


Single Responsibility Principle (SRP)

```
class Employee
{
    public string Name { get; set; }
    public string Email { get; set; }
    public decimal Salary { get; set; }

    public void CalculateSalary()
    {
        // logic to calculate salary
    }

    public void SendEmail()
    {
        // logic to send email
    }

    public void SaveToDatabase()
    {
        // logic to save employee data to database
    }
}
```



This class violates the SRP because it has more than one reason to change. It has three responsibilities: calculating salary, sending email, and saving data to database. These responsibilities are not related to each other and belong to different layers of the application.

A better way to design this class is to separate these responsibilities into different classes. For example, you can create a SalaryCalculator class that handles the salary calculation logic, an EmailService class that handles the email sending logic, and an EmployeeRepository class that handles the data access logic.

```
class Employee
{
    private readonly SalaryCalculator _salaryCalculator;
    private readonly EmailService _emailService;
    private readonly EmployeeRepository _employeeRepository;


    public Employee(SalaryCalculator salaryCalculator,
        EmailService emailService,
        EmployeeRepository employeeRepository)
    {
        _salaryCalculator = salaryCalculator;
        _emailService = emailService;
        _employeeRepository = employeeRepository;
    }

    public string Name { get; set; }
    public string Email { get; set; }
    public decimal Salary { get; set; }

    public void CalculateSalary()
    {
        Salary = _salaryCalculator.CalculateSalary(this);
    }

    public void SendEmail()
    {
        _emailService.SendEmail(this);
    }

    public void SaveToDatabase()
    {
        _employeeRepository.Save(this);
    }
}
```



Open-Closed Principle (OCP)

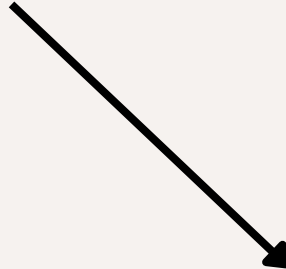
```
class Calculator
{
    public double Calculate(double x, double y, char op)
    {
        switch (op)
        {
            case '+':
                return x + y;
            case '-':
                return x - y;
            case '*':
                return x * y;
            case '/':
                return x / y;
            default:
                throw new ArgumentException("Invalid operator");
        }
    }
}
```



This class works fine for basic arithmetic operations. But what if you want to add more operations, such as power, square root, or modulus?

You have to modify the Calculate method and add more cases to the switch statement. This violates the OCP and makes the code more complex and prone to errors.

Then, you can implement this interface in different classes that represent different operations. For example, you can create classes such as Addition, Subtraction, Multiplication, Division, Power, SquareRoot, and Modulus.



```
interface IOperation
{
    double Perform(double x, double y);
}
```

```
class Addition : IOperation
{
    public double Perform(double x, double y)
    {
        return x + y;
    }
}
```

```
class Subtraction : IOperation
{
    public double Perform(double x, double y)
    {
        return x - y;
    }
}
```

```
class Multiplication : IOperation
{
    public double Perform(double x, double y)
    {
        return x * y;
    }
}
```

```
class Division : IOperation
{
    public double Perform(double x, double y)
    {
        return x / y;
    }
}
```

```
class Power : IOperation
{
    public double Perform(double x, double y)
    {
        return Math.Pow(x, y);
    }
}
```



A better way to design this class is to use an abstract class or an interface for the operator parameter. For example, you can create an interface called IOperation that has a method called Perform that takes two numbers and returns the result of the operation.

```
class Calculator
{
    private readonly IOperation _operation;

    public Calculator(IOperation operation)
    {
        _operation = operation;
    }

    public double Calculate(double x, double y)
    {
        return _operation.Perform(x, y);
    }
}
```

Now, you can inject this interface as a dependency into the Calculator class using constructor injection. This way, the Calculator class only depends on an abstraction, not a concretion.

```
class Factorial : IOperation
{
    public double Perform(double x, double y)
    {
        // logic to calculate factorial of x
    }
}
```

```
Calculator calculator = new Calculator(new Factorial());
double result = calculator.Calculate(5); // This will calculate
```

This design follows the OCP and makes the code more open for extension and closed for modification. It also makes it easier to add new operations without changing the existing code. For example, you can create a new class that implements IOperation and pass it to the Calculator constructor.

Liskov Substitution Principle (LSP)

```
abstract class Shape
{
    2 references
    public abstract double Area();
}


1 reference
class Rectangle : Shape
{
    2 references
    public double Width { get; set; }
    2 references
    public double Height { get; set; }

    1 reference
    public override double Area()
    {
        return Width * Height;
    }
}

0 references
class Square : Shape
{
    2 references
    public double Width { get; set; }
    0 references
    public double Height { get; set; }

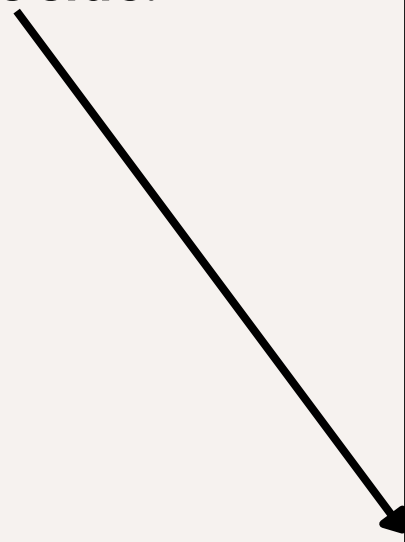
    1 reference
    public override double Area()
    {
        return Width * Width;
    }
}

0 references
void SetRectangleDimensions(Rectangle rectangle,
    double width, double height)
{
    rectangle.Width = width;
    rectangle.Height = height;
}
```



A better way to design this example is to use an interface instead of an abstract class for the base type. For example, you can create an interface called IShape that has a method called Area.

