Design Patterns (Padrões de Projeto)

Existem diversas formas de estruturar o código e o projeto da sua aplicação web e você pode gastar muito ou pouco esforço pensando na sua arquitetura. Mas geralmente é uma boa ideia seguir à padrões comuns, pois isso irá fazer com que seu código seja mais fácil de manter e de ser entendido por outros desenvolvedores.

- Padrões de Arquitetura na Wikipedia
- Padrões de Design de Software na Wikipedia
- Coleção de exemplos de implementação

Factory (Fábrica)

Um dos padrões de design mais utilizados é o padrão "Factory" (Fábrica). Através dele uma classe simplesmente cria o objeto que você gostaria de usar. Considere o seguinte exemplo desse padrão de design:

```
<?php
class Automobile
{
    private $vehicle_make;
    private $vehicle_model;

    public function __construct($make, $model)
    {
        $this->vehicle_make = $make;
        $this->vehicle_model = $model;
    }

    public function get_make_and_model()
    {
        return $this->vehicle_make . ' ' . $this->vehicle_model;
    }
}

class AutomobileFactory
{
    public static function create($make, $model)
    {
        return new Automobile($make, $model);
    }
}

// Solicita a "Factory" que crie o objeto Automobile
$veyron = AutomobileFactory::create('Bugatti', 'Veyron');

print_r($veyron->get_make_and_model()); // imprime "Bugatti Veyron"
```

Esse código usa uma "Factory" para criar o objeto do tipo "Automobile". Existem dois possíveis benefícios para criar seu código dessa forma, o primeiro é que se você precisar mudar, renomear ou substituir a classe Automobile futuramente você pode fazer e só terá que modificar o código na "Factory", em vez de em todos os lugares do seu projeto onde você usa a classe Automobile. O segundo benefício possível é que caso a criação do objeto seja um processo complicado, você pode executar todo esse trabalho na factory, em vez de repetí-lo toda vez que precisar criar uma nova instância da classe.

Usar o padrão de “Factory” não é sempre necessário (ou esperto). O código de exemplo usado aqui é tão simples que essa “Factory” estaria simplesmente adicionando complexidade indesejada. Entretanto se você estiver realizando um projeto um pouco maior ou mais complexo você pode se salvar de muitos problemas com o uso do padrão “Factory”.

- Padrão “Factory” na Wikipedia

Singleton (Única Instancia)

Quando arquitetando uma aplicação web, é comum fazer sentido tanto conceitualmente quanto arquitetonicamente permitir o acesso a somente uma instância de uma classe em particular, o padrão “Singleton” nos permite realizar essa tarefa.

```
<?php
class Singleton
{
    /**
     * Retorna uma instância única de uma classe.
     *
     * @staticvar Singleton $instance A instância única dessa classe.
     *
     * @return Singleton A Instância única.
     */
    public static function getInstance()
    {
        static $instance = null;
        if (null === $instance) {
            $instance = new static();
        }

        return $instance;
    }

    /**
     * Construtor do tipo protegido previne que uma nova instância da
     * Classe seja criada através do operador `new` de fora dessa classe.
     */
    protected function __construct()
    {
    }

    /**
     * Método clone do tipo privado previne a clonagem dessa instância
     * da classe
     *
     * @return void
     */
    private function __clone()
    {
    }

    /**
     * Método unserialize do tipo privado para prevenir a desserialização
     * da instância dessa classe.
     *
     * @return void
     */
    private function __wakeup()
    {
    }
```

```
}

class SingletonChild extends Singleton
{
}

$obj = Singleton::getInstance();
var_dump($obj === Singleton::getInstance());             // bool(true)

$anotherObj = SingletonChild::getInstance();
var_dump($anotherObj === Singleton::getInstance());      // bool(false)

var_dump($anotherObj === SingletonChild::getInstance()); // bool(true)
```

O código acima implementa o padrão “Singleton” usando uma variável *estática* e o método estático de criação `getInstance()`. Note o seguinte:

