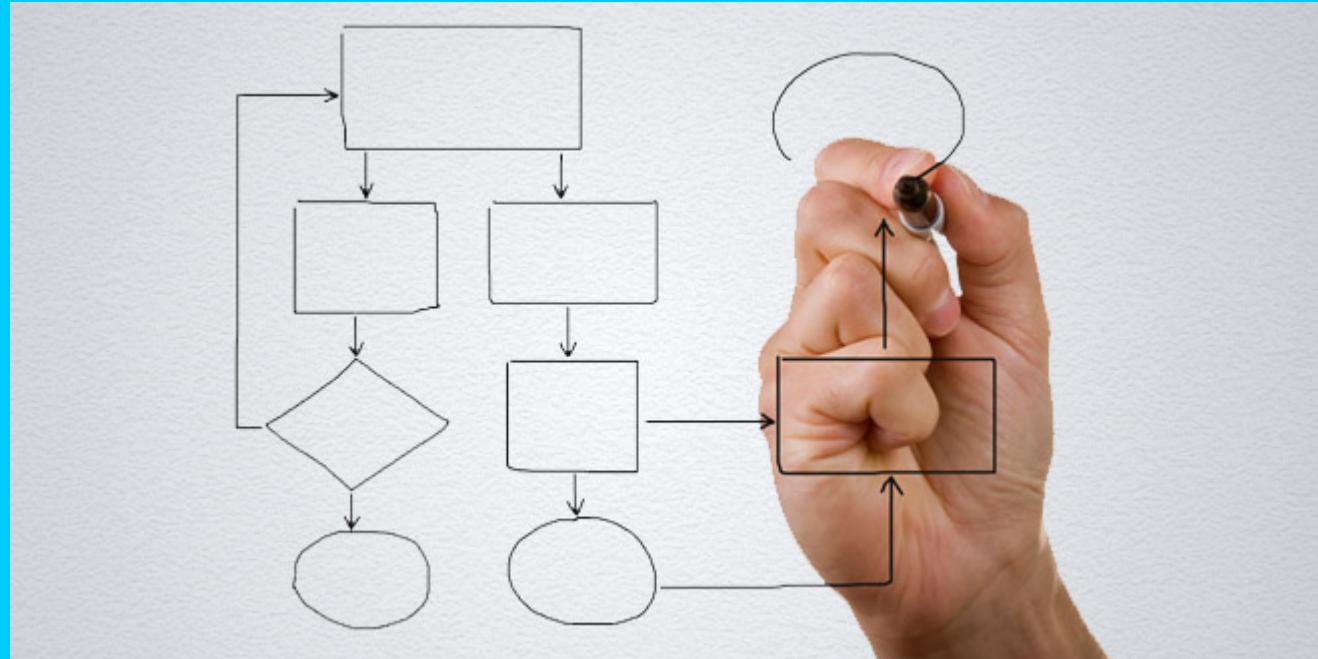


# Programação PHP orientada a objetos com testes unitários

FLÁVIO GOMES DA SILVA LISBOA

The slide features several graphical elements: 1) A diagram of an object structure showing a circle labeled 'Object' containing a green box for 'Public Methods' and an orange box for 'Private Data / Private Methods'. An 'Interaction Interface' box is connected to the object's methods. 2) The official PHP logo in a blue oval. 3) A large number '3'. 4) A diagram illustrating the Test-Driven Development (TDD) cycle with the words 'RED', 'GREEN', and 'REFACTOR' in a loop. 5) The PHPUnit logo, which includes a purple square with a white circle and the text 'PHPUnit'.

# Planejamento



# Programação PHP orientada a objetos

## Parte 3

- *Fronteiras dos testes*
- *Traits*
- *PHP 7*
- *Spl Datastructures*
- *Spl Iterators*
- *Spl Interfaces*
- *Spl Exceptions*
- *Spl File Handling*
- *Spl Miscellaneous*
- *Spl Functions*
- *xUnit*
- *Padrões de teste*
- *Testes cheirando mal*
- *Esclarecimento de dúvidas sobre o trabalho de avaliação da disciplina*

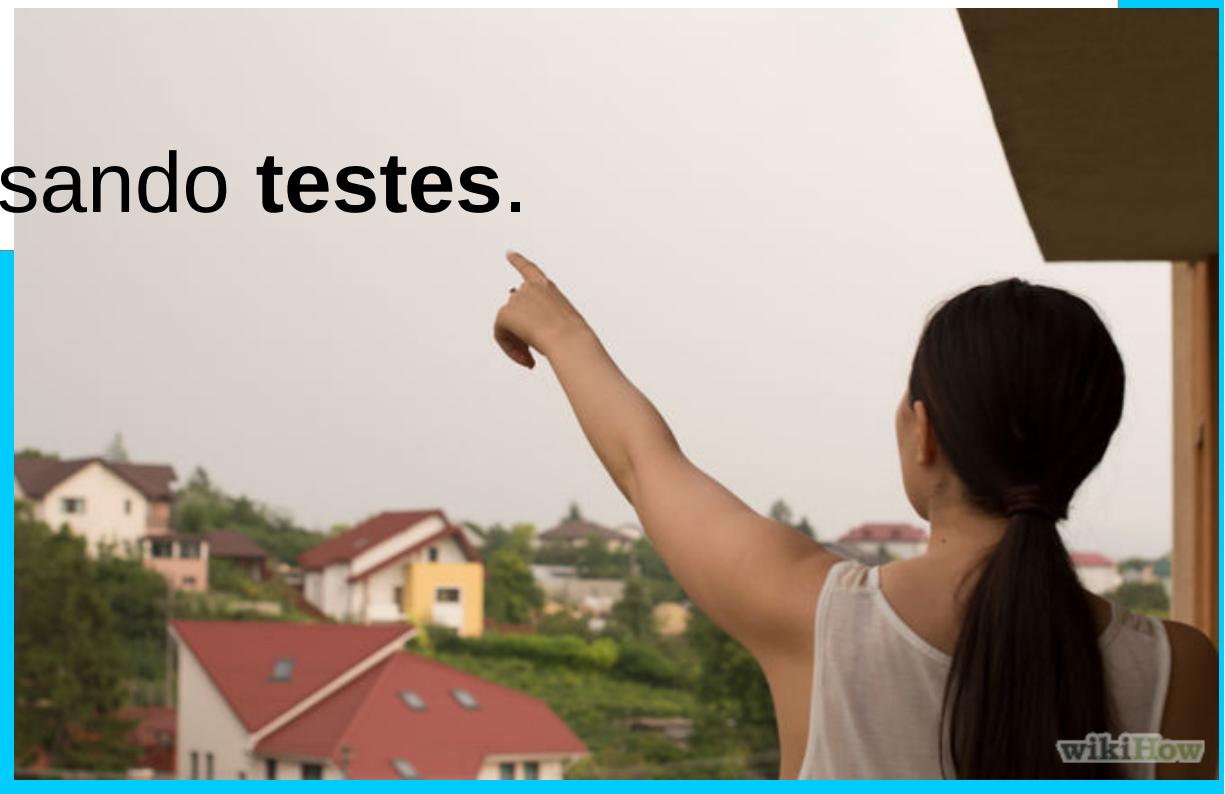
# Mas não havia um exercício?



# 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**.



# Fronteiras do Teste Unitário

- O exercício envolve o acesso a **recursos externos**.
- Como manter testes unitários se eles usarem recursos externos?



# Fronteiras do Teste Unitário

*“Algumas vezes é difícil executar um teste sob o sistema porque isso depende de outros componentes que não podem ser usados no ambiente de teste. Isso poderia ser porque eles não estão disponíveis, eles não retornarão os resultados necessários para o teste ou porque a execução deles teria efeitos colaterais indesejáveis. Em outros casos, nossa estratégia de teste requer que tenhamos mais controle ou visibilidade do comportamento interno de um teste sob o sistema.*

*Quando nós estamos escrevendo um teste no qual nós não podemos (ou escolhemos não) usar um componente de dependência real, nós podemos substituí-lo por um **Teste Dublê**. O Teste Dublê não tem de comportar-se exatamente como o componente real ele meramente tem de prover a mesma API que o componente real de modo que o teste sob o sistema pensa que ele é o componente real!”*



Gerard Meszaros (APUD Bergmann, 2014)

# Stubs

- **Stubbing** é a prática de substituir um objeto por um dublê que (opcionalmente) retorna valores de retorno configurados.



# Usando Stubs em PHPUnit

Crie um objeto **stub**:

```
$stub = $this->createMock( 'NomeDaClasse' );
```

Diga o que deve ser retornado para um determinado método:

```
$stub->method( 'nomeDoMetodo' )->willReturn( 'valor-de-retorno' );
```

# Mocks

**Mocking** é a prática de substituir um objeto por um dublê que verifica comportamentos que são esperados em chamadas indiretas.



# Stubs X Mocks

A diferença entre usar um **stub** e um **mock** é que o teste do mock não envolve a chamada direta aos métodos do mock. Na verdade, nós **injetamos** o mock em um outro objeto, que deve chamá-lo. O que queremos testar é se os métodos do mock são chamados pelo objeto real.

# Usando Mocks em PHPUnit

Crie um objeto **mock**:

```
$mock = $this->createMock( 'NomeDaClasse' );
```

Diga o que o deve ser retornado para um determinado método:

```
$mock->expects( 'comportamento Esperado' )
->method( 'nomeDoMetodo' )
->with( 'argumento1' , 'argumento2' , ... );
```

Crie um objeto **real** e **injete** o mock nele.

Chame um método do objeto real que deve chamar o método do objeto mock.

# Definindo comportamentos esperados

\$this->any()

\$this->at(int \$count)

\$this->atLeastOnce()

\$this->exactly(int \$count)

\$this->once()

\$this->never()

**QUANTAS  
VEZES?**

# Definindo argumentos

- \$this->anything()
- \$this->equalTo()
- \$this->greaterThan()
- \$this->identicalTo()
- \$this->lessThan()
- \$this->stringContains()



# Combinando argumentos

- \$this->logicalAnd()
- \$this->logicalOr()
- \$this->logicalNot()
- \$this->logicalXor()

p	q	$p \wedge q$
V	V	V
V	F	F
F	V	F
F	F	F

# Cobertura de Testes

*“Você sempre pode escrever **mais testes**. Entretanto, você rapidamente irá descobrir que somente **uma fração dos testes** que você pode imaginar são **realmente úteis**. O que você quer é escrever testes que falham mesmo quando você acha que eles devem funcionar, ou testes que tem sucesso mesmo quando você acha que eles devem falhar. Outro modo de pensar está em termos de custo/benefício. Você quer escrever testes que **retribuirão** você com **informação**.”*

Erich Gamma



# Cobertura de Testes

## **PHP\_CodeCoverage**

Ferramenta para medir **quanto** código está **coberto** por testes.

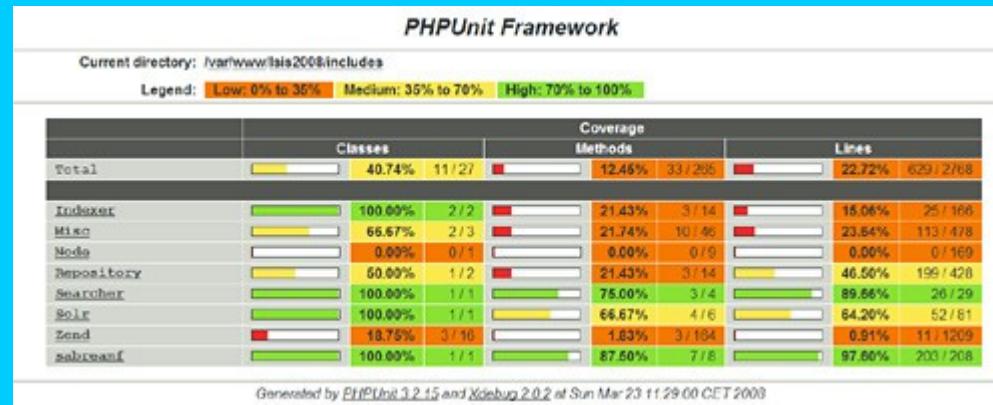
Indica cobertura por linhas, por funções e métodos e por classes e *traits*.

Mede a distribuição da cobertura, a relação entre complexidade e cobertura, a insuficiência da cobertura e os riscos de mudança.

# Cobertura de Testes

Como gerar o relatório de cobertura de testes

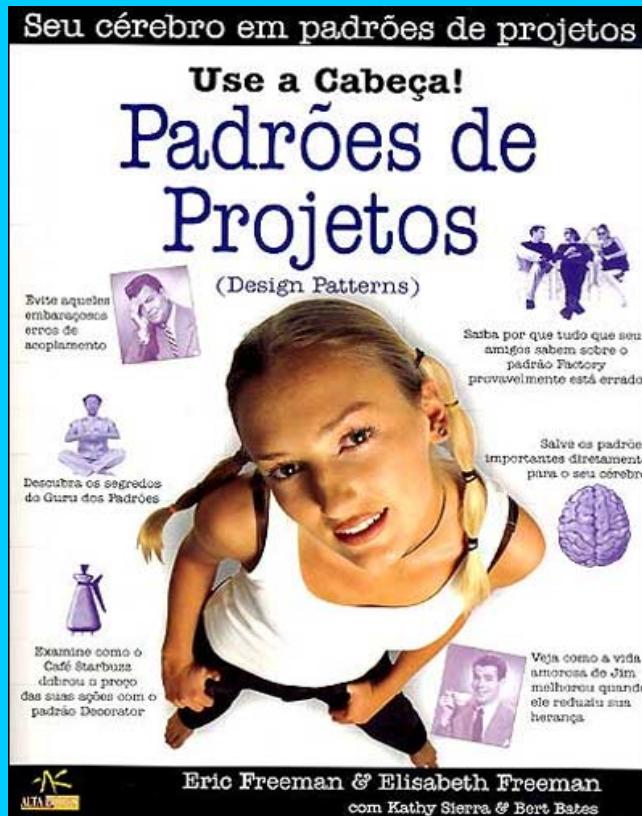
**phpunit --bootstrap tests/bootstrap.php tests/ --coverage-html [pasta do relatório] --whitelist [pasta a ser avaliada]**



# *Traits*



Não seria maravilhoso se existisse uma maneira de criar um software de modo que quando precisássemos alterá-lo, pudéssemos fazer isso com o menor impacto possível no código existente? Poderíamos perder menos tempo retrabalhando o código e mais tempo para permitir que o programa faça coisas mais legais...



