Convenção de Código

Grupo SMA(D&D)

2014

Este documento traz informações sobre as convenções de codificação utilizadas durante o desenvolvimento e evolução do código-fonte do projeto de SMA (Sistema Multi Agente) na FGA, Universidade de Brasília.

FGA, Universidade de Brasília

08 de Junho de 2014

SUMÁRIO

1. INTRODUÇÃO

1.1. MOTIVAÇÃO

Convenções de código são importantes para programadores por muitos motivos.

- 80% dos custos durante o tempo de vida de um *software* é gasto com manutenção.
- Dificilmente um *software* é mantido por uma única pessoa.
- Convenções de código permitem maior legibilidade do *software*, tornando o entendimento dos engenheiros mais rápido e melhor. (HOMMEL, KING, *et al.*)

Para que a convenção funcione, todas as pessoas envolvidas com o projeto devem ater-se às convenções propostas.

1.2. OBSERVAÇÕES

Grande parte das convenções propostas para o projeto são derivadas, parcialmente ou na íntegra, das convenções propostas pela *Sun Microsystems* e pelas convenções propostas nos livros da DEITEL.

2. ORGANIZAÇÃO DE ARQUIVOS

Um arquivo é constituído por seções, que devem ser separadas por uma linha em branco e um comentário opcional para cada seção.

Além disso, deve-se evitar que esses arquivos ultrapassem mais de 2500 linhas. Dessa forma evita-se arquivos muito complicados.

2.1. ARQUIVOS FONTE JAVA

Cada arquivo fonte Java deve conter apenas uma classe, ou interface, pública. Quando uma classe e/ou interface privada estiver associada a uma classe pública, pode-se colocá-la no mesmo arquivo que a classe pública. A classe, ou interface, pública deve ser a primeira coisa visível no arquivo.

Arquivos fonte Java devem seguir a seguinte ordem de organização:

- Comentários Iniciais.
- Instruções de pacotes e importação.
- Declaração de classe ou interface.

2.1.1. COMENTÁRIOS INICIAIS

Todo arquivo fonte deve iniciar com um comentário ao estilo C, que liste o nome da classe, ou interface, seu propósito e notas de *copyright*. Segue um exemplo:

```
1 /*
2  * File:    ExampleClass
3  * Purpose: This is a example for explain how to write a beginning comment.
4  *
5  * Copyright: any copyright notice, if there.
6  */
7
```

2.1.2. Instruções de Pacotes e Importação

A primeira linha do arquivo que não seja um comentário deve ser uma instrução de pacote. Feito isso, segue-se as instruções de importação. Exemplo:

```
5  package com.OMM.application.user.dao;
6
7  import java.util.ArrayList;
8  import java.util.List;
9
10 import android.content.ContentValues;
11 import android.content.Context;
12 import android.database.Cursor;
13 import android.database.sqlite.SQLiteDatabase;
14
15 import com.OMM.application.user.exceptions.NullParlamentarException;
16 import com.OMM.application.user.helper.LocalDatabase;
17 import com.OMM.application.user.model.Parliamentarian;
18
```

2.1.3. Declaração de Classes e Interfaces

A tabela que se segue descreve as partes de uma declaração de classe ou interface. Essas partes devem aparecer na ordem em que são apresentadas na tabela.

Ordem	Parte da Classe/Interface	Descrição
1	Comentário de Documentação da Classe/Interface	Ver "Comentários de Documentação" para mais detalhes.
2	Instrução *class* ou *interface*	
3	Comentário de Implementação da Classe/Interface	Este comentário contém qualquer informação adicional sobre a Classe/Interface que não for apropriado ao comentário de Documentação.
4	Variáveis *Static* da Classe	Primeiro as variáveis `public`, seguidas das variáveis `protected`. Depois as variáveis `private`.
5	Variáveis de Instância da Classe	Primeiro as variáveis `public`, seguidas das variáveis `protected`. Depois as variáveis `private`.
6	Construtores	
7	Comentários de Documentação do Método	Ver "Comentários de Documentação" para mais detalhes.
8	Métodos	Primeiro os métodos `public`, seguidas dos métodos `protected`. Depois os métodos `private`.

3. IDENTAÇÃO

3.1. COMPRIMENTO DA LINHA

As linhas não devem ultrapassar um comprimento de 80 caracteres, já que acima disso as linhas deixam de ser tratadas por muitos terminais e ferramentas.