- O construtor `__construct` é declarado como protegido para prevenir que uma nova instância seja criada fora dessa classe pelo operador `new`.
- O método mágico `__clone` é declarado como privado para prevenir a clonagem dessa instância da classe pelo operador `clone`.
- O método mágico `__wakeup` é declarado como privado para prevenir a desserialização de uma instância dessa classe pela função global `unserialize()`.
- Uma nova instância é criada via late static binding no método de criação `getInstance()` declarado como estático. Isso permite a criação de classes “filhas” da classe Singleton no exemplo

O padrão Singleton é útil quando você precisa garantir que somente uma instância da classe seja criada em todo o ciclo de vida da requisição em uma aplicação web. Isso tipicamente ocorre quando você tem objetos globais (tais como uma classe de Configuração) ou um recurso compartilhado (como uma lista de eventos).

Você deve ser cauteloso quando for usar o padrão “Singleton” já que pela sua própria natureza ele introduz um estado global na sua aplicação reduzindo a possibilidade de realização de testes. Na maioria dos casos Injeção de Dependências pode (e deve) ser usado no lugar de uma classe do tipo Singleton. Usar Injeção de Dependências significa não introduzir acoplamento desnecessário no design da sua aplicação, já que o objeto usando o recurso global ou compartilhado não necessita de conhecimento sobre uma classe concretamente definida.

- Padrão Singleton na Wikipedia

Strategy (Estratégia)

Com o padrão “Strategy” (Estratégia) voce encapsula famílias específicas de algoritimos permitindo com que a classe cliente responsável por instanciar esse algoritimo em particular não necessite de conhecimento sobre sua implementação atual. Existem várias variações do padrão “Strategy” o mais simples deles é apresentado abaixo:

O primeiro bloco de código apresenta uma familia de algorítimos; você pode querer uma array serializado, um JSON ou talvez somente um array de dados:

```
<?php
```

```
interface OutputInterface
{
    public function load();
}

class SerializedArrayOutput implements OutputInterface
{
    public function load()
    {
        return serialize($arrayOfData);
    }
}

class JsonStringOutput implements OutputInterface
{
    public function load()
    {
        return json_encode($arrayOfData);
    }
}

class ArrayOutput implements OutputInterface
{
    public function load()
    {
        return $arrayOfData;
    }
}
```

Através do encapsulamento do algoritimo acima você está fazendo seu código de forma limpa e clara para que outros desenvolvedores possam facilmente adicionar novos tipos de saída sem que isso afete o código cliente.

Você pode ver como cada classe concreta ‘output’ implementa a OutputInterface - isso serve a dois propósitos, primeiramente isso prevê um simples contrato que precisa ser obedecido por cada implementação concreta. Segundo, através da implementação de uma interface comum você verá na próxima seção que você pode utilizar Indução de Tipo para garantir que o cliente que está utilizando esse comportamento é do tipo correto, nesse caso ‘OutputInterface’.

O próximo bloco de código demonstra como uma classe cliente relizando uma chamada deve usar um desses algorítimos e ainda melhor definir o comportamento necessário em tempo de execução:

```
<?php
class SomeClient
{
    private $output;

    public function setOutput(OutputInterface $outputType)
    {
        $this->output = $outputType;
    }

    public function loadOutput()
    {
        return $this->output->load();
    }
}
```

A classe cliente tem uma propriedade private que deve ser definida em tempo de execução e ser do tipo ‘OutputInterface’ uma vez que essa propriedade é definida uma chamada a loadOutput() irá chamar o método load() na classe concreta do tipo ‘output’ que foi definida.

```
<?php
$client = new SomeClient();

// Quer um array?
$client->setOutput(new ArrayOutput());
$data = $client->loadOutput();

// Quer um JSON?
$client->setOutput(new JsonStringOutput());
$data = $client->loadOutput();
```

- Padrão “Strategy” na Wikipedia

Front Controller

O padrão front controller é quando você tem um único ponto de entrada para sua aplicação web (ex. index.php) que trata de todas as requisições. Esse código é responsável por carregar todas as dependências, processar a requisição e enviar a resposta para o navegador. O padrão Front Controller pode ser benéfico pois ele encoraja o desenvolvimento de um código modular e provê um ponto central no código para inserir funcionalidades que deverão ser executadas em todas as requisições (como para higienização de entradas).

