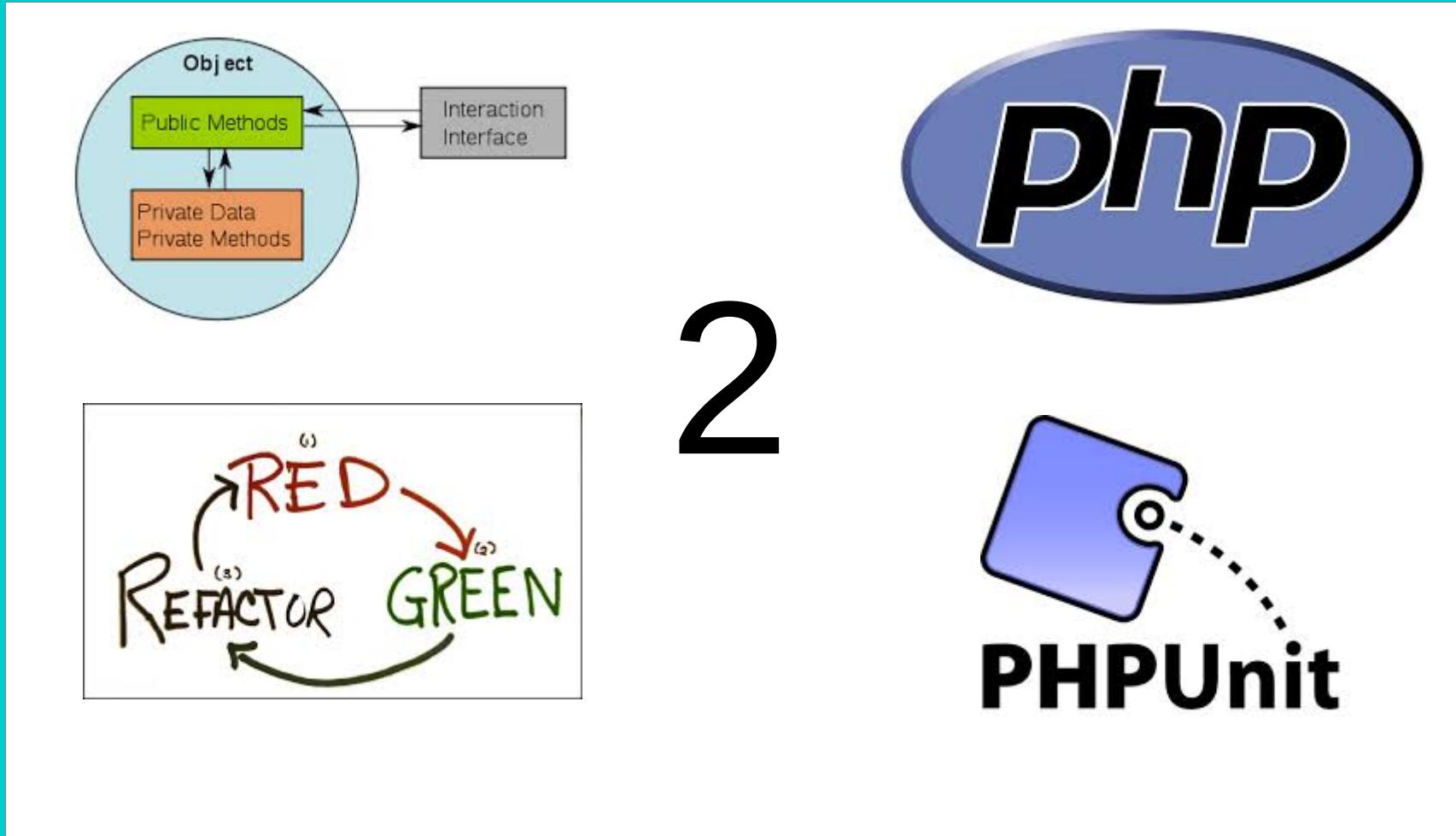
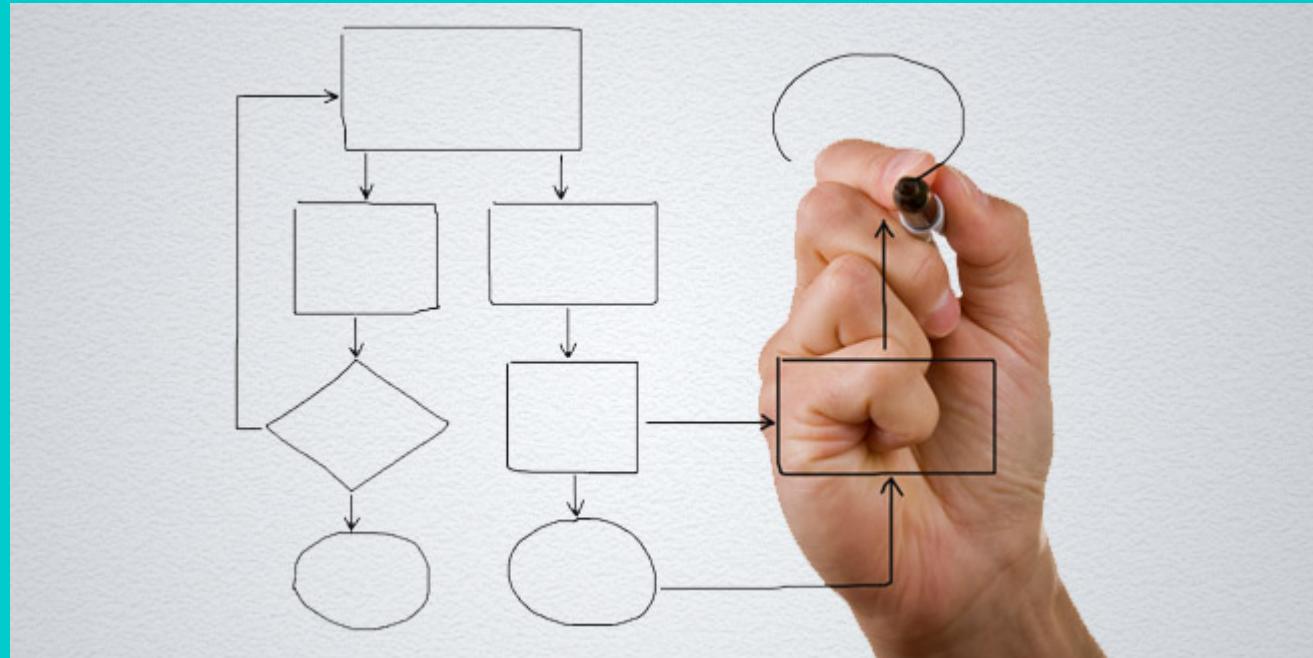


