**Método abstrato**

**Método abstrato**

A ideia de isolar o comportamento de atribuir os valores da classe produto no método atualizaBaseadoEm foi justa, afinal será muito mais fácil evoluir esse comportamento sem quebrar o restante do código. Apareceu um atributo novo? Muda apenas a classe Produto e pronto, todos que usam o método automaticamente passam a funcionar.

Mas repare na quantidade de ifs da nossa implementação, ainda não está legal. Cada novo atributo de uma classe filha específica, adicionamos um novo if?

* 1. function atualizaBaseadoEm($params)
  2. {
  3. if (method\_exists($this, "setIsbn")) {
  4. $this->setIsbn($params["isbn"]);
  5. }
  6. if (method\_exists($this, "setWaterMark")) {
  7. $this->setWaterMark($params["waterMark"]);
  8. }
  9. if (method\_exists($this, "setTaxaImpressao")) {
  10. $this->setTaxaImpressao($params["taxaImpressao"]);
  11. }
  12. $this->setNome($params["nome"]);
  13. $this->setPreco($params["preco"]);
  14. $this->setDescricao($params["descricao"]);
  15. $this->setCategoriaId($params["categoria\_id"]);
  16. $this->setUsado($params['usado']);
  17. }

Se apenas Ebook tem waterMark, porque a classe Produto tem que atribuir esse valor? O acoplamento entre a classe pai e suas filhas está muito alto, isso não é bom sinal. A ::superclasse:: nunca deveria ter regras de negócio específicas de uma de suas filhas. Mas como resolver?

Assim como já vimos, é possível e totalmente recomendado que as classes filhas reescrevam os métodos da classe pai quando precisam se comportar de forma diferente. A classe Ebook, por exemplo, poderia reescrever o métodoatualizaBaseadoEm adicionando sua propriedade específica. O mesmo para as demais. Assim o método da classeProduto ficaria assim, sem nenhum if:

* 1. function atualizaBaseadoEm($params)
  2. {
  3. $this->setNome($params["nome"]);
  4. $this->setPreco($params["preco"]);
  5. $this->setDescricao($params["descricao"]);
  6. $this->setCategoriaId($params["categoria\_id"]);
  7. $this->setUsado($params['usado']);
  8. }

E agora, que temos esse método para atribuir os valores, não precisamos mais dos setters. O método ficará assim:

* 1. function atualizaBaseadoEm($params)
  2. {
  3. $this->nome = $params["nome"];
  4. $this->preco = $params["preco"];
  5. $this->descricao = $params["descricao"];
  6. $this->categoriaId = $params["categoria\_id"];
  7. $this->usado = $params['usado'];
  8. }

Perfeito! Se um produto não tem isbn, waterMark e tão menos taxaImpressao, porque essas informações deveriam estar em sua classe? Não devem.

Seguindo essa ideia, a classe Ebook ficará assim:

* 1. <?php
  2. class Ebook extends Livro {
  3. private $waterMark;
  4. // getter omitido
  5. function atualizaBaseadoEm($params)
  6. {
  7. $this->isbn = $params["isbn"];
  8. $this->waterMark = $params["waterMark"];
  9. $this->taxaImpressao = $params["taxaImpressao"];
  10. $this->nome = $params["nome"];
  11. // outras propriedades
  12. }
  13. }

O mesmo para as demais implementações. O LivroFisico, que também tem suas particularidades, pode ficar assim:

* 1. <?php
  2. class LivroFisico extends Livro {
  3. private $taxaImpressao;
  4. // getter omitido
  5. function atualizaBaseadoEm($params)
  6. {
  7. $this->isbn = $params["isbn"];
  8. $this->taxaImpressao = $params["taxaImpressao"];
  9. $this->nome = $params["nome"];
  10. // outras propriedades
  11. }
  12. }

Bem mais interessante, mas ainda longe de ser ideal. Um dos problemas dessa nossa solução é que um simples erro de digitação colocaria tudo a perder. Se errarmos o nome do método, chamando de atualizaBaseadoem (com E minúsculo, por exemplo), a classe vai ter dois métodos. O que ela herda do Produto, que não considera ataxaImpressao, e também o novo método escrito errado.

O outro problema vai um pouco além. A cada novo tipo de produto, eu preciso **lembrar de reescrever esse método**, caso contrário, o comportamento padrão da classe Produto será aplicado. Nem sempre isso faz sentido.