- Padrão Front Controller na Wikipedia

Model-View-Controller

O padrão model-view-controller (MVC) e os demais padrões relacionados como HMVC and MVVM permitem que você separe o código em diferentes objetos lógicos que servem para tarefas bastante específicas. Models (Modelos) servem como uma camada de acesso aos dados onde esses dados são requisitados e retornados em formatos nos quais possam ser usados no decorrer de sua aplicação. Controllers (Controladores) tratam as requisições, processam os dados retornados dos Models e carregam as views (Visões) para enviar a resposta. E as views são templates de saída (marcação, xml, etc) que são enviadas como resposta ao navegador.

O MVC é o padrão arquitetônico mais comumente utilizado nos populares Frameworks PHP.

Leia mais sobre o padrão MVC e os demais padrões relacionados:

- MVC
- HMVC
- MVVM

PHP Singleton - Aplicando o padrão de projeto na prática

Veja neste artigo como aplicar o padrão de projeto Singleton na prática na linguagem PHP.

Artigos PHP PHP Singleton - Aplicando o padrão de projeto na prática

Introdução aos Padrões de Projetos

Este conceito de padrões de projeto é antigo e não teve origem na área de informática, muito pelo contrário, este foi herdado da Arquitetura onde se viu a necessidade de seguir padrões.

Porque seguir padrões se desenvolvo de uma forma totalmente diferente e sempre dá certo? A grande questão em seguir padrões é uniformizar o desenvolvimento e permitir que qualquer outro analista possa olhar seu código e entende-lo. Fazer sua aplicação funcionar está muito longe do sinônimo para software de qualidade, imagine se os fabricantes de automóveis julgassem um carro como "perfeito" apenas ligando o mesmo e ouvindo o barulho do motor.

Mesmo que você trabalhe por conta própria, sem nenhuma equipe, é ideal a utilização de padrões, isso porque a manutenção do software pode-se se tornar onerosa se nem ao menos você entender o que você mesmo fez a 5 anos naquele determinado trecho de código.

Para quem já trabalhou em grandes projetos, com toda uma estrutura de multicamadas, SVN, testes e etc., sabe muito bem que é indispensável à padronização de toda a aplicação, e quando falamos de padronização estamos falando também de cores, fontes, títulos, tamanhos de telas e endentações.

Que padrões usar?

De fato não existe um padrão de projeto ideal, na verdade muitos projetos utilizam diversos padrões para tentar atingir um nível de excelência, um padrão que é implementado em quase todo projeto é o **SINGLETON**, que daremos foco neste artigo. Existem ainda muitos outros, e na verdade são tantos que algumas vezes acabam confundindo a equipe sobre qual padrão está sendo aplicado naquele momento. A maioria dos frameworks, se não todos, utilizam algum framework como base: ZendFramework, Hibernate, Spring e etc.

No início a mudança de paradigma para que não está acostumado com padrões é um verdadeiro choque, mas depois de assimilado todo o padrão e a boa utilização do mesmo, com certeza é um caminho sem volta, isso porque o aumento da produtividade é consideravelmente alta e satisfatória.

Aplicando o padrão Singleton

Este padrão de projeto faz com que objetos sejam criados apenas uma vez na memória. Significa que sempre que o objeto por chamado em qualquer parte do código, o padrão de projeto Singleton se encarregará de retornar sempre a mesma instância do objeto, e caso este não existe, então ele solicitará a criação de um novo pelo construtor da classe. Podemos dizer que este é um dos projetos mais utilizados em qualquer projeto, isso porque ele previne um possível *stack overflow* (estouro de pilha), ainda mais em aplicações Web.

O código abaixo é a aplicação do Singleton em um caso muito comum: a criação de um objeto para conexão com o banco de dados.