# Programação PHP orientada a objetos com testes unitários

FLÁVIO GOMES DA SILVA LISBOA



# Planejamento



# Programação PHP orientada a objetos

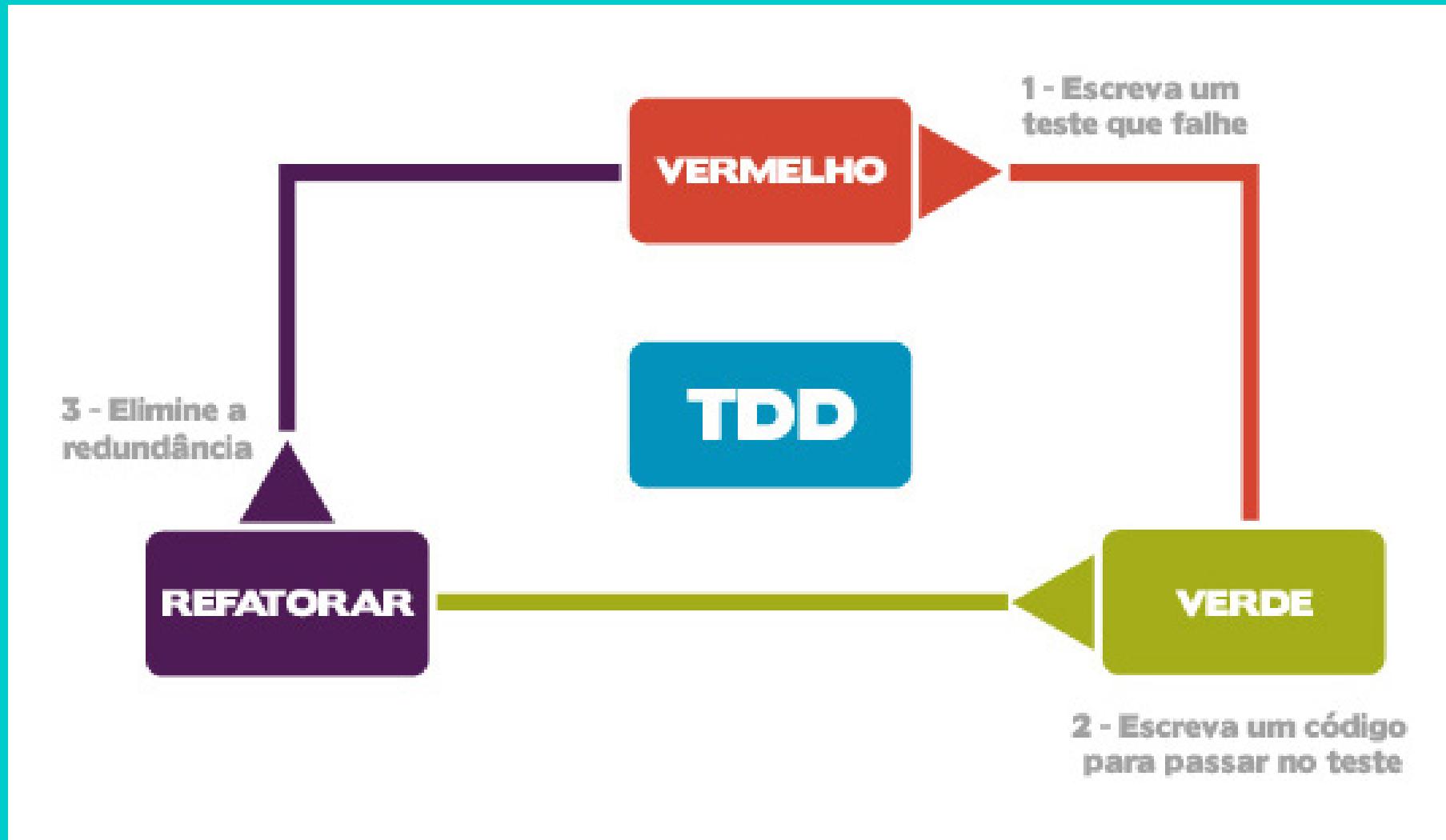
## Parte 2

- Clonagem de objetos
- Comparação de objetos
- Indução de tipo
- Objetos e Referências
- *Late Static Bindings*
- *Serialização de objetos*
- Métodos mágicos
- Padrões PHP
- Continuação do trabalho de avaliação da disciplina

# E os testes unitários?



# TDD (*Test Driven Development*)



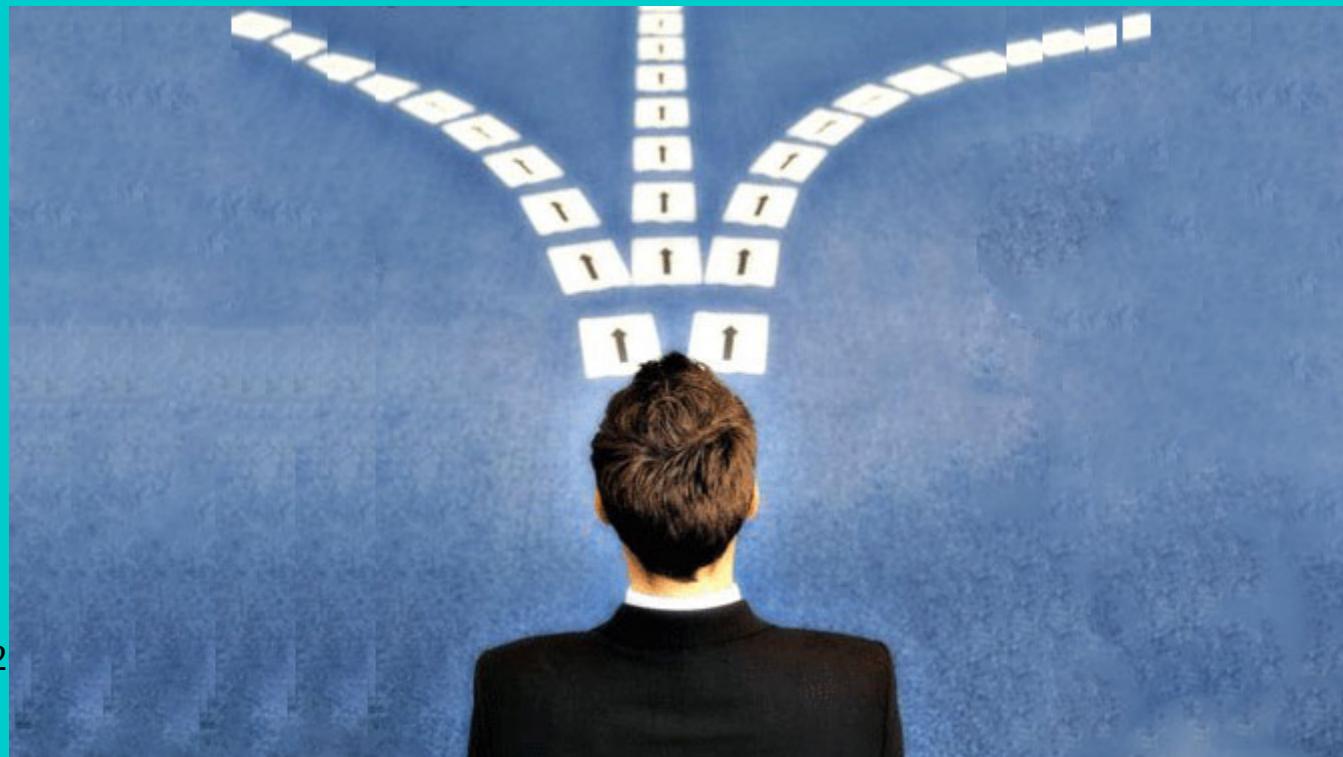
# Anteriormente, nos últimos capítulos...

- Escrevemos um *script* PHP
- Nesse *script* escrevíamos o que esperávamos que nosso código fizesse.
- Imprimíamos um texto indicando o resultado:  
**“Passou no teste” ou “Não passou no teste”**



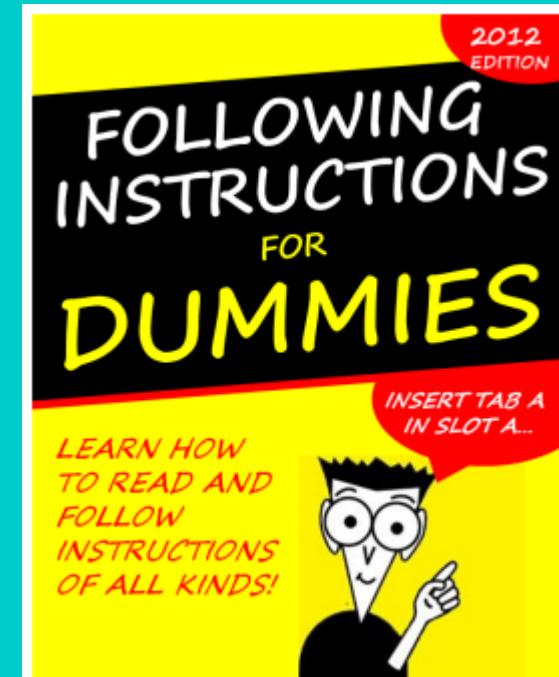
# Possibilidades de um teste

- Ter sucesso (o que era esperado ocorreu)
- Falhar (não ocorreu o que era esperado)
- Não ser concluído (erro fatal)



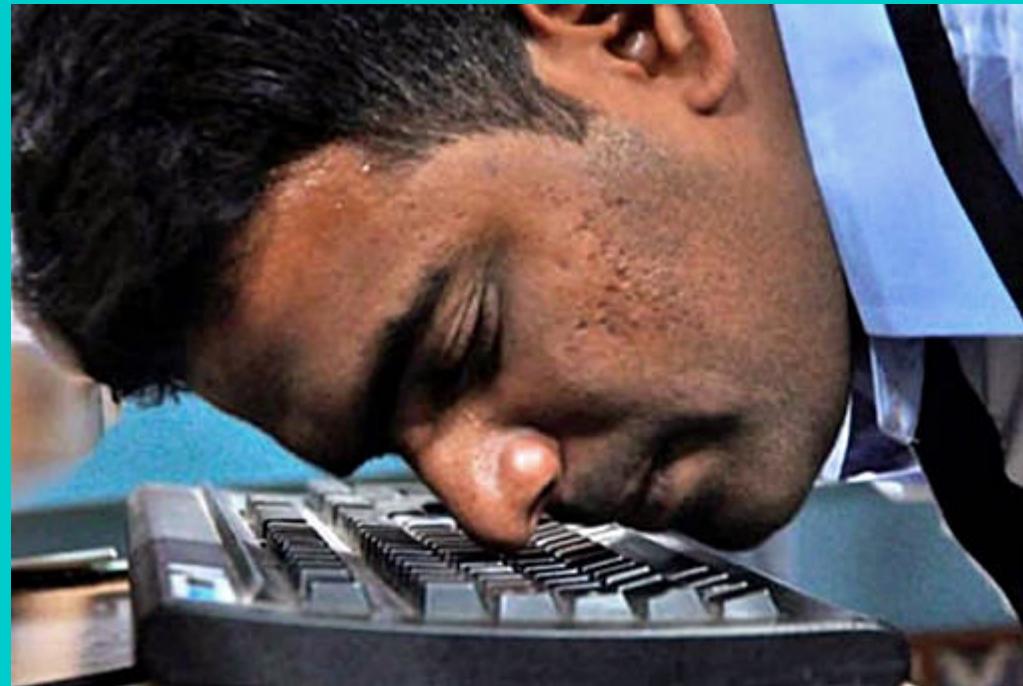
# Nosso algoritmo

- Execute o código
- Verifique o resultado do código
- Se for o esperado, escreva “Passou no teste”
- Senão, escreva “Não passou no teste”



