Qualiti Software Processes

Padrão de Codificação Java

Versão <1.0>

Conteúdo

1 Introdução

O maior esforço no desenvolvimento de software é dispensado às atividades de manutenção, desta forma, quanto mais fácil for o entendimento do código do sistema, mais produtiva será a equipe de desenvolvimento. Freqüentemente as pessoas que escrevem o código não são as mesmas que o mantém e, quando são, geram uma dependência com o código desenvolvido que dificilmente é dissolvida. Um padrão de codificação visa minimizar esses problemas, pois estabelece regras definindo como o código deve ser escrito para favorecer a impessoalidade do artefato.

Este documento tem como objetivo definir um padrão de codificação que, quando usado, garante um melhor entendimento por qualquer pessoa que conheça e siga o mesmo.

Os componentes desenvolvidos utilizando a metodologia da Receita Federal seguem as regras deste padrão, destacadas com marcadores do tipo: ❶,❷,❸; desta forma, é possível fazer referência a uma determinada regra pelo seu número, por exemplo, a regra 2.1.1, localizada na seção 2.1 deste documento, refere-se ao nome e número de classes que são declaradas em um arquivo.

Ao final deste documento, é apresentado no Apêndice B um exemplo de código utilizando as regras definidas no padrão. Para codificar seu próprio componente, entretanto, o programador pode se utilizar do modelo genérico definido em tipoTemplate.java

2 Arquivos

2.1 Nomenclatura

Os arquivos têm o mesmo nome da classe pública que contêm. ❶ Não se define mais de uma classe por arquivo, exceto para as *Inner Classes*[1] e classes auxiliares, declaradas privadas.

2.2 Documentação

Há três formas de comentário em Java. O estilo Javadoc[2] inicia o comentário com "/**" e termina com "*/" e é usado para gerar documentação do código em páginas *HTML*. Outro estilo de documentação em Java foi herdado da linguagem C e, por isto, é freqüentemente chamado de "estilo C"; iniciado com "/*" e finalizado com "*/" é usado para comentar código que não está sendo usado mas que o programador prefere manter comentado caso mude de idéia. Por último, os comentários em linha que iniciam com "//" e tem o efeito de comentar o restante da linha após a sua presença. Este último é usado em métodos para explicar regras de negócio e a lógica de execução dos métodos.

---

[1] A especificação de *Inner Classes* pode ser encontrada no site da JavaSoft http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/index.html.

[2] Não faz parte do escopo deste documento apresentar o padrão Javadoc, para isto, é recomendada a leitura do artigo http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/javadoc.html disponível no site da Sun, entretanto, no Apêndice A deste documento há uma breve introdução ao padrão Javadoc, mostrando a utilidade das suas principais construções.

❶ Cada arquivo começa com um bloco de comentários contendo as seguintes informações: título do projeto, nome da classe ou interface e informação de *copyright* relevante ao conteúdo do arquivo[3].

❷ Se um arquivo possuir mais de um tipo (classe ou interface), é inserida uma lista com uma pequena descrição de cada tipo que compõe o arquivo. É importante destacar uma explicação que justifique a declaração de mais de um tipo por arquivo, pois Java só permite um tipo público por arquivo, dificultando a busca dos tipos não públicos.

Abaixo, segue o modelo de documentação de arquivo.

```
/* Projeto: <Título do projeto>
 *
 * Tipo1: <Nome da classe ou interface pública>
 * Tipo2: <Nome da classe ou interface>, descrição: <descrição da classe ou
interface>
 * ...
 * TipoN: <Nome da classe ou interface>, descrição: <descrição da classe ou
interface>
 *
 * Informação de Copyright  opcional
 */
```

2.3 Declaração

❶ A classe ou interface pública é a primeira a ser declarada em um arquivo.

❷ A seguir, uma tabela apresenta a ordem das declarações em um arquivo.

