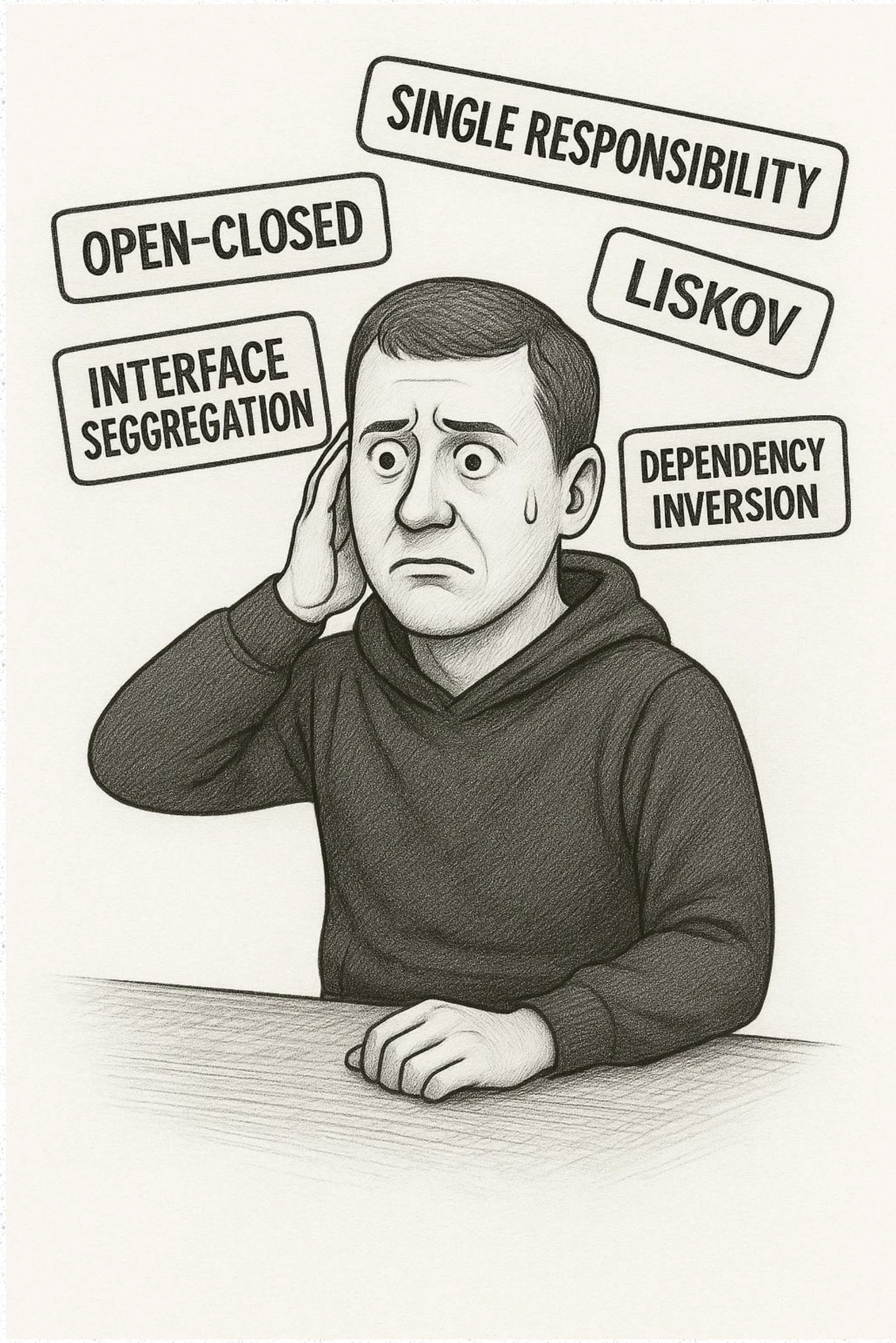
The only SOLID Principles Guide You Need




Anton Martyniuk
antondevtips.com

01 S = Single Responsibility Principle

One job. One reason to change.

- Keep features isolated (logging ≠ business logic).
- Smaller methods → easier tests → faster refactors.

```
// ❌ Violates SRP
public record Order(Guid Id, decimal Amount);
public class OrderService
{
    public void Save(Order order) { /* DB write */ }
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

01 S = Single Responsibility Principle

One job. One reason to change.

- Keep features isolated (logging ≠ business logic).
- Smaller methods → easier tests → faster refactors.

