**Classe Abstrata**

**Classe Abstrata**

Quando estamos trabalhando com Livros, temos sempre que pensar em suas possíveis variações. Um Livropode ser físico, virtual, mini-livro, entre outros. Todo livro virtual (**ebook**), por exemplo, tem o que chamamos de marca d'agua para identificar seu proprietário -- ou ::waterMark::, como é conhecido. Além disso, quando um livro for físico, ele precisa ter uma taxa de impressão. Isso tudo precisa ser considerado em nosso código. Mas como?

Uma forma seria adicionar essas novas propriedades na classe Livro, que pode ficar assim:

<?php

class Livro extends Produto

{

private $isbn;

private $waterMark;

private $taxaImpressao;

// gets e sets omitidos

}

Resolveria o problema, mas já vimos que essa não é a condição ideal. Com essa alteração, todo Livro teriawaterMark e taxaImpressao, fazendo ou não sentido. Não é justo um **ebook** ter taxa de impressão, assim como um**livro físico** com ::water mark::. Como controlar isso?

A mesma motivação que nos levou a criar uma nova classe para representar um Livro, vale aqui! Se a classe tem estado e características próprias, nada mais justo do que criar um novo tipo para representá-la. Fizemos isso comLivro, que diferente de todo Produto, tem um isbn, e agora podemos fazer isso com seus tipos mais específicos.

Vamos criar a classe Ebook, que por enquanto tem apenas a propriedade waterMark:

<?php

class Ebook {

private $isbn;

private $waterMark;

function getWaterMark()

{

return $this->waterMark;

}

function setWaterMark($waterMark)

{

$this->waterMark = $waterMark;

}

function getIsbn()

{

return $this->isbn;

}

function setIsbn($isbn)

{

$this->isbn = $isbn;

}

}

Ainda não está legal, não é? O isbn é valido para todos os livros, sejam fisicos ou virtuais. No lugar de duplicar esse código, podemos tirar proveito da herança para definir que um Ebook é um Livro:

<?php

class Ebook extends Livro {

private $waterMark;

// get e set omitidos

}

Ótimo, já está bem melhor. Como ele **é um** livro, vai herdar o isbn. O mesmo para a classe LivroFisico, que por enquanto terá apenas uma taxaImpressao e herdará o isbn e tudo mais o que for comum entre esses tipos da classe Livro.

<?php

class LivroFisico extends Livro {

private $taxaImpressao;

function getTaxaImpressao()

{

return $this -> taxaImpressao;

}

function setTaxaImpressao($taxaImpressao)

{

$this -> taxaImpressao = $taxaImpressao;

}

}

Agora que temos uma forma forte de representar o tipo de um Livro, que pode ser Ebook ou LivroFisico, podemos nos expressar bem ao adicionar um novo Produto de algum desses tipos. Um exemplo:

Ebook ebook = new Ebook();

// seta as propriedades, etc

O mesmo vale para um LivroFisico, quando for desse tipo, damos new em sua classe que está muito bem definida. Mas o que acontece agora quando alguém criar um Livro? O código funcionará perfeitamente, afinal não se trata de um problema de sintaxe ou algo do tipo. O problema está na semântica. O que é um Livro agora? Quando alguém cria um novo Livro, qual seu tipo? Ebook? Físico? **Nenhum!**

Da mesma forma que não podemos ter um Livro sem tipo na realidade, porque permmitir isso em nosso código? Da mesma forma que não podemos ir à uma livraria e comprar um livro sem saber se é fisico ou virtual, não podemos deixar que isso aconteça em nosso sistema.

A partir de agora, um livro é apenas uma **abstração** de tudo que os diferentes tipos de livros devem ter em comum, como por exemplo o isbn. Todo livro, independente de seu tipo, precisa ter seu isbn único, e não queremos repetir isso toda hora em nosso código. É justo que a classe exista, para encapsular as regras de negócio em comum entre seus dois ou mais tipos, mas ela deve ser usada estritamente pra isso.