3.2. LINHAS DE ACONDICIONAMENTO

Quando uma expressão não cabe em uma única linha deve-se quebrá-la, de acordo com os seguintes princípios gerais:

- Quebre após alguma vírgula.
- Quebre depois de um operador.
- Alinhar a nova linha com o início da expressão, ao mesmo nível da linha anterior.

Abaixo seguem alguns exemplos de quebra de expressões bastante comuns na chamada de métodos.

```
someMethod( firstParcel, secondParcel,
        fourthParcel, name );
```

```
sum = someMethod( firstParcel, secondParcel,
        fourthParcel, name );
```

Abaixo segue-se algum exemplo de como deve-se tratar uma quebra de linha de alguma expressão aritmética contendo parênteses.

```
sum = firstParcel + ( secondParcel + 4 ) +
        fourthParcel + 7;
```

Termina-se a expressão dentro do parênteses antes de fazer a quebra de linha.

Para a declaração de métodos segue as seguintes convenções, em caso de necessidade de quebra de linha.

```
public void someMethod( int firstParameter, int secondParameter,
        int thirdParameter, String fourthParameter ) {

}
```

Para instruções de `If` utilize a identação que se segue:

```
// Don't use this indentation...
if( ( firstParcel > secondParcel )
        || ( sum < fourthParcel ) ) {
        sum = 3 + 4;
}

// Use this indentation!!
if( ( firstParcel > secondParcel )
            || ( sum < fourthParcel ) ) {
        sum = 3 + 4;
}
```

A segunda é preferível, na medida que evita a confusão com as instruções que seguem dentro do corpo de instruções do `If`.

4. COMENTÁRIOS

4.1. COMENTÁRIOS DE IMPLEMENTAÇÃO

Os comentários de implementação devem ser usados para dar detalhes sobre uma dada linha ou bloco de instruções que permitam uma facilidade maior sobre o entendimento daquela parte específica da implementação. É importante observar comentários de implementação não devem explicar o que está ocorrendo naquele trecho de código, pois isso já deve ser feito pela prática do código auto explicativo. Comentários de implementação servem para dar detalhamentos cujo único meio de passá-los é por palavras, como a unidade de medida de um determinado dado, ou a razão para a implementação ser de um jeito e não de outro.

Todos os comentários devem iniciar com letra maiúscula e terminar com um ponto final.

4.1.1. COMENTÁRIOS EM BLOCO

Os comentários em bloco são usados quando não é possível dizer o necessário numa única linha. São utilizados para fornecer as descrições de estruturas de dados e algoritmos. Comentários em bloco podem ser utilizados dentro de métodos. Os comentários em bloco dentro de um método devem ser recuados para o mesmo nível que o código que eles descrevem.

Segue um exemplo de comentário em bloco.

```
/*
 * This is a block comment. Observe that in block comment there is
 * blank space between the text and the marks that define the margin
 * of comment and text.
 */
```

4.1.2. COMENTÁRIOS DE LINHA

Comentários curtos podem aparecer em uma única linha recuado para o nível do código que se segue. Se um comentário não pode ser escrito em uma única linha, deve seguir o formato bloco de comentário.

Um comentário de uma única linha deve ser precedida por uma linha em branco.

Um exemplo de comentário de uma linha só segue abaixo:

```
public void someMethod( int firstParameter, int secondParameter,
        int thirdParameter, String fourthParameter ) {

    /* Write instructions here. */
}
```

4.1.3. COMENTÁRIOS DE FIM DE LINHA

O `//` delimitador de comentário pode comentar uma linha completa ou apenas uma linha parcial. Não deve ser usado em várias linhas consecutivas para comentários em texto.

Deve ser usado para pequenos comentários sobre detalhes em uma linha, como comentar unidades de medida de uma determinada variável. Segue um exemplo.

```
private double area = 0; // Square meters.
```

4.2. COMENTÁRIOS DE DOCUMENTAÇÃO

Comentários de documentação descrevem classes Java, interfaces, construtores, métodos e campos. Cada comentário de documentação deve ser colocado dentro de delimitadores de comentário `/ ** ... * /`, com um comentário por classe, interface ou membro. Este comentário deve aparecer pouco antes da declaração. Um exemplo de como deve ser escrito:

```
/**
 * Exemplifies how a method with many parameters must be indented.
 * <p>
 * @param firstParameter first parcel in expression
 * @param secondParameter second parcel in expression
 * @param thirdParameter third parcel in expression
 * @param fourthParameter fourth parcel in expression
 * @return sum of the four parcel
 */
```

Notas:

- A primeira parte serve para descrever a classe, a interface ou o método.
- O marcador `<p>` serve para separar dois parágrafos (STACKOVERFLOW, 2011).