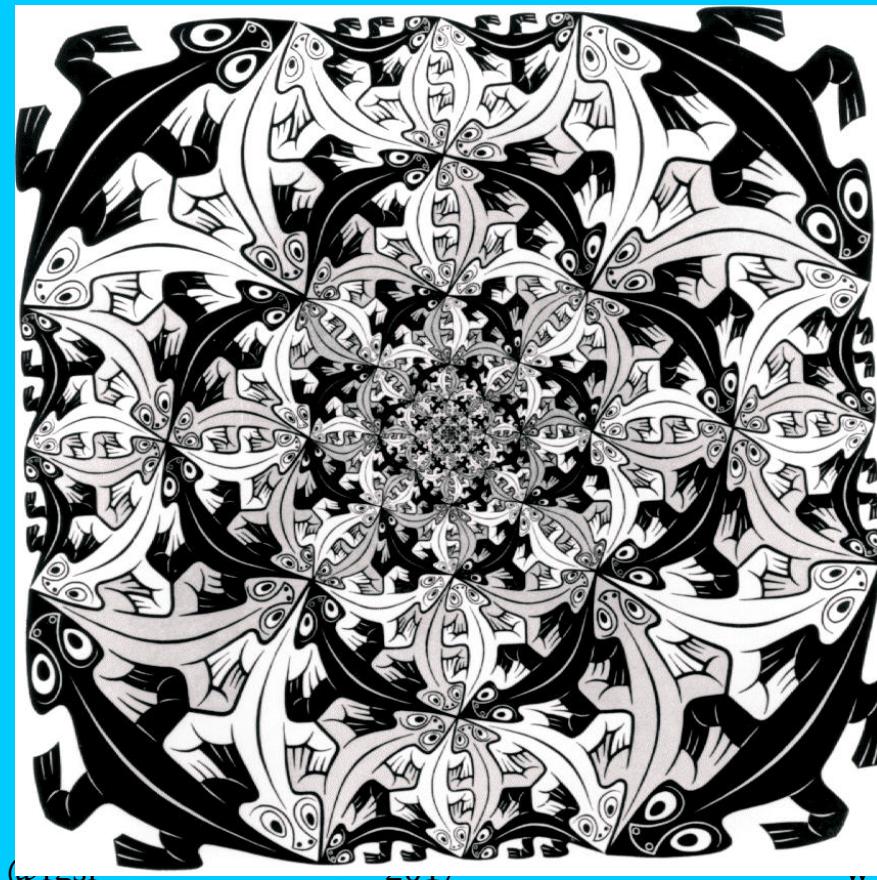
[www.fgsl.eti.br](http://www.fgsl.eti.br)



[flavio.lisboa@fgsl.eti.br](mailto:flavio.lisboa@fgsl.eti.br)

# Princípio de Design

*“Identifique os aspectos de seu aplicativo que variam e separe-os do que permanece igual”.*



© 2017

[www.fgsl.eti.br](http://www.fgsl.eti.br)

Eric e Elisabeth Freeman



[flavio.lisboa@fgsl.eti.br](mailto:flavio.lisboa@fgsl.eti.br)

# Resumo

*Traits* resolvem um problema relativo à **limitação** do reuso por herança em orientação a objetos e como são implementados no PHP 5.4.

# Tudo começou aqui

## European Conference on Object-Oriented Programming



*Paper*

# *Traits: “unidades de comportamento que podem compor algo”*

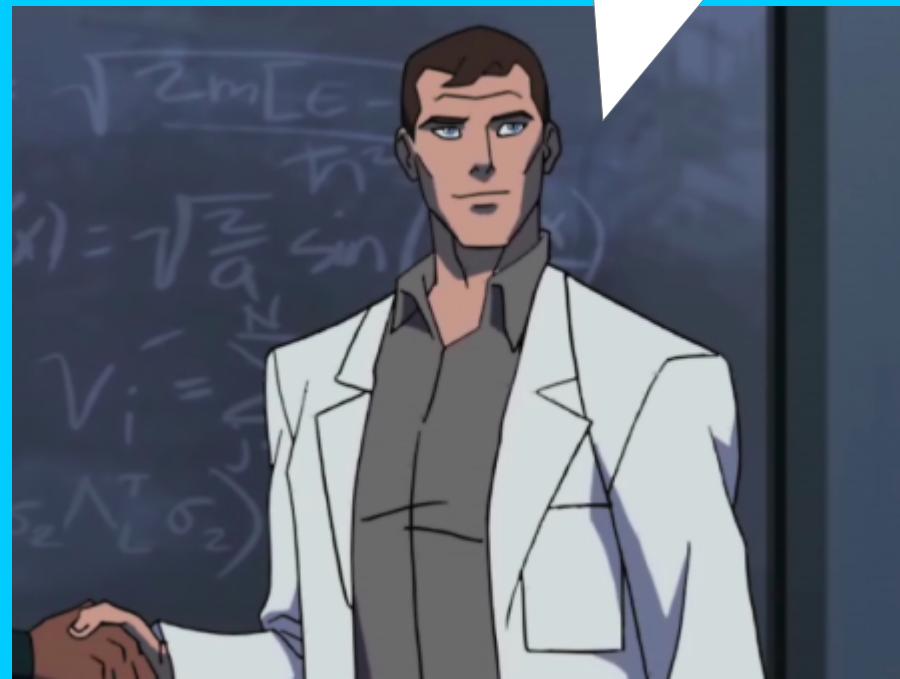
Schärli, Ducasse, Nierstraz e Black



**Fato:** “A herança é o **mecanismo de reuso fundamental** nas linguagens de programação orientadas a objetos”



**Fato:** “Todas as variantes de herança  
sofrem de problemas conceituais e  
práticos”



**Fato:** “A herança é um mecanismo com muitos **significados** e interpretações **conflitantes**”

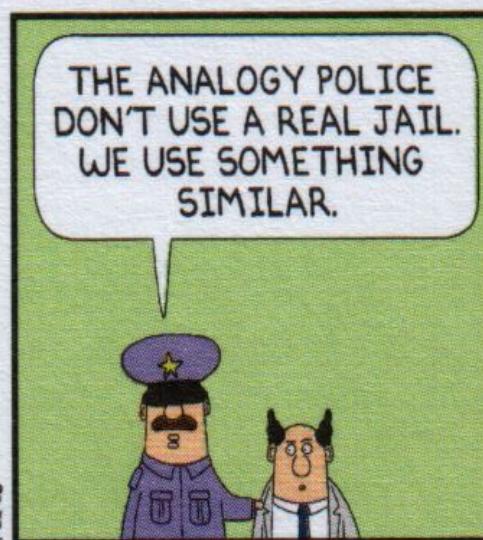
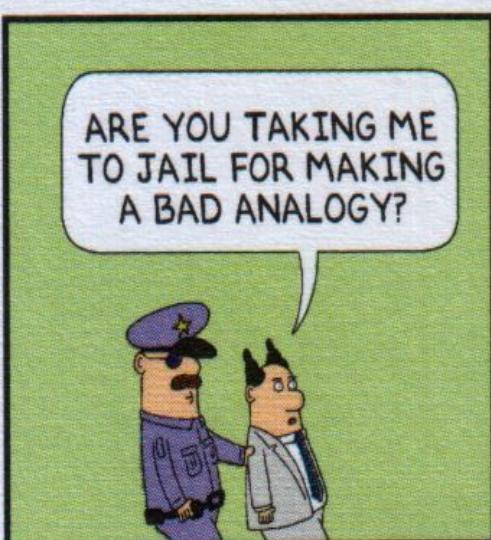
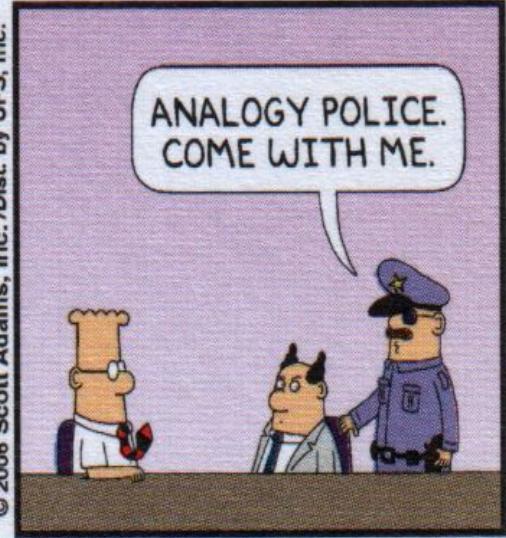
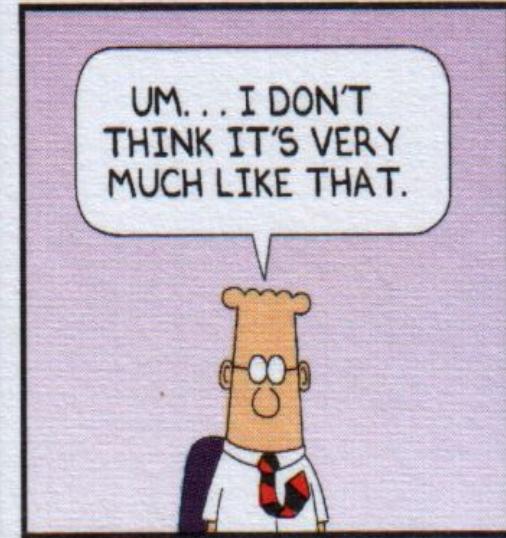
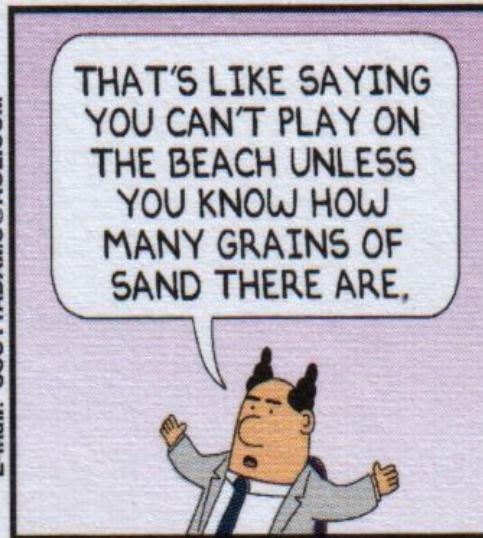
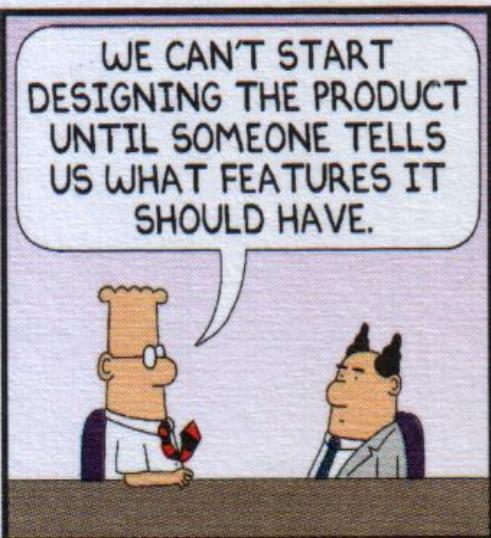


