02

Herança e o Liskov Substitutive Principle

Olá! Neste capítulo, eu vou falar um pouquinho pra vocês sobre **herança**. Herança, lá no começo das linguagens orientadas a objeto, era o grande marketing delas, porque "nossa senhora, você vai conseguir reaproveitar código de maneira fácil, basta colocar a palavrinha "extends" aqui, ou dois pontos lá no C#, não sei que das quantas, e seu código vai ser reutilizado sozinho, a classe filho vai ganhar os métodos da classe pai e tudo mais." E isso foi um grande fator de "venda" da orientação a objetos, mas hoje, a indústria percebeu que não é lá tão fácil assim usar herança.

Eu vou mostrar pra vocês, como sempre, com exemplo. Dá uma olhada aqui nessa classe `ContaComum`:

```
public class ContaComum {

    protected double saldo;

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

    public void deposita(double valor) {
        this.saldo += valor;
    }

    public double getSaldo() {
        return saldo;
    }

    public void rende()      {
        this.saldo*= 1.1;
    }
}
```

Ela tenta representar, obviamente, de maneira simplificada, uma conta de um banco. Dá uma olhada. Eu tenho ali um método `deposita`, um método `getSaldo` e o método `rende`. Esse método `rende`, ele faz uma conta qualquer com o saldo. Multiplica por 1.1 e dá o rendimento pra essa conta.

Agora imagina que no meu banco apareceu agora `ContaDeEstudante`

```
public class ContaDeEstudante extends ContaComum {

    public void rende()  {
        throw new ContaNaoRendeException();
    }
}
```

A `ContaDeEstudante` é igualzinha a uma conta comum, com a diferença de que ela não rende. Então, dá uma olhada. Classe `ContaDeEstudante` que *extends* de `ContaComum`, e aí, sobrescrevi o método `rende`, e lancei uma exceção: `throw`

`new ContaQueNaoRendeException` . Certo? Normal. Sobrescrevi o método na classe filho, mudei o comportamento pra alguma coisa diferente.

É assim que fazemos, certo? Ótimo. Só que esse código, apesar de pequeno, tem um problemão. Que problema que é esse? Dá uma olhada aqui nesse método `main` simples da vida:

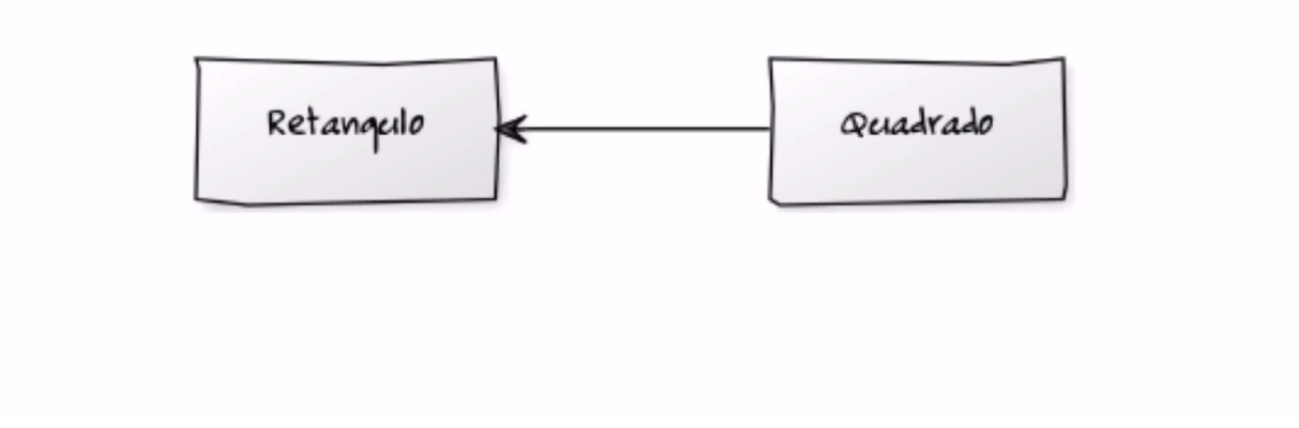
```
public class ProcessadorDeInvestimentos {

    public static void main(String[] args) {

        for (ContaComum conta : contasDoBanco()) {
            conta.rende();

            System.out.println("Novo Saldo:");
            System.out.println(conta.getSaldo());
        }
    }
}
```

