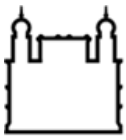

Ministério da Saúde

FIOCRUZ

Fundação Oswaldo Cruz

COGETIC – Coordenação de Gestão de Tecnologia da Informação e Comunicação

Padrões de codificação de Software

Histórico da Versão

Data	Versão	Descrição	Autor(es)
31/01/2020	1.0	Primeira versão dos padrões	Jean-Gabriel Nguema N.

Índice

1 Objetivo

Este documento tem por objetivo propor e descrever regras e convenções de codificação a serem seguidas nos projetos de desenvolvimento de sistemas da Fundação Oswaldo Cruz (FIOCRUZ), visando garantir a qualidade dos softwares produzidos. Estas regras e convenções são denominadas simplesmente de Padrão de codificação de software.

Seguir as regras e convenções do assim chamado padrão possibilitará que os códigos sejam fáceis de ler e entender pelos atuais e futuros integrantes da equipe, tendo uma representação uniformizada. Isso resultará em ganhos de produtividade tanto na implementação quanto na manutenção do software, contribuindo seu aumento da qualidade.

2 Referências

- https://pmd.github.io/latest/
- https://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html
- https://www.oracle.com/technetwork/java/index-135089.html
- https://www.devmedia.com.br/padroes-de-codificacao/16529
- https://www.trt9.jus.br/pds/pdstrt9/guidances/concepts/coding_standard_880DD5D.html

3 REGRAS DE CODIFIFICAÇÃO

O padrão é organizado através de uma conjuntos de regras organizadas em grupos.

3.1 Regras de tamanho de código

Conjunto de regras que possibilitam identificar problemas relacionados ao tamanho de código.

3.1.1 Complexiade ciclomática (*CyclomaticComplexity*)

Métrica da Engenharia de Software que calcula o número de pontos de decisão em um método. Um ponto de decisão corrrsponde a uma das seguintes instruções: if, while, for, cada case de um switch. O próprio método contabiliza também um ponto de decisão. Com pontos de decisão calculados, temos os seguintes cenários:

- Valores entre 1 e 4 indicam uma **complexidade baixa**
- 5 a 7: compexidade moderada
- 8 a 10: **complexidade alta**
- Acima de 11: complexidade muito alta.

Propriedade	Valor	Descrição
reportLevel	10	Permitido até 10 pontos de decisão
showClassesComplexity	true	Indica se a media de violações na classe deve ser adicionado ao relatório.
showMethodsComplexity	true	Indica se a media de violações nos metodos deve ser adicionado ao relatório.

3.1.2 Tamanho excessivo do método (*ExcessiveMethodLength*)

Violações desta regra indicam um alto acoplamento do método, ou seja, o mesmo executa mais tarefas que não deveria. Para reduzir o tamanho deve-se usar métodos auxiliadores e remover trechos de código repetidos ou não utilizados.

Propriedade	Valor	Descrição
minimum	100	Valor limite permitido, caracterizando violação quando ultrapassado.
Ignore-whitespace	false	Conta as linhas com os espaços em branco.

3.1.3 Tamanho excessivo da classe (*ExcessiveClassLength*)

Uma classe com tamanho grande é uma indicação ela tentam executar muitas funcionalidades. Desta forma deve ser particionada em classes menores.

Propriedade	Valor	Descrição
minimum	1500	Valor limite permitido, caracterizando violação quando ultrapassado.
Ignore-whitespace	false	Conta as linhas com os espaços em branco.

3.1.4 Número excessivo de parâmetros (*ExcessiveParameterList*)

Lista excessivo de parâmetros é uma indicação que um novo objeto (classe) deve ser criada para armazenar e agrupar os parâmetros.

Propriedade	Valor	Descrição
minimum	10	Valor limite permitido, caracterizando violação quando ultrapassado.

3.1.5 Número excessivo de membros públicos (*ExcessivePublicCount*)

Uma quantidade alta de membos (métodos e atributos) declarados na classe pode indicar que a esta última deve ser divida, já que seus testes teriam uma necessidade de um grande esforço.

Propriedade	Valor	Descrição
minimum	45	Valor limite permitido, caracterizando violação quando ultrapassado.

3.1.6 Número excessivo de campos (*TooManyFields*)

Classes com muitos campos podem ser redesenhada para conter menos campos, por exemplo através de objetos embutidos (*nested object*) agrupando campos. Por exemplo, classes com os campos cidade, estado e cep podem ter um único campo endereço.

