Interfaces em C# - Organizando sistemas

Mário Leite

Quando começamos a programar, é natural pensar apenas em **classes** e **objetos**; criamos uma classe, instanciamos, chamamos métodos e tudo parece funcionar; porém, à medida que os sistemas crescem, surge um problema sério: o acoplamento excessivo entre as partes do código. Uma classe passa a depender diretamente de outra, que depende de outra, formando uma cadeia difícil de manter e quase impossível de modificar sem quebrar tudo. E é nesse ponto que entram as **interfaces**, que em algumas linguagens é muito difícil de simular e convencer (como em Python); em outras, como as tradicionais C, Pascal. Fortran, COBOL é impossível). Em **C#** isto é relativamente bem fácil, pois pode ser entendida como um contrato: ela define quais métodos uma classe deve possuir, mas não define como esses métodos são implementados; assim, a interface não contém lógica de negócio, apenas a assinatura das operações.

Quando uma classe implementa uma interface, ela se compromete a cumprir esse contrato; isto permite que diferentes classes ofereçam comportamentos distintos, mas sejam tratadas de forma uniforme pelo sistema. Em outras palavras, o código passa a depender de abstrações e não de implementações concretas; e o grande ganho nesse cenário está no desacoplamento, pois um módulo não precisa saber qual é a classe real que está sendo utilizada, desde que ela implemente a interface esperada. Assim, é possível trocar uma implementação por outra sem alterar o restante do sistema. Este conceito é a base de práticas modernas como injeção de dependência, arquitetura em camadas, testes automatizados e praticamente todos os *frameworks* profissionais do mercado. Sem interfaces, esses modelos simplesmente não funcionam de forma elegante.

Outro ponto importante é o polimorfismo: com interfaces, uma variável pode representar diferentes objetos em tempo de execução, desde que todos cumpram o mesmo contrato. Isto torna o código mais flexível, mais genérico e muito mais reutilizável. Diferente da herança tradicional, uma classe pode implementar várias interfaces ao mesmo tempo, o que resolve uma limitação clássica das linguagens orientadas a objetos. Deste modo, uma mesma classe pode assumir múltiplos papéis no sistema sem ficar presa a uma hierarquia rígida. Em termos de arquitetura, pensar em interfaces é pensar primeiro no “o que o sistema faz” e só depois no “como ele faz”. Esta inversão de raciocínio muda completamente a forma de projetar *software* e é um dos marcos que separa programação iniciante de programação profissional.

Por fim, uma boa regra prática: sempre que perceber que um código pode ter mais de uma implementação possível, ou que pode mudar no futuro, provavelmente ele deveria ser representado por uma **interface**. Interfaces não servem apenas para organizar código, mas para organizar o pensamento de quem projeta o sistema. Os códigos abaixo demonstram um exemplo prático de uso de interfaces, criado em C#, com o seguinte cenário:

Imagine um sistema que precisa aceitar vários meios de pagamento:

- Cartão, PIX e boleto ==>sSem interface, o sistema ficaria cheio de **if/else**.
- Com interface, fica elegante e extensível.

```
Pagamento de R$ 150 realizado com CARTÃO.
Pagamento de R$ 80 realizado via PIX.
Boleto gerado no valor de R$ 300.
```

Saída da simulação

1 - PIX

```
public class PagamentoPix : IPagamento
{
    public void Pagar(decimal valor)
    {
        Console.WriteLine($"Pagamento de R$ {valor} realizado via PIX.");
    }
}
```

2 - Boleto

```
public class PagamentoBoleto : IPagamento
{
    public void Pagar(decimal valor)
    {
        Console.WriteLine($"Boleto gerado no valor de R$ {valor}.");
    }
}
```

3. Classe que usa a interface (ponto-chave)

```
public class Caixa
{
    private IPagamento formaPagamento;

    public Caixa(IPagamento formaPagamento)
    {
        this.formaPagamento = formaPagamento;
    }

    public void ProcessarCompra(decimal valor)
    {
        formaPagamento.Pagar(valor);
    }
}
```

Caixa não sabe se é cartão, pix ou boleto; ela só conhece a interface.

4. Uso real (polimorfismo em ação)

```
class Program
{
    static void Main()
    {
        Caixa c1 = new Caixa(new PagamentoCartao());
        c1.ProcessarCompra(150);

        Caixa c2 = new Caixa(new PagamentoPix());
        c2.ProcessarCompra(80);

        Caixa c3 = new Caixa(new PagamentoBoleto());
        c3.ProcessarCompra(300);
    }
}
```