RICH MEDIA DEVELOPER CODE COMMENT STANDARD

by Mark Martino
2011 JAN 10

CODE FOR HUMANS

A computer doesn't care. It will execute your code with as much concern as it has for an empty infinite loop. While it's fun to anthropomorphize computers, it's a bad idea for programmers to write comments with that model in their heads. On the flip side, this lack of concern allows computers to execute code even if there are no comments and even if the names of variables, functions, methods, and classes are indecipherable.

So, why bother with comments and understandable names? Because code gets modified and comments help us keep track of what we want the code to do. Comments are code that is read and processed by humans. While humans can work around spelling and syntax errors better than computers, don't rely on it. The humans who read your comment may be new to the code or the language. They may be under time pressure, sleep deprived or having a bad day. Your comments must be understood despite these conditions.
The computer cannot and does not care what you intended. Only you and the people you work with care. We write comments for ourselves, not those ungrateful machines.

A LITTLE CONTEXT

To write useful comments, we rely on context. A symbol's meaning depends on the context in which it is used. Programmers use this principle by creating logically nested contexts. This allows us to understand a great deal with a minimum of syntax and symbols. The commenting techniques described below are a guide to writing comments that, along with readable names, create context.

In the context of this article, the term "module" means any code module such as: an application or other computer program; a class; an object; or a function or function set.

WRITING STYLE FOR COMMENTS

There's plenty of complexity in code. Don't add to it by writing complex comments. Use simple, declarative sentences. Write complete sentences starting with a capital letter and ending with a period. You should know what an item is and does, so you shouldn't need question marks. Comments should be boring, so you don't need exclamation points.

Use as few modifiers, i.e., adjectives and adverbs, as you can to describe an item. Use the modifiers consistently.

Articles matter a lot more than you think. When a new item is introduced, use "a" or "an." When referring to an already introduced item, use "the."

Use pronouns carefully. Be sure that the reader can determine exactly what item a pronoun refers to. If it's not clear, use the name of the item.

Select one name for each item. Always use the same name for the same item. If you later

invent a better name for an item, do a search and replace and change the name everywhere the item is used.

You will be tempted to use code or psuedocode as a shortcut. You will crack yourself up with funny names and puns. Write them, delete them and replace them with plain, direct boring English. The more boring the better. Boring means that the above guidelines have been followed. It also means that changes will be easy to make and unusual items will be easy to find.

VISUAL STYLE FOR COMMENTS

Humans developed aesthetics as a survival mechanism. Studies have shown that beautiful things are literally easier on the eyes. It takes less visual processing to perceive a beautiful item. Hence, visual style is vital, not extra. And, if you don't care about how something looks, programming graphics and animation may not be your niche.

Avoid the "stars and bars," style of commenting. It's an ugly relic of a time before colored type. Use only /* */ or /** */ for large comment blocks and // for one to five line comments. Though many editors and templates still insert asterisks in front of each line, avoid that if you can. Avoid using upper case for emphasis. Leaner and cleaner is easier to read and maintain.

Put a space between // and the comment so you know it's a comment and not commented out code. Whenever possible, use only // in the body of the module so you can easily comment blocks of code out with /* */.

Stick with the default text color for comments in the editor your using, e.g., light green or blue.

COMMENT STRUCTURE

There are two types of comments: line comments and block comments. A line comment occupies one line. A block comment starts with "// " and occupies more than one line. Notice that there is a space after the slashes. A block comment can be built from one to five line comments or six or more lines enclosed by /* */ or, for doc tools, e.g. asdoc, jsdoc, /** */.

For a file containing a module, the outermost context is provided by the header comment, i.e., the block comment at the top of the file. As much as possible, all other comments in the file should be line comments, i.e., comments that begin with "// ".

COMMENT CONTENT

Comments should concisely provide the following information:
- the purpose of the module
- the modules that use the module or the types of modules that may use it
- the modules the module uses or the types of modules that the module uses
- describe dependencies, especially unusual ones and why they are necessary
- features or aspects that are not standard practice and why they are necessary

If the module in part of a library, make sure it is clear what library it is part of and how it used with other parts of library. If the module is a collection of functions, e.g., JavaScript functions, describe what types of functions are in the module, what types of functions should not be in the module, and how the functions work together.

All of the above items should be described in relation to what the module does in the real world. For example, if a module causes a panel to appear on a screen, describe that. If the module changes data in a database, describe that.

If the module happens to be the topmost module, e.g., the application's class, the comment would provide a top level description of the application and its purpose. The modules that it uses would be mentioned with little or no description. The details about those modules should be written in the header comments in their respective files.

Write the header comment as you design the module. Write and rewrite the header comment until you are satisfied that you have precisely described the module and its parts. If, when writing a header comment, you find that the module is using more than, say, a half dozen other modules, you may want to rethink your design.