Propriedade	Valor	Descrição
maxfields	30	Número de campos permitidos.

3.1.7 Número excessivo de métodos (*TooManyMethods*)

Classes com muitos metodos é suspeita para *refactoring* para reduzir sua complexidade e obter objetos com granulariade fina. Por default são ignorados métodos iniciando com get ou set.

Propriedade	**Valor**	**Descrição**
maxmethods	25	Número máximo de métodos permitidos.
Ignorepattern	(^(set|get))i	Ignora métodos que iniciam com get/set

3.1.8 Número excessivo de métodos públicos (*TooManyPublicMethods*)

Idem à regra anterior, mas com métodos públicos.

Propriedade	Valor	Descrição
maxmethods	10	Número máximo de métodos permitidos.
Ignorepattern	(^(set|get))i	Ignora métodos que iniciam com get/set

3.1.9 Complexidade excessiva da classe (*ExcessiveClassComplexity*)

A métrica WMC (*Weighted Method Count*) de uma classe é um ótimo indicador do esforço requerido para sua manutenção. A WMC é soma das complexidades de todos os métodos declarados. Assim, um grande número de métodos também significa que esta classe possui um alto potencial de ser derivados em outras classes.

Propriedade	Valor	Descrição
maximum	50	Valor máximo tolerado por uma classe

3.1.10 Complexidade NPath (*NPathComplexity*)

A complexidade NPath de um método é a quantidade de caminhos de execução acíclica dentro deste método. Valores de até 200 indicam uma complexade adequada.

Propriedade	Valor	Descrição
minimum	200	Valor limite permitido

3.2 Regras sobre código não usado

Este grupo contém um conjunto de regras que encontram códigos não usados.

3.2.1 Campo privado não usada: *UnusedPrivateField*

Detecta quando propriedades/campos privados (nível da classe) são de declarados, mas não usados.

```
class Something
{
    private static $FOO = 2; // Unused
    private $i = 5; // Unused
    private $j = 6;
    public function addOne()
    {
        return $this->j++;
    }
}
```

Figura 1: DeveloppmentCodeFragment

3.2.2 Variavel local não usadas: *UnusedLocalVariable*

Detecta quando variaveis locais (dentro de métodos, blocos, etc) são declaradas tem valores atribuídos, mas não são usadas ao longo do código.

```
class Foo {
    public function doSomething()
    {
        $i = 5; // Unused
    }
}
```

Figura 2: DeveloppmentCodeFragment

3.2.3 Metodos não usados: *UnusedPrivateMethod*

Detecta quando metodos privados são declarados, mas não usados no código. Exemplo:

```
class Something
{
    private function foo() {} // unused
}
```

Figura 3: DeveloppmentCodeFragment

3.2.4 Variaveis local não usadas: *UnusedFormalParameter*

Evitar passar parâmetros ao metodos ou construtores sem seu uso efeitivo dentro dos mesmos.. Exemplo:

```
class Foo
{
    private function bar($howdy)
    {
        // $howdy is not used
    }
}
```

Figura 4: DeveloppmentCodeFragment

3.3 Regras de nomenclaturas

Conjunto de regras que que verificam violações em nomes de variáveis, métodos, propriedades.

3.3.1 Variavel com nome curto (*ShortVariable*)

Detecta quando um campo (propriedade) de classe ou parâmetro de método tem nome curto.

Propriedade	Valor	Descrição
minimum	3	Comprimento mínimo para o nome.
exceptions		Lista de exceção separada por virgulas.

3.3.2 Variavel com nome longo (*LongVariable*)

Detecta quando um campo (propriedade) de classe ou parâmetro de método tem nome muito comprido.

Propriedade	Valor	Descrição
maximum	40	Comprimento maximo para o nome.

3.3.3 Método com nome curto (*ShortMethodName*)

Detecta quando métodos tem nomes muito curtos.

Propriedade	Valor	Descrição
minimum	3	Comprimento mínimo para o nome.
exceptions		Lista de exceção separada por virgulas.

3.3.4 Métodos booleanos (*BooleanGetMethodName*)

Verifica se os métodos que retornam um valor boleano possuem nomes corretos. Como convenção seus nomes devem ser na forma "***isX***" ou "***hasX***".

Propriedade	Valor	Descrição
checkParameterizedMethods	false	A regra se não se aplica aos métodos com parâmetros.

