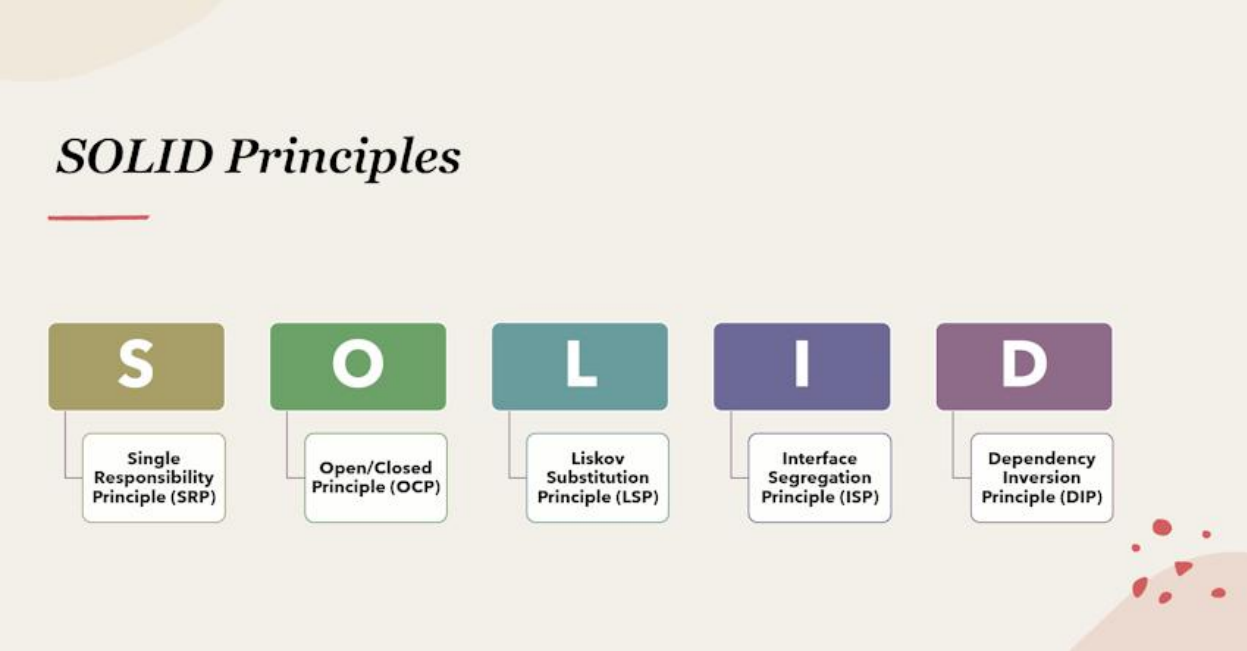
1. Overview of SOLID Principles

SOLID is an acronym that represents:

1. **S – Single Responsibility Principle (SRP)**
2. **O – Open/Closed Principle (OCP)**
3. **L – Liskov Substitution Principle (LSP)**
4. **I – Interface Segregation Principle (ISP)**
5. **D – Dependency Inversion Principle (DIP)**

Each principle targets a specific design problem, but together they form a powerful guideline for object-oriented programming and system architecture.



1.1 Single Responsibility Principle (SRP).

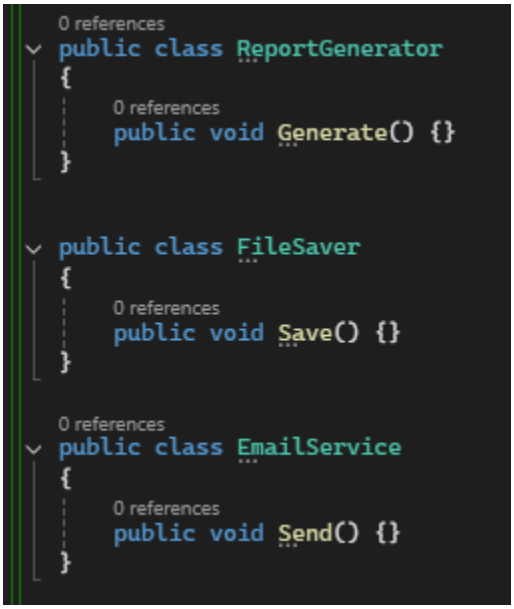
Description:

There should be a single reason for change in a class.