Se eu esquecer de sobrescrever o método atualizaBaseadoEm na classe Ebook, não vou persistir seu waterMark, que é muito importante pra nossa regra de negócio. Nunca devemos depender da memória humana, que por melhor que seja, falha.

Mas se não podemos depender disso, como lembrar de sempre sobrescrever um método?

É bem simples. Na orientação a objetos podemos garantir que todas as classes filhas implementam determinado método de sua ::superclasse::, deixando esse **método abstrato**. Podemos, neste caso, deixar o métodoatualizaBaseadoEm abstrato:

* 1. class Produto
  2. {
  3. // ...
  4. abstract function atualizaBaseadoEm($params);
  5. }

Isso mesmo, um método sem corpo! Ele é apenas uma abstração, que todos os filhos de Produto são obrigado a implementar, cada um com sua regra própria. Não seguir essa regra resultaria em um erro fatal. Mas ainda tem um detalhe que não conhecemos. O que acontece se alguém dar new na classe Produto e chamar seu métodoatualizaBaseadoEm? Ele não tem corpo, não tem implementação! Por esse motivo, antes mesmo disso o PHP vai gerar o seguinte erro:

* 1. Fatal error: Class Produto contains 1 abstract
  2. method and must therefore be declared abstract
  3. or implement the remaining methods: atualizaBaseadoEm

A mensagem é bem explicativa. Se tem 1 ou mais métodos abstratos, a classe tem que ser abstrata! Ninguém mais vai poder fazer new na classe Produto, já que ela tem métodos sem corpo, métodos **abstratos**. Faz sentido.

* 1. abstract class Produto
  2. {
  3. // ...
  4. abstract function atualizaBaseadoEm($params);
  5. }

Claro, a classe ainda poderá sim ter métodos não abstratos. Métodos comuns. As propriedades, getters e setters continuam valendo. Ainda podemos herdar tudo isso, mas a partir de agora, para cada produto, precisamos de uma classe para representá-lo. Se além de Livros, nossa loja for vender cursos, criamos uma classe Curso que herde de Produto e pronto.

Pro nosso código voltar a funcionar, podemos mudar a classe ProdutoFactory tirando o Produto de sua lista de classes que podem ser instanciadas e deixando o LivroFisico como implementação padrão. O código ficará assim:

* 1. class ProdutoFactory
  2. {
  3. private $classes = array("Ebook", "LivroFisico");
  4. function criaPor($tipoProduto)
  5. {
  6. if (in\_array($tipoProduto, $this->classes)) {
  7. return new $tipoProduto;
  8. }
  9. return new LivroFisico();
  10. }
  11. }

Além disso, podemos tirar a opção "geral" da lista de produtos que podem ser adicionados no arquivo adiciona-produto.php, já que não temos mais a classe Produto, que era muito genérica.

* 1. <td>Tipo de produto</td>
  2. <td>
  3. <select name="tipoProduto">
  4. <optgroup label="Livro">
  5. <option value="Ebook">Ebook</option>
  6. <option value="LivroFisico">Livro Físico</option>
  7. </optgroup>
  8. </select>
  9. </td>

Vamos testar? Basta adicionar um novo produto de cada tipo. Tudo continua funcionando.

**Relembrando algumas regras de classe e método abstrato**

Apensar de ser bem simples, vimos que existem algumas regras que precisam ser respeitadas quando estamos trabalhando com classes e métodos abstratos. Vamos lembrar algumas delas:

* Se a classe é abstrata, não podemos instanciá-la. A classe pode ser utilizada para herança e polimorfismo, mas ninguém pode dar new nesse tipo.
* Se o método é abstrato, todas as classes filhas são obrigadas a ter esse método implementado. E se eu não implementar? Em uma linguagem compilada, como Java, C# e outras, isso resultaria em um erro de compilação. Como PHP é uma linguagem interpretada, ao tentar traduzir a classe filha que não tenha o método declarado, recebemos um erro fatal.
* Note que a classe Livro não precisou implementar o método abstrato da classe Produto. Isso aconteceu porque, se a classe é abstrata, ela não precisa implementar os métodos abstratos de sua classe pai. Seus filhos sim.
* Uma classe abstrata pode sim ter propriedades e métodos que não sejam abstratos, mas a partir do momento em que tenha um método abstrato, a classe tem que ser abstrata também.