3.3.5 Nome do construtor da classe (*ConstructorWithNameAsEnclosingClass*)

Um construtor de uma classe não deve ter o mesmo nome do que a classe. Construtor devem devem ser criados usando o método ***__construct*** do PHP 5.

```
class MyClass {
     // this is bad because it is PHP 4 style
    public function MyClass() {}
    // this is good because it is a PHP 5 constructor
    public function __construct() {}
}
```

3.3.6 Nomes de constantes (*ShortMethodName*)

Nomes de constantes de Classe/Interface devem ser descritos em maiúsculos.

```
class Foo {
    const MY_NUM = 0; // ok
    const myTest = ""; // fail
}
```

3.4 Design Rules

Este grupo contem uma coleção de regras que encontram problemas relatados ao *design de software*.

3.4.1 Quantidade de subclasses: *NumberOfChildren*

Uma classe com uma quantidade excessive de filhos é um indicador para uma hierarquia não balanceada, que deve ser refatorada. Por default são consideradas no máximo 15 subclasses possivéis.

Propriedade	Valor	Descrição
minimum	15	Número máximo de subclasses permitidas para uma classe

3.4.2 Máxima altura de uma hierarquia: *DepthOfInheritance*

De modo similar à regra anterior, uma classe não deverá ter muitos parentes, pois isto traduz uma hierarquia não balanceada, que deve ser refatorada. Por default são consideradas no máximo 6 subclasses possiveis.

Propriedade	Valor	Descrição
minimum	6	Número máximo de parentes permitidos para uma classe

3.4.3 Acoplamento entre objetos: *CouplingBetweenObjects*

Uma classe com muitas dependências possui impacto negativo sobre critérios de qualidade como estabilidade, manutenablidade e compreensibilidade. Como boa prática, uma classe não deve ter mais de 13 dependências.

```
class Foo {
    /**
     * @var \foo\bar\X
     */
    private $x = null;
    /**
     * @var \foo\bar\Y
     */
    private $y = null;

    /**
     * @var \foo\bar\Z
     */
    private $z = null;

    public function setFoo(\Foo $foo) {}
    public function setBar(\Bar $bar) {}
    public function setBaz(\Baz $baz) {}

    /**
     * @return \SplObjectStorage
     * @throws \OutOfRangeException
     * @throws \InvalidArgumentException
     * @throws \ErrorException
     */
    public function process(\Iterator $it) {}

    // ...
}
```

Figura 5: EvalExpression correta

Propriedade	Valor	Descrição
maximum	13	Número máximo de dependências permitido

3.4.4 Fragmentos de códigos em desenvolvimento: *DevelopmentCodeFragment*

Algumas funções como var_dump(), print_r(), entre outras, são usadas em códigos no ambiente de desenolvimento, e devem ser evitados no ambiente de produção. Exemplo:

```
class SuspectCode {

    public function doSomething(array $items)
    {
        foreach ($items as $i => $item) {
            // …

            if ('qafoo' == $item) var_dump($i);

            // …
        }
    }
}
```

Figura 6: DeveloppmentCodeFragment

Propriedade	Valor	Descrição
unwanted-functions	var_dump, print_r, debug_zval_dump, debug_print_backtrace	Lista de images de funções suspeitas.
ignore-namespaces	false	Ignorar os namespaces na verificação de código em produção.

3.4.5 Expressão de saída: *ExitExpression*

Uma **expressão de saída** dentro do código regular não pode ser testada e, portanto, deve ser evitada (figura 3). Considere mover a expressão de saída para algum tipo de script de inicialização em que um código de erro ou exceção é retornado para a chamada.

```
class Foo {
    public function bar($param)  {
        if ($param === 42) {
            exit(23);
        }
    }
}
```

Figura 7: ExitExpression em um código

3.4.6 Expressão eval: *EvalExpression*

Uma **expressão eval** dentro não pode ser testada, representa um risco de segurança e não é uma boa prática. Deve portanto ser evitada. Considere substituir a expressão de evaliação por um código regular (figura 2).

```
class Foo {
    public function bar($param)  {
        if ($param === 42) {
            exit(23);
        }
    }
}
```

Figura 8: EvalExpression correta

3.4.7 Declaração Goto

Declarações goto devem ser evitadas, pois tornam o Código difícil a ler e impossível de entender o fluxo de controle de uma aplicação.

```
class Foo {
    public function bar($param)  {
        A:
        if ($param === 42) {
            goto X;
        }
        Y:
        if (time() % 42 === 23) {
            goto Z;
        }
        X:
        if (time() % 23 === 42) {
            goto Y;
        }
        Z:
        return 42;
    }
}
```