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

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, opa! Esse código não vai funcionar, porque o método `rende` vai lançar uma exceção. Então veja só, imagine que meu sistema já existe, e eu só tenha contas comuns, e eu tenha um monte desses loops espalhados pelo meu sistema. E vários deles invocam esse método `rende` . Agora imagina que eu criei a classe filho `ContaDeEstudante` e sobrescrevi um método com esse `throw new` e passei uma nova exceção.

Puxa, um monte de código que já funcionava vai parar de funcionar. Isso não é uma boa ideia. Veja só, eu criei uma classe filho e essa classe filho quebra o comportamento das outras classes do sistema que usavam antes a abstração `ContaComum` .

Veja só que na hora de usar herança, isso não é tão fácil assim. Vou dar um outro exemplo pra vocês. Veja só esse exemplo aqui:



Eu tenho a classe `Retangulo` e aí eu tenho uma classe filho dela, que é a classe `Quadrado` . Esse é um exemplo bastante comum, muito professor usa esse exemplo na primeira aula de herança, e coisa e tal.

A classe `Retangulo` é simples, ela tem 2 lados, porque todo retângulo tem 2 lados, e tem lá qualquer método, qualquer comportamento dessa classe retângulo. O que ele faz? Ele quer criar a classe `Quadrado`. E ele sabe que o quadrado é um tipo especial de retângulo, certo? O quadrado é aquele retângulo cujos dois lados são iguais. Então, ele vai lá e faz classe `Quadrado` que *extends* `Retangulo`. Faz uma classe ser filho 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`. Tem um princípio da Orientação a objetos, pessoal, cuja sigla é **LSP**. No SOLID é o L, certo? A sigla LSP significa *Liskov Subtitutive Principle*, Princípio de Substituição de Liskov.

Qual é a ideia? A pesquisadora, na época – e esse artigo é antigo, é de por volta de 85, 86, 87 – percebeu que, pra você usar herança, você tem que pensar muito bem nas pré-condições da sua classe e nas pós-condições da sua classe. Se você pensar, todo método, quando recebe parâmetros, você tem pré-condições: "Ah, o inteiro que eu estou recebendo no método `Saldo`, no método X, no método `fazUmDepósito`, sei lá o quê, 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). O retorno do método, não sei, `getSaldo`, é sempre um valor positivo, nunca é menor que 0, e `getSaldo` nunca retorna uma exception.

Então, todo método tem lá as suas pré-condições e as suas pós-condições. Como que 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.

Esse exemplo aqui do retângulo e do quadrado deixa bem claro isso, que as pré-condições de um e de outro são diferentes. Veja só, no retângulo, os lados não têm pré-condições, eles podem ser quaisquer números, inclusive diferentes. No quadrado, a pré-condição é diferente: os dois lados têm que ser iguais. E ela mostra – o Princípio de Liskov mostra – que, quando você tem uma classe filho, a classe filho nunca pode apertar as pré-condições. Você nunca pode criar uma pré-condição que seja mais restrita do que da classe pai.

A classe filho só pode afrouxar a pré-condição. Pensa no caso onde eu tenho a classe pai, e a classe pai tem um método que pode receber inteiros de 1 a 100. Aí a classe filho muda isso, ela só deixa receber inteiros de 1 a 50. Veja só que 1 a 50 é mais restritivo do que 1 a 100. Eu apertei a pré-condição. Isso pode complicar as classes clientes, porque as classes clientes sabem que a classe pai pode receber de 1 a 100. Então elas vão passar de 1 a 100 sem pensar muito.

Só que, se a classe filho apertou essa restrição, quer dizer que a classe filho vai ter um comportamento que é inesperado.

