Introduction to SOLID

- Origin: Introduced by Robert C. Martin (Uncle Bob)
- Purpose: Improve software design, maintainability, and flexibility
- Acronym:
  - S – Single Responsibility Principle
  - O – Open/Closed Principle
  - L – Liskov Substitution Principle
  - I – Interface Segregation Principle
  - D – Dependency Inversion Principle

What is SRP?

Single Responsibility Principle (SRP)
– The “S” in SOLID

> “ **A class should have only one reason to change.** ”

In other words:
A class should **only do one thing** and **do it well**.

Bad Example — Violating SRP

```
public class Report
{
    public string Title { get; set; }
    public string Content { get; set; }

    public void SaveToFile(string path)
    {
        File.WriteAllText(path, Title + "\n" + Content);
    }

    public void Print()
    {
        Console.WriteLine(Title);
        Console.WriteLine(Content);
    }
}
```

What's wrong here?

This class has **multiple responsibilities**:

1. **Holds report data**
2. **Handles file saving**
3. **Handles printing**

➡️ If printing logic changes, this class must change.
➡️ If file I/O changes, this class must also change.

⚠️ **Violates SRP**

Let's break this into separate classes:

```
public class Report
{
    public string Title { get; set; }
    public string Content { get; set; }
}

public class ReportPrinter
{
    public void Print(Report report)
    {
        Console.WriteLine(report.Title);
        Console.WriteLine(report.Content);
    }
}
```

```csharp
public class ReportSaver
{
    public void SaveToFile(Report report, string path)
    {
        File.WriteAllText(path, report.Title + "\n" + report.Content);
    }
}
```

Now:

- `Report` → stores data
- `ReportPrinter` → prints the report
- `ReportSaver` → saves to a file

Each class has **only one reason to change**.

Summary

✅ **SRP helps you**:

- Keep code clean and modular
- Improve readability and maintainability
- Reduce risk of bugs when requirements change

Always ask:
"Does this class do more than one thing?"
If yes → time to refactor!

What is OCP?

Open/Closed Principle (OCP)
– The "O" in SOLID

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

Meaning:
You should be able to add new behavior **without modifying** existing code.

Bad Example — Violating OCP

```
public class DiscountCalculator
{
    public double CalculateDiscount(string customerType, double total)
    {
        if (customerType == "Regular")
            return total * 0.9;
        else if (customerType == "Premium")
            return total * 0.8;
        else if (customerType == "VIP")
            return total * 0.7;
        else
            return total;
    }
}
```

What's wrong?

- Adding a **new customer type** (e.g., "Gold") means editing the method.
- Every change violates **OCP**.
- High **risk of bugs** and code duplication over time.

We can apply **polymorphism** to extend behavior.

```
public interface IDiscountStrategy
{
    double ApplyDiscount(double total);
}

public class RegularDiscount : IDiscountStrategy
{
    public double ApplyDiscount(double total) => total * 0.9;
}

public class PremiumDiscount : IDiscountStrategy
{
    public double ApplyDiscount(double total) => total * 0.8;
}

public class VipDiscount : IDiscountStrategy
{
    public double ApplyDiscount(double total) => total * 0.7;
}
```

Now the calculator uses dependency injection:

```
public class DiscountCalculator
{
    private readonly IDiscountStrategy _strategy;

    public DiscountCalculator(IDiscountStrategy strategy)
    {
        _strategy = strategy;
    }

    public double CalculateDiscount(double total)
    {
        return _strategy.ApplyDiscount(total);
    }
}
```

➡️ To add a new strategy (e.g., `GoldDiscount`), you just implement the interface.
No changes required in existing classes.

Summary

✅ **Open/Closed Principle** promotes:

- **Extensibility**: Add features without touching stable code
- **Maintainability**: Fewer bugs when requirements change
- **Testability**: Smaller, focused classes

Ask yourself:
"Am I modifying existing code every time I need to support a new case?"
If yes → consider using OCP and abstraction.

What is Liskov Substitution Principle (LSP)?

Definition:

> “ *If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering the correctness of the program.* ”

— Barbara Liskov, 1987

In simple terms:
A subclass should behave in such a way that any code using the base class still works if we substitute it with the subclass.

Bad Example — Violating LSP

```
public class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }

    public int GetArea() => Width * Height;
}

public class Square : Rectangle
{
    public override int Width
    {
        set { base.Width = base.Height = value; }
    }

    public override int Height
    {
        set { base.Width = base.Height = value; }
    }
}
```

Problem:

```
public void PrintArea(Rectangle r)
{
    r.Width = 5;
    r.Height = 10;
    Console.WriteLine(r.GetArea()); // Expected: 50
}
```

But with `Square`, it prints `100` instead!
Why? Because setting width also changes height and vice versa.