	Declaração de um arquivo	**Nota**
1	Comentários do arquivo	Conforme apresentado na Seção 2.2
2	Declaração do pacote	Conforme apresentado na Seção 3.3
3	Declaração de classe e interface	Conforme apresentado nas Seções 4.3 e 5.3. A classe pública do arquivo é declarada no início do mesmo. Este item de declaração é repetido para cada interface ou classe que compõe o arquivo.

No Apêndice B é apresentado um código ilustrando as declarações acima.

3 Pacotes

3.1 Nomenclatura

❶ Os identificadores – as palavras que compõem o nome de um pacote – são separados por pontos e não há restrição quanto ao seu número; são iniciados com letras minúsculas e não devem conter caracteres especiais, como *underscores*, ou caracteres específicos de uma língua.

[3]Caso o código fonte possa ser utilizado para fins indevidos.

Os pacotes são nomeados de acordo com os conceitos que agrupam. ❷ Para um sistema de gestão empresarial, por exemplo, poderiam ser definidos os pacotes `gestao.financeiro.contasReceber` e `gestao.contabilidade` para agrupar, respectivamente, as classes do subsistema de contas a receber do sistema financeiro e as classes da contabilidade.

❸ Caso seja necessário distribuir o pacote geograficamente, o início do nome do pacote é formado pela identificação do grupo de desenvolvimento dentro da organização, por exemplo, o pacote `gestao.contabilidade` é nomeado como `srf.<grupo>.gestao.contabilidade`, onde "srf" é uma abreviação para o nome da organização e <grupo> refere-se ao nome do grupo dentro da organização que desenvolveu o pacote.

3.2 Documentação

3.3 Declaração

❶ Após o comentário de início do arquivo, discutido na seção 2.2, é declarada a sentença `package`, após isto, a lista de classes importadas é relacionada através da cláusula `import`.

❷ As classes dos pacotes do núcleo (*core*) de Java, pacotes tipo "`java.<nome do pacote>`", são listadas primeiro, após estas, são listadas as classes das extensões de Java, pacotes tipo "`javax.<nome do pacote>`", e, por último, as classes específicas do sistema e outras APIs utilizadas.

4 Classes

4.1 Nomenclatura

❶ Os nomes das classes não são abreviados, excetuando-se os casos em que as abreviações são mais conhecidas que o seu nome completo, pois a economia de palavras na elaboração de um nome não compensa a perda de expressividade associada. ❷ Não usa-se artigos, preposições para conectar substantivos e adjetivos, nem caracteres específicos de uma língua como é o caso do "ç" e os acentos da língua portuguesa. ❸ A primeira letra de cada palavra que compõe o nome de uma classe é maiúscula.

O nome de uma classe representa um objeto da mesma e não o seu conjunto de objetos. ❹ Assim, o nome da classe é sempre no singular, exceto nos casos em que o próprio objeto represente uma coleção de outros objetos; nestes casos, usa-se a palavra que represente a coleção no singular e o nome dos objetos no plural como, por exemplo, `RepositorioAlunos`.

❺ Toda classe que define uma exceção contém a palavra `Exception` no final de seu nome.

❻ O nome de uma classe não é mais abrangente nem mais específico que o conceito que representa, pois, no primeiro caso, o programador pode querer usar uma funcionalidade apesar de não ser oferecida pela classe; no outro caso, o programador poderia não encontrar a classe por não representar o conceito mais genérico que procura.

Seguem alguns exemplos de nomes de classes:

```
Aluno
AlunoGraduacao
CPF
ProvaInvalidaException // uma exceção
```

4.2 Documentação

Cada classe começa com um comentário "`/**...*/`" descrevendo o propósito da classe, instruções de uso e, opcionalmente, alguns exemplos para facilitar o uso da mesma. Em seguida, tem-se lembretes sobre possíveis melhoramentos e defeitos existentes na classe. ❶ No final do comentário, adiciona-se o nome dos autores e referências úteis para o entendimento da classe.

