Adobe Flex Coding Guidelines

MXML e ActionScript 3

Padrões de codificação utilizados na DClick
para programação com o Adobe Flex

v1.2 – Fevereiro/2007

Fabio Terracini
fabio.terracini@dclick.com.br

1. Introdução

Este documento visa estabelecer padrões de codificação para aplicações escritas com o Adobe Flex 2 e ActionScript 3.0.

Estabelecer uma padronização de código é importante por que muito do tempo no ciclo de desenvolvimento de um software é manutenção. Assim, facilitar a compreensão de determinada passagem de código é imperativo, ainda mais levando-se em conta que nem sempre o autor de determinada passagem será o que fará a manutenção. Estabelecendo uma linguagem comum, os desenvolvedores poderão entender o código mais rapidamente. Igualmente, o código de aplicações ou componentes podem ser distribuidos ou vendidos para terceiros.

As premissas para a padronização do código são:

- Consistência;
- Compreensão do código.

As práticas desses documentos são baseadas no modo de trabalho da DClick, nas práticas já utilizadas em programação Java e nos padrões utilizados pela própria Adobe no Flex SDK.

2. Arquivos

2.1. Sufixos de arquivos

- Código MXML: .mxml
- Código ActionScript: .as
- Código CSS: .css

2.2. Nomes de arquivos

- Não devem conter espaços, pontuação ou caracteres especiais;
- ActionScript
    - Classes e Interfaces utilizam *UpperCamelCase*;
    - Interfaces sempre contém uma letra *I* inicial;
        - *IUpperCamelCase*
    - Includes utilizam *lowerCamelCase*;
    - Definições de namespaces utilizam *lower_case*.
- MXML
    - Sempre utilizam *UpperCamelCase*.
- CSS
    - Sempre utilizam *lowerCamelCase*.

2.3. Codificação

- Os arquivos sempre devem estar em UTF-8.

3. ActionScript 3.0

3.1. Organização do arquivo

Um arquivo ActionScript deve conter a seguinte estrutura:

#	**Elemento**	**Observação**
1	Comentário inicial	
2	Definição de pacote	
3	Declaração de namespace • Se o for, esta é a última seção do arquivo	Um arquivo que define um namespace só deve ser responsável por isso.
4	Declarações de *import* 1. Package *flash* 2. Package *mx* 3. Package *com.adobe* 4. Package *br.com.dclick* (componentes da DClick) 5. Packages de componentes de terceiros em ordem alfábetica 6. Package referente ao projeto no qual aquele arquivo está contido Utilize imports *fully qualified*, isto é, sem o asterisco. Exceto quando grande parte deste pacote for utilizada na classe em questão. • Prefira: `import mx.core.Application;` • Evite: `import mx.core.*;`	Dentro das segmentações, os imports devem estar ordenados alfabéticamente. Se utilizar o import de um namespace, este deve suceder o import de classes daquele mesmo pacote.
5	Declarações *use* (namespace)	Em ordem alfabética.
6	Metadados 1. Event 2. Style 3. Effect 4. Demais metadados em ordem alfabética	
7	Definição da classe ou interface.	
8	Variáveis static 1. *public* a. *const* b. Demais *public static* 2. *internal* 3. *protected*	

	4. *private* 5. namespaces customizados a. Em ordem alfabética	
9	Variáveis de instância não gerenciados por *get* e *set* 1. *public* 2. *internal* 3. *protected* 4. *private* 6. namespaces customizados a. Em ordem alfabética	
10	Construtor	

11	Variáveis gerenciadas por *get* e *set*, junto com os métodos de *get* e *set*, bem como variáveis de relacionadas (agrupadas por funcionalidade). Exemplo:
	
	`private var _enabled:Boolean = true;`
	`private var enabledChanged:Boolean = false;`
	
	`public function get enabled():Boolean`
	`{`
	`  return _enabled;`
	`}`
	
	`public function set enabled(value:Boolean):void`
	`{`
	`  _enabled = value;`
	`  enabledChanged = true;`
	`}`
		Vide seção específica de variáveis para regras de variáveis manipuladas por get e set.