Figura 9: Exemplo de Goto que deve ser evitado

3.4.8 Bloco Catch vazio: *EmptyCatchBlock*

Um bloco catch vazio não é uma boa prática, pois inibe uma condição de erro continuando executando.

```
class Foo {

    public function bar()
    {
        try {
            // ...
        } catch (Exception $e) {} // empty catch block
    }
}
```

Figura 10: Exemplo de count em loop que deve ser evitado

3.4.9 Contagem dentro do loop: *CountInLoopExpression*

O uso de countagem dentro de loops através de count/sizeOf não é uma boa prática, pois causa muitos bugs, especialmente quando o loop manipula um array.

```
class Foo {

  public function bar()
  {
    $arr = array();

    for ($i = 0; count($arr); $i++) {
      // ...
    }
  }
}
```

Figura 11: Exemplo de count em loop que deve ser evitado

3.5 Códigos limpos

Este grupo contém um conjunto de regras são boas práticas que garantem que o código será limpo e mais fácil de de ser entendido por terceiros.

3.5.1 Variavel não decaradas: *UndefinedVariable*

Detecta quando uma variável usada não foi declarada antes. Exemplo:

```
function printX() {
    echo $x;
}
```

Figura 12: UndefinedVariable

3.5.2 Falta de import: *MissingImport*

Uma boa prática é importar todas as classes em um arquivo através da instruções "use", garantindo que seja claramente visíveis.

3.5.3 Evitar acesso estático: *StaticAcces*

Acessos **estáticos** (váriaves e métodos estáticos) devem ser evitados, pois causam dependências que não mudam entre classes conduzindo a códigos difíceis de testar. No lugar desses acessos, é recomendado usar injeções das dependências em constructores.

O único caso de acesso estático autorizado é quando for usado para métodos factory.

Propriedade	Valor	Descrição
exceptions		List de exceptions, quando existirem

3.5.4 Evitar declaração Else: *ElseExpression*

Uma expressão ***if*** com apenas um else não é necessária. As condições podem ser reescritas eliminando o else para que o código fique mais fácil de ser lido. Para tanto, é necessário usar a instrução "***return***", e o código deve ser quebrado em pequenas partes em métodos.

```
class Foo
{
    public function bar($flag)
    {
        if ($flag) {
            // one branch
        } else {
            // another branch
        }
    }
}
```

Figura 13: UndefinedVariable

3.5.5 Evitar atribuições dentro do if: *IfStatementAssignment*

Atribuições dentro de if devem ser evitadas, pois não são boa prática e podem conduzir a muitos bugs difícil de corrigir.

```
class Foo
{
    public function bar($flag)
    {
        if ($foo = 'bar') { // possible typo
            // ...
        }
        if ($baz = 0) { // always false
            // ...
        }
    }
}
```

Figura 14: Exemplo de atribuições que devem ser evitadas

3.5.6 Valores duplicados: *DuplicatedArrayKey*

Deve-se evitar definir outro valor para a mesma chave em array de literais (chave/valor), pois isso impossibilita o reuso do valor antigo.

```
function createArray() {
    return [
        'non-associative 0-element', // not applied
        0 => 'associative 0-element', // applied
        false => 'associative 0-element', // applied
        'foo' => 'bar', // not applied
        "foo" => 'baz', // applied
    ];
}
```

Figura 15: Exemplo de valores duplicados a ser evitado

3.5.7 Argumento com booleano: *BooleanArgumentFlag*

Um argumento com flag booleano é um indicador confiável da violação do principio da responsabilidade única em classe. Isso pode ser corrigido extraindo a logica do flag booleano em uma classe ou método.

```
class Foo {
    public function bar($flag = true) {
    }
}
```

Figura 16: Exemplo de flag booleano a ser evitado

4 Convenções

Convenções são boas praticadas esperadas nos códigos do software. As convesões apresentadas a seguir são baseadas em boas práticas do mercado, particularmente da Oracle[1].

4.1 Convenções de nomenclatura

As convenções de nomenclatura tornam os programas mais compreensíveis, facilitando a leitura. Possibilitam um melhor entendimento do código.