# Problemas com os scripts...

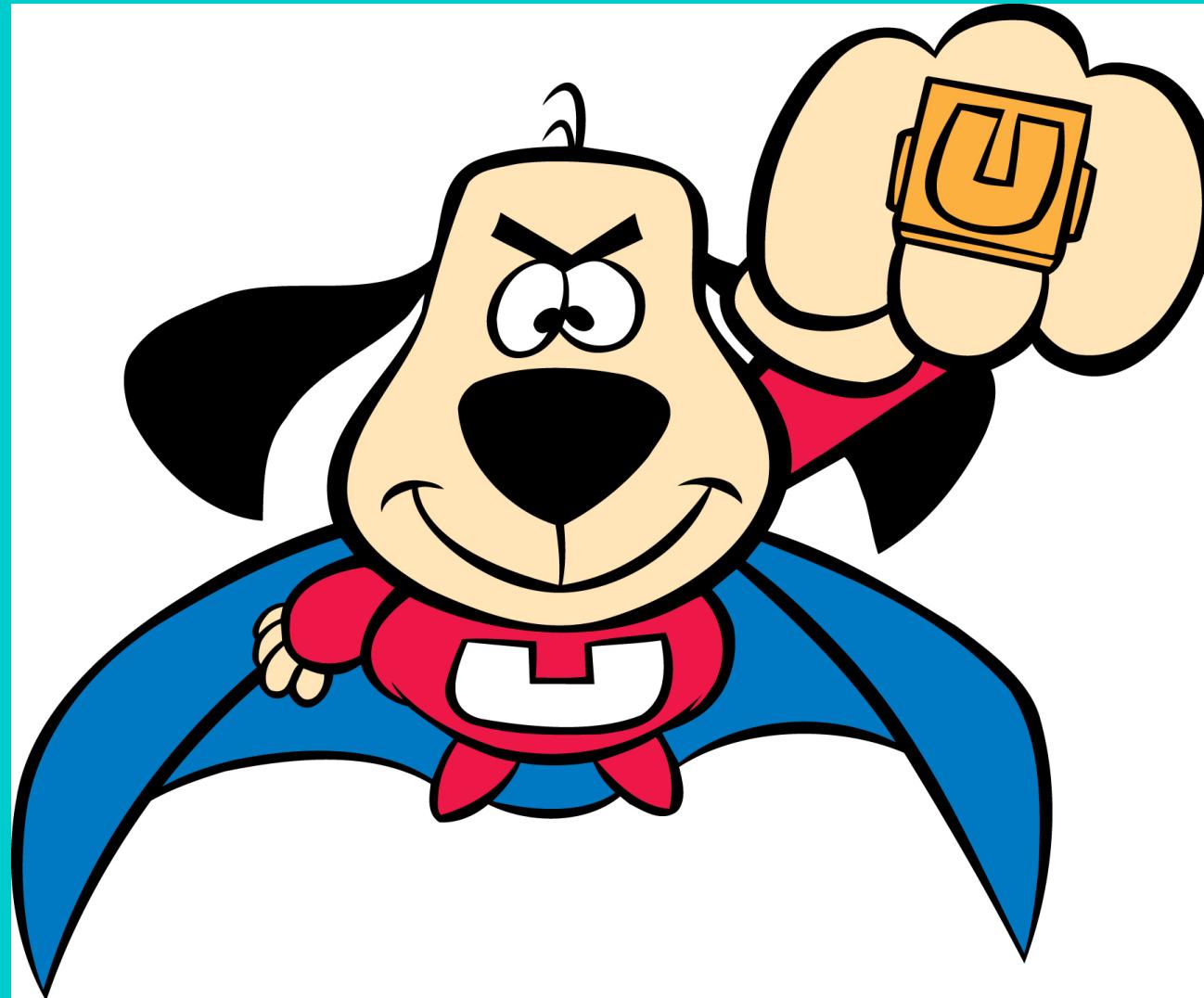
- Temos de executar um por um



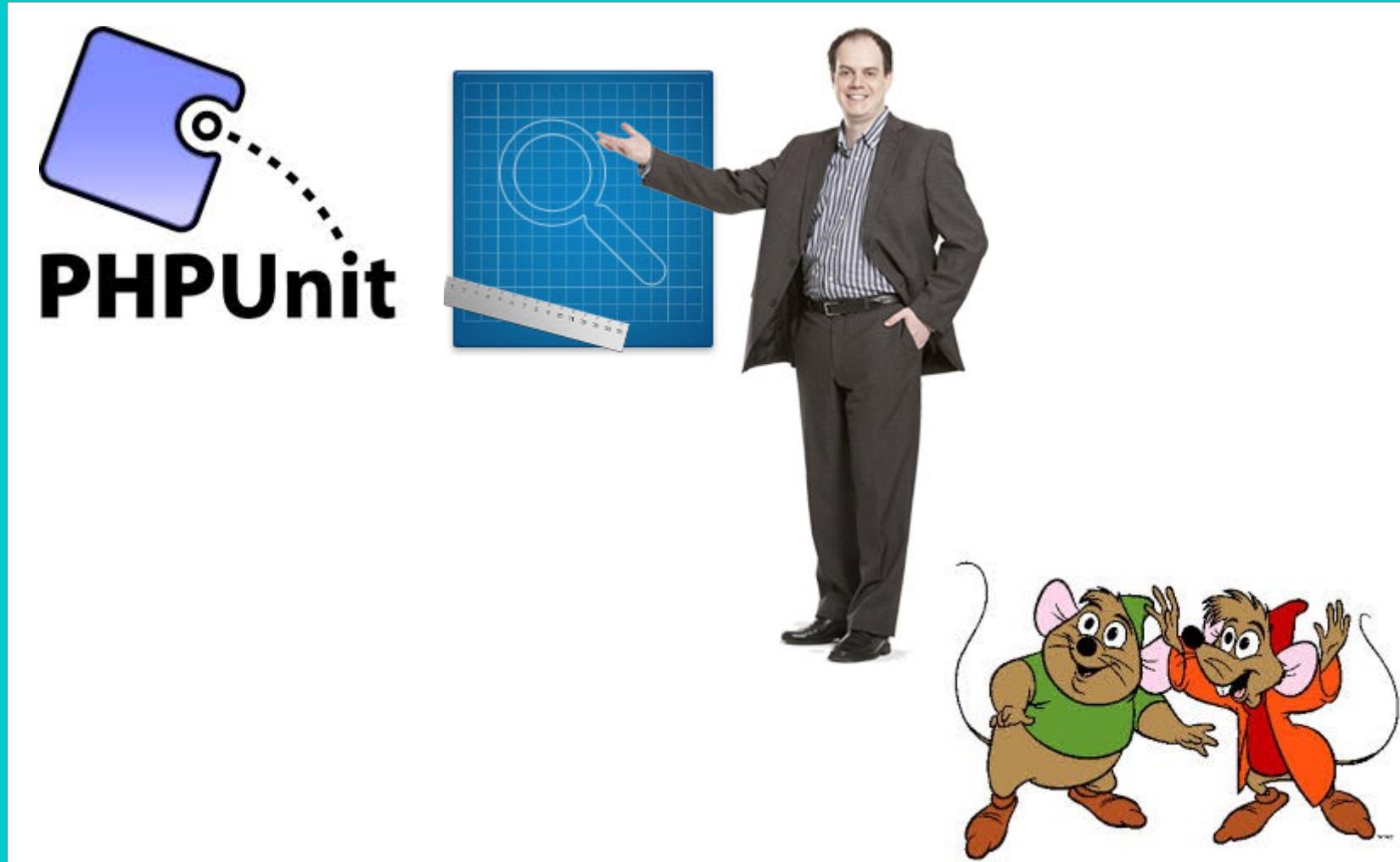


**COMO SERIA  
MARAVILHOSO SE  
HOUVESSE UM JEITO  
MAIS FÁCIL DE  
ESCREVER E  
EXECUTAR TESTES...  
MAS DEVE SER UM  
SONHO...**

# Seu problema acabou!



# PHPUnit chegou!



# PHPUnit

## Code

src/Money.php

```
<?php
class Money
{
    private $amount;

    public function __construct($amount)
    {
        $this->amount = $amount;
    }

    public function getAmount()
    {
        return $this->amount;
    }

    public function negate()
    {
        return new Money(-1 * $this->amount);
    }

    // ...
}
```

## Test Code

tests/MoneyTest.php

```
<?php
class MoneyTest extends PHPUnit_Framework_TestCase
{
    // ...

    public function testCanBeNegated()
    {
        // Arrange
        $a = new Money(1);

        // Act
        $b = $a->negate();

        // Assert
        $this->assertEquals(-1, $b->getAmount());
    }

    // ...
}
```

# PHPUnit

Um caso de teste PHPUnit é uma classe que estende **PHPUnit\_Framework\_TestCase**.

Cada método de uma classe *TestCase* iniciado pelo prefixo **test** é um **teste unitário**.

O método **setUp()** é executado antes de todos os testes unitários e o método **tearDown()** é executado ao final da execução dos testes. O primeiro serve para preparar o ambiente para os testes e o segundo serve para retornar ao estado inicial, antes dos testes.

