Guia do Codificador

Sumário

1. Introdução 4

2. Preparação do Ambiente de Desenvolvimento 4
- 2.1 Softwares para desenvolvimento 4
- 2.2 Redmine 4
- 2.3 Repositório 4
- 2.4 Registrando a evolução diária (redmine) 5
- 2.5 Teste 5
- 2.6 Produção 5

3. Convenções 5
- 3.1 Sufixos 5
- 3.2 Nomes Comuns 5
- 3.3 Organização dos Arquivos 5
- 3.4 Começando com um comentário 5
- 3.5 Identação 6
- 3.6 Comprimento da Linha 6
- 3.7 Quebrando Linhas 6
- 3.8 Comentários 6
  - 3.8.1 Comentário de bloco 6
  - 3.8.2 Descrições breves 6
  - 3.8.3 Comentando métodos 6
  - 3.8.4 Comentando Atributos 7
- 3.9 Declarações 7
  - 3.9.1 Uma declaração por linha 7
  - 3.9.2 Localização 7
  - 3.9.3 Inicialização 8
  - 3.9.4 Declaração de Classes e Interfaces 8
- 3.10 Intruções 8
  - 3.10.1 Instruções simples 8
  - 3.10.2 Instrução return 9
  - 3.10.3 Instruções if, if-else, if-else-if-else 9
  - 3.10.4 Instruções for 9
  - 3.10.5 Instruções while 9
  - 3.10.6 Instrução do-while 10
  - 3.10.7 Instruções switch 10
  - 3.10.8 Regras Gerais para Instruções 10
- 3.11 Espaço em branco 11
  - 3.11.1 Linhas em branco 11
  - 3.11.2 Espaços em branco 11

4 Convenções de Nomes 12
- 4.1 Práticas de programação 12
  - 4.1.1 Prover acesso a instâncias e membros de classe 12
  - 4.1.2 Referindo-se a variáveis de classe e métodos 13
  - 4.1.3 Constantes 13
  - 4.1.4 Atribuição de variáveis 13
- 4.2 Práticas diversas 13
  - 4.2.1 Parênteses 13
  - 4.2.2 Valores de retorno 13
  - 4.2.3 Expressões antes de ‘?’ no Operador Condicional 14

4.2.4 Comentários Especiais 14
4.2.5 Declaração de Tipos 14
4.2.6 Incluindo Arquivos 15

1. Introdução

O Guia do Codificador objetiva disciplinar a atividade de codificação, abrangendo a Preparação do Ambiente de Desenvolvimento, o Ciclo de Trabalho do Desenvolvedor e finalmente a Padronização de Código. Cada um destes aspectos será detalhado nas seções a seguir.

2. Preparação do Ambiente de Desenvolvimento

2.1 Softwares para desenvolvimento

A lista de softwares necessários para configurar o ambiente de desenvolvimento encontra-se na secção "1.6 Preparando o Ambiente de Desenvolvimento (JAVA e PHP)" do PDS.

2.2 Redmine

O Redmine é a ferramenta para Gerenciamento de Projetos utilizada pela DTI. Nesta seção serão introduzidos alguns conceitos básicos sobre o uso do Redmine.

O histórico completo do ciclo de vida de um Projeto está no Redmine. O codificador receberá suas tarefas e poderá registrar o andamento do seu trabalho através de tarefas do Redmine. A documentação e os códigos-fonte do Projeto encontram-se no Repositório do Projeto (Subseção 2.3).

2.3 Repositório

Um **sistema de controle de versão** (ou *versionamento*), **VCS** (do inglês *version control system*) ou ainda **SCM** (do inglês *source code management*), é um *software* com a finalidade de gerenciar diferentes versões no desenvolvimento de um documento qualquer. Esses sistemas são comumente utilizados no desenvolvimento de *software* para controlar as diferentes versões — histórico e desenvolvimento — dos códigos-fontes e também da documentação. A DTI adotou o Subversion (SVN) como sistema de controle de versões. Para integrar o SVN ao menu de contexto do Windows, adotamos a utilização do TortoiseSVN.

