FMX Academy

INFORMAÇÕES		
MÓDULO	AULA	INSTRUTOR
06 ASP.NET Core Web API – Fundamentos	01 – SOLID e Refatoração	William Julio

SOLID: 5 princípios da Programação Orientada a Objetos (POO)

OBJETIVO: Aplicação de boas práticas de design para criar código mais limpo, manutenível e escalável

- Entender cada princípio SOLID com exemplos práticos.
- Identificar code smells comuns.
- Aplicar técnicas de refatoração incremental.
- Melhorar arquitetura e legibilidade sem alterar comportamento existente.
- Criar um fluxo contínuo de melhoria do código.

1. O que é SOLID?

SOLID é um conjunto de cinco princípios de design orientado a objetos, criados para aumentar:

- **Manutenibilidade**
- **Extensibilidade**
- **Testabilidade**
- **Coerência do código**

Esses princípios ajudam a evitar acoplamento excessivo, códigos rígidos e baixa reutilização.

2. Os 5 Princípios SOLID (Explicados com Exemplos Reais)

S – Single Responsibility Principle (SRP)
Uma classe deve ter apenas um motivo para mudar.
💡 *Problema:* Classe assumindo responsabilidades demais.
💡 *Solução:* Dividir comportamentos distintos.

✔ Exemplo Anti-SRP

```csharp
public class PedidoService
{
    public void CriarPedido(Pedido pedido) { /* lógica */ }
    public void EnviarEmailConfirmacao(Pedido pedido) { /* envia e-mail */ }
    public void GerarNotaFiscal(Pedido pedido) { /* nf */ }
}
```

✔ **Aplicando SRP**

```csharp
public class PedidoService { /* apenas pedido */ }

public class EmailService { /* apenas e-mail */ }

public class NotaFiscalService { /* apenas NF */ }
```

O – Open/Closed Principle (OCP)

Aberto para extensão, fechado para modificação.
Evite alterar código existente — prefira *estender* com novas classes/comportamentos.

✔ **Exemplo com Estratégias**

Sem OCP:

```csharp
if(tipoFrete == "Sedex") { ... }
else if(tipoFrete == "PAC") { ... }
```

Com OCP:

```csharp
public interface IFreteStrategy { decimal Calcular(decimal peso); }

public class Sedex : IFreteStrategy { ... }
public class PAC : IFreteStrategy { ... }
```

L – Liskov Substitution Principle (LSP)

Subtipos devem ser substituíveis por seus tipos base.

Evite herança que quebra comportamento.
Errado:

Errado:

```csharp
public class Pato : Ave { public override void Voar() { } }
public class Pinguim : Ave { public override void Voar() { throw new Exception(); }}
```

Certo:

```csharp
public abstract class Ave { }
public class AveQueVoa : Ave { void Voar(); }
public class AveQueNaoVoa : Ave {}
```

I – Interface Segregation Principle (ISP)

Evite interfaces gordas. Divida em interfaces menores e específicas.

Errado:

```csharp
public interface IFuncionario {
    void BaterPonto();
    void EnviarRelatorio();
    void DesenvolverSoftware();
}
```

Certo:

```csharp
public interface IPontual { void BaterPonto(); }
public interface IReport { void EnviarRelatorio(); }
public interface IDev { void DesenvolverSoftware(); }
```

D – Dependency Inversion Principle (DIP)

Dependa de abstrações, não de implementações.

Errado:

```csharp
var repo = new UsuarioRepository();
```

Certo:

```csharp
public class UsuarioService
{
    private readonly IUsuarioRepository _repo;
    public UsuarioService(IUsuarioRepository repo) { _repo = repo; }
}
```

Code Smells – O que procurar antes de refatorar

Code Smell	Sintoma	Consequência
God Class	Classe gigante	Difícil de testar/manter
Long Method	Método com dezenas de linhas	Quebra a legibilidade
Duplicação de Código	Mesma lógica em vários lugares	Bugs replicados
Acoplamento excessivo	Classes dependem diretamente	Impacto em cascata
Switch/ifs gigantes	Lógica condicional enorme	Viola OCP
Falta de coesão	Classe faz "de tudo um pouco"	Viola SRP

4. Técnicas de Refatoração (Com Exemplo Prático)

✔ 4.1 Extract Method

Simplifique métodos gigantes.

Antes:

```csharp
public void Processar()
{
    // validações
    // cálculos
    // logs
    // persistência
}
```

Depois:

```csharp
public void Processar()
{
    Validar();
    Calcular();
    RegistrarLog();
    Persistir();
}
```

✔ 4.2 Introduce Interface / Abstração

✔ 4.2 Introduce Interface / Abstração

Para aplicar DIP.

Antes:

```csharp
var repo = new ProdutoRepository();
```

Depois:

```csharp
public ProdutoService(IProdutoRepository repo) { ... }
```

✔ 4.3 Replace Conditional with Polymorphism

Perfeito para OCP.

Antes:

```csharp
if (plano == "Premium") { ... }
else if (plano == "Gold") { ... }
```

Depois:

```csharp
public interface IPlano { decimal Calcular(); }
```

✔ **4.4 Extract Class**

Para dividir classes enormes (SRP).

5. Fluxo de Refatoração Contínua (Ciclo Ideal)

1. Identificar code smells
2. Criar testes automáticos (se não existirem)
3. Aplicar refatoração incremental
4. Garantir que os testes continuam passando
5. Revisão entre pares (Code Review)
6. Documentar padrões e boas práticas

6. Mini-Case: Aplicando SOLID no Mundo Real

Cenário:

- Um projeto possui uma classe *PagamentoService* que:
- Processa pagamento
- Envia e-mail
- Atualiza estoque
- Faz log
- Gera recibo
- Violação: **SRP, OCP, DIP**

Solução:

- Criar interfaces para cada responsabilidade
- Usar injeção de dependência
- Aplicar estratégias de pagamento
- Extrair serviços especializados
- Resultado:
- Código modular
- Testável
- Escalável
- Leitura clara

Resultado:

- Código modular
- Testável
- Escalável
- Leitura clara

7. Conclusão da Aula

A aplicação dos princípios SOLID e das técnicas de refatoração não são “regras rígidas”, mas **guias práticos** que te ajudam a evoluir sistemas complexos com segurança e clareza.

Continue sempre refatorando — código bom hoje não será código bom daqui 1 ano.