# Suíte de Teste

```
class TestSuite
{
    /**
     * run all tests
     */
    public static function main()
    {
        \PHPUnit_TextUI_TestRunner::run(self::suite());
    }
}
```

# Suíte de Teste

```
/**  
 * @return \PHPUnit_Framework_TestSuite  
 */  
  
public static function suite()  
{  
    $suite = new \PHPUnit_Framework_TestSuite('Our Tests');  
    $suite->addTestSuite('Namespace\ModuleTest');  
    return $suite;  
}  
}
```

# PHPUnit

Em PHPUnit não imprimimos nenhuma saída para informar o resultado. O PHPUnit imprime para nós.

```
PHPUnit X.Y.Z by Sebastian Bergmann.
```

```
Configuration read from [nome do arquivo]
```

```
.
```

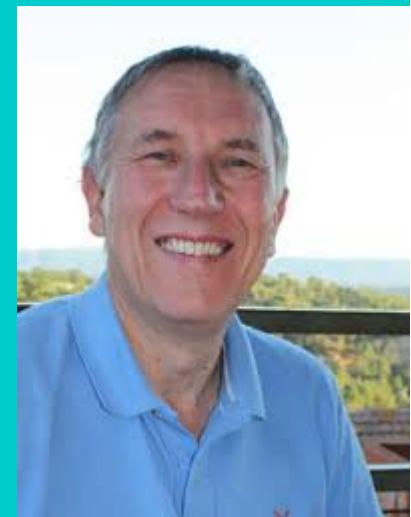
```
Time: [NN] ms, Memory: [N.MM]Mb
```

```
OK ([número de testes unitários], [número de assertivas])
```

# Assertivas

*“O primeiro estágio da **tolerância a defeitos** é detectar se um defeito (um estado errôneo do sistema) ocorrerá a não ser que alguma ação seja tomada imediatamente. Para fazer isso, você precisa saber quando o valor de uma **variável de estado** é **illegal** ou quando **relacionamentos entre variáveis de estados** não são **mantidos**”*

Sommerville (2009, p. 314)



# Assertivas

*“As assertivas podem ajudar a detectar erros **antecipadamente**, em especial, em sistemas **grandes**, sistemas de **alta confiabilidade** e bases de código que **mudam com muita frequência**”*

McConnell (2005, p. 238)

# Assertivas



- assertArrayHasKey()
- assertClassHasAttribute()
- assertClassHasStaticAttribute()
- assertContains()
- assertContainsOnly()
- assertContainsOnlyInstancesOf()
- assertCount()

# Assertivas



- assertEquals()
- assertEqualXMLStructure()
- assertFalse()
- assertFileEquals()
- assertFileExists()
- assertGreaterThan()
- assertGreaterThanOrEqual()

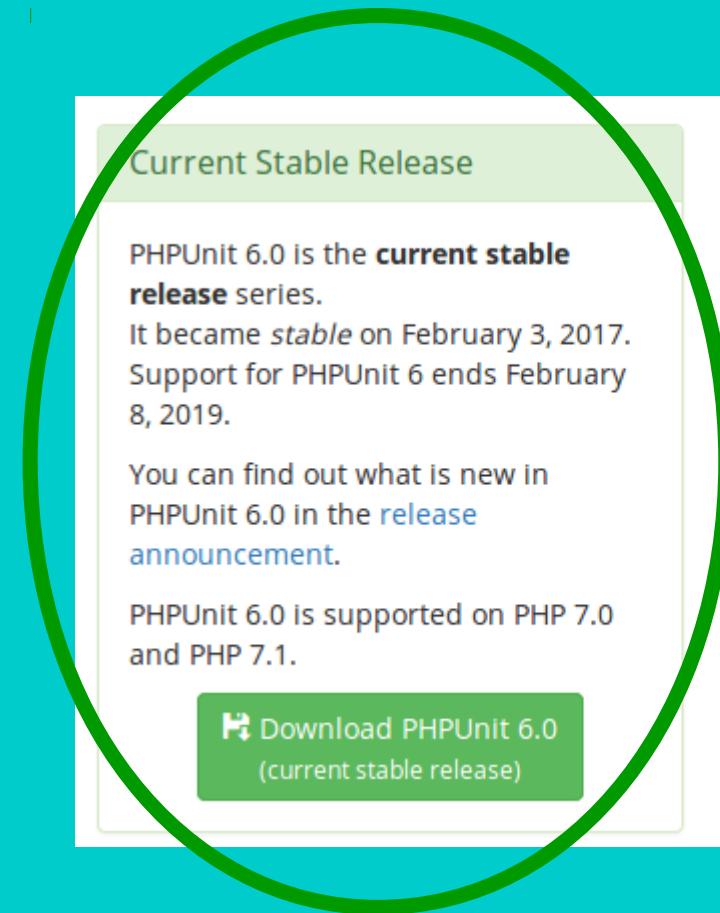
# Assertivas



- assertInstanceOf()
- assertInternalType()
- assertJsonFileEqualsJsonFile()
- assertJsonStringEqualsJsonFile()
- assertJsonStringEqualsJsonString()
- assertLessThan()
- assertLessThanOrEqual()
- assertNull()

# Vamos baixar o PHPUnit?

<https://phpunit.de/>



**Current Stable Release**

PHPUnit 6.0 is the **current stable release** series.  
It became *stable* on February 3, 2017.  
Support for PHPUnit 6 ends February 8, 2019.

You can find out what is new in PHPUnit 6.0 in the [release announcement](#).

PHPUnit 6.0 is supported on PHP 7.0 and PHP 7.1.

**Download PHPUnit 6.0  
(current stable release)**

**Old Stable Release**

PHPUnit 5.7 is the **old stable release** series.  
It became *stable* on December 2, 2016.  
Support for PHPUnit 5 ends on February 2, 2018.

You can find out what was new in PHPUnit 5.7 in the [release announcement](#).

PHPUnit 5.7 is supported on PHP 5.6, PHP 7.0, and PHP 7.1.

**Download PHPUnit 5.7  
(old stable release)**

**No Longer Supported**

PHPUnit 4.8 is **no longer supported**.  
It became *stable* on August 7, 2015.  
Support for PHPUnit 4 ended on February 3, 2017.

You can find out what was new in PHPUnit 4.8 in the [release announcement](#).

PHPUnit 4.8 was supported on PHP 5.3 – 5.6.

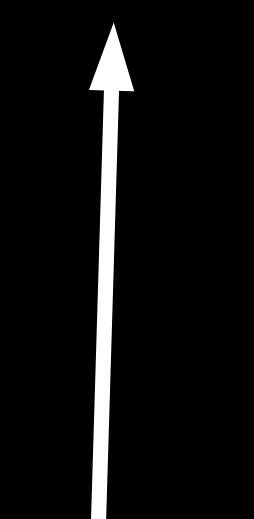
**Download PHPUnit 4.8  
(no longer supported)**

# Vamos rever os testes que fizemos?



# Executando um caso de teste PHPUnit

```
phpunit NameOfClassTest.php
```



# Executando vários casos de teste PHPUnit

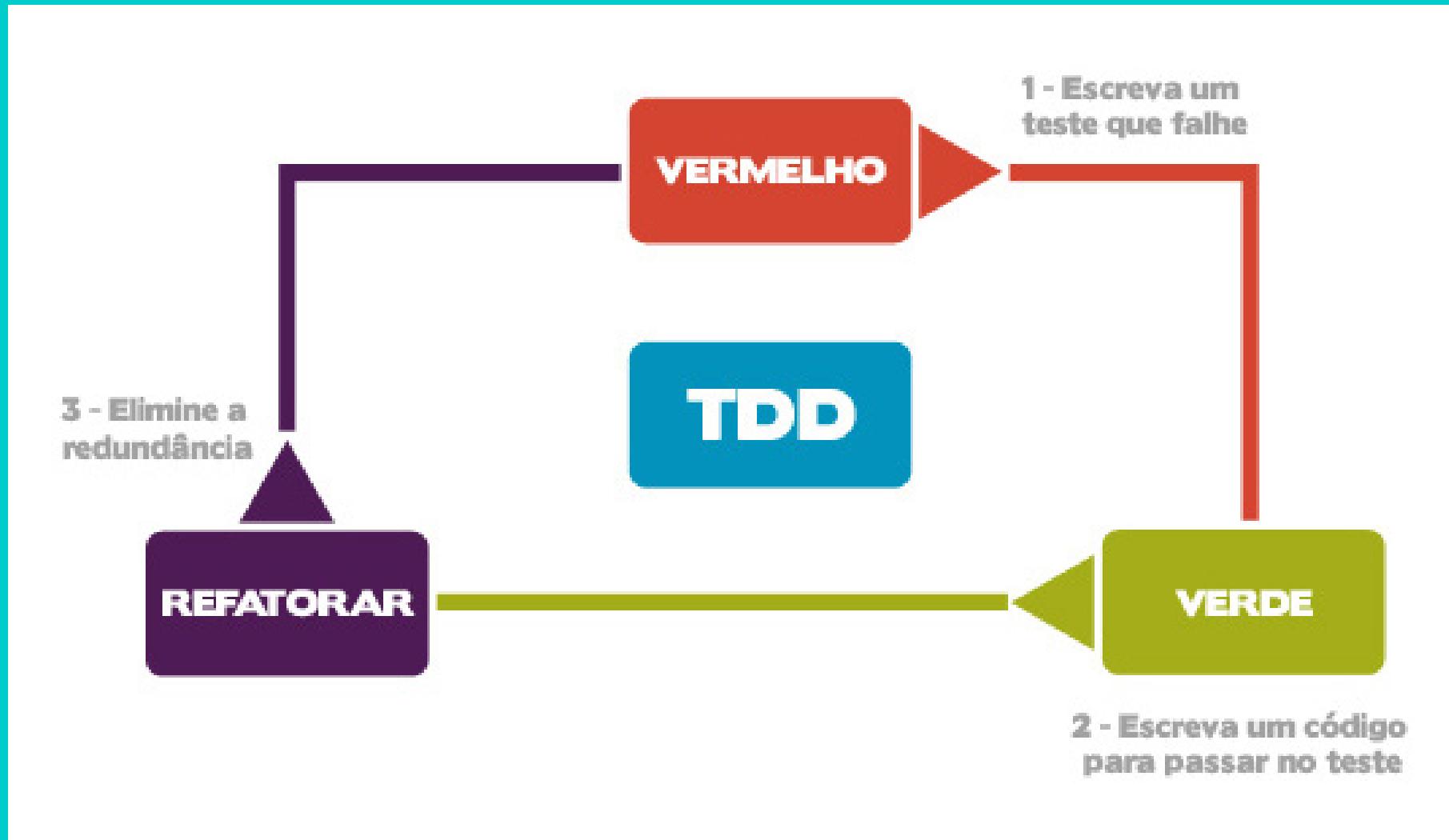
Um arquivo XML pode definir uma pasta com casos de teste para o PHPUnit executar.

```
phpunit -c tests.xml
```

