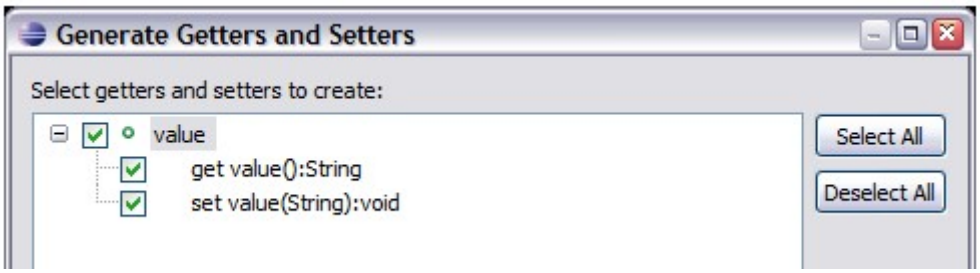
Como não aprender Java e Orientação a Objetos: getters e setters

Postado dia 14/02/2017 por Paulo Silveira em Arquitetura, Programação 145

Muitas pessoas perguntam:

> como aprender orientação a objetos?

Há várias maneiras de aprender O.O., creio que não tenha uma melhor, mas existem maneiras de **não** aprender. Esse post, criado em 2006 e atualizado em 2017, mostra como o design de classes continua seguindo algumas regras de boas práticas importantes.



Uma das práticas mais controversas que aprendemos no início do aprendizado de muitas linguagens orientadas a objeto é a geração indiscriminada de getters e setters. Os exemplos básicos de centenas de tutoriais Java estão recheados com getters e setters da pior espécie: aqueles que não fazem sentido algum. Considere:

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

Rapidamente os tutoriais explicam o `private` para o encapsulamento. Mas aí como acessar? Getters e setters nela!

```
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`? e o `getLimite`? Você vai usar esses métodos? **Nunca crie um getter ou setter sem sentir uma real necessidade por ele.**

Isso é uma regra para qualquer método, mas particularmente os getters e setters são campeões: muitos deles **nunca** serão invocados, e grande parte do restante poderia e deveria ser substituído por métodos de negócios.

Códigos do tipo `conta.setSaldo(conta.getSaldo() + 100)` se espalharão por todo seu código, o que pode ser muito nocivo: como você vai logar aumentos de saldo, se a forma de alterar saldo de uma conta está espalhada por todo seu sistema?

Tirar dinheiro gera uma situação ainda mais delicada: se precisar verificar se há saldo na conta antes de subtrair um valor do saldo, onde ficará esse `if`? Espalhado por todo o sistema em diversos pontos? O que acontecerá quando precisar alterar essa regra de negócio?

Segue então a nossa classe `Conta` reformulada de acordo com essa necessidade:

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

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

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

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

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

E nem estamos falando de test driven development! Testando a classe `Conta` rapidamente você perceberia que alguns dos getters e setters anteriores não têm uso algum e sentiria falta de alguns métodos mais voltados a lógica de negócio da sua aplicação, como o `saca` e o `deposita`.

Esse exemplo é muito trivial, mas você pode encontrar por aí muitas classes que não tem a cara de um java bean que expõem atributos como `Connection`, `Thread`, etc, sem necessidade alguma.

Existem sem dúvida práticas piores: utilização de ids para relacionar os objetos,

arrays não encapsuladas como estruturas, código fortemente baseado em comparação de Strings hardcoded (que o Guilherme Silveira sarcasticamente batizou de POS, ou programação orientada a strings), milhares de métodos estáticos, entre outros.

Todas essas práticas são comuns quando estamos começando a quebrar o paradigma procedural e confesso já ter sido um grande praticante de algumas. Você só vai utilizar bem o paradigma da orientação a objetos depois de errar muito.

O Phillip Calçado aborda esse tema, chamando essas classes de **classes fantoches** (puppets). Uma classe fantoche é a que não possui responsabilidade alguma, a não ser carregar um punhado de atributos! Onde está a orientação a objetos? Uma grande quantidade de classes fantoches são geradas quando fazemos Value Objects, entidades do hibernate, entre outros.

> Mas o que colocar nas minhas entidades do hibernate além de getters e setters?

Antes de tudo, verifique se você **realmente precisa** desses getters e setters.

Para que um setID na sua chave primária se o seu framework vai utilizar reflection ou manipulação de bytecode para pegar o atributo privado, ou se você pode passá-la pelo construtor? Sobre value objects, você realmente precisa dos seus setters?

Em muitos casos VOs são criados apenas para expor os dados, e não há necessidade alguma para os setters... bastando um bom construtor! Dessa forma evitamos um modelo de domínio anêmico.

No hibernate costumo colocar alguns métodos de negócio, algo parecido como na classe `Conta`. Minhas entidades possuem uma certa responsabilidade, em especial as que dizem respeito aos atributos pertencentes a elas.

Para quem ainda está aprendendo, a apostila da caelum de java e orientação a objetos tem algumas dessas discussões e procura ensinar OO comentando sempre dessas más práticas, como por exemplo o uso de herança sem necessidade.

Vale relembrar que o exemplo é apenas didático. Trabalhar com dinheiro exige inúmeros cuidados, como não usar double. O foco desse post é para você parar

para pensar antes de criar o getter e o setter. E, claro, pode ser que o seu caso justifique a criação desses métodos.

Vemos bastante esse tópicos nos cursos de Java e de .NET na Caelum.

Tags: design patterns, fj-11, fj-16, Java, oo

Paulo Silveira (Google+)

MAIS SOBRE O AUTOR >

OUTROS POSTS DO AUTOR >