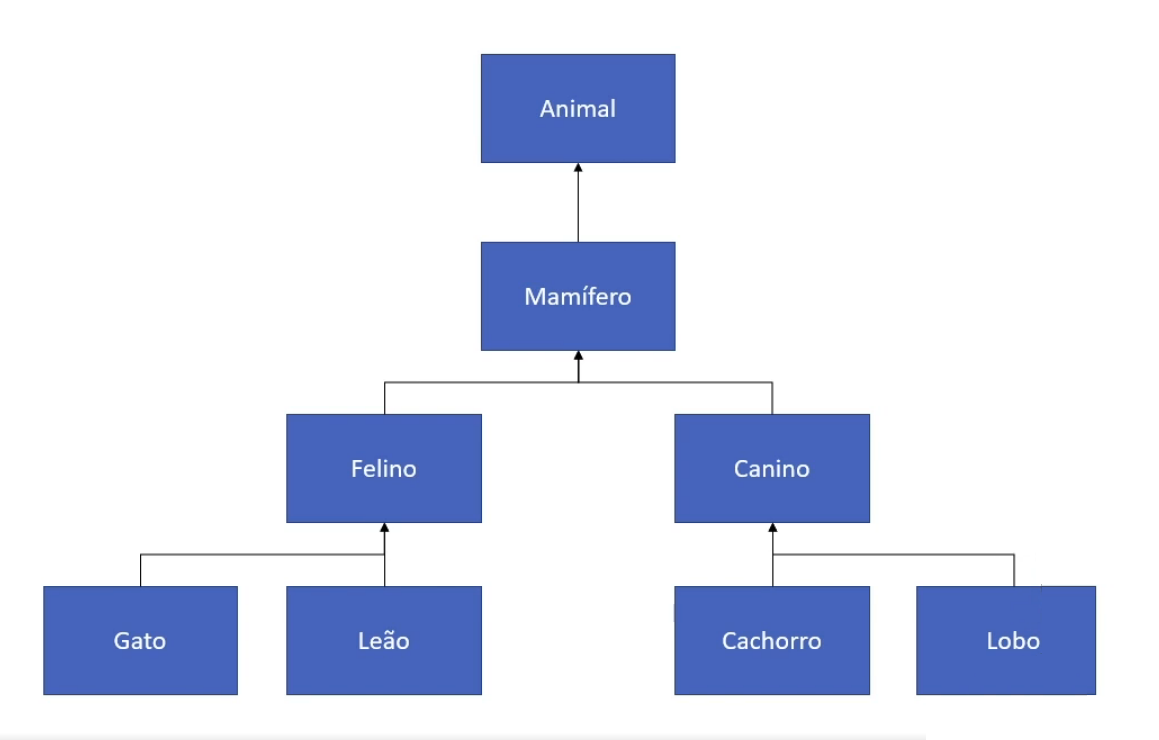
**Usando Herança**

Antes de começarmos a trabalhar efetivamente no código, faremos uma breve digressão e discutiremos uma divisão, talvez um pouco simplória, do reino animal.

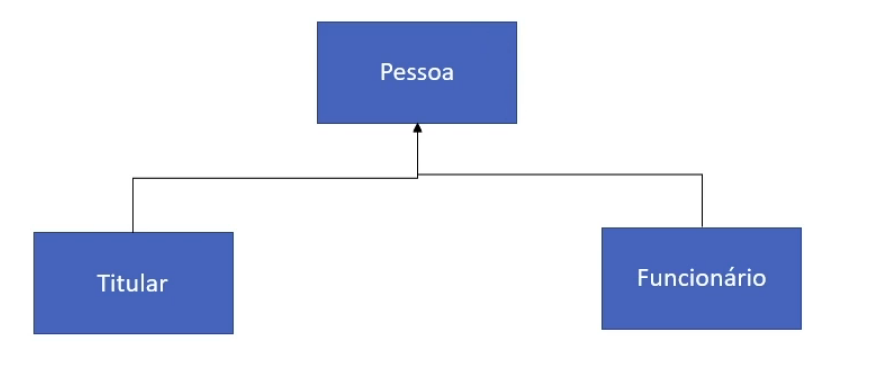


Imagine que tenhamos quatro animais: gato, leão, cachorro e lobo. Se definirmos as características desses animais, perceberemos que eles possuem várias em comum: o gato e o leão são felinos, e o cachorro e o lobo são caninos. Tanto felinos quanto caninos são mamíferos, e todo mamífero é um animal. Ou seja, são características cada vez mais genéricas e comuns a esses indivíduos.

Sabemos que todo animal respira de alguma forma; todo mamífero é parido e mama, e fornece leite se for fêmea; felinos têm suas características próprias, como os bigodes; e os caninos também têm características próprias, como os dentes.

Seguindo essa lógica, todo gato tem as características de um felino, que tem as características de um mamífero e que, por sua vez, tem as características de um animal. Da mesma forma, um lobo tem as características de um canino, que tem as de um mamífero e que tem as de um animal.

Mas o que isso tem a ver com a nossa estrutura de banco? Sabemos que um titular tem os dados de uma pessoa, que são CPF e nome, assim como um funcionário também possui esses dados. Ou seja, temos uma relação de hierarquia na qual uma definição base pode ser estendida para outras definições.



Esse é um conceito chamado de **herança**, e é o que queremos aplicar no nosso código. Na pasta "src", criaremos a classe Pessoa na qual definiremos os atributos $nome e $cpf. No construtor, receberemos uma string $nomee um $cpf do tipo CPF, fazendo a atribuição deles no corpo do método.

Teremos também os métodos de acesso recuperaNome(), que retornará uma string $this->nome, e recuperaCpf(), que também retornará uma string, mas por meio da chamada de $this->cpf->recuperaNumero().

class Pessoa

{

private $nome;

private $cpf;

public function \_\_construct(string $nome, CPF $cpf)

{

$this->nome = $nome;

$this->cpf = $cpf;

}

public function recuperaNome(): string

{

return $this->nome;

}

public function recuperaCpf(): string

{

return $this->cpf->recuperaNumero();

}

}

A ideia agora é fazermos com que nosso titular seja uma pessoa, ou seja, com que a classe Titular tenha tudo que Pessoa tem, com a adição do $endereco. Para isso, usaremos a palavra-chave extends, determinando que Titular estende aquilo que a classe Pessoajá definiu.

class Titular extends Pessoa

{

private $cpf;

private $nome;

private $endereco;

public function \_\_construct(CPF $cpf, string $nome, Endereco $endereco)

{

$this->cpf = $cpf;

$this->validaNomeTitular($nome);

$this->nome = $nome;

$this->endereco = $endereco;

}

//...

Como $nome e $cpf foram definidos em Pessoa, poderemos remover esses atributos da classe. A partir de agora o PhpStorm passará a apontar um erro no construtor, indicando que os membros $this->cpf e $this->nome têm acesso privado. Se eles são privados, só poderemos acessar a partir da própria classe Pessoa, e não por meio da classe Titular. Por enquanto corrigiremos isso tornando tais atributos públicos.

class Pessoa

{

public $nome;

public $cpf;

//...

Feito isso, a atribuição dos valores aos atributos no construtor de Titular deixará de apresentar um erro. Também poderemos remover os métodos recuperaNome() e recuperaCpf(), já que são duplicados. No futuro conversaremos também sobre o método validaTitular().

class Titular extends Pessoa

{

private $endereco;

public function \_\_construct(CPF $cpf, string $nome, Endereco $endereco)

{

$this->cpf = $cpf;

$this->validaNomeTitular($nome);

$this->nome = $nome;

$this->endereco = $endereco;

}

private function validaNomeTitular(string $nomeTitular)

{

if (strlen($nomeTitular) < 5) {

echo "Nome precisa ter pelo menos 5 caracteres";

exit();

}

}

public function getEndreco(): Endereco

{

return $this->endereco;

}

}

Repetiremos esse processo para a classe Funcionario, fazendo com que ela estenda de Pessoa e removendo os atributos $nome e $cpf que já estão definidos nesta classe. No construtor, inicializaremos corretamente a variável $cpf recebida por parâmetro, agora sendo um objeto CPF, e removeremos os métodos repetidos recuperaNome() e recuperaCpf().

class Funcionario extends Pessoa

{

private $cargo;

public function \_\_construct(string $nome, CPF $cpf, string $cargo)

{

$this->nome = $nome;

$this->cpf = $cpf;

$this->cargo = $cargo;

}

public function recuperaCargo(): string

{

return $this->cargo;

}

}

Seria interessante que a classe Funcionario também possuísse uma validação de nome. Sendo assim, recortaremos o método validaTitular() de Titular usando o atalho "Ctrl + X" e o colaremos na classe Pessoa, tornando-o público para que possamos acessá-lo de fora da classe.

class Pessoa

{

public $nome;

public $cpf;

public function \_\_construct(string $nome, CPF $cpf)

{

$this->nome = $nome;

$this->cpf = $cpf;

}

//... código omitido ...//

public function validaNomeTitular(string $nomeTitular)

{

if (strlen($nomeTitular) < 5) {

echo "Nome precisa ter pelo menos 5 caracteres";

exit();

}

}

}

No construtor de Funcionario, passaremos a chamar o método $this->validaNomeTitular() antes da atribuição do $nome.

class Funcionario extends Pessoa

{

private $cargo;

public function \_\_construct(string $nome, CPF $cpf, string $cargo)

{

$this->validaNomeTitular($nome);

$this->nome = $nome;

$this->cpf = $cpf;

$this->cargo = $cargo;

}

//...