Listagem 1: Aplicação do Padrão 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;
    }

}

?>
```

Não se atente a linguagem, pois o legal de padrões de projeto é que você poderá aplicar ele a praticamente qualquer linguagem que tenha no mínimo um paradigma orientado a objetos. O objeto conexão é um ótimo exemplo para aplicação deste padrão, isso porque se você parar para pensar só precisamos de 1 objeto conexão em toda nossa aplicação, ou seja, você cria a "ponte" de conexão com o banco de dados somente 1 vez e a utiliza durante toda aplicação, imagine se você criasse uma nova conexão cada vez que fosse fazer um SELECT, com certeza você estaria sobrecarregando sua aplicação.

1. A primeira regra para se implementar o Singleton é sempre deixar o seu construtor privado, assim ninguém poderá instanciar a Classe diretamente e deixar o padrão Singleton cuidar disso.
2. Você precisará também de um atributo publico e estático do mesmo tipo da Classe que você está utilizando. Como o PHP é uma linguagem não tipada, não precisamos definir nenhum tipo para o atributo “instance”, mas no caso do Java você precisaria definir o tipo.
3. É no método "getInstance()" que toda mágica acontece, e você perceberá o quão simples é a aplicação do Singleton. Ele simplesmente verifica se a variável “instance” já foi criada, ou seja, se seu valor não é nulo, assim sendo ele apenas retorna a variável para o método que a chamou, caso a variável for nula, significa que esta classe ainda não foi instanciada nenhuma

vez, e o própria getInstance() chama o construtor da classe, criando a primeira e única instancia da classe durante toda a aplicação.

4. Toda vez chamarmos a classe que tem um padrão Singleton, devemos fazer: "MinhaClasse::getInstance()". Só em fazer isso, já estamos dizendo: "Quero que você me retorne essa Classe no padrão Singleton que foi implementado".

O Singleton previne que você crie uma nova instancia da Classe diretamente, usando a palavra reservada "new", isso porque você definiu o construtor da classe como PRIVATE. Mas aqui entra uma questão para quem está iniciando com padrões de projeto, quem me impediria de não definir o construtor como private, ou mesmo usar o nome getInstance() na minha classe? A resposta é curta e direta: Ninguém.

Caso você queira implementar um padrão de projeto, mas prefere fazer do seu "modo", podemos chamar de tudo, menos padrão de projeto. Imagine se você utiliza um Framework que cria suas classes Singleton automaticamente, e para isso chama sempre o método getInstance() (que é o padrão), mas você decidiu implementar pegarInstancia() ou criarApenasUmaVez(), simplesmente o seu framework não funcionaria.

Na figura abaixo temos uma definição simples e ao mesmo tempo completa do padrão singleton, isso porque apenas nessa imagem definimos todo o funcionamento da mesma.

Figura 1: Implementação do Padrão Singleton

Com isso fechamos este artigo e bom uso dos Padrões de Projeto!

Refatorando código PHP para Strategy Pattern

Quando falamos de Design Patterns, fica muito nebuloso entender o que podemos fazer com eles, pois o que muitos desenvolvedores não compreendem é que os padrões resolvem problemas de código ou de design já identificados ou simplesmente melhoram a manutenibilidade do projeto. Para que isso aconteça primeiro precisamos passar pelo problema, conhecer o que o padrão resolve e enfim aplicar uma refatoração.

Vamos falar neste artigo sobre o **Strategy**, que é um padrão comportamental, muito usado para quando temos regras de uma determinada atividade que podem conter muita lógica, ele organiza e separa o uso dessas lógicas, padronizando a usabilidade das classes de forma que novas implementações possam ser adicionadas no futuro sem muita mudança no uso de uma determinada classe de ação.

Para ilustrá-lo usaremos a refatoração, veremos o problema e aplicaremos o padrão resolvendo uma problemática específica. Para que fique menos complexo tudo será demonstrado em código e comentários, assim acredito que chegaremos a um entendimento melhor.

Problemática

Temos um sistema de gestão de logística e precisamos comunicar aos cliente quando o produto chegou na distribuidora, quando está a caminho da entrega e por fim o momento em que ele foi recebido pelo cliente.

Para tratar das mensagens que são enviadas aos clientes, foi criada uma classe **Message** da qual comunica por e-mail os clientes, veja abaixo a representação:

```
class Message
{
    // ...
    public function send(string $message)
    {
        // implementação de mensagem por email
    }
}

// ...
$message->send('Seu produto está a caminho da entrega');
```

Com o evoluir da aplicação foi preciso implementar um novo modelo de mensagem, o SMS, pois alguns clientes não possuíam cadastro online, e o sistema só possui cadastro simples como: o nome, telefone e endereço.

Resolvemos então criar duas classes, seguindo o principio de SOLID, single responsibility, a classe *EmailMessage* e a *SMSMessage*, pensando em futuras implementações fizemos a classe *MessageManager* retornar o objeto de acordo com o tipo de mensagem necessária.

```
class MessageManager
{
    // ...
    public function getSender()
    {
        if ($this->customer->fromOnline() && $this->customer->hasEmail()) {
            return new EmailMessage;
        }
        if ($this->customer->hasEmail() === false) {
            return new SMSMessage;
        }
        // ...
    }
}

// ...
$sender = $messageManager->getSender();
$sender->send('Seu produto está a caminho da entrega');
```

Mas como nada é perfeito, uma nova forma de comunicação entrou no projeto, os desenvolvedores precisam agora de um novo modo de envio, este agora via *push notification*, ou seja, a notificação será via aplicativo de celular, da forma atual nos forçou a ter essa implementação:

```
class MessageManager
{
    // ...
    public function getSender()
    {
        if ($this->customer->fromOnline() && $this->customer->hasEmail()) {
            return new EmailMessage;
        }
        if ($this->customer->hasEmail() === false) {
            return new SMSMessage;
        }
        if ($this->customer->hasCellPhone()) {
            return new PushMessage;
        }
        // ...
    }
}

// ...
```

Uma solução

Claramente não estamos seguindo corretamente a responsabilidade única, a classe *MessageManager* está com muitas regras de mensagem que se mesclam entre si, observamos que as regras de cada classe de mensagem deve estar dentro delas mesmas e o manager só deve invocar o objeto pedido no momento, pois ainda que surjam novas formas de mensagens, seria muito doloroso dar manutenção e testar esse método *getSender()*.

Com o **Strategy** transformamos cada regra em sua classe, que deve ser passada pelo desenvolvedor, elas seguirão uma interface:

```
interface MessageSenderInterface
{
    // ...
```

```
    public function send(string $message) : bool;
    public function isValidSender() : bool;
}

// ...
```

Essa interface fará o contrato das estratégias para que todas sigam esses métodos, implementaremos uma das classes de mensagens adicionando Strategy no nome original:

```
class EmailMessageStrategy implements MessageSenderInterface
{
    // ...
    public function send(string $message) : bool
    {
        // Valida de acordo com a regra deste tipo de mensagem
        if ($this->isValidSender()) {
            return $this->sendMessage($message);
        }
        return false;
    }

    public function isValidSender() : bool
    {
        if ($this->customer->fromOnline() && $this->customer->hasEmail()) {
            return true;
        }
        return false;
    }
}
// ...
```

Isso deve ser feito também para as demais classes de mensagem, essa prática isola as regras/validações nas classes *Strategy* de mensagens. Agora faremos o *MessageStrategy*, que invocará o método da classe em que lhe for informado. Abaixo a implementação do padrão:

```
class MessageStrategy implements MessageSenderInterface
{
    // ...
    public function __construct(MessageSenderInterface $sender)
    {
        // Guardamos o objeto Message com suas regras e implementações
        $this->sender = $sender;
    }

    public function send(string $message) : bool
    {
        // Retorna a validação de acordo com a regra do objeto
        return $this->sender->send($message);
    }

    public function isValidSender() : bool
    {
        return $this->sender->isValidSender();
    }
}

// ...
// Ao criar o objeto MessageStrategy devemos informar qual o tipo de envio
$message = new MessageStrategy(new EmailMessageStrategy);

