02

Herança e o Liskov Substitutive Principle

Olá!

Neste último capítulo do curso falaremos sobre **herança**.

Herança, lá no começo das linguagens orientadas a objetos, era o grande marketing delas por que isso permitia reaproveitar o código de maneira fácil, bastava colocar os ":" ou o "extends" no Java e seu código era reutilizado sozinho, fazendo com que a classe filha ganhasse os métodos da classe mãe e etc. Isso foi um grande fator de "venda" da orientação a objetos. Porém, hoje, a indústria percebeu que não é tão fácil assim usar herança.

Começaremos, como sempre, com um exemplo.

Dá uma olhada nessa classe `ContaComum`:

```
class ContaComum
   {
       public double Saldo { get; private set; }

       public ContaComum()
       {
           this.Saldo = 0;
       }

       public virtual void Deposita(double valor)
       {
           this.Saldo += valor;
       }

       public void Saca(double valor)
       {
           if (valor <= this.Saldo)
           {
               this.Saldo -= valor;
           }
           else
           {
               throw new ArgumentException();
           }
       }

       public virtual void somaInvestimento()
       {
           this.Saldo += this.Saldo * 0.01;
       }
   }
```

Ela tenta representar, de maneira simplificada, uma conta de um banco. Temos ali um método `deposita`, um `Saldo` e o método `somaInvestimento`. Esse método `rende` faz uma conta qualquer com o saldo, multiplica por 1.1 e dá o rendimento para essa conta.

Imagine agora que no meu banco apareceu `ContaDeEstudante` :

```
class ContaEstudante : ContaComum
   {

   public override void SomaInvestimento() {
       throw new Exception("Conta de estudante não rende");
   }
   }
```

A `ContaDeEstudante` é parecida com uma conta comum, mas ela não rende.

Então, a classe `ContaDeEstudante` que *herda* de `ContaComum` , pede que nós sobrescrevamos o método `SomaInvestimento` , e lancemos uma exceção. certo?

Normal!

Eu sobrescrevi o método na classe filha e mudei o comportamento para alguma coisa diferente.

É assim que fazemos, certo?

Ótimo.

Mas observe que esse código, apesar de pequeno, possui um grande problema.

E que problema que é esse?

Observe esse método `main` simples:

```
public class ProcessadorDeInvestimentos {

    public static void main(string[] args) {

        foreach (ContaComum conta in contasDoBanco()) {
            conta.SomaInvestimento();

            Console.WriteLine("Novo Saldo:"+ conta.Saldo);
        }
    }
}
```

Eu tenho uma lista de contas do banco e estou tratando-as como uma conta comum, afinal, eu posso, vou usar polimorfismo para tratar todas elas a partir da abstração maior - que é a `ContaComum` - e estou fazendo um *loop*. Depois, eu fiz `conta.SomaInvestimento` .

Se eu rodar este programa, eu não sei o que vai acontecer, porque se o banco só tiver contas comum, tudo vai funcionar.

Mas, se o banco tiver uma conta de estudante, esse código não vai funcionar, porque o método `rende` lançará uma exceção.

Então, veja só, imagine que meu sistema já existe e que eu só tenha contas comuns, tendo vários desses loops espalhados pelo meu sistema, e vários deles invocam o método `rende`. Agora, imagine que eu criei a classe filha `ContaDeEstudante` e sobrescrevi um método com `throw new` e passei uma nova exceção.

Vamos códigos que já funcionavam vão parar de funcionar! Isso não é uma boa ideia!

Veja só, eu criei uma classe filha e essa classe filha quebra o comportamento das outras classes do sistema que usavam antes a abstração `ContaComum`.

Observe que na hora de usar herança isso não é tão fácil assim.

Vou dar um outro exemplo pra você.

Veja esse exemplo: eu tenho a classe `Retangulo` e uma classe filha dela, que é a classe `Quadrado`. Esse é um exemplo bastante comum, muitos professores usam esse exemplo na primeira aula de herança.

