeDetails(int id)
        {
            //In real time get the employee details from database
            //but here we have hard coded the employee details
            Employee emp = new Employee()
            {
                ID = id,
            }
        }
    }
}
```

```

        Name = "Pranaya",
        Department = "IT",
        Salary = 10000
    };
    return emp;
}
}
}

```

Next, we need to change the DataAccessFactory class. Here, we need to change the return type of the GetEmployeeDataAccessObj to IEmployeeDataAccessLogic instead of EmployeeDataAccessLogic. Internally, the method creates an instance of the EmployeeDataAccessLogic class but we return that instance to the user using the Parent Interface i.e. IEmployeeDataAccessLogic. This is possible because a Parent Class Reference Variable can hold the child class object reference. And here, IEmployeeDataAccessLogic is the Parent class and EmployeeDataAccessLogic is the Child class of the IEmployeeDataAccessLogic Parent class.

```

namespace SOLID_PRINCIPLES.DIP
{
    public class DataAccessFactory
    {
        public static IEmployeeDataAccessLogic GetEmployeeDataAccessObj()
        {
            return new EmployeeDataAccessLogic();
        }
    }
}

```

Now, we need to change EmployeeBusinessLogic class which will use IEmployeeDataAccessLogic instead of the concrete EmployeeDataAccessLogic class as shown below. Now, you can see, the EmployeeBusinessLogic class is not using the concrete EmployeeDataAccessLogic class instead it is using the not concrete IEmployeeDataAccessLogic class.

```

namespace SOLID_PRINCIPLES.DIP
{
    public class EmployeeBusinessLogic
    {
        IEmployeeDataAccessLogic _IEmployeeDataAccessLogic;
        public EmployeeBusinessLogic()
        {
            _IEmployeeDataAccessLogic = DataAccessFactory.GetEmployeeDataAccessObj();
        }
        public Employee GetEmployeeDetails(int id)
        {
            return _IEmployeeDataAccessLogic.GetEmployeeDetails(id);
        }
    }
}

```

We have implemented the Dependency Inversion Principle in our example using C# language where the High-Level module (EmployeeBusinessLogic) and Low-Level module (EmployeeDataAccessLogic) depend on abstraction (IEmployeeDataAccessLogic). Also, abstraction (IEmployeeDataAccessLogic) does not depend on details (EmployeeDataAccessLogic) but details depend on abstraction.

Now, you can test whether the application code is working as expected or not by modifying the Main method of the Program class as follows. Here, we are simply creating an instance of the EmployeeBusinessLogic class and calling the GetEmployeeDetails method and then printing the employee details on the Console window.