❷ Em seguida, tem-se a declaração do nome da classe e seus supertipos, se necessário, com algum comentário "`/*...*/`" que não deve aparecer no documento gerado pelo Javadoc.

Abaixo, segue o comentário de uma classe usando os marcadores comentados anteriormente, que devem aparecer na ordem ilustrada abaixo:

```
/**
* Esta classe representa uma janela na tela.
*
* Exemplo de uso:
* <pre>
* Janela win = new Janela(parent);
* win.show();
* </pre>
*
* Limitações: Uma janela não pode ser criada dentro de outra por...
*
* @author Marcos Silveira
* @see java.awt.Window
* @see java.awt.Component
*/
public class Janela extends Container{
/* Comentários sobre opções de implementação invisíveis ao Javadoc são feitos
aqui */
```

4.3 Declaração

❶ Após a declaração do nome da classe e seus supertipos, são declaradas as constantes, variáveis de classe[4], variáveis de instância, construtores, finalizador, métodos de classe e métodos de instância, nesta seqüência. ❷ Quanto aos modificadores de acesso, primeiro declara-se as variáveis públicas, depois as protegidas, as sem modificadores, e, por último, as privadas.

❸ Os métodos são agrupados por funcionalidade e não pela forma de acesso ou sua condição de estático ou de instância.

A tabela seguinte ilustra a ordem de declarações dos componentes de uma classe:

	Declaração da Classe	Nota
1	Comentários da classe (`/**...*/`)	Descrito na seção 4.2
2	Sentença `class`	
3	Comentários de implementação da classe	Caso necessário

4 Usa-se o termo variável e métodos de classe com o mesmo significado de variável e métodos estático, respectivamente.

4	Constantes	
5	Variáveis de classe	Na seguinte ordem: públicas, protegidas, sem modificadores (pacote), privadas.
6	Variáveis de instância	Na seguinte ordem: públicas, protegidas, sem modificadores (pacote), privadas.
7	Construtores	Na seguinte ordem: públicas, protegidos, sem modificadores (pacote), privados.
8	Finalizadores	
9	Métodos	Os métodos devem ser agrupados por **funcionalidade** e não por forma de acesso.

5 Interfaces

5.1 Nomenclatura

❶ O nome de uma interface é um adjetivo ou substantivo, e segue as mesmas regras para nomenclatura de classes definidas na Seção 4.1.

Abaixo, segue exemplos de possíveis nomes para interfaces:

```
Executavel
EntradaDados
```

5.2 Documentação

❶ A forma de documentação das interfaces é idêntica a das classes. Veja a Seção 4.2.

5.3 Declaração

❶ As declarações de interface seguem a ordem apresentada no modelo abaixo:

	Declaração da Interface	Nota
1	Comentários da interface "`/** ...*/`"	Descrito na Seção 5.2
2	Sentença `interface`	
4	Constantes	Na seguinte ordem: públicas, protegidas, sem modificadores (pacote), privadas
5	Métodos	Os métodos devem ser agrupados por **funcionalidade**

6 Variáveis de classe, de instância e constantes

6.1 Nomenclatura

❶ Os nomes das variáveis de classe e instância atendem aos requisitos abaixo:

- A primeira letra das palavras, exceto da primeira palavra, é maiúscula
- Os nomes não são abreviados, exceto nos casos que a sua abreviação seja mais sugestiva que o nome completo, ou no caso de variáveis que armazenam componentes visuais
- Não se mistura palavras de mais de uma língua
- Não se utiliza nenhum caracter especial nem específico de uma língua
- Variáveis que armazenam componentes visuais são nomeadas utilizando o padrão **TTTD$^+$**, onde **TTT** corresponde ao tipo do componente visual (conforme a tabela do Apêndice C), mnemônicos com apenas 3 (três) caracteres, e **D$^+$** refere-se à descrição do componente.

❷ Os nomes das constantes são compostos de palavras não abreviadas com todas as letras maiúsculas utilizando *underscores* como separadores.