Sabemos que copiar e colar código não é uma das melhores práticas, e ainda temos códigos repetidos, mas estamos avançando no sentido de melhorar a nossa aplicação. Agora temos as classes Funcionario e Titular estendendo as funcionalidades de uma classe Pessoa.

Abriremos então o terminal interativo com php -a, faremos um require 'src/Pessoa.php', um require 'src/Funcionario.php' e um require 'src/CPF.php'. Em seguida, criaremos $umFuncionario recebendo uma nova instância de Funcionario para qual passaremos os valores ''Vinicius, new CPF('123.456.789-10'), 'Desenvolvedor'.

$umFuncionario = new Funcionario('Vinicius', new CPF('123.456.789-10'), 'Desenvolvedor');

Se fizermos o var\_dump() de $umFuncionario, teremos um objeto com "cargo", "nome" e "cpf".

object(Funcionario)#1 (3) {

["cargo":"Funcionario":private]=>

string(13) "Desenvolvedor"

["nome"]=>

string(8) "Vinicius"

["cpf"]=>

object(CPF)#2 (1) {

["numero":"CPF":private]=>

string(14) "123.456.789-10"

}

}

Repare que em Funcionario temos apenas a definição de $cargo, demonstrando que a definição de Pessoa agora também faz parte dessa classe. O mesmo vale para o Titular. Com isso, conseguimos remover bastante código repetido, como a definição dos atributos comuns e dos seus métodos de acesso, agora contidos em uma única classe genérica.

Porém, os atributos de Pessoa ainda estão públicos, o que sabemos que não é interessante (já que a qualquer momento é possível modificar um CPF, por exemplo), e podemos validar o nome de um titular mesmo que um titular não esteja sendo criado. No próximo vídeo conversaremos sobre essas questões de visibilidade.

**Parent e Protected**

Antes de prosseguirmos, faremos uma breve revisão do conteúdo abordado no vídeo anterior. Tínhamos duas classes separadas, Titular e Funcionario, com código duplicado, mas que indicava que tais classes eram de certa forma relacionadas. No caso, ambas possuíam os atributos $nome e $cpf, que foram extraídos para uma classe Pessoa.

Ao definirmos que um Funcionario e Titular estendem Pessoa, estamos indicando que Funcionario e Titular também são, essencialmente, uma Pessoa. Isso também significa que tanto Funcionario quanto Titular possuem $cpf, $nome e todas as funcionalidades incluídas na classe Pessoa, inclusive o seu construtor.

Se quisermos, ainda é possível criarmos uma instância de Pessoa no terminal interativo, passando os dados no formato correto, já que ela possui esse construtor.

require 'src/Pessoa.php';

$umaPessoa = new Pessoa('Vinicius', new CPF('123.456.789-10'));

Repare que tanto em Titular quanto em Funcionario estamos repetindo um trecho de código que atribui valores a $nome e $cpf, inclusive validando o $nome recebido por parâmetro - validação esta que não é feita na classe base, Pessoa.

class Titular extends Pessoa

{

private $endereco;

public function \_\_construct(CPF $cpf, string $nome, Endereco $endereco)

{

$this->cpf = $cpf;

$this->validaNomeTitular($nome);

$this->nome = $nome;

$this->endereco = $endereco;

}

//...

class Funcionario extends Pessoa

{

private $cargo;

public function \_\_construct(string $nome, CPF $cpf, string $cargo)

{

$this->validaNomeTitular($nome);

$this->nome = $nome;

$this->cpf = $cpf;

$this->cargo = $cargo;

}

//...

Nosso objetivo agora é que, ao criarmos uma nova instância de Titular, por exemplo, seja chamado o instrutor da classe base (também chamada de "classe mãe" ou "classe pai"). Isso pode ser feito por meio da palavra reservada parent, que se refere à classe mãe daquela que está fazendo a execução. A partir dela, poderemos chamar o construtor \_\_construct() passando o $nome e o $cpf recebidos por parâmetro.

class Titular extends Pessoa

{

private $endereco;

public function \_\_construct(CPF $cpf, string $nome, Endereco $endereco)

{

parent::\_\_construct($nome, $cpf);

$this->endereco = $endereco;

}

//...

Note que removemos a validação, que passará a ser feita no construtor de Pessoa.

class Pessoa

{

public $nome;

public $cpf;

public function \_\_construct(string $nome, CPF $cpf)

{

$this->validaNomeTitular($nome);

$this->nome = $nome;

$this->cpf = $cpf;

}

//...

Repetiremos esse processo no Funcionario, chamando o construtor da classe mãe.

class Funcionario extends Pessoa

{

private $nome;

private $cpf;

private $cargo;

public function \_\_construct(string $nome, CPF $cpf, string $cargo)

{

parent::\_\_construct($nome, $cpf);

$this->validaNomeTitular($nome);

$this->nome = $nome;

$this->cpf = $cpf;

$this->cargo = $cargo;

}

//...

Se você prestou atenção, notou que anteriormente o PhpStorm realçava o construtor das classes Titular e Funcionario em amarelo. Isso informava a ausência do construtor da classe base. Nosso código continuou funcionando pois no PHP essa ausência é permitida, ainda que não siga os princípios da orientação a objetos.

O construtor é um método utilizado para inicializar os atributos que um objeto de determinada classe terá. Ter uma classe base com um construtor significa que alguma coisa precisa ser inicializada, e, portanto, é interessante sempre chamarmos o construtor dela, mesmo que no PHP isso não seja obrigatório. Com isso evitamos duplicação de código e garantimos a consistência da nossa aplicação.

Outra possibilidade que essa alteração nos traz é a de tornar os atributos de Pessoa privados.

class Pessoa

{

private $nome;

private $cpf;

public function \_\_construct(string $nome, CPF $cpf)

{

$this->validaNomeTitular($nome);

$this->nome = $nome;

$this->cpf = $cpf;

}

//..

Se quisermos acessar o nome ou o CPF, poderemos fazer isso por meio dos métodos recuperaNome() e recuperaCpf() da própria classe Pessoa. Não temos a necessidade de acessar esses atributos fora da classe Pessoa e, portanto, podemos encapsulá-los nos objetos desse tipo.

Temos um método validaNomeTitular() em Pessoa, mas podemos permitir que os funcionários alterem seus nomes, por exemplo com um método alteraNome(), que funciona como um *setter*.

public function alteraNome(string $nome): void

{

$this->nome = $nome;

}

Temos um problema, pois precisamos validar o nome que é recebido nesse método.

public function alteraNome(string $nome): void

{

$this->validaNomeTitular($nome);

$this->nome = $nome;

}

Entretanto, não queremos que o método validaNomeTitular() de Pessoa seja público, pois queremos que ele só seja chamado dentro da própria classe ou nas classes filhas. A mesma coisa com a propriedade $nome.

O modificador de acesso que permite a visibilidade apenas nas classes filhas é chamado de protected. Quando qualquer membro, seja atributo ou método, é protegido (protected), ele pode ser acessado tanto pela classe na qual ele é definido quanto pelas classes filhas, mas não pelo mundo externo.

class Pessoa

{

protected $nome;

private $cpf;

public function \_\_construct(string $nome, CPF $cpf)

{

$this->validaNomeTitular($nome);

$this->nome = $nome;

$this->cpf = $cpf;

}

//... código omitido ...//

protected function validaNomeTitular(string $nomeTitular)

{

if (strlen($nomeTitular) < 5) {

echo "Nome precisa ter pelo menos 5 caracteres";

exit();

}

}

}

Agora, no construtor de Funcionario, deixaremos de inicializar os atributos $nome e $cpf, mantendo apenas o $cargo.

class Funcionario extends Pessoa

{

private $nome;

private $cpf;

private $cargo;

public function \_\_construct(string $nome, CPF $cpf, string $cargo)

{

parent::\_\_construct($nome, $cpf);

$this->cargo = $cargo;

}

//...

Aprendemos, então, dois conceitos muito importantes: é possível chamar métodos da classe base por meio da palavra parent, e podemos acessar atributos e métodos da classe base se eles forem definidos como protected. Não deixe de fazer os exercícios dessa aula para fixar esses conteúdos!

Uma coisa que está incomodando é que começamos a armazenar muitos arquivos na mesma pasta. Seria interessante começarmos a organizar melhor o nosso projeto, tanto com pastas separadas de acordo com as responsabilidades dos arquivos, quanto logicamente na escrita desse código. Conversaremos sobre isso no próximo capítulo.

**Separação física e lógica (namespace)**

Até o momento temos relativamente pouca coisa no nosso projeto, que é consistido basicamente de classes que fazem sentido no nosso modelo, como Conta, Funcionario, Pessoa, Titular e assim por diante. No futuro, poderemos ter também classes que realizam ações, ao invés de representarem um modelo de domínio - por exemplo, uma classe que encerra uma conta. A ideia é sermos capazes de separar classes de modelo das classes de serviço, que são as responsáveis por executar ações e funcionalidades. Portanto, começaremos a organizar o projeto com isso em mente.

Começaremos criando em "src" uma pasta "Modelo", na qual armazenaremos todas as classes criadas até agora, já que elas representam o modelo do nosso banco. Podemos organizar ainda melhor esses arquivos. Por exemplo, sabemos que Titular e Conta são entidades muito relacionadas. Sendo assim, dentro de "Modelo", criaremos uma pasta "Conta" para a qual moveremos ambos esses arquivos.

Claro, existem diversas formas de montar a estrutura de pastas do seu projeto, e muitos estudos de arquitetura visando montar essa estrutura da melhor forma, mas esses são detalhes que não abordaremos agora.