A classe `Retangulo` é simples, ela tem 2 lados iguais - porque todo retângulo tem 2 lados iguais - e possui qualquer método, qualquer comportamento dessa classe retângulo.

O que ele faz depois?

Ele quer criar a classe `Quadrado`, e ele sabe que o quadrado é um tipo especial de retângulo, certo? O quadrado é aquele retângulo que possui todos os lados iguais. Então, ele faz a classe `Quadrado` que *herda* `Retangulo`. Faz uma classe ser filha da outra.

Isso não é uma boa ideia.

O que estou fazendo aqui é exatamente a mesma coisa que eu fiz lá na classe `ContaComum` com a classe `ContaDeEstudante`.

Existe um princípio da Orientação a objetos cuja sigla é **LSP**.

No SOLID é o L, certo?

A sigla LSP significa *Liskov Subtitutive Principle* ou Princípio de Substituição de Liskov.

E qual é a ideia?

A pesquisadora, na época – esse artigo é antigo, da década de 80 –, percebeu que para usar herança, é preciso pensar muito bem nas pré-condições da classe e nas pós-condições da mesma. Pense que todo método, quando recebe parâmetros, possui pré-condições: “Ah, o inteiro que eu estou recebendo no método `Saldo`, no método X, no método `FazUmDepósito`, sei lá, esse valor pode ser qualquer coisa que seja positiva, tem que ser maior que 0. No `deposita`, a mesma coisa. O `double` que eu recebo tem que ser maior que 0. O retorno do método getter, não sei, `Saldo`, é sempre um valor positivo, nunca é menor que 0, e `Saldo` nunca retorna uma exception". Então, todo método tem as suas pré-condições e as suas pós-condições - Como ela vai receber os dados de entrada, quais são as *constraints*, as restrições dos dados de entrada e quais são as restrições do dado que ela gera como uma saída.

O exemplo do retângulo e do quadrado deixa isso bem claro ao destacar que as pré-condições de um e de outro são diferentes.

Veja só: no retângulo os lados não possuem pré-condições, eles podem ser quaisquer números, inclusive diferentes. No quadrado, a pré-condição é diferente: todos os lados devem ser iguais.

O Princípio de Liskov mostra que quando você tem uma classe filha, ela nunca pode apertar as pré-condições. Você nunca pode criar uma pré-condição que seja mais restrita do que da classe mãe. A classe filha só pode afrouxar a pré-condição.

Pense no seguinte caso: eu tenho a classe mãe e essa classe tem um método que pode receber inteiros de 1 a 100. Digamos que a classe filha mude isso, ou seja, ela só deixa receber inteiros de 1 a 50. Observe que 1 a 50 é mais restritivo do que 1 a 100; dessa forma, a classe filha apertou a pré-condição e isso pode complicar as classes clientes, porque as classes clientes sabem que a classe mãe pode receber de 1 a 100, então, elas vão passar de 1 a 100 sem pensar muito. E observe que, se a classe filha apertou essa restrição, ela terá um comportamento que é inesperado.

Com a pós-condição é a mesma coisa.

A classe filha nunca pode afrouxar a pós-condição – o contrário da pré.

Consegue ver?

A pós-condição nunca pode afrouxar, por que, voltando ao exemplo, eu tenho um método que devolve um inteiro e esse inteiro é de 1 a 100. Só que agora, a classe filha sobrescreve o método e passa a retornar de 1 a 200. Isso pode quebrar as classes clientes! Por que imagine que o cliente está esperando um retorno de 1 a 100 e ele trata isso, que é o que ele espera, só que nesse caso, a classe filha pode devolver mais valores. Ela devolve um 150, por exemplo, que a classe cliente não estava esperando. Se ela fizer isso, o código passa a não vai funcionar de maneira ideal.

Então, para usar herança de maneira decente, é preciso pensar muito nas pré-condições e nas pós-condições.