Veja abaixo nomes de constantes e variáveis de classes e instância:

```
VALOR_MAXIMO // constante
pbOkInsereAutor // componente visual
lbLivros // componente visual
autor // variável de instância
livro // variável de instância
contador // variável de classe
```

6.2 Documentação

❶ A documentação de uma variável ou constante é feita em linha (veja seção 3.2) exceto nos casos em que não caiba em uma linha de código. ❷ Faz parte da documentação a descrição do invariante que se aplique à variável como, por exemplo, o intervalo de 1 a 31 para a variável `diaMes`.

❸ Há uma justificativa caso alguma variável ou constante não seja declarada privada.

6.3 Declaração

❶ Apenas uma variável ou constante é declarada por linha de código.

7 Métodos

7.1 Nomenclatura

❶ Os nomes dos métodos atendem aos requisitos abaixo:

- A primeira letra das palavras, exceto da primeira palavra, é maiúscula
- Os nomes não são abreviados
- Não se mistura palavras de mais de uma língua, exceto nos métodos de acesso a variáveis

- Não se utiliza nenhum caracter especial
- A primeira palavra deve ser um verbo no infinitivo representando a utilidade do método, com exceção dos métodos que retornam um `boolean`, que devem começar com um verbo no presente

Apesar da aplicação deste padrão resultar em nomes maiores, necessitando digitação extra, o efeito da sua conformidade é um código mais fácil de compreender, pois o propósito do método já é esclarecido no seu nome.

❷ Métodos de acesso a variáveis iniciam com *get* ou *set*[5] e finalizam com o nome da variável tendo a primeira letra de cada palavra maiúscula.

Veja abaixo alguns exemplos de nomes de métodos:

```
adicionarAluno(Aluno aluno)
removerAluno(Aluno aluno)
existeAluno(int codigoAluno)
getNumero() // método de acesso
```

7.2 Documentação

❶ Todo método contém um cabeçalho de documentação que fornece informações suficientes para seu entendimento e uso adequado. Inicialmente, documenta-se o que o método faz e porque faz, após isto, relaciona-se todos os parâmetros necessários para chamar o método, sua cláusula de retorno, e as possíveis exceções que pode levantar[6]. Caso a decisão de visibilidade do método possa ser questionada, documenta-se a razão pela qual foi tomada esta decisão. Se necessário, são declaradas ao final do comentário referências a outras classes e métodos, assim como, a data da criação do método.

Veja um exemplo de documentação de método:

```
/**
 *...Este método calculará o imposto devido bruto do cliente ...
 *
 *...Caso a decisão de visibilidade possa ser questionada a
 * justificativa será feita neste trecho
 *
 * @param taxa    Valor da taxa que será usada para calcular o imposto
 *
 * @return        Valor do imposto bruto que será cobrado do cliente
 *
 * @exception     Exceção1     Descrição da Exceção1
 * @exception     Exceção2     Descrição da Exceção2
 *
 * @see            Tributo
 * @since          Data da criação do método opcional
 */
protected float calculaImposto(float taxa) throws ExcecaoTaxaInvalida
```

[5] Padrão Java usado na sua tecnologia de componentes – JavaBeans.

[6] Inclusive as classes de `java.lang.RunTimeException`.

❷ Caso necessário, outros itens são acrescentados ao cabeçalho acima como, por exemplo, pré-condições e pós-condições, histórico de alterações do método, questões de concorrência, limitações e erros detectados no método.

❸ O comentário no estilo C (veja Seção 3.2), apesar de ser bastante prático, pois têm o efeito de comentar todas as linhas entre o início e o fim da declaração, é utilizado em blocos isolados de código, caso contrário, pode surgir problemas quando comentários no mesmo estilo precisam ser feitos em escopo mais geral.