Fizemos então a separação física dos arquivos em pastas do nosso próprio sistema operacional. Mas e se quisermos utilizar alguma classe externa, por exemplo baixada da internet, para ajudar em alguma funcionalidade do banco? As chances de termos uma classe com nome idêntico aos do nosso projeto, como Conta e CPF, são bem grandes, e o PHP não saberá qual classe usar.

Podemos ter dois arquivos com o mesmo nome no sistema operacional, desde que eles estejam em pastas diferentes. O PHP fornece essa separação por meio de *namespaces*. Se temos uma pasta chamada "modelo", os arquivos contidos nela também estarão em um namespace chamado "Modelo". Começaremos essa definição por Pessoa.php.

<?php

namespace Modelo;

class Pessoa

{

protected $nome;

private $cpf;

//...

Não é obrigatório que o nome do *namespace* seja o mesmo da pasta, mas isso ajuda na organização, mesmo se mudarmos de IDE ou se passarmos o projeto para outra pessoa. Repetiremos esse processo para as outras classes (CPF, Endereco e Funcionario), e as organizaremos em ordem alfabética.

Já as classes Conta e Titular estarão no *namespace \*"Modelo\Conta". A contrabarra \ é utilizada para separar \*namespaces* no PHP, independentemente do sistema operacional.

Agora temos os arquivos separados em pastas e as classes separadas em *namespaces*, que são como "pacotes". Portanto, podemos ter várias classes dentro de um pacote: no pacote "Modelo\Conta" temos as classes Conta e Titular, e no pacote "Modelo" temos CPF, Endereco, Funcionario e Pessoa.

No futuro entenderemos melhor a utilidade dessa separação, mas, por enquanto, é o suficiente termos organizado melhor o nosso projeto. No próximo vídeo falaremos mais sobre situações onde temos projetos separados, mas com arquivos de nomes repetidos.

**Namespace raiz**

Se estivermos utilizando um projeto baixado da internet para nos auxiliar na criação do sistema do banco, ou mesmo se estamos utilizando um projeto feito por outra equipe para nos auxiliar com outro módulo da aplicação, existe a possibilidade de criarmos uma classe com mesmo nome, afinal Conta e CPF são títulos bastante genéricos.

Nesse cenário, se alguém precisar criar um arquivo com o mesmo nome, poderá fazer isso em outra pasta. Da mesma forma, podemos criar ou incluir uma nova classe em outro *namespace*. Porém, também não deve ser incomum ter uma classe CPF em um *namespace* chamado "Modelo". Pensando nisso, é interessante organizarmos os nossos *namespaces* de outra forma.

Uma forma bastante conhecida, e que nos trará uma grande vantagem no futuro, é termos um *namespace* raiz da nossa aplicação, e que será válido para todas as classes do projeto. A partir desse *namespace* raiz, todas as pastas do nosso sistema serão também *namespaces* próprios.

Por exemplo, o *namespace* raiz da nossa aplicação será "Alura\Banco". Sendo assim, a classe CPF fará parte do *namespace* "Alura\Banco\Modelo".

<?php

namespace Alura\Banco\Modelo;

class CPF

{

private $numero;

//...

Já a classe Conta estará no *namespace* "Alura\Banco\Modelo\Conta".

<?php

namespace Alura\Banco\Modelo\Conta;

class Conta

{

private $titular;

private $saldo;

private static $numeroDeContas = 0;

//...

Lembre-se que é uma ótima prática que o nome da classe seja o mesmo nome do arquivo!

Com isso, já incluímos diversas convenções da programação em PHP e da orientação a objetos em nosso projeto. Repetiremos esse processo para as demais classes, corirgindo o seu *namespace*.

Mesmo se baixarmos outro projeto da internet ou feito por outra equipe pra utilizarmos, as chances dos seus arquivos estarem no mesmo *namespace* são muito pequenas.

Porém, agora temos um problema. O PhpStorm passará a indicar diversos erros no arquivo banco.php, pois não está mais encontrando os caminhos informados para os arquivos. Corrigiremos isso passando o caminho correto.

<?php

require\_once 'src/Modelo/Conta/Conta.php';

require\_once 'src/Modelo/Conta/Titular.php';

require\_once 'src/Modelo/Endereco.php';

require\_once 'src/Modelo/CPF.php';

$endereco = new Endereco('Petrópolis', 'um bairro', 'minha rua', '71B');

$vinicius = new Titular(new CPF('123.456.789-10'), 'Vinicius Dias', $endereco);

$primeiraConta = new Conta($vinicius);

$primeiraConta->deposita(500);

$primeiraConta->saca(300); // isso é ok

//...

Entretanto, mesmo após as alterações, o PHP continuará não encontrando as classes, informando que elas não estão definidas. No próximo vídeo entenderemos como utilizar as classes agora que elas estão armazenadas em *namespaces*.

**Utilizando namespaces**

Com as alterações que fizemos no vídeo anterior, o PHP não consegue mais encontrar as classes do nosso projeto, fazendo com que nosso sistema deixasse de funcionar. Ao executarmos, recebemos erros informando, por exemplo, que a classe Pessoa não foi criada no momento da criação da classe Titular. Portanto, faremos o require dessa classe.

<?php

require\_once 'src/Modelo/Conta/Conta.php';

require\_once 'src/Modelo/Endereco.php';

require\_once 'src/Modelo/Pessoa.php';

require\_once 'src/Modelo/Conta/Titular.php';

require\_once 'src/Modelo/CPF.php';

É trabalhoso e pouco prático termos que fazer o require de cada classe que utilizaremos, ainda mais que a IDE não parece nos ajudar nesse processo. Manteremos isso em mente para resolvermos no futuro. Se executarmos novamente, continuaremos recebendo o erro de que a classe Pessoa não está sendo encontrada.

Isso acontece pois a classe |Titular está tentando encontrar uma classe Pessoa no *namespace* "Alura\Banco\Modelo\Conta", mas esta última está localizada em outro *namespace*. Sendo assim, precisaremos informar o *namespace* correto, algo que pode ser feito de duas maneiras.

A primeira delas é passarmos o endereço completo da classe, como no exemplo abaixo:

<?php

namespace Alura\Banco\Modelo\Conta;

class Titular extends Alura\Banco\Modelo\Pessoa

{

private $endereco;

//...

Ou seja, a classe Titularpassará a estender de Alura\Banco\Modelo\Pessoa, o que provavelmente fará com que a classe seja encontrada corretamente. Entretanto, usar um nome tão grande para definir a classe tornará o nosso código menos legível. Uma alternativa é, antes da definição de Titular, utilizarmos a instrução use seguida do mesmo endereço.

<?php

namespace Alura\Banco\Modelo\Conta;

use Alura\Banco\Modelo\Pessoa;

class Titular extends Pessoa

{

private $endereco;

//...

Com isso, estamos informando ao PHP que queremos utilizar a classe Pessoa no *namespace* informado. Em seguida, repetiremos o processo para todas as classes utilizadas em Titular - ou seja, CPF e Endereco.

<?php

namespace Alura\Banco\Modelo\Conta;

use Alura\Banco\Modelo\CPF;

use Alura\Banco\Modelo\Endereco;

use Alura\Banco\Modelo\Pessoa;

class Titular extends Pessoa

{

private $endereco;

public function \_\_construct(CPF $cpf, string $nome, Endereco $endereco)

{

parent::\_\_construct($nome, $cpf);

$this->endereco = $endereco;

}

//...

Com isso, a classe Titular deixará de apresentar erros, mas o nosso banco.php não. Ou seja, teremos que fazer o mesmo processo nesse arquivo, ou utilizando o nome completo da classe ou adicionando a instrução use antes das nossas operações. Faremos do segundo modo. Quando escrevemos o use seguido do nome de uma classe, por exemplo CPF, o PhpStorm nos ajuda mostrando opções de autocompletar com o seu *namespace* completo.

<?php

require\_once 'src/Modelo/Conta/Conta.php';

require\_once 'src/Modelo/Endereco.php';

require\_once 'src/Modelo/Pessoa.php';

require\_once 'src/Modelo/Conta/Titular.php';

require\_once 'src/Modelo/CPF.php';

use Alura\Banco\Modelo\Conta\Titular;

use Alura\Banco\Modelo\Endereco;

use Alura\Banco\Modelo\CPF;

use Alura\Banco\Modelo\Conta\Conta;

$endereco = new Endereco('Petrópolis', 'um bairro', 'minha rua', '71B');

$vinicius = new Titular(new CPF('123.456.789-10'), 'Vinicius Dias', $endereco);

$primeiraConta = new Conta($vinicius);

$primeiraConta->deposita(500);

$primeiraConta->saca(300); // isso é

Ao executarmos novamente o código, tudo funcionará corretamente. Repare que a IDE não nos auxilia tanto na importação de arquivos, mas consegue nos ajudar a informar os *namespaces*. Inclusive, se escrevermos new Pessoa e pressionarmos "Enter", a própria IDE preencherá o *namespace* completo.

Portanto, a utilização de *namespaces* é uma vantagem para a organização do nosso código, e qualquer boa IDE para desenvolvimento em PHP se encarregará do trabalho de importá-los. Porém, note que aparentemente estamos importando as classes duas vezes, usando require e use. Além disso, ao fazermos o require, tivemos que lembrar da ordem de execução para inserir os arquivos corretamente, algo que não é tão prático assim.

No próximo vídeo aprenderemos uma nova funcionalidade do PHP que nos livrará desses problemas.

**Mais sobre namespaces**

É muito útil e uma boa prática fazermos uso de **namespaces** para separar as nossas classes em nossa aplicação. Para fazer bom uso deste recurso, devemos entender alguns detalhes.