Eu nunca posso apertar uma pré-condição e eu nunca posso afrouxar uma pós-condição.

Veja que isso é bastante complicado, bastante difícil de analisar na hora que você está desenvolvendo - usar herança de maneira decente, criando classes filhas que nunca vão quebrar quando elas entrarem numa referência que recebe a classe mãe, a abstração.

Para programar desse jeito, devemos pensar nessas coisas. Isso não é trivial. É bem complicado, na verdade.

É por essa razão que muita gente fala: “Olha, em vez de usar herança, favoreça a composição. Faça sua classe depender de outra classe, faça sua outra classe depender, talvez, dessa mesma classe, mas fuja de reaproveitar código por herança."

Justamente por isso.

Por outro lado, com a composição você não tem esse problema, porque a classe `Retangulo` e a classe `Quadrado` são classes diferentes. Desa forma, elas possuem pré-condições que são diferentes e isso não é um problema. A partir do momento em que você usa herança, a classe filha deve conhecer as pré e pós-condições da mãe. Isso dificulta mais. Ao favorecer a composição, minimizamos esse problema.

Então, vamos lá!

Eu tenho a minha `ContaDeEstudante`, que é filha de `ContaComum`, e o método `rende` lança uma exceção. A `ContaComum`, bem parecida com aquela que eu mostrei no slide onde o método `rende` faz uma conta qualquer, quebra por que ela não lança nenhuma exceção, mas a classe filha passou a lançar.

Não faz sentido.

Esse é um problema de herança, vamos refatorar isso.

Nesse caso em particular, eu não vou refatorar e melhorar a herança que está feita. Eu vou fazer uso de composição, por que nesse contexto em particular, `ContaComum` tem de semelhante com a `ContaDeEstudante` o saldo, nada além disso. Ambas possuem operações que acontecem em cima do saldo.

Vamos melhorar isso com uma classe que eu vou chamar de `ManipuladorDeSaldo`. Essa classe vai ter - e eu vou até aproveitar e copiar o código da classe `ContaComum` - um saldo. Vou começar como `private` - veja só que ele estava como `protected` pra filha enxergar:

```
public class ManipuladorDeSaldo {

    public double Saldo {get;private set;};
}
```

Olha só! Estamos afrouxando um pouquinho o encapsulamento - por que se você parar pra pensar, existe encapsulamento até quando você pensa em herança -, por que nós vamos esconder a classe usando `private`.

Ao colocar `protected`, você está deixando a filha mexer na classe mãe, no atributo que a mãe declarou. Você não sabe se as mudanças que a mãe fizer vão bater com as mudanças que a filha vai fazer também. Dessa forma, é complicado usar o `protected` por que ele afrouxa o encapsulamento, deixando a filha mexer na classe mãe - às vezes sem poder.

Perigoso!

Então vou jogar fora daqui o `protected double saldo`. O método `deposita`, `saca` e `Saldo` serão levados para o `ManipuladorDeSaldo`.

```
public class ManipuladorDeSaldo  {

    public double Saldo {get;private set;}

    public void Deposita(double valor) {
        this.Saldo += valor;
     }

    public void Saca(double valor) {
        if (valor <= this.Saldo)  {
            this.Saldo -= valor;
        } else {
            throw new ArgumentException();
        }
    }

}
```

Legal. Tudo compilando.

Vou criar também o método `SomaInvestimento`, passando um `double taxa`:

```
public void SomaInvestimento(double taxa)  {
    this.Saldo *= taxa;
```

```
}
```

Ótimo.

A `ContaComum` não possui mais o saldo 0. Ela vai fazer `private ManipuladorDeSaldo` , e vou chamar de `manipulador` . No construtor, ela vai dar o `new` nessa classe:

```
public class ContaComum  {

    private     ManipuladorDeSaldo manipulador
    public ContaComum()  {
        this.manipulador = new ManipuladorDeSaldo();
}
```

Legal.