# A programação de computadores se apropria de termos de outras áreas de conhecimento fazendo analogias

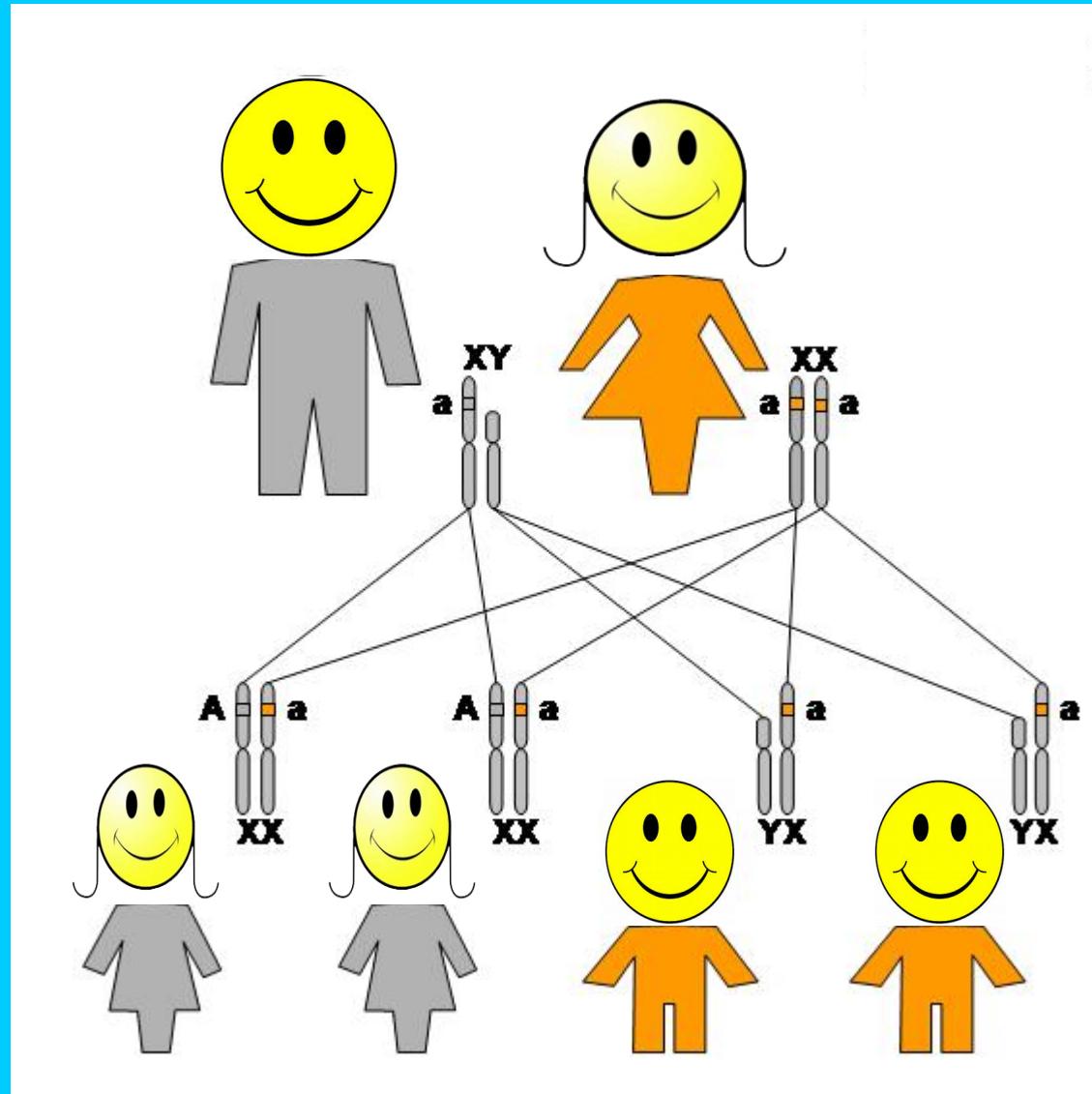


# DILBERT®

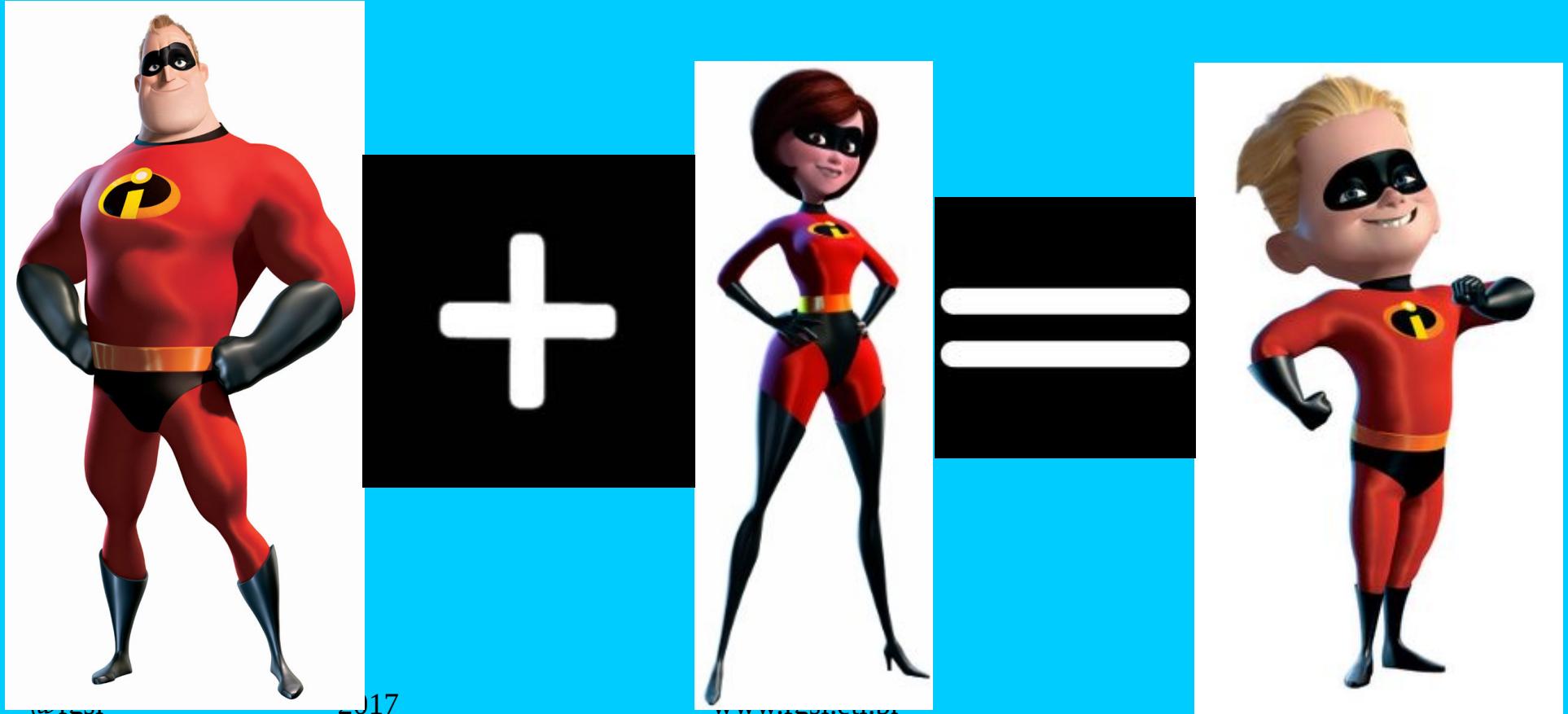
BY SCOTT ADAMS



# Seria a herança genética?



Na herança genética, cada filho  
recebe uma **combição** de  
características do pai e da mãe... e  
isso é *naturalmente* probabilístico



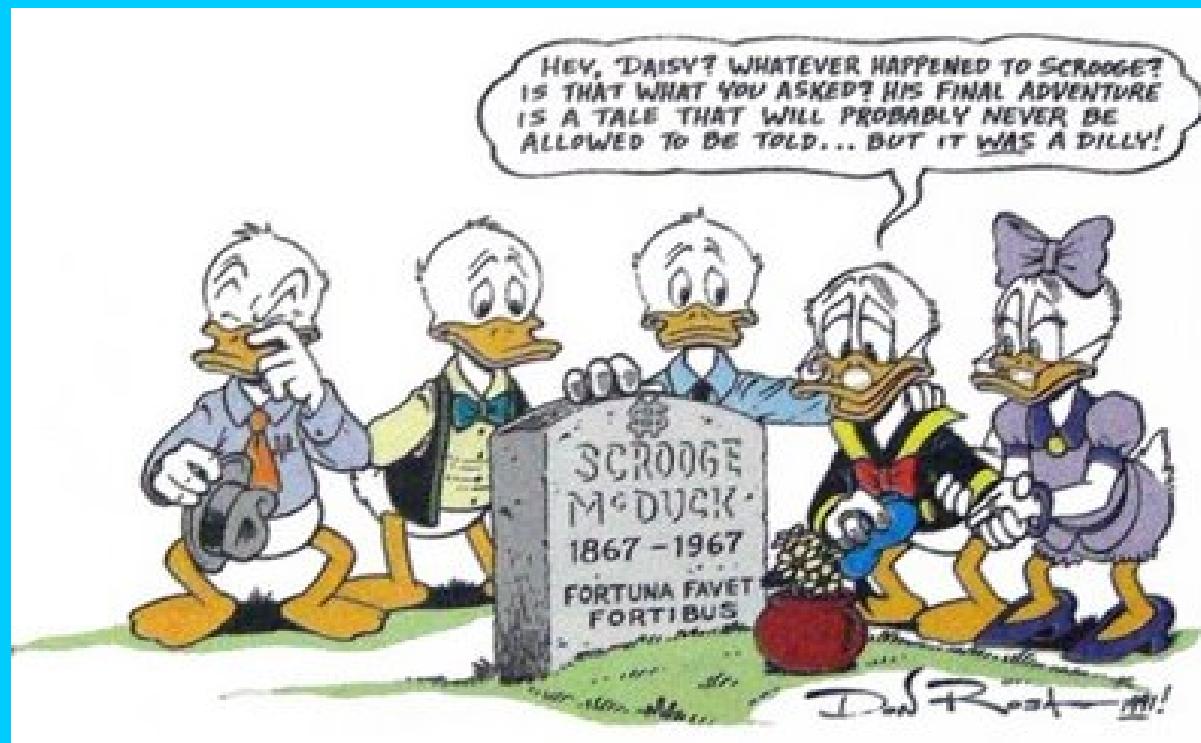
# Existem **dois** tipos de herança na programação orientada a objetos

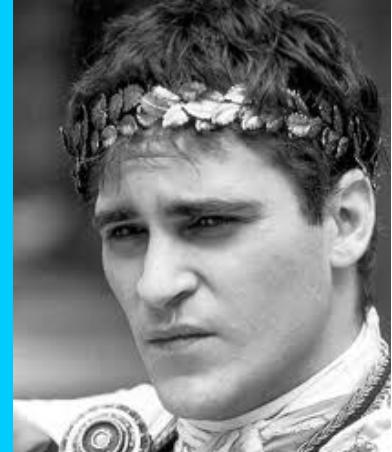
- **Simples**: uma classe herda TODAS as características herdáveis de outra
- **Múltipla**: uma classe herda TODAS as características herdáveis de mais de uma classe.

# Esses dois tipos são determinísticos

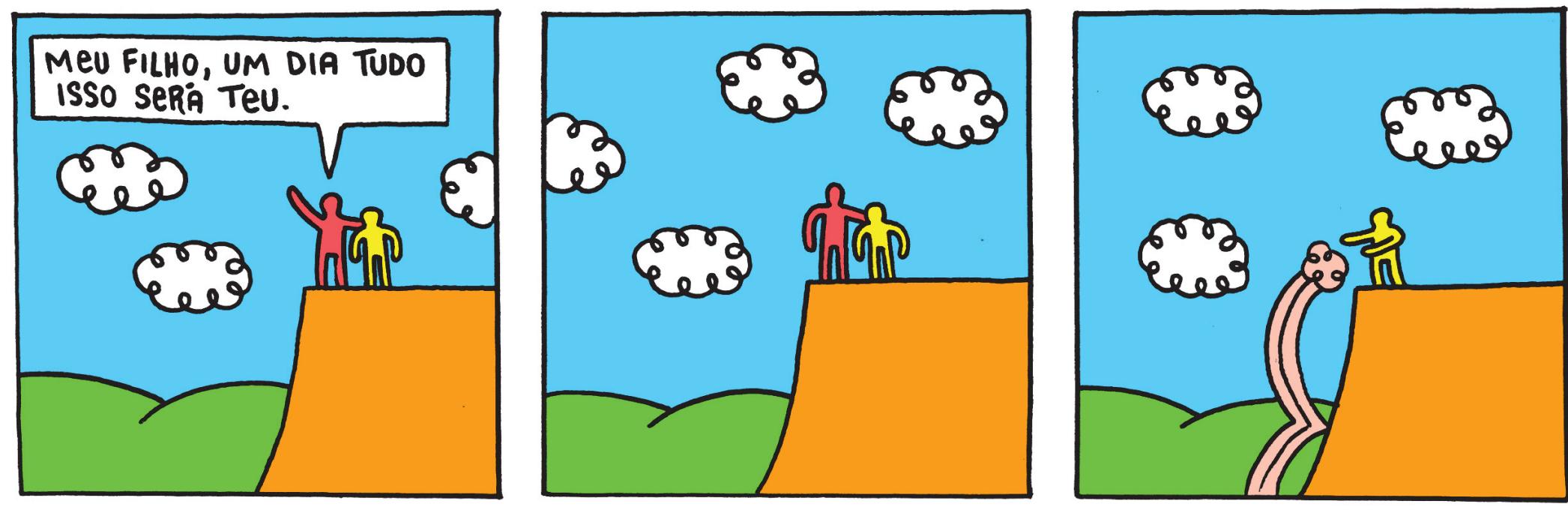
- Em ambos os tipos, existe uma forma de dizer o que será ou não herdado, que é o escopo de visibilidade de atributos e métodos.

A herança em orientação a objetos  
seria um testamento, em que as  
classes pais transmitem todos os  
seus bens (atributos e métodos)  
para as classes filhas?





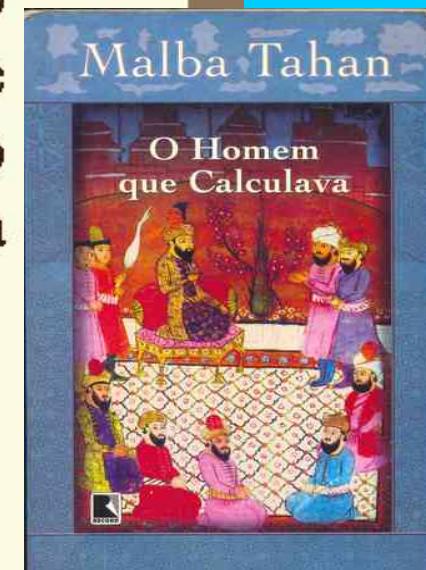
# Mas esta **não** é a herança simples da orientação a objetos



# Nem esta:

## O PROBLEMA DOS 35 CAMELOS

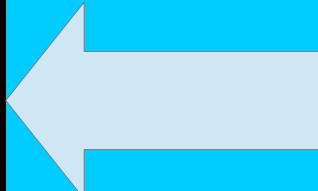
- Nosso herói Beremiz viajava com um amigo pelo deserto, ambos montados em um único camelo, quando encontram três homens discutindo acaloradamente. Eram três irmãos. Haviam recebido uma herança de 35 camelos do pai, sendo a metade para o mais velho, a terça parte para o irmão do meio e a nona parte para o irmão mais moço. O motivo da discussão era a dificuldade em dividir a herança:



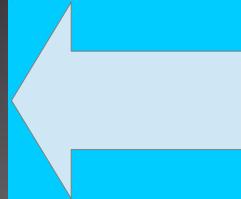
O problema com a analogia do testamento, é que nesse caso cada herdeiro leva uma **parte** da herança.

E não é isso que acontece com a orientação a objetos, na qual a cada classe filha leva **toda** a herança.