Quando definimos uma classe sem informar o seu namespace, ela é criada no namespace padrão do PHP. Este namespace tem um "nome vazio", ou seja, uma classe Conta sem namespace tem como seu "nome completo" \Conta.

Se em uma classe com namespace definido precisarmos acessar classes de outro namespace, precisamos adicionar o use ou informar o seu nome completo (namespace + nome da classe).

Estando na classe Alura\Banco\Modelo\Conta\Conta, por exemplo, e querendo acessar a classe Alura\Banco\CPF, precisamos informar o nome completo da classe, ou utilizar use Alura\Banco\CPF no início do arquivo, o que permite que no restante do arquivo utilizemos apenas CPF como nome.

Outro detalhe interessante é que, se precisamos "importar" (com use) mais de uma classe do mesmo namespace, podemos fazer na mesma linha, envolvendo os nomes das classes em chaves. Por exemplo:

<?php

namespace Alura\Banco\Conta;

use Alura\Banco\{CPF, Endereco};

use \SplFileInfo;

class Conta

{

public function umMetodoQualquer()

{

// ...

$cpf = new CPF();

// ...

$cpf = new Endereco();

// ...

$fileInfo = new SplFileInfo();

// ...

$fileObject = new \SplFileObject();

// ...

}

}

Pra ler tudo isso com mais detalhes, a documentação é o melhor lugar: <https://www.php.net/manual/en/language.namespaces.php>.

**Método Abstrato**

No último vídeo reparamos que os métodos saca() das classes Conta e ContaPoupanca são praticamente iguais, com exceção da tarifa que é aplicada. Ou seja, temos muito código duplicado. Pensando nisso, mudaremos de abordagem. Primeiro, removeremos o método saca() da ContaPoupanca. Em Conta, ao invés de um valor fixo 0.05, passaremos a chamar um novo método privado percentualTarifa() que simplesmente retornará um float com o valor a ser descontado - nesse caso, os mesmos 0.05.

class Conta

{

//... código omitido ...//

public function saca(float $valorASacar): void

{

$tarifaSaque = $valorASacar \* $this->percentualTarifa();

$valorSaque = $valorASacar + $tarifaSaque;

if ($valorSaque > $this->saldo) {

echo "Saldo indisponível";

return;

}

$this->saldo -= $valorSaque;

}

//... código omitido ...//

private function percentualTarifa(): float

{

return 0.05;

}

}

Portanto, no método saca() estamos indicando qeu a $tarifaSaque é conseguida multiplicando o $valorASacar pelo percentualTarifa(), que nesse caso é de 5%. A ideia, então, seria sobrescrevermos o método percentualTarifa() na classe ContaPoupanca. Entretanto, já que o definimos como private, isso não será possível, mesmo que a classe seja uma filha de Conta.

Para que a sobrescrita seja possível, o método precisa estar possível, algo que conseguiremos com o modificador de acesso protected.

protected function percentualTarifa(): float

{

return 0.05;

}

Com isso, poderemos implementar o método percentualTarifa() na ContaPoupanca, simplesmente retornando o valor 0.03.

class ContaPoupanca extends Conta

{

protected function percentualTarifa(): float

{

return 0.03;

}

}

Assim, continuaremos conseguindo modificar a tarifa a depender do tipo de conta, mas agora executando bem menos linhas de código.

Passaremos, então, a refletir sobre outros aspectos do nosso sistema. Quando instanciamos uma Conta, que tipo de conta exatamente ela é? Atualmente ela atua como uma conta corrente e, portanto, poderíamos renomeá-la com tal, certo?

Entretanto, a nossa ContaPoupanca estende de Conta, e uma conta poupança não é uma conta corrente - pelo menos não no nosso banco. Por exemplo, segundo a nossa regra de negócios, uma conta poupança não pode realizar transferências - algo que atualmente é possível por meio da herança.

Isso significa que precisaremos modificar a nossa estrutura. A entidade Conta continuará sendo uma conta genérica, com toda as suas funcionalidades, e já temos uma ContaPoupanca. Passaremos para a criação de uma ContaCorrente, classe que também estenderá as funcionalidades de uma Conta genérica e terá um percentualTarifa() de 5%.

class ContaCorrente extends Conta

{

protected function percentualTarifa(): float

{

return 0.05;

}

}

Além disso, como não queremos que o método transfere() exista na ContaPoupanca, vamos removê-lo da Conta genérica e passá-lo para a ContaCorrente.

class ContaCorrente extends Conta

{

protected function percentualTarifa(): float

{

return 0.05;

}

public function transfere(float $valorATransferir, Conta $contaDestino): void

{

if ($valorATransferir > $this->saldo) {

echo "Saldo indisponível";

return;

}

$this->saca($valorATransferir);

$contaDestino->deposita($valorATransferir);

}

}

Agora, no arquivo teste-saque.php, não conseguiremos chamar o método transfere() a partir de uma instância de Conta, e o próprio PhpStorm indicará que não conhece esse método. Já se instanciarmos uma ContaCorrente, o método poderá ser chamado.

Conseguimos implementar uma funcionalidade específica da conta corrente, e cada tipo de conta está fornecendo o percentual da sua tarifa. Porém, na Conta genérica, estamos definindo um percentual padrão. Se criássemos uma conta salário que fosse isenta de tarifa e esquecêssemos de implementar um método percentualTarifa(), ela continuaria recebendo os 5% definidos na conta genérica, algo que não queremos que aconteça.

A ideia é tornarmos a implementação do método percentualTarifa() a todas as classes que estenderem de Conta. Além disso, não teremos mais um valor padrão, tornando esse método uma abstração daquilo que desejamos que seja feito nas classes filhas. Isso é possível por meio da palavra-chave **abstract**.

abstract protected function percentualTarifa(): float;

Ao definirmos um método como **abstrato**, o próprio PhpStorm passará a indicar um erro em todas as classes que não o implementem. Note que, nesse momento, teremos um erro na classe Conta. Isso ocorre porque, para que uma classe possua métodos abstratos, ela mesma também precisa ser definida como abstrata.

<?php

namespace Alura\Banco\Modelo\Conta;

abstract class Conta

{

//... código omitido ...//

abstract protected function percentualTarifa(): float;

}

Note que todas as classes que estendem de Conta precisarão implementar o método abstrato percentualTarifa(), mas a Conta em si não possui esse método. Portanto, se tentássemos instanciá-la diretamente, a implementação do método não existiria. Isso significa que a classe em si não está "pronta", ou seja, não está inteiramente implementada, e é exatamente isso que a palavra "abstrata" quer dizer.

Quando criamos uma classe abstrata, estamos indicando que ela ainda está incompleta, e são as classes que estendem dela que irão completá-la. É exatamente isso que a ContaPoupanca e a ContaCorrente estão fazendo. A partir de agora também será impossível instanciar uma Conta genérica diretamente, sendo necessário criar uma instância de ContaCorrente ou ContaPoupanca, o que faz todo o sentido.

No futuro conversaremos um pouco mais sobre as classes abstratas. No momento, focaremos na ideia de que, quando queremos obrigar que todas as classes filhas implementam esse método, basta o definirmos como abstrato na classe mãe.

Repare também que, na classe Conta, estamos utilizando o método percentualTarifa() na funcionalidade saca().

public function saca(float $valorASacar): void

{

$tarifaSaque = $valorASacar \* $this->percentualTarifa();

$valorSaque = $valorASacar + $tarifaSaque;

if ($valorSaque > $this->saldo) {

echo "Saldo indisponível";

return;

}

$this->saldo -= $valorSaque;

}

Isso é possível pois, por mais que não tenhamos a sua implementação no momento, temos certeza de que as classes filhas farão tal implementação.

Estamos começando a nos aprofundar em conceitos mais interessantes e avançados da orientação a objetos. Já conhecemos três dos quatro pilares da orientação a objetos: abstração, encapsulamento e herança. No próximo capítulo conheceremos o último deles!

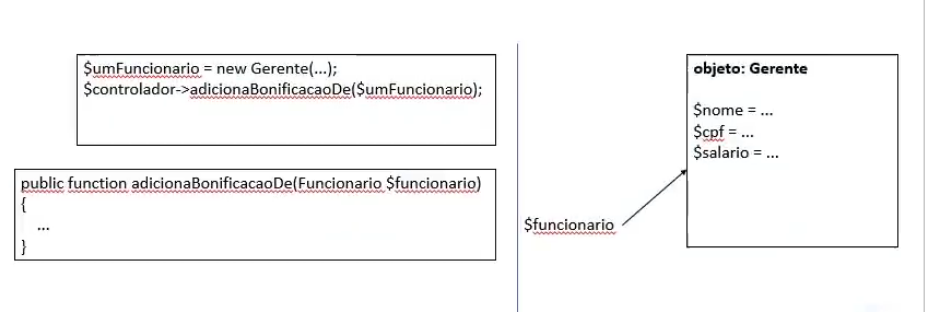
**Polimorfismo**

Nos capítulos anteriores nós conhecemos três dos quatro pilares da programação orientada a objetos: **herança**, **abstração** e **encapsulamento**. Agora introduziremos o quarto e último pilar, um conceito complexo e que provavelmente não ficará completamente explícito nesse momento.

Note que no arquivo bonificacoes.php estamos trabalhando com referências de três tipos diferentes, no caso Desenvolvedor, Gerente e Diretor. Todas elas são passadas para o método adicionaBonificacaoDe(), que teoricamente recebe uma referência do tipo Funcionario. Entretanto, nenhuma execução do nosso código quebrou.