Para cada projeto do Redmine é disponibilizado um **repositório** SVN, que nada mais é do que um diretório raiz sob controle de um vocês. Sempre que uma alteração ocorre neste diretório é criada uma **revisão**, ou seja, é armazenada uma versão com o estado do repositório antes que as alterações sejam efetuadas. Caso um arquivo tenha sido substituído por acidente, é possível restaurá-lo; através do **diff** é possível verificar as mudanças que ocorreram entre diferentes revisões do arquivo no repositório.

O desenvolvedor realiza três operações básicas para utilizar o repositório:

- **Checkout**: faz uma cópia do repositório remoto para trabalhar localmente.
- **Update**: atualiza a cópia local do repositório com os novos arquivos e alterações existentes no repositório remoto.
- **Commit**: envia os arquivos alterados localmente para o repositório remoto.

Através do seu login no Redmine, o desenvolvedor terá acesso ao Repositório do Projeto que está trabalhando. Será disponibilizada na Wiki do projeto as instruções para realização do checkout. Em linhas gerais, o desenvolvedor deverá ter instalado em seu computador o TortoiseSVN e conhecer o endereço do repositório.

Esta seção não objetiva fazer uma apresentação extensa sobre controle de versões, procurando apenas introduzir as nomenclaturas que serão empregadas neste e em outros documentos. Para maiores esclarecimentos sobre o controle de versões e sobre a ferramenta TortoiseSVN, sugerimos a leitura do seguintes materiais na web:

2.4 Registrando a evolução diária (redmine)

Atualize sua tarefa no Redmine com o andamento do trabalho diariamente. Informe o que foi feito, ou ainda falta fazer, quais dificuldades foram encontradas, etc. À medida que a tarefa vai sendo executada, altere o percentual de conclusão em conformidade com o andamento do seu trabalho.

2.5 Teste

O Servidor de teste disponibiliza uma estrutura semelhante ao servidor de Produção e está acessível a todos os envolvidos no Projeto. O Testador deverá publicar os arquivos fontes e se certificar que a base de dados está condizente com a necessidade do software.

2.6 Produção

Após a conclusão bem sucedida das tarefas de Teste e Correção de Bugs, o Gerente de Código solicita ao Gerente de Projeto que o sistema seja colocado em Produção.

3. Convenções

As convenções de código deste Guia estão baseadas no *Code Convention* da SUN, que está disponível no endereço http://www.oracle.com/technetwork/java/codeconv-138413.html. Para maiores esclarecimentos, procure ler *Code Convention* na íntegra.

3.1 Sufixos

- Arquivo Fonte PHP: .php

3.2 Nomes Comuns

- README (em caixa alta): explicar o conteúdo do diretório.

3.3 Organização dos Arquivos

- Evitar arquivos com mais de 2000 linhas

3.4 Começando com um comentário

```
/**
 * \file NomeDoArquivo.php
 *
 * Descrição completo do arquivo.
 *
 * \author nome do autor 1
 * \author nome do autor 2
 * \author nome do autor n
 * \date data da criação do arquivo
 * */
```

3.5 Identação

Quatro espaços devem ser usado como unidade de identação. Tabs devem ser setados exatamente para 8 espaços.

3.6 Comprimento da Linha

Evite linhas com mais do que 80 caracteres

3.7 Quebrando Linhas

Quando as expressões não cabem em uma única linha, quebre-as de acordo com os seguintes princípios:

- Quebre após uma vírgula
- Quebre depois do operador
- Alinhe a nova linha com o começo da expressão no mesmo nível da linha acima

Veja alguns exemplos diretamente no *Code Conventions* da SUN.

3.8 Comentários

3.8.1 Comentário de bloco

Comentários em C-style, iniciando com *'s:

```
/**
 * ... text ...
 */
```

3.8.2 Descrições breves

Para descrições breves utilize `\brief`. Este comando termina no final do parágrafo, portanto, a descrição detalhada aparece após a linha em branco.

```
/*! \brief Brief description.
 *         Brief description continued.
 *
 *  Detailed description starts here.
 */
```

3.8.3 Comentando métodos

parâmetros

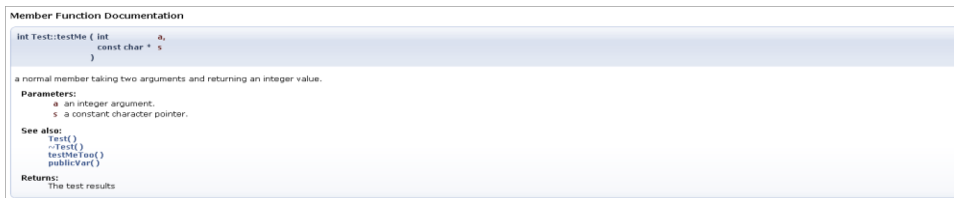
```
/**
 * a normal member taking two arguments and returning an integer value.
 * @param a an integer argument.
 * @param s a constant character pointer.
```