# A herança simples da orientação a objetos, então, é esta:



# Ou melhor...

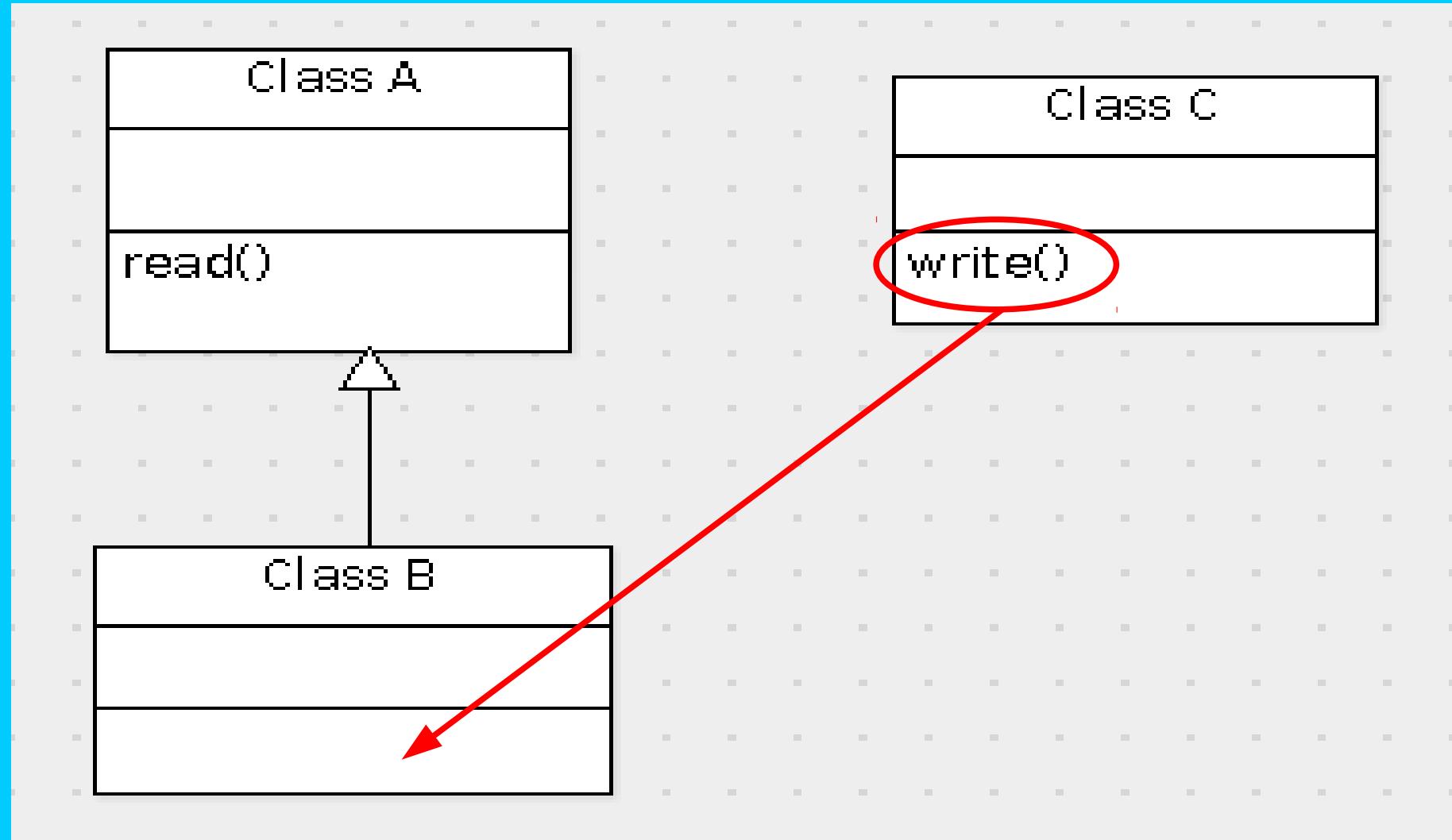


Na herança simples, várias classes podem reutilizar atributos e métodos de uma classe ancestral.

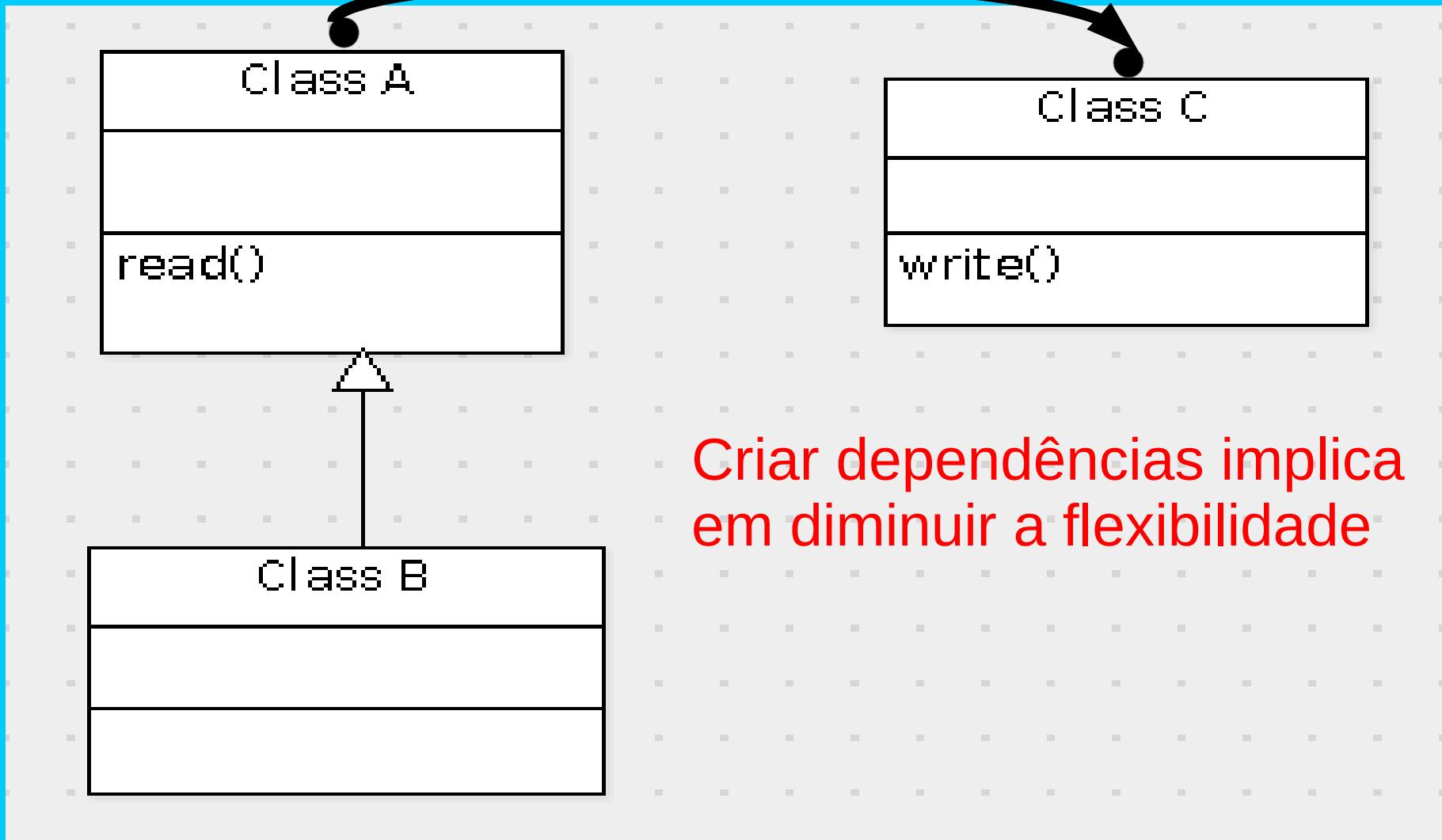
Mas o reuso não é **pleno**, já que, por herdar de apenas uma classe, você pode acabar “copiando” atributos e métodos de outra classe, gerando **código duplicado**.

E o código duplicado é **mutante**: você não tem um **controle** fácil sobre **implementações**.

# Herança simples e código duplicado

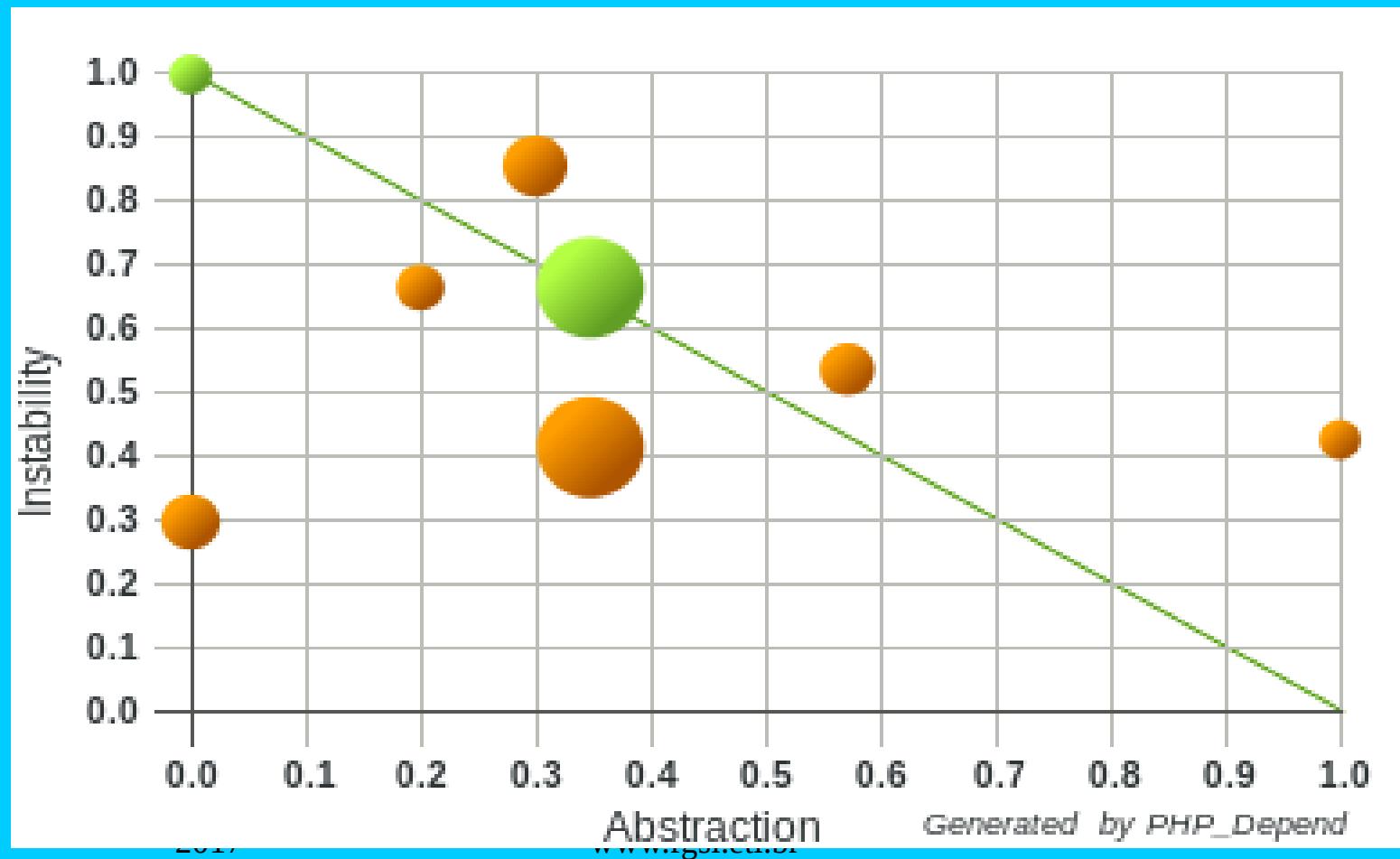


# Nem pense nisso!

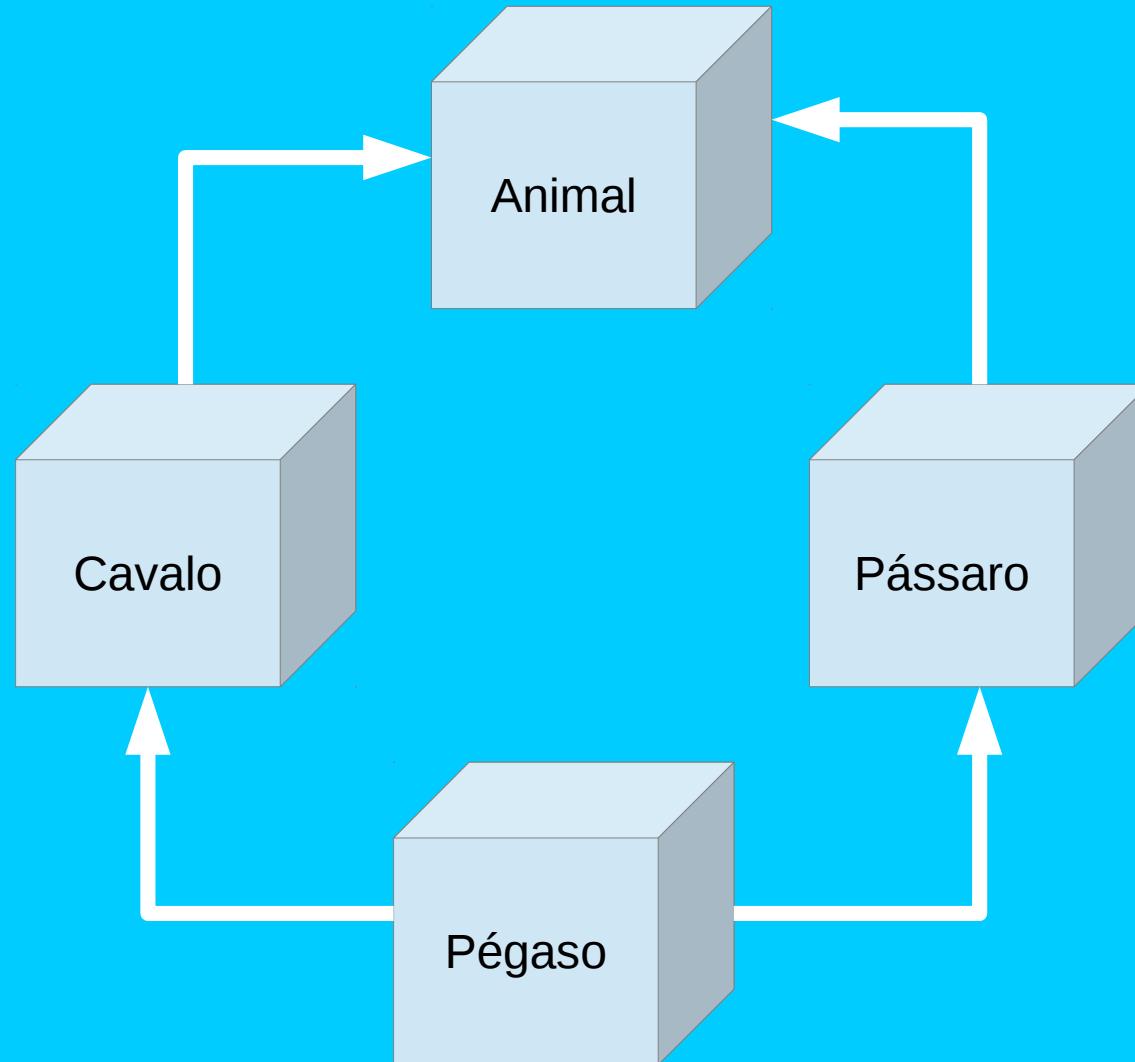


Criar dependências implica  
em diminuir a flexibilidade

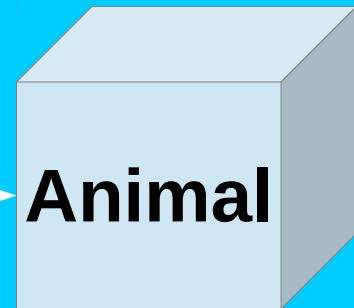
Na arquitetura do software é necessário encontrar um ponto de equilíbrio entre instabilidade e abstração



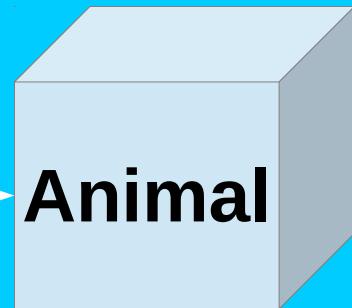
# E a herança múltipla?



