Flash Project & Coding Guidelines

Project and Actionscript 2.0 guidelines at Rokkan
v1.0 – March 2007
Matt Wright
matt.wright@rokkan.com

1. Introduction

1.1 Document Overview

This document aims to establish project structure and coding standards for Adobe Flash projects written (most often) in Actionscript 2.0.

1.2 Why Guidelines Are Necessary

Coding standards are a set of guidelines, rules and regulations on how to write code. Usually a coding standard includes guide lines on how to name variables, how to indent the code, how to place parenthesis and keywords etc. The idea is to have consistent looking code across a project(s) in order to make it easier for other team members to understand what another team member has done. Even for individual programmers, and especially for beginners, it becomes very important to adhere to a standard when writing the code. Hopefully, when we look at our own code after some time, if we have followed a coding standard, it takes less time to understand or remember what we meant when we wrote some piece of code.

It must be remembered that coding standards are strongly debated and often come down to personal preferences. However, as a company, we must strive to keep our code as similar as possible in order to help others that may be on a project during development or later on during maintenance.

1.3 Philosophy & Approach

Flash developers at Rokkan are encouraged to strive for an “artisan” level of code. This means that code should be visually clean, easily understood, and follow Object Oriented Programming concepts whenever possible.

2. Actionscript Editors

2.1 Overview

It is strongly suggested that all Flash developers avoid using the Adobe Flash IDE for editing large amounts of Actionscript. The only exception is when placing simple code on the timeline such as stop actions or calls to class functions during a timeline animation sequence.

2.2. FlashDevelop

FlashDevelop is the preferred Actionscript editor at Rokkan due to its simplicity, excellent code hinting, and expandability. FlashDevelop is a .NET open source script editor designed mostly for Actionscript 2 development. FlashDevelop is very quick to setup and easy to use as an external editor for the Flash IDE or as a complete open source development environment.

Download the latest version from: http://www.flashdevelop.org

2.3. SEPY

SEPY is very similar to FlashDevelop but has proven buggy in the past and we have found the project panel to be less useful with our workflow. However, this does not mean that it is prohibited from being used.

Download the latest version from: http://www.sephiroth.it/python/sepy.php

3. Project Guidelines

3.1. Files

3.1.1. File Suffixes

- Flash file: *.fla
- Actionscript code: *.as
- XML markup: *.xml
- CSS markup: *.css

3.1.2. File Names

- Must not contain spaces, punctuations or special characters
- Actionscript
    - Classes and Interfaces use *UpperCamelCase*
    - Interfaces always start with an upper case I
        - *IUpperCamelCase*
    - Includes use *lowerCamelCase*
- XML
    - Always use *UpperCamelCase*
- CSS
    - Always use *lowerCamelCase*

3.1.3. Encoding

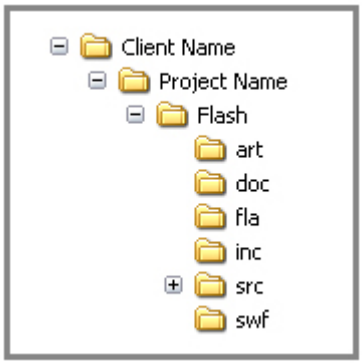
- All files must be in UTF8 format.

3.2. Version Control

Rokkan currently uses Visual Source Safe (VSS) for version control. In most instances, the project lead will setup the initial folder structure in VSS. All project members will check out appropriate files from VSS to work on your local machine. If you are unfamiliar with VSS or do not have VSS installed on your machine please consult the project lead.

3.3. Folder Structure

Folder structure for all Flash projects is very important for consistency. Rokkan currently uses the following folder structure:



art	(artwork and graphical assets)
doc	(documentation, if necessary)
fla	(all .fla files)
inc	(Actionscript includes)
src	(project classpath)
swf	(local publish location)

3.4. FLA Structure

3.4.1. Publish Settings

When publishing your .swf files, publish them directly to the swf folder (or appropriate sub folder therein) by setting the Flash (.swf) setting to ../swf/[filename].swf. While you're at it, uncheck the HTML option as it is unnecessary. Under the Flash tab, click the "Settings" button and add ../src as a classpath. Note: If you are using FlashDevelop, be sure to add ../src ass a classpath in your project file.