Vamos refletir sobre isso. Uma referência de Desenvolvedor consegue se passar por uma referência de Funcionario? Na verdade sim, pois um Desenvolvedor estende de Funcionario, possuindo todas as características dessa classe. O mesmo vale para o Gerente e o Diretor. Sendo assim, quando uma referência chega ao método adicionaBonificacaoDe(), o importante é que seu tipo equivala a um Funcionario, independentemente do fato dessa referência ser mais específica.

Esse conceito fica um pouco mais nebuloso no PHP, pois ainda não conseguimos definir o tipo de uma variável. Sendo assim, vamos ilustrar.



Temos aqui a definição de uma variável $umFuncionario que faz referência a um Gerente. Em seguida, chamamos o método adicionaBonificacoesDe() passando esse $umFuncionario. Na implementação do método, ele recebe a variável $funcionario, que é uma referência de Funcionario.

A variável $funcionario que é passada para o método é, na verdade, uma referência de Funcionario, independentemente de qual seja o objeto instanciado. Da mesma forma, se instanciarmos $umFuncionario como Diretor, a referência recebida em adicionaBonificacaoDe() continuará sendo do tipo Funcionario.

O método em si não se importa com qual o tipo do objeto, desde que ele seja um Funcionario. Esse é o conceito de **polimorfismo**, que estipula que uma referência pode ter vários tipos e se comportar de formas diferentes, mas existindo semelhanças o suficiente de modo a atender a alguns requisitos.

No nosso caso, tanto faz termos um Gerente, Diretor ou Desenvolvedor, desde o método adicionaBonificacaoDe() seja capaz de calcular a sua bonificação (por meio da chamada de calculaBonificacao() e obstante a forma que o objeto faça isso.

Note que uma mesma referência pode ter comportamentos diferentes em situações diferentes. Por exemplo, em bonificacoes.php, o nosso $umFuncionario consegue se comportar como Desenvolvedor, chamando o método sobeDeNivel(), ou como um Funcionario genérico ao ser capaz de ser passado como parâmetro para adicionaBonificacaoDe().

O polimorfismo é um conceito bastante complexo, principalmente em questões teóricas, mas é muito utilizado durante o desenvolvimento. Na prática, podemos simplesmente pensar que se um Desenvolvedor é um Funcionario, faz sentido sermos capazes de utilizá-lo em métodos que precisam de um. Para se aprofundar no assunto, não deixe de consultar essa [atividade sobre polimorfismo](https://cursos.alura.com.br/course/php-oo-heranca-polimorfirmo-interfaces/task/67553).

Agora que conhecemos os quatro pilares da orientação a objetos, vamos revisitar alguns conceitos importantes e pensar no que pode ser feito quando nos deparamos com a necessidade de uma herança mais complexa.

**Mais sobre classes abstratas**

O banco nos passou uma nova demanda: a partir de agora cada cargo terá sua própria bonificação, e a bonificação padrão deixará de existir. Isso implica também que todo novo cargo precisará, obrigatoriamente, de uma bonificação própria. Antes de mais nada, corrigiremos um problema da nossa classe Funcionario, que ainda possui um atributo $cargo não mais necessário.

abstract class Funcionario extends Pessoa

{

private $salario;

public function \_\_construct(string $nome, CPF $cpf, float $salario)

{

parent::\_\_construct($nome, $cpf);

$this->salario = $salario;

}

//...

Feito isso, removeremos também os pontos em que definíamos o cargo dos funcionários no arquivo bonificacoes.php.

$umFuncionario = new Desenvolvedor(

'Vinicius Dias',

new CPF('123.456.789-10'),

1000

);

$umFuncionario->sobeDeNivel();

$umaFuncionaria = new Gerente(

'Patricia',

new CPF('987.654.321-10'),

3000

);

$umDiretor = new Diretor(

'Ana Paula', new CPF('123.951.789-11')

, 5000

);

Como não existe mais bonificação padrão, removeremos o método calculaBonificacao() da classe Funcionario. Sem ele, teremos que implementar a bonificação do Desenvolvedor, que passará a receber um valor fixo de 500.

class Desenvolvedor extends Funcionario

{

public function sobeDeNivel()

{

$this->recebeAumento($this->recuperaSalario() \* 0.75);

}

public function calculaBonificacao(): float

{

return 500.0;

}

}

Executando o arquivo bonificacoes.php, tudo continuará funcionando corretamente. Passaremos para a criação de um novo cargo, EditorVideo, que será armazenado no diretório "Funcionario" e fará parte do namespace Funcionario.

<?php

namespace Alura\Banco\Modelo\Funcionario;

class EditorVideo extends Funcionario

{

}

Em bonificacoes.php, criaremos uma nova instância de Editor, chamada $umEditor, com o nome "Paulo", o CPF "456.987.231-11" e o salário 1500. Além disso, incluiremos a bonificação de $umEditor pelo método adicionaBonificacao().

//...código omitido ...//

$umEditor = new EditorVideo('Paulo',

new CPF('456.987.231-11'),

1500

);

$controlador = new ControladorDeBonificacoes();

$controlador->adicionaBonificacaoDe($umFuncionario);

$controlador->adicionaBonificacaoDe($umaFuncionaria);

$controlador->adicionaBonificacaoDe($umDiretor);

$controlador->adicionaBonificacaoDe($umEditor);

echo $controlador->recuperaTotal();

Ao executarmos, receberemos um erro informando que o método calculaBonificacao() não foi definido na classe EditorVideo. Seria mais interessante se a IDE tivesse nos avisado da ausência desse método, já que todo Funcionario precisa de uma bonificação. Felizmente já sabemos fazer isso: se uma funcionalidade precisa existir para todas as classes, mas não tem uma implementação padrão, podemos lançar mão dos métodos abstratos. Nesse caso, em Funcionario, incluiremos o método abstrato calculaBonificacao() devolvendo um float.

abstract public function calculaBonificacao(): float;COPIAR CÓDIGO

Dessa forma, a classe EditorVideo passará a apresentar um erro informando a necessidade da implementação do método calculaBonificacao(), algo que pode ser feito automaticamente com o atalho "Alt + Enter". Em seguida, definiremos que a bonificação devolvida é o valor fixo 600.

class EditorVideo extends Funcionario

{

public function calculaBonificacao(): float

{

return 600;

}

}

A ideia desse exercício é nos aprofundarmos um pouco mais nas classes e métodos abstratos. Sendo assim, vamos recapitular o que fizemos até agora. Sabemos que todo funcionário de uma empresa tem um cargo. Portanto, podemos afirmar que ser um funcionário de uma empresa é algo mais *abstrato* do que, por exemplo, ser um diretor ou gerente de uma empresa; por sua vez, também podemos afirmar que os cargos são conceitos mais *concretos* dentro de uma empresa do que simplesmente ser um funcionário.

No nosso sistema, a classe Funcionario é um conceito, e não está pronta/apta para ser utilizada como objeto. Justamente por isso a chamamos de abstrata. Já um método abstrado é uma indicação de que aquela implementação é necessária em todas as classes que também representem um funcionário, ou seja, as classes filhas, mas não existe uma implementação padrão desse método.

Sempre que tivermos a palavra abstract no código, sabemos que ela está relacionada a herança. No caso, alguma classe precisa estender de Funcionario para que suas características façam sentido. Tais classes, por sua vez, precisarão implementar os métodos abstratos da classe base/mãe.

Isso garante que, ao cessarmos uma instância de Funcionario no método adicionaBonificacaoDe(), teremos também acesso ao método calculaBonificacao(). Se removermos a implementação abstrata de calculaBonificacao() da classe Funcionario e tentarmos acessar esse método em adicionaBonificacaoDe(), a própria IDE nos indicará que será impossível encontrá-lo.

O PHP funciona de maneira diferente de outras linguagens estritamente/estaticamente tipadas, pois chamará o método calculaBonificacao() mesmo que o objeto recebido em adicionaBonificacaoDe() não seja do tipo Funcionario. Se o método existir, ele será encontrado; do contrário, incorreremos em um erro.

Da mesma forma que Funcionario, faz sentido transformarmos a classe Pessoa em abstrata - afinal, em nosso sistema, temos funcionários (com seus respectivos cargos) ou titulares de uma conta.

abstract class Pessoa

{

protected $nome;

private $cpf;

public function \_\_construct(string $nome, CPF $cpf)

{

$this->validaNomeTitular($nome);

$this->nome = $nome;

$this->cpf = $cpf;

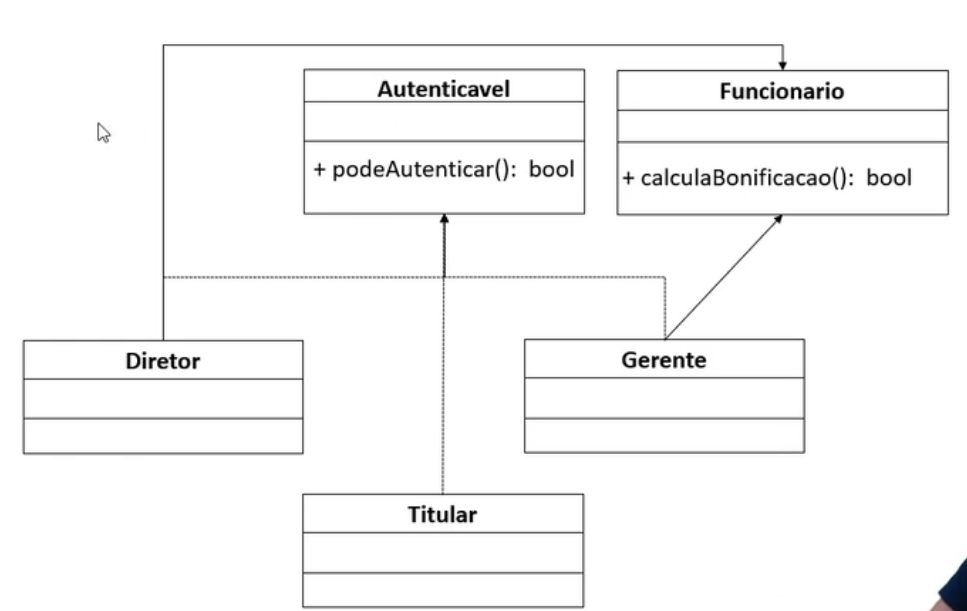
}

//...