# Executando vários casos de teste PHPUnit

```
<phpunit>
<testsuites>
    <testsuite name="webdev">
        <directory>tests</directory>
    </testsuite>
</testsuites>
</phpunit>
```

# TDD (*Test Driven Development*)



# O Mantra do TDD

- **Vermelho** – Escrever um pequeno teste que não funcione e que talvez nem mesmo compile inicialmente.
- **Verde** – Fazer rapidamente o teste funcionar, mesmo cometendo algum pecado necessário no processo.
- **Refatorar** – Eliminar todas as duplicatas criadas apenas para que o teste funcione.

Fonte: Beck (2010, p. x)

# As 3 Leis do TDD

- **Primeira Lei:** Você não pode escrever código de produção até ter escrito um teste unitário que falhe.
- **Segunda Lei:** Você não pode escrever mais de um teste unitário do que o suficiente para falhar, e não compilar é falhar.
- **Terceira Lei:** Você não pode escrever mais código de produção do que o suficiente para passar pelo atual teste que falha.

Fonte: Martin (2009, p. 153)



# Voltando aos objetos...



# Objetos

*“Objetos são, potencialmente, componentes reusáveis porque eles são encapsulamentos independentes de estado e operações”*

Sommerville (2009, p. 209)



# Clonagem de objetos



# Clonagem de objetos

- Um clone **não** é um filho.
- Uma classe herdeira **não** é clone de uma classe abstrata, pois não tem todos os atributos da classe mãe.



# Clonagem de objetos

- O operador **new** cria um novo objeto.

```
$objeto = new Classe();
```

- O operador **clone** cria um objeto idêntico a outro.

```
$copia = clone $objeto;
```

O **clone** é um outro objeto, com outro identificador, mas com os mesmos valores de atributos.

# Comparação de objetos





# Comparação de objetos

- A comparação entre objetos pode ser feita por **igualdade**, quando se compara se os objetos são da mesma classe e se seus atributos tem os mesmos valores. O operador é o ==
- A comparação entre objetos pode ser feita por **identidade**, quando se compara se os objetos exatamente o mesmo, a mesma referência em duas variáveis diferentes. O operador é o ===





# Comparação de classes

- A comparação de classes (se a classe de um objeto é igual a uma determinada classe) é feita com o operador **instanceof**.
- A função **get\_class()** retorna a classe de um objeto.
- A constante **\_\_CLASS\_\_** retorna o nome da classe onde ela está sendo chamada.
- A função **get\_called\_class()** retorna a classe que iniciou a chamada a um método.

# Vamos testar?



# Indução de Tipo

Funções e métodos podem forçar seus parâmetros a serem objetos pela especificação de uma **classe** ou **interface** na assinatura. Caso um valor não seja uma instância da classe especificada ou que não implemente a interface especificada, ocorrerá um erro fatal, a menos que o parâmetro tenha o valor padrão **NULL**.

(**Tipo** \$parametro)

# Vamos testar?





# Objetos e referências

*“Uma **referência PHP** é um alias, que permite duas variáveis diferentes escreverem para o **mesmo valor**. A partir do PHP 5, uma variável objeto não contém mais o próprio objeto como valor. Ela contém um **identificador do objeto** que permite que os acessadores do objeto encontrem o objeto real. Quando um objeto é enviado por argumento, retornado ou atribuído para outra variável, as variáveis diferentes não são aliases: elas armazenam **uma cópia do identificador**, que aponta para o mesmo objeto.”*

Fonte: [http://br1.php.net/manual/pt\\_BR/language.oop5.references.php](http://br1.php.net/manual/pt_BR/language.oop5.references.php)

