**Encapsulamento**

**Problema do código não encapsulado**

Ao criar o método subtraiDesconto, resolvemos apenas uma parte do problema. Ele já torna nosso comportamento mais centralizado na classe Produto, evitando repetições, já que isola essa regra em comum. Mas ainda é possivel fazer:

$produto->preco -= $produto->preco \* 99999;

Em outras palavras, ninguém garante que o usuário dessa classe vai sempre usar o método subtraiDesconto, já que ele pode acessar o $preco do produto diretamente. Isso é um problema bem grave, nunca queremos permitir que nossa regra de negócio seja violada.

Como resolver isso? A melhor forma seria proibindo o acesso direto ao atributo $preco, assim o usuário da classeProduto só conseguirá aplicar um desconto utilizando o método subtraiDesconto, que faz todas as validações necessárias.

Para fazer isso, basta modificar a visibilidade do atributo para private. Dessa forma ele será acessivel apenas por dentro da própria classe:

class Produto

{

private $preco;

// outras propriedades escondidas

function subtraiDesconto($valor)

{

if ($valor > 0 && $valor < 1) {

// eu posso acessar, estou na mesma classe

$this->preco -= $this->preco \* $valor;

return $this->preco;

}

}

}

O private que utilizamos agora é, assim como o public, um modificador de visibilidade (acesso). Ao manter um atributo ou método como private, estamos fechando completamente sua visibilidade para todas as demais classes do sistema, exceto a própria classe que o declarou.

**Modificadores de visibilidade**

Além do public e private, há ainda um modificador de visibilidade chamado protected. Esse modificador restringe o acesso para a classe que o está definindo e também para suas classes filhas, quando estamos trabalhando com herança (que veremos mais à frente). Se quiser, você pode ler mais sobre esse e outros modificadores em:

<http://br2.php.net/manual/pt_BR/language.oop5.visibility.php>

Isso é excelente e já resolve nosso problema! Ninguém mais consegue aplicar um desconto em um produto sem ser passando pelo método subtraiDesconto. Mas ainda não terminamos, pois o seguinte código da listagem passa a dar erro:

<table class="table table-striped table-bordered">

<?php

$produtos = listaProdutos($conexao);

foreach ($produtos as $produto) :

?>

<tr>

<td><?= $produto->nome ?></td>

<td><?= $produto->preco ?></td>

<td><?= $produto->subtraiDesconto(0.1) ?></td>

<!-- continuação da tabela -->

O problema está na linha que imprime o preço do produto:

<td><?= $produto->preco ?></td>

Esse código agora é inválido, já que não podemos mais acessar o preço do produto por fora de sua classe.

É muito comum, e faz todo sentido, que suas propriedades sejam private e quase todos seus métodos sejampublic (não é uma regra!). Desta forma, toda conversa de um objeto com outro é feita por troca de mensagens, isto é, acessando seus métodos. Algo muito mais educado que mexer diretamente em um atributo que não é seu!

Melhor ainda! O dia em que precisarmos mudar como é feito um desconto em nosso Produto, adivinhe onde precisaríamos modificar? Apenas no método subtraiDesconto, o que faz pleno sentido. Como exemplo, imagine que o desconto em algum momento precise ser no máximo 30%. Basta você modificar ali, na classe Produto e nenhum outro código fora dela. Além disso, as classes que usam esse método nem precisam ficar sabendo de tal modificação! Ganhamos muito em esconder o funcionamento do nosso método na hora de dar manutenção e fazer modificações.

A idéia de esconder os membros de uma classe (como fizemos com o preço do produto), além de esconder o funcionamento interno de seus métodos, é conhecida como **encapsulamento**. Encapsular é fundamental para que seu sistema seja suscetível a mudanças: não precisaremos mudar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que essa regra está **encapsulada**.

**Getters e Setters**

Já que não conseguimos acessar o preco do produto diretamente, como faremos para mostrá-lo na tela de listagem, já que não podemos acessá-lo nem mesmo apenas para leitura?