O método `SomaInvestimento` não vai mais implementar, ele vai passar pro manipulador - `this.manipulador.SomaInvestimento(1.1)` , por exemplo, 10%:

```
    public void SomaInvestimento()  {
        this.manipulador.SomaInvestimento(1.1);
    }
```

"Ah, mas a classe `ContaComum` tem um saque!"

Então, vamos lá:

```
    public void Saca(double valor)  {
        manipulador.Saca(valor);
    }
```

O que ele vai fazer?

O `manipulador.saca` vai passar um valor.

Aqui eu estou só repassando o método de uma classe pra outra! Não tem problema! Se amanhã aparecer uma outra regra de negócio, você pode por aqui:

```
    public void Saca(double valor)  {
        manipulador.Saca(valor - 100);
    }
```

O saque desconta 100 reais a menos e etc. Estou repassando para o manipulador. Não tem problema! Às vezes, é assim que acontece quando fazemos composição.

Ele tem `deposita` também:

```
    public void Deposita(double valor)  {
        manipulador.deposita(valor);
    }
```

Ótimo.

Eu poderia fazer o mesmo com a `ContaDeEstudante`.

Vou tirar o `Deposita`, por que agora eu vou ter que começar a ter um `ManipuladorDeSaldo` também. Vou chamar de `m`, escrever um pouco menos - `m` é um péssimo nome de variável -, de forma que `m` é um `new ManipuladorDeSaldo`. No `deposita`, eu faço `m.Deposita` e passo o valor:

```
public class ContaDeEstudante : ContaComum  {
    private ManipuladorDeSaldo m;
    public int Milhas {get;private set;}

    public ContaDeEstudante()  {
        m = new ManipuladorDeSaldo();
    }
    public void Deposita(double valor)  {
        m.Deposita(valor);
        this.Milhas += (int)valor;
    }
}
```

Não precisamos mais do `SomaInvestimento`. Antes eu tinha essa exceção sendo lançada por que eu tinha uma classe mãe, a `ContaComum`. Agora, essa herança não existe mais. Então, não precisamos do método `SomaInvestimento` na classe `ContaDeEstudante`.

Vamos lá.

Veja só o que eu fiz: a `ContaComum` depende do `ManipuladorDeSaldo`. E ele repassa as chamadas, os métodos aqui - o `saca`, o `deposita`, o `rende` -, para o manipulador.

Nesse código em particular, parece que eu só estou repassando chamadas de uma classe pra outra. Mas, em um mundo mais complicado, eu poderia ter regras de negócio, certo? Regra de negócio dos saques, específica da conta comum. É a mesma coisa com a `ContaDeEstudante`. No `deposita`, veja só, eu tenho uma regra em particular. Porque eu faço um depósito no manipulador e somo as milhas - por que conta de estudante tem milhas, por exemplo.

E o `ManipuladorDeSaldo` abstraiu o que as duas classes tinham em comum, que era manipular um saldo, ok?

Então, é assim que eu refatorei, tirei a herança e coloquei composição.

Legal.

Na refatoração eu fiz uso de composição, fugindo um pouco da herança.

Relembrando tudo o que discutimos nessa aula: Princípio de Liskov, toda classe filha deve pensar nas pré-condições e pós-condições da mãe e ela nunca pode quebrar. Na pré-condição, ela nunca pode apertar. Na pós-condição, ela nunca

pode afrouxar. Se não, as referências que apontam pra classe mãe, quando receberem uma classe filha, não vão funcionar da maneira esperada.

Herança e composição, duas maneiras de reutilizar código. Você tem que optar. As pessoas falam muito: “Olha, nunca use herança!”, mas eu não sou tão extremista assim. Herança é legal e faz sentido em muitos casos, mas o ponto é que ela é mais difícil de ser usada do que a composição.

Então, não descarte herança, mas favoreça a composição.

Essa é a principal lição dessa aula: perceber a diferença entre essas duas maneiras de reaproveitar código - quando usar herança de maneira decente e como usar composição.

Obrigado.