This implies that a class should be concerned with one task or responsibility. When a class has several responsibilities, modifications in a single responsibility may have adverse impacts on other responsibilities, which enhances bugs and complexity in maintenance.

Analogy:

One subject is taught by a school teacher. When the same individual is compelled to teach Math, Science, and English, when one curriculum is altered, it affects all the workloads of the individual.



1.2 Open/Closed Principle (OCP)

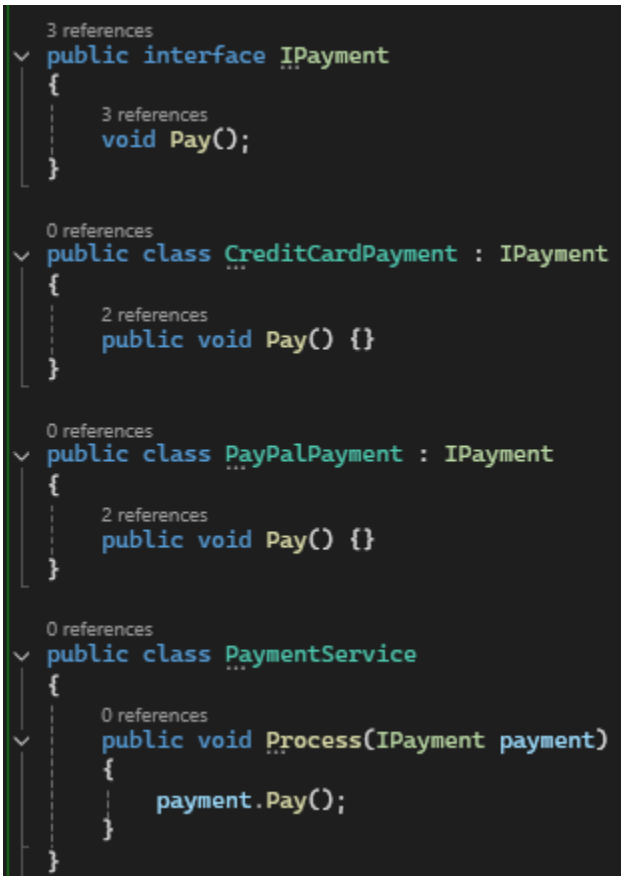
Description:

A class must be allowed to be extended but not allowed to be changed.

This implies that you must not add any new functionality and modify the already tested code.

Analogy:

The smartphone enables one to install new applications. The OS of the phone does not require internal modification each time you desire to have new features extensions without having to alter core code.



```
3 references
public interface IPayment
{
    3 references
    void Pay();
}

0 references
public class CreditCardPayment : IPayment
{
    2 references
    public void Pay() {}
}

0 references
public class PayPalPayment : IPayment
{
    2 references
    public void Pay() {}
}

0 references
public class PaymentService
{
    0 references
    public void Process(IPayment payment)
    {
        payment.Pay();
    }
}
```

