Princípios de Design de Software - SOLID

Matheus Osmédio Araujo

1 Introdução

SOLID é um conjunto de princípios de Design de Software focado para a Programação Orientadas a Objetos criado por Robert C. Martin, um dos autores do *Manifesto Ágil* e do livro *The Cleaner Code*. Esses princípios são usados para aumentar a escalabilidade e a manutenibilidade do código.

2 Single-responsability Principle

Em português, "Princípio da Responsabilidade Única", diz que uma classe só deveria ter um trabalho/responsabilidade, não colocando múltiplas tarefas desconexas para uma única classe suportar.

```
namespace cozinha;

public class Bolo
{
    public Bolo(string Sabor, float Raio)
    {
        if (Raio <= 0)
        {
            throw new ArgumentOutOfRangeException("Erro");
        }
        if (Sabor is null)
        {
            throw new ArgumentNullException("Erro");
        }
        sabor = Sabor;
        raio = Raio;
        total_pedacos = 18;
    }

    public string sabor { get; }
    public float raio { get; }
    public int total_pedacos { get; set; }

}
```

Listing 1: Código em C#/C

Código no GitHub.

Nesse exemplo temos uma classe para Bolo, nela definimos todas as características importantes do bolo no nosso programa, como o seu sabor e o raio, mas quando envolve a parte de fatiar o bolo e distribuir para as pessoas, colocamos essas tarefas fora dele porque já não mais envolvem o bolo, mas sim um trabalho que seria do chefe, cozinheiro ou aniversariante.

3 Open-Closed Principle

O princípio "Open-Closed", em português "Aberto-Fechado", diz que classes e funções devem ser abertos para extensão das mesmas mas fechado para sua modificação, evitando assim problemas de desconfigurações dessas classes e funções e consequentemente seu mal-funcionamento.

```
namespace cozinha;

public class Balanca
{
```

```
    public float volume(Bolo bolo_alvo)
    {
        return (float) 3.14 * bolo_alvo.raio * bolo_alvo.raio * bolo_alvo.altura;
    }

    public float peso(Bolo bolo_alvo)
    {
        return bolo_alvo.densidade * volume(bolo_alvo);
    }

    public float soma_volumes(float[] volumes)
    {
        float soma_volumes = 0;
        foreach (float volume in volumes)
        {
            soma_volumes += volume;
        }
        return soma_volumes;
    }

    public float soma_pesos(float[] pesos)
    {
        float soma_pesos = 0;
        foreach (float peso in pesos)
        {
            soma_pesos += peso;
        }
        return soma_pesos;
    }
}
}
```

Listing 2: Código em C#/C

Código no GitHub.

No código acima podemos ver que agora temos a classe *Balanca* para retornar os volumes e pesos de cada bolo. Com o princípio do OCP (Open-Close Principle) "fechamos"essa classe para alterações que modifiquem o que já existe, como os bolos ou as funções de volume e peso, mas abrimos ela para a possibilidade expansão com a criação de funções que somam o peso e o volume.

4 Liskov's Substitution Principle

O princípio de Substituição de Liskov tem o seu nome vindo da cientista da computação Barbara Liskov, que basicamente diz que uma classe que seja derivada de uma outra classe precisa poder substituir essa mesma classe sem que haja problemas.

```
namespace geometria;

public class Isoceles : Triangulo
{
    public Isoceles()
    {
        tipo = "isoceles";
    }
    public string tipo { get; }
}
}
```

Listing 3: Código em C#/C

Código no GitHub.

Nesse exemplo nos temos o exemplo de uma classe de Triangulo e uma subclasse de Isoceles, onde ele herda todas as características do Triangulo mas difere quando se trata de tipo que é atribuído como "isoceles". E uma classe Isoceles pode ser substituída perfeitamente por uma classe triangulo por possuir todos os atributos de um triangulo.

