08

Para saber mais: Cuidado com o Modelo Anêmico

Durante o aprendizado dos Getters e Setters é normal pensar sempre na necessidade deles para alterar algum estado dos nossos objetos.

Mas o uso dessa prática nem sempre é a mais indicada e expressa a realidade.

Observe a classe `Conta` representada abaixo que usa apenas getter e setters como métodos:

```
class Conta{
    private String titular;
    private double saldo;

    public void setTitular(String titular){
        this.titular = titular;
    }

    public String getTitular(){
        return titular;
    }

    public void setSaldo(double saldo){
        this.saldo = saldo;
    }

    public double getSaldo(){
        return saldo;
    }

}
```

Continuamos usando atributos privados e nosso modelo parece seguir perfeitamente a proposta do encapsulamento onde a própria classe gerencia o seus estados(atributos). Uma utilização clássica dessa Conta nos levaria ao seguinte cenário:

```
Conta conta = new Conta();
conta.setTitular("Fábio")
conta.setSaldo(100.0);
```

Tudo parece perfeito, agora imagine que seja necessário sacar 50.0 dessa conta. Essa operação vai exigir que o saldo seja suficiente. Uma simples verificação como a seguir asseguraria que o saldo não tenha ficado negativo. Nesse nosso exemplo não há limite além do saldo :)

```
Conta conta = new Conta();
conta.setTitular("Fábio")
conta.setSaldo(100.0);

double valorSaque = 50.0
```

```
if(conta.getSaldo() >= valorSaque){
    double novoSaldo = conta.getSaldo() - valorSaque;
    conta.setSaldo(novoSaldo)
}
```

Funcionou! Mas um problema é que essa lógica de restringir o saque à quantidade de saldo vai ter que ser refeita toda vez que for necessária uma operação de saque na nossa conta. Além do problema de duplicações desse trecho, um problemão para encapsulamento é que quem está de fato controlando as regras do saldo da conta é quem está usando a Conta. Em outras palavras nada impede que alguém implemente um limite extra para isso e tenha uma regra completamente diferente dos demais objetos do tipo Conta:

```
Conta conta = new Conta();
conta.setTitular("Fábio");
conta.setSaldo(100.0);

double valorSaque = 50.0;

if(conta.getSaldo() + 1000.0 >= valorSaque){
    double novoSaldo = conta.getSaldo() - valorSaque;
    conta.setSaldo(novoSaldo)
}
```

Quando construímos classes que se limitam a ter atributos privados com os setters e getters normalmente dizemos que são classes que só carregam valor, por isso são comumente chamados de classes fantoches ou `Value Objects`.

Uma classe fantoche é a que não possui responsabilidade alguma, a não ser carregar um punhado de atributos! Definitivamente isso não é a Orientação a Objetos! Esse modelo embora usado em alguns momentos não deve ser prática comum ao desenvolver o domínio do nosso projeto com risco de se cair no `Modelo Anêmico` que é exatamente o que a `Conta` hoje é. Uma classe onde os dados e comportamentos/lógicas não estão juntos.

Voltando ao nosso exemplo da Conta, percebe-se que no mundo real as operações poderiam ser representadas com métodos como `saca( )` e `deposita( )` em vez de só termos `setSaldo( )`:

```
class Conta{
    private String titular;
    private double saldo;

    public void setTitular(String titular){
        this.titular = titular;
    }

    public String getTitular(){
        return titular;
    }

    public void saca(double valor){
        if(valor > 0 && saldo >= valor){
            saldo -= valor;
        }
    }

    public void deposita(double valor){
        if(valor>0){
```

```
            saldo += valor;
        }
    }

    public double getSaldo(){
        return saldo;
    }

}
```

Perceba que as lógicas de saque e depósito estão representados dentro da classe e além disso nosso `setSaldo()` deixa de fazer sentido para o usuário da Conta. A maneira de interagir com o saldo é sempre via uma das operações anteriores:

```
Conta conta = new Conta();
conta.setTitular("Fábio");
conta.deposita(100.0);

double valorSaque = 50.0;
conta.saca(valorSaque);

double valorDeposito = 70.0;
conta.deposita(valorDeposito)
```

Muito melhor não é mesmo? Nada de duplicações de código por aí e muito menos outras classes controlando o estado da nossa Conta como tínhamos anteriormente.

Conclusão

Setters e Getters devem ser usados com cautela e nem todos os atributos privados precisam ter expostos esses dois métodos com riscos de cairmos em um `modelo anêmico` que tem os seus comportamentos controlados por outras classes.