"veja também"

```
        * @see Test()
        * @see ~Test()
        * @see testMeToo()
        * @see publicVar()
        * @return The test results
        */
        int testMe(int a,const char *s);
```

retorno

Resultado (HTML):



3.8.4 Comentando Atributos

```
int var; /* Detailed description after the member */
```

3.9 Declarações

3.9.1 Uma declaração por linha

Uma declaração por linha é recomendável, uma vez que encoraja o uso de comentários. Em outras palavras:

```
int level;  // identation level
int size;   // size of table
```

Use o mesmo nível de identação para os comentários.

3.9.2 Localização

Coloque as declarações somente no início dos blocos (um bloco é qualquer código envolvido por chaves "{" e "}".

Exemplos:

```
void MyMethod() {
        int int1; // beginning of method block

        if (condition) {
                int int2; // beginning of "if" block
                ...
        }
}
```

A única exceção para esta regra são os índices dos laços FOR:

```
for (int i = 0; i < maxLoops; i++) { ...
```

Evite declarações locais que escondem declarações de alto nível. Por exemplo, não declare o mesmo nome de variável em blocos internos:

```
int count;
...
func() {
        if (condition) {
                int count; // EVITE!
                ...
        }
        ...
}
```

3.9.3 Inicialização

Procure inicializar variáveis locais onde elas são declaradas. A única razão para inicializar a variável onde ela não foi declarada é se o valor inicial precise que alguma computação seja realizada antes.

3.9.4 Declaração de Classes e Interfaces

Observe as seguinte regras:

- Sem espaço entre o nome do método e o parênteses "(" que inicia a lista de parâmetros
- Abra chaves "{" no final da mesma linha da declaração
- Feche chaves "}" em uma linha separada de modo que a identação corresponda com a abertura de chaves "{", exceto quando abrir e fechar sem nenhum conteúdo entre eles.

Exemplos:

```
class Sample extends Object {
        int ivar1;
        int ivar2;

        Sample(int i, int j) {
                ivar1 = i;
                ivar2 = j;
        }

        int emptyMethod() {}

        ...

}
```

- Métodos são separados por uma linha em branco.

3.10 Intruções

3.10.1 Instruções simples

Cada linha deve conter uma única instrução. Exemplo:

```
argv++; argc--; // EVITE!
```

Não use o operador vírgula para agrupar múltiplas instruções

3.10.2 *Instrução* `return`

Uma instrução return com um valor não deve utilizar parênteses, a menos que eles tornem o valor de retorno mais óbvio de alguma forma. Exemplos:

```
return;
return myDisk.size();
return (size ? size : defaultSize);
```

3.10.3 *Instruções if, if-else, if-else-if-else*

Utilize as formas abaixo:

```
if (condition) {
     statements;
}

if (condition) {
     statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else if (condition) {
    statements;
}
```

Nota: instrução `if` sempre usa chaves {}. Evite a seguinte forma:

```
if (condition) //EVITE! ESTA FORMA OMITE OS {}!
    statement;
```

3.10.4 *Instruções* `for`

Uma instrução `for` deve ter a seguinte forma:

```
for (initialization; condition; update) {
     statements;
}
```

Uma instrução `for` em branco deve ter a seguinte forma:

```
for (initialization; condition; update);
```

3.10.5 *Instruções while*

Uma instrução `while` deve ter a seguinte forma:

```
while (condition) {
      statements;
}
```

Uma Instrução `While` em branco deve ter a seguinte forma:

```
while (condition);
```

3.10.6 Instrução do-while

A instrução `do-while` deve ter a seguinte forma:

```
do {
   statements;
} while (condition);
```

3.10.7 Instruções switch

A instrução `switch` deve ter a seguinte forma:

```
switch (condition) {
case ABC:
       statements;
       /* falls through */
case DEF:
       statements;
       break;

case XYZ:
       statements;
       break;

default:
       statements;
       break;
}
```

- Sempre que um `case` *falls through* (não inclui uma declaração `break`), adicione um comentário onde a instrução `break normalmente deveria estar.`
- Toda a instrução `switch` deve incluir um `default case`. O break no default case é redundante, mas previne que um erro fall-through ocorra caso outro case seja adicionado.