// O método invoca o send original da classes anteriormente adicionada
$message->send('Seu produto está a caminho da entrega');
```

Pronto! Agora a implementação é injetada no *MessageStrategy* podendo haver outras implementações sem precisar alterar a classe *MessageStrategy* e, o melhor: as regras específicas para cada modelo de mensagem estarão em suas respectivas classes.

Conclusão

A vantagem em usar esse padrão está em facilitar a organização das classes, centralizar a usabilidade e além disso também dar poder ao desenvolvedor de adicionar camadas na estratégia principal que será replicada a todas as outras estratégias que são injetadas.

Espero que tenham gostado do artigo, a ideia foi ilustrar uma forma de implementar o padrão **Strategy** de forma simples, lembrando que existem várias maneiras de implementá-lo e os exemplos apresentados acima são meramente uma prova de conceitos.

Até a próxima!

O padrão Singleton com PHP

Escrito por Diogo Bemfica

Introdução

Esse é para ser o primeiro de uma série de artigos que vou escrever para documentar os meus estudos sobre os designer pattern(Padrões de Projeto) e aproveitar para compartilhar esses estudos com a comunidade em forma de artigos.

Nessa série vou demonstrar alguns dos principais padrões de projeto do mercado. A minha ideia não é falar de todos os padrões existentes, mas sim explicar alguns dos principais e demonstrar com um exemplo a sua implementação na linguagem PHP.

Design Patterns

Antes de partirmos para o nosso primeiro padrão como esse é o primeiro artigo dessa série. Achei melhor dar um pequena explicação sobre o que são os padrões de projeto de uma forma mais ampla e contar um pouco da sua história.

Para começar nada melhor do citar um dos livros de maior referência sobre o assunto, estou falando do "Padrões de Projeto - Soluções Reutilizáveis de Software Orientado a Objetos" da famosa Gang of Four(Gangue dos Quatro) de 1994. A onde os quatro autores "Erich Gamma", "John Vlissides", "Ralph Johnson" e "Richard Helm" descrevem 24 design pattern. Eles os separam em 3 categorias: padrões de criação, padrões estruturais e padrões comportamentais. Nessa série eu vou escolher dois padrões de cada categoria para explicar e demonstrar o seu funcionamento.

Apesar existirem mais do que os 24 padrões mencionados, este livro mesmo nos dias atuais como mencionado anteriormente é uma das maiores referências sobre o assunto.

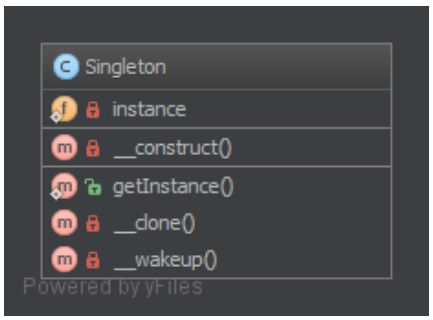
Singleton Pattern

Agora podemos começar, e para isso achei melhor falar sobre o padrão de criação Singleton. Vou explicar a sua estrutura, qual o problema ele busca resolver e mostrar um exemplo com um como podemos usar esse padrão nos nossos projetos desenvolvidos com o PHP.

Problema a resolver

O padrão Singleton prega a ideia da instância única. A onde quando você tenta criar uma instância da sua classe, essa classe deve verificar a existência da instância para que se a instância não existir criar uma nova e caso contrário retornar a mesma instância sem criar uma nova. Assim garantindo menor consumo de memória e um único ponto de acesso global para esse recurso.

O Singleton tem uma estrutura bem simples e de fácil compreensão. como mostrado na imagem abaixo.



Esse é o diagrama UML do padrão Singleton

Esse padrão é bastante usado para o desenvolvimentos de classes responsáveis pelo o gerenciamento da conexão com o banco de dados ou o gerenciamentos de logs. Nós permitindo ter acesso a esses recursos na nossa aplicação de forma rápida e descomplicada.

Implementação normal

Agora vamos finalmente falar de código. Mas primeiro vamos criar um classe da forma convencional, uma pequena classe de Logs. Vou usar esse exemplo demonstra que a cada instância criada estamos literalmente criando uma nova instância na memória.