Este problema é apresentado abaixo, ilustrando a situação em que um comentário no estilo C é aninhado e o efeito causado por isto.

```
/*
try{
    socket.open();
    ...
    while(true){
        ...
        socket.receive(packet);
----->  /*
        ...
        */
        jogo.moveBack( conjuntoP );// callback para o sistema
    }//while
}//try
catch(ClassNotFoundException classException) {
    ...
}
finally {
    socket.close();
}
}*/
```

❹ O comentário em linha é realizado ao lado de uma declaração ou comando caso o mesmo não exceda o número máximo de caracteres[7], caso contrário, é inserido acima do trecho de código sendo comentado.

Abaixo, segue um exemplo de uso deste comentário:

```
// Aplica 5% de desconto para todos os pedidos
// acima de R$1000,00 de acordo com a campanha
// "Vamos acabar com a fome" iniciada em fevereiro
// de 1997
if (totalPedido >= VALOR_DESCONTO) {
    totalPedido = totalPedido * (1 - PERCENTUAL_DESCONTO);
}
```

❺ Caso algum trecho de código possa ser melhorado, embora esteja funcionando corretamente, coloca-se um comentário no código informando a possível melhoria, utilizando-se o seguinte formato:

```
//CORRIGIR Comentário
```

[7] Na Seção 11 deste documento são discutidos aspectos de apresentação visual do código onde são abordados problemas como indentação, limite de caracteres por linha de código, espaços em branco, linhas em branco e outros.

❻ Caso exista algum trecho de código que não esteja correto e que deve ser corrigido posteriormente, comenta-se utilizando o seguinte formato:

```
//REVISAR Comentário
```

7.3 Declaração

A declaração de métodos que retornam Arrays é feita da seguinte forma:

```
double[] metodo()
```

e não da forma abaixo:

```
double metodo() []
```

❶ Todas as declarações de variáveis locais são realizadas no início do código do método.

8 Variáveis locais e parâmetros

8.1 Nomenclatura

❶ A nomenclatura de variáveis locais é a mesma usada para variáveis de instância e classes (veja Seção 6.1), entretanto, por força de conveniência, essa nomenclatura é relaxada para os casos abaixo:

- Contadores – Usam-se letras do alfabeto para nomear contadores. A letra **`i`** é a primeira a ser usada; sendo necessário o uso de outros contadores, declara-se a letra **`j`** e assim sucessivamente até a última letra do alfabeto.
- Tipos definidos por Bibliotecas escritas em outras línguas – Usa-se o nome do tipo em letras minúsculas e uma descrição para o conceito que o objeto representa.
- Exceções – Como o uso de exceções é muito comum em Java, usa-se a letra **`e`** para declarar exceções genéricas.

Abaixo, seguem exemplos de nomes de variáveis locais:

```
streamDadosRetornoBancario; // uma Stream
i; // variável de laço
```

8.2 Documentação

❶ A documentação das variáveis locais é feita da mesma forma que a de variáveis de instância e de classe (veja 6.2).

8.3 Declaração

❶ Para facilitar a documentação e organização do método, apenas uma declaração de variável local é feita por linha de código.

9 Comandos

9.1 *return*

❶ Uma sentença `return` com valor de retorno não utiliza parêntesis, a menos que a sentença fique mais clara.

9.2 *while, do-while e for*

❶ Abaixo, são apresentados os estilos de formatação válidos dos comandos:

```
while (condição) {
    comandos;
}

do {
    comandos;
} while (condição);

for (inicialização; condição; atualização) {
    comandos;
}
```

❷ As sentenças `while` e for vazias têm as seguintes formas:

```
while (condição) {}

for (inicialização; condição; atualização) {}
```

9.3 *if-then-else*

❶ O comando `if-then-else` é usado com as chaves – “`{}`” – para evitar ambigüidade no escopo do comando.