5 Interface Segregation Principle

Esse princípio, traduzindo ficaria Princípio da Separação de Interfaces, fala sobre evitar que o usuário passe por várias interfaces desnecessárias para o resultado final que o usuário deseja, tendo como foco deixar o código, especificamente o flow de interfaces, mais limpos.

```
// Pergunta se o usuario vai querer bebida, caso nao, nem mostra pra ele as opceoes

using menu;

Pratos pratos_1 = new Pratos("Parmegiana de Frango", (float) 45, (float) 750);
Pratos pratos_2 = new Pratos("Bife Acebolado", (float) 35, (float) 670);
Pratos pratos_3 = new Pratos("Omelete com Fritas", (float) 30, (float) 500);
Pratos pratos_4 = new Pratos("Picanha", (float) 65, (float) 800);

Bebidas bebidas_1 = new Bebidas("Suco de Uva", (float) 600, "suco");
Bebidas bebidas_2 = new Bebidas("Coca-Cola", (float) 350, "refrigerante");
Bebidas bebidas_3 = new Bebidas("Guarana", (float) 2000, "refrigerante");
Bebidas bebidas_4 = new Bebidas("Agua", (float) 500, "agua");

Console.WriteLine("Ola, voce vai querer bebida? (S/N)");

if (Console.ReadKey().Key == ConsoleKey.S)
{
    Console.WriteLine("\n");
    Console.WriteLine("Bebidas: \n");
    Console.WriteLine($"{bebidas_1.nome}");
    Console.WriteLine($"{bebidas_2.nome}");
    Console.WriteLine($"{bebidas_3.nome}");
    Console.WriteLine($"{bebidas_4.nome}");
}
```

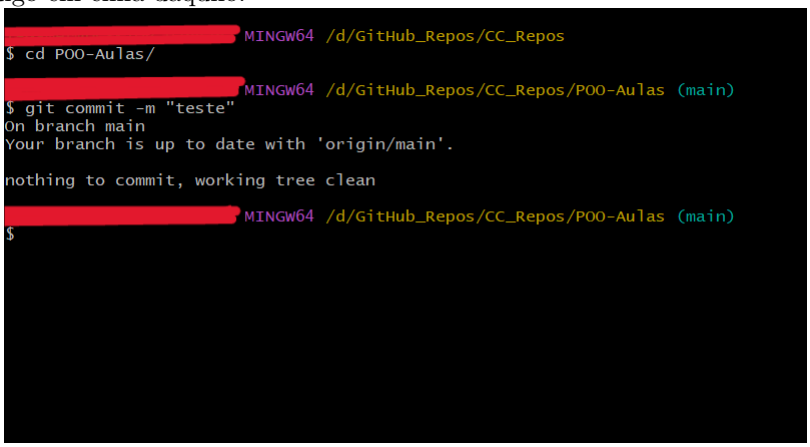
Listing 4: Código em C#/C

Código no GitHub.

Nesse exemplo perguntamos primeiro se o usuário vai querer alguma bebida e só mostramos as opções de bebida caso ele tenha afirmado que gostaria de ver todo o cardapio de bebidas.

6 Dependency Inversion Principle

Por fim, o último princípio dentro do SOLID é o da Inversão de Dependência, dizendo basicamente que as classes, e o código como um todo, não deveria depender de funcionamentos e entendimentos muito técnicos, mas sim, passar por uma abstração dos conceitos mais usados para facilitar a construção de código em cima daquilo.



Um exemplo seria o uso de Git para versionamento de código, fazendo com que os conceitos mais técnicos para esse versionamento passem pela abstração de uma aplicação como a do Git ou até mesmo do GitHub.

```
int remover_produto(int id_inicial, int id_final)
{
```

```
3       char *comando_delete = "DELETE FROM produtos WHERE id >= @id_inicial AND id <=
        @id_final;";
4
5       int retorno_delete = sqlite3_prepare_v2( banco_dados, comando_delete, -1, &
        handler_sql, 0 );
6
7       if (retorno_delete != SQLITE_OK)
8           return 0;
9
10      const int pos_id_inicio = sqlite3_bind_parameter_index( handler_sql, "@id_inicial"
        );
11      const int pos_id_fim = sqlite3_bind_parameter_index( handler_sql, "@id_final");
12
13      retorno_delete = sqlite3_bind_int( handler_sql, pos_id_inicio, id_inicial );
14
15      if (retorno_delete != SQLITE_OK)
16          return 0;
17
18      retorno_delete = sqlite3_bind_int( handler_sql, pos_id_fim, id_final );
19
20      if (retorno_delete != SQLITE_OK)
21          return 0;
22
23      retorno_delete = sqlite3_step( handler_sql );
24
25      if (retorno_delete == SQLITE_DONE)
26          return 1;
27      else
28          return 0;
29  }
```

Listing 5: Código em C#/C

Código em C no GitHub.

Nesse outro exemplos temos a integração do SQLite com uma aplicação de C, não precisamos entender ou fazer algum intermediário entre os dois por meio de um arquivo csv, mas apenas com uma biblioteca podemos utilizar as features do SQLite dentro de nossa aplicação em C;

7 Fontes

Geeks For Geeks.

Digital Ocean.