SOLID

SOLID, C#

SOLID

1. Single Responsibility Principle
2. Open/Closed Principle
3. Liskov Substitution Principle
4. Interface Segregation Principle
5. Dependency Inversion Principle

What is SOLID?

The **SOLID principles** are fundamental design principles in object-oriented programming that help create clean, maintainable, and scalable code. Here's a breakdown of each SOLID principle in C# with examples:

1. Single Responsibility Principle (SRP)

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

Example: Consider a class `Invoice` that calculates the total and also saves the invoice to a file. This violates SRP because it has two responsibilities.

```
public class Invoice
{
    public decimal CalculateTotal() { /* ... */ }
    public void SaveToFile() { /* ... */ }  // This is a separate responsibility.
}
```

Solution: Split the responsibilities into two classes.

```
public class Invoice
{
    public decimal CalculateTotal() { /* ... */ }
}
public class InvoiceSaver
{
    public void SaveToFile(Invoice invoice) { /* ... */ }
}
```

Now, `Invoice` only calculates the total, and `InvoiceSaver` handles saving, making each class focused on a single responsibility.

2. Open/Closed Principle (OCP)

Definition: Classes should be open for extension but closed for modification, allowing behavior to be extended without modifying existing code.

Example: Suppose we have a `DiscountCalculator` that calculates discounts for different customer types. Instead of adding conditions in the `CalculateDiscount` method, we can use polymorphism.

```
public class DiscountCalculator
{
        public decimal CalculateDiscount(Customer customer)
        {
        if (customer.Type == "Regular")
        return customer.PurchaseAmount * 0.1M;
        if (customer.Type == "VIP")
        return customer.PurchaseAmount * 0.2M;
        return 0;
        }
}
```

Solution: Apply inheritance to handle different customer types.

```
public abstract class DiscountCalculator
{
        public abstract decimal CalculateDiscount(decimal amount);
}
public class RegularCustomerDiscount : DiscountCalculator
{
        public override decimal CalculateDiscount(decimal amount) => amount * 0.1M;
}
public class VIPCustomerDiscount : DiscountCalculator
{
        public override decimal CalculateDiscount(decimal amount) => amount * 0.2M;
}
```

Now, we can add new customer types by extending `DiscountCalculator` without modifying existing code.

Liskov Substitution Principle

Definition: Subtypes must be substitutable for their base types. Derived classes should be usable in place of their base classes without affecting the correctness of the program.

Example: Suppose we have a `Bird` base class and a `Penguin` subclass. Penguins can't fly, so overriding `Fly()` with an exception violates LSP.

```
public class Bird

{

        public virtual void Fly() { /* ... */ }

}

public class Penguin : Bird

{       public override void Fly() => throw new InvalidOperationException("Penguins can't fly!");

}
```

Solution: Redesign the class hierarchy to respect LSP.

```
public abstract class Bird { }


public class FlyingBird : Bird
{
        public void Fly() { /* ... */ }
}


public class Penguin : Bird
{
        // Penguins don't have a Fly method.
}
```

Now `Penguin` doesn’t need to override a behavior it doesn’t support.

4. Interface Segregation Principle (ISP)

Definition: Clients should not be forced to depend on interfaces they don't use. Create smaller, more specific interfaces.

Example: If we have a large `IPrinter` interface that includes methods for both printing and scanning, classes implementing it are forced to implement all methods even if they don't use them.

```
public interface IPrinter
{       void Print();
        void Scan();
}
public class SimplePrinter : IPrinter
{       public void Print() { /* ... */ }
        public void Scan() { throw new NotImplementedException(); } // Not needed
}
```

Solution: Split `IPrinter` into smaller interfaces.

```
public interface IPrint
{
        void Print();
}
public interface IScan
{
        void Scan();
}
public class SimplePrinter : IPrint
{
        public void Print() { /* ... */ }
}
```

Now `SimplePrinter` only implements `IPrint`, while other classes that require scanning can implement `IScan`.

5. Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions, and abstractions should not depend on details.

Example: In the example below, `OrderProcessor` directly depends on `EmailNotification`.

```
public class EmailNotification
{       public void SendEmail() { /* ... */ }
}
public class OrderProcessor
{       private EmailNotification _notification = new EmailNotification();
        public void ProcessOrder()
        {       // Process order logic       _notification.SendEmail();       }
}
```

Solution: Introduce an abstraction `INotification` that `OrderProcessor` depends on.

```
public interface INotification

{       void Notify(); }

public class EmailNotification : INotification

{       public void Notify() { /* ... */ }  }

public class OrderProcessor

{       private readonly INotification _notification;

        public OrderProcessor(INotification notification)

        {       _notification = notification;     }

        public void ProcessOrder()

        {       // Process order logic

        _notification.Notify();

        }

}
```

Now `OrderProcessor` depends on the `INotification` interface, making it flexible to use other notification types, such as SMS or push notifications.

Summary

- **SRP**: A class should have only one reason to change.
- **OCP**: A class should be open for extension but closed for modification.
- **LSP**: Subclasses should be substitutable for their base classes.
- **ISP**: Make interfaces specific to client needs.
- **DIP**: Depend on abstractions, not on concrete implementations.