❷ Abaixo, são apresentados os estilos de formatação válidos do comando:

```
if (condição) {
    comandos;
}

if (condição) {
    comandos;
} else {
    comandos;
}

if (condição1) {
    comandos;
} else if (condição2) {
    comandos;
} else{
    comandos;
}
```

9.4 *switch*

❶ Abaixo são apresentados os estilos de formatação válidos do comando:

```
switch (variável) {
case ABC:
    comandos;
    break;
case DEF:
    comandos;
    break;
case XYZ:
    comandos;
    break;
default:
    comandos;
    break;
}
```

O último comando de toda sentença `case` é o `break`.

Toda sentença `switch` possui uma cláusula `default`.

9.5 *try-catch-finally*

Abaixo, são apresentados os estilos de formatação válidos do comando:

```
try {
    comandos;
} catch (ExceptionClass e) {
    comandos;
}

try {
    comandos;
} catch (ExceptionClass e) {
    comandos;
} finally {
    comandos;
}
```

9.6 Documentação de blocos

❶ As declarações de blocos são comentadas no estilo em linha após o caracter de fechamento de bloco "`}`".

Abaixo, são apresentados exemplos de uso do comentário de blocos:

```
if (condição) {
...
} // fim do if (condição)

switch (variável) {
...
} // fim do switch (variável)

for (exp1; exp2; exp3) {
...
```

```
} // fim do for(exp1; exp2; exp3)
```

10 Espaçamento e Indentação[8]

❶ Quatro (**4**) espaços em branco são usados como unidade de indentação.

10.1 Tamanho de Linha

❶ As linhas têm menos de 80 caracteres para facilitar impressão e visualização do código.

10.2 Quebra de Linha

❶ Quando uma expressão não cabe em uma linha, as seguintes regras são utilizadas:

- A linha é quebrada depois de uma vírgula.
- A linha é quebrada antes de um operador.
- Prefira quebrar uma linha em uma expressão de um nível mais alto.
- A nova linha é alinhada com o começo da expressão do mesmo nível da linha anterior.

Veja alguns exemplos de chamadas de métodos onde é necessário utilizar mais de uma linha:

```
this.someMethod(longExpression1, longExpression2, longExpression3,
                longExpression4, longExpression5);

var = this.someMethod1(longExpression1,
                       someMethod2(longExpression2,
                                   longExpression3));
```

Veja alguns exemplos de expressões aritméticas onde é necessário utilizar mais de uma linha:

```
longName1 = longName2 * (longName3 + longName4 - longName5)
            + 4 * longname6; // PREFIRA !
longName1 = longName2 * (longName3 + longName4
                         - longName5) + 4 * longname6; // EVITE !
```

❷ A quebra de linha da condição do comando `if` e de métodos utiliza uma indentação de **8** espaços, conforme ilustrado nos exemplos abaixo:

```
//Não use esta indentação
if ((condition1 && condition2)
    || (condition3 && condition4)
    || !(condition5 && condition6)) {
    doSomethingAboutIt();
}

//USE ESTA
if ((condition1 && condition2)
        || (condition3 && condition4)
        || !(condition5 && condition6)) {
    doSomethingAboutIt();
```

[8] É possível utilizar uma ferramenta de formatação de código para respeitar a maioria dos padrões definidos nesta seção (http://www.c-lab.de/~jindent/)

```
}

//OU ESTA
if ((condition1 && condition2) || (condition3 && condition4)
        || !(condition5 && condition6)) {
    doSomethingAboutIt();
}

//Não use esta indentação
private static synchronized int getQuantidadeEstoque(Produto produto,
                                                      Date ateData ) {

//USE ESTA
private static synchronized int getQuantidadeEstoque(Produto produto,
        Date ateData ) { ...
```

10.3 Linhas em Branco

❶ Tem-se duas (**2**) linhas em branco nas seguintes situações:

- Entre os `imports` e os comentários da classe.
- Entre a declaração da classe e o bloco das variáveis de instância e estáticas.

❷ Usa-se uma (**1**) linha em branco nas seguintes situações:

- Entre o bloco de comentários do cabeçalho do arquivo e a cláusula `package`.
- Entre a cláusula `package` e a `import`.
- Entre declarações de métodos.
- Entre as variáveis locais de um método e a primeira sentença.
- Antes de um comentário.
- Entre blocos lógicos dentro do corpo de um método.
- Entre a última declaração de método da classe e o final da classe ("}").