# Seria isto?



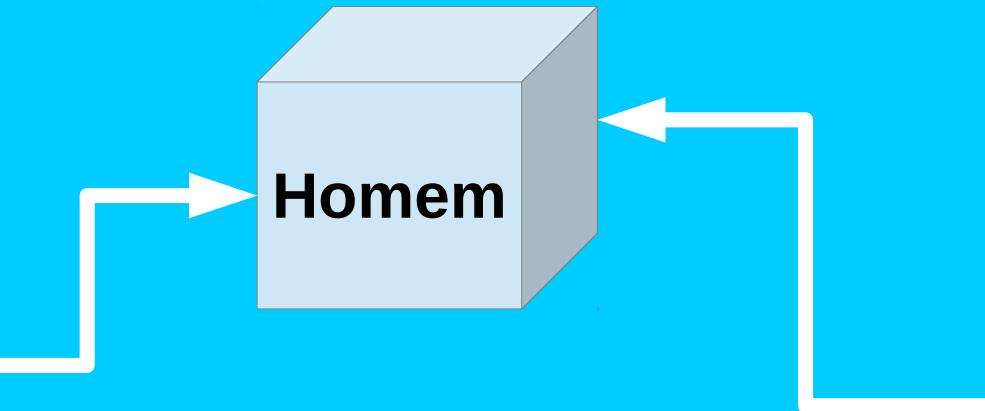
# Seria isto?



# NÃO!



# Seria mais parecido com isso:



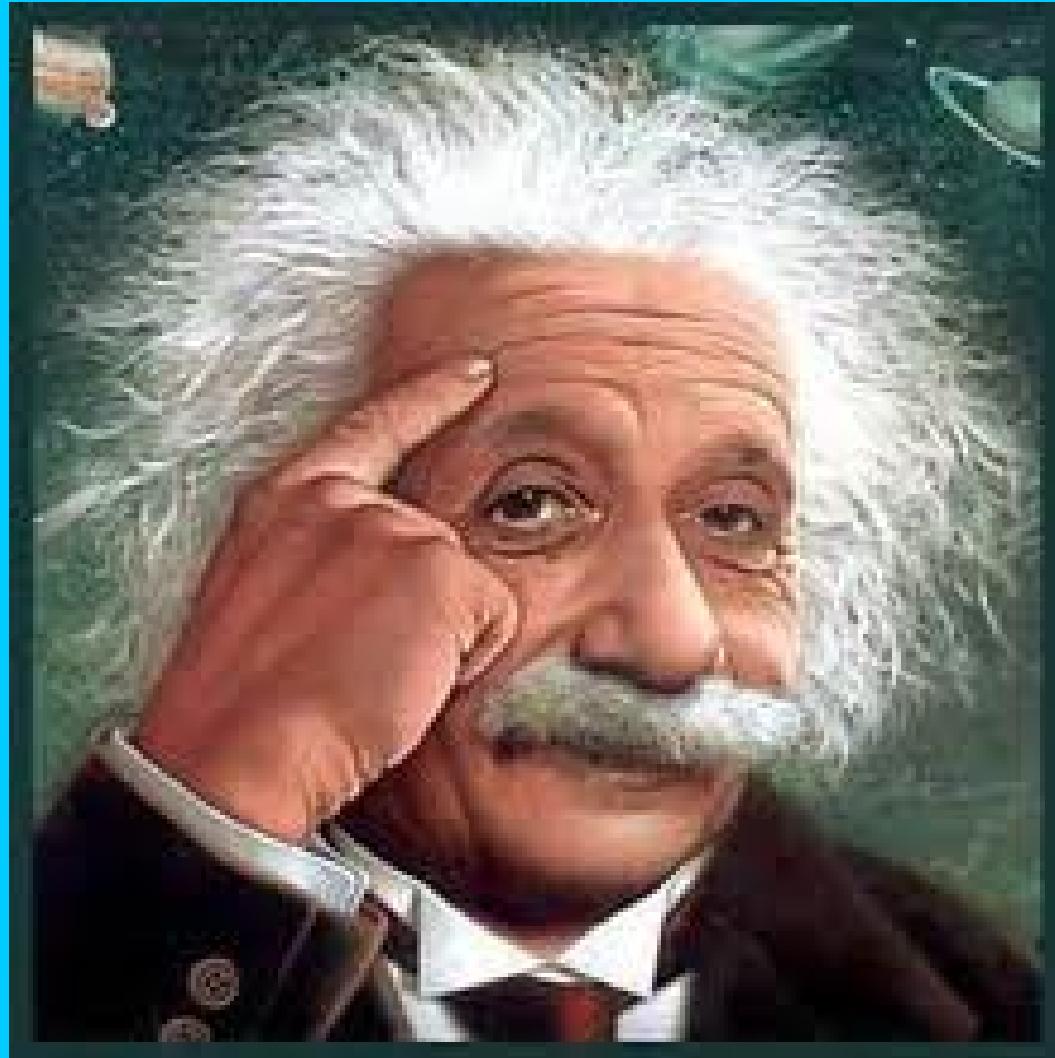
*“A herança múltipla é boa, mas não há um bom modo de fazê-la”*

Steve Cook

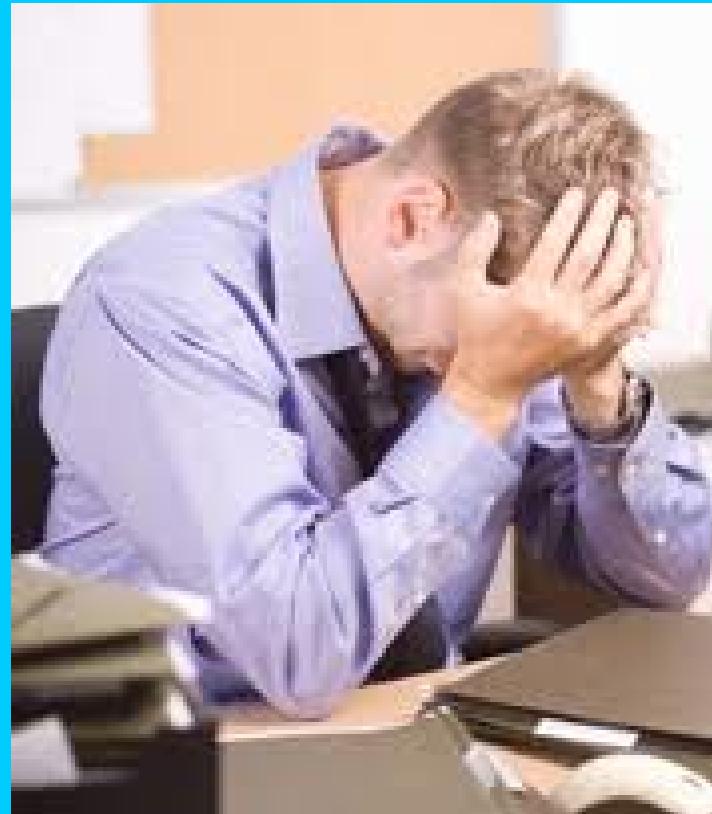
A herança múltipla permitiria **combinar** atributos e métodos de várias classes em uma nova classe.

Mas existem **vários** problemas decorrentes da herança múltipla... por isso PHP não a implementa.

# Ah, mas eu posso usar interfaces...



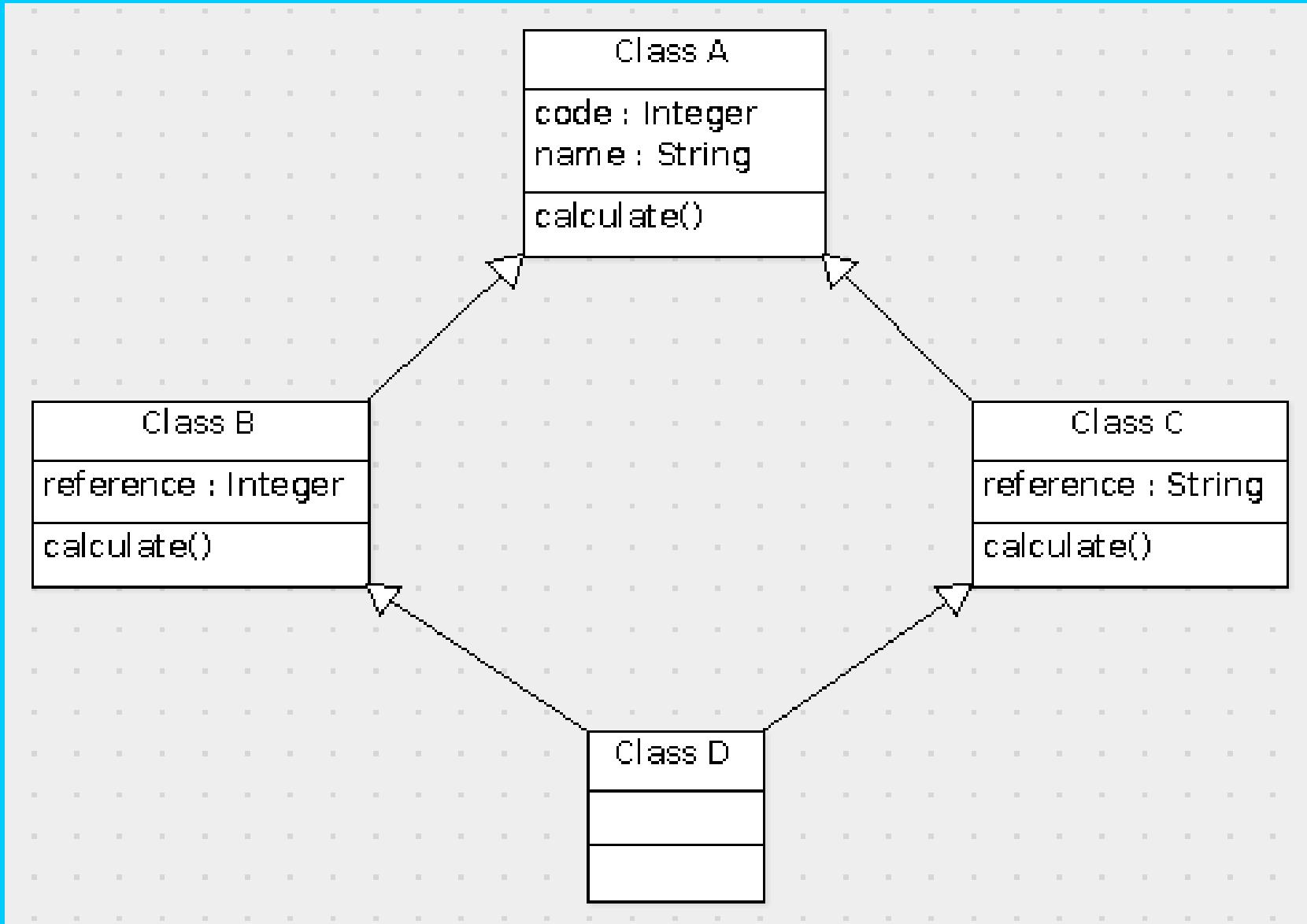
Sim, mas interfaces não tem implementação. O problema não é padronizar a comunicação, mas reutilizar código.



