SINGLE RESPONSIBILITY PRINCIPLE

Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class, module or function.

A class should have one, and only one, reason to change.

Robert C.Martin (Uncle Bob)

Bad code

```
public class Assistant
{
    public int RetirementAge = 65;
    public int Age { get; set; }

    public Assistant(int age)
    {
        this.Age = age;
    }

    public void HandleEmployee()
    {
        Console.WriteLine("Logging data...");
        Console.WriteLine(this.RetirementAge - this.Age);
    }
}
```

Refactored (would be great idea to use solution from DIP)

```
public class Assistant
{
    public FinancesAssistant FinancesAssistant { get; set; }
    public Logger Logger { get; set; }

    public Assistant(int age)
    {
        this.FinancesAssistant = new FinancesAssistant(age);
        this.Logger = new Logger();
    }

    public void HandleEmployee()
    {
        this.Logger.Log();
        this.FinancesAssistant.Calculate();
    }
}
```

OPEN CLOSE PRINCIPLE

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Meyer Bertrand

Bad code

```
class Finances
{
    public void CalculateSalaries(dynamic[] employees)
    {
        foreach (var employee in employees)
        {
            if (employee is Ceo)
            {
                var ceo = (Ceo)employee;
                ceo.CalculateSalary();
            }
            if (employee is Programmer)
            {
                var programmer = (Programmer)employee;
                programmer.CalculateSalary();
            }

        }
    }
}
```

Refactored

```
public class Ceo : IEmployee
{
    public void CalculateSalary()
    {
        Console.WriteLine("Pay Ceo");
    }

    public void ShowIdCard()
    {
        Console.WriteLine("Greet Ceo");
    }
}
```

```
public class Finances
{
    public void CalculateSalaries(List<IEmployee> employees)
    {
        foreach (var employee in employees)
        {
            employee.CalculateSalary();
        }
    }
}
```

LISKOV SUBSTITUTION PRINCIPLE

If class S (subtype) is a child of class T (type), then objects of type T may be replaced with objects of type S, without breaking the program.

Barbara Liskov

Bad code

```
public class Ceo : Employee
{
    public override void CalculateSalary()
    {
        Console.WriteLine("Calculate for Ceo");
    }

    public override void ShowIdCard()
    {
        Console.WriteLine("Show card Ceo");
    }
}
```

```
public class Employee
{
    public virtual void CalculateSalary()
    {
        Console.WriteLine("Calculate for Employee");
    }

    public virtual void ShowIdCard()
    {
        Console.WriteLine("Show card Employee");
    }
}
```

```
public class Finances
{
    public void PaySalaries(List<Employee> employees)
    {
        foreach (var employee in employees)
        {
            employee.CalculateSalary();
        }
    }
}
```

Refactored

```
public class Ceo : IVisitor, IPayableEmployee
{
    public void CalculateSalary()
    {
        Console.WriteLine("Calculate for Ceo");
    }

    public void ShowIdCard()
    {
        Console.WriteLine("Show card Ceo");
    }
}
```

```
public interface IPayableEmployee
{
    void CalculateSalary();
}

public interface IVisitor
{
    void ShowIdCard();
}
```

```
public class Finances
{
    public void PaySalaries(List<IPayableEmployee> employees)
    {
        foreach (var employee in employees)
        {
            employee.CalculateSalary();
        }
    }
}
```

INTERFACE SEGREGATION PRINCIPLE

Clients should not be forced to depend upon interfaces that they do not use.

Robert C. Martin

Bad code

```
class Computer : IProduct
{
    public int Price { get; set; }
    public string OS { get; set; }
    public int Weight { get; set; }
}
```

```
public interface IProduct
{
    int Price { get; set; }
    int Weight { get; set; }
    string OS { get; set; }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        var computer = new Computer() { Price = 100, OS = "Windows" };
        var server = new Server() { Price = 500, OS = "Linux" };
    }
}
```

Refactored

```
class Computer : IProduct, IDeliverable, IComputer
{
    public int Price { get; set; }
    public string OS { get; set; }
    public int Weight { get; set; }
}

class Training : IProduct
{
    public int Price { get; set; }
}
```

```
interface IComputer
{
    string OS { get; set; }
}

interface IDeliverable
{
    int Weight { get; set; }
}

interface IProduct
{
    int Price { get; set; }
}
```

DEPENDENCY INVERSION PRINCIPLE

1. High-level modules should not depend on low-level modules.
   Both should depend on abstractions (e.g. interfaces).
2. Abstractions should not depend on details.
   Details (concrete implementations) should depend on abstractions.

Robert C. Martin

Bad code

```
class Program
{
    static void Main(string[] args)
    {
        var reporter = new Reporter();
        reporter.CreateReport();
    }
}
```

```
namespace V1
{
    class Reporter
    {
        private CsvParser csvParser;

        public Reporter()
        {
            this.csvParser = new CsvParser();
        }

        public void CreateReport()
        {
            this.csvParser.Read();
        }
    }

    public class CsvParser
    {
        public void Read()
        {
            Console.WriteLine("Reading csv...");
        }
    }
```

Refactored

```
class Program
{
    static void Main(string[] args)
    {
        var csvParser = new CsvParser();
        var reporter = new Reporter(csvParser);
        reporter.CreateReport();

        Console.Read();
    }
}

class Reporter
{
    private IDataProvider dataReader;

    public Reporter(IDataProvider dataReader)
    {
        this.dataReader = dataReader;
    }

    public void CreateReport()
    {
        this.dataReader.Read();
    }
}
```

```
    public interface IDataProvider
    {
        void Read();
    }

    public class CsvParser : IDataProvider
    {
        public void Read()
        {
            Console.WriteLine("Reading csv...");
        }
    }
}
```