

Ministério da Saúde

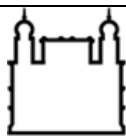
FIOCRUZ

Fundação Oswaldo Cruz

COGETIC

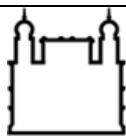
Coordenação-Geral de Gestão de Tecnologia de Informação

Padrões de codificação de Software



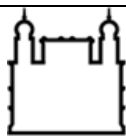
Histórico da Versão

Data	Versão	Descrição	Autor(es)
31/01/2020	1.0	Primeira versão dos padrões	Jean-Gabriel Nguema N.
10/10/2024	2.0	Atualizações de regras	Jean-Gabriel Nguema N.

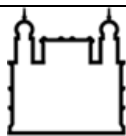


Índice

1	Objetivo	5
2	Escopo.....	5
3	REGRAS DE CODIFICAÇÃO	5
3.1	Regras de tamanho de código	5
3.1.1	Complexidade ciclomática (<i>CyclomaticComplexity</i>)	5
3.1.2	Tamanho excessivo do método (<i>ExcessiveMethodLength</i>).....	6
3.1.3	Tamanho excessivo da classe (<i>ExcessiveClassLength</i>).....	6
3.1.4	Número excessivo de parâmetros (<i>ExcessiveParameterList</i>)	6
3.1.5	Número excessivo de membros públicos (<i>ExcessivePublicCount</i>)	6
3.1.6	Número excessivo de campos (<i>TooManyFields</i>)	6
3.1.7	Número excessivo de métodos (<i>TooManyMethods</i>)	7
3.1.8	Número excessivo de métodos públicos (<i>TooManyPublicMethods</i>)	7
3.1.9	Complexidade excessiva da classe (<i>ExcessiveClassComplexity</i>)	7
3.1.10	Complexidade NPath (<i>NPathComplexity</i>).....	7
3.2	Regras sobre código não usado	7
3.2.1	Campo privado não usada: <i>UnusedPrivateField</i>	7
3.2.2	Variável local não usadas: <i>UnusedLocalVariable</i>	8
3.2.3	Métodos não usados: <i>UnusedPrivateMethod</i>	8
3.2.4	Variáveis local não usadas: <i>UnusedFormalParameter</i>	8
3.3	Regras de nomenclaturas	8
3.3.1	Variável com nome curto (<i>ShortVariable</i>)	8
3.3.2	Variável com nome longo (<i>LongVariable</i>)	8
3.3.3	Método com nome curto (<i>ShortMethodName</i>)	9
3.3.4	Métodos booleanos (<i>BooleanGetMethodName</i>)	9
3.3.5	Nome do construtor da classe (<i>ConstructorWithNameAsEnclosingClass</i>)	9
3.3.6	Nomes de constantes (<i>ShortMethodName</i>).....	9
3.3.7	Nomes de classes (<i>LongClassName</i>)	9
3.3.8	Nomes de classe (<i>ShortClassName</i>)	9
3.4	Regras de <i>Design</i>	10
3.4.1	Quantidade de subclasses: <i>NumberOfChildren</i>	10
3.4.2	Máxima altura de uma hierarquia: <i>DepthOfInheritance</i>	10
3.4.3	Acoplamento entre objetos: <i>CouplingBetweenObjects</i>	10
3.4.4	Fragmentos de códigos em desenvolvimento: <i>DevelopmentCodeFragment</i>	11
3.4.5	Expressão de saída: <i>ExitExpression</i>	11
3.4.6	Expressão eval: <i>EvalExpression</i>	11



3.4.7	Declaração Goto	12
3.4.8	Bloco Catch vazio: <i>EmptyCatchBlock</i>	12
3.4.9	Contagem dentro do loop: <i>CountInLoopExpression</i>	13
3.5	Regras de tipo Controversial	13
3.5.1	Variáveis superglobais (Superglobals)	13
3.5.2	Nome de classe no formato <i>CamelCaseClassName</i>	13
3.5.3	Nome de método no formato <i>CamelCaseMethodName</i>	13
3.6	Códigos limpos	14
3.6.1	Variável não declaradas: <i>UndefinedVariable</i>	14
3.6.2	Falta de <i>import</i> : <i>MissingImport</i>	14
3.6.3	Evitar acesso estático: <i>StaticAcces</i>	14
3.6.4	Evitar declaração Else: <i>ElseExpression</i>	14
3.6.5	Evitar atribuições dentro do if: <i>IfStatementAssignment</i>	15
3.6.6	Valores duplicados: <i>DuplicatedArrayKey</i>	15
3.6.7	Argumento com booleano: <i>BooleanArgumentFlag</i>	15
4	Convenções	16
4.1	Convenções de nomenclatura	16
4.2	Convenções de Comentários	16
4.2.1	Comentários no código	16
4.2.2	Comentários em classe	17
4.2.3	Comentários em método/função	17
4.3	Convenções de Indentação	18
4.3.1	Unidade indentação	18
4.3.2	Comprimento da linha	18
4.3.3	Quebra de linha	18
5	Avaliação	18
6	Referências	18



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 Escopo

Tendo em vista que a tecnologia mais utilizada na COGETIC é a linguagem PHP, o presente padrão se aplica inicialmente a esta tecnologia.

