SOLID Principles Roadmap

Report as of **April, 2025**

Part 1: Single Responsibility

Part 2: Open/Closed

Part 3: Liskov Substitution

Part 4: Interface Segregation

Part 5: Dependency Inversion

Anton Martyniuk

antondevtips.com

S = Single Responsibility Principle

One job. One reason to change.

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

S = Single Responsibility Principle

```
// ❌ 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);
    }
}
```

```
// ✅ 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}");
    }
}
```

Anton Martyniuk

O = Open / Closed Principle

Extend ✅ - modify ❌

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

O = Open / Closed Principle

```
// ❌ 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()
        };
    }
}
```

```
// ✅ 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;
}
```

Anton Martyniuk

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.

L = Liskov Substitution Principle

```
// ❌ 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
}
```

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

public record Eagle : Bird, IFlyable
{
    public void Fly()
    {
        Console.WriteLine("🦅");
    }
}
public record Penguin : Bird
{
    /* no Fly() */
}
```

Anton Martyniuk

I = Interface Segregation Principle

Small, purpose-built interfaces.

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

I = Interface Segregation Principle

```
// ❌  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();
    }
}
```

```
// ✅  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() { /* ... */ }
}
```

Anton Martyniuk

D = Dependency Inversion Principle

Depend on abstractions, not concretes.

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

D = Dependency Inversion Principle

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

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

```
// ✅ 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>();
```

Anton Martyniuk

Next Steps

Hello there!
I'm Anton Martyniuk — a Microsoft MVP and Senior Tech Lead.

I have over 10 years of hands-on experience in .NET development and architecture. I've dedicated my career to empowering developers to excel in building robust, scalable systems.

Join my newsletter readers and let's build the future of .NET together!

01

Follow me on LinkedIn

I share amazing .NET and Software Development tips every day

02

Repost to your network

Share the knowledge about SOLID principle with your network

03

Subscribe to my free newsletter

5 minutes every week to improve your .NET skills and learn how to craft better software

Anton Martyniuk

antondevtips.com