- No marcador `@param` deve-se por o nome do parâmetro e, em seguida a um espaço em branco, uma pequena descrição daquele parâmetro.
- No marcador `@return` deve-se por uma pequena descrição do que é o retorno daquele método.
- Uma dica para gerar o javadoc é, após escrever o marcador `/**`, pressione a tecla ENTER. O Eclipse irá gerar os parâmetros e o retorno automaticamente. Fica faltando apenas as descrições.

4.3 COMENTÁRIOS COMUNS (MARCAÇÕES)

Alguns comentários são bastantes comuns durante a codificação e, portanto, serão tratados como marcações.

A tabela a seguir traz os comentários comuns.

Marcação	Descrição
`Nothing To Do`	Deve ser usado quando o escopo de uma instrução não possui efeito algum. Normalmente usado quando uma instrução *default*, como o `else` ou o próprio `default` de um `switch` não possui efeitos.
`Write Instructions Here`	Usado para avisar sobre a necessidade de uma implementação numa dada parte do código.
`Empty Constructor`	Usado para alertar que o construtor é vazio e deve permanecer vazio.
`Empty Method`	Usado para alertar que o método é vazio e deve permanecer vazio.

Esse tipo de comentário deve ser escrito da forma que é demonstrada no exemplo:

```
/*! Empty Constructor. */
```

5. DECLARAÇÕES

5.1. UMA DECLARAÇÃO POR LINHA

Apenas uma declaração de variável é permitida por linha.

```
private int sum = 0;
private int firstParcel = 0;
private int secondParcel = 0;
private int fourthParcel = 0;
```

A forma a seguir é considerada errada e não é aceita.

```
private int firstParcel = 0, secondParcel = 0, fourthParcel = 0;
```

5.2. INICIALIZAÇÃO

Todas as variáveis devem ser inicializadas ao serem declaradas.

5.3. DECLARAÇÃO DE CLASSES E INTERFACES

Ao codificar as classes e interfaces Java, as seguintes regras de formatação devem ser seguidas:

- Sem espaço entre um nome de método e os parênteses que contém a lista de parâmetros.
- Abrir chave "{" ao final da mesma linha que a instrução de declaração da classe ou interface.
- Chave de fechamento "}" inicia uma linha por si só recuado para coincidir com a sua declaração de abertura correspondente. Em métodos ou construtores vazios a regra deve se manter, escrevendo um comentário dentro do escopo do método/construtor vazio com os seguintes dizeres: "`Empty Constructor`" ou "`Empty Method`".
- Os métodos são separados por uma linha em branco.

Um exemplo dessa declaração:

```
public class ExampleClass {
    private int sum = 0;
    private int firstParcel = 0;
    private int secondParcel = 0;
    private int fourthParcel = 0;

    public ExampleClass() {
        /* Empty Constructor. */
    }

    public int sum() {
        sum = firstParcel + ( secondParcel + 4 ) +
                fourthParcel + 7;

        return sum;
    }

    public void someMethod( int firstParameter, int secondParameter,
            int thirdParameter, String fourthParameter ) {

        /* Write instructions here. */
    }
}
```

6. Instruções e Estruturas

6.1. Instruções Simples

Cada linha deve conter no máximo uma instrução.

6.2. Instruções Compostas

Instruções compostas são declarações que contêm listas de instruções entre chaves "`{ instruções }`".

6.3. Instruções de Retorno

Instruções de retorno devem ser sucedidas exclusivamente por variáveis, exceto que haja uma boa razão para o contrário.

6.4. Estrutura `If` e `Else`

Declarações de `If - Else` devem seguir os seguintes formatos:

```
if( condition ) {
    /* Write instructions here. */

} else {
    /* Write instructions here.
     * or
     * Nothing To Do.
     */
}
```

```
if( condition ) {
    /* Write instructions here. */

} else if( condition ) {
    /* Write instructions here. */

} else {
    /* Write instructions here.
     * or
     * Nothing To Do.
     */
}
```

Deve-se lembrar que numa instrução `If` deve-se sempre utilizar chaves e o comportamento *default* (`else`).

