## Cuidado com o modelo anêmico

Um dos conceitos fundamentais da orientação a objetos é o de que você não deve expor seus detalhes de implementação. Encapsulando a implementação, podemos trocá-la com facilidade já que não existe outro código dependendo desses detalhes e o usuário só pode acessar seu objeto através do contrato definido pela interface pública do mesmo [6]. Costumeiramente aprendemos que o primeiro passo nessa direção é declarar todos seus atributos como `private`:

```
public class Conta {
    private double limite;
    private double saldo;
}
```

Como então acessar esses atributos? Um programador que ainda está aprendendo vai rapidamente cair em algum tutorial sugerindo a criação de *getters* e *setters* para poder acessar e modificar esses atributos, dado que agora não é possível trabalhar diretamente com eles:

```
public class Conta {
    private double limite;
    private double saldo;

    public double getSaldo() {
        return saldo;
    }

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

    public double getLimite() {
        return limite;
    }

    public void setLimite(double limite) {
        this.limite = limite;
    }
}
```

Qual é o sentido desse código? Para que esse `setSaldo`? e esse `setLimite`? Indo mais longe, e o `getLimite`? Alguém vai inovcar esse métodos? Quando vamos ao banco nós efetuamos operações de *setSaldo* em nossa conta? Ou efetuamos depósitos e saques?

**Nunca crie um *getter* ou *setter* sem sentir uma necessidade real por eles**. Assim como precisamos que essa necessidade seja clara para criar qualquer outro método que colocamos em uma classe. Particularmente, os *getters* e *setters* são campeões quando falamos em métodos que acabam nunca sendo invocados, além de que grande parte dos que existem e são utlizados poderiam ser substituídos por métodos de negócio [29].

Criando classes dessa forma, isto é, adicionando *getters* e *setters* sem ser criterioso, códigos como `conta.setSaldo(conta.getSaldo() + 100)` estarão espalhados por toda a aplicação. Se for preciso, por exemplo, que uma taxa seja debitada toda vez que um depósito é realizado, será necessário percorrer todo o código e modificar todas essas invocações. Um *search/replace* ocorreria aqui: péssimo sinal. Podemos tentar contornar isso e pensar em criar uma classe responsável por esta lógica:

```
public class Banco {
    public void deposita(Conta conta, double valor) {
        conta.setSaldo(conta.getSaldo() + valor);
    }
}
```

Esse tipo de classe tem uma característica bem procedural, fortemente sinalizada pela ausência de atributos e excesso do uso de métodos estáticos (`deposita` poderia aqui ser estático). Algumas pessoas classificam essa classe como o pattern *Business Object*. Da forma que está, temos agora separados nossos dados na classe `Conta` e a lógica de negócio na classe `Banco`, rompendo o príncipio de manter junto comportamento e estado relacionados em uma única classe. É o que muitos chamam de **modelo anêmico** (*anemic domain model*) [17, 8].

Podemos unir a lógica de negócio aos dados de uma maneira simples, inserindo métodos na classe `Conta` e removendo os que acessam e modificam diretamente os seus atributos:

```
class Conta {
    private double saldo;
    private double limite;

    public Conta(double limite) {
        this.limite = limite;
    }

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

    public void saca(double valor) {
        if (this.saldo + this.limite >= valor) {
            this.saldo -= valor;
        } else {
          throw new IllegalArgumentException("estourou limite!");
        }
    }

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

Aqui ainda mantivémos o `getSaldo`, pois faz todo sentido ao nosso domínio. Também adicionamos algumas manipulações ao método `saca`, e poderíamos debitar algum imposto em cima de qualquer movimentação financeira no método `deposita`. **Enriqueça suas classes com métodos de negócio, para que elas não virem apenas estruturas de dados**. Para isso, cuidado ao colocar *getters* e *setters* insdiscriminadamente.

*"Mas o que colocar nas minhas entidades do Hibernate além de getters e setters?"*. Antes de tudo, verifique se você realmente precisa desses métodos. Para que um `setID` na sua chave primária se o seu framework vai utilizar reflection ou manipulação de bytecode para ler atributos privados? Não seria melhor receber o `id` pelo construtor?

Algumas vezes, os *getters* e *setters* são sim necessários e alguns patterns até mesmo precisam de uma separação de lógica de negócios dos respectivos dados [**?**]. Boas práticas como o *Test Driven Development* auxiliam bastante em não criar métodos sem necessidade.

Porém, frequentemente, entidades sem lógica de negócio, com comportamentos codificados isoladamente nos denominados *business objects* caracterizam fortemente o modelo de domínio anêmico (*anemic domain model*). É muito fácil acabar jogando a lógica de negócio que poderia estar de certa forma em nossas entidades diretamente em `Action`s do Struts, `ActionListener`s do Swing e *managed beans* do JSF. Este modelo acaba ficando com um uma forte apelo procedural, e vai diretamente na contra mão de boas práticas de orientação a objetos [6] e do *Domain-Driven Design* [15].