Por nossa sorte, a orientação a objetos nos oferece uma forma bem simples de resolver isso. Se a classe Livro é apenas uma **abstração**, podemos representar isso explicitamente adicionando o modificador abstract em sua assinatura:

abstract class Livro extends Produto

{

private $isbn;

// restante do código omitido

}

Legal, não é? Na prática isso significa que ninguém poderá instanciar um objeto desse tipo, mas sim suas ::subclasses::. Vamos tentar adicionar um novo Livro pra ver o resultado? Assim como esperado, ao executar umnew Livro(), receberemos um erro fatal:

Fatal error: Cannot instantiate abstract class Livro

in /Users/Turini/Desktop/cursophp/loja/banco-produto.php

on line 19

A mensagem já diz tudo, mas e agora? Como fazer esse o código de adição voltar a funcionar?

Podemos começar pelo nosso produto-formulario.php, que atualmente tem um ::select:: com as opções produto e livro.

<select name="tipoProduto">

<option value="geral">geral</option>

<option value="livro">livro</option>

</select>

No lugar de livro, vamos mudar as opções para **Livro Físico** e **Ebook**. O html ficará assim:

<select name="tipoProduto">

<option value="geral">geral</option>

<option value="Ebook">Ebook</option>

<option value="LivroFisico">Livro Físico</option>

</select>

Mas o resultado não será tão legal, precisamos agrupar ebook e livro físico de alguma forma, pra ficar visivelmente perceptível que são elementos do mesmo tipo. Vamos mudar um pouco, criando um grupo pros livros. Podemos fazer isso utilizando a tag optgroup do html:

<select name="tipoProduto">

<option value="Produto">geral</option>

<optgroup label="Livro">

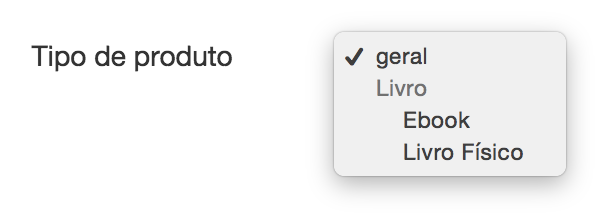
<option value="Ebook">Ebook</option>

<option value="LivroFisico">Livro Físico</option>

</optgroup>

</select>

Visualmente a mudança já resultará alguma diferença em nosso formulário:



Mas ainda precisamos mudar a lógica de adicionar, que atualmente está assim:

if (strcasecmp($\_POST["tipoProduto"], "livro") == 0) {

$produto = new Livro();

$produto->setIsbn($\_POST['isbn']);

} else {

$produto = new Produto();

}

$produto->setNome($\_POST["nome"]);

$produto->setPreco($\_POST["preco"]);

// outros setters

Um caminho simples seria apenas mudar o if, verificando o tipo do livro:

if (strcasecmp($\_POST["tipoProduto"], "LivroFisico") == 0) {

$produto = new LivroFisico();

$produto->setIsbn($\_POST['isbn']);

}

if (strcasecmp($\_POST["tipoProduto"], "Ebook") == 0) {

$produto = new Ebook();

$produto->setIsbn($\_POST['isbn']);

}

else {

$produto = new Produto();

}

Mas, apesar de resolver, não seria uma solução tão elegante. A cada novo tipo, adicionaremos um novo if. Além disso, qual a chance de esquecer de alterar? E se tiver isso em varios lugares, como saber que alterei todos? O código fica cada vez maior, mais complicado e difícil de manter. Mas como torná-lo mais simples e fácil de manter?

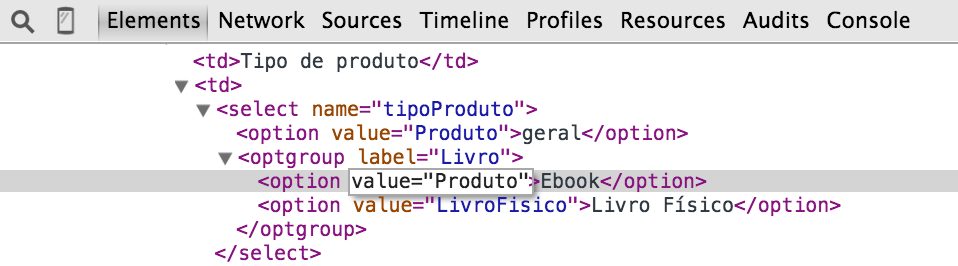
Uma forma bastante utilizada pelos desenvolvedores de PHP é utilizar o próprio valor do $\_POST["tipoProduto"], que é o nome da classe, para instanciá-la dinâmicamente. No lugar dos vários ifs, nosso código ficará assim:

$tipoProduto = $\_POST["tipoProduto"];

$produto = new $tipoProduto;

Simples, não é? Mas precisaremos ter dois cuidados importantes. O primeiro é garantir que o parâmetrotipoProduto terá exatamente o nome da classe, que por enquanto pode ser um Produto, Ebook ou LivroFisico. Afinal, se o parâmetro conter um nome inválido, de uma classe que não existe, por exemplo, um erro fatal ocorreria ao tentar instanciar esse tipo.

Mas isso é simples, é com o segundo ponto que realmente temos que nos preocupar. O que acontece se um usuário editar o html da página, mudando o ::name:: do ::select::? Afinal, qualquer um pode clicar com o botão direito no item do html e fazer um ::inspect element::. O html poderá ser editado pelo próprio navegador, sem nenhuma dificuldade:



Nesse exemplo, quando alguém selecionar um Ebook é a classe ProdutoDAO que seria instanciada. Um caos, não só pelo erro fatal que vamos receber em um ponto futuro de nosso código, mas sim pela falha de segurança grave que estaríamos deixando passar. Como resolver? É simples, basta limitar as classes que podem ser intanciadas. Um único if já pode cuidar disso:

$classes = array("Produto", "Ebook", "LivroFisico");

if (in\_array($tipoProduto, $classes)) {

$produto = new $tipoProduto;

}

Perfeito, agora apenas as classes permitidas poderão ser instanciadas. O único problema dessa solução é que esse código possívelmente vai se repetir, não queremos deixar essa regra de negócio espalhada. Vamos criar um classe com essa única responsábilidade, criar os produtos que são permitidos:

<?php

class CriadorDeProdutos

{

private $classes = array("Produto", "Ebook", "LivroFisico");

function criaPor($tipoProduto)

{

if (in\_array($tipoProduto, $this->classes)) {

return new $tipoProduto;

}

return new Produto();

}

}

Veja que o uso dessa classe é simples, dado um tipo de produto, ela criará ele pra você -- contanto que seja um tipo válido. Apesar de simples, o papel dessa classe é bastante importante, encapsular o processo de criação de nossos produtos. Ela é uma **fábrica de produtos**, portanto é uma prática comum chamar esse tipo de classe de Factory. Nesse caso ProdutoFactory, que é o nome do que ela fabrica, seguido do sufixo Factory.

class ProdutoFactory

{

// ...

}

**O padrão de projeto factory method**

Não foi a toa que a classe foi chamada de ProdutoFactory, esse é um padrão muito comum e conhecido no mundo orientado a objetos. É um dos famosos **design patterns**. Quer saber mais sobre esse e outros padrões de projeto? Então com certeza você vai gostar do curso de design patterns do Alura:

<http://www.alura.com.br/>

Agora que essa regra de negócio está **encapsulada** em uma classe, não precisamos nos preocupar em adicionar novos tipos de produto e ter que sair alterando um zilhão de pontos do nosso código. Isso será feito em um único lugar, na fábrica de produtos.

No lugar do if, agora nosso código de adição de produtos fica assim:

$tipoProduto = $\_POST["tipoProduto"];

$factory = new ProdutoFactory();

$produto = $factory->criaPor($tipoProduto);

// ...

Já podemos testar, a adição de produtos deve estar funcionando como esperado. Mas há um porém, não estamos persistindo a taxaImpressao e waterMark no banco de dados. Pra que isso funcione precisamos popular esses parâmetros, assim como estamos fazendo com os demais, e também modificar o método insereProduto doProdutoDAO. O adiciona-produto.php ficará assim:

$tipoProduto = $\_POST["tipoProduto"];

$factory = new ProdutoFactory();

$produto = $factory->criaPor($tipoProduto);

$produto->setIsbn($\_POST["isbn"]);

$produto->setWaterMark($\_POST["waterMark"]);

$produto->setTaxaImpressao($\_POST["taxaImpressao"]);

// ...