Com a pós-condição, é a mesma coisa. A classe filho, ela nunca pode afrouxar a pós-condição – o contrário da pré. Consegue ver? A pós-condição, ela nunca pode afrouxar. Porque, pensa o seguinte, eu tenho um método que devolve um inteiro. E esse inteiro é de 1 a 100. Aí a classe filho sobrescreve o método, e passa a retornar de 1 a 200. Isso pode quebrar as classes clientes. Porque imagina só, o cliente está esperando um retorno de 1 a 100, e ele trata isso, que é o que ele espera. E a classe filho vem, e pode devolver mais valores. Ela devolve um 150, que a classe cliente não estava esperando. O código não vai funcionar de maneira ideal.

Então, veja só que, para usar herança de maneira decente, eu tenho que pensar muito nas pré-condições e muito nas pós-condições. Eu nunca posso apertar uma pré-condição. E eu nunca posso afrouxar uma pós-condição. Veja que isso é bastante complicado, bem 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 pai, a abstração.

Pra você programar desse jeito, você tem que pensar nessas coisas. Isso não é lá tão trivial. É bem complicado, na verdade.

É até por isso 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. Certo? Com a composição, você não tem muito esse problema, porque a classe `Retangulo` e a classe `Quadrado` são classes diferentes. E aí elas têm pré-condições que são diferentes, e não tem problema. A partir do momento em que você usa herança, o filho tem que conhecer as pré e pós-condições do pai. Isso dificulta mais. Quando eu favoreço a composição, eu tenho menos esse problema.

Então, vamos lá. Eu tenho a minha `ContaDeEstudante`, que é filho 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. Uma conta qualquer aqui, uma exceção aqui, perceba que um quebrou a pré-condição do outro, porque o método pai não lança nenhuma exceção, o método filho passou a lançar. Não faz sentido. Problema de herança, vamos refatorar isso.

Nesse caso aqui, em particular, eu não vou refatorar e melhorar a herança que está feita. Mas eu vou fazer uso de composição. Porque, veja só, nesse contexto em particular, `ContaComum` tem de semelhante com `ContaDeEstudante` com saldo. Mais nada do que isso. Ambas têm operações que acontecem em cima do saldo.

O que eu vou fazer aqui é criar uma classe, que eu vou chamar de `ManipuladorDeSaldo`. Essa classe vai ter – e eu vou até aproveitar aqui e copiar o código da classe `ContaComum` ? vai ter um saldo. Vou começar como `private`, veja só que ele até estava como `protected` pro filho enxergar:

```
public class ManipuladorDeSaldo {

    private double saldo;
}
```

Olha só, a gente já está afrouxando um pouquinho o encapsulamento, né, porque se você parar pra pensar, existe encapsulamento até quando você pensa em herança. Porque você tem que esconder da sua classe usando `private`. A hora que você coloca `protected`, você está deixando o filho mexer na classe pai, no atributo que o pai declarou. Você não sabe se as mudanças que o pai fizer vão bater com as mudanças que o filho vai fazer também. Então, é complicado usar o `protected`, você está afrouxando o encapsulamento, está deixando o filho mexer na classe pai, às vezes sem poder. Perigoso!

Então vou jogar fora daqui (o `protected double saldo`). O método `deposita`, `saca`, e `getSaldo`, eu vou levar pra lá, para o `ManipuladorDeSaldo`.

```
public class ManipuladorDeSaldo  {

    private double saldo;

    public void deposita(double valor) {
        this.saldo += valor;
     }

    public void saca(double valor) {
        if (valor <= this.saldo)  {
            this.saldo -= valor;
        } else {
```

```
                throw new IllegalArgumentException();
            }
        }

        public double getSaldo()  {
            return saldo;
        }
    }
```

Veio para cá. Tudo compilando. Aqui também eu vou criar o método `rende`. Vou até passar aqui um `double taxa`, que vai fazer

```
public void rende(double taxa)  {
    this.saldo *= taxa;
}
```

Ótimo. O `ContaComum`, então, não tem mais esse negócio de saldo 0. Ele vai fazer `private ManipuladorDeSaldo`, e vou chamar de `manipulador`. No construtor, ele vai dar o `new` nessa classe:

```
public class ContaComum  {

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

Legal. O método `rende`, ele não vai mais implementar na unha. Ele vai passar pro manipulador. `this.manipulador.rende(1.1)`, por exemplo, 10%.