3.4.2. Layers

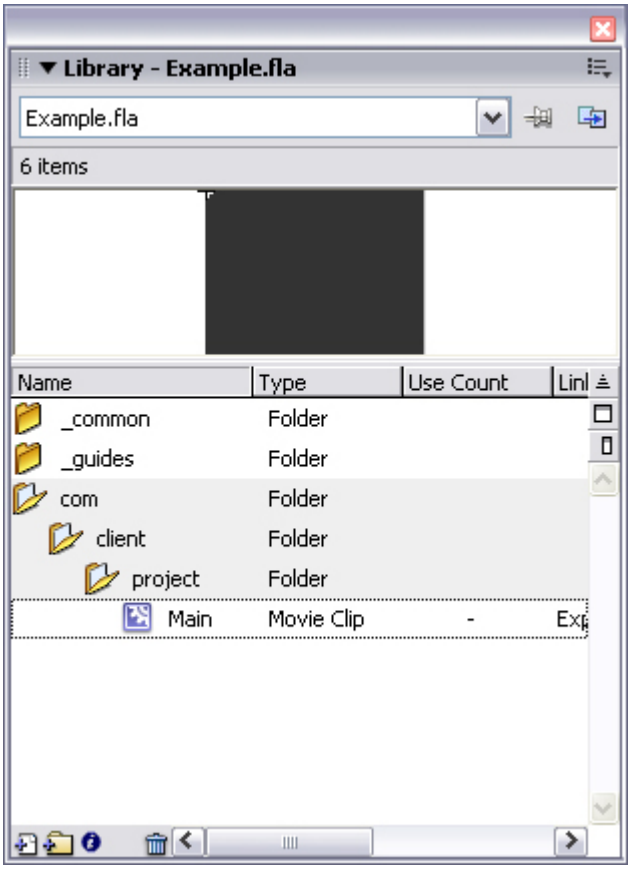
Well organized layers in your .fla files are very important. All layers in your .fla files should be named according to what type of object is on it. Each movieclip instance should be placed on its own layer and that layer should have the same name as the instance, for example: myButton_mc. All generic artwork, bitmaps, text, etc. may be grouped together in some cases, but should also reside on their own layers in order to

ease future editing. These layers should also be named appropriately by using simple descriptors. Eg: Background. When dealing with many layers within a movieclip, group layers together with folders and name the group with an appropriate descriptor.

3.4.3. Library

Well organized assets in your .fla's library is also very important. At the root of the library there should be two folders: _common and _guides. The _common folder will contain generic artwork that is reused amongst other movieclips such as generic shapes that are scaled or tinted and possibly an empty movieclip. The _guides folder will contain imported bitmaps that will serve as design/layout guides for various parts of the project.

Most often, the “main” movieclips within your .fla will have a class attached to them. With that in mind, create a folder structure within your .fla's library that looks exactly like the package folders of the class that is attached to it. For instance, if you have a movieclip in your library named “Main” and have attached the class com.rokkan.client.website.Main attached to it, the folder structure should look like:



4. Actionscript 2.0

4.1 File Organization

An Actionscript class file shall follow this structure:

#	Element	Notes
1	Initial comment	
2	Import statements	All imports should be made in alphabetical order. Use qualified imports only. Do not use an asterisk as a wild card unless importing more than 10 classes from a package.
3	Class definition	
4	Static variables	Place private variables before public variables. Variables do not need to be in alphabetical order but should be ordered and grouped according to their use.
5	Instance variables	Place private variables before public variables. Variables do not need to be in alphabetical order but should be ordered and grouped according to their use.
6	Constructor	
7	Init or Initialize method	Only if necessary
8	Event handlers	Grouped by functionality
9	Private methods	Grouped by functionality
10	Public methods	Grouped by functionality
11	Getters and setters	Ordered the same as in which they are declared

4.2. Style

4.2.1. Line and Line Wrap

When an expression extends past the normal viewable area in your editor, break it in more than one line. In these cases the line break must follow these rules:

- Break it after a comma;
- Break it before an operator;
- Prefer line break at higher level code;
- Align the new line at the beginning of the expression;
- If the previous rule does not feel appropriate, indent with two tabs.

