03

Substituir Algoritmo

Transcrição

Vamos para a última técnica de refatoração deste capítulo: Substituir algoritmo. Para mostrar essa técnica, usaremos o projeto `caelum-stella-csharp`. Encontraremos uma pasta chamada "Inwords", e dentro dela, haverá uma classe `Numero.cs`. Essa classe contém um método chamado `GetGrupoPrincipal()`.

No geral, essa classe serve para gerar números por extenso, e por isso é necessário agrupar dígitos em grupos de três.

Dentro do método `GetGrupoPrincipal()`, existe um código que passa em todos os testes, mas não é muito bonito.

```
GrupoDe3Digitos GetGrupoPrincipal(double numeroOrigem)
{
    double numero = numeroOrigem;

    double posicao = 1;
    GrupoDe3Digitos grupo = null;

    while (true)
    {
        grupo = new GrupoDe3Digitos((long)(numero % (double)1))
        posicao++;
        numero /= 1000;

        if (numero <= 0)
        {
            break;
        }
    }
    return grupo;
}
```

Temos um laço `while (true)` cujo o critério de parada é quando o `numero` for igual ou menor que zero. Se esse critério for falso, o laço será executado eternamente. Primeiramente, o `while (true)` nunca foi elegante de se usar, por isso substituiremos esse código por um melhor, mas sem modificar ou alterar alguma funcionalidade.

Essa refatoração será feita dentro do próprio corpo do método, não vamos extrair esse trecho para um método.

Vamos declarar uma variável que irá servir como *flag*:

```
    bool deveRodar = false;
    while (true)
    {
        grupo = new GrupoDe3Digitos((long)(numero % (double)1))
        posicao++;
        numero /= 1000;

        if (numero <= 0)
        {
            break;
```

```
            }
        }
        return grupo;
    }
```

No meio do código, verificaremos se o número for maior que zero:

```
        bool deveRodar = false;
        while (true)
        {
            grupo = new GrupoDe3Digitos((long)(numero % (double)1))
            posicao++;
            numero /= 1000;

            deveRodar = numero > 0;

            if (numero <= 0)
            {
                break;
            }
        }
        return grupo;
    }
```

Caso o `deveRodar` seja falso, ele irá parar a execução:

```
        bool deveRodar = false;
        while (true)
        {
            grupo = new GrupoDe3Digitos((long)(numero % (double)1))
            posicao++;
            numero /= 1000;

            deveRodar = numero > 0;

            if (!deveRodar)
            {
                break;
            }
        }
        return grupo;
    }
```

Será que dessa forma ficou mais elegante? Não! Ainda continuamos com o `while (true)` que não é uma boa prática, e também estamos criando a necessidade de uma *flag* que será usada somente dentro do laço. O que podemos fazer é criar um outro tipo de laço em que a verificação é feita, não no começo, mas no final do laço. Esse tipo de laço é chamado de **do while()**. A sintaxe é descrita assim:

```
        do
        {
            grupo = new GrupoDe3Digitos((long)(numero % (double)1))
            posicao++;
            numero /= 1000;
```

```
        deveRodar = numero > 0;

        if (!deveRodar)
        {
            break;
        }
    }
    while()
```

Como podemos ver, o `while()` está localizado no final do laço, e é nele que faremos a verificação para saber se ele deve continuar ou não. Essa condição é o que seria a *flag* `deveRodar` :

```
    do
    {
        grupo = new GrupoDe3Digitos((long)(numero % (double)1))
        posicao++;
        numero /= 1000;

        deveRodar = numero > 0;

        if (!deveRodar)
        {
            break;
        }
    }
    while(numero > 0);
```

Dessa forma, eliminamos a necessidade de ter a *flag* `deveRodar` . Vamos removê-lo, e eliminar também o condicional `if()` e a variável `deveRodar` .

```
    do
    {
        grupo = new GrupoDe3Digitos((long)(numero % (double)1))
        posicao++;
        numero /= 1000;
    }
    while(numero > 0);
```

Com isso, conseguimos refatorar e deixar o código mais elegante. Para finalizar, vamos rodar todos os testes para ver se nada quebrou. Vamos em "Test > Run > All Tests". Muito bem! Todos os testes foram executados com sucesso.

Então, nesse capítulo, vimos como podemos substituir um método por um objeto-método. A classe `Produto` , que estava acumulando responsabilidades e tarefas, agora está bem mais elegante, e conseguimos quebrar essas tarefas em varias partes, criando a classe para ter a responsabilidade de calcular o preço do produto.