

SOLID Principles

1. Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change, meaning it should have only one job or responsibility

Violation: SRPViolation.cs

```
public class Invoice
{
    public int Id { get; set; }

    public double Amount { get; set; }

    public void Print() => Console.WriteLine($"Printing Invoice {Id}");

    public void Save() => Console.WriteLine($"Saving Invoice {Id}");
}
```

WHats Wrong?

- Class handles 3 responsibilities: data, printing, and saving.
- If any responsibility changes, the class must change → **bad cohesion**.

Refactored: SRPRefactored.cs

```
public class Invoice
{
    public int Id { get; set; }

    public double Amount { get; set; }
```

```
public Invoice(int id, double amount)
{
    Id = id;
    Amount = amount;
}
}
```

```
public class InvoicePrinter
{
    public void Print(Invoice invoice) => Console.WriteLine($"Invoice ID: {invoice.Id}, Amount: {invoice.Amount}");
}
```

```
public class InvoiceSaver
{
    public void Save(Invoice invoice) => Console.WriteLine($"Saving Invoice {invoice.Id}");
}
```

What's right?

- **Invoice** only holds data.
- **InvoicePrinter** handles printing.
- **InvoiceSaver** handles saving.

2. Open/Closed Principle (OCP)

Definition: Software entities should be open for extension but closed for modification.

Violation: OCPViolation.cs

```
public class InvoiceProcessor
{
    public void Process(Invoice invoice, string discountType)
    {
        double finalAmount = invoice.Amount;

        if (discountType == "Seasonal")
            finalAmount *= 0.9;

        else if (discountType == "None")
            finalAmount *= 1;

        Console.WriteLine($"Final amount: {finalAmount}");
    }
}
```

What's wrong?

To add new discounts, we modify this class — **violates OCP**.

Refactored: OCPRefactored.cs

```
public interface IDiscount
{
    double ApplyDiscount(double amount);
}
```

```
}
```

```
public class NoDiscount : IDiscount
```

```
{
```

```
    public double ApplyDiscount(double amount) => amount;
```

```
}
```

```
public class SeasonalDiscount : IDiscount
```

```
{
```

```
    public double ApplyDiscount(double amount) => amount * 0.9;
```

```
}
```

```
public class InvoiceProcessor
```

```
{
```

```
    private readonly IDiscount _discount;
```

```
    public InvoiceProcessor(IDiscount discount) => _discount = discount;
```

```
    public void Process(double amount)
```

```
{
```

```
        double finalAmount = _discount.ApplyDiscount(amount);
```

```
        Console.WriteLine($"Final amount: {finalAmount}");
```

```
}
```

```
}
```

What's right?

- Easy to extend: just add a new class implementing IDiscount.

3. Liskov Substitution Principle (LSP)

Definition: Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

Violation: LSPViolation.cs

```
public class Bird
{
    public virtual void Fly() => Console.WriteLine("Flying");
}

public class Penguin : Bird
{
    public override void Fly() => throw new NotSupportedException("Penguins can't fly");
}
```

What's wrong?

- Subclass Penguin breaks the behavior of Bird (throws exception).
- **LSP Violation:** Subclasses must be replaceable for base classes.

Refactored: LSPRefactored.cs

```
using System;
```

```
namespace LSP.Refactored
```

```

{
    // Base class for all birds
    public abstract class Bird
    {
        public abstract void MakeSound();
    }

    // Interface for birds that can fly
    public interface IFlyable
    {
        void Fly();
    }

    public class Parrot : Bird, IFlyable
    {
        public override void MakeSound() => Console.WriteLine("Parrot says: Squawk");
        public void Fly() => Console.WriteLine("Parrot is flying");
    }

    public class Penguin : Bird
    {
        public override void MakeSound() => Console.WriteLine("Penguin says: Honk");
        // No Fly method here — Penguins do not fly
    }
}

```

What's right?

- ☐ ☐ Bird is an abstract class that has only common behavior — in this case, MakeSound().
- ☐ IFlyable is an interface implemented **only by birds that can fly**, like Parrot.

❑ Penguin does **not** implement IFlyable, so calling Fly() on a Penguin is not even possible — this enforces correct design and behavior.

❑ Now, **LSP is preserved** because we can safely substitute Parrot or Penguin where a Bird is expected without errors.

4. Interface Segregation Principle (ISP)

Definition: Clients should not be forced to depend on interfaces they do not use.

Violation: ISPViolation.cs

```
public interface IMachine
```

```
{
```

```
    void Print();
```

```
    void Scan();
```

```
}
```

```
public class OldPrinter : IMachine
```

```
{
```

```
    public void Print() => Console.WriteLine("Printing...");
```

```
    public void Scan() => throw new NotImplementedException();
```

```
}
```

What's wrong?

- OldPrinter must implement Scan() even though it doesn't scan.

