08

Faça o que eu fiz na aula

Substituir Array por Objeto

Vamos utilizar os arquivos que estão na pasta substituir_array_por_objeto, nesta pasta existe um arquivo bancodedados.db e um arquivo index.php, no arquivo index.php existe um array com dados, uma conexão com o banco de dados e uma consulta para inserir estes dados no banco.

Depois temos uma série de chamadas para o método `bindParam` , este método faz com que o parâmetro da consulta tenha um valor vinculado a ele. Porém, no vídeo vimos como fica confuso quando temos um array, o que a posição um do array contém? e a dois? isso deixa o código mais difícil de entender.

Podemos utilizar uma refatoração chamada Substituir Array por Objeto, e trocar todo esse array por chamadas de métodos com nomes mais descritivos.

Então vamos criar uma classe chamada `Usuario` e vamos adicionar os campos preparados para receberem as posições do array.

```php
<?php declare(strict_types=1);

namespace Alura\SubstituirArrayPorObjeto;

class Usuario
{
    private $nome;
    private $sobrenome;
    private $empresa;
    private $cargo;

    public function __construct(string $nome, string $sobrenome, string $empresa, string $cargo)
    {
        $this->nome = $nome;
        $this->sobrenome = $sobrenome;
        $this->empresa = $empresa;

        $this->cargo = $cargo;
    }

    public function getNome(): string
    {
        return $this->nome;
    }

    public function getSobrenome(): string
    {
        return $this->sobrenome;
    }

    public function getEmpresa(): string
    {
        return $this->empresa;
    }
```

```
    public function getCargo(): string
    {
        return $this->cargo;
    }
}
```

Depois disso, no arquivo index.php, vamos inicializar um objeto a partir dessa classe usando a sintaxe:

```
$usuario = new Usuario(
    $dadosUsuario[0],
    $dadosUsuario[1],
    $dadosUsuario[2],
    $dadosUsuario[3]
);
```

E agora, ao invés de chamarmos uma posição de um array, podemos chamar as propriedades, o que torna o código muito mais legível, e conseguimos ver o que estamos chamando.

Para facilitar mais ainda, podemos ainda utilizar uma funcionalidade do PDO chamada parâmetros nomeados, ou named parameters em inglês, onde ao invés de chamarmos os parâmetros na query com os pontos de interrogação, podemos utilizar `:nome` , `:sobrenome` .

```
$stmt->bindValue(':nome', $usuario->getNome());
$stmt->bindValue(':sobrenome', $usuario->getSobrenome());
$stmt->bindValue(':empresa', $usuario->getEmpresa());
$stmt->bindValue(':cargo', $usuario->getCargo());
```

E com isso, podemos deixar mais fácil a leitura do parâmetro também, fazendo uso de uma refatoração que vimos no curso, chamada Substituir Número Mágico.

Se recarregarmos a página, podemos ver que continuamos com o mesmo comportamento e a refatoração foi bem sucedida.

Encapsular Campo

Neste vídeo, os arquivos que vão ser utilizados estão na pasta encapsular_campo, nesta pasta existem três arquivos, Empresa.php, Funcionario.php e index.php. No arquivo index.php, são incluídas e instanciadas as classes e é chamado o método `adicionarFuncionario` da classe Empresa.

No método `adicionarFuncionario` , podemos ver que o método adiciona um funcionário em uma lista de funcionários e é chamado o método `promoveFuncionario` .

No método `promoveFuncionario` é passado um objeto do tipo `Funcionario` e se o usuário existir, ele aumenta o salário atribuindo um valor na propriedade pública chamada `salario` .

Como vimos no vídeo, não é uma boa prática a exposição direta de campos com acessos públicos, pois o encapsulamento do campo se quebra, e ao invés de apenas aumentar o salário como queríamos, podemos cometer um erro e acabar substituindo o valor inteiro sem querer.

Podemos aplicar uma refatoração chamada Encapsular Campo, para fazermos essa refatoração vamos para a classe `Funcionario` e vamos deixar os campos `$nome` e `$salario` como privados e adicionar os métodos `getNome` e `getSalario`

```
    private $nome;
    private $salario;


    public function getNome(): string
    {
        return $this->nome;
    }

    public function getSalario(): float
    {
        return $this->salario;
    }
```

Mas ainda temos o problema com o salário, vamos criar um método chamado `aumentaSalario` na classe Funcionario para encapsular a lógica de aumentar o salário:

```
    public function aumentaSalario(float $aumento): void
    {
        $this->salario += $aumento;
    }
```

Agora no método `promoveFuncionario` da classe `Empresa` , podemos chamar este método `aumentaSalario` ao invés de fazermos a alteração diretamente na propriedade.

```
    public function promoveFuncionario(
        Funcionario $possivelFuncionario,
        float $aumento
    ) {
        foreach ($this->funcionarios as $funcionario) {
            if ($funcionario->getNome() === $possivelFuncionario->getNome()) {
                $funcionario->aumentaSalario($aumento);
            }
        }
    }
```

No arquivo index.php agora podemos chamar estes novos métodos que criamos, ficando assim:

```
$alura->adicionarFuncionario($funcionario);

echo $funcionario->getSalario();

$alura->promoveFuncionario($funcionario, 50);
```

Com isso ficamos com um código mais limpo, livre de possíveis erros futuros que vão facilitar a manutenção do código.

Substituir Subclasses por Campo

Neste vídeo, vamos utilizar os arquivos que estão na pasta substituir_subclasses_por_campo, dentro dessa pasta existem quatro arquivos, vamos abrir o arquivo index.php.

No arquivo index.php, estão sendo incluídas as classes `Microondas110` e `Microondas220`, estas duas classes implementam a interface Microondas, que tem um método chamado `getVoltagem`.

Porém, vimos no vídeo que criar duas classes para mudar apenas o valor da voltagem do `Microondas` é uma complexidade desnecessária, podemos simplificar este código utilizando a refatoração chamada Substituir Subclasses por Campo, e armazenar a voltagem como um campo.

Vamos alterar a classe `Microondas` e transformá-la numa classe que não é abstrata, adicionar um campo privado chamado `$voltagem`, atribuí-lo no construtor com uma visibilidade privada e criar um método chamado `getVoltagem` que retorna esse campo.

```
<?php declare(strict_types=1);

namespace Alura\SubstituirSubclassesPorCampo;

class Microondas
{

    private $voltagem;

    private function __construct(int $voltagem)
    {
        $this->voltagem = $voltagem;
    }

    public function getVoltagem(): int
    {
        return $this->voltagem;
    }
}
```

Não vamos precisar mais dessas duas classes `Microondas110` e `Microondas220`, então vamos apagá-las. E para conseguirmos instanciá-las, vamos utilizar um padrão de projeto chamado Método Fábrica, ou no inglês, Factory Method.

```
    public static function criarMicroondas220(): Microondas
    {
        return new Microondas(220);
    }

    public static function criarMicroondas110(): Microondas
    {
        return new Microondas(110);
    }
```

No arquivo index.php, podemos remover os requires para as classes que não existem mais, e vamos trocar a instanciação por uma chamada de método estático:

```
$microondas110 = Microondas::criarMicroondas110();
$microondas220 = Microondas::criarMicroondas220();
```

Com isso, podemos acessar pelo navegador e ver que o comportamento continua o mesmo, mas conseguimos remover classes desnecessárias e tornar o projeto mais simples.