1.3 Liskov Substitution Principle (LSP)

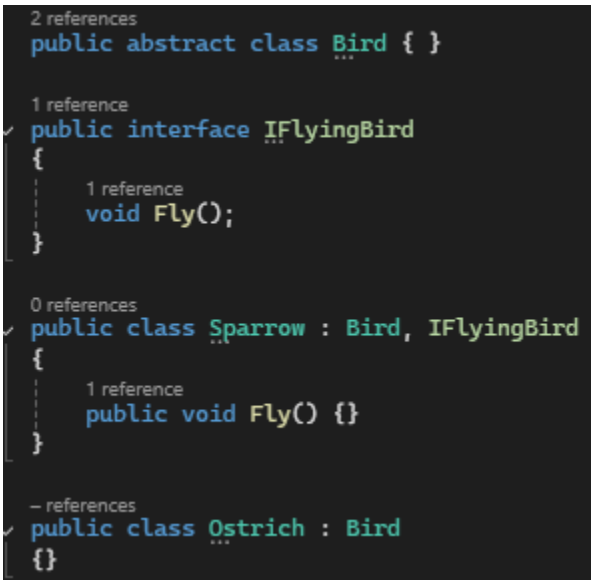
Description

Subclasses are supposed to replace their base classes without changing their correctness.

In case B is a subclass of A, any instance of B must be usable in place of A without any unanticipated behaviour.

Analogy

When a system anticipates a vehicle, it must be capable of taking a car, bike or bus without the system failing.



1.4 Interface Segregation Principle (ISP)

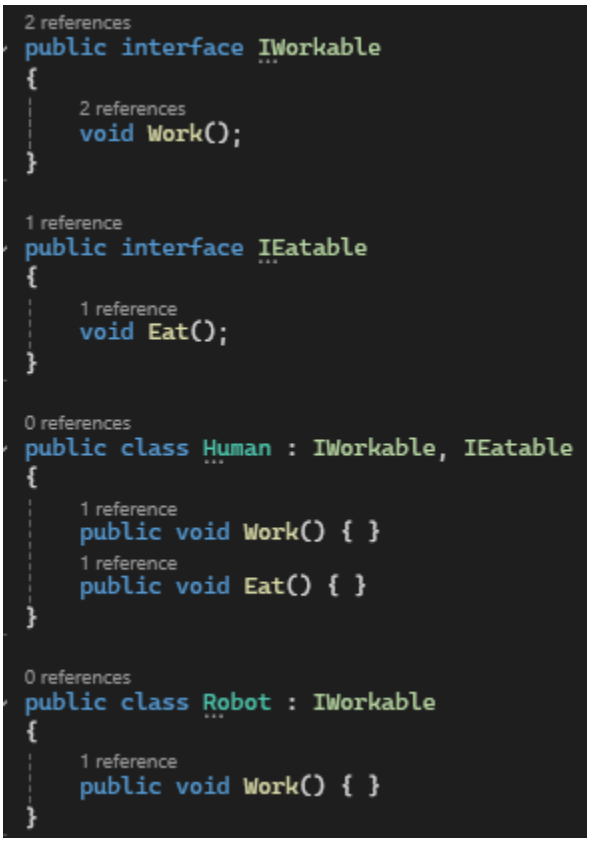
Description:

No customer must be made to rely on the practices that it does not practice.

Big interfaces must be broken down to small specific interfaces.

Analogy

:There are several categories of a restaurant menu. A vegetarian is not supposed to be compelled to order meat. Likewise, the class should not be subjected to irrelevant methods.



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

1 reference
public interface IEatable
{
    1 reference
    void Eat();
}

0 references
public class Human : IWorkable, IEatable
{
    1 reference
    public void Work() { }
    1 reference
    public void Eat() { }
}

0 references
public class Robot : IWorkable
{
    1 reference
    public void Work() { }
}
```

1.5 Dependency Inversion Principle (DIP).

Description:

Low-level modules should not be relied upon by high-level modules.

Both are supposed to rely on abstractions.

The details must not be reliant on the abstractions, but the abstractions should be reliant on the details.

Analogy:

Standard plug interface is used in a laptop charger. It is not dependent on the wiring of the house.

```
public interface IMessageService
{

    void Send();
}


public class EmailService : IMessageService
{

    public void Send() { }
}

0 references
public class SMSService : IMessageService
{
    2 references
    public void Send() { }
}

1 reference
public class Notification
{
    private readonly IMessageService _service;

    0 references
    public Notification(IMessageService service)
    {
        _service = service;
    }

    0 references
    public void Send() => _service.Send();
}
```