```
<?php
class Log
{
}

$log = new Log;
$log2 = new Log;
$log3 = new Log;
$log4 = new Log;
$log5 = new Log;

var_dump($log);
var_dump($log2);
var_dump($log3);
var_dump($log4);
var_dump($log5);
```

Se executarmos o exemplo acima podemos ver que a cada nova chamada da classe *Log* nós criamos uma nova instância. começando com *#1* e indo até *#5*

```
object(Log)#1 (0) {
}
object(Log)#2 (0) {
}
object(Log)#3 (0) {
}
object(Log)#4 (0) {
}
object(Log)#5 (0) {
}
```

Implementação Singleton

Agora vamos começar a implementar o padrão Singleton na nossa classe, vamos começar a fazer isso alterando a visibilidade dos nosso métodos mágicos *__construct* o *__wakeup* e do *__clone*.

```
private function __construct()
{
}

private function __clone()
{
}

private function __wakeup()
{
}
```

Com isso garantimos que a classe não pode ser mais instanciada de forma normal.

Vamos criar uma propriedade privada estática *$instance*

```
private static $instance;
```

E também vamos criar um método estático *getInstance* que vai ser o único responsável por retornar a nova instância da nossa classe. É neste método que vamos verificar se já existe uma instância na nossa classe.

```
public static function getInstance()
{
    if(self::$instance === null){
        self::$instance = new self;
    }
    return self::$instance;
}
```

Se der certo vamos ter um arquivo como no exemplo abaixo.

```
<?php
class Log
{
    private static $instance;

    private function __construct()
    {
    }

    private function __clone()
    {
    }

    private function __wakeup()
    {
    }

    public static function getInstance()
    {
        if(self::$instance === null){
            self::$instance = new self;
        }
        return self::$instance;
    }
}
```

Agora com a nossa classe devidamente modificada podemos testar o nosso exemplo

```
$log = Log::getInstance();
$log2 = Log::getInstance();
```

```
$log3 = Log::getInstance();
$log4 = Log::getInstance();
$log5 = Log::getInstance();

var_dump($log);
var_dump($log2);
var_dump($log3);
var_dump($log4);
var_dump($log5);
```

No exemplo acima agora com o nosso Singleton funcionando vamos ver que a nossa instância é sempre a mesma *#1*. Conseguimos garantir a mesma instância não importa quantas vezes chamamos a nossa classe.

```
object(Log)#1 (0) {
}
object(Log)#1 (0) {
}
object(Log)#1 (0) {
}
object(Log)#1 (0) {
}
object(Log)#1 (0) {
}
```

Singleton é um anti pattern?

Para muitos desenvolvedores o Singleton é considerado um anti patterns, mas por que será? Acredito que uns dos principais motivos é pela natureza estática. Isso nos traz alguns problemas como não podermos trabalhar com interfaces e o aumento de acoplamento na classe. Além disso como estamos falando de acesso global, existe um perigo de sobrescrever o comportamento da classe e gerar comportamentos inesperados no sistema.

Acredito que esses dois são os principais motivos para que o Singleton tenha esse estigma. Mas na minha opinião ele é um padrão como qualquer outro, tem suas vantagens e desvantagens cabe ao desenvolvedor ter a sabedorias para saber quando e onde usá-lo.

Conclusão

Como falei anteriormente esse artigo é primeiro de uma série que pretendendo escrever para compartilhar os meus estudos com a comunidade. Comecei falando do Singleton por ser um padrão bastante famoso e de fácil compreensão. Espero que com esses pequenos passos que possa ter de demostrado como podemos facilmente criar a nossa classe usando o Singleton no PHP.

Referências

https://www.schoolofnet.com/curso-design-patterns-pt2-padroes-de-criacao/
https://www.casadocodigo.com.br/products/livro-design-paterns-php
https://github.com/webfatorial/PadroesDeProjetoPHP/tree/master/Creational/Singleton