Precisamos de uma **forma de fazer** esse acesso, um comportamento (**método**) com essa responsabilidade. Podemos fazer isso de forma simples, criando um método getPreco:

class Produto

{

private $preco;

// outras propriedades omitidas

function subtraiDesconto($valor)

{

if ($valor > 0 && $valor < 1) {

$this->preco -= $this->preco \* $valor;

return $this->preco;

}

}

public function getPreco()

{

return $this->preco;

}

}

E a partir de agora, podemos acessar o $preco do produto apenas por esse método. Em nossa listagem faremos:

<td><?= $produto->getPreco() ?></td>

Note que agora estamos permitindo acesso a esse atributo **de forma controlada**. E não chamamos esse método degetSaldo por acaso, essa é uma convenção muito conhecida. Sempre que queremos retornar (**pegar**) um valor, chamamos o método de get seguido do nome do atributo. Neste caso, getPreco. De forma similar, quando queremos mudar (**inserir**) um valor, chamamos o método de set. Um exemplo seria:

public function setPreco($preco)

{

$this->preco = $preco;

}

Note como fica nossa classe, agora com **todos** as propriedades private e seus devidos ::getters:: e ::setters::.

class Produto

{

private $id;

private $nome;

private $preco;

private $descricao;

private $categoria;

private $usado = false;

public function subtraiDesconto($valor)

{

if ($valor > 0 && $valor < 1) {

$this->preco -= $this->preco \* $valor;

return $this->preco;

}

}

public function getId()

{

return $this->id;

}

public function setId($id)

{

$this->id = $id;

}

public function getNome()

{

return $this->nome;

}

public function setNome($nome)

{

$this->nome = $nome;

}

public function getPreco()

{

return $this->preco;

}

public function setPreco($preco)

{

$this->preco = $preco;

}

public function getDescricao()

{

return $this->descricao;

}

public function setDescricao($descricao)

{

$this->descricao = $descricao;

}

public function isUsado()

{

return $this->usado;

}

public function setUsado($usado)

{

$this->usado = $usado;

}

public function getCategoria()

{

return $this->categoria;

}

public function setCategoria($categoria)

{

$this->categoria = $categoria;

}

}

Note que o setter sempre recebe um argumento, que é o valor que deve ser atribuido. Para adicionar o preço do produto, por exemplo, faremos:

<?php

$produto = new Produto();

$produto->setPreco(59.90);

?>

**Criamos getters e setter para tudo?**

Como já vimos, não devemos criar ::getters:: e ::setters:: sem um motivo explicito. No blog da Caelum há um artigo que ilustra bem esses casos. Apesar do artigo citar a linguagem Java, a regra aplica-se a qualquer linguagem OO:

<http://blog.caelum.com.br/2006/09/14/nao-aprender-oo-getters-e-setters/>

Feito isso, precisaremos modificar todos os lugares onde acessamos as propriedades da classe Produtodiretamento. O código agora está encapsulado! Um exemplo será no arquivo \*\*produto-lista.php%%. Ele deve ficar assim:

<?php include 'conecta.php'; ?>

<?php include 'banco-produto.php'; ?>

<?php include 'cabecalho.php'; ?>

<table class="table table-striped table-bordered">

<?php

$produtos = listaProdutos($conexao);

foreach ($produtos as $produto) :

?>

<tr>

<td><?= $produto->getNome() ?></td>

<td><?= $produto->getPreco() ?></td>

<td><?= substr($produto->getDescricao(), 0, 15) ?></td>

<td>

<form action="remove-produto.php" method="post">

<input type="hidden" name="id" value="<?=$produto->getId() ?>" />

<button class="btn btn-danger">remover</button>

</form>

</td>

<td>

<a href="produto-altera-formulario.php?

id=<?=$produto->getId()?>">Alterar</a>

</td>

</tr>

<?php endforeach ?>

</table>

<?php

if(array\_key\_exists("removido", $\_GET) && $\_GET['removido']== true) { ?>

<p class="alert-success">Produto apagado com sucesso.</p>

<?php } ?>

<?php include 'rodape.php'; ?>

Esse está pronto, mas ainda precisamos mudar os outros arquivos que fazem o acesso direto. Mãos a massa!