12	Métodos	Agrupados por funcionalidade, não por escopo.

3.2. Estilo

3.2.1. Linha e quebra de linha

Quando uma expressão não couber em apenas uma linha, quebre-a em mais de uma linha. Assim, caso haja a necessidade de uma quebra de linha, esta deve seguir as seguintes regras:

- Quebre após uma vírgula;
- Quebre antes de um operador;
- Prefira quebras de linha em níveis mais altos de código;
- Alinhe a nova linha no começo da expressão a qual ela pertence na linha superior;
- Ou, se a regra acima não for uma boa opção, idente com dois tabs.

Prefira:

```
// linha #1: quebra de linha antes do operador implements
// linha #2: quebra de linha após a virgula
// linhas #2 e #3: identadas com dois tabs
public class Button extends UIComponent
            implements IDataRenderer, IDropInListItemRenderer,
            IFocusManagerComponent
```

Evite:

```
public class Button extends UIComponent implements
      IDataRenderer, IDropInListItemRenderer,
      IFocusManagerComponent
```

Prefira:

```
// quebra em um nivel mais alto, ocorre fora do parenteses
// quebra de linha não quebra o conteúdo dentro do parenteses
variable1 = variable2 + (variable3 * variable4 - variable5)
            - variable6 / variable7;
```

Evite:

```
// quebra de linha separa o código do parenteses em duas linhas
variable1 = variable2 + (variable3 * variable4
             - variable5) - variable6 / variable7;
```

Exemplos de quebra de linha em operadores ternários:

```
b = (expression) ? expression
                 : gamma;      // alinhado!

c = (expression)
      ? beta
      : gamma;
```

3.2.2. Declarações

Faça apenas uma declaração por linha.

Certo:

```
var a:int = 10;
var b:int = 20;
var c:int = 30;
```

Errado:

```
var a:int = 10, b:int = 20, c:int = 30;
```

Sempre que possível inicialize a variável. Só não é necessário inicializar uma variável quando seu valor inicial depende de algum processamento ocorrer. Inicialize a variável mesmo que utilize seu valor padrão.

Certo:

```
public var isAdmin:Boolean = false;
```

Errado:

```
public var isAdmin:Boolean; // default é false
```

Coloque as declarações de variáveis no início do respectivo bloco, exceto por variáveis utilizadas em loop.

```
public function getMetadata():void
{
      var valor:int = 123;    // começo do bloco do método

      ...

      if (condition)
      {
            var valor2:int = 456;   // começo do bloco de if
            ...
      }

      for (var i:int = 0; i < valor; i++) // declarada no for
      {
            ...
      }

}
```

Não declare variáveis com nomes que já foram utilizados fora daquele bloco, mesmo que o escopo não seja o mesmo.

3.2.3. Chaves e parentêses

Regras de formatação:

- Não se usa espaço entre o nome do método e a abertura do parenteses, assim como não se usa espaço entre o parenteses e seus argumentos;
- Não se usa espaço para separar o nome do objeto de seu tipo;

- Abre-se chaves na linha inferior à última linha do bloco, na mesma posição no qual se inicia a declaração do método;
- Fecha-se chaves com a sua própria linha, na mesma posição em que a chave foi aberta.
- Métodos estão separados por uma linha vazia.

```
public class Exemplo extends UIComponent implements IExemplo
{

	private var _item:Object;

	public function addItem(item:Object):void
	{
		_item = item;
		...
	}

	public function anotherFunction():void
	{
		while (true)
		{
			...
		}
	}

}
```

3.2.4. Statements

Simples

Statements simples devem ser apenas um por linha e devem finalizar com ponto e vírgula.

Certo:

```
i++;
resetModel();
```

Errado:

```
i++; resetModel();
```

Compostos

Statements compostos (os que requerem `{` e `}`, como `switch`, `if`, `while`, entre outros) seguem as seguintes regras:

- O código dentro do statement deve estar identado em mais um nível;
- A chave deve estar na linha seguinte a do início da declaração do statement, alinhada na mesma posição. A chave fecha utilizando sua própria linha, na mesma posição da chave que abriu o statement;
- Chaves são utilizadas em todos os statements, mesmo que estes tenham apenas uma linha.