6.5. ESTRUTURA `For`

Declarações de `For` devem seguir o seguinte formato:

```
for( int i = 0; i < length; i++ ) {
    /* Write instructions here. */
}
```

Ao usar o operador vírgula na cláusula de inicialização ou atualização de uma instrução `For`, evitar a complexidade do uso de mais de três variáveis.

6.6. ESTRUTURA `While`

Declarações de `While` devem seguir o seguinte formato:

```
while( condition ) {
    /* Write instructions here. */
}
```

6.7. ESTRUTURA `Do - For`

Declarações de `While` devem seguir o seguinte formato:

```
do {
    /* Write instructions here. */

} while( condition );
```

6.8. ESTRUTURA SWITCH

Declarações de `Switch` devem seguir o seguinte formato:

```
switch( sum ) {
    case 1:
        /* Write instructions here. */
        break;

    case 2:
        /* Write instructions here. */
        break;

    case 3:
        /* Write instructions here. */
        break;

    default:
        /* Write instructions here. */
}
```

6.9. ESTRUTURA TRY - CATCH

Declarações de `Try - Catch` devem seguir o seguinte formato:

```
try {
    /* Write instructions here. */

} catch( Exception e ) {
    /* Write instructions here. */
}
```

A instrução `Try - Catch` pode também ser seguido por, `finally`, que executa independentemente de haver ou não concluído o bloco `Try` com êxito.

```
try {
    /* Write instructions here. */

} catch( Exception e ) {
    /* Write instructions here. */

} finally {
    /* Write instructions here. */
}
```

7. ESPAÇOS EM BRANCO

7.1. LINHAS EM BRANCO

Duas linhas em branco devem ser sempre usadas nas seguintes circunstâncias:

- Entre seções de um arquivo fonte.
- Entre definições classes e interfaces.

Uma linha em branco deve ser sempre usada nas seguintes circunstâncias:

- Entre métodos.
- Entre uma variável local em um método e sua primeira instrução.
- Antes de um comentário em bloco ou um comentário de uma única linha.

7.2. ESPAÇOS EM BRANCO

7.2.1. EM DECLARAÇÕES

Em qualquer instrução que faça uso de múltiplos campos separados por vírgulas, coloque um espaço em branco após a vírgula.

Em qualquer instrução que faça uso de parênteses, coloque um espaço em branco separando o conteúdo interno dos parênteses em si.

```
void make( int arg1, int arg2, String arg3 ) throws EOFException, Exception {
    /* Write instructions here. */
}
```

7.2.2. Em Estruturas

Em qualquer estrutura que faça uso de parênteses, coloque um espaço em branco separando o conteúdo interno dos parênteses em si.

```
if( condition ) {
    /* Write instructions here. */

} else {
    /* Write instructions here.
     * or
     * Nothing To Do.
     */
}
```

1 ESTRUTURA DE IF-ELSE

```
for( int i = 0; i < length; i++ ) {
    /* Write instructions here. */
}
```

2 ESTRUTURA DE FOR

```
do {
    /* Write instructions here. */

} while( condition );
```

3 ESTRUTURA DO-WHILE

```
while( condition ) {
    /* Write instructions here. */
}
```

4 ESTRUTURA WHILE

```
switch( sum ) {
    case 1:
        /* Write instructions here. */
        break;

    case 2:
        /* Write instructions here. */
        break;

    case 3:
        /* Write instructions here. */
        break;

    default:
        /* Write instructions here. */
}
```

5 ESTRUTURA SWITCH

```
try {
    /* Write instructions here. */

} catch( Exception e ) {
    /* Write instructions here. */

} finally {
    /* Write instructions here. */
}
```

6 ESTRUTURA TRY-CATCH

7.2.3. Em Expressões

Em qualquer instrução que faça uso de operadores binários, coloque um espaço em branco em ambos os lados do operador binário.

```
int a = -4 + -9;
b = a++ / --sum;
c += 4;
```

Na chamada de métodos, coloque um espaço e branco depois de abrir o parêntese e antes de fechá-lo.

```
surf( );
make( a, b, d );

String string = new String( );
Point point = new Point( a, b );
```

No uso de parênteses em expressões, coloque um espaço e branco depois de abrir o parêntese e antes de fechá-lo.

```
result = ( a * ( b + c + g ) * ( e - f ) );
```

7.2.4. EM *ARRAYS*

Na alocação de *arrays*, coloque um espaço e branco depois de abrir o colchete e antes de fechá-lo.

