ÍNDICE

**1 Padrões de formatação de código** 2

1.1 **Geral** 2

1.1.1 Padrão 2

1.1.2 Indentação 2

1.1.3 Namespace e bloco use 2

1.1.4 Delimitadores 2

1.1.5 Visibilidade 3

1.1.6 Controles de estrutura 3

1.1.7 Fim de linha 3

1.1.8 Fim de Arquivo 3

1.1.9 Palavras reservadas 3

1.2 **Classes, Propriedades e Métodos** 3

1.3 **Formatação de linhas** 4

1.4 **Nomenclatura** 4

1.4.1 Nomes em inglês 4

1.4.2 Nomes informativos/significativos 4

1.4.3 Evite desinformação 5

1.4.4 Nomes pronunciáveis 5

1.4.5 Interfaces e implementações 6

1.4.6 Nomes de classes 6

1.4.7 Nomes de métodos 6

1.4.8 Use nomes do domínio dasolução 7

1.4.9 Use nomes do domínio doproblema 7

1.5 **Configurações** 7

1.5.1 Instalando o CodeSniffer 7

**1.5.2 Configurando o CodeSniffer na IDE** 7

1.5.2.1 PhpStorm 7

1. Padrões de formatação de código

Serão seguidos os padrões de codificação definidos pelo grupo fig-php (http://www.php-fig.org/) PSR-1[4] e PSR-2[5].

Abaixo é exibido um exemplo de código ideal:

1. <?php
2. namespace Vendor\Package;
4. **use** FooInterface;
5. **use** BarClass **as** Bar;
6. **use** OtherVendor\OtherPackage\BazClass;
8. **class** Foo **extends** Bar **implements** FooInterface
9. {
10. **public** **function** sampleFunction($a, $b = null)
11. {
12. **if** ( $a === $b ) {
13. bar();
14. }
15. **elseif** ($a > $b) {
16. $foo->bar($arg1);
17. }
18. **else** {
19. BazClass::bar($arg2, $arg3);
20. }
21. }
23. final **public** **static** **function** bar()
24. {
25. // method body
26. }
27. }
    1. Geral
       1. **Padrão**

Código deve seguir o padrão [PSR-1](https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-1-basic-coding-standard.md).

* + 1. **Indentação**

Deverá ser de 4 espaços. Não deverão ser usadas tabulações (Configurar IDE para substituir tabulações por espaços).

* + 1. **Namespace e bloco use**

Quando presente deverá ser declarado logo após a abertura do PHP “<?php”.

Deve haver uma linha em branco após a declaração do **namespace**, assim como após a declaração do último **use**, cada **use** deverá ser declarado em uma linha

* + 1. **Delimitadores**

Abertura de chaves para classes deverá ser na linha abaixo da declaração, o mesmo vale para métodos.

Abertura de chaves para condições e repetições, deverão estar na mesma linha da declaração

* + 1. **Visibilidade**

Sempre declarar a visibilidade de métodos e variáveis ( public, private, protected ) ; **abstract** e **final** deverão ser declaradas antes da visibilidade; **static** deverá ser declarada depois da visibilidade;

* + 1. **Controles de estrutura**

Deverão ter um espaço antes e um depois da abertura/fechamento de parênteses ( “if ($arg < 10) { ...” ), sem utilizar espaçamentos após a abertura de parênteses e antes do fechamento.

( “ if ~~( $arg < 10 )~~ ” ).

* + 1. **Fim de linha**

Todos os arquivos deverão utilizar Unix LF como padrão de fim de linha.

* + 1. **Fim de Arquivo**

Todos os arquivos deverão finalizar com uma linha em branco.

A tag “?>” deverá ser omitida em arquivos que contenham apenas código PHP.

* + 1. **Palavras reservadas**

Palavras chave e constantes do PHP ( true, false e null ) deverão estar em minúsculo.

* 1. **Classes, Propriedades e Métodos**

Extends e implements deverão ser declarados na mesma linha do nome da classe.