Return

O `return` de um valor não precisa utilizar parentêses a não ser que forneça um maior grau de entendimento:

```
return;

return getFinalImage();

return (phase ? phase : initPhase);
```

Condicional if, else if, else

```
if (condition)
{
      simpleStatement;
}


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

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

```
        statements;
}
```

Condicional switch, case

Statements de switch tem o seguinte formato:

```
switch (condition)
{

        case ABC:
        {
                statements;
                // continua, sem break
        }

        case DEF:
        {
                statements;
                break;
        }

        case JKL:
        case XYZ:
        {
                statements;
                break;
        }

        default:
        {
                statements;
                break;
        }
}
```

Regras de uso do break:

- Sempre utilize o `break` no `default` case. Normalmente é redudante, mas reforça a idéia.
- Se um case não tiver o `break`, adicione um comentário onde normalmente seria o break.
- Não é necessário utilizar `break` se o statement interno for um `return`.

Loop for

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

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

Loop for..in

```
for (var iterator:type in someObject)
{
      statements;
}
```

Loop for each..in

```
for each (var iterator:Type in someObject)
{
      statements;
}
```

Loop while

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

Loop do..while

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

Tratamento try..catch..finally

```
try
{
      statements;
}
catch (e:Type)
{
      statements;
}
```

Pode incluir o finally:

```
try
{
```

```
        statements;
}
catch (e:Type)
{
        statements;
}
finally
{
        statements;
}
```

With

```
with (this)
{
        alpha = 0.5;
}
```

3.2.5. Espaçamentos

Quebras de linha

Quebras de linha aumentam a clareza do código criando agrupamentos lógicos.

Use uma quebra de linha:

- Entre funções;
- Entre variáveis locais de um método e seu primeiro statement;
- Antes de um bloco;
- Antes de um comentário de uma linha ou de comentários de várias linhas sobre um pequeno pedaço do código;
- Entre seções lógicas de código para melhorar a clareza

Espaços em branco

Utilize um espaço em branco para separar uma palavra-chave de seu parenteses e não utilize espaço para separar o nome de um método de seu parenteses.

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

Um espaço em branco deve existir depois de uma virgula em listas de argumentos.

```
addSomething(data1, data2, data3);
```

Todos os operadores binários (os com dois operandos: +, -, =, ==, etc) devem ser separados de seus operandos por um espaço. Não se utiliza espaços para separar operadores unários (++, --, etc).

```
a += (5 + b) / c;

while (d as int < f)
{
        i++;
}
```

Operadores ternários devem ser separados por espaços e quebrados em mais de uma linha se necessário. Veja formas aceitáveis de uso:

```
a = (expression) ? expression : expression;
```

Expressões de for devem ser separadas por espaços em branco.

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

3.3. Comentários

3.3.1. Comentário de documentação

Comentários de documentação são para classes, interfaces, variáveis, métodos e metadados, com um comentário por elemento, antes da declaração. O comentário de documentação é para ser lido – e totalmente compreendido – por alguém que vai usar determinado componente mas não necessariamente tem acesso ao código fonte do arquivo.

O formato do comentário é o utilizado pela ferramenta ASDoc, e a sintáxe deve ser a contida no documento da ferramenta:
http://labs.adobe.com/wiki/index.php/ASDoc:Creating_ASDoc_Comments

Exemplo:

```
/**
 *  The Button control is a commonly used rectangular button.
 *  Button controls look like they can be pressed.
 *
 *  @mxml
 *
 *  <p>The <code>&lt;mx:Button&gt;</code> tag inherits all the
 *  tag attributes of its superclass, and adds the following:</p>
 *
 *  <pre>
 *  &lt;mx:Button
 *    <b>Properties</b>
```

```
 *    autoRepeat="false|true"
 *    ...
 *    <b>Styles</b>
 *    borderColor="0xAAB3B3"
 *    ...
 *  /&gt;
 *  </pre>
 *
 *  @includeExample examples/ButtonExample.mxml
 */
public class Button extends UIComponent
```

Outro exemplo:

```
/**
 *  @private
 *  Displays one of the eight possible skins,
 *  creating it if it doesn't already exist.
 */
mx_internal function viewSkin():void
{
```

3.3.2. Comentário de implementação

Comentários de implementação tem o propósito de documentar seções de código que não estejam evidentes. Os comentários devem iniciar com `//`, independente se são de múltiplas linhas ou de apenas uma linha.

Se ele for utilizar uma linha inteira, deve ser precedido de uma linha em branco e anteceder o pedaço de código relacionado:

```
// bail if we have no columns
if (!visibleColumns || visibleColumns.length == 0)
```

O comentário pode estar na mesma linha se não ultrapassar o tamanho permitido da linha:

```
colNum = 0; // visible columns compensate for firstCol offset
```

Não utilize comentários para traduzir o código.

```
colNum = 0; // seta a variável de número de colunas para zero
```

4. MXML

4.1. Organização do arquivo

Um arquivo MXML deve conter a seguinte estrutura:

#	Elemento	Observação
1	XML Header: `<?xml version="1.0" encoding="UTF-8"?>`	Sempre declare o encoding no XML header, e sempre utilize UTF-8.
2	Componente raiz	Já deve conter todos os namespaces utilizados no arquivo.
3	Metadados 1. Event 2. Style 3. Effect 4. Demais metadados em ordem alfabética	
4	Definições de estilo	Prefiro estilos externos, em arquivos .css
5	Scripts	Utilize apenas um bloco de Script.
6	Componentes não visuais	
7	Componentes visuais	

4.2. Estilo

4.2.1. Linha e quebra de linha

Utilize linhas em branco para dar clareza ao código, criando agrupamentos visuais de componentes.

Sempre adicione uma linha em branco entre dois componentes, filhos de um mesmo componente pai, caso a a declaração de pelo menos um deles (incluindo seus próprios filhos) ocupar mais de uma linha.

```
<mx:series>
      <mx:ColumnSeries yField="prev" displayName="Previsto">
            <mx:stroke>
```

```
                        <mx:Stroke color="0xB35A00" />
                  </mx:stroke>

                  <mx:fill>
                        <mx:LinearGradient angle="0">
                              <mx:entries>
                                    <mx:GradientEntry ... />
                                    <mx:GradientEntry ... />
                              </mx:entries>
                        </mx:LinearGradient>
                  </mx:fill>
            </mx:ColumnSeries>

            <comp:ColumnSeriesComponente />
      </mx:series>
```

Ou seja, se o componente tiver apenas um filho, não se deixa uma linha em branco. O `LinearGradient` abaixo contém apenas o filho `entries`.

```
<mx:LinearGradient angle="0">
      <mx:entries>
            <mx:GradientEntry ... />
            <mx:GradientEntry ... />
      </mx:entries>
</mx:LinearGradient>
```

Igualmente, como os filhos de `entries` não ocupam mais de uma linha, não há linha de espaçamento entre eles.

```
<mx:entries>
      <mx:GradientEntry ... />
      <mx:GradientEntry ... />
</mx:entries>
```

4.2.2. Aninhando componentes

Componentes filhos devem estar identados em relação ao seu componente pai:

```
<mx:TabNavigator>
      <mx:Container>
            <mx:Button />
      </mx:Container>
</mx:TabNavigator>
```

4.2.3. Atributos

Ordene os atributos em:

- Propriedades
    - A primeira propriedade deve ser o `id`, se existir.

    - Lembrando que `width`, `height` e `styleName` são propriedades, não estilos.
- Eventos
- Efeitos
- Estilos

Se estiver presente, o atributo `id` deve ser o primeiro a ser declarado:

```
<mx:ViewStack id="mainModules" height="75%" width="75%">
```

Os atributos devem estar identados em relação à declaração do componente caso a declaração total utilize mais de uma linha.

```
<mx:Label
      width="100%" height="100%" truncateToFit="true"
      text="Aqui vem um texto qualquer longo o suficiente..."/>
```

Em declaração que ocupem mais de uma linha, o único atributo que pode ir na mesma linha da declaração do componente é o `id`. Os outros atributos sempre devem vir nas demais linhas seguindo a ordenação.

```
<mx:ViewStack id="mainModules"
      height="75%" width="75%"
      paddingTop="10" paddingLeft="10" paddingRight="10">

<mx:ViewStack
      height="75%" width="75%"
      paddingTop="10" paddingLeft="10" paddingRight="10">
```

Coloque atributos relacionados na mesma linha. No exemplo abaixo, a segunda linha apenas contém propriedades, a terceira apenas eventos, a quarta apenas estilos, e a quinta apenas efeitos.

```
<mx:Panel
      title="VBox Container Example" status="Algum status"
      hide="doSomething()" creationComplete="doSomething()"
      paddingTop="10" paddingLeft="10" paddingRight="10"
      resizeEffect="Resize" />
```

Em casos que mais de uma linha for necessária para atributos do mesmo tipo, utilize mais de uma linha (mantendo a consistência de ordenação e de tamanho da linha), mantendo-os agrupados dentro do mesmo tipo. No exemplo abaixo, a primeira linha contém a declaração do componente com seu id, a segunda linha contém as propriedades, a terceira linha contém eventos, a quarta linha alguns estilos, a quinta linha outros estilos (padding agrupados na mesma linha). Não há declaração de efeitos.

```
<mx:Panel id="pnLoginInfo"
      title="VBox Container Example" height="75%" width="75%"
      resize="resizeHandler(event)"
```

```
titleStyleName="titleLogin" headerHeight="25"
paddingTop="10" paddingLeft="10" paddingRight="10" />
```

4.2.4. Script

Esse é o estilo a ser utilizado em blocos de `Script`:

```
<mx:Script>
      <![CDATA[

            code;

      ]]>
</mx:Script>
```

4.3. Comentários

4.3.1. Comentários de documentação

O ASDoc não suporta a geração de comentários de documentação a partir de documentos MXML. Contudo, fazê-lo é encorajado se o arquivo MXML for um componente que poderá ser utilizado por várias aplicações (e não apenas uma tela). Assim, ele deve conter um comentário de documentação seguindo o modelo do ActionScript - e dentro de um bloco de `Script`.

```
<mx:Script>
      <![CDATA[

            /**
             * Comentário de documentação de componente em MXML
             * Utiliza o mesmo molde do documentário ActionScript
             */

      ]]>
</mx:Script>
```

4.3.2. Comentários de implementação

Utilize comentários de implementação para descrever elementos da interface que não estejam evidentes ou seus comportamentos.

```
<!-- só aparece se perfil for de admin -->
```

Ou comentários de múltiplas linhas:

```
<!--
Comentário de múltiplas linhas...
...
...
-->
```

5. Estilo

5.1. Regras Gerais

- Idente utilizando tabs. A referência do tamanho de um tab é de 4 espaços, e é o sugerido a ser configurado na IDE pelo motivo de alinhamento em quebras de linha.
- As linha de código não devem ultrapassar 100 caracteres[1].

[1] Numa resolução de 1280 pixels de largura (ideal para monitores de 17") utilizando-se o Eclipse, se 70% da largura estiver disponível para o código (e os outros 30% para o Navigator), é possível escrever aproximadamente 103 caracteres. O limite para imprimir em uma página A4 é 80 caracteres.

6. Nomenclatura

6.1. Regras gerais

- Acrônimos: Evite o uso de acrônimos, a não ser que a abreviação seja mais utilizada que sua escrita por extenso (como por exemplo URL, HTML, etc). Nomes de projetos podem ser acrônimos, se este for o modo como ele é chamado;
- Apenas caracteres ASCII, sem acentos, espaço, pontuações ou sinais especiais;
- Não utilize o mesmo nome de um componente nativo do Flex SDK (do package `mx` como `Application`, `DataGrid`, etc) nem do Flash Player (do package `flash`, como `IOError`, `Bitmap`, etc);
- Como escrever em MXML é apenas um facilitador da escrita em ActionScript, as regras de nomenclatura do MXML são as mesmas do ActionScript. (Um novo arquivo MXML é correspondente à uma classe ActionScript, e seus componentes internos e variáveis são propriedades, por exemplo);
- O arquivo principal de uma aplicação deve-se chamar Main.mxml
    - Nunca utilize Index como nome de arquivo (para evitar conflitos com o ASDoc).

6.2. Idioma

O código do software deve estar em inglês no que corresponde à tratamentos, verbos e nomes que não sejam relativos do *domínio do negócio* (área de conhecimento específica no qual um determinado sistema será desenvolvido, ou seja, a parte do mundo real que é relevante ao desenvolvimento do sistema).

Assim, os exemplos abaixo são aceitáveis:

```
DEFAULT_CATEGORIA
addNotaFiscal()
getProdutosByCategoria()
changeState()
UsuarioVO
screens/Reports.mxml
```

Os exemplos abaixo, em contra partida, não são utilizações válidas:

```
popularCombo()
mudarEstado()
UsuarioObjetoDeTransferencia
```

6.3. Packages

O nome do pacote deve ser escrito em *lowerCamelCase*, com a inicial minúscula e as demais letras iniciais das palavras em maísculas.

O primeiro elemento do pacote será um nome de domínio de primeiro nível (com, org, mil, edu, net, gov) , ou um código de duas letras identificando um país de acordo com o ISO 3166 seguido de um domínio de primeiro nível (br.com, ar.edu, uk.gov, etc)

O próximo é a empresa responsável por aquele pacote ou o cliente daquele projeto. Os elementos subsequentes podem variar, mas como base pode ser utilizado projeto e depois módulos:

```
br.com.empresa.projeto.modulo
```

Exemplos:

```
br.com.dclick.mediaManager.uploadModule

com.apple.quicktime.v2
```

6.4. Classes

Nomes de classes devem ser preferencialmente substantivos, podendo conter respectivos adjetivos, em *UpperCamelCase*.

Exemplos:

```
class LinearGradient

class DataTipRenderer
```

6.5. Interfaces

Nomes de interfaces devem seguir a nomenclatura de classes, precedidos de uma letra “I” maíuscula.

Exemplos:

```
interface ICollectionView

interface IStroke
```

6.6. Métodos

Métodos devem iniciar com verbos, escritos em *lowerCamelCase*, ou se forem métodos disparados na ocorrência de eventos, deve ser o nome do evento seguido de `Handler`:

Exemplos:

```
makeRowsAndColumns()

getObjectsUnderPoint()

mouseDownHandler()
```

6.7. Variáveis

Variáveis devem utilizar *lowerCamelCase* e descreverem objetivamente o que elas representam. Variáveis devem iniciar com underscore (_) se estas tiverem métodos de *get* e *set* que as gerenciem.

```
private var _enabled:Boolean = true;

public function get enabled():Boolean
{
      return _enabled;
}

public function set enabled(value:Boolean):void
{
      _enabled = value;
}
```

Não há prefixos para nomes de variáveis. Além do ActionScript ter tipagem de objetos, um nome claro e conciso é mais valioso do que o tipo daquele objeto. Contudo, sugere-se que variáveis boleanas iniciem com *can*, *is* ou *has*.

```
private var isListeningForRender:Boolean = false;

private var canEditUsers:Boolean = true;

private var hasAdminPrivileges:Boolean = false;
```

Variáveis temporárias (como as utilizadas em statements de loop) podem ser de apenas um caracter, utilizando as mais comuns, como por exemplo `i`, `j`, `k`, `m`, `n`, `c`, `d`. A letra "L" minúscula não deve ser utilizada.

```
for (var i:int = 0; i < 10; i++)
```

A variável de um `catch` deve se chamar `e`, independente se o erro for de um tipo específico (tiver uma classe que o descreva).

```
catch (e:Error)
{
      hasFieldName = false;
      ...
}
```

6.8. Constantes

Constantes devem ser escritas com todas as letras maiusculas, separando as palavras por underscore (_).

```
public static const DEFAULT_MEASURED_WIDTH:Number = 160;

public static const AUTO:String = "auto";
```

6.9. Namespaces

Nomes de namespaces devem estar sempre em letras minúsculas e as palavras separadas por underline:

```
mx_internal

object_proxy
```

O arquivo que define um namespace deve ter o mesmo nome do namespace.

7. Práticas Gerais

- Use a palavra FIXME em comentários (tanto ActionScript quanto MXML) para marcar algo que não funciona e precisa ser corrigido. Use TODO para marcar algo que precisa ser feito, como uma refatoração (algo que funciona, mas pode ser melhorado) ou implementar uma funcionalidade, por exemplo. Para tal, use o *Flex Builder 2 Task Plugin*.

- Assinale o valor de iterator para uma variavel antes de utiliza-la em statements de loops caso o ganho de performance seja significativo (para arrays, por exemplo, não é necessário).

  Certo:

```
var maxPhase:int = funcaoMuitoLenta();

for (var i:Number = 0; i < maxPhase; i++)
{
      statements;
}
```

  Certo:

```
var meses:Array = ['Jan','Fev','Mar'];

// é muito rápido de calcular o length de um array
for (var i:Number = 0; i < meses.length; i++)
{
      trace(meses[i]);
}
```

  Errado:

```
for (var i:Number = 0; i < funcaoMuitoLenta(); i++)
{
      statements;
}
```

- É encorajada a criação e uso de componentes desacoplados (*loose coupled*). Quanto menos um componente souber de outro, maior a possibilidade de reuso de componentes.

- Em processos boleanos, coloque primeiro os que são mais rápidos de serem determinados.

  Certo:

```
if (isAdmin && funcaoMuitoLenta(item))
```

Errado:

```
if (funcaoMuitoLenta(item) && isAdmin)
```

- Utilize constantes quando existentes.

  Certo:

```
myButton.addEventListener(MouseEvent.CLICK, myHandler);
```

  Errado:

```
myButton.addEventListener("click", myHandler);
```

- Use tipos específicos quando possível.

  Certo:

```
private function mouseMoveHandler(event:MouseEvent):void
```

  Errado:

```
private function mouseMoveHandler(event:Event):void
```

- Para tratamento de eventos simples, com apenas um statement, prefira funções anônimas. Os dois estilos abaixo são permitidos.

```
myBytton.addEventListener(MouseEvent.CLICK,
      function(event:MouseEvent):void
      {
            simpleStatement;
      }
);

myBytton.addEventListener
(
      MouseEvent.CLICK,
      function(event:MouseEvent):void
      {
            simpleStatement;
      }
);
```

8. Apêndice: Palavras reservadas

A seguinte tabela apresenta a lista de palavras reservadas do ActionScript 3:

as	break	case	catch
class	const	continue	default
delete	do	else	extends
false	finally	for	function
if	implements	import	in
instanceof	interface	internal	is
native	new	null	package
private	protected	public	return
super	switch	this	throw
to	true	try	typeof
use	var	void	while
with			

Há também um conjunto de palavras-chave chamadas de syntatic keywords que têm significado especial apenas em alguns contextos, mas mesmo assim tais palavras-chaves não devem ser utilizadas:

each	get	set	namespace
include	dynamic	final	native
override	static		

Por fim há as palavras reservadas que podem ser utilizadas em versões futuras, e igualmente não devem ser utilizadas:

abstract	boolean	byte	cast
char	debugger	double	enum
export	float	goto	intrinsic
long	prototype	short	synchronized
throws	to	transient	type
virtual	volatile		

9. Histórico deste documento

Release 1.2
Fabio Terracini, 9/Fabruary/2007

Tradução para inglês e pequenas correções de digitação.

Versão 1.1
Fabio Terracini, 13/Dezembro/2006

Maior detalhamento sobre quebra de linhas (ítem 3.2.1). Exemplo de get e set na parte de variáveis. Exemplo de statement composto. Correções de pontuações diversas.

Versão 1
Fabio Terracini, 6/Dezembro/2006

Primeira versão do documento com os padrões de MXML e ActionScript. Foco na padronização e legibilidade. Baseado nos padrões do Java, da DClick e também no Flex 2 SDK.