3.10.8 Regras gerais para instruções

Estruturas de controle são por exemplo, “if”, “for”, “foreach”, “while”, “switch”, etc. A baixo, um exemplo com “if”:

```
if ((expr_1) || (expr_2)) {
   // ação_1;
} elseif (!(expr_3) && (expr_4)) {
   // ação_2;
} else {
   // ação_padrão;
}
```

- Nas estruturas de controle deve existir 1 (um) espaço antes do primeiro parêntese e 1 (um) espaço entre o último

parêntese e a chave de abertura.
- Sempre use chaves nas estruturas de controle, mesmo que não sejam necessárias. Elas melhorar a leitura do código e tendem a causar menos erros lógicos.
- A abertura da chave deve ser posicionada na mesma linha que a estrutura de controle. A chave de fechamento deve ser colocada em uma nova linha e ter o mesmo nível de identação que a estrutura de controle. O conteúdo de dentro das chaves deve começar em uma nova linha e receber um novo nível de identação.
- Atribuições em linha não devem ser usadas dentro de estruturas de controle.

3.11 Espaço em branco

3.11.1 Linhas em branco

Linhas em branco melhoram a legibilidade ao seccionar logicamente os blocos de código.

Duas linhas em branco devem sempre ser utilizadas:

- Entre seções do código fonte
- Entre definições de classe e interface

Uma linha deve sempre ser utilizada:

- Entre métodos
- Entre variáveis locais do método e a primeira instrução
- Antes de um bloco ou comentários de uma linha
- Entre seções lógicas para melhorar a legibilidade

3.11.2 Espaços em branco

Espaços em branco devem ser utilizados nas seguintes circunstâncias:

- Uma palavra chave seguida de parênteses deve ser separada por espaço: Exemplo:

```
while (true) {
      ...
}
```

  - *Espaço em branco não deve ser utilizado entre o nome de um método e o seu parênteses de abertura. Isto ajuda a distinguir palavras chave de chamadas de métodos.*

- Um espaço em branco deve aparecer depois de vírgulas em listas de argumento

- Todo operador binário, exceto "." deve ser separado de seus operandos por espaços. Espaços em branco nunca separar operados unários, tais como menos unário, incremento ("++"), e decremento ("—") de seus operandos. Exemplo:

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

while (d++ = s++) {
       n++;
}
prints("size is " + foo + "\n");
```

- As expressões na instrução `for` devem ser separadas por espaços em branco. Exemplo:

```
for (expr1; expr2; expr3)
```

4. Convenções de Nomes

Convenções de nomes tornam os programas mais compreensíveis ao torná-los fáceis de ler. Elas também podem dar informações sobre a função e o identificador – por exemplo, seja uma constante, um pacote, uma classe – o que pode útil na compreensão do código.

Tenha em mente que as convenções desta seção são de alto nível.

Tipo de Identificador	Regras para nomeação	Exemplos
Classes	Nomes que começam com a primeira letra em maiúsculo e com a primeira letra dos nomes internos também em maiúsculo. Procure deixar o nome das suas classes simples e descritivas. Uso palavras completas – evite acrônimos e abreviações (exceto quando a abreviação é amplamente utilizada, como URL ou HTML, por exemplo).	`class Raster;` `class ImageSprite;`
Intertface	Se aplica a mesma regra para nomeação de classes.	`interface RasterDelegate;` `interface Storing;`

Methods	Métodos devem ser verbos, começando com letra minúscula e separando os nomes internos com letra maiúscula.	`run();`
		`runFast();`
		`getBackground();`

Varíaveis	Começam com letra minúscula e os nomes internos são separados com letra maiúscula.
	Variáveis devem ter nomes curtos, mas compreensíveis. A escolha do nome da variável deve ser mnemônica – ou seja, indica a um observador casual a intenção de usá-la. Nome de variáveis com uma letra deve ser evitado, exceto para variáveis temporárias ("descartadas" após o uso). Nomes comuns para variáveis temporárias são `i`, `j`, `k`, `m`, `n` para inteiros, e `e` para caracteres.
		`int i;`
		`char *cp;`
		`float myWidth;`

Constantes	O nome de uma constante deve ser todo maiúsculo, com palavras separadas por underscores ("_").	`int MIN_WIDTH = 4;`
		`int MAX_WIDTH = 999;`
		`int GET_THE_CPU = 1;`

4.1 Práticas de programação

4.1.1 Prover acesso a instâncias e membros de classe

Um exemplo de instância pública de variáveis é quando a classe é essencialmente uma estrutura de dados, sem comportamento. Em outras palavras, se você precisar usar uma `struct` de uma classe, então é apropriado tornar as instâncias de classe públicas.

4.1.2 Referindo-se a variáveis de classe e métodos

Evite usar um objeto para acessar variáveis e métodos de classe (estáticos). Use o nome da classe ao invés disso. Exemplos:

