

# PHPattern: Uma visão simplista sobre Design Pattern

# Quem sou eu?

- Ruan Sales - Mid Level Developer
- Acadêmico de Análise e Desenvolvimento de Sistemas
- Ênfase Back-End em PHP / Laravel



# Design Pattern



# Definição

Do inglês Design Patterns, podemos definir Padrões de Projeto como modelos de soluções para algum problema específico encontrado frequentemente dentro de um projeto de software. Eles servem como templates a serem aplicados para desenvolver uma solução para os problemas.

# Detalhes

Não são códigos implementados que podem ser copiados para outros softwares (na maioria das vezes), mas apenas a definição de sua aplicação. Por isso, eles funcionam nos mais variados tipos de escopos e para diversos fins. Você pode aplicar um Padrão de Projeto específico tanto em um jogo quanto em um site de vendas, tudo depende de sua necessidade.

Outra consideração é que Padrões de Projeto são independentes da linguagem usada. Eles funcionam na maioria das linguagens orientadas a objetos, no entanto, podem conter algumas diferenças em suas implementações – o que vai depender das funcionalidades disponíveis e das peculiaridades de cada uma dessas linguagens.

# Quais são os Design Patterns Existentes?

Nós podemos dividir os Patterns em 3 tipos bem definidos:

- 1. Padrões de Criações**
- 2. Padrões de Estruturas**
- 3. Padrões de Comportamentos**

# Padrões de Criações

Existem diversos padrões de criações, como por exemplo: **Factory Method**, **Abstract Factory**, **Builder**, **Prototype** e **Singleton**, porém, hoje nós iremos nos focar nos seguintes **Patterns**:

# Factory Method ou Método de Fábrica

## Para que serve ?

O Factory Method é um padrão criacional de projeto que fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.



# Prós

- Você pode ter certeza que os produtos que você obtém de uma fábrica são compatíveis entre si.
- Você evita um vínculo forte entre produtos concretos e o código cliente.
- Princípio de responsabilidade única. Você pode extrair o código de criação do produto para um lugar, fazendo o código ser de fácil manutenção.
- Princípio aberto/fechado. Você pode introduzir novas variantes de produtos sem quebrar o código cliente existente.

# Contra

- O código pode tornar-se mais complicado do que deveria ser, uma vez que muitas novas interfaces e classes são introduzidas junto com o padrão.

## Exemplo de Implementação Factory Method

```
//Criação da Classe Comida  
  
<?php  
  
class Comida  
{  
    public $tipo;  
}
```

```
//Criação da Classe Pizza

<?php

require_once "Comida.php";

class Pizza extends Comida
{
    public $sabor;
    public $ingredientes;
    public $valor;
}
```

```
//Criação ENUM de Pizzas
```

```
<?php
```

```
enum SaboresPizza: int {  
    case Calabresa = 1;  
    case QuatroQueijos = 2;  
}
```

```

//Criação Fábrica de Pizzas

<?php
require_once "ComidaFactory.php";
require_once "PizzaEnum.php";
require_once "Pizza.php";

class PizzaFactory implements ComidaFactory
{
    public $sabores = [
        1 => [
            'sabor' => 'Calabresa',
            'ingredientes' => [
                'molho de tomate',
                'queijo mussarela',
                'calabresa',
                'oregano'
            ]
        ],
        2 => [
            'sabor' => 'Quatro Queijos',
            'ingredientes' => [
                'molho de tomate',
                'queijo mussarela',
                'queijo prato',
                'queijo cheddar',
                'queijo porvolone'
            ]
        ]
    ];
}

```

```
//...

public function verificaId($id)
{
    return SaboresPizza::from($id);
}

public function criarComida($id): Pizza
{
    $sabor = $this->verificaId($id);
    $pizza = new Pizza();
    $data = $this->sabores[$sabor->value];
    $pizza->sabor = $data['sabor'];
    $pizza->ingredientes = $data['ingredientes'];
    $pizza->valor = 29.00;
    $pizza->tipo = "Pizza";

    return $pizza;
}
}
```