Mas tente adicionar um produto agora, o resultado será:

Fatal error: Call to undefined method

Produto::setWaterMark()

in /Users/Turini/Desktop/cursophp/loja/adiciona-produto.php

on line 14

Não podemos simplesmente chamar o set dessas propriedades, que estão presentes apenas em um ou outro tipo de livro. Precisamos de uma estratégia mais segura, como por exemplo:

$tipoProduto = $\_POST["tipoProduto"];

$factory = new ProdutoFactory();

$produto = $factory->criaPor($tipoProduto);

if (method\_exists($produto, "setIsbn")) {

$produto->setIsbn($\_POST["isbn"]);

}

if (method\_exists($produto, "setWaterMark")) {

$produto->setWaterMark($\_POST["waterMark"]);

}

if (method\_exists($produto, "setTaxaImpressao")) {

$produto->setTaxaImpressao($\_POST["taxaImpressao"]);

}

A função method\_exists, presente na API do PHP, simplifica nosso trabalho. Só tentaremos atribuiu o valor se o método que faz esse trabalho estiver presente, nos livramos do fatal error! Mas sinceramente, o código não está muito atraente, não é? O mesmo problema de sempre, varios ifs, vamos precisar copiar e colar toda vida, etc.

Pra começar a melhorar esse processo, vamos encapsular esse comportamento todo na classe Produto, em um método chamado atualizaBaseadoEm. Ele pode receber o array com os parâmetros e internamente o produto se vira pra atribuir ou não essas informações que chegaram da request. O código pra adicionar ficará assim:

$tipoProduto = $\_POST["tipoProduto"];

$factory = new ProdutoFactory();

$produto = $factory->criaPor($tipoProduto);

$produto->atualizaBaseadoEm($\_POST);

E na classe Produto:

function atualizaBaseadoEm($params)

{

if (method\_exists($this, "setIsbn")) {

$this->setIsbn($params["isbn"]);

}

if (method\_exists($this, "setWaterMark")) {

$this->setWaterMark($params["waterMark"]);

}

if (method\_exists($this, "setTaxaImpressao")) {

$this->setTaxaImpressao($params["taxaImpressao"]);

}

$this->setNome($params["nome"]);

$this->setPreco($params["preco"]);

$this->setDescricao($params["descricao"]);

$this->setCategoriaId($params["categoria\_id"]);

$this->setUsado($params['usado']);

}

Ótimo! Ainda não chegamos a uma solução perfeita, mas logo vamos voltar e melhorar um pouco mais o comportamento desse método. Por enquanto vamos nos preocupar com a adição de produtos, que ainda não está completa: precisamos atualizar o DAO.

Seguindo a mesma ideia, o método insereProduto poderia ficar assim:

function insereProduto($conexao, $produto){

if(array\_key\_exists('usado', $\_POST)) {

$produto->setUsado(true);

} else {

$produto->setUsado(false);

}

$isbn = "";

if(method\_exists($produto, "getIsbn")) {

$isbn = $produto->getIsbn();

}

$waterMark = "";

if(method\_exists($produto, "getWaterMark")) {

$waterMark = $produto->getWaterMark();

}

$taxaImpressao = "";

if(method\_exists($produto, "getTaxaImpressao")) {

$taxaImpressao = $produto->getTaxaImpressao();

}

$query =

"insert into produtos (nome, preco, descricao,

categoria\_id, usado, isbn, waterMark, taxaImpressao) values

('{$produto->getNome()}', '{$produto->getPreco()}',

'{$produto->getDescricao()}', '{$produto->getCategoriaId()}',

'{$produto->isUsado()}', '{$isbn}', '{$waterMark}',

'{$taxaImpressao}')";

return mysqli\_query($conexao, $query);

}

Lembrando que essas propriedades precisam ser adicionadas na tabela do banco de dados. Pra isso, basta na aba ::SQL:: do ::phpMyAdmin:: executar:

ALTER TABLE produtos ADD COLUMN waterMark VARCHAR(255);

ALTER TABLE produtos ADD COLUMN taxaImpressao VARCHAR(255);

Pronto pra testar? Basta acessar o formulário e adicionar um produto de cada tipo. Tudo está funcionando como esperado.