```
    public void rende()  {
        this.manipulador.rende(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? `manipulador.saca` passando um valor.

Ah, mas 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, coisa e tal. Estou repassando para o manipulador. Não tem problema, às vezes é assim que acontece quando fazemos composição.

Ah, ele tem `deposita` também:

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

Ótimo. A `ContaDeEstudante`, eu poderia fazer a mesma coisa. Deixa eu tirar o `deposita`, o `getMilhas`. Aliás até posso deixar, certo, vou deixar o milhas. Vou só tirar o `super.deposita` aqui, porque aqui 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. `m` é um `new ManipuladorDeSaldo`. No `deposita` aqui, eu faço `m.deposita` e eu passo o valor.

```
public class ContaDeEstudante extends ContaComum  {
    private ManipuladorDeSaldo m;
    private int milhas;

    public ContaDeEstudante()  {
        m = new ManipuladorDeSaldo()
    }
    public void deposita(double valor)  {
        m.deposita(valor);
        this.milhas += (int)valor;
    }
    public int getMilhas()  {
        return.milhas;
    }
}
```

E o método `rende`, nem preciso dele. Antes eu tinha essa exceção sendo lançada, porque eu tinha uma classe pai, que eu vou até tirar daqui, `extends ContaComum`. Não tem mais essa herança acontecendo. Então, não preciso mais do método `rende` na classe `ContaDeEstudante`.

Veja só, o que eu fiz? Eu criei – esse `ProcessadorDeInvestimentos` para de compilar, porque não tenho o `getSaldo`. Não implementei aqui, mas poderia implementar, certo:

```
    public double getSaldo()  {
        return manipulador.getSaldo();
    }
```

Então, vamos lá. Veja só o que eu fiz. `ContaComum` depende de `ManipuladorDeSaldo`. E ele repassa as chamadas, os métodos aqui, o `saca`, o `deposita`, o `rende`, repassa as chamadas pro manipulador.

Nesse código em particular, parece que eu só estou repassando chamadinha de uma classe pra outra. Mas num mundo mais complicado, eu poderia ter regras de negócio aqui, certo? Regra de negócio dos saques, específicos da conta comum aqui.

A mesma coisa na `ContaDeEstudante`. Aqui no `deposita`, veja só, eu tenho uma regra em particular. Porque eu faço um depósito no manipulador, e aí eu somo as milhas, porque conta de estudante tem milhas, por exemplo.

E o `ManipuladorDeSaldo` abstraiu ali o que as duas classes tinham em comum, que era manipular um saldo. Está legal? Então é assim que eu refatorei e tirei a herança, e coloquei composição.

Legal. Na refatoração, vocês viram que eu fiz uso ali de composição, fugi um pouco da herança. Tá certo? Relembra tudo o que eu falei nessa aula: Princípio de Liskov, toda classe filho tem que pensar nas pré-condições e pós-condições do pai, e ela nunca pode quebrar. Na pré-condição, ela nunca pode apertar. E na pós-condição, ela nunca pode afrouxar. Se não, as referências que apontam pra classe pai, quando receberem uma classe filho, não vão funcionar da maneira esperada.

Herança, composição, duas maneiras de reutilizar código. Você tem que optar. As pessoas falam muito "Olha, nunca use herança!". Eu não sou tão extremista assim. Herança é legal e faz sentido em muitos casos.

Mas o ponto é só que ela é mais difícil de ser usada do que composição. Então, não descarte herança, mas favoreça a composição. Acho que essa é a principal lição dessa aula: é perceber a diferença entre essas duas maneiras de reaproveitar código. Quando usar herança de maneira decente, e como usar composição. Nessa aula é isso! Obrigado!.