3 REGRAS DE CODIFICAÇÃO

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.

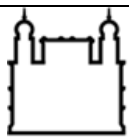
3.1.1 Complexidade ciclomática (*CyclomaticComplexity*)

Métrica da Engenharia de Software que calcula o número de pontos de decisão em um método. Um ponto de decisão corresponde 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: complexidade moderada
- 8 a 10: **complexidade alta**
- Acima de 10: complexidade muito alta.

Propriedade	Valor	Descrição
<i>reportLevel</i>	10	Permitido até 10 pontos de decisão
<i>showClassesComplexity</i>	true	Indica se a média de violações na classe deve ser adicionado ao relatório.
<i>showMethodsComplexity</i>	true	Indica se a média de violações nos métodos deve ser adicionado ao relatório.

Tabela 1 - Parametrização de complexidade ciclomática



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 auxiliares e remover trechos de código repetidos ou não utilizados.

Propriedade	Valor	Descrição
<i>minimum</i>	120	Valor limite permitido, caracterizando violação quando ultrapassado.
<i>ignore-whitespace</i>	false	Conta as linhas com os espaços em branco.

Tabela 2 - Parametrização do tamanho excessivo dos métodos

3.1.3 Tamanho excessivo da classe (*ExcessiveClassLength*)

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

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

Tabela 3 - Parametrização do tamanho excessivo da classe

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
<i>minimum</i>	12	Valor limite permitido, caracterizando violação quando ultrapassado.

Tabela 4 - Parametrização do número de parâmetros

3.1.5 Número excessivo de membros públicos (*ExcessivePublicCount*)

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

Propriedade	Valor	Descrição
<i>minimum</i>	45	Valor limite permitido, caracterizando violação quando ultrapassado.

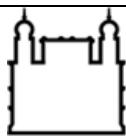
Tabela 5 – Parametrização do número excessivo de membros

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
<i>maxfields</i>	20	Número de campos permitidos.

Tabela 6 – Parametrização do número máximo permitido



3.1.7 Número excessivo de métodos (*TooManyMethods*)

Classes com muitos métodos é suspeita para *refactoring* para reduzir sua complexidade e obter objetos com granularidade 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

Tabela 7 – Parametrização do número máximo de métodos na classe

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

Tabela 8 – Parametrização do número máximo de métodos permitidos

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

Tabela 9 – Parametrização da complexidade máxima da 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 complexidade adequada.

Propriedade	Valor	Descrição
minimum	200	Valor limite permitido

Tabela 10 – Parametrização da Complexidade NPath

3.2 Regras sobre código não usado

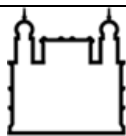
Este grupo contém um conjunto de regras que encontram códigos não usados.

3.2.1 Campo privado não usada: *UnusedPrivateField*

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

```
class Something
{
    private static $F00 = 2; // Unused
    private $i = 5; // Unused
    private $j = 6;
    public function addOne()
    {
        return $this->j++;
    }
}
```

Figura 1: DevelopmentCodeFragment



3.2.2 Variável local não usadas: *UnusedLocalVariable*

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

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

Figura 2: DevelopmentCodeFragment

3.2.3 Métodos não usados: *UnusedPrivateMethod*

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

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

Figura 3: DevelopmentCodeFragment

3.2.4 Variáveis local não usadas: *UnusedFormalParameter*

Evitar passar parâmetros aos métodos ou construtores sem seu uso efetivo dentro deles, por exemplo:

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

Figura 4: DevelopmentCodeFragment

3.3 Regras de nomenclaturas

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

3.3.1 Variável 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 vírgulas.

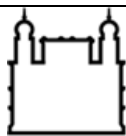
Tabela 11 – Parametrização do comprimento mínimo de uma variável

3.3.2 Variável 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 máximo para o nome.

Tabela 12 – Parametrização do comprimento máximo da variável



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.

Tabela 13 – Parametrização do comprimento mínimo de método

3.3.4 Métodos booleanos (*BooleanGetMethodName*)

Verifica se os métodos que retornam um valor booleano 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.