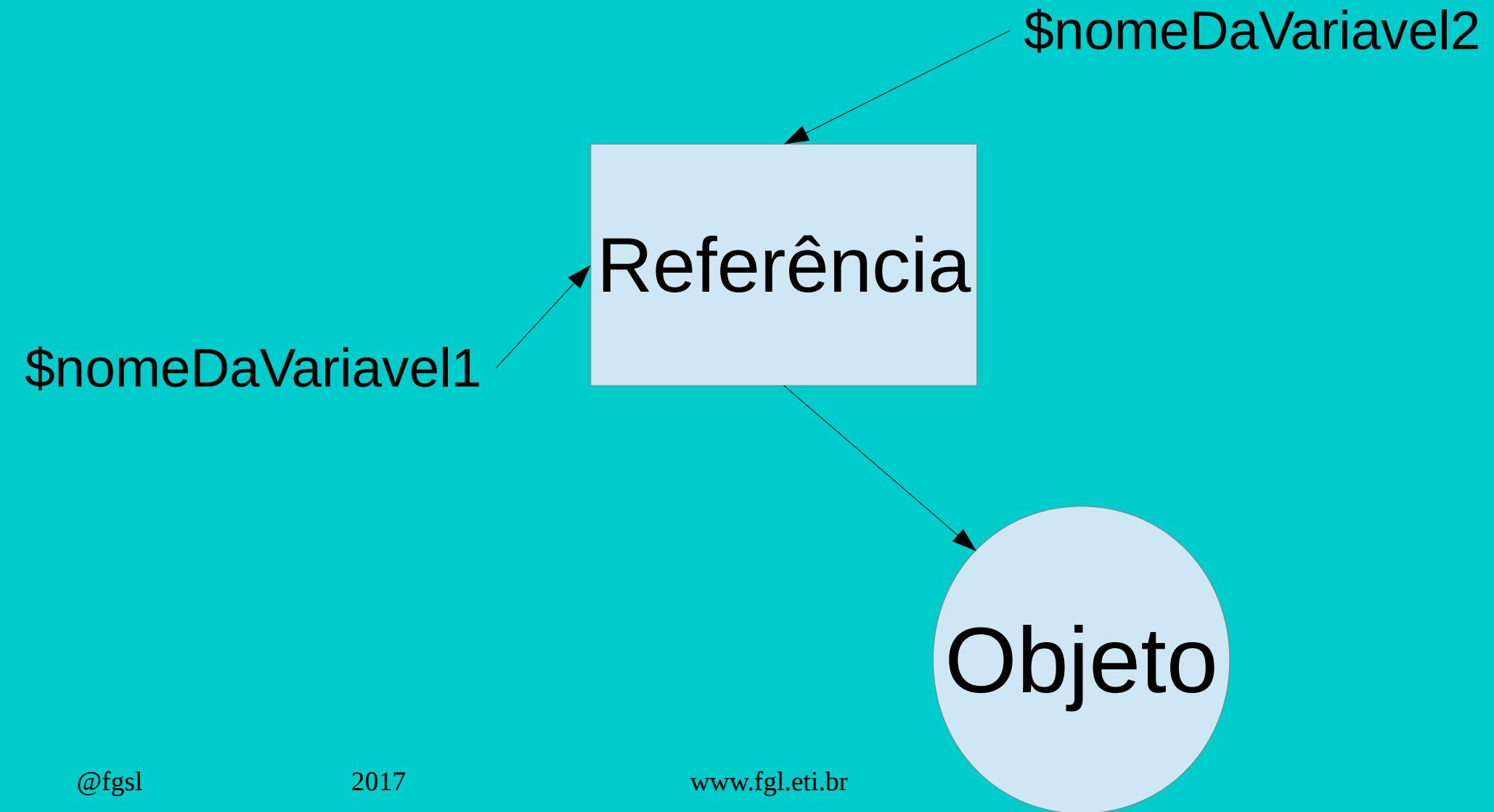
# Objetos e referências

`$nomeDaVariavel`



Referência

# Objetos e referências



# Vamos testar?



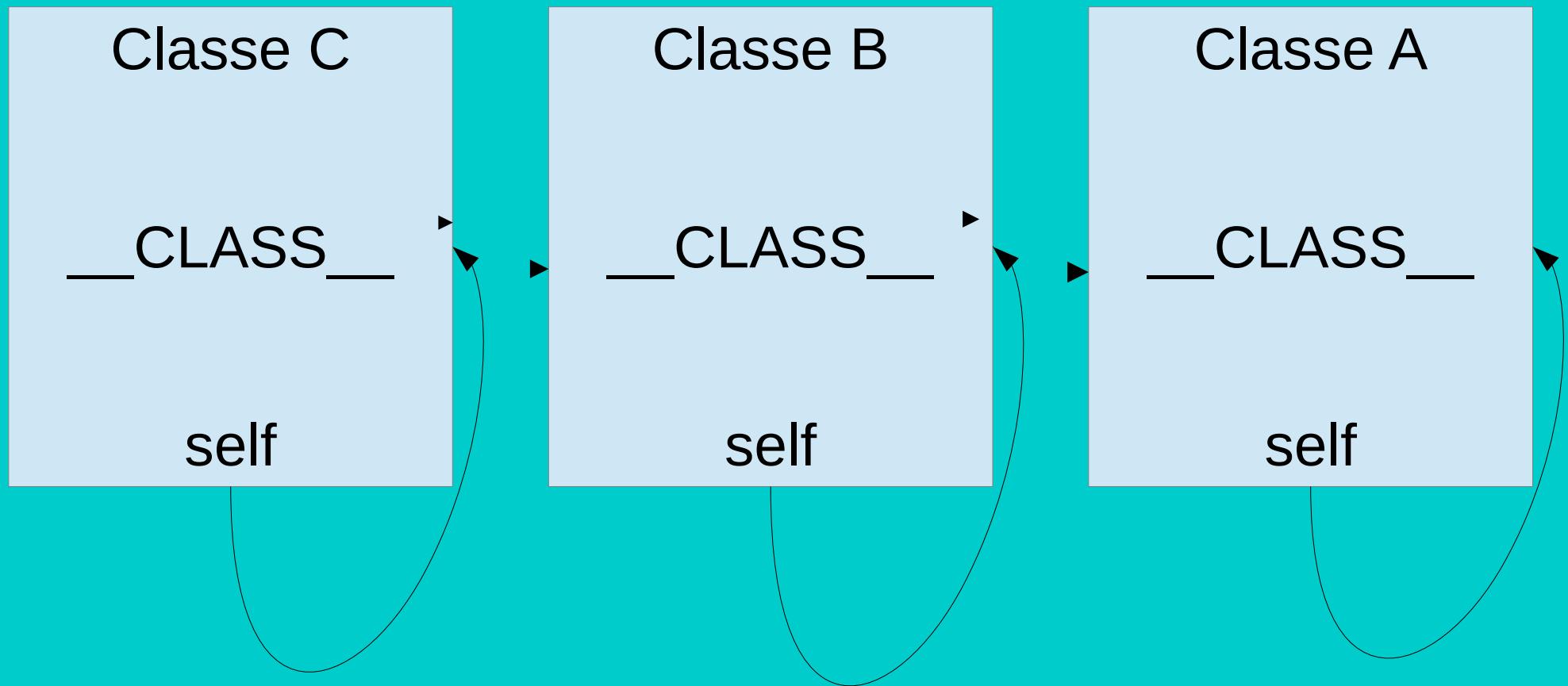
# *Late Static Bindings*

Referências estáticas para a classe atual como `self::` ou `__CLASS__` são resolvidas usando a classe a qual o método pertence.

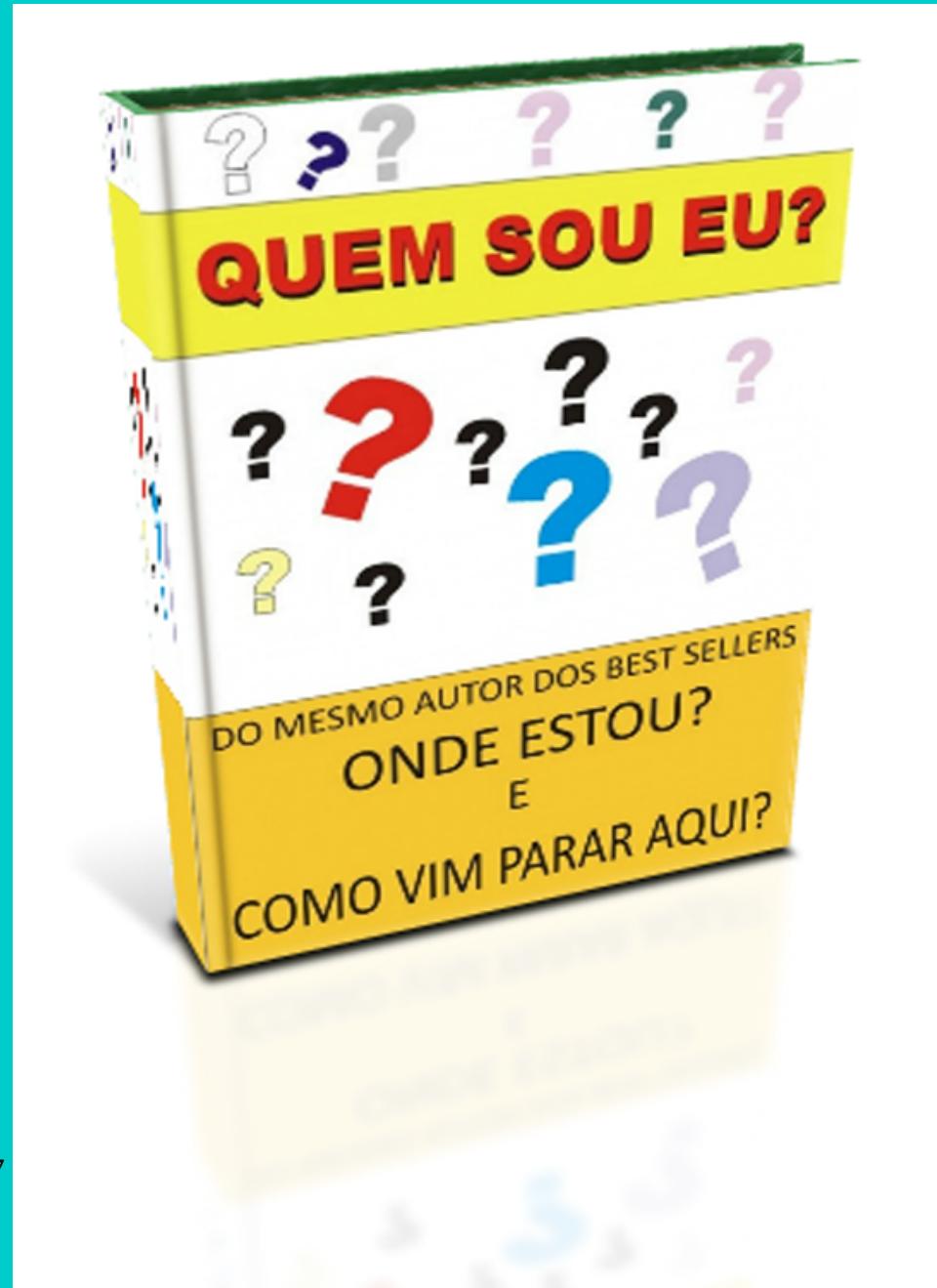
Isso significa que se uma classe usa um método herdado que contenha a palavra `__CLASS__`, o conteúdo dessa constante será o nome da classe mãe e não da filha.

E `self` chama os atributos e métodos da classe onde está declarado e não de onde foi chamado.

# *Late Static Bindings*



# *Late Static Bindings*



# *Late Static Bindings*

*Late static bindings* tenta resolver a limitação de **\_\_CLASS\_\_** e **self** introduzindo uma palavra-chave que referencia a classe que foi inicialmente chamada em *runtime*: **static**.



# Vamos testar?



# Serialização de objetos

**Serializar** um objeto significa transformá-lo em um texto que preserve seus dados e que permita reconstruí-lo depois.

A função **serialize()** cria uma representação textual de um valor em PHP. Para objetos, o formato do texto segue este modelo:

```
O:[id do objeto]:"[nome da classe]":
[numero de atributos]:{[definição de
atributos e seus valores]}
```

# Serialização de objetos

A definição de atributos contém geralmente as seguintes informações:

```
{N[se o atributo tiver valor];s:[comprimento do nome do atributo]:"[nome do atributo]";[tipo de atributo]s:[comprimento do valor do atributo se aplicável]:"[valor do atributo]";}
```