⚠️ **This breaks the Liskov Substitution Principle.**

Let’s separate the shapes with an interface:

```
public interface IShape
{
    int GetArea();
}

public class Rectangle : IShape
{
    public int Width { get; set; }
    public int Height { get; set; }

    public int GetArea() => Width * Height;
}

public class Square : IShape
{
    public int Side { get; set; }

    public int GetArea() => Side * Side;
}
```

Now:

```
public void PrintArea(IShape shape)
{
    Console.WriteLine(shape.GetArea());
}
```

✅ **Substitution is safe and behavior is consistent.**

Key Takeaways

- Subclasses **must not** break the behavior expected from the base class.
- Respect **contracts**, **preconditions**, and **postconditions**.
- When in doubt, **rethink your inheritance hierarchy**.

LSP ensures maintainable, reliable, and extendable code.

What is ISP?

Interface Segregation Principle (ISP)
– The “I” in SOLID

> “ **Clients should not be forced to depend on interfaces they do not use.** ”

In simple terms:
It’s better to have many **small, specific interfaces** than a large, bloated one.

Bad Example — Violating ISP

```
public interface IWorker
{
    void Work();
    void Eat();
}
```

Now imagine we have two classes:

```
public class HumanWorker : IWorker
{
    public void Work() => Console.WriteLine("Working...");
    public void Eat() => Console.WriteLine("Eating lunch...");
}

public class RobotWorker : IWorker
{
    public void Work() => Console.WriteLine("Working...");

    public void Eat()
    {
        // Not applicable for robots!
        throw new NotImplementedException();
    }
}
```

What's wrong?

- `RobotWorker` is forced to implement a method (`Eat`) it **does not need**.
- Violates ISP: the interface is **too general**.
- Leads to **fragile and confusing** code.

Split the interface into more specific ones:

```
public interface IWorkable
{
    void Work();
}

public interface IEatable
{
    void Eat();
}
```

Now use only what's needed:

```
public class HumanWorker : IWorkable, IEatable
{
    public void Work() => Console.WriteLine("Working...");
    public void Eat() => Console.WriteLine("Eating lunch...");
}

public class RobotWorker : IWorkable
{
    public void Work() => Console.WriteLine("Working...");
}
```

✅ Now:

- `RobotWorker` only depends on what it needs.
- Interfaces are **small and focused**.
- Easy to **extend**, test, and maintain.

✅ Interface Segregation Principle helps you:

- Avoid bloated interfaces
- Prevent dummy or unimplemented methods
- Keep classes clean and focused
- Write code that’s easier to refactor

Ask yourself:
"Is this interface forcing classes to implement things they don’t need?"

If yes → split it!

Dependency Inversion Principle (DIP)?

> **High-level modules should not depend on low-level modules. Both should depend on abstractions. > Abstractions should not depend on details. Details should depend on abstractions.**

In simpler terms:

- Code should depend on **interfaces**, not **concrete implementations**.
- This makes your code more flexible, testable, and maintainable.

Bad Example — Violating DIP

```
public class FileLogger
{
    public void Log(string message)
    {
        File.AppendAllText("log.txt", message);
    }
}

public class UserService
{
    private FileLogger _logger = new FileLogger();

    public void RegisterUser(string username)
    {
        // logic to register user
        _logger.Log("User registered: " + username);
    }
}
```

What's wrong?

- `UserService` (a high-level class) **directly depends** on `FileLogger` (a low-level class).
- Tight coupling: hard to replace logger, or test `UserService`.
- Violates DIP.

✅ Good Example — DIP Respected

Step 1: Introduce an abstraction

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

Step 2: Implement concrete loggers

```
public class FileLogger : ILogger
{
    public void Log(string message)
    {
        File.AppendAllText("log.txt", message);
    }
}

public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

Step 3: Use abstraction in the high-level class

```
public class UserService
{
    private readonly ILogger _logger;

    public UserService(ILogger logger)
    {
        _logger = logger;
    }

    public void RegisterUser(string username)
    {
        // logic to register user
        _logger.Log("User registered: " + username);
    }
}
```

Now:

- `UserService` depends on the **interface**, not the concrete logger.
- You can **swap in any logger** at runtime or in tests.
- ✅ DIP is respected.

Summary

✅ **Dependency Inversion Principle brings:**

- **Loose coupling** between modules
- Easier **testing and mocking**
- Better **scalability and flexibility**

Ask yourself:
"Am I depending on abstractions, or on concrete classes?"

If it’s the latter → introduce interfaces!