COMMENTING INTO A DESIGN

A big difference between object oriented programming and other kinds of programming is that many design decisions are made up front and refined instead of deciding as you code. Writing comments first is a great way to do this and when the code is complete, so are the comments.

The following process is described as though you were writing a new module. You can use a reduced version of the process to make changes to existing modules. Ideally, at the end of the process, the resulting code has no more complexity than it requires. By complexity we mean many items interacting in many ways. Whether or not you're comfortable with all those items and intimately understand them, they are still complex. Hence, it may be difficult for others to understand them, or hard for you to understand if you forget what you now know about them.

The strategy of this process is to work on a design in a way that allows you to pay attention to only a few parts at a time interacting in a few ways. It's worth taking the time to step through this process. It takes less time than you think. It saves time over the long haul and reduces stress. Anxiety leads to impatience. Impatience leads to lack of planning and poor execution which leads to failure which leads to anxiety.

Write A Problem Description

Write the problem description in the code file or in an external text file. You'll be writing lots of notes and changing them a lot so be sure the file you use makes this easy to do.

This is where you want reason to steer intuition. Intuition wants you to move quickly to the first description that comes into your head and race ahead to a solution. Don't fall for it. Even if it's a problem you think you've seen hundreds of times, the technology may have

changed and you might be wrong. If you're doing interesting work, this will happen a lot.

Instead, write a description of the problem you're trying to solve. If you have written requirements, paraphrase the most important requirement into a description of the problem. Don't try to get it right on the first attempt. You're typing on a computer. It's easy to change. It 's okay to follow your intuition about what to write first. Start with what you think you know for sure and build on it. Apply reason to rewrite and rearrange it later.

This doesn't take as much time as you think. Most of the thinking you'll be doing will be thinking you'd need to do even if you didn't write it down. If you're doing interesting work, you'll be learning something new each time you do this. If you think you're spending too much time on the problem description, ask yourself how much time it would take to solve the problem if you didn't learn anything new.

Write A Model of a Solution
Copy the problem description in your notes and reform it into a model of a solution. Don't worry yet about how to code the solution. Describe what the solution needs to do. For example, if the problem description says the user needs to access certain data on the server, the model might include a button, a link or something new.

For an existing module, think about what can be removed or combined or otherwise simplified to solve the problem before adding new parts to the module.

For larger problems, e.g., designing a new application or tool, develop a model for a one button application. Except for games, users want an application to do a job for them and do it with as little thought and interaction from them as possible. Start describing your model by assuming that you can write such magic code. On your next pass, break out the parts that may need to be written to make the magic happen.

Another way to think about your model is to ask these questions:
How do you stop it? How do you start it? How do you find out what's going on?

Write About A Solution Using Named Items
Go through the model description and pick out items that need names. Such an item may at first have different names in your model. Do a search and replace to give the item a consistent name. This often helps clarify a solution because names and naming scheme affects how you think about the problem, the objects, the variables and the functions.

Developing an accurate name for an item is critical because the name tells us the nature and purpose of the item. Unlike almost any other kind of engineering or human activity, whatever you name a software entity tends to turn it into what you name it. This is great, if you named it what you intended it to be.

Use nouns for objects and verbs for functions and methods. Modify them with one adjective or adverb, at most two. A very useful type of noun is the agent noun. A agent noun is formed by appending the suffix "er" to a verb. For example, the verb "interpret"

becomes the noun "interpreter." Hence, an agent noun denotes an agent who does the action denoted by the verb from which the noun is derived.

When selecting names, avoid using the terms: system, manager and global. If you think you need these terms, do a web search on object oriented design patterns. You'll notice that there are no Manager, System, or Global patterns.

Everything in engineering is a system. Calling an item an system provides no new information.

In most cases of object oriented design, calling an item a manager implies that the objects cannot manage themselves. In object oriented and distributed design, they should. If you're sure you need a managing object, ask what main action it does to manage items and derive a name from that action.

Global is the worst of information free terms. Global is to a computer program what the World Series is to the world. Certain items may be widely accessible within a certain large but well-defined context. If for no other reason than security, items should never be accessible to the whole world. Hence, calling an item global is always a misrepresentation.

Keep rewriting the solution until you have a description of each item that explains what it is, what it does and how it works with the other items in the solution.

Use the Comments to Write the Code

When you have a workable solution, organize, condense and refine your notes to generate comments. Copy them into the header comment for the file for the topmost class or object. Split out the notes as you write each file for the classes, objects or function sets. Move the notes from the topmost file into the files for the other classes, objects or function sets. Write the comments using the guidelines described in the previous sections. In each file, write code that expresses the comments. Make sure the names of variables and functions and methods match the terms you use in the header.

Adapt Comments As You Correct Code Flaws

If you find that your original solution has flaws, change the comments and code together. Two types of flaws to look for and correct are premature optimization and generalization.