```
// ✅ SRP-friendly
public class OrderService
{
    private readonly ILogger _logger;
    private readonly IOrderRepository _repository;

    public OrderService(ILogger logger,
        IOrderRepository repository)
    {
        _logger = logger;
        _repository = repository;
    }

    public void Save(Order order)
    {
        _repository.Save(order);
        _logger.Info($"Saved order {order.Id}");
    }
}
```

02 O = Open / Closed Principle

Extend ✅ - modify ❌

- Add new behaviour with inheritance, strategy, DI.
- Proven code stays untouched.

```
// ❌ Breaks OCP — must edit class for every new payment type
public enum PaymentType { Card, Cash }
public class PaymentCalculator
{
    public decimal Fee(PaymentType type, decimal amount)
    {
        return type switch
        {
            PaymentType.Card ⇒ amount * 0.02m,
            PaymentType.Cash ⇒ 0m,
            _ ⇒ throw new NotSupportedException()
        };
    }
}
```

02 O = Open / Closed Principle

Extend ✅ - modify ❌

- Add new behaviour with inheritance, strategy, DI.
- Proven code stays untouched.

```
// ✅ Open for extension via Strategy
public interface IPaymentFee
{
    decimal Fee(decimal amount);
}

public class CardFee : IPaymentFee
{
    public decimal Fee(decimal a) ⇒ a * 0.02m;
}

public class CashFee : IPaymentFee
{
    public decimal Fee(decimal a) ⇒ 0m;
}
```

03 L = Liskov Substitution Principle

Subclasses must fully honour the base contract.

- If you replace the base with a subtype, nothing breaks.
- Don’t force penguins to fly.

```
// ❌ LSP violation — Penguin can't Fly
public abstract record Bird;
public record Penguin : Bird;
public record Eagle : Bird;

public void LetItFly(Bird bird)
{
    bird.Fly();  // Runtime Error
}
```

03 L = Liskov Substitution Principle

Subclasses must fully honour the base contract.

- If you replace the base with a subtype, nothing breaks.
- Don’t force penguins to fly.

```
// ✅ Split capability interfaces
public interface IFlyable { void Fly(); }

public record Eagle : Bird, IFlyable
{
    public void Fly()
    {
        Console.WriteLine("🦅");
    }
}

public record Penguin : Bird
{
    /* no Fly() */
}
```

04 I = Interface Segregation Principle

Small, purpose-built interfaces.

- Clients use only what they need.
- Fewer “god” interfaces, more cohesion.

```
// ❌ Bloated interface
public interface IWorker { void Work(); void Eat(); }

public class Robot : IWorker // forced to implement Eat()
{
    public void Work() { }

    public void Eat()
    {
        throw new NotSupportedException();
    }
}
```

04 I = Interface Segregation Principle

Small, purpose-built interfaces.

- Clients use only what they need.
- Fewer “god” interfaces, more cohesion.

```
// ✅ Segregated interfaces
public interface IWorkable { void Work(); }
public interface IFeedable { void Eat(); }

public class Robot : IWorkable
{
    public void Work() { /* ... */ }
}

public class Human : IWorkable, IFeedable
{
    public void Work() { /* ... */ }
    public void Eat() { /* ... */ }
}
```

05 D = Dependency Inversion Principle

Depend on abstractions, not concretes.

- High-level code knows nothing about HTTP requests or SQL.
- Swap implementations without rewriting logic.

```
// ❌ Tight coupling
public class OrderProcessor
{
    // Concrete class!
    private readonly SqlOrderRepository _repository = new();

    public void Process(Order order)
    {
        _repository.Save(order);
    }
}
```

05 D = Dependency Inversion Principle

Depend on abstractions, not concretes.

- High-level code knows nothing about HTTP requests or SQL.
- Swap implementations without rewriting logic.

```
// ✅ Inverted dependency
public interface IOrderRepository { void Save(Order order); }

public class OrderProcessor
{
    private readonly IOrderRepository _repo;
    public OrderProcessor(IOrderRepository repo) { _repo = repo; }

    public void Process(Order order) { _repo.Save(order); }
}

// Program.cs registration
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IOrderRepository, SqlOrderRepository>();
builder.Services.AddScoped<OrderProcessor>();
```

Repost & Follow Anton for more content like this

Want to improve .NET Skills?

Subscribe to my AntonDevTips newsletter