Elemento	Convenções
Classes	O nome de uma classe deve formado pela combinação de substantivos, sendo que a primeira letra de cada substantivo deve estar em maiúscula. Como boa prática, o nome da classe deve sugerir a responsabilidade por ela desempenhada. **Exemplos:** BuscadorWeb
Interfaces	A primeira letra de uma interface é a letra I (em maiúscula), para indicar que trata-se de uma interface. A partir dai, o restante do nome deve seguir as mesmas orientações de nomes de classes. **Exemplos:** IBuscador
Métodos/Funções	Nomes de métodos/funções devem iniciar com verbos seguidos ou não de uma sequência de palavras. A primeira letra de um método deve ser minúscula, e a primeira letra de cada palavra subsequente deve ser em maiúscula. **Exemplos:** executar, realizarLogin, obterNome, cadastrarMatrizDeRisco, calcularSuper.
Variaveis	Os nomes das variáveis devem ser curtos, mas bastante sugestivos. Eles iniciam com letra, e palavras internas iniciam com palavra em maiúscula. Excepto no caso de processamento local em loops, variáveis com uma única letra devem ser evitadas: i, j, k, etc. **Exemplos:** $idadeMedia, $pesoMaximoPermitido
Constantes	Nomes de constantes devem ser em letra maiúscula. Exemplos: PI=3.14

4.2 Convenções de Comentários

4.2.1 Comentários no código

É uma boa prática descrever as principais ações realizadas ao longo do código, de forma sucinta e com informações relavantes. Comentários feitos no código devem estar em conformidade com a ferramenta PHPDoc [2](phpDocumentor 2).

4.2.2 Comentários em classe

Deve haver um comentário imediatamente antes da declaração de cada classe, descrevendo de forma sucinta e com informações relavantes o propósito pretendido por ela. Comentários de classes devem ser estar em conformidade com a ferramenta PHPDoc. Exemplo:

[1]https://www.oracle.com/technetwork/java/javase/documentation/codeconventions-136091.html#262

[2] https://www.phpdoc.org/about

```
/**
* Exemplo_De_Class é uma classe para desmostrar um comentário de classe usando PHPDoc
*
* Exemplo_De_Class é uma classe que não possui código real, mas existe apenas para
* ilustrar como comentar classes usando.
*
* Example de uso:
* if (Exemplo_De_Class::exemplo()) {
*    print "Sou um exemplo .";
* }
*
* @package Example
* @author FIOCRUZ
* @author COGETIC
* @version $Revision: 1.3 $
* @access public
* @see http://www.example.com/pear
*/
class Exemplo_De_Class
{
}
```

Figura 17: UndefinedVariable

4.2.3 Comentários em método/função

Deve haver um comentário imediatamente antes da declaração de cada método/função, descrevendo de forma sucinta e com informações relavantes sua funcionalidade. Deve-se descreve os parâmetros de entrada, bem como a saída retornada pelo método. Exemplos:

```
/**
 * Realiza a soma de dois números
 *
 * @param int   $parameter1 número, primeiro operando.
 * @param int   $parameter2 número, segundo operando.
 *
 * @return string  Descrição do retorno.
 */
function soma($parameter1, $parameter2)
{
   x= $parameter1 + $parameter2;
   return x;
}
```

Figura 18: UndefinedVariable

```
/**
 * Imprime um 'Hello World' em função da entrada
 * @param bool $quiet when true 'Hello world' is echo-ed.
 *
 * @return void
 */
function outputHello($quiet)
{
    if ($quiet} {
        return;
    }
    echo 'Hello world';
}
```

Figura 19: UndefinedVariable

4.3 Convenções de Indentação

4.3.1 Unidade indentação

Uma boa prática para esta unidade são quatro espaços em branco, e não uma tabulação.

4.3.2 Comprimento da linha

Deve-se se evitar linhas com mais de 80 caracteres, pois não são devidamente tratadas por muitas ferramentas e terminais.

4.3.3 Quebra de linha

Quando for necessário quebrar uma linha, esta quebra deve obedecer os seguintes princípios:

- Quebrar antes da virgula
- Quebrar antes de um operador
- Alinhar sempre a nova linha em função da anterior

5 Avaliação

Todos os projetos de software conduzidos na COGETIC ou em outras unidades da Fiocruz. Empresas terceirizadas contratadas para desenvolver e entregar um produto de software deverão entregar código fontes alinhados com o padrão atual.

A avaliação do código será feita para cada novo incremento de software (por exemplo, em cada *spring* e *release* de software no caso de projeto com a MDS ágil). Manuntenções corretivas e evolutivas de um software existente também serão sujeitas a avaliação de códigos.