The SOLID principles are a set of five design principles intended to make software designs more understandable, flexible, and maintainable. They were introduced by Robert C. Martin, also known as Uncle Bob, and they serve as a foundation for good object-oriented programming practices.

Here's a breakdown of each principle with examples and explanations:

---

1. Single Responsibility Principle (SRP)

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

Explanation: When a class has only one responsibility, it is easier to understand, test, and maintain. If a class handles multiple responsibilities, a change in one responsibility could impact the others.

Example: Suppose we have a `Report` class that handles report generation and saving the report to a file.

Before SRP:

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

    public void GenerateReport()
    {
        // Logic for report generation
    }

    public void SaveReportToFile(string filename)
    {
        // Logic to save report to a file
    }
}
```

After SRP: To adhere to SRP, we can separate the responsibilities into two classes: `ReportGenerator` for generating reports and `FileSaver` for saving them.

```
public class Report
{
```

```
    public string Content { get; set; }
}

public class ReportGenerator
{
    public Report GenerateReport()
    {
        // Logic for generating a report
        return new Report();
    }
}

public class FileSaver
{
    public void SaveReportToFile(Report report, string filename)
    {
        // Logic to save report to a file
    }
}
```

---

2. Open/Closed Principle (OCP)

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

Explanation: A class should allow its behavior to be extended without modifying its source code. This can often be achieved through inheritance or interfaces, allowing new functionality by adding new code rather than changing existing code.

Example: Suppose we have a `PaymentProcessor` class that processes payments with different methods (credit card, PayPal).

Before OCP:

```
public class PaymentProcessor
{
    public void ProcessPayment(string paymentMethod)
    {
        if (paymentMethod == "CreditCard")
        {
            // Process credit card payment
```

```
        }
        else if (paymentMethod == "PayPal")
        {
            // Process PayPal payment
        }
    }
}
```

After OCP: To make it more extendable, we can use an interface and add new payment methods without modifying the `PaymentProcessor`.

```
public interface IPaymentMethod
{
    void Process();
}

public class CreditCardPayment : IPaymentMethod
{
    public void Process()
    {
        // Process credit card payment
    }
}

public class PayPalPayment : IPaymentMethod
{
    public void Process()
    {
        // Process PayPal payment
    }
}

public class PaymentProcessor
{
    public void ProcessPayment(IPaymentMethod paymentMethod)
    {
        paymentMethod.Process();
    }
}
```

Now, if we need to add a new payment method, we simply create a new class that implements `IPaymentMethod` without modifying the `PaymentProcessor`.

---

3. Liskov Substitution Principle (LSP)

Definition: Subtypes must be substitutable for their base types without altering the correctness of the program.

Explanation: This principle ensures that derived classes extend the base class without changing its behavior, allowing the derived class to be used anywhere the base class is expected.

Example: Suppose we have a `Bird` class and a `Penguin` class.

Before LSP:

```
public class Bird
{
    public virtual void Fly()
    {
        // Logic for flying
    }
}

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

The `Penguin` class violates LSP because it cannot fulfill the behavior of the `Bird` class (it cannot fly). To follow LSP, we should avoid forcing subclasses to implement behaviors they can’t support.

After LSP:

```
public abstract class Bird
{
    // Common bird properties and methods
}
```

```
public class FlyingBird : Bird
{
    public void Fly()
    {
        // Logic for flying
    }
}

public class Penguin : Bird
{
    // Penguin-specific properties and methods
}
```

Now, `FlyingBird` and `Penguin` both inherit from `Bird`, but only `FlyingBird` has the `Fly` method.

---

4. Interface Segregation Principle (ISP)

Definition: A class should not be forced to implement interfaces it does not use.

Explanation: ISP encourages the creation of smaller, more specific interfaces rather than large, general-purpose ones. This way, a class only implements the methods it needs.

Example: Suppose we have a `IMultiFunctionDevice` interface that includes methods for printing, scanning, and faxing.

Before ISP:

```
public interface IMultiFunctionDevice
{
    void Print();
    void Scan();
    void Fax();
}

public class Printer : IMultiFunctionDevice
{
    public void Print() { /* Implementation */ }
    public void Scan() { /* Implementation */ }
    public void Fax() { throw new NotSupportedException(); }
```

```
}
```

The `Printer` class does not need to implement `Fax()`, but it's forced to because of the large interface.

After ISP:

```
public interface IPrinter
{
    void Print();
}

public interface IScanner
{
    void Scan();
}

public interface IFax
{
    void Fax();
}

public class Printer : IPrinter
{
    public void Print() { /* Implementation */ }
}
```

Now, `Printer` only implements `IPrinter` and is not forced to implement unnecessary methods.

---

5. Dependency Inversion Principle (DIP)

Definition: 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.

Explanation: DIP helps decouple code, allowing high-level modules to interact with low-level modules through interfaces or abstractions, which makes the code more flexible and testable.

Example: Suppose we have a `LightSwitch` class that depends on a `LightBulb` class.

Before DIP:

```
public class LightBulb
{
    public void TurnOn() { /* Turn on light */ }
    public void TurnOff() { /* Turn off light */ }
}

public class LightSwitch
{
    private LightBulb _bulb;

    public LightSwitch()
    {
        _bulb = new LightBulb();
    }

    public void Toggle(bool on)
    {
        if (on) _bulb.TurnOn();
        else _bulb.TurnOff();
    }
}
```

Here, `LightSwitch` is tightly coupled to `LightBulb`.

After DIP: To decouple `LightSwitch` from `LightBulb`, we can use an interface.

```
public interface ILight
{
    void TurnOn();
    void TurnOff();
}

public class LightBulb : ILight
{
    public void TurnOn() { /* Turn on light */ }
    public void TurnOff() { /* Turn off light */ }
}
```

```
public class LightSwitch
{
    private readonly ILight _light;

    public LightSwitch(ILight light)
    {
        _light = light;
    }

    public void Toggle(bool on)
    {
        if (on) _light.TurnOn();
        else _light.TurnOff();
    }
}
```

Now, `LightSwitch` depends on the `ILight` interface rather than a specific `LightBulb` implementation, making it more flexible and easier to test.