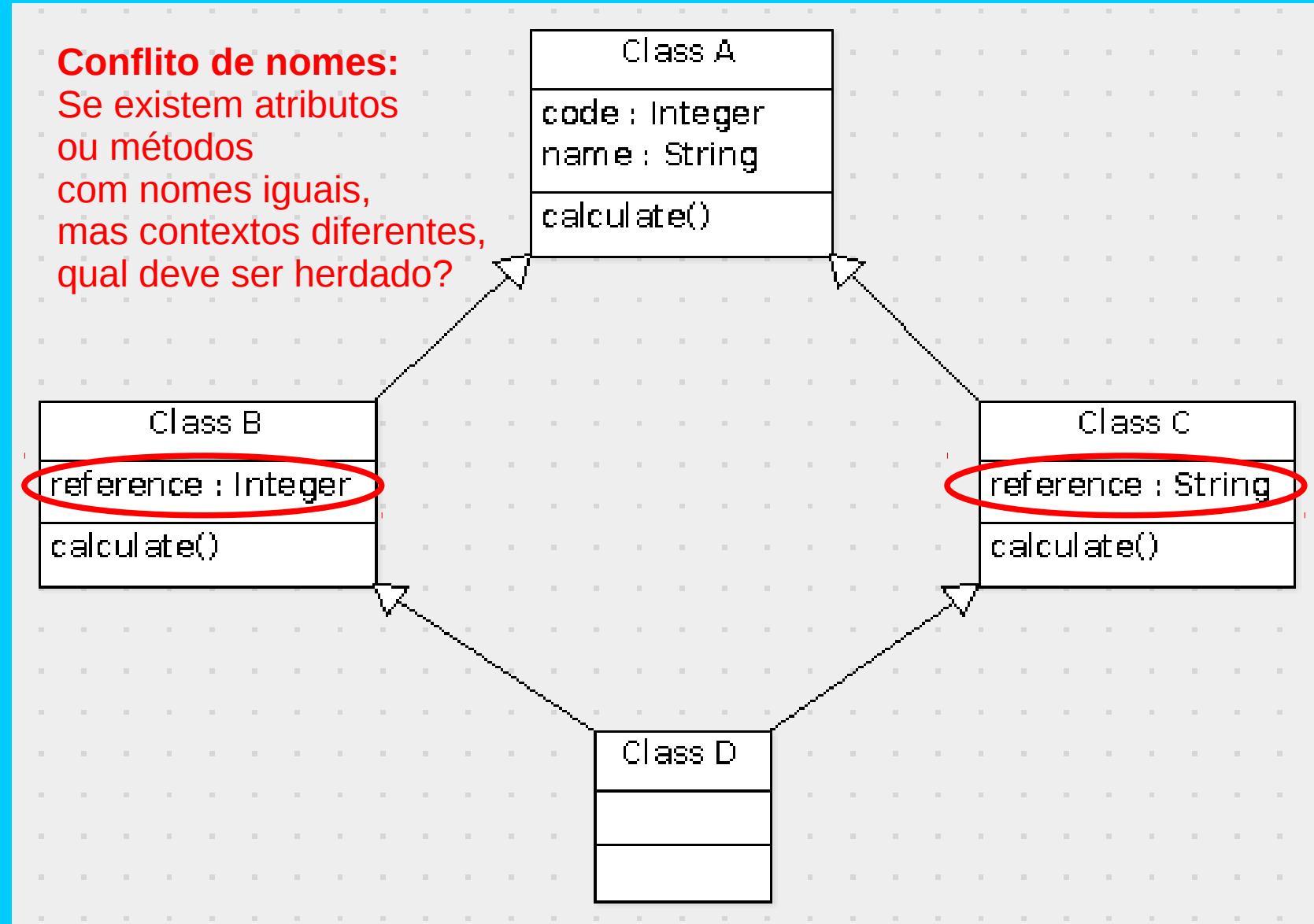
Existe um problema:  
o Diamante da Morte



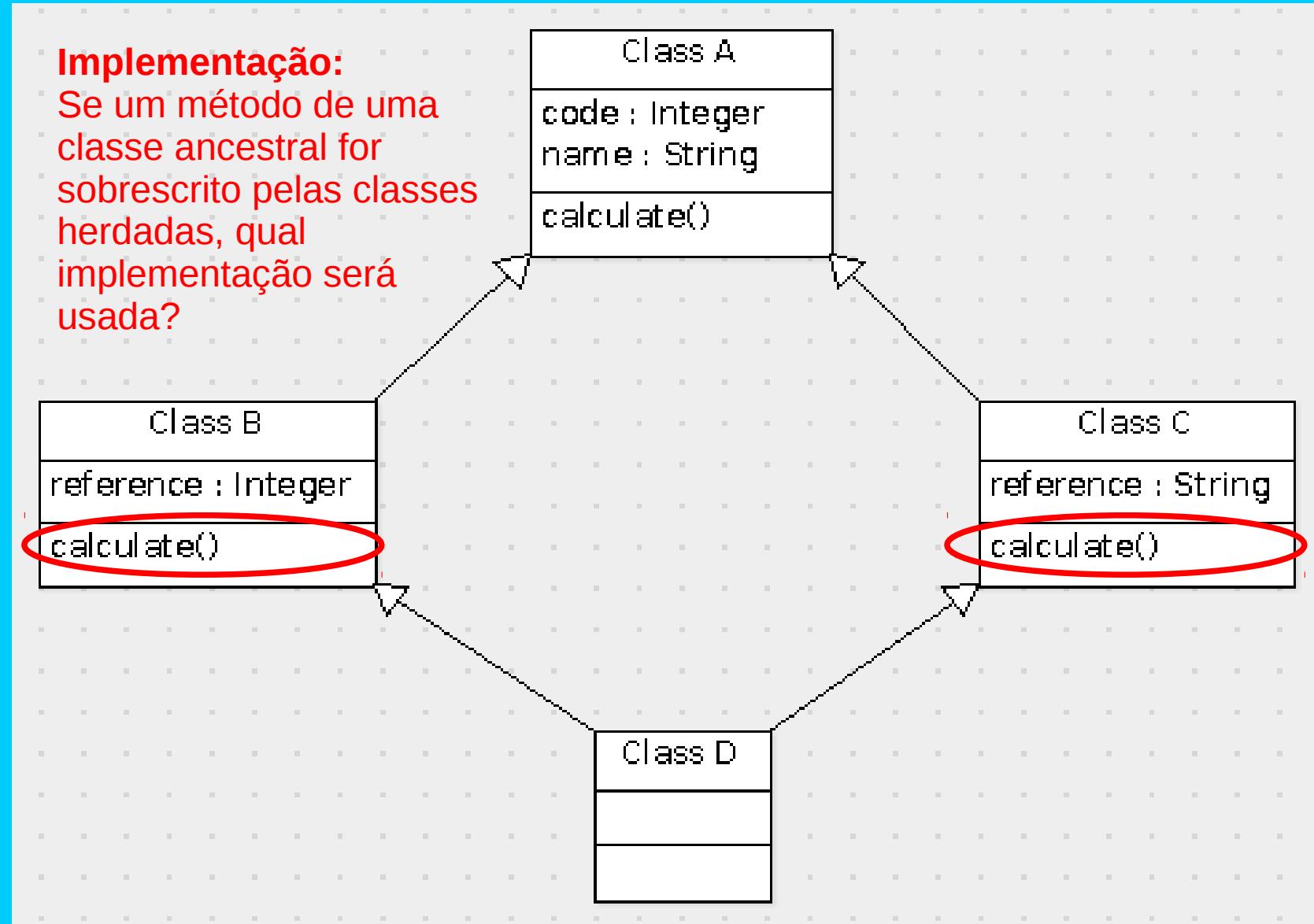
# Problema do Diamante



# Problema do Diamante



# Problema do Diamante



# Herança múltipla não é genética...

Herança múltipla traz tudo de todos os pais.  
E nem sempre (talvez na maioria dos casos)  
queremos combinar tudo de todos, mas algumas  
coisas de todos.

Não queremos gerar um clone (até porque não dá  
pra gerar um clone de mais de um original) mas  
uma combinação de traços (atributos e métodos).

# Composição de classes é um paliativo

Você pode contornar a inexistência de herança múltipla fazendo com que uma classe contenha outra, em vez de estendê-la.

Com essa abordagem, uma classe pode artificialmente herdar tanto atributos quanto métodos de múltiplas classes.

Mas persiste o problema de que é tudo ou nada.

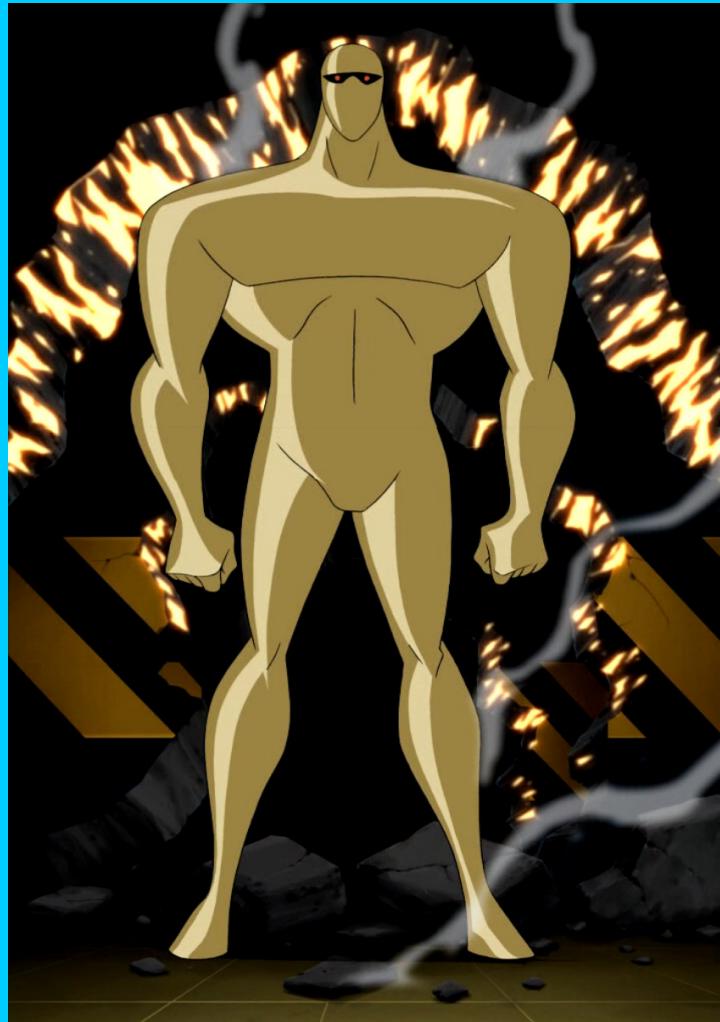
# Composição de classes é um paliativo

```
<?php
namespace DCComics\Characters;

class Amazo
{
    private $decorators = array();

    public function __construct()
    {
        $this->decorators[] = new Superman();
        $this->decorators[] = new MartianManhunter();
    }
}
```

# Composição de classes é um paliativo



# Composição de classes é um paliativo



# *Paradoxo da Herança de Classes*

Uma classe tem um papel primário de geradora de instâncias, logo ela deve ser completa.

Mas como unidade de reuso, uma classe deve ser pequena, com a menor implementação possível.

Isso é contraditório!



# *Separação de papéis*



Se a classe deve ser a geradora de instâncias, alguém tem de assumir o papel de unidade de reuso. É aí que entram os **traits**.

*Traits* são unidades primitivas de reuso de código.

# *Traits*

- Um *trait* provê um conjunto de métodos que implementam comportamento.
- Classes e *traits* podem ser compostos por outros *traits*.
- Métodos conflitantes devem ser explicitamente resolvidos.
- A composição com *traits* não afeta a semântica da classe. É como se os métodos fossem declarados nela.

# *Classes e Verbos*

- Herança de classe: Ser
- Composição de classe: Ter (ter é melhor que ser)
- Traits: Usar (usar é melhor que ter)

# Declarando um *trait* em PHP

```
trait GenericTrait
{
    public function doSomeSingle()
    {
        echo 'something single';
    }
}
```

*Traits* não podem ser instanciados, apenas usados!

# Usando um *trait* em PHP

```
class ClassC
{
    use SpecialTrait;
}
```

Se uma classe herda um método que tem o mesmo nome de um método contido em um *trait*, o *trait* sobrescreve o método herdado.

Tesoura corta papel. *Trait* corta classe mãe.

# Usando vários *traits* em PHP

```
class Amazo
{
    use Aquaman, Flash;
}
```



# Sobrescrevendo um *trait* em PHP

```
class ClassB
{
    use GenericTrait;

    public function doSomeSingle()
    {
        echo 'Something single in
fact';
    }
}
```

O método da classe tem precedência sobre o *trait*!

# Resolução de conflitos

```
trait Fifer
{
    public function buildHouse()
    {
        echo 'building a straw house';
    }
}
```

# Resolução de conflitos

```
trait Fiddler
{
    public function buildHouse()
    {
        echo 'building a stick house';
    }
}
```

# Resolução de conflitos

```
trait Practical
{
    public function buildHouse()
    {
        echo 'building a brick house';
    }
}
```

# Resolução de conflitos

```
class PerfectPig
{
    use Fifer, Fiddler, Practical;
}
```



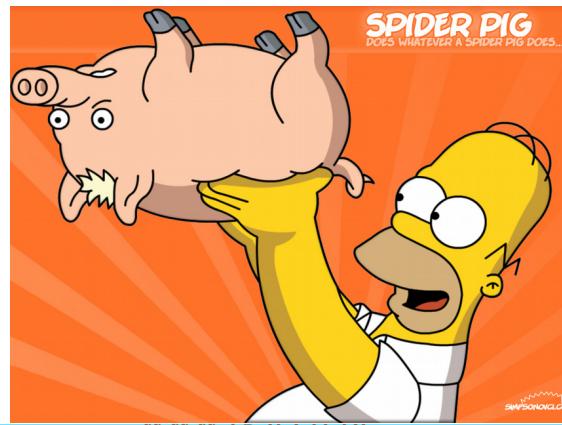
# Resolução de conflitos

```
class PerfectPig
{
    use Fifer, Fiddler, Practical
    {
        Practical::buildHouse insteadof
Fifer, Fiddler;
    }
}
```



# Resolução de conflitos

```
class PerfectPig
{
    use Fifer, Fiddler, Practical
    {
        Practical::buildHouse insteadof
Fifer, Fiddler;
        Fifer::buildHouse as buildShitHouse;
    }
}
```



# *Trait > (Interface + Implementação)*

- Uma classe **não pode** implementar duas interfaces que tenham métodos com mesmo nome.
- Interfaces **não suportam atributos**, apenas constantes.
- Os métodos de uma interface **tem de ser públicos**.