No acesso de elementos do *array*, coloque um espaço e branco depois de abrir o colchete e antes de fechá-lo.

```
int[] array1 = new int[] { 1, 2, 3  };
int[] array2 = new int[ 3 ];

array2[ 1 ] = array1[ 2 ];
```

1. Convenção de Nomes

TABELA 2: CONVENÇÃO DE NOMES

Identificador	Regras de Nome	Exemplo
Pacotes	Os prefixos do pacote devem seguir os mostrados na Tabela 3. O sufixo do nome do pacote deve ser sempre em letras minúsculas.	`br.unb.sma.dd.world.items`

Classes	Os nomes de classe devem ser substantivos, em *upper camel case*. Tente manter seus nomes de classe simples e descritivos. Use palavras completas - evitar siglas e abreviaturas (a não ser que a sigla seja muito mais usada do que a forma longa, como URL ou HTML).	`class Item;`
		`class Dice;`
		`class LongSword;`

Interfaces	Nomes de interface devem ser escritos como nomes de classe.	`interface Storing;`
Agentes	Nomes de agentes devem ser escritos como nomes de classe, mas sempre terminados com a palavra `Agent`.	`class ItemsHandlerAgent;`
Behaviour	Nomes de agentes devem ser escritos como nomes de classe, mas sempre terminados com a palavra `Behaviour`.	`class WalkBehaviour;` `class FightBehaviour;`

Métodos	Os nomes de métodos devem ser iniciados com um verbo e escritos utilizando *lower camel case.*	`run();`
		`addBehaviour();`
		`walkOnStreet();`

Variáveis	Os nomes de variáveis devem ser substantivos e escritos utilizando *lower camel case.*
	Os nomes das variáveis **não** devem
		`int i;`
		`double fisrtParcel;`
		`String personName;`

	começar com underscore _ ou sinal de dólar $, mesmo que ambos sejam permitidos pela linguagem Java. Os nomes das variáveis devem ser curtos, mas significativos. Nomes de variáveis de um caractere devem ser evitados, exceto os nomes `i`, `j` e `k` em seus locais usuais como o `for`.	
Constantes	Os nomes de variáveis constantes devem possuir todas as letras maiúsculas com palavras separadas por `underscore`.	`static final int CAR = 1;`

TABELA 3: CONVENÇÃO DE PREFIXOS DE PACOTE

Prefixos de Pacote	Descrição do Pacote
br.unb.sma.dd.world	Devem estar presentes em pacotes com classes e agentes relacionados diretamente com o mundo de jogo.
br.unb.sma.dd.metaworld	Devem estar presentes em pacotes com classes e agentes que tenham relacionamento com o mundo de forma indireta.
br.unb.sma.dd.rules	Estes prefixos devem estar presentes em pacotes que carreguem classes e agentes responsáveis por moldar as regras de jogo, mas não possuam relação direta ou indireta com o mundo de jogo.
br.unb.sma.dd.behaviours	Estes prefixos devem estar presentes em pacotes que carreguem classes do tipo `behaviour`.
br.unb.sma.dd.protocols	Estes prefixos devem estar presentes em

	pacotes que carreguem classes que moldem protocolos de troca de mensagem específicas ao sistema implementado.
br.unb.sma.dd.infrastructure	Estes prefixos devem estar presentes em pacotes que carreguem classes e agentes responsáveis por suportar a estrutura do jogo, mas não possuam relação direta ou indireta com o mundo de jogo. Um exemplo seriam classes de persistência de dados.

REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. **Java:** Como Programar. 8ª Edição. ed. [S.l.]: Prentice Hall, 2010.

HOMMEL, S. et al. **Java Code Conventions**. Sun Microsystems. [S.l.].

STACKOVERFLOW. Javadoc - Paragraph Separator. **StackOverFlow**, 2011. Disponivel em: <http://stackoverflow.com/questions/5260368/javadoc-paragraph-separator>. Acesso em: 8 Junho 2014.