# Utilização Factory Method

```
//index.php

<?php

require_once "PizzaFactory.php";

$fabricaPizza = new PizzaFactory();

$produto = $fabricaPizza->criarComida(1);

/*
object(Pizza)#3 (4) {
    ["tipo"]=>
    string(5) "Pizza"
    ["sabor"]=>
    string(9) "Calabresa"
    ["ingredientes"]=>
    array(4) {
        [0]=>
        string(15) "molho de tomate"
        [1]=>
        string(16) "queijo mussarela"
        [2]=>
        string(9) "calabresa"
        [3]=>
        string(7) "oregano"
    }
    ["valor"]=>
    float(29)
}
*/
```

# Singleton

## Definição

O Singleton é um padrão de projeto criacional que permite a você garantir que uma classe tenha apenas uma instância, enquanto provê um ponto de acesso global para essa instância.



## Prós

- Você pode ter certeza que uma classe só terá uma única instância.
- Você ganha um ponto de acesso global para aquela instância.
- O objeto singleton é inicializado somente quando for pedido pela primeira vez.

## Contras

- Viola o princípio de responsabilidade única. O padrão resolve dois problemas de uma só vez.
- O padrão Singleton pode mascarar um design ruim, por exemplo, quando os componentes do programa sabem muito sobre cada um.
- O padrão requer tratamento especial em um ambiente multithreaded para que múltiplas threads não possam criar um objeto singleton várias vezes.

# Exemplo Singleton

```
<?php
class Conexao {
    public static $instance;

    private function __construct() {
        //
    }

    public static function getInstance() {
        if (!isset(self::$instance)) {
            self::$instance = new PDO('mysql:host=localhost;dbname=mydatabase', 'root', '123', array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8"));
            self::$instance->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
            self::$instance->setAttribute(PDO::ATTR_ORACLE_NULLS, PDO::NULL_EMPTY_STRING);
        }

        return self::$instance;
    }
}
```

# Padrão Comportamental

## Definição

Padrões comportamentais são voltados aos algoritmos e a designação de responsabilidades entre objetos.

# Strategy

## Definição

O Strategy é um padrão de projeto comportamental que permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.

# Exemplo Strategy

## Criação Classe Pai (Conta)

```
//Definição Classe Conta

<?php

class Conta
{
    public $agencia;
    public $saldo = 0;
    public $numeroConta;
    public $cliente;
}
```

## Interface

```
//Definição Interface para Contas

<?php

interface ContaBancariaInterface
{
    public function depositar($valor);
    public function sacar($valor);
    public function saldo();
}
```

# Conta Corrente

```
//Definição ContaCorrente

<?php

require_once "ContaBancariaInterface.php";
require_once "Conta.php";

class ContaCorrente extends Conta implements ContaBancariaInterface
{
    public $juros = 0.03;

    public function __construct($agencia, $numeroConta, $cliente, $saldo)
    {
        $this->agencia = $agencia;
        $this->numeroConta = $numeroConta;
        $this->cliente = $cliente;
        $this->saldo = $saldo;
    }

    public function depositar($valor)
    {
        $this->saldo+= $valor;
    }

    public function sacar($valor){
        $calculoSJuros = $valor * $this->juros;

        if($this->saldo < $valor) {
            echo "Saldo insuficiente, operação não realizada";
            exit;
        }

        return $this->saldo -= $valor + $calculoSJuros;
    }

    public function saldo(){
        echo "Saldo da Conta Corrente é: $this->saldo" . PHP_EOL;
    }
}
```

# Conta Poupança

```
//Definição ContaPoupança

<?php

require_once "ContaBancariaInterface.php";
require_once "Conta.php";

class ContaPoupanca extends Conta implements ContaBancariaInterface
{
    public function __construct($agencia, $numeroConta, $cliente, $saldo)
    {
        $this->agencia = $agencia;
        $this->numeroConta = $numeroConta;
        $this->cliente = $cliente;
        $this->saldo = $saldo;
    }

    public function depositar($valor)
    {
        $this->saldo+= $valor;
    }

    public function sacar($valor){
        if($this->saldo < $valor) {
            echo "Saldo insuficiente, operação não realizada";
            exit;
        }

        return $this->saldo -= $valor;
    }

    public function saldo(){
        echo "Saldo da Conta Poupança é: $this->saldo" . PHP_EOL;
    }
}
```