10.4 Espaços em Branco

❶ Uma palavra reservada seguida por parêntesis é separada destes por um espaço em branco:

```
while (true) {
    comandos;
}
```

❷ Um espaço em branco aparece depois de cada vírgula em uma lista de parâmetros ou na inicialização dos elementos de um array:

```
umObjeto = metodo(argumento1, argumento2);
```

❸ Todos os operadores binários, exceto o ".", são separados dos operandos por um espaço em branco, conforme os exemplo abaixo:

```
a += c + d;
```

```
a = (a + b) / (d * c);
```

As expressões de um comando for são separadas por espaços em branco. Exemplo:

```
for (exp1; exp2; exp3){
    comandos;
}
```

❹ Um *cast* é seguido por um espaço em branco:

```
umaMoto = (Motocicleta) objeto.getVeiculo()
```

Apêndice A- Principais construções do Javadoc

Esta seção descreve as principais construções do padrão Javadoc. Abaixo, segue uma tabela com as principais construções, as situações de uso e suas utilidades.

Cláusula	Situação	Utilidade
@author nome	classes e interfaces	Indica o autor de uma classe ou interface. Caso a classe ou interface possua mais de um autor, deve ser criada uma entrada diferente para cada autor.
@deprecated	classes e métodos	Indica que o método, interface ou classe foi depreciado e não deve ser mais usado. Deve ser uma prática utilizar esta cláusula antes de retirar métodos da interface pública ou protegida das classes, evitando que outras classes façam referência para estes métodos.
@exception [nome] [descrição]	métodos	Descreve as exceções que um determinado método pode levantar.
@return [descrição]	métodos	Descreve o retorno do método e, se necessário, como deve ser usado.
@since	classes e métodos	Indica a quanto tempo o método ou classe existe.
@see nome	classes e interfaces	Cria um *hyperlink* para a classe ou interface referenciada como parâmetro. Se a classe estiver em um outro pacote, é necessário qualificar completamente o seu nome.
@see nome#metodo	métodos e variáveis de instância e de classe	Cria um *hyperlink* com um método, variável de instância ou de classe. Não é necessário se preocupar com a criação de âncoras HTML nos métodos, pois isto já é feito pelo Javadoc.
@version texto	classes e interfaces	Indica a versão de uma classe ou interface.