1. **class** ClassName **extends** ParentClass **implements** \ArrayAccess, \Countable

Entretanto, quando houver mais de uma implementação e a declaração ultrapassar o limite de 80 caracteres, inserir as implementações em linhas separadas, como no exemplo:

1. <?php
2. namespace Vendor\Package;
4. **use** FooClass;
5. **use** BarClass **as** Bar;
6. **use** OtherVendor\OtherPackage\BazClass;
8. **class** ClassName **extends** ParentClass **implements**
9. \ArrayAccess,
10. \Countable,
11. \Serializable
12. {
13. // constants, properties, methods
14. }
    1. **Formatação de linhas**

Ideal de 80 caracteres por linha

Será permitido até 120 caracteres por linha. (Configurar style Checker para lançar aviso na IDE).

Não poderá haver linhas em branco com espaços.

Não fazer mais do que um statement por linha ( “~~$this->foo(); $this->bar() ;~~“ )

* 1. **Nomenclatura**
     1. **Nomes em inglês**

Nomes de classes, métodos, variáveis e constantes devem estar em língua inglesa.

* + 1. **Nomes informativos/significativos:**

Nomes de variáveis, classes, métodos ou funções devem revelar porque ela existe, o que ela faz e como ela deve ser usada.

Exemplo 1:

1. **public** $d; // elapsed time in days
2. //versus
3. **public** $elapsedTimeInDays;

O nome 'd' não significa nada, tanto que um comentário é necessário. Já o segundo nome nos mostra o que será guardado na variável e em qual unidade.

Nomes que revelam a intenção tornam o código muito mais legível, prazeroso de ler e mais fácil de mudar.

Exemplo 2:

1. **public** List<int[]> getThem() {
2. List<int[]> list1 = **new** ArrayList<int[]>();
3. **for** (int[] x : theList)
4. **if** (x[0] == 4)
5. list1.add(x);
6. **return** list1;
7. }

O que o código acima faz?

O código acima é bastante simples, sem expressões complexas, apenas 3 variáveis e 2 constantes, não existem classes abstratas nem mesmo polimorfismo. Contudo é muito difícil dizer, com certeza, o que este código faz, pois o problema não é a simplicidade do código e sim o quanto as coisas estão *implícitas* no código.

Ao ler o código surgem algumas perguntas:

* 1. Que tipo de coisas estariam representadas na variável 'theList'?
  2. Qual o significado do item 0 da variável 'theList'?
  3. O que o valor '4' significa?
  4. Como a lista retornada deve ser usada?

As respostas destas perguntas não estão no código, mas deveriam estar:

1. **public** List<Cell> getFlaggedCells() {
2. List<Cell> flaggedCells = **new** ArrayList<Cell>();
3. **for** (Cell cell : gameBoard)
4. **if** (cell.isFlagged())
5. flaggedCells.add(cell);
6. **return** flaggedCells;
7. }
   * 1. **Evite desinformação:**

Tão ruim quanto um nome que não significa nada é um nome que parece significar algo, mas, na verdade, significa outra coisa.

Exemplo:

|  |
| --- |
| Não se refira a um grupo de usuários como *userList* se a estrutura de dados usada para agrupar estes usuários não for uma Lista, a palavra Lista possui um significado para programadores e usar ela dar nome a algo que não é uma lista pode levar a falsas conclusões. Então *userGroup* ou apenas *users* seria uma escolha melhor. |

* + 1. **Nomes pronunciáveis:**

Não esqueça que você está escrevendo código para humanos lerem. Então, use nomes que possam ser facilmente pronunciados por quem está lendo.

Exemplo:

1. //O que é melhor?
3. **class** DtaRcrd102 {
4. **private** Date genymdhms;
5. **private** Date modymdhms;
6. **private** final String pszqint = "102";
7. /\* ... \*/
8. }
9. //ou
10. **class** Customer {
11. **private** Date generationTimestamp;
12. **private** Date modificationTimestamp;;
13. **private** final String recordId = "102";
14. /\* ... \*/
15. }
    * 1. **Interfaces e implementações:**

Nomes de interfaces não devem ser precedidos com um 'I' ou qualquer outro tipo de diferenciação.