Tabela 14 – Parametrização do nome de métodos booleanos

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 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  
}
```

3.3.7 Nomes de classes (*LongClassName*)

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

Propriedade	Valor	Descrição
minimum	40	Comprimento mínimo para o nome.
exceptions		Lista de exceção separada por virgulas.

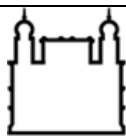
Tabela 15 – Parametrização de comprimento de constantes

3.3.8 Nomes de classe (*ShortClassName*)

Detecta quando uma classe/interface o tem nome curto.

Propriedade	Valor	Descrição
minimum	3	Comprimento mínimo para o nome.
exceptions		Lista de exceção separada por virgulas: Log, URL, FTP

Tabela 16 – Parametrização de comprimento de nomes de classes



3.4 Regras de Design

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

3.4.1 Quantidade de subclasses: *NumberOfChildren*

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

Propriedade	Valor	Descrição
<i>minimum</i>	15	Número máximo de subclasses permitidas para uma classe

Tabela 17 – Parametrização de quantidade máximo de filhos

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 possíveis.

Propriedade	Valor	Descrição
<i>minimum</i>	6	Número máximo de parentes permitidos para uma classe

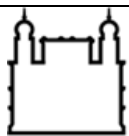
3.4.3 Acoplamento entre objetos: *CouplingBetweenObjects*

Uma classe com muitas dependências possui impacto negativo sobre critérios de qualidade como estabilidade, manutenibilidade 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
<i>maximum</i>	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 desenvolvimento, 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: *DevelopmentCodeFragment*

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

3.4.5 Expressão de saída: *ExitExpression*

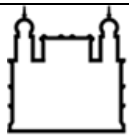
Uma **expressão de saída** dentro do código regular não pode ser testada e, portanto, deve ser evitada (figura 3). Considere mover a expressão de saída para algum tipo de script de inicialização em que um código de erro ou exceção é retornado para a chamada.

```
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 avaliação 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;  
    }  
}
```

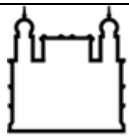
Figura 9: Exemplo de Goto que deve ser evitado

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 contagem 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 Regras de tipo Controversial

Este conjunto de regras contém uma coleção de regras controversas.

3.5.1 Variáveis superglobais (Superglobals)

Acessar diretamente uma variável superglobal é considerado uma má prática. Essas variáveis devem ser encapsuladas em objetos fornecidos por um framework, por exemplo.

```
class Foo {  
    public function bar() {  
        $name = $_POST['foo'];  
    }  
}
```

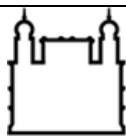
Figura 12. Ilustração de Violação de Regras

3.5.2 Nome de classe no formato *CamelCaseClassName*

É considerada uma prática recomendada usar a notação `CamelCase` para nomear classes.

3.5.3 Nome de método no formato *CamelCaseMethodName*

É considerada uma prática recomendada usar a notação `camelCase` para nomear métodos.



3.6 Códigos limpos

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

3.6.1 Variável não declaradas: *UndefinedVariable*

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

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

Figura 13: *UndefinedVariable*

3.6.2 Falta de import: *MissingImport*

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

3.6.3 Evitar acesso estático: *StaticAcces*

Acessos **estáticos** (variáveis 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 construtores.

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

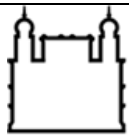
Propriedade	Valor	Descrição
exceptions		Lista de exceções, quando existirem

3.6.4 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 14: *UndefinedVariable*



3.6.5 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 15: Exemplo de atribuições que devem ser evitadas

3.6.6 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
    ];
}
```

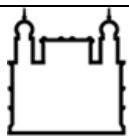
Figura 16: Exemplo de valores duplicados a ser evitado

3.6.7 Argumento com booleano: *BooleanArgumentFlag*

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

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

Figura 17: Exemplo de flag booleano a ser evitado



4 Convenções

Convenções são boas praticadas esperadas nos códigos do software. As conversões apresentadas a seguir são baseadas em boas práticas do mercado, particularmente da Oracle¹.

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 se trata de uma interface. A partir daí, 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.
Variáveis	Os nomes das variáveis devem ser curtos, mas bastante sugestivos. Eles iniciam com letra, e palavras internas iniciam com palavra em maiúscula. Exceto 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

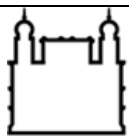
4.2 Convenções de Comentários

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 relevantes. Comentários feitos no código devem estar em conformidade com a ferramenta PHPDoc ²(phpDocumentor 2).

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

² <https://www.phpdoc.org/about>



4.2.2 Comentários em classe

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

```
/**
 * 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
{
}
```

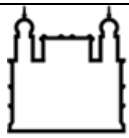
Figura 18: UndefinedVariable

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 relevantes sua funcionalidade. Deve-se descrever 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 19: 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 20: 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 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 aos seguintes princípios:

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

5 Avaliação

Todos os projetos de desenvolvimento ou sustentação de software na Fiocruz devem estar em conformidade com os padrões especificados neste documento, sendo estes projetos conduzidos pela Fiocruz ou pelas empresas terceirizadas.

6 Referências

- <https://phpmd.org/>
- <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