Refactored: ISPRefactored.cs

```
public interface IPrinter
```

```
{  
    void Print();  
}
```

```
public interface IScanner
```

```
{  
    void Scan();  
}
```

```
public class SimplePrinter : IPrinter
```

```
{  
    public void Print() => Console.WriteLine("Printing...");  
}
```

```
public class MFP : IPrinter, IScanner
```

```
{  
    public void Print() => Console.WriteLine("Printing...");  
    public void Scan() => Console.WriteLine("Scanning...");  
}
```

What's right?

☐ Interfaces are small and focused.

- Each class only implements what it needs.

5. Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules; both should depend on abstractions.

Violation: DIPViolation.cs

```
public class InvoiceLogger
{
    public void Log(string message) => Console.WriteLine($"[Log]: {message}");
}
```

```
public class InvoiceService
{
    private InvoiceLogger _logger = new InvoiceLogger();
    public void GenerateInvoice(string id)
    {
        _logger.Log($"Generated invoice {id}");
    }
}
```

What's wrong?

- InvoiceService is tightly coupled to InvoiceLogger.

Refactored: DIPRefactored.cs

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

public class ConsoleLogger : ILogger
{
    public void Log(string message) => Console.WriteLine($"[Log]: {message}");
}

public class InvoiceService
{
    private readonly ILogger _logger;
    public InvoiceService(ILogger logger) => _logger = logger;

    public void GenerateInvoice(string id)
    {
        _logger.Log($"Generated invoice {id}");
    }
}
```

What's right?

Depends on an abstraction (ILogger), not a concrete class.

Program.cs – Ties Everything Together

```
static void Main()
{
    var invoice = new Invoice(101, 5000);

    // SRP
    new InvoicePrinter().Print(invoice);
    new InvoiceSaver().Save(invoice);

    // OCP
    var processor = new InvoiceProcessor(new SeasonalDiscount());
    processor.Process(invoice.Amount);

    // LSP
    Bird parrot = new Parrot();
    Bird penguin = new Penguin();
    parrot.MakeSound();
    penguin.MakeSound();

    // ISP
    IPrinter printer = new SimplePrinter();
    printer.Print();
    var mfp = new MFP();
    mfp.Print();
    mfp.Scan();

    // DIP
```

```

ILogger logger = new ConsoleLogger();

var service = new InvoiceService(logger);

service.GenerateInvoice("INV-101");

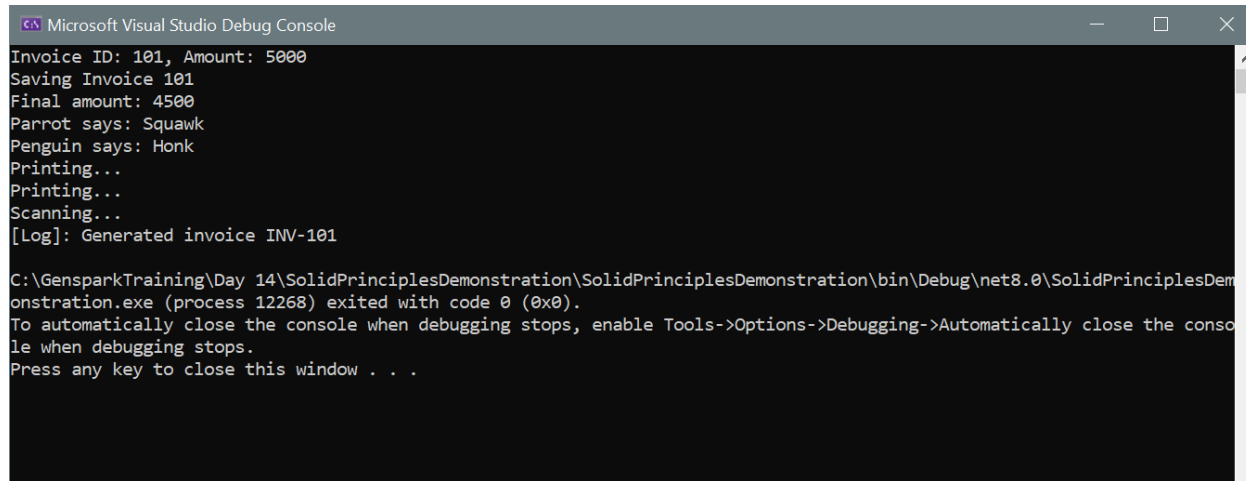
}

```

Summary of SOLID

Principle	Violation	Refactored Benefit
SRP	Class with multiple responsibilities	Each class has one reason to change
OCP	if-else for discount	Add new logic via new classes
LSP	Subclass breaks behavior	Subclasses follow base contract
ISP	Fat interface	Segregated into smaller interfaces
DIP	Concrete dependency	Depend on abstraction (interface)

Output:



```

Microsoft Visual Studio Debug Console
Invoice ID: 101, Amount: 5000
Saving Invoice 101
Final amount: 4500
Parrot says: Squawk
Penguin says: Honk
Printing...
Printing...
Scanning...
[Log]: Generated invoice INV-101

C:\GensparkTraining\Day 14\SolidPrinciplesDemonstration\SolidPrinciplesDemonstration\bin\Debug\net8.0\SolidPrinciplesDem
onstration.exe (process 12268) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .

```