Apêndice B- Exemplo de uma classe documentada em Java

Este exemplo tem como objetivo ilustrar algumas práticas recomendadas no padrão de codificação Java.

```
/*
 * Projeto: CESAR, Vestibular Interativo
 *
 * Tipo: Usuario
 *
 * Marca Registrada (c) 1996-2000 CESAR
 * Av. Professor Luis Freire, s/n, Cidade Universitária, Recife-PE
 * CEP 50740-540, BRAZIL
 * Todos os direitos reservados.
 *
 * Este software é confidencial e propriedade intelectual do
 * CESAR - Centro de Estudos e Sistemas Avançados do Recife.
 * Você não deve utilizar indevidamente este produto em desacordo com o
 * contrato estabelecida com a empresa.
 */

package covest.usuarios;

import covest.util.*;


/**
* Usuario é um tipo que representa o comportamento de um usuário
* do sistema. Um usuário pode ser um aluno, uma escola ou um professor,
* todos envolvidos com o processo do vestibular interativo.
*
* Exemplo de Uso:
* Usuario usuarioJoao = new Aluno("jbd", "ugh99", "Joao Barbosa Dueire",
*                                  "jbd@yol.com.br",
*                                  "Av. Prof. Luis Vanderlei, 449, Rio de Janeiro- RJ");
* usuarioJoao.equals(usuario);
*
* @author      Marcos Silveira
*
* @see         covest.usuarios.Aluno
* @see         covest.usuarios.Professor
* @see         covest.usuarios.Escola
*/
public abstract class Usuario {

    private String login;
    private String senha;
    private String nome;
    private String email;
    private Endereco endereco;
```

```
/**
 * Construtor da classe Usuario.
 *
 * @param login     login do usuário no sistema
 * @param senha     senha do usuário no sistema
 * @param lembrete  lembrete para senha no sistema
 * @param nome      nome do usuário
 * @param email     endereço de email do usuário
 * @param endereco  Endereço para contato
 *
 */
 public Usuario(String login, String senha, String lembrete,
                String nome, String email, Endereco endereco) {
    this.login = login;
    this.senha = senha;
    this.lembrete = lembrete;
    this.nome = nome;
    this.email = email;
    this.endereco = endereco;
 }

/**
 * Seta o login do usuário
 *
 * @param        Uma String contendo a senha do usuário
 *
 */
 public void setLogin(String login) {
     this.login = login;
 }

/**
 * Retorna o login do usuário
 *
 * @return        Uma String contendo a senha do usuário
 *
 */
 public String getLogin() {
     return login;
 }

/**
 * Retorna a senha do usuário
 *
 * @return        Uma String contendo a senha do usuário
 *
 */
 public String getSenha() {
    return senha;
 }

/**
 * Seta a senha do usuário
 *
 * @param        Uma String contendo a senha do usuário
 *
 */
 public void setSenha(String senha) {
     this.senha = senha;
 }
```

```
/**
 * Retorna o nome do usuário
 *
 * @return         Uma String contendo o nome do usuário
 *
 */
public String getNome() {
    return nome;
}

/**
 * Seta o nome do usuário
 *
 * @param          Uma String contendo o nome do usuário
 *
 */
public void setNome(String nome) {
    this.nome = nome;
}

/**
 * Retorna o email do usuário
 *
 * @return         Uma String contendo o endereço de email do usuário
 *
 */
public String getEmail() {
    return email;
}

/**
 * Seta o email do usuário
 *
 * @param          Uma String contendo o endereço de email do usuário
 *
 */
public void setEmail(String email) {
    this.email = email;
}

/**
 * Retorna o endereço de contato do usuário
 *
 * @return         Uma String contendo o endereço de contato do usuário
 *
 */
public Endereco getEndereco() {
    return endereco;
}

/**
 * Seta o endereço de contato do usuário
 *
 * @param          Uma String contendo o endereço de contato do usuário
 *
 */
public void setEndereco(Endereco endereco) {
    this.endereco = endereco;
}
```

```
    /**
     * Compara se dois usuários são o mesmo objeto
     *
     * @param         Uma String contendo o endereço de contato do usuário
     *
     * @return        true se os dois objetos são o mesmo ou se os dois forem
     *                nulos, caso contrário, retornará falso.
     */
    public boolean equals(Usuario usuario) {
        return Funcao.equals(this,usuario);
    }

}
```

Apêndice C- Tabela de nomes para componentes de interface gráfica

Os componente visuais oferecidos no pacote AWT (*Abstract Windowing Toolkit*), *Swing* e outros pacotes de terceiros são amplamente usados na elaboração de interfaces gráficas em Java. Visando padronizar a declaração destes componentes e manter a uniformidade do código, a tabela abaixo relaciona mnemônicos que representam tipos de componentes de interface gráfica freqüentemente usados na elaboração de formulários, caixas de diálogo e outros componentes visuais:

Tipo do Componente	Abreviação do Tipo
List	lb (lst)
Choice	cmb (cho)
Memo	mem
Edit	edt
Button	but (btn)
Tree	tre
Table	tbl
Menu	mnu
ComboBox	cmb
CheckBox	chb
Frame	frm
Label	lbl
Panel	pnl
PopupMenu	pop
ProgressBar	pgb
RadioButton	rdb
Timer	tmr

A medida que novos componentes surjam, a tabela é estendida para se adaptar aos novos tipos.