# Serialização de objetos

**Reverter a serialização de um objeto** significa reconstruir um objeto a partir do texto que o representa.

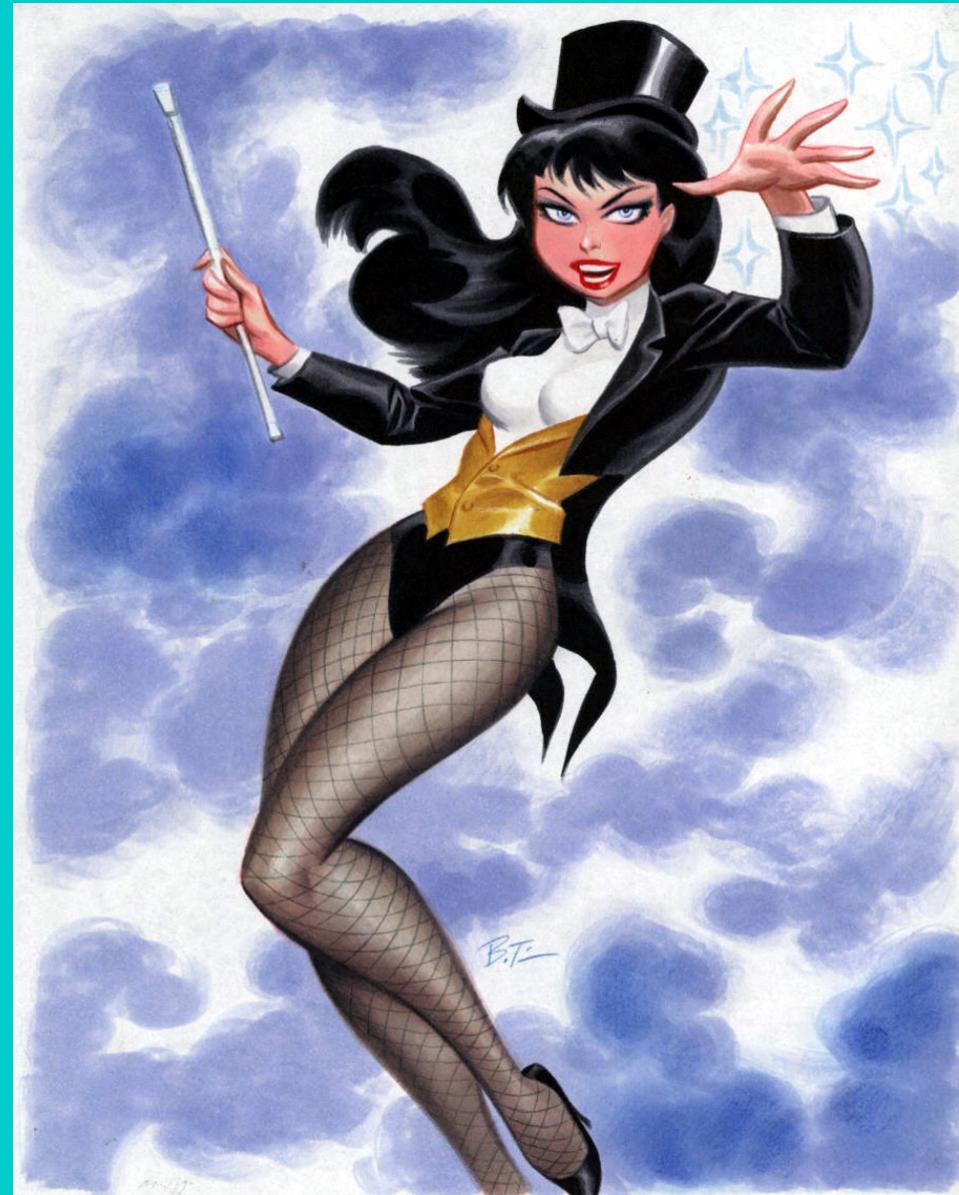
A função **unserialize()** cria um valor baseado em uma representação textual. Para reconstruir objetos, é necessário que a declaração da classe esteja disponível. Se ela não estiver, o PHP criará um objeto da classe **PHP\_Incomplete\_Class\_Name**, sem métodos.

Os dados serializados referem-se somente aos **valores** dos atributos. A informação sobre os métodos é da **classe**.

# Vamos testar?



# Métodos Mágicos



# Métodos Mágicos

PHP reserva todas as funções e métodos com nomes começando com \_\_ como **mágicos**. Por isso é recomendado que você não use funções e métodos com nomes com \_\_ no PHP.

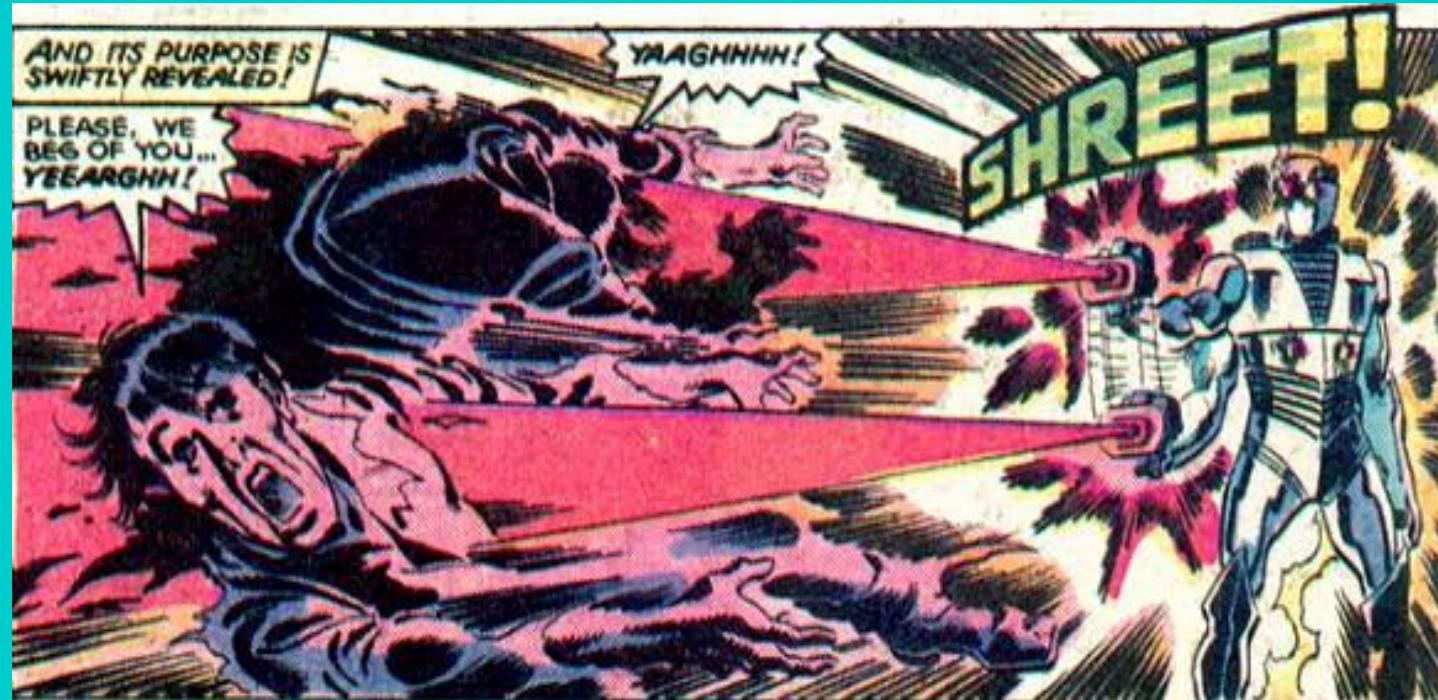
Métodos mágicos não são chamados **diretamente**. Eles são executados em resposta a **eventos**.

Métodos mágicos podem ser **sobre carregados**.

# Métodos Mágicos

Os métodos `__construct()` e `__destruct()` são exemplos de métodos mágicos.

Métodos mágicos podem **neutralizar** erros.



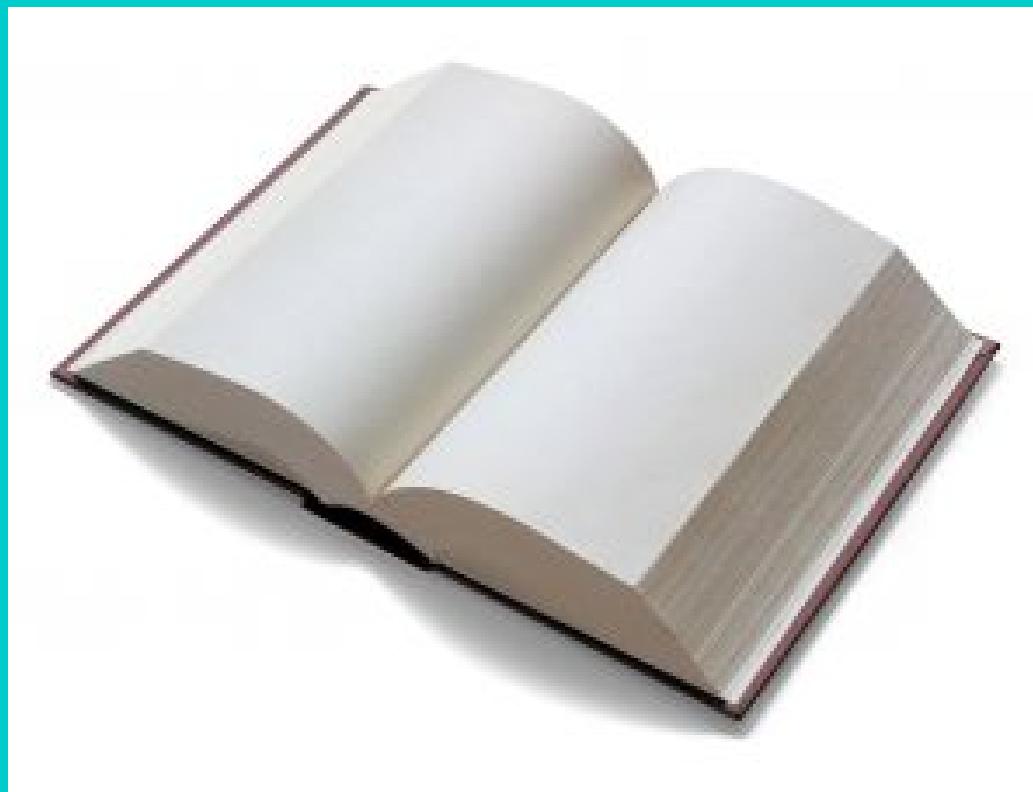
# Métodos Mágicos: `__set()`

O método `__set()` é chamado quando há uma tentativa de atribuição de valor para um atributo inexistente em um objeto.



# Métodos Mágicos: `__get()`

O método `__get()` é chamado quando há uma tentativa de leitura de um atributo inexistente.