Exemplo:

|  |
| --- |
| Uma interface que representa um repositório de usuários deve apenas se chamar *UserRepository* e não *IUserRepository* ou *UserRepositoryInterface.* Nomes mais específicos devem ser colocados nas implementações da interface, neste exemplo poderíamos ter uma implementação chamada *RelationalUserRepository* para uma implementação de repositório que usa um SGBD relacional para tanto. Quando tivermos apenas uma implementação 'default' para a interface este implementação pode possuir o sufixo “Impl”. Neste exemplo teríamos *UserR*e*positoryImpl.* |

* + 1. **Nomes de classes:**

Nomes de classes devem ser substantivos ou frases com substantivos como *Customer*, *WikiPage*,

*Account*, e *AddressParser.* Evite nomes como *Manager*, *Processor*, *Data* ou *Info*, estes são nomes genéricos e quando usados tentem a dar um nome pouco significativo a classe. O nome de uma classe não deve ser um verbo.

* + 1. **Nomes de métodos:**

Os nomes de métodos devem ser verbos ou frases que possuam verbos. Devem representar uma ação, como: postPayment, deletePage, ou save. Métodos de acesso, métodos de set e predicados devem ser nomesdos com o nome da variável mais um prefixo “get”, “set” ou “is” de acordo com o padrão javabean.

Exemplo:

1. **lic** **class** MyClass {
3. **var** name;
4. **var** enable;
6. **public** getName() {
7. **return** $this->name;
8. }
10. **public** setName($name) {
11. $this->name = $name;
12. }
14. **public** isEnable() {
15. **return** $this->enable;
16. }
18. }
    * 1. **Use nomes do domínio da solução:**

Quem está lendo o seu código é um programador, então se você está usando um pattern ou um algoritmo conhecido para resolver um problema, use o nome deste pattern ou algoritmo no nome da classe ou do método. Assim ao ler o nome da classse ou do método um programador já sabe o que esperar, já possui uma ideia de o que aquele código deve fazer.

* + 1. **Use nomes do domínio do problema:**

Quando estivermos trabalhando com algo que não possa ser traduzido para a “linguagem de programador” devemos usar nomes que remetam ao domínio do problema, assim quando um outro programador se deparar com o nosso código ele poderá consultar uma pessoa que entenda do domínio do problema, a partir do código.

* 1. **Configurações**
     1. Instalando o CodeSniffer

Para instalar o CodeSniffer no Windows execute o seguinte comando[1]:

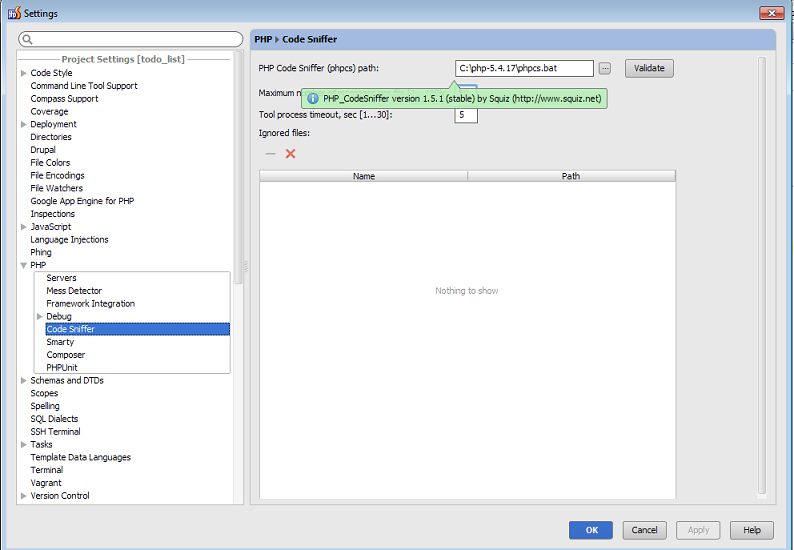
pear install PHP\_CodeSniffer

Por padrão, o PHP\_CodeSniffer vai usar o padrão de codificação PEAR. É possível alterar o padrão configurando a opção default\_standard com o seguinte comando[2]:

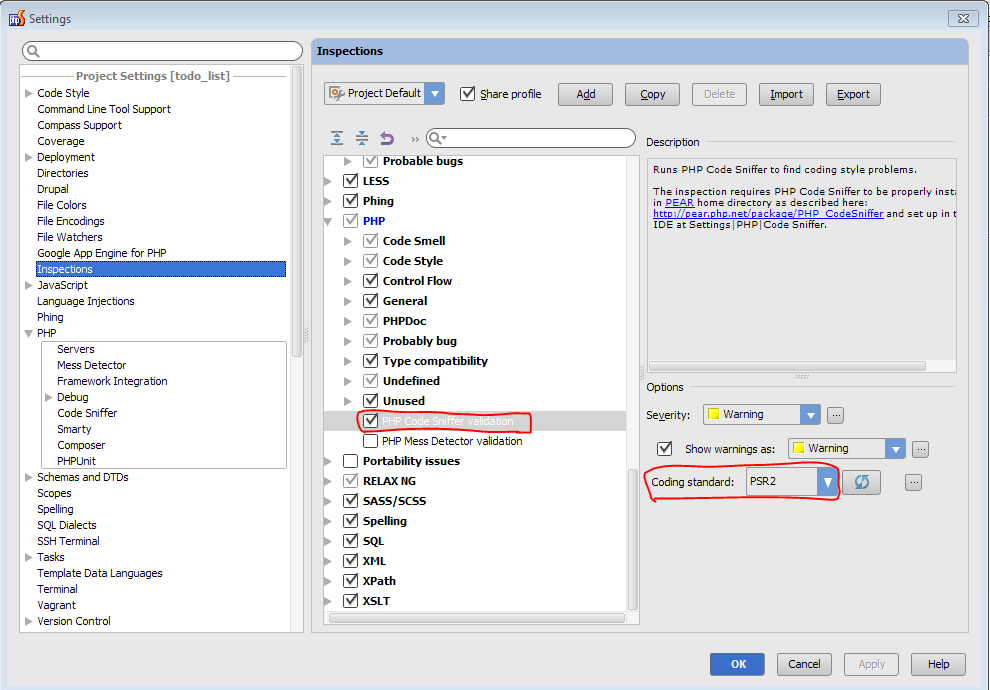
phpcs --config-set default\_standard PSR2

* + 1. Configurando o CodeSniffer na IDE
       1. PhpStorm

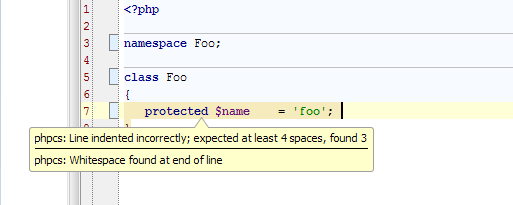
Vá até File > Settings > PHP > CodeSniffer > PHP Code Sniffer (phpcs) path, selecione o arquivo e após clique em Validate.



Após vá até File > Settings > Inspections > PHP > PHP Code Sniffer validation, marque esta opção, selecione como vai exibir os erros de formatação e sete a opção Coding standard para PSR2 e aplique as alterações.



Resultado:



**REFERÊNCIAS**

1. **Package Information: HP\_CodeSniffer**. Disponível em <http://pear.php.net/package/PHP\_CodeSniffer> acesso 05 de fevereiro de 2014
2. **PHP\_CodeSniffer Configuration Options**. Disponível em <http://pear.php.net/manual/pt\_BR/package.php.php-codesniffer.config-options.php> acesso 05 de fevereiro de 2014
3. Martin, C. R**. Clean Code – A Hand Book of Agile Software Craftsmanship**. Capítulo 2.
4. **fig-standards.** Disponível em <https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-1-basic-coding-standard.md> acesso em 05 de fevereiro de 2014.
5. **Coding Style Guide**. Disponível em <http://www.php-fig.org/psr/psr-2/> acesso em 04 de fevereiro de 2014