```
classMethod(); //OK
AClass.classMethod(); //OK
anObject.classMethod(); //EVITE!
```

4.1.3 Constantes

Constantes numéricas não devem ser codificados diretamente, exceto para -1, 0 e 1; que podem aparecer no laço `for` como valores dos contadores do laço.

4.1.4 Atribuição de variáveis

Evite múltiplas atribuições de variáveis ao mesmo tempo. Isto é difícil de ler. Exemplo:

```
fooBar.fChar = barFoo.lchar = 'c'; // EVITE!
```

Não atribua operadores no lugar onde seja facilmente confundido com o operador de igualdade. Exemplos:

```
if (c++ = d++) { // EVITE! (não é permitido em algumas linguagens)
    ...
}
```

Deve ser escrito assim

```
if ((c++ = d++) != 0) {
    ...
}
```

Não use operadores embutidos na tentativa de melhorar a performance. Isto é trabalho para os compiladores, e raramente isto ajuda. Exemplo:

```
d = (a = b + c) + r; // EVITE!
```

deve ser escrito:

```
a = b + c;
d = a + r;
```

4.2 Práticas diversas

4.2.1 Parênteses

Geralmente é uma boa ideia usar parênteses livremente nas expressões envolvendo mistura de operadores para evitar problemas de precedência. Ainda que a precedência de operadores parece clara para você, ela pode não ser outros – você não deve assumir que o outro programador conhece as precedências tão bem quanto você.

```
if (a == b && c == d) // EVITE!
if ((a == b) && (c == d)) // CERTO
```

4.2.2 Valores de retorno

Procure tornar a estrutura dos seus programas compatíveis com a sua intenção. Exemplos:

```
if (booleanExpression) {
   return TRUE;
} else {
   return FALSE;
}
```

deve ser escrito como:

```
return booleanExpression;
```

Similarmente

```
if (condition) {
   return x;
} else {
   return y;
}
```

deveria ser escrito como

```
return (condition ? x : y);
```

4.2.3 Expressões antes de '?' no Operador Condicional

Se uma expressão contém um operador binário antes do ? no operador ternário ?:, ele deve receber parênteses. Exemplo:

```
(x >= 0) ? x : -x
```

4.2.4 Comentários especiais

Use `GBR` para indicar (flag) para indicar uma "gambiarra" que funciona. Use `FIXME` para indicar que alguma coisa é gambiarra e não funciona muito bem. Códigos que possuam a tag `FIXME` não deverão ser comitados; esta tag serve para controle do próprio programador.

4.2.5 Declaração de Tipos

Ao implementar uma nova função, a descrição dos tipos recebidos deve ser comentada logo acima da sua declaração, assim como uma descrição detalhada sobre o método/função, indicando também o retorno recebido pelo mesmo:

```
/**
 * Descrição do método.
 *
 * @param $data Algum valor do tipo date, formato "YYYY-mm-dd".
 * @param $dia Algum valor em formato inteiro.
 * @param $boolean Algum valor booleano.
 */
public function foo($data, $dia, $boolean)
```

```
{
}
```

4.2.6 Incluindo Arquivos

Include, require, include_once e require_once não tem parênteses:

```
// errado = com parênteses
require_once('ClassFileName.php');
require_once ($class);

// certo = sem parênteses
require_once 'ClassFileName.php';
require_once $class;
```