Then, you can implement this interface in both Rectangle and Square classes. The Rectangle class remains the same as before, but the Square class only has one property: Side, which represents the length of its side.



```
interface IShape
{
    3 references
    double Area();
}


0 references
class Rectangle : IShape
{
    1 reference
    public double Width { get; set; }
    1 reference
    public double Height { get; set; }

    2 references
    public double Area()
    {
        return Width * Height;
    }
}

0 references
class Square : IShape
{
    2 references
    public double Side { get; set; }

    2 references
    public double Area()
    {
        return Side * Side;
    }
}

0 references
void PrintArea(IShape shape)
{
    Console.WriteLine(shape.Area());
}
```



This method works fine when you pass a Rectangle object to it. But what if you pass a Square object to it? Since Square is a subclass of Rectangle, you can do that. But this will break the functionality of the Square class, because it will change its width and height to different values, which violates the definition of a square.

Interface Segregation Principle (ISP)

```
2 references
interface IPrinter
{
    2 references
    void Print();
    2 references
    void Scan();
    2 references
    void Fax();
    2 references
    void Copy();
}

0 references
class LaserPrinter : IPrinter
{
    1 reference
    public void Print()
    {
        // logic to print
    }

    1 reference
    public void Scan()
    {
        // logic to scan
    }

    1 reference
    public void Fax()
    {
        // logic to fax
    }

    1 reference
    public void Copy()
    {
        // logic to copy
    }
}

0 references
class InkjetPrinter : IPrinter
{
    1 reference
    public void Print()
    {
        // logic to print
    }

    1 reference
    public void Scan()
    {
        // logic to scan
    }

    1 reference
    public void Fax()
    {
        throw new NotImplementedException();
    }

    1 reference
    public void Copy()
    {
        throw new NotImplementedException();
    }
}
```



The InkjetPrinter class, however, can only perform two operations: print and scan. It cannot fax or copy. So, it implements only two methods and throws an exception for the other two.

This design violates the ISP because the InkjetPrinter class depends on methods it does not use. It also violates the LSP because the InkjetPrinter class is not substitutable for its base interface IPrinter. It can cause errors and confusion when using polymorphism.

```
2 references
interface IPrint
{
    2 references
    void Print();
}

2 references
interface IScan
{
    2 references
    void Scan();
}

1 reference
interface IFax
{
    1 reference
    void Fax();
}

1 reference
interface ICopy
{
    1 reference
    void Copy();
}

1 reference
class LaserPrinter : IPrint, IScan, IFax, ICopy
{
    1 reference
    public void Print() { }

    1 reference
    public void Scan() { }

    1 reference
    public void Fax() { }

    1 reference
    public void Copy() { }
}

0 references
class InkjetPrinter : IPrint, IScan
{
    1 reference
    public void Print() { }

    1 reference
    public void Scan() { }
}
```



A better way to design this example is to split the IPrinter interface into smaller interfaces that have only one responsibility. For example, you can create four interfaces: IPrint, IScan, IFax, and ICopy.

Then, you can implement only the interfaces that are relevant for each class. For example, the LaserPrinter class can implement all four interfaces, while the InkjetPrinter class can implement only two interfaces.

Dependency Inversion Principle (DIP)

```
class OrderService
{
    private readonly EmailSender _emailSender;
    private readonly DatabaseLogger _databaseLogger;

    public OrderService()
    {
        _emailSender = new EmailSender();
        _databaseLogger = new DatabaseLogger();
    }

    public void PlaceOrder(Order order)
    {
        // logic to place order
        _emailSender.SendEmail(order.Customer.Email, "Your order has been placed.");
        _databaseLogger.Log("Order placed: " + order.Id);
    }
}
```



This design violates the DIP because the OrderService class depends on concrete implementations of EmailSender and DatabaseLogger.

This makes the code more tightly coupled and less flexible. For example,

if you want to change the email sender or the logger to use different implementations, you have to modify the OrderService class. Also, if you want to test the OrderService class in isolation, you have to mock or stub the EmailSender and DatabaseLogger classes.

```
class EmailSender : IEmailSender
{
    public void SendEmail(string email, string message)
    {
        // logic to send email
    }
}

class DatabaseLogger : ILogger
{
    public void Log(string message)
    {
        // logic to log data to database
    }
}
```

```
interface IEmailSender
{
    void SendEmail(string email, string message);
}


interface ILogger
{
    void Log(string message);
}
```

A better way to design this example is to use interfaces or abstract classes for the dependencies. For example, you can create an interface called IEmailSender that has a method called SendEmail, and an interface called ILogger that has a method called Log.

```
class OrderService
{
    private readonly IEmailSender _emailSender;
    private readonly ILogger _logger;

    public OrderService(IEmailSender emailSender,
                        ILogger logger)
    {
        _emailSender = emailSender;
        _logger = logger;
    }

    public void PlaceOrder(Order order)
    {
        // logic to place order
        _emailSender.SendEmail(order.Customer.Email, "Your order has been placed.");
        _logger.Log("Order placed: " + order.Id);
    }
}
```



Now, you can inject these interfaces as dependencies into the OrderService class using constructor injection. This way, the OrderService class only depends on abstractions, not concretions.

FOLLOW FOR MORE CONTENT



- ✓ **5 YEARS OF EXPERIENCE**
- ✓ **C#, ASP.NET CORE, MVC, APIS, BLAZOR, SQL**
- ✓ **BLUE CHIP + SMALL CLIENTS**
- ✓ **TOP RATED ON UPWORK & FIVERR**
- ✓ **CHOOSE ME FOR YOUR PROJECT AND LET'S WORK TOGETHER TO CREATE SOMETHING GREAT!**



YASH DUDHAGARA