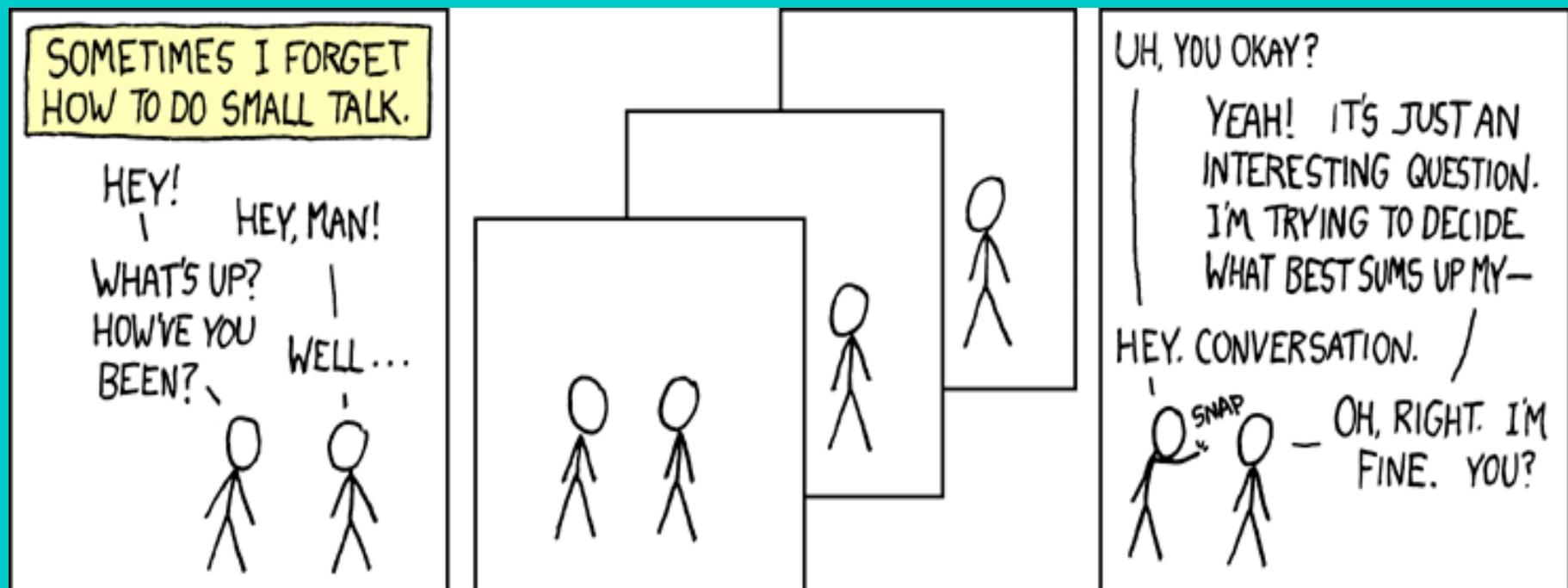


# Vamos testar?



# Métodos Mágicos: \_\_call()

O método `__call()` é chamado quando é feita uma tentativa de invocar um método inexistente.



# Vamos testar?



# Métodos Mágicos: `__isset()`

O método `__isset()` é chamado quando um atributo do objeto é passado como argumento para o método `isset()`.

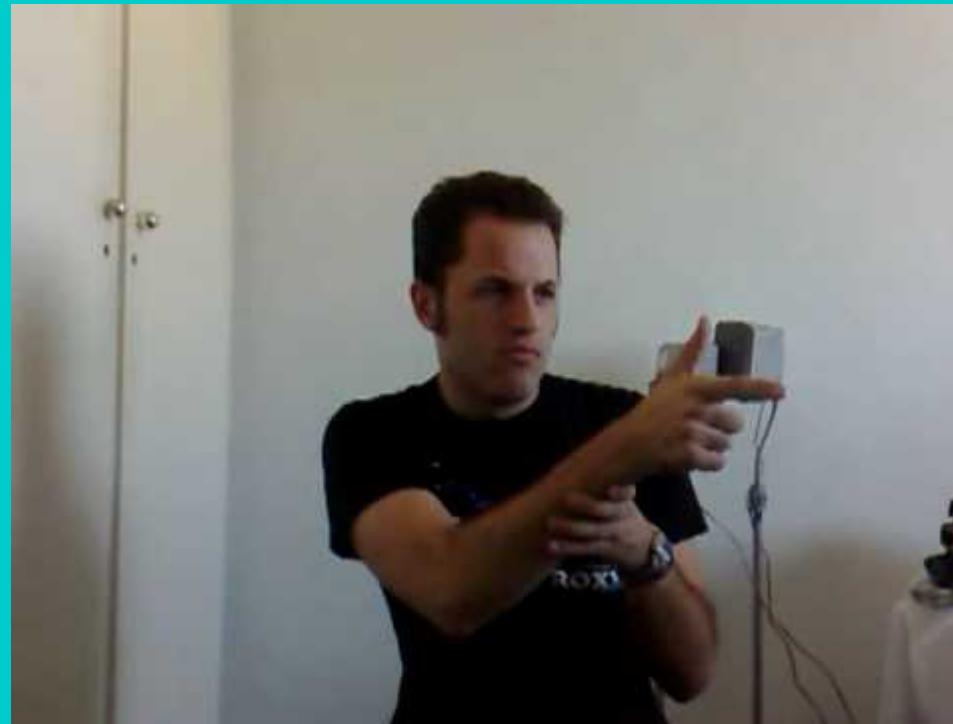


# Vamos testar?



# Métodos Mágicos: \_\_invoke()

O método **\_\_invoke()** é chamado quando uma variável contendo um objeto é usada como se fosse uma função.



# Vamos testar?



# Métodos Mágicos: `__callStatic()`

O método `__callStatic()` é chamado quando é feita uma tentativa de chamar um método estático inexistente.



# Vamos testar?



# Métodos Mágicos: `__toString()`

O método `__toString()` é chamado quando se trata um objeto como se fosse texto.



# Vamos testar?



# Métodos Mágicos: `__clone()`

O método `__clone()` é chamado quando um objeto é criado com o operador `clone`.



# Vamos testar?



# Métodos Mágicos: `__sleep()`

O método `__sleep()` é chamado quando um objeto é serializado.



# Métodos Mágicos: `__wakeup()`

O método `__wakeup()` é chamado quando é feita uma tentativa (com sucesso) de reconstruir um objeto com dados serializados.



# Vamos testar?



# Padrões PHP



## Framework Interoperability Group

Um grupo de representantes de projetos PHP que falam sobre as semelhanças entre seus projetos e encontram maneiras de trabalhar juntos.

# Alguns membros do PHP-FIG

Agavi



Joomla!



Symfony

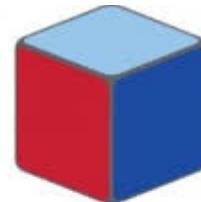


PRESTASHOP



laravel

SabreDAV



SUGARCRM.



ZF zend  
framework<sub>2</sub>

# PSR 1 - Basic Coding Standard

- <?php ou <?=
- Há dois tipos de arquivos, de **declaração** ou de **ação, nunca os dois ao mesmo tempo.**
- Nomes de classes seguem o padrão **StudlyCaps**.
- Nomes de métodos seguem o padrão **camelCase**.
- Constantes são escritas em letras maiúsculas com *underscores*.

# PSR 2 - Coding Style Guide

- 4 espaços para indentar.
- Linhas devem ter 80 caracteres.
- Abertura de chaves de classes e métodos na linha seguinte.
- Abertura de chaves em estruturas de controle na mesma linha.
- Visibilidade deve ser declarada.
- Deve haver uma linha em branco após a instrução **namespace** e após o conjunto de instruções **use**.

# PSR 3 - Logger Interface

- Classes de log devem seguir a interface `LoggerInterface`
- A exceção `Psr\Log\InvalidArgumentException` deve ser lançada para nível de log desconhecido.

# PSR 4 – Improved Autoloading

- Um nome de classe completamente qualificado deve ter a seguinte forma:

```
\<NamespaceName>(\<SubNamespaceNames>) *\<ClassName>
```

- O namespace de nível mais alto é o namespace do fornecedor.
- O prefixo namespace do nome da classe pode mapear para um diretório base que contenha uma pasta correspondente ao prefixo namespaces, mas não limitado a ela.

# PSR 7 – Interfaces de mensagens HTTP

Psr\Http\Message\MessageInterface

Psr\Http\Message\RequestInterface

Psr\Http\Message\ResponseInterface

Psr\Http\Message\StreamInterface

Psr\Http\Message\UriInterface

Psr\Http\Message\UploadedFileInterface

# Exercício

Crie um **cadastro de telefones** que guarde números de telefone e nomes associados a eles. Deve ser possível incluir, alterar, excluir e pesquisar um número ou o nome.

Crie esse cadastro usando **testes**.

# Bibliografia

- **Beck, K.** *TDD Desenvolvimento Guiado por Testes*. Porto Alegre. Bookman, 2010.
- **Bergmann, S.** *PHPUnit*. Disponível em <<https://phpunit.de/getting-started.html>>
- **Martin, R. C.** *Clean Code: A Handbook of Agile Software Craftsmanship*. Boston. Pearson Education, 2009.
- **McConnell, S.** *Code Complete: Um guia prático para a construção de software*. 2.ed. Porto Alegre. Bookman, 2005.
- **Sommerville, I.** *Engenharia de Software*. 8.ed. São Paulo. Pearson Addison-Wesley, 2007.