# *Traits suportam atributos*

- Você pode definir atributos, inclusive estáticos, da mesma forma como faz em classes.

```
trait StaticTrait
{
    public static $instances;
    public $description;
}
```

# *Traits podem usar variáveis estáticas*

- Só não podem defini-las.

```
trait StaticTrait
{
    public function $playDice()
    {
        static $side = 1;
        $side++;
        if ($side > 6) $side = 1;
        return $side;
    }
}
```

# Classes podem alterar a visibilidade de métodos de *traits*

- Métodos públicos podem se tornar privados ou protegidos.

```
class ClassC
{
    use SpecialTrait
    {
        doSomeSpecial as protected;
    }
}
```

# *Traits* podem declarar métodos abstratos

- Dessa forma, um *trait* pode também se tornar um contrato, como uma interface

```
trait BaseTrait
{
    public function doSomeAbstract();
}
```

# *Traits podem usar traits*

- Métodos públicos podem se tornar privados ou protegidos.

```
trait OneTrait
{
    use OtherTrait;
}
```

# *Existem funções para Traits*

- **class\_uses**: retorna os *traits* usados pela classe dada.
- **get\_declared\_traits**: retorna os array de todos os *traits* declarados.
- **trait\_exists**: verifica se o *trait* existe.

# Vamos aplicar *traits*?



# Como encontrar código duplicado?

## PHP Copy/Paste Detector (PHPCPD)

<https://github.com/sebastianbergmann/phpcpd>



# Tipificação de Variáveis

*“Em algumas linguagens, todo valor de tempo de execução inclui uma marca (ou alguma outra informação) para indicar seu tipo. Isso é chamado de tipificação **dinâmica** ou **latente**(...)”*

*Em uma linguagem com tipificação dinâmica, é possível saber, em tempo de execução, quando uma operação é apropriada ou inapropriada: simplesmente verificando as marcas”.*

Friedman, Wand e Haynes (2001, p. 136)

# Tipificação de Variáveis

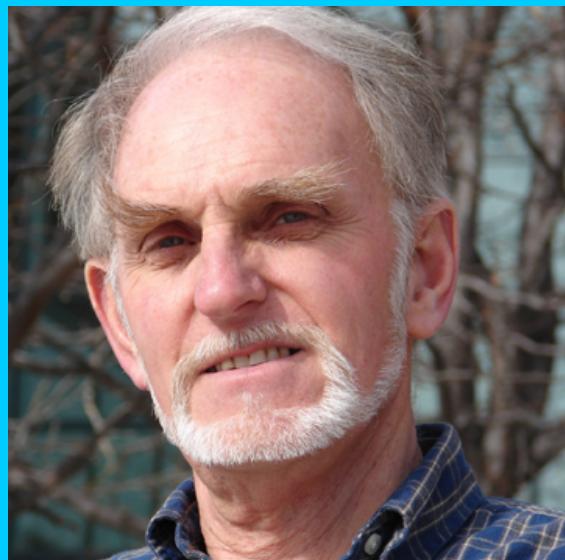
*“O tipo de uma variável geralmente não é definido pelo programador: isto é decidido em **tempo de execução** pelo PHP, dependendo do contexto na qual a variável é usada.”*

[http://php.net/manual/pt\\_BR/language.types.intro.php](http://php.net/manual/pt_BR/language.types.intro.php)

# Tipificação de Variáveis

*“A seguir, apresentamos uma definição simples, mas incompleta, de uma linguagem com **tipificação forte**: ela consiste na relação em que cada nome de um programa escrito na linguagem tem um **único tipo** associado a ela”*

Sebesta (2000, p. 171)



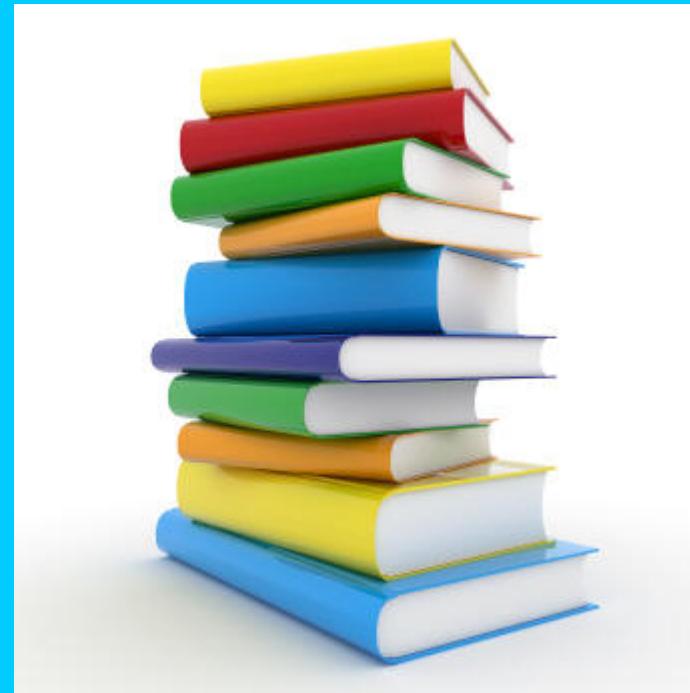
# PHP RFC: Scalar Type Declarations

- Diretiva `declare(strict_types=1)`
- Obriga a declaração dos tipos dos argumentos para os primitivos int, float, string e bool e do tipo de retorno
- Programado para o PHP 7



# SIM! PHP TEM RFCs!

- <https://wiki.php.net/rfc>



# *Spl Datastructures*

- `SplDoublyLinkedList`
- `SplStack`
- `SplQueue`
- `SplHeap`
- `SplMaxHeap`
- `SplMinHeap`
- `SplPriorityQueue`
- `SplFixedArray`
- `SplObjectStorage`

# *Spl Datastructures*

## **SplDoublyLinkedList**

*Uma lista ligada linear é uma estrutura na qual “cada item na lista é chamado nó e contém dois campos, um campo de informação e um campo do endereço seguinte. O campo de informação armazena o real elemento da lista”*

Tenenbaum (1995, p. 224)



# *Spl Datastructures*

## **SplStack**

*“Uma pilha é um conjunto ordenado de itens no qual novos itens podem ser inseridos e a partir do qual podem ser eliminados itens em uma extremidade chamada topo da pilha”.*

Tenenbaum et alli (1995, p. 86)



# *Spl Datastructures*

## **SplQueue**

*“Uma **fila** é um conjunto ordenado de itens a partir do qual podem-se eliminar itens numa extremidade (chamada **início** da fila) e no qual podem-se inserir itens na outra extremidade (chamada **final** da fila)”.*

Tenenbaum et alli (1995, p. 207)

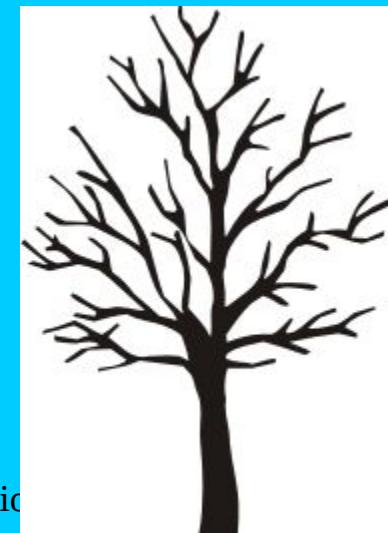


# *Spl Datastructures*

## **SplHeap**

*“Uma árvore binária é um conjunto finito de elementos que está vazio ou é particionado em três subconjuntos disjuntos. O primeiro subconjunto contém um único elemento, chamado **raiz** da árvore. Os outros dois subconjuntos são em si mesmos árvores binárias, chamadas **subárvores esquerda e direita** da árvore original. Uma subárvore esquerda ou direita pode estar vazia. Cada elemento de uma árvore binária é chamada **nó** da árvore.”*

Tenenbaum (1995, p. 303)



# *Spl Datastructures*

## **SplFixedArray**

O array PHP é **elástico**. Podemos adicionar quantos elementos quisermos. Ele também pode ser usado como um **mapa de dados**, aceitando strings como chaves.

**SplFixedArray** cria um array de **tamanho fixo**, que admite apenas **índices numéricos**.



# *Spl Datastructures*

## **SplObjectStorage**

Essa classe gera um objeto que **agrega** vários objetos.





KEEP  
CALM  
AND  
STUDY  
DATA STRUCTURES

# Processando coleções de objetos

A estrutura **foreach** pode percorrer os elementos de um objeto, desde que ele implemente a interface **Iterator**.

```
foreach($objects as $object) {  
}  
}
```

# *Spl Iterators*

- AppendIterator
- ArrayIterator
- CachingIterator
- CallbackFilterIterator
- DirectoryIterator
- EmptyIterator
- FilesystemIterator
- FilterIterator
- GlobIterator
- InfiniteIterator
- IteratorIterator
- LimitIterator

# *Spl Iterators*

- MultipleIterator
- NoRewindIterator
- ParentIterator
- RecursiveArrayIterator
- RecursiveCachingIterator
- RecursiveCallbackFilterIterator
- RecursiveDirectoryIterator
- RecursiveFilterIterator
- RecursiveIteratorIterator
- RecursiveRegexIterator
- RecursiveTreeIterator
- RegexIterator



KEEP  
CALM  
AND  
ITERATE  
ON

# *Spl Interfaces*

- Countable
- OuterIterator
- RecursiveIterator
- SeekableIterator
- SplObserver
- SplSubject



KEEP  
CALM  
AND  
PROGRAM TO AN INTERFACE  
NOT AN IMPLEMENTATION

# Princípio de Projeto Reutilizável Orientado a Objetos

*“Programe para uma interface, não para uma implementação.*

*Não declare variáveis como instâncias de classes concretas específicas. Em vez disso, prenda-se somente a uma interface definida por uma classe abstrata”.*

Gamma et alli. (2000, p. 33)

# Princípio de Projeto Reutilizável Orientado a Objetos

Em vez de:

```
/**  
 * @var ClassName  
 */  
private $attribute;
```

Faça:

```
/**  
 * @var InterfaceName  
 */  
private $attribute;
```

# Exceções



Toda regra tem exceção. E se toda regra tem exceção, então, esta regra também tem exceção e deve haver, perdida por aí, uma regra absolutamente sem exceção.

(Millôr Fernandes)

```
try {  
}  
} catch (Exception $e)  
{  
}  
}
```

# Exceções

O processamento só é desviado para dentro do bloco **catch** se o código dentro de **try** lançar uma exceção.

```
throw new Exception();
```

# *Spl Exceptions*

- BadFunctionCallException
- BadMethodCallException
- DomainException
- InvalidArgumentException
- LengthException
- LogicException

# *Spl Exceptions*

- OutOfBoundsException
- OutOfRangeException
- OverflowException
- RangeException
- RuntimeException
- UnderflowException
- UnexpectedValueException

# Como testar exceções?

```
/**  
 * @expectedException BadFunctionCallException  
 */  
public function testBadFunctionCallException()  
{  
throw new BadFunctionCallException(__METHOD__);  
}
```

# Como testar erros do PHP?

```
/**
 * @expectedException PHPUnit_Framework_Error
 */
public function testBadFunctionCallException()
{
    //PHP errors, warnings, and notices
}
```



**KEEP  
CALM  
CAUSE YOU'RE  
THE ONLY  
EXCEPTION**

# *Spl File Handling*

- `SplFileInfo`
- `SplFileObject`
- `SplTempFileObject`





KEEP  
CALM  
AND  
FILE  
OFTEN

# *Spl Miscellaneous*

- `ArrayObject`
- `SplObserver`
- `SplSubject`



# *Spl Miscellaneous*

## **ArrayObject**

Em PHP, o tipo **array** é um tipo primitivo, como **boolean**, **float**, **integer** e **string**.

Embora argumentos de funções e métodos possam ter indução do tipo **array**, ele não é uma **classe**, por isso não tem **atributos** e **métodos**.

# *Spl Miscellaneous*

## **ArrayObject**

Implementa o padrão de projeto **Iterator**.

**Iterator** é um padrão utilizado para “fornecer um meio de acessar, **sequencialmente**, os elementos de um objeto agregado sem expor a sua representação subjacente” (Gamma et alli, 2000, p. 244)

# *Spl Miscellaneous*

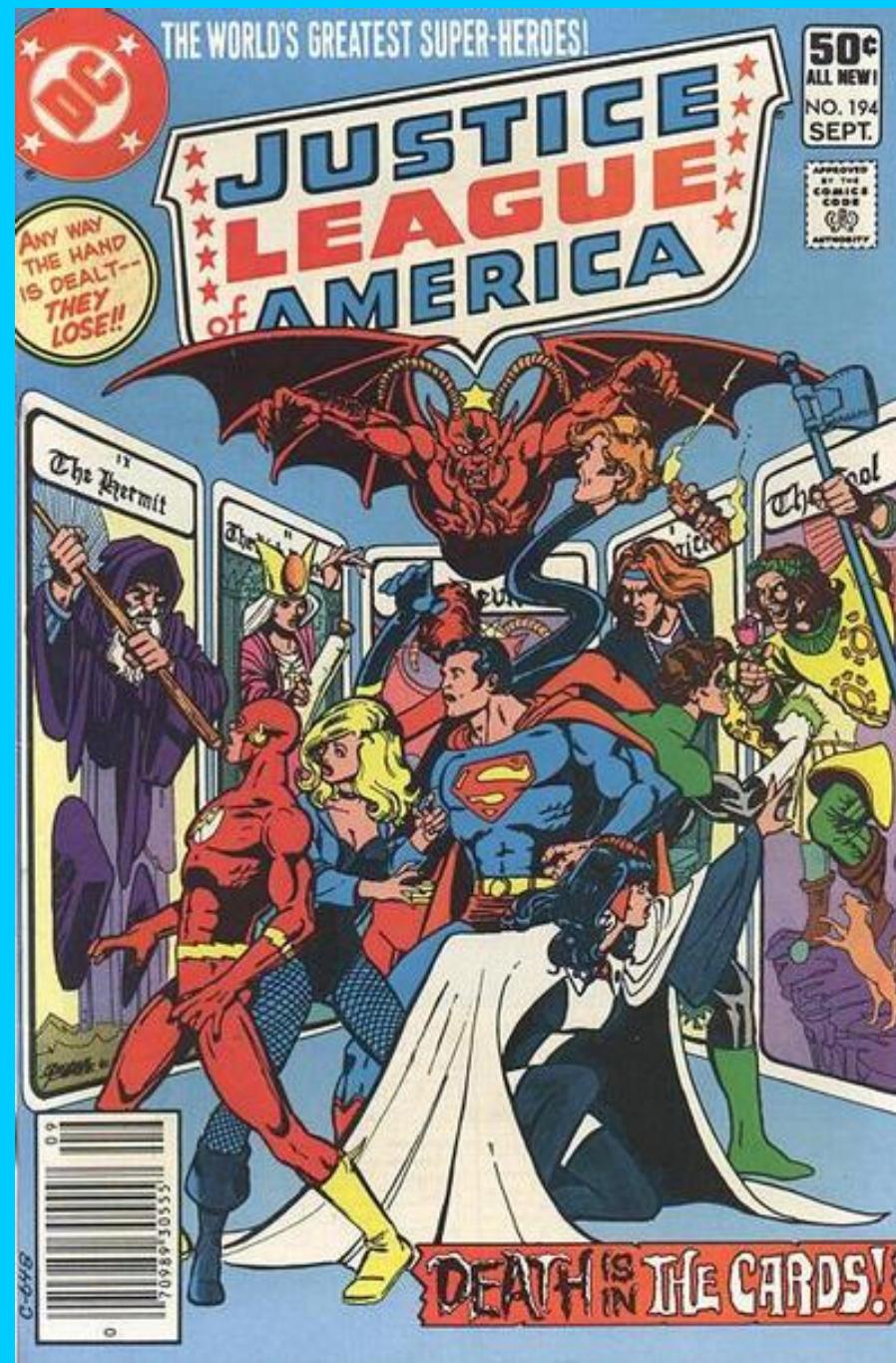
## **SplObserver e SplSubject**

Essas duas interfaces definem a estrutura básica para uma implementação do padrão de projeto **Observer**.