Repare que uma classe pode ser abstrata mesmo que ela não tenha métodos abstratos. Agora que fizemos uma revisão dos conceitos de classes e métodos abstratos, não deixe de fazer os exercícios e expor as suas dúvidas no fórum!

**Interface**

Nosso objetivo agora é mantermos a herança de Funcionario em Gerente e Diretor, mas encontrarmos uma forma de Titular também conseguir acesso ao método podeAutenticar().



A ideia é recebermos, no nosso Autenticador, um objeto do tipo Autenticavel. Inclusive, alteraremos o nome do parâmetro recebido de $diretor para $autenticavel. No método, verificaremos se $autenticavel pode se autenticar com a $senha que é passada. Em caso positivo, ele será logado; do contrário, informaremos que a senha está incorreta.

class Autenticador

{

public function tentaLogin(Autenticavel $autenticavel, string $senha): void

{

if ($autenticavel->podeAutenticar($senha)) {

echo "Ok. Usuário logado no sistema";

} else {

echo "Ops. Senha incorreta.";

}

}

}

No vídeo anterior, quando conversamos sobre o problema da herança múltipla, vimos que surgem conflitos quando uma classe tenta herdar implementações diferentes de um mesmo método a partir de heranças diferentes. Mas e se, ao invés de uma implementação, tivermos somente um método abstrato, ou seja, o conceito do método?

Para testarmos, criaremos no diretório "Modelo" uma classe Autenticavel. Note que, na tela de criação de classes do PhpStorm, existe uma opção "Template" que, além de "Class", oferece também as opções "Interface" e "Trait". Como ainda não sabemos o que significam, continuaremos normalmente.

Sabemos que essa classe precisa de um método podeAutenticar(), mas não queremos implementá-lo. Sendo assim, vamos criá-lo como abstrato e tornar também a classe Autenticável abstrata.

abstract class Autenticavel

{

abstract public function podeAutenticar(string $senha): bool;

}

Com isso temos uma classe na qual todos os métodos são abstratos, certo? Mesmo assim, não conseguiremos fazer com que Diretor herde tanto de Funcionario quanto de Autenticavel, com o próprio PhpStorm nos indicando essa impossibilidade. Queremos uma forma de utilizarmos a assinatura do método podeAutenticar(), sem necessariamente recebermos essa classe, algo que é possível por meio de uma **interface**.

O conceito de interface representa, basicamente, uma classe abstrata com todos os seus métodos abstratos. Portanto, ao invés de chamarmos nossa Autenticavel de classe abstrata, vamos defini-la como interface. Como por definição os métodos de uma interface são abstratos, não precisaremos incluir a palavra reservada abstract na assinatura de podeAutenticar().

interface Autenticavel

{

public function podeAutenticar(string $senha): bool;

}

Feito isso, poderemos fazer com que nosso Diretor, além de estender de Funcionario, implemente a interface Autenticavel, algo que é feito com a palavra implements.

<?php

namespace Alura\Banco\Modelo\Funcionario;

use Alura\Banco\Modelo\Autenticavel;

class Diretor extends Funcionario implements Autenticavel

{

public function calculaBonificacao(): float

{

return $this->recuperaSalario() \* 2;

}

public function podeAutenticar(string $senha): bool

{

return $senha === '1234';

}

}

Uma classe pode implementar quantas interfaces forem necessárias, sem limitação. Por exemplo, o PHP possui uma interface \JsonSerializable que nos obriga a implementar o seu método jsonSerialize().

class Diretor extends Funcionario implements Autenticavel, \JsonSerializable

{

public function calculaBonificacao(): float

{

return $this->recuperaSalario() \* 2;

}

public function podeAutenticar(string $senha): bool

{

return $senha === '1234';

}

public function jsonSerialize()

{

// TODO: Implement jsonSerialize() method.

}

}

Feito esse teste, removeremos a interface \JsonSerializable do nosso código e implementaremos nossa nova interface também na classe Gerente. Com o atalho "Alt + Enter", conseguiremos inserir automaticamente o esqueleto do método podeAutenticar().

namespace Alura\Banco\Modelo\Funcionario;

use Alura\Banco\Modelo\Autenticavel;

class Gerente extends Funcionario implements Autenticavel

{

public function calculaBonificacao(): float

{

return $this->recuperaSalario();

}

public function podeAutenticar(string $senha): bool

{

// TODO: Implement podeAutenticar() method.

}

}

No caso do Gerente, a implementação do método simplesmente retornará uma $senha com o valor 4321.

public function podeAutenticar(string $senha): bool

{

return $senha === '4321';

}

Repetiremos o processo com a classe Titular, dessa vez retornando uma $senha de valor abcd.

namespace Alura\Banco\Modelo\Conta;

use Alura\Banco\Modelo\Autenticavel;

use Alura\Banco\Modelo\CPF;

use Alura\Banco\Modelo\Endereco;

use Alura\Banco\Modelo\Pessoa;

class Titular extends Pessoa implements Autenticavel

{

private $endereco;

public function \_\_construct(CPF $cpf, string $nome, Endereco $endereco)

{

parent::\_\_construct($nome, $cpf);

$this->endereco = $endereco;

}

public function getEndreco(): Endereco

{

return $this->endereco;

}

public function podeAutenticar(string $senha): bool

{

return $senha === 'abcd';

}

}

No Autenticador, passaremos a importar a classe Autenticavel, fazendo com que o PhpStorm identifique a presença do método podeAutenticar().

namespace Alura\Banco\Service;

use Alura\Banco\Modelo\Autenticavel;

class Autenticador

{

public function tentaLogin(Autenticavel $autenticavel, string $senha): void

{

if ($autenticavel->podeAutenticar($senha)) {

echo "Ok. Usuário logado no sistema";

} else {

echo "Ops. Senha incorreta.";

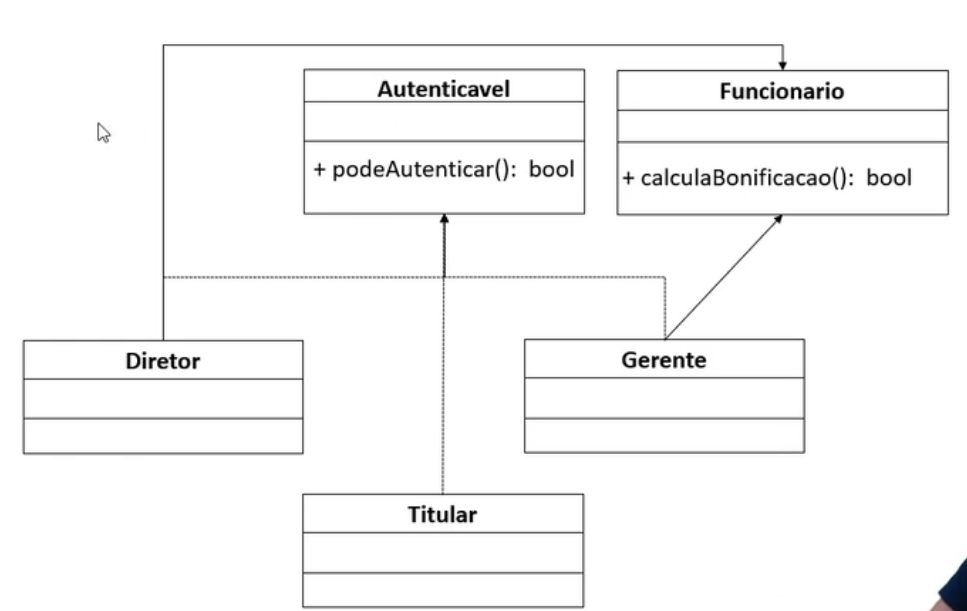
}

}

}

Assim conseguiremos receber, no método tentaLogin(), um objeto do tipo Autenticavel e que possui o método podeAutenticar(), obstante o fato de ser um Diretor, Gerente, Titular ou qualquer outra quase. Quando implementamos uma interface, estamos nos comprometendo a implementar os métodos definidos nela.

Por exemplo, se incluirmos na interface Autenticavel um método teste(), por exemplo, seremos obrigados a implementá-lo nas classes Diretor, Gerente e Titular. Utilizando a interface, conseguimos chegar em algo próximo de uma herança múltipla, de maneira semelhante ao desenho que apresentamos no início do vídeo.



Aqui as linhas pontilhadas representam a implementação de uma interface. O desenho também é *parecido* com o **diagrama de classes da UML**, mas não é fiel a um. Aqui na Alura existem treinamentos de UML caso você queira entender esse conceito.