## Utilização (index.php)

```
<?php
require_once "ContaPoupanca.php";
require_once "ContaCorrente.php";
require_once "Conta.php";
$conta = new ContaPoupanca('0123', '2020', 'Ruan Sales', 75);
var_dump($conta) . PHP_EOL;
$conta->sacar(28);
$conta->saldo();

$conta2 = new ContaCorrente('1234', '09876', 'Danzel', 920);
var_dump($conta2) . PHP_EOL;
$conta2->sacar(200);
$conta2->saldo();
```

# Resultados

```
/*
object(ContaPoupanca)#1 (4) {
  ["agencia"]=>
  string(4) "0123"
  ["saldo"]=>
  int(75)
  ["numeroConta"]=>
  string(4) "2020"
  ["cliente"]=>
  string(10) "Ruan Sales"
}
Saldo da Conta Poupança é: 47
object(ContaCorrente)#2 (5) {
  ["agencia"]=>
  string(4) "1234"
  ["saldo"]=>
  int(920)
  ["numeroConta"]=>
  string(5) "09876"
  ["cliente"]=>
  string(6) "Danzel"
  ["juros"]=>
  float(0.03)
}
Saldo da Conta Corrente é: 714
*/
```

# Padrão de Projeto Estrutural

## Definição

Os padrões estruturais explicam como montar objetos e classes em estruturas maiores mas ainda mantendo essas estruturas flexíveis e eficientes.

# Facade

## Definição

O Facade é um padrão de projeto estrutural que fornece uma interface simplificada para uma biblioteca, um framework, ou qualquer conjunto complexo de classes.

# Exemplo Facade Pattern

## Classe Estoque

```
<?php

class Estoque
{
    public static function retornarPrecoDoProdutoPeloID($id){
        if($id == 10){
            return 1000;
        }else{
            return 0;
        }
    }
}
```

## Classe Pagamento

```
<?php

class Pagamento
{
    public static function pagarComCartao($valor){
        echo "Pagamento de R$ $valor com Cartão de Crédito" . PHP_EOL;
    }

    public static function pagarComBoleto($valor){
        echo "Pagamento de R$ $valor no Boleto Bancário" . PHP_EOL;
    }
}
```

## Classe Entrega

```
<?php  
  
class Entrega  
{  
    public $endereco;  
    public $cep;  
    public $transportadora;  
  
    public function calcularFrete(){  
        return 200;  
    }  
}
```

# Implementação Facade

```
<?php

require "../Estoque.php";
require "../Entrega.php";
require "../Pagamento.php";

class CompraFacade
{
    public static function finalizarCompra($idProduto, $endereco, $cep, $transportadora, $meioDePagamento){
        $valorDoProduto = Estoque::retornarPrecoDoProdutoPeloID($idProduto);
        $entrega = new Entrega();
        $entrega->endereco = $endereco;
        $entrega->cep = $cep;
        $entrega->transportadora = $transportadora;
        $valorDoFrete = $entrega->calcularFrete();
        $valorTotal = $valorDoProduto + $valorDoFrete;
        if($meioDePagamento == 1){
            Pagamento::pagarComCartao($valorTotal);
        }else{
            Pagamento::pagarComBoleto($valorTotal);
        }
    }
}
```



## Utilizando a Facade

```
<?php  
  
require "CompraFacade.php";  
  
CompraFacade::finalizarCompra(10, 'Rua do cliente', 'zep do cliente', 'transportadora', 1);  
  
/*  
Pagamento de R$ 1200 com Cartão de Crédito  
*/
```

# Obrigado por assistir a palestra!

- Repositório e Slides:  
<https://github.com/RuanSalles/HPatterns>