Donald Knuth said that premature optimization is the root of all evil and it's still true. To avoid it, don't project too much about what might be needed later. A simple code module is the best way to prepare for the future. Don't put in anything you don't need for the problem you're solving now. Adding code solely to save keystrokes is usually not worth it. Use getters and setters only if you will be using them to solve the current problem. Don't include excessive debugging code. Don't write buggy code in the first place.

Premature generalization is the second root of all evil. It's tempting to imagine how many classes could benefit from your brilliant code. You're probably wrong. If the code does need to be generalized, it will become clear as it gets used. You can abstract it out then

with more knowledge of what's needed. If you generalize too much too early and you're wrong, it will be very hard to undo.

Avoid passing data around. In object oriented design, the object that owns the data does the work. Data usually doesn't need to be accessed unless an action is performed using the data. Ask, what is that action and how can it be done within the object that encapsulates the data. Again, use getters and setters only if there's no other way to solve the current problem and only the ones you need to solve the problem.

Design code so that it can be easily changed rather than trying to predict everything that might happen and trying to code it so you never have to change it. A part of this strategy is to code only what you need now. Do or don't do, but mostly, don't do. Wherever possible, design items out, not in.

CHANGING COMMENTS

When code changes, make sure the comments that refer to the code match. If they don't, determine if the code change is what was intended. If so, change the comment; otherwise, change the code. If more than 20% of the comments need to be changed, it is likely that the purpose of the code has radically changed or wasn't understood in the first place.

COMMENTS THAT AREN'T COMMENTS

Remove all commented out code. If you must leave it in, include your name or initials, the date and an explanation of why you are leaving it in. Uncomment and include or remove the code as soon as possible.

THE WORST COMMENT

Certain types of comments are worse than no comments. The absolute worst comment looks like this:

```
/****** THIS CODE IS A KLUDGE AND SHOULD BE FIXED. *******/
```

Besides the excessive use of upper case characters and asterisks, the message is vague and useless. We can suppose that the next line of code is part of the problem, but we don't know how many lines are part of the problem. The message doesn't tell us what the problem is, who to consult for more information or how long the situation has existed.

This is not a made up example. Comments like this have brought entire companies to a halt while programmers struggled to discover, what, if any problem exists.

It is also an example of a judgmental comment, i.e., a comment that uses language describing the quality of the code. There is no place for such language in a comment. The code is what it is. In your comments, don't apologize for your code or judge other people's code. Either fix it or leave it. If there is a problem you cannot fix in the time you have, insert a comment that starts with your name or initials and the date and then describe the problem, not the quality of the code.

A COMMENTING EXAMPLE

Below is an example of the commenting described above. It is the contents of an ActionScript file for a class that, as of this writing, is used in production. In this copy, the comments are bolded.

```
/**
An instance of this class provides the GTP with the ability to collect
and transmit trend tracking information for a tutorial. One instance of
this class should be created for an instance of the GTP. In
ActionScript 2.0, that instance is available as _root.wtTracker.

The data is transmitted via one of two Javascript methods. If the tutorial
is contained directly by an HTML page, dcsMultiTrack() is used. If the
tutorial is contained in an iframe, iframeDcsMultiTrack() is used. Both
methods are invoked via ExternalInterface. The first argument is the name
of either Javascript method. The remaining arguments are key-value pairs
of the trend tracking parameter names and their values, e.g., wtPN, "Tutorial Name".
*/
import flash.external.ExternalInterface;
import mx.controls.Alert;

class GTPWebTrendsTracker
{
	private var dcsParams:Array;

	private var flowCode:String;
	private var linkName:String;
	private var tutorialName:String;
	private var mediaID:String;
	private var slideCount:String;

	private var maxSlideViewed:Number;

	private var exitSlide:String;
	private var successFlag:String;
	private var statusMsg:String;

	private var userAnswer:String;

	public function GTPWebTrendsTracker()
	{
		// Set the default values for specific key-value pairs.
		// The key-value pairs for these parameters are set prior to sending.
		tutorialName = 'Global Tutorial';
		flowCode = 'NONE';
		linkName = 'Flash Tutorial';
		mediaID = '000';
```

```
        slideCount = '2';
        maxSlideViewed = 1;
        exitSlide = '1';
        successFlag = '1';
        statusMsg = 'SUCCESS';
        userAnswer = 'Did not answer';
}

// Call this function when tutorial is about to close.
public function gtpSendOnTutClose():Void
{
        completeParameters();
        gtpSend();
}

public function gtpFlowCode(fc:String):Void
{
        flowCode = fc;
}

public function gtpMediaID(mID:String):Void
{
        mediaID = mID;
}

public function gtpTutorialName(tutName:String):Void
{
        tutorialName = tutName;
}

public function gtpSlideCount(sc:String):Void
{
        slideCount = sc;
}

public function gtpMaxSlideViewed(msv:Number):Void
{
        if(msv > maxSlideViewed) maxSlideViewed = msv;
}

public function gtpExitSlide(es:String):Void
{
        exitSlide = es;
}

public function gtpUserFeedback(ua:String