Agora que implementamos a interface e temos um sistema completo, podemos testar a autenticação executando o arquivo autenticacao.php.

use Alura\Banco\Modelo\CPF;

use Alura\Banco\Modelo\Funcionario\Diretor;

use Alura\Banco\Service\Autenticador;

require\_once 'autoload.php';

$autenticador = new Autenticador();

$umDiretor = new Diretor(

'João da Silva',

new CPF('123.456.789-10'),

10000

);

$autenticador->tentaLogin($umDiretor, '1234');

Como retorno, teremos a mensagem "Ok. Usuário logado no sistema", indicando que tudo ocorreu corretamente. Se substituirmos o Diretor por um Gerente, receberemos a mensagem informando que a senha está incorreta. Para corrigirmos isso, passaremos a senha correta, que é 4321.

<?php

use Alura\Banco\Modelo\CPF;

use Alura\Banco\Modelo\Funcionario\Diretor;

use Alura\Banco\Modelo\Funcionario\Gerente;

use Alura\Banco\Service\Autenticador;

require\_once 'autoload.php';

$autenticador = new Autenticador();

$umDiretor = new Gerente(

'João da Silva',

new CPF('123.456.789-10'),

10000

);

$autenticador->tentaLogin($umDiretor, '4321');

Assim a mensagem de sucesso voltará a ser exibida. Da mesma forma, podemos passar um Titular, sem nos esquecermos de corrigir os parâmetros do construtor.

use Alura\Banco\Modelo\Conta\Titular;

use Alura\Banco\Modelo\CPF;

use Alura\Banco\Modelo\Endereco;

use Alura\Banco\Modelo\Funcionario\Diretor;

use Alura\Banco\Modelo\Funcionario\Gerente;

use Alura\Banco\Service\Autenticador;

require\_once 'autoload.php';

$autenticador = new Autenticador();

$umDiretor = new Titular(

new CPF(

'123.456.789-10'),

'João da Silva',

new Endereco(

'',

'',

'',

'')

);

$autenticador->tentaLogin($umDiretor, '4321');

Como estamos passando uma senha inválida, receberemos a mensagem de erro:

Ops. Senha incorreta.

Agora temos uma forma de autenticação funcional para o Diretor, Gerente ou Titular, pois todos eles implementam a interface Autenticavel. O conceito de interface é bem simples, mas a sua utilização é muito importante na programação orientada a objetos. Inclusive, existe uma premissa no mundo da orientação a objetos dizendo que sempre devemos programar para interfaces ou abstrações, e nunca para as implementações.

É interessante que nossas funcionalidades dependam de classes abstratas ou interfaces, pois isso torna nosso sistema mais extensível e maleável. Como exemplo, podemos facilmente criar um novo tipo de cliente, que não é titular de uma conta mas pegou empréstimo no banco, e torná-lo autenticável.

Vamos recapitular? Interfaces são basicamente classes abstratas que possuem somente métodos abstratos, o que nos permite a implementação de múltiplas interfaces sem problemas, já que evitam o problema de herdar dois métodos "iguais" de classes diferentes. Esse é um tema denso, portanto não deixe de expôr as suas dúvidas no fórum e de fazer os exercícios.

No próximo e último capítulo deste treinamento conheceremos mais alguns conceitos interessantes da orientação a objetos em PHP!

**Exibindo como string**

Chegamos ao capítulo final dessa segunda parte do treinamento de introdução à orientação a objetos com o PHP. Nesse momento, nos chegou uma nova demanda pedindo a preparação de um relatório com os endereços que temos cadastrados. Começaremos pensando em uma maneira de exibirmos esses endereços.

Criaremos na raiz do projeto um arquivo de testes enderecos.php no qual importaremos o autoload e criaremos os objetos $umEndereco, com as informações 'Petrópolis', 'bairro qualquer', 'Minha rua', '71b', e $outroEndereco com as informações 'Rio', 'Centro', 'Uma rua aí', '50'.

<?php

use Alura\Banco\Modelo\Endereco;

require\_once 'autoload.php';

$umEndereco = new Endereco(

'Petrópolis',

'bairro qualquer',

'Minha rua',

'71b'

);

$outroEndereco = new Endereco(

'Rio',

'Centro',

'Uma rua aí',

'50'

);

Um formato interessante de exibirmos esses dados é "Rua, número, bairro, cidade". Para isso, poderíamos executar um echo e concatenar as chamadas de $umEndereco->recuperaRua(), $umEndereco->recuperaBairro() e assim por diante, tomando cuidado para formatar o texto corretamente. Parece trabalhoso e realmente é, quando na verdade só queremos exibir o endereço como string.

Pensando nisso, poderíamos ter na classe Endereco um método como formataEndereco() que nos devolvesse os dados cadastrados no formato desejado. A ideia é que, quando tentarmos acessar $umEndereco como uma string, por exemplo exibindo-o com o echo, recebamos as informações nesse formato.

<?php

use Alura\Banco\Modelo\Endereco;

require\_once 'autoload.php';

$umEndereco = new Endereco(

'Petrópolis',

'bairro qualquer',

'Minha rua',

'71b'

);

$outroEndereco = new Endereco(

'Rio',

'Centro',

'Uma rua aí',

'50'

);

echo $umEndereco;

Quando escrevemos o código dessa forma, o PhpStorm nos exibe uma mensagem indicando a ausência do método \_\_toString. Como comentado nos treinamentos anteriores, todos os métodos iniciados com \_\_ são conhecidos como "métodos mágicos" do PHP, e que em alguns momentos são chamados de forma automática.

Sendo assim, implementaremos na classe Endereco o método \_\_toString(), que retornará uma string e, no corpo, definirá a formatação do texto. Para isso, retornaremos primeiro $this->rua, seguido de $this->numero, $this->bairro e $this->cidade. Para nos precavermos em relação a erros, rodearemos cada um desses dados com chaves.

public function \_\_toString(): string

{

return "{$this->rua}, {$this->numero}, {$this->bairro}, {$this->cidade}";

}

Feito isso, se executarmos o arquivo endereco.php, teremos como retorno:

Minha rua, 71b, bairro qualquer, Petrópolis

Faremos um novo teste concatenando $umEndereco com . PHP\_EOL, de modo a pularmos uma linha, e em seguida incluindo um echo de $outroEndereco.

echo $umEndereco . PHP\_EOL;

echo $outroEndereco;

Como resultado, teremos:

Minha rua, 71b, bairro qualquer, Petrópolis

Uma rua aí, 50, Centro, Rio

Perceba que o método mágico \_\_toString() nos permite representar qualquer objeto como string. Lembrando que todos os métodos mágicos do PHP começam com \_\_, e existem vários deles. Justamente por essa convenção, a documentação da linguagem não recomenda a criação de métodos com \_\_.

Agora que conhecemos um novo método mágico, queremos acessar$umEndereco->rua diretamente, sem chamarmos o método recuperaRua(). Conversaremos sobre essa possibilidade no próximo vídeo.

**Acessando atributos livremente**

Nosso objetivo agora é acessarmos de forma livre os atributos dos objetos do tipo Endereco, e queremos fazer isso por meio de métodos mágicos. Quando digitamos \_\_ no PhpStorm, nos é exibida a lista de métodos mágicos disponíveis. Por exemplo, temos o \_\_call(), que é chamado quando tentamos executar um método que não existe ou que é privado; o \_\_clone(), usado para criar uma cópia de um objeto; \_\_debugInfo(), executado quando fazemos o var\_dump() de uma referência que aponta para uma instância; e o \_\_destruct(), o método destrutor que já conhecemos anteriormente.

O método que estamos buscando é o \_\_get(), que recebe como parâmetro uma string representando o nome do atributo que queremos acessar. Para provarmos isso, faremos um echo do $nomeAtributo recebido no método.

public function \_\_get(string $nomeAtributo)

{

echo $nomeAtributo;

exit();

}

Com isso, se incluirmos um $umEndereco->rua no nosso script endereco.php, o texto "rua" será exibido no console. A ideia é que, quando recebermos o nome do atributo rua, seja executado o método recuperaRua() - ou seja, queremos colocar a primeira letra do nome em maiúsculo e adicionar o texto recupera como prefixo.

Para transformarmos a primeira letra do nome recebido em maiúscula, podemos usar a função ucfirst() do PHP (de *upper case first*). Armazenaremos o retorno em uma variável $metodo e concatenaremos o seu conteúdo com a string recupera.

public function \_\_get(string $nomeAtributo)

{

$metodo = ucfirst($nomeAtributo);

$metodo = 'recupera' . $metodo;

}

Podemos, inclusive, colocar esses dois passos em uma só linha. Em seguida, retornaremos a chamada de $this->$metodo().

public function \_\_get(string $nomeAtributo)

{

$metodo = 'recupera' . ucfirst($nomeAtributo);

return $this->$metodo();

}

Em endereco.php, incluiremos o echo de $umEndereco->bairro e pararemos a execução do script com um exit().

require\_once 'autoload.php';

$umEndereco = new Endereco(

'Petrópolis',

'bairro qualquer',

'Minha rua',

'71b'

);

$outroEndereco = new Endereco(

'Rio',

'Centro',

'Uma rua aí',

'50'

);

echo $umEndereco->bairro;

exit();

Como retorno, teremos "bairro qualquer", indicando que tudo foi executado corretamente. Se alterarmos a chamada para $umEndereco->cidade, o retorno será "Petrópolis". Com isso, não mais precisaremos dos métodos de acesso.

Repare, entretanto, que a IDE não nos ajuda sugerindo tais atributos, mostrando apenas os métodos que definimos anteriormente. É possível contornar esse "problema" adicionando *annotations* (anotações) no código. No PhpStorm, podemos fazer isso digitando /\*\* antes da classe e pressioando "Enter", o que fará com que o esqueleto de uma *annotation* seja incluído automaticamente.