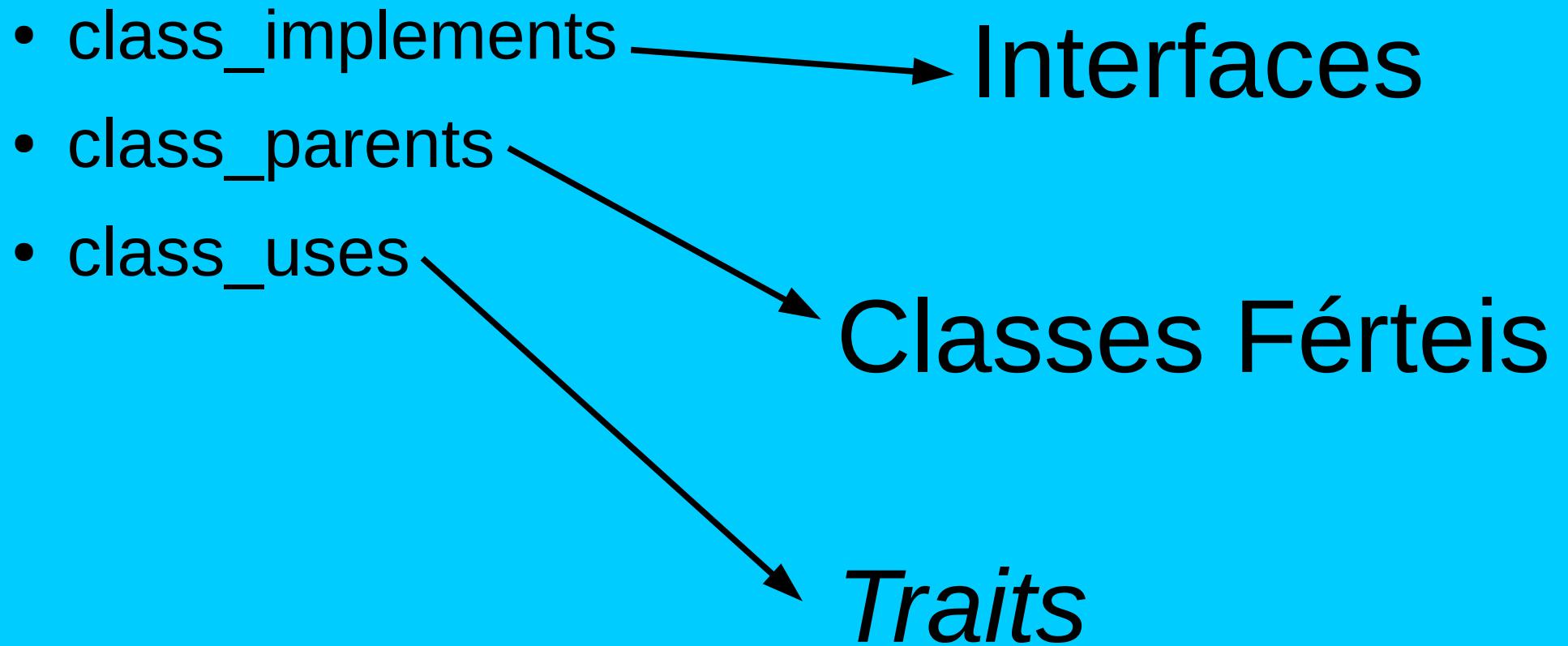
O padrão **Observer** visa “definir uma **dependência um-para-muitos** entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente” (Gamma et alli, 2000, p. 274).



KEEP  
CALM  
AND  
USE  
Design Patterns



# *Spl Functions*



# *Spl Functions*

Funções que recebem uma implementação de **Iterator** como argumento:

- iterator\_apply
- iterator\_count
- iterator\_to\_array

# *Spl Functions*

- `spl_autoload_call`
- `spl_autoload_extensions`
- `spl_autoload_functions`
- `spl_autoload_register`
- `spl_autoload_unregister`
- `spl_autoload`
- `spl_classes`
- `spl_object_hash`

**METADADOS**

# XUnit

“O **xUnit**, ferramenta de testes unitários para software, pretendia ser um pacote de programas/ampliações e modelos de teste para ajudar quem desenvolve projectos de software a criar os seus próprios testes de erros. Por norma o xUnit costuma ser software opensource , o que também ajudou a sua grande difusão” (Brandão, H. A. et alli, 2015).



“**Kent Beck**, foi o criador do conceito de xUnit, tendo a sua primeira implementação sido feita no início dos anos noventa, usando a linguagem de programação SmallTalk, ficou conhecido como Sunit”. (idem acima)

# Padrões de Testes

## Padrões Básicos xUnit

### Método de teste

Cada teste é codificado como um único método de teste em uma classe.

### Teste em quatro fases

Cada teste tem quatro partes distintas executadas em sequência: inicialização, exercício, verificação, destruição.

# Padrões de Testes

## Padrões Básicos xUnit

### Método de asserção

Um método avalia se uma saída esperada foi obtida.

### Mensagem de asserção

Cada método de asserção emite uma mensagem em caso de falha.

# Padrões de Testes

## Padrões Básicos xUnit

### Classe caso de teste

Métodos de teste relacionados são agrupados em uma única classe caso de teste.

### Executor de teste

Uma aplicação instancia um objeto suíte de teste e executa todos os objetos caso de teste que ele contém.

# Testes cheirando mal

- Cheiros do código;
- Cheiros do comportamento;
- Cheiros do projeto.



# Cheiros do código

- É difícil entender o teste em um relance;
- O teste contém código que pode ou não ser executado;
- O código é difícil de testar;
- O mesmo código de teste é repetido muitas vezes;
- O código em produção contém lógica que deve ser exercitada somente durante os testes.

# Cheiros do comportamento

- É difícil dizer qual das assertivas dentro do mesmo método de teste causou uma falha de teste;
- Um ou mais testes comportam-se errATICAMENTE: algumas vezes eles passam e algumas vezes eles falham;
- Um teste falha em compilar ou rodar quando o sistema sob teste é alterado de modos que não afetam a parte que está sendo testada;

# Cheiros do comportamento

- Depuração manual é requerida para determinar a causa da maioria das falhas de teste;
- Um teste requer que uma pessoa execute uma ação manual cada vez que ele é executado;
- Os testes levam muito tempo para serem executados.

# Cheiros do projeto

- Bugs são encontrados regularmente nos testes automatizados;
- Desenvolvedores não estão escrevendo testes automatizados;
- Muito esforço é despendido mantendo testes existentes;
- Muitos bugs são encontrados durante testes formais ou em produção.

# E agora, o momento que todos esperavam...



# Exercício de Avaliação

Crie um conjunto de classes PHP que simule o comportamento de um elevador e que valide esse comportamento com testes unitários.

O caso de uso a ser implementado é a notificação do **elevador** ao **andar** sobre qual é sua posição. A cada mudança de andar, provocada pelo pressionamento do botão correspondente ao andar, todos os andares devem ser informados da posição do elevador.

# Exercício de Avaliação

Nessa implementação, limite o uso do elevador à uma só **pessoa** de cada vez. Uma pessoa chama o elevador, ele vai até o andar, a pessoa entra, indica o andar para onde vai, o elevador a leva e a pessoa sai.

A classe de teste deve informar, para o estudo de caso implementado, onde o elevador está e onde a pessoa está. Cada mudança de andar deve atualizar essas informações.

# Exercício de Avaliação

Exemplo de simulação:

- O elevador está no 5º andar e uma pessoa que está no 3º o chama.
- O elevador desce, a pessoa entra e aperta o 8º andar.
- O elevador vai até o 8º andar e a pessoa desce.

# Exercício de Avaliação

Exemplo de saída:

Pessoa está no 3º andar. Elevador está no 5º andar.

Pessoa chama elevador. Elevador está no 5º andar.

Pessoa está no 3º andar. Elevador está no 4º andar.

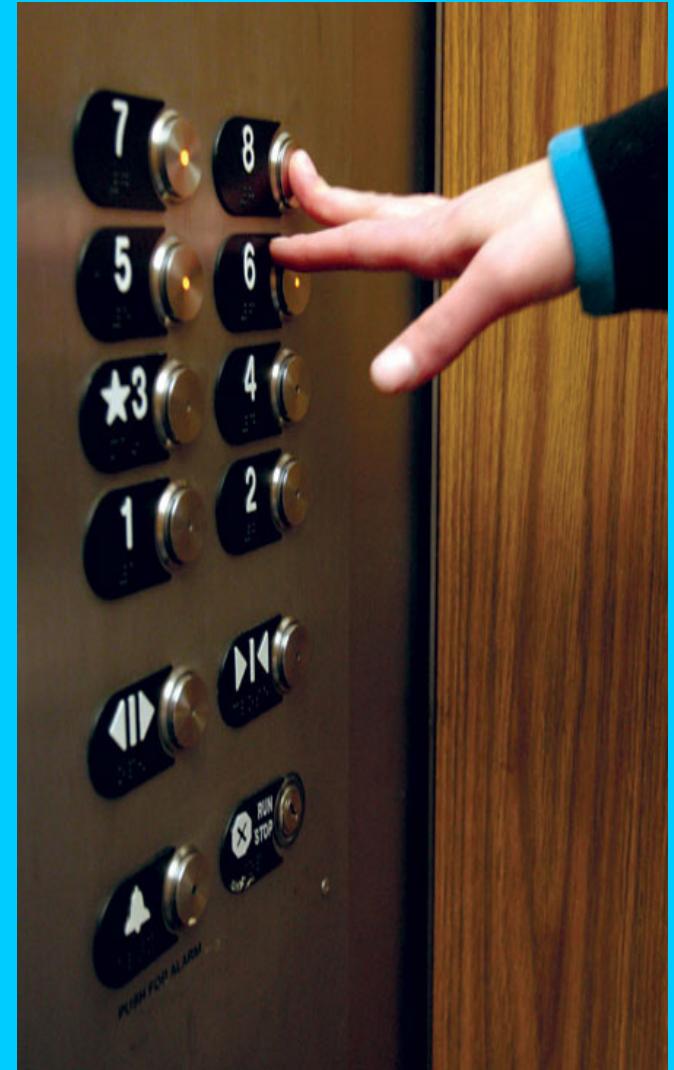
Pessoa está no 3º andar. Elevador está no 3º andar.

Pessoa entra no elevador. Elevador está no 3º andar.

Pessoa aperta 8º andar. Elevador está no 3º andar.

Pessoa está no elevador. Elevador está no 4º andar.

Pessoa aperta 8º andar. Elevador está no 3º andar.



# Exercício de Avaliação

Exemplo de saída (continuação):

Pessoa está no elevador. Elevador está no 4º andar.

Pessoa está no elevador. Elevador está no 5º andar.

Pessoa está no elevador. Elevador está no 6º andar.

Pessoa está no elevador. Elevador está no 7º andar.

Pessoa está no elevador. Elevador está no 8º andar.

Pessoa sai do elevador. Elevador está no 8º andar.

Pessoa está no 8º andar. Elevador está no 8º andar.



# Boa sorte!



Boa Sorte  
Charlie

# Bibliografia

- **Bergmann, S.** *PHPUnit Manual*. <https://phpunit.de/manual/current/en/phpunit-book.html>. Acesso em 26/09/2014.
- **Brandão, H. A.; Campos, J. T. M.; Freitas, T. M.; Guerreiro, J. M.; Oliveira, V. M.; Pinto, J. M. M.** *xUnit – Testes Unitários Automatizados*. 2005. Disponível em: <[http://paginas.fe.up.pt/~aaguiar/es/artigos%20finais/es\\_final\\_6.pdf](http://paginas.fe.up.pt/~aaguiar/es/artigos%20finais/es_final_6.pdf)>. Acesso em 21/09/2015.
- **Friedman, D. P. Wand, M. e Haynes, C. T.** *Fundamentos de Linguagens de Programação*. 2.ed. São Paulo. Berkeley Brasil, 2001.
- **Gamma, E. Helm, R. Johnson, R. e Vlissides, J.** *Padrões de projeto: soluções reutilizáveis de software orientado a objetos*. Porto Alegre. Bookman, 2000.
- **Meszaros, G.** *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.
- **Schärli, N. et alli.** *Traits: Composable Units of Behaviour*. <http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf>. Acesso em 26/09/2014.
- **Sebesta, R. W.** *Conceitos de linguagens de programação*. 4.ed. Porto Alegre. Bookman, 2000.
- **Tenenbaum, A. M. Langsam, Y. Augenstein, M. J.** *Estruturas de dados usando C*. São Paulo. Pearson Education do Brasil, 1995.
- **Weldon, G.** *PHP 5.4: Begin your love affair with Traits*. <http://www.slideshare.net/predominant/php-54-begin-your-love-affair-with-traits-10118525>. Acesso em 26/09/2014.