Programa de Aceleração para Programadores em PHP

Lista de Exercícios

// Práctica Individual

****Atenção para cada exercício crie um arquivo com a nomenclatura: exercicionumerodoexercicio.php.**

90 - Você tem um objeto de Log que precisa ser acessado globalmente e precisa ter apenas uma única instância desse objeto. Crie uma classe Logs que implemente o design pattern **Singleton** garantindo uma instância única desse objeto, e além disso terá funções responsáveis pela gerência do arquivo de log como por exemplo: escreverlog, buscarLog e exibirlog; Em seguida crie uma classe de testes que vai instanciar a classe log, crie 5 diretórios distintos da sua preferência. Em seguida crie um arquivo chamado log-dataatualhoraatual.txt e grave em cada diretorio, a cada arquivo criado insira no arquivo de log essa informação de arquivo criado na respectiva data e hora. Por fim, exiba o arquivo de log de cada diretório criado.

91 - Dado o código a seguir, temos uma solução de desconto para 2 tipos de carros obedecendo condicionais, porém dado nova feature do projeto precisamos adicionar o modelo de carro Uno, no qual terá os seguintes descontos: Se for 4x4 receberá 10% OFF, se for esportivo 15% OFF. Usando o design pattern Strategy, resolva o problema do código a seguir. Perceba que cada $car é uma string e não um objeto complexo, transforme cada tipo em uma classe e aplique para cada carro seu respectivo desconto com suas respectivas caracteristicas. Obs todos os descontos são em porcentagem e o método geradorDeCupom precisa ser refatorado.

```
function geradorDeCupom($car)
{
  $discount = 0;
  $is4x4 = false;
  $ret = false;
  $esportivo = false;


  if($car == "bmw")
```

```
  {
    if(!$is4x4) {$discount += 5;}

    if($ret) {$discount += 7;}

    if($esportivo) {$discount += 9;}
  }
  else if($car == "mercedes")
  {
    if(!$is4x4) {$discount += 10;}

    if($ret) {$discount += 18;}

    if($esportivo) {$discount += 19;}
  }

  return $cupoun = "Cupom {$discount}% off para passar no pedágio";
}


echo cupounGenerator("bmw");
echo cupounGenerator("mercedes");
```

92 - Dado o código a seguir. Um determinado cliente deseja transferir alguns dados do array para JSON, XML ou uma string separada por vírgulas. Em cada caso, temos que tratar os dados de uma maneira diferente e produzir o resultado na forma de uma string. O uso deste componente deve ser o mais fácil possível. Sendo assim, usando o design pattern Strategy construa uma interface chamada Saida e implemente em cada um dos tipos possíveis de saída. Em seguida crie uma classe ClienteTransfer de dentro do __construct delegue a baseado no tipo de saída a classe responsável pelo retorno.

93 - Dado o código a seguir, usando o builder, resolva o problema desse construtor gigante:

```
class PasswordGenerator
{
private $upperIncluded;
private $lowerIncluded;
private $digitIncluded;
```

```
private $punctuationIncluded;
private $ambiguousExcluded;
private $minUpper;
private $minLower;
private $minDigits;
private $minPunctuation;
// More fields here ...
public __construct(
bool $upperIncluded = true,
bool $lowerIncluded = true,
bool $digitIncluded = true,
bool $punctuationIncluded = true,
bool $ambiguousExcluded = true;
int $minUpper = 0,
int $minLower = 0,
int $minDigits = 0,
int $minPunctuation = 0,
// And so on ...
) {
$this->upperIncluded = $upperIncluded;
$this->lowerIncluded = $lowerIncluded;
$this->digitIncluded = $digitIncluded;
$this->punctuationIncluded = $punctuationIncluded;
$this->ambiguousExcluded = $ambiguousExcluded;
$this->minUpper = $minUpper;
$this->minLower = $minLower;
$this->minDigits = $minDigits;
$this->minPunctuation = $minPunctuation;
// More initialization code here ...
}
// More methods here ...
}
```

94 - Um aplicativo da web pode suportar diferentes mecanismos de renderização ao mesmo tempo, mas apenas se suas classes forem independentes das classes concretas dos mecanismos de renderização. Consequentemente, os objetos do aplicativo devem se comunicar com objetos de modelo apenas por meio de suas interfaces abstratas. Seu código não deve criar os objetos de modelo diretamente, mas delegar sua criação a

objetos de fábrica especiais. Finalmente, seu código também não deve depender dos objetos de fábrica, mas, em vez disso, deve trabalhar com eles por meio da interface de fábrica abstrata. Por isso, implemente um código com Abstract Factory, capaz de fornecer uma infraestrutura para criar vários tipos de modelos para diferentes elementos de uma página da web.

95 - Utilizando o principio SRP, reescreva o código abaixo:

```php
<?php
class Report
{
    public function getTitle()
    {
        return 'Report Title';
    }

    public function getDate()
    {
        return '2018-01-22';
    }

    public function getContents()
    {
        return [
            'title' => $this->getTitle(),
            'date' => $this->getDate(),
        ];
    }

    public function formatJson()
    {
        return json_encode($this->getContents());
    }
}
```

96 - Analise o código abaixo e reescreva ele para que ele seja capaz de atender ao princípio da segregação de interface, de modo a observar que alguns métodos que estão sendo forçados pela interface não recebem nenhuma implementação nas classes que são chamados, criando assim funções que nunca serão usadas “código morto”.

Segre da forma necessária para resolver esse problema.

```
<?php
interface Aves
{
    public function andar();
    public function voar();
    public function nadar();
}

class Pato implements Aves
{
    public function voar()
    {
        //lógica
    }

    public function nadar()
    {
        //lógica
    }

    public function andar()
    {
        //lógica
    }
}
```

```
class Pinguim implements Aves
{
    public function voar()
    {
        //lógica
    }

    public function nadar()
    {
        //lógica
    }

    public function andar()
    {
        //lógica
    }
}

class Andorinha implements Aves
{
    public function voar()
    {
        //lógica
    }

    public function nadar()
    {
        //lógica
    }

    public function andar()
    {
        //lógica
    }
}
```

97 - Usando o princípio da inversão de dependência resolva o código a seguir, observe onde você tem uma forte dependência e inverta ela:

```php
<?php
class Email
{
    public function enviar($mensagem)
    {
        //lógica
    }
}

class Notificacao
{
    public __construct()
    {
        $this->mensagem = new Email;
    }

    public function enviar($mensagem)
    {
        $this->mensagem->enviar($mensagem)
    }
}
```

98 - Utilizando uma classe PayPal para pegar o valor. Deste modo, estamos criando diretamente o objeto da classe PayPal e pagando via PayPal. Você tem esse código espalhado em vários lugares. Portanto, podemos ver que o código está usando o $paypal->sendPayment('amount here'); Dado uma atualização, o PayPal mudou o nome do método da API de **sendPayment** para **payAmount**. Isso deve indicar claramente um problema para aqueles que têm usado o método **sendPayment**. Especificamente, precisamos alterar todas as chamadas de método **sendPayment** para **payAmount**. Sabendo disso refatore o código abaixo usando o padrão Adpater. Crie uma interface

wrapper que torne isso possível. Não faça alterações na biblioteca de classes externas, afinal lembre-se que você não tem controle sobre ela e ela pode mudar a qualquer momento.

```php
<?php
class PayPal {

    public function __construct() {
        // Your Code here //
    }

    public function sendPayment($amount) {
        // Paying via Paypal //
        echo "Paying via PayPal: ". $amount;
    }
}

$paypal = new PayPal();
$paypal->sendPayment('2629');
```