Adicionaremos, então, uma nova *annotation* chamada @property-read, que representa propriedades que só podem ser lidas. Em seguida, definiremos o tipo da propriedade (string) e passaremos o seu nome, no caso $cidade. Continuaremos repetindo esse processo até listarmos todas as propriedades de Endereco.

/\*\*

\* Class Endereco

\* @package Alura\Banco\Modelo

\* @property-read string $cidade

\* @property-read string $bairro

\* @property-read string $rua

\* @property-read string $numero

\*/

class Endereco

{

private $cidade;

private $bairro;

private $rua;

private $numero;

public function \_\_construct(string $cidade, string $bairro, string $rua, string $numero)

{

$this->cidade = $cidade;

$this->bairro = $bairro;

$this->rua = $rua;

$this->numero = $numero;

}

//

Terminadas as anotações, o PhpStorm passará a nos sugerir os atributos como opção de autocompletar - por exemplo, se digitarmos $umEndereco->r, a propriedade rua será sugerida. Além disso, não conseguiremos mais atribuir um valor a esse campo, por exemplo com $umEndereco->rua = '', pois o definimos como somente leitura.

Vamos recapitular? Nós implementamos o método mágico \_\_get(), que é chamado sempre tentamos acessar um atributo/propriedade que não existe ou é privado. Por meio dele, montamos uma lógica que coloca a primeira letra do nome desse atributo em letra maiúscula e adiciona a string recupera ao início, resultando no padrão que utilizamos nos nossos métodos de acesso (*getters*). Com isso, conseguimos o nome do método, a partir do qual conseguimos recuperar o conteúdo do atributo.

Para que as IDE nos ajude com sugestões de autocompletar, incluímos na classe Endereco anotações informando a existência de propriedades de leitura chamadas $cidade, $bairro, $rua e $numero.

Já começamos a entender os métodos mágicos e, como dito anteriormente, existem vários outros, como o \_\_invoke(), que é chamado qunado tentamos utilizar uma referência como se fosse uma função.

Para que você se aprofunde mais nesse assunto, fica o desafio de fazer com o método \_\_set() o mesmo que fizemos com o \_\_get(). Tal método recebe dois parâmetros: o nome do atributo e o valor que será atribuído a ele. Não se esqueça de deixar a sua solução no fórum do curso para que nossos instrutores e alunos possam corrigi-la e dar dicas de como aprimora-la!

Agora voltaremos ao assunto da herança. No nosso sistema a classe Endereco representa uma entidade o mais específica possível. Ou seja, não existiria, por exemplo, uma classe EnderecoDosEUA que herdasse dessa classe, adicionando uma nova informação. Pensando nisso, queremos impedir a herança dessa classe. Descobriremos se isso é possível no próximo vídeo.

**Impedindo a herança**

Existem alguns casos, ainda que raros, nos quais desejamos impedir a herança. Por exemplo, não queremos que nenhuma classe estenda de Endereco ou CPF, pois essas são classes definitivas e únicas. Para isso, precisamos informar ao PHP que, independentemente de como for a hierarquia até aquele ponto, a classe com que estamos trabalhando é a final - algo que é feito usando exatamente a palavra **final**.

final class CPF

{

private $numero;

public function \_\_construct(string $numero)

{

$numero = filter\_var($numero, FILTER\_VALIDATE\_REGEXP, [

'options' => [

'regexp' => '/^[0-9]{3}\.[0-9]{3}\.[0-9]{3}\-[0-9]{2}$/'

]

]);

if ($numero === false) {

echo "Cpf inválido";

exit();

}

$this->numero = $numero;

}

//

Quando informamos que uma classe final, a própria IDE adiciona um "pin" visual indicando que ela foi fixada e não pode mais ser herdada. Inclusive, se tentarmos estender CPF em algum ponto do código, o PhpStorm nem mesmo encontrará a classe. Mesmo passando todo o endereço de CPF, incluindo o seu namespace, teremos um erro indicando que não é possível herdar de uma classe que tem final na sua definição.

Repetiremos esse processo para a classe Endereco, impedindo que ela também seja herdada. Continuando nesse raciocínio, temos na classe Pessoa um método validaNomeTitular(), que renomearemos para validaNome() de modo a adequá-lo melhor à sua funcionalidade.

protected function validaNome(string $nomeTitular)

{

if (strlen($nomeTitular) < 5) {

echo "Nome precisa ter pelo menos 5 caracteres";

exit();

}

}

O método validaNome() pode ser usado também pelas classes filhas, mas não queremos que ele seja modificado, por exemplo criando em Funcionario um novo método validaNome() que não faz absolutamente nada. Para que isso se torne proibido, também podemos usar a palavra-chave final.

final protected function validaNome(string $nomeTitular)

{

if (strlen($nomeTitular) < 5) {

echo "Nome precisa ter pelo menos 5 caracteres";

exit();

}

}

A classe Pessoa continuará sendo herdada sem problemas, mas o método validaNome() não mais poderá ser sobrescrito. Com isso ganhamos segurança no nosso sistema de hierarquia de classes, controlando quais comportamentos podem ou não ser adicionados. Isso é bastante interessante, por exemplo, em situações nas quais criamos classes que serão utilizadas por várias outras pessoas.

**Traits**

Nesse capítulo falamos sobre algumas peculiaridades do PHP, como os métodos mágicos, e nesse vídeo bônus comentaremos sobre outra: as *traits*.

Em Endereco, estamos usando o método \_\_get() para pegarmos o nome de um atributo e o transformarmos no nome do método acessor desse mesmo atributo. Porém, não é só o Endereco que possui atributos privados que gostaríamos de acessar. Por exemplo, poderíamos ler o $nome e o $cpf da classe Pessoa, sem precisarmos chamar os métodos acessores desses atributos.

Uma solução para isso seria criarmos uma classe extra que servisse de herança para Endereco e Pessoa e possuísse o método desejado. Entretanto, estaríamos obrigando classes completamente diferentes a herdarem de uma terceira simplesmente para ganharmos acesso a um método, às vezes até mesmo impedindo uma classe mais importante de ser herdada, já que o PHP não permite a herança múltipla. Além disso, o método recuperaRua(), por exemplo, não poderia ser chamado de uma classe base, incorrendo em mais um problema com a herança.

Outra alternativa é simplesmente copiarmos a estrutura do método \_\_get() e colarmos na classe Pessoa, algo que sabemos que não é ideal. Seria interessante se existisse uma forma do próprio PHP copiar código de algum local e injetá-lo na classe desejada. O PHP na verdade possui tal funcionalidade, e ela se chama **trait**.

Para testarmos, criaremos no diretório "Modelo" uma nova classe "AcessoPropriedades" utilizando o template "Trait" disponibilizado pela IDE.

<?php

namespace Alura\Banco\Modelo;

trait AcessoPropriedades

{

}

Repare que a estrutura é bem parecida com a de uma classe, alterando somente a palavra class para trait. Removeremos a definição do método \_\_get() da classe Endereco e o passaremos para a trait que criamos.

trait AcessoPropriedades

{

public function \_\_get(string $nomeAtributo)

{

$metodo = 'recupera' . ucfirst($nomeAtributo);

return $this->$metodo();

}

}

Na classe Endereco, incluiremos a instrução use AcessoPropriedades.

final class Endereco

{

use AcessoPropriedades;

private $cidade;

private $bairro;

private $rua;

private $numero;

//

Na aula de *namespaces* comentamos que a palavra use poderia significar coisas diferentes dependendo de onde ela é colocada em uma arquivo. Nesse caso, como estamos colocando-a diretamente dentro de uma classe, quer dizer que estamos utilizando uma *trait*. Isso implica que o PHP irá "colar" o conteúdo da *trait* no local informado, como se estivéssemos fazendo um "copia e cola" dos seus métodos, permitindo acesso ao \_\_get().

Isso também é vantajoso pois podemos utilizar quantas *traits* forem necessárias. Isso não é uma herança pois o PHP executa processos diferentes por baixo dos panos. Da mesma forma que incluímos o use AcessoPropriedades na classe Endereco, podemos fazer o mesmo em Pessoa.

abstract class Pessoa

{

use AcessoPropriedades;

protected $nome;

private $cpf;

//

Com isso, podemos, por exemplo, criar um novo $desenvolvedor e tentar acessar a sua propriedade $desenvolvedor->nome.

$desenvolvedor = new Desenvolvedor('Vinicius', new CPF('123.456.789-10'), 2000);

$desenvolvedor->nome;

O próprio PhpStorm não nos indicará um erro, pois o método mágico \_\_get() existe na classe Pessoa, e todos os herdeiros dessa classe também terão acesso a ele. A utilização de *traits* é interessante, mas também perigosa. Devemos ponderar bastante a inclusão de métodos nas *traits* de modo a evitar confusões e problemas de manutenção.

**Organização de pastas**

Durante os treinamentos de Orientação a Objetos, nós organizamos nossas pastas utilizando uma estrutura bastante comum.

Existe uma pasta raiz para nosso código (src) que inclusive facilita que a gente siga à PSR-4.

Dentro dessa pasta raiz, temos sub pastas que identificam responsabilidades em nosso sistema

## **Arquitetura**

Organização de pastas em nosso código é um início bem simplório de um estudo sobre arquitetura. Como organizar nosso projeto é um estudo muito importante e existem vários livros, palestras e conteúdo em geral sobre isso

## Alternativas à nossa abordagem

É muito comum e recomendado por muitos que ao invés de separar as sub pastas por responsabilidade (Model, Service, etc.), separemos por conceitos de domínio, parecido com o que fizemos dentro da pasta Model. Estudos específicos de arquitetura e design de código abordam estes conceitos com muito mais detalhes.