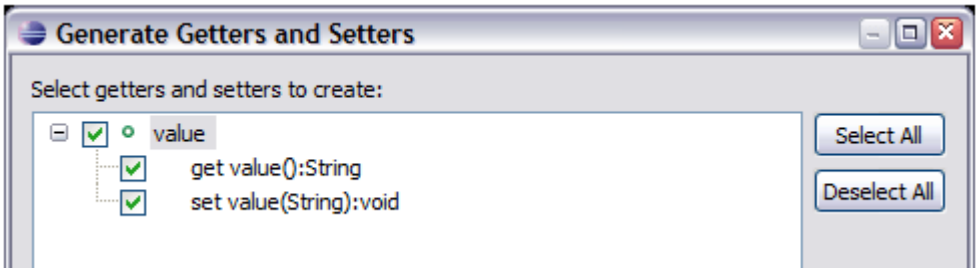
Cómo no aprender Java y Orientación a Objetos: getters y setters

Paulo Silveira

31/03/2021

Mucha gente pregunta:**¿Cómo aprender orientación a objetos?** Hay varias formas de aprender O.O., no creo que haya una mejor, pero hay formas de **no** aprender. Esta publicación, creada en 2006 y actualizada en 2017, muestra cómo el diseño de clases continúa siguiendo algunas importantes reglas de buenas prácticas.



Una de las prácticas más controvertidas que aprendimos al comienzo del aprendizaje de muchos lenguajes orientados a objetos es la generación indiscriminada de getters y setters. Los ejemplos básicos de cientos de tutoriales Java están repletos de getters y setters del peor tipo: aquellos que no tienen ningún sentido. Considera:

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

Rápidamente, los tutoriales explican el private para la encapsulación. Pero entonces, ¿cómo acceder? ¡Con Getters y setters!

```
class Cuenta {
    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;
    }
}
```

¿Cuál es el significado de este código? ¿Para qué este setSaldo? ¿y este setLimite? ¿y el getLimite? ¿Utilizará estos métodos? **Nunca cree un getter o setter sin sentir una necesidad real de ello.**

Esta es una regla para cualquier método, pero particularmente los getters y setters son campeones: muchos de ellos **nunca** se invocarán, y gran parte del resto podría y debería ser reemplazado por métodos de negocios.

Códigos del tipo cuenta.setSaldo(cuenta.getSaldo() + 100) se extenderán por todo tu código, lo que puede ser muy dañino: ¿cómo vas a conectar aumentos de saldo, si la forma de cambiar el saldo de una cuenta se extiende por todo tu sistema? Tomar dinero crea una situación aún más delicada: si necesitas verificar si hay un saldo en la cuenta antes de restar una cantidad del saldo, ¿dónde estará este if? ¿Esparcido por todo el sistema en varios puntos? ¿Qué pasará cuando necesites cambiar esta regla de negocio?

Entonces sigue nuestra clase Cuenta reformulada de acuerdo a esta necesidad:

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

    public Cuenta(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("¡sobrepasó el límite!"
    }

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

¡Y ni siquiera estamos hablando de *test driven development*! Probando la clase Cuenta notarás rápidamente que algunos de los getters y setters anteriores no tienen ningún uso y echaría de menos algunos métodos más enfocados en la lógica de negocio de tu aplicación, como el saca y el deposita.

Este ejemplo es muy trivial, pero puedes encontrar muchas clases que no parecen un *java bean* que exponen atributos como Connection, Thread, etc., sin ninguna necesidad. Indudablemente, existen peores prácticas: usar id para relacionar objetos, arrays no encapsuladas como estructuras, código basado en gran medida en la comparación de *Strings hardcoded* (que Guilherme Silveira llamó sarcásticamente de *POS*, o *programación orientada a strings*), miles de métodos estáticos, entre otros.

Todas estas prácticas son habituales cuando estamos empezando a romper el paradigma procedimental y confieso ya haber sido un gran practicante de algunas. Solo usarás bien el paradigma de la orientación a objetos después de cometer muchos errores.

Phillip Calçado aborda este tema, llamando a estas clases de **clases de títeres** (*puppets*). ¡Una clase de títeres es aquella que no tiene ninguna responsabilidad, aparte de llevar un puñado de atributos! ¿Dónde está la orientación a objetos? Muchas clases de títeres se generan cuando hacemos Value Objects, entidades de *hibernate*, entre otros.*Pero, ¿qué poner en mis entidades de hibernate además de getters y setters?*

Primero que nada, asegúrate si **realmente necesitas** de estos getters y setters.

¿Para qué un *setID* en su clave primaria si su framework usará la reflection o manipulación de bytecode para obtener el atributo privado, o si puedes pasarla por el constructor? Acerca de value objects, ¿realmente necesitas de tus setters? En muchos casos, se crean VO solo para exponer los datos, y no hay ninguna necesidad para los setters... ¡todo lo que se necesita es un buen constructor! De esa manera evitamos un modelo de dominio anémico.

En hibernate suelo utilizar algunos métodos de negocio, algo así como en la clase Cuenta. Mis entidades tienen una cierta responsabilidad, especialmente las relacionadas con los atributos que les pertenecen. Para aquellos que todavía están aprendiendo,está el artículo de POO de Alura, que tiene algunas de estas discusiones y busca enseñar OO comentando siempre estas malas prácticas, como el uso innecesario de la herencia.

Vale la pena recordar que, el ejemplo es solo didáctico. Trabajar con dinero requiere innumerables precauciones, como no usar double. El enfoque de esta publicación es que te detengas y pienses antes de crear el getter y setter. Y, por supuesto, puede ser que su caso justifique la creación de estos métodos.