```

using System;
namespace SOLID_PRINCIPLES.DIP
{
    public class Program

```



```

{
    static void Main(string[] args)
    {
        EmployeeBusinessLogic employeeBusinessLogic = new EmployeeBusinessLogic();
        Employee emp = employeeBusinessLogic.GetEmployeeDetails(1001);
        Console.WriteLine($"ID: {emp.ID}, Name: {emp.Name}, Department: {emp.Department},
Salary: {emp.Salary}");
        Console.ReadKey();
    }
}
}

```

## How to use the Dependency Inversion Principle in C#?

Using the Dependency Inversion Principle (DIP) in C# involves designing classes and modules to promote loose coupling and dependency inversion. Here's a step-by-step guide on how to use the Dependency Inversion Principle effectively in C#:

- **Identify High-Level and Low-Level Modules:** Identify your application's high-level and low-level modules, with the high-level modules containing business logic and the low-level modules containing implementation details.
- **Define Abstractions:** Create interfaces or abstract classes to define the behavior or contract that high-level modules need.
- **Implement Abstractions:** Implement the interfaces or abstract classes in the low-level modules to fulfill the contracts defined by the abstractions with concrete details.
- **Invert Dependencies:** Make high-level modules depend on abstractions (interfaces or abstract classes) instead of low-level module implementations.
- **Use Dependency Injection:** Implement dependency injection to provide instances of implementations to high-level modules. We can control the high-level module behavior without modifying their code.
- **IoC Containers (Optional):** Consider using an IoC container for dependency injection and object lifetime management. IoC containers simplify the process of injecting dependencies and managing object lifetimes.
- **Configure Dependencies:** Configure your application to inject the appropriate implementations into high-level modules through constructors, properties, or methods.
- **Isolate Unit Testing:** When conducting unit testing, abstract implementations are replaced with mock implementations to isolate high-level modules from the actual implementations.

## Advantages and Disadvantages of Dependency Inversion Principle:

Implementing the Dependency Inversion Principle (DIP) in C# has advantages and disadvantages. Dependency Inversion Principle (DIP) is a SOLID principle of object-oriented design that emphasizes the significance of having abstractions that both high-level and low-level modules depend on, rather than high-level modules depending on low-level modules. Here are the advantages and disadvantages of following the Dependency Inversion Principle (DIP) in C#:

### Advantages:

- **Loose Coupling:** When classes depend on abstractions (interfaces or abstract classes) instead of concrete implementations, replacing implementations without affecting higher-level modules becomes easier.
- **Flexibility and Extensibility:** The Dependency Inversion Principle (DIP) enables the introduction of new implementations without modifying high-level modules, making the system more adaptable to changing requirements.
- **Testability:** When high-level modules depend on abstractions, we can easily substitute mock implementations for abstractions during unit testing. This enables isolated testing and better test coverage.
- **Parallel Development:** This allows teams to work independently on modules. Changes to one module are less likely to impact other modules.
- **Isolation of Concerns:** The DIP (Dependency Inversion Principle) promotes a clear separation of concerns between high-level and low-level modules. High-level modules focus on the application's business logic, while low-level modules handle implementation details.



- **Plug and Play Architecture:** The DIP (Dependency Inversion Principle) enables easy integration of new components and implementations in a plug-and-play architecture.

**Disadvantages:**

- **Design Overhead:** Following the Dependency Inversion Principle (DIP) may require more abstraction layers and interfaces, resulting in increased upfront design work.
- **Learning Curve:** Effective implementation of interfaces and abstractions requires additional learning and understanding of design patterns for developers.
- **Complexity in Simple Cases:** In some cases, strictly to the Dependency Inversion Principle (DIP) in basic applications or modules may lead to unnecessary complications.
- **Performance Overhead:** Using abstractions could introduce unnecessary performance overhead due to indirection and runtime binding.
- **Over-Abstracting:** Excessive abstraction can lead to complex code (excessive interfaces and abstractions) that isn't easy to understand.

The benefits of applying the Dependency Inversion Principle (DIP) are numerous, such as loose coupling, flexibility, and testability. However, it may increase the complexity of the design and require a good understanding of the design pattern to balance abstraction and implementation.

For Multiple Real-Time Examples of the Dependency Inversion Principle (DIP) using C#, please check the below link:

<https://dotnettutorials.net/lesson/real-time-examples-of-dependency-inversion-principle-in-csharp/>

My LinkedIn Profile: <https://www.linkedin.com/in/pranaya-rout/>

My YouTube Profile: <https://www.youtube.com/c/DotNetTutorials>

My Facebook Group: <https://www.facebook.com/tutorialsdotnet/>

My Twitter Profile: <https://twitter.com/RoutPranaya>

**Contact us For Online Dot Net Core Training:**

WhatsApp Number: 91 7021801173

Mobile Number: 91 7021801173

Email: [pranaya@dotnettutorials.net](mailto:pranaya@dotnettutorials.net)

Join our Official Telegram Group: <https://telegram.me/dotnettutorials>

**Join Our Advanced C# Online Training:**

Course Syllabus: <https://dotnettutorials.net/lesson/csharp-dot-net-online-training/>

C# Training Telegram Group: <https://telegram.me/+dghg6SdNRfkWNjll>

**Join Our ASP.NET Core Online Training:**

Course Syllabus: <https://dotnettutorials.net/lesson/asp-net-core-online-training/>

ASP.NET Core Training Telegram Group: <https://telegram.me/aspnetcoretraining>