Prefer:

```
// line #1: line break before the implements operator
// line #2: indented with two tabs
class com.rokkan.layout.Panel extends MovieClip
            implements ILiquid, IDraggable
```

Avoid:

```
class com.rokkan.layout.Panel extends MovieClip implements
      ILiquid, IDraggable
```

Prefer:

```
// line break at higher level, occurs outside the parentheses
// line break doesn't break what is inside the parentheses
variable1 = variable2 + (variable3 * variable4 - variable5)
            - variable6 / variable7;
```

Avoid:

```
// line break splits the parentheses contents in two lines
variable1 = variable2 + (variable3 * variable4
            - variable5) – variable6 / variable7;
```

Line break example with ternary operators:

```
b = (expression) ? expression
                 : gamma;

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

4.2.2. Declarations

Do only one declaration per line:

Prefer:

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

Avoid:

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

Initialize the variable if possible in the constructor or the class initialize method. There's no need to initialize the variable if its start value depends on a process to occur. Initialize the variable even if it is the default value.

Right:

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

Wrong:

```
public var isAdmin:Boolean; // undefined by default
```

Variables declarations should come on block beginning, except for variables used in loops.

```
public function getMetadata():Void
{
	var value:Number = 123; // method block beginning
	...

	if (condition)
	{
		var value:Number = 456; // if block beginning
		...
	}

	for (var i:Number = 0; i < counter; i++) // in the for
	{
		...
	}
}
```

Don't declare variables with names that were used before in another block, even if with different scope.

4.2.3. Curly Braces and Parentheses

Styling rules:

- No spaces between the method name and the opening parentheses
- One space between the parentheses and the method's arguments;
- Don't put a space between the object name and its type;
- Open curly braces in a new line at the same position in which the method declaration begins;
- Close curly braces in its own line at the same position in which the open curly brace is.
- Methods are separated by an empty line.

```
public class Example extends MovieClip
{

	private var _myVar:Object;

	public function methodName( arg:Object ):Void
	{
		_item = item;
		...
	}

	public function anotherMethod():void
	{
		while( true )
		{
			...
		}
	}
}
```

4.2.4. Statements

Simple

Simple statements must be one per line and should finish with a semicolon.

Right:

```
i++;
doSomething();
```

Wrong:

```
i++; doSomething();
```

Compound

Statements which require `switch, if, while,` etc. must follow these rules:

- The code inside the statement must be indented by one level;
- The curly brace must be in a new line after the declaration's beginning, aligned at the same position. The curly braces are closed in its own line, at the position the curly brace that opened the statement.
- Curly braces are used in all statements, even if it's only a single line.

Return

The `return` does not need to use parentheses unless it raises the understandability:

```
return;

return isActive();

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

Conditional: `if, else if, else`

```
if ( condition )
{
      simpleStatement;
}

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

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

Conditional: `switch, case`

Switch statements have the follow style:

```
switch( condition )
{
      case ABC:
            statements;
            break;
      case DEF:
            statements;
```

```
            // No break comment
        case GHI:
            statements;
            return value;
        default:
            statements;
            break;
    }
```

Break rules:

- Always use break in the default case.
- If a case doesn't have a break, add a comment where the break should be.
- It's not necessary to use a break if the statement contains a return.

Loop: `for`

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

Loop: `for..in`

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

Loop: `while`

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

Loop: `do..while`

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

Error handling: `try..catch..finally`

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

It can have the finally statement:

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

With

```
with ( object )
{
      Statements;
}
```

4.2.5. Spaces

Wrapping lines

Line breaks makes code easier to understand and analyze at a glance.

Use a line break when:

- Between functions;
- Between the method local variables and its first statement;
- Before a block;
- Before a single line comment or before a multi-line comment
- Between logical sections to make it clearer.

Blank spaces

Use a blank space to separate a keyword from its parentheses and also use a space to

separate parentheses from the operands.

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

A blank space must exist after every comma in an arguments list:

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

All binary operators (the ones with two operands: +, -, =, ==, etc) must be separated from it's operands by a space. Do not use a space to separate unary operators (++, --, etc).

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

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

Ternary operators must be separated by blank spaces and broken in more than one line if necessary:

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

The for expressions must be separated by blank spaces:

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

4.3 Comments

4.3.1. Documentation Comments

Documentation comments are for classes, interfaces, variables, and methods with one comment per element before its declaration. The documentation comment is meant to serve as a comprehensible explanation or overview of the code which follows it. This is primarily for someone that will use the class, component, or interface but doesn't necessarily have access to the source code.

Use discretion when documenting classes, functionality, variables, etc. Not every method or variable needs to be documented. Use your own judgment when writing documentation comments for code that is not publicly accessible. However, when writing an API or static methods of a utility class that will be used either internally or by a client or the public, always use thorough documentation comments and generate documentation either by using an appropriate tool. Currently, Rokkan does not have a standard for styling documentation comments and generating the documentation. However, when

necessary, an easy to use tool is as2api. Information about as2api is located here: http://www.badgers-in-foil.co.uk/projects/as2api/

4.3.2. Implementation Comments

An implementation comment has the intention to document specific code sections that are not evident. The comments must use the // format, whether they are single or multi-line. If it's going to use a whole line, it must succeed a blank line and precede the related code:

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

The comment can be in the same code line if doesn't exceed the line's maximum size:

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

Never use comment to translate code:

```
colNum = 0; // sets column numbers variables to zero
```

5. Style

5.1. General Rules

- Indent using tabs. The tab reference size is 4 spaces, and it’s suggested to configure the IDE this way
- Code lines must not exceed 100 characters

6. Naming

6.1. General Rules

- Acronyms: avoid acronyms unless the abbreviation form is more usual than its full form (like URL, HTML, etc). Project names can be acronyms if this is the way it's called;
- Only ASCII characters, no accents, spaces, punctuations or special characters;
- Do not use the names of any native Flash classes from the mx (such as Delegate, Proxy, etc) or the flash package (BitmapData, Point, etc);
- The main application class should be named Main.as or a generic descriptor of the project, such as RouteMap.as

6.2. Language

All methods, variables, class names, etc must be written in English.

6.3. Packages & Classes

Package names must be written in *lowercase* and follow the structure:

```
domain.client.project.module(s)...
```

The first element in a package name is the first level domain (com, org, mil, edu, net, gov). The next element is the company or client that owns the package followed by the project's name and subsequent modules.

In Actionscript 2, you must declare the class name in the package as well. The class name should be *UpperCamelCase.* Example:

```
com.rokkan.controls.ComboBox
```

6.4. Interfaces

Interface names must follow the same packages and classes naming rules, except all interfaces will start with an *uppercase* "I". Examples:

```
com.rokkan.controls.IComboBox
```

6.5. Methods

Methods must start with a verb and are written in *lowerCamelCase*. If the method is called on an event it should begin with handler. Examples:

```
makeRowsAndColumns();

getBookTitles();

handleRollOver();
```

6.6. Variables

Variables must use *lowerCamelCase* and obviously describe what the variable is used for. Variables must start with and underscore if they are going to be manipulated by get and set methods.

```
private var bookCollection:Array;
private var _enabled:Boolean;

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

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

There is no need for variable prefixes or suffixes. Typing your variables and having a clear and concise name is sufficient. Although, Boolean variables should start with can, is or has. Examples:

```
public var isActivated:Boolean ;
public var hasAdminPrivileges:Boolean ;
```

Temporary variables used in loop statements should be only one character. The most commonly used are i, j, k, m, n, c, d. The lowercase “L” must not be used.

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

Tempory variables used within a method’s scope should be lowerCamelCase and obviously describe what the variable is used for. Example:

```
private function handleRollOver():Void
{
      var rollOverClip:MovieClip = getRollOverClip();
      rollOverClip._alpha = 50;
}
```

Catch variables must be name e, no matter what the error type

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

6.7. Constants

Constants must be all *uppercase*, splitting the words with an underscore:

```
public static var CLICK:String = “click”;
public static var WITHIN_BOUNDS:String = “withinBounds";
```

7. General Practices

- Create temporary variables for use within iterators. It is widely accepted that this, albeit very minor, improves performance.

```
var maxAmt:Number = reallySlowMethod();

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

- All developers should be encouraged to make use of Object Oriented Programming concepts at all times. This also means that developers are encouraged to create loosely coupled classes and components. This promotes easier reuse of code.

- All developers should be encouraged to use Event objects rather than generic objects as handlers for events. This is so that event objects may carry specifically typed variables that IDE's such as FlashDevelop can easily code hint when accessing the objects properties. These Event classes should also contain string constants for event types. This technique is similar to that of Actionscript 3's event framework.

- Interface elements, mostly commonly buttons, should not use anonymous functions for event handling. This is because buttons often have many event handlers that mean nothing to the class which they are being defined from. This simply clutters a class and does not promote code reuse. Always create a class for specific interface elements such as buttons so that event handlers (particularly onRollOver, onRollOut, etc) are defined within that object.

  Avoid:

```
myButton.onRollOver = function()
{
      this._alpha = 100;
}
```

8. Document History

Release 1
Matt Wright, March 2nd, 2007

This is the first release of the Rokkan Flash and Actionscript Guidelines document. It focuses mainly on standardizations of style and readability for code and individual project management.