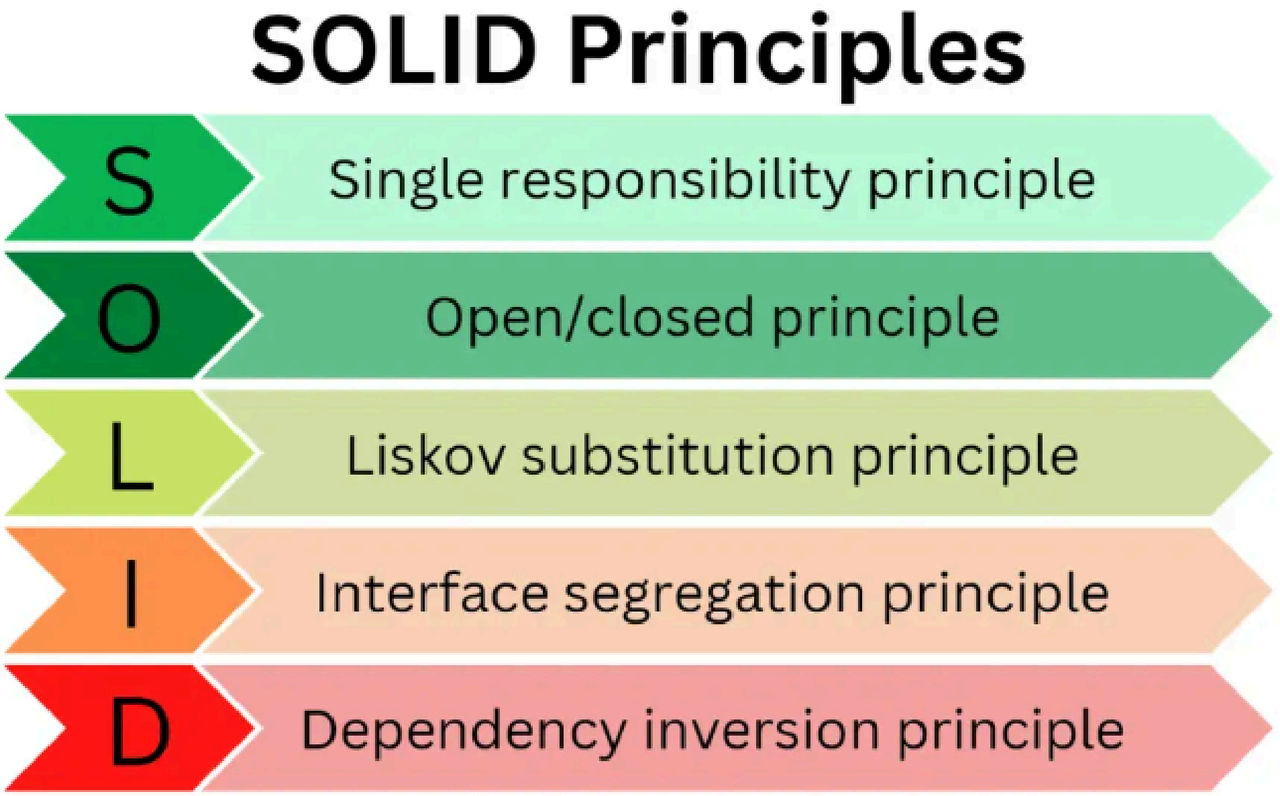
Session 4 :
Solid Principle

Md Abdullah All Naim

Software Engineer Level II
Associate Program Manager @ CodeCamp by Astha.IT

Table of Contents



SOLID Principle

SOLID is a set of 5 object-oriented design principles that help us write:

- ✓ Clean code
- ✓ Maintainable code
- ✓ Scalable code
- ✓ Easy-to-test code

These principles were introduced by Robert C. Martin (Uncle Bob).

In short:

SOLID teaches us how to design good classes and systems in OOP.

Remember the target is not to achieve it 100% all the time, the intention is to achieve it as much as feasible.

SOLID Principle

🧠 Simple Real-Life Analogy

Think of building a house 🏠

- S → Electrician only does wiring
- O → You can add a new room without breaking old rooms
- L → Any door replacement should still fit the door frame
- I → Don’t force someone to use tools they don’t need
- D → Use switches (abstraction), not direct wires

SOLID Principle

🧱 Why Do We Need SOLID?

Imagine a real project without SOLID:

- **One class** does everything 😵
- Small **changes break many features**
- Code is **hard to understand**
- Difficult to test
- Nightmare for team work

SOLID principles prevent these problems.

SOLID Principle

🎯 When Should You Use SOLID?

- Medium to large projects
- Team-based development
- Long-term software maintenance
- Enterprise / production systems

⚠️ ***Not mandatory for tiny scripts, but best practice.***

🔑 One Powerful Line to Remember

SOLID helps you write code that doesn’t break when requirements change.

S -> SOLID Principle

S = Single Responsibility Principle

Definition:
A class should have only **one responsibility and only one reason to change**.

🧠 **Meaning in Simple Words**
👉 One class = one job
👉 One reason to change
👉 **Don’t mix different responsibilities in one class**

S -> SOLID Principle

❌ Why SRP is Needed (Problem First)

Imagine this situation 👇

- Business logic changes
- Database changes
- Email system changes

If one class handles all of these, then:

- One change breaks many things
- Debugging becomes painful
- Testing is hard

S -> SOLID Principle

🧩 Real-Life Analogy

❌ Without SRP

A student who:

- Studies 📚
- Teaches class 👨‍🏫
- Manages accounts 💰
- Cleans classroom 🧹

Too many responsibilities ❌

Any change affects the same person.

✅ With SRP

Separate roles:

- Student → studies
- Teacher → teaches
- Accountant → manages money
- Cleaner → cleans

Each role has one responsibility ✓

See : Example 1

O -> SOLID Principle

O = Open for extension, Closed for modification

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

🧠 Meaning in Simple Words
👉 You should be able to **add new behavior**
👉 **Without changing existing**, tested code

Why?
Because **changing old code = risk of bugs** 🚨

O -> SOLID Principle

💳 Payment Gateway Analogy (Super Simple)

❌ WITHOUT OCP

Imagine an online shop with **only one cashier**.
The cashier knows:

- Card payment
- Bkash
- Nagad

Now a new payment comes: PayPal

👉 The cashier must **re-learn everything**
👉 Old payment flow is **disturbed**
👉 High chance of **mistake** ❌

Every new payment = change the cashier

O -> SOLID Principle

✅ **WITH OCP**

Now imagine:

- The shop has a Payment Counter
- Behind it are different payment machines

Each machine knows only one payment:

- Card machine
- Bkash machine
- Nagad machine

When PayPal comes:

- Just plug in a new PayPal machine
- Payment counter works the same

✓ **No old system changed**
✓ **New payment added easily**

O -> SOLID Principle

🎯 Why OCP Is Critical in Real Projects

✔ Prevents **breaking existing features**
✔ Encourages **polymorphism**
✔ Makes system **scalable**
✔ **Core principle** in plugin systems

🔑 Interview Gold Line

“OCP is achieved using abstraction + polymorphism, not if-else.”

See Example 2

L -> SOLID Principle

L = Liskov Substitution Principle

A child class must be **usable anywhere its parent class is used**, without breaking the program.

🧠 Meaning in Very Simple Words
👉 If B is a child of A
👉 Then B should behave like A, not surprise the system

If replacing parent with child breaks logic → ❌ LSP violation

L -> SOLID Principle

🧩 Super Easy Real-Life Analogy (Payment Gateway)

❌ WITHOUT LSP

Imagine:

- PaymentGateway promises: “I can process payments”

Now a child gateway:

- **CashOnDeliveryGateway**
  - **Throws error: “Online payment not supported”**

So if system says: “Process payment”

And COD says: “I can’t do that”

👉 **Child breaks the promise ❌**

👉 **LSP violated**

L -> SOLID Principle

✅ **WITH LSP**

All payment gateways:

- Card
- Bkash
- Nagad
- PayPal

All of them:

✓ Accept amount
✓ Process payment
✓ Return success / failure

Now system doesn’t care which gateway it uses.

👉 Parent replaced safely ✓
👉 LSP followed ✓

L -> SOLID Principle

🧠 **One-Line Rule to Remember**

- Child class should not **remove or weaken parent behavior.**
- LSP ensures **inheritance is used correctly**, so **polymorphism doesn't break behavior.**

🔁 **Relationship with O & S**

- SRP → clean responsibilities
- OCP → extend safely
- LSP → replace safely

See Exampl 3,4

I -> SOLID Principle

I = Interface Segregation Principle

Definition (Simple):
“No client should be forced to depend on methods it does not use.”

In simple words:

- Don’t make **huge interfaces** with stuff some classes **don’t need.**
- **Split big interfaces into smaller, focused ones.**

I -> SOLID Principle

❌ **Without ISP (Bad Design)**

Imagine a Payment Interface for an online store:

- CardPayment class only needs PayWithCard()
- But it must implement all the other methods, even if it does nothing or throws exceptions

💥 **Problem: Classes are forced to implement stuff they don't need**

```
interface IPaymentGateway
{
    void PayWithCard(decimal amount);
    void PayWithBkash(decimal amount);
    void PayWithNagad(decimal amount);
    void PayWithCashOnDelivery(decimal amount);
}
```

I -> SOLID Principle

✅ With ISP (Good Design)

Split the interface into small focused interfaces:

```
interface ICardPayment
{
    void PayWithCard(decimal amount);
}

interface IBkashPayment
{
    void PayWithBkash(decimal amount);
}

interface ICashOnDelivery
{
    void PayWithCOD(decimal amount);
}
```

Now:

- **CardPayment implements only ICardPayment**
- **BkashPayment implements only IBkashPayment**
- **CODPayment implements only ICashOnDelivery**

✅ **Classes only depend on what they actually use**

I -> SOLID Principle

SRP (Single Responsibility Principle)

- Meaning: Everyone does only one job.
- House example: Electrician → only wires, Plumber → only pipes, Painter → only paint walls.
- Easy line: “One person, one job.” ✅

ISP (Interface Segregation Principle)

- Meaning: Give people only the tools they need.
- House example: Electrician doesn’t get paintbrushes or wrenches — only screwdrivers and wires.
- Easy line: “Don’t give extra tools nobody uses.” ✅

Easy connection:

- SRP → who does the job
- ISP → what tools they get

D-> SOLID Principle

D = Dependency Inversion Principle

“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.**

🧠 **Meaning in Simple Words**

1. High-level modules → Your **main business logic**
2. Low-level modules → **Helper classes**, concrete implementations
3. Abstraction → Interface or abstract class

Don’t let your main code directly depend on concrete implementations.
Depend on interfaces, not classes.

D-> SOLID Principle

🧩 Real-Life Analogy: Payment Gateway

❌ Without DIP (Bad Design)

```
class OnlineStore
{
    private CardPayment _cardPayment;

    public OnlineStore()
    {
        _cardPayment = new CardPayment(); // Direct dependency
    }

    public void Checkout(decimal amount)
    {
        _cardPayment.Pay(amount); // Tightly coupled
    }
}

class CardPayment
{
    public void Pay(decimal amount)
    {
        Console.WriteLine($"Paid {amount} using Card");
    }
}
```

Problem:

- OnlineStore only works with CardPayment
- Add BkashPayment, PayPal, or CashOnDelivery → you must modify OnlineStore
- High-level module depends on low-level module ❌

D-> SOLID Principle

🧩 Real-Life Analogy: Payment Gateway
✅ With DIP (Good Design)

```
// Abstraction
interface IPaymentMethod
{
    void Pay(decimal amount);
}

// Low-level modules
class CardPayment : IPaymentMethod
{
    public void Pay(decimal amount)
    {
        Console.WriteLine($"Paid {amount} using Card");
    }
}

class BkashPayment : IPaymentMethod
{
    public void Pay(decimal amount)
    {
        Console.WriteLine($"Paid {amount} using Bkash");
    }
}

// High-level module depends on abstraction
class OnlineStore
{
    private readonly IPaymentMethod _paymentMethod;

    public OnlineStore(IPaymentMethod paymentMethod)
    {
        _paymentMethod = paymentMethod;
    }

    public void Checkout(decimal amount)
    {
        _paymentMethod.Pay(amount); // Works with any gateway
    }
}
```

✔ No need to modify OnlineStore

✔ Add new payment gateways easily

- ✔ High-level module depends on abstraction, not details

D-> SOLID Principle

With DIP ✅

Benefits:

- **Loose Coupling**: Depend on abstractions, not concrete implementations
- **Easy Testing**: Mock abstractions for unit tests
- **Flexible Design**: Swap implementations without changing code
- **Reduced Fragility**: Changes in low-level don't affect high-level
- **Better Scalability**: Add new implementations easily
- **Dependency Injection**: Externalize dependencies

D-> SOLID Principle

🔷 Dependency Inversion Principle (DIP) vs Dependency Injection (DI)

They are NOT the same thing.
DIP is a **principle**. DI is a **technique**.

Concept	What it is
DIP	A **design principle** (rule of good design)
DI	A **way to implement** that principle

D-> SOLID Principle

❌ Without DIP (design problem)

```
public class OrderService
{
    private CardPaymentService _payment = new CardPaymentService();
}
```

✅ With DIP (design is correct)

```
public class OrderService
{
    private IPaymentMethod _payment;

    public OrderService(IPaymentMethod payment)
    {
        _payment = payment;
    }
}
```

SOLID Principle

```
SRP (Single Responsibility)
  ↓
  └→ Makes classes focused and cohesive
  └→ Easier to apply OCP (less to extend)
  └→ Supports ISP (smaller interfaces)

OCP (Open/Closed)
  ↓
  └→ Enabled by DIP (abstractions)
  └→ Supported by ISP (focused interfaces)
  └→ Complements SRP (extend without modifying)

LSP (Liskov Substitution)
  ↓
  └→ Ensures polymorphism works correctly
  └→ Supports OCP (substitutable implementations)
  └→ Enables flexible inheritance
```

```
ISP (Interface Segregation)
  ↓
  └→ Reduces coupling (depend only on needed methods)
  └→ Supports DIP (focused abstractions)
  └→ Enables better SRP (small interfaces = single purpose)

DIP (Dependency Inversion)
  ↓
  └→ Decouples high and low-level modules
  └→ Enables OCP (swap implementations)
  └→ Supports testing (mock abstractions)
```

SOLID Principle

Principle Dependencies

Principle	Supports	Enabled By
SRP	All others	None
OCP	ISP, DIP, LSP	SRP, DIP
LSP	OCP	SRP
ISP	All others	SRP
DIP	OCP	ISP, SRP

Other Principles

KISS — Keep It Simple, Stupid

- Solve the problem in the simplest possible way

❌ Not clever
❌ Not over-engineered
✅ Easy to read
✅ Easy to change

.

Other Principles

KISS — Keep It Simple, Stupid

```
class Light
{
    public void TurnOn()
    {
        if (CheckFuse() && CheckPowerLine() && CheckSwitch())
        {
            Console.WriteLine("Light is ON");
        }
    }

    private bool CheckFuse() { return true; }
    private bool CheckPowerLine() { return true; }
    private bool CheckSwitch() { return true; }
}

class Program
{
    static void Main()
    {
        Light light = new Light();
        light.TurnOn();  // Too many checks for a simple task
    }
}
```

```
class Light
{
    public void TurnOn()
    {
        Console.WriteLine("Light is ON");
    }
}

class Program
{
    static void Main()
    {
        Light light = new Light();
        light.TurnOn();  // Simple and easy
    }
}
```

Other Principles

DRY — Don’t Repeat Yourself

Every piece of knowledge should have a single source of truth.
❌ Copy–paste logic
❌ Same validation everywhere
✅ One place → reuse

❌ Bad (Violates DRY)

```
public void RegisterUser(string email)
{
    if (!email.Contains("@"))
        throw new Exception("Invalid email");
}

public void UpdateEmail(string email)
{
    if (!email.Contains("@"))
        throw new Exception("Invalid email");
}
```

✅ DRY Version

```
public bool IsValidEmail(string email)
{
    return email.Contains("@");
}

public void RegisterUser(string email)
{
    if (!IsValidEmail(email))
        throw new Exception("Invalid email");
}

public void UpdateEmail(string email)
{
    if (!IsValidEmail(email))
        throw new Exception("Invalid email");
}
```

Other Principles

YAGNI — You Aren’t Gonna Need It

🎯 What YAGNI Really Means

- Build only what is required today
- Avoid guessing the future

Other Principles

```
// Bad: Adding a caching layer before it's needed
public class ProductService
{
    private Dictionary<int, Product> _cache = new Dictionary<int, Product>();

    public Product GetProduct(int id)
    {
        if (_cache.ContainsKey(id))
            return _cache[id];
        // ...fetch from database...
        var product = FetchFromDatabase(id);
        _cache[id] = product;
        return product;
    }

    private Product FetchFromDatabase(int id)
    {
        // Simulate DB fetch
        return new Product { Id = id, Name = "Sample" };
    }
}
```

```
// Good: Only fetch from database, add caching later if needed
public class ProductService
{
    public Product GetProduct(int id)
    {
        // ...fetch from database...
        return FetchFromDatabase(id);
    }

    private Product FetchFromDatabase(int id)
    {
        // Simulate DB fetch
        return new Product { Id = id, Name = "Sample" };
    }
}
```

Other Principles

🧠 **Senior Engineer Rule (Very Important)**

Start with:
KISS + YAGNI

As system grows:
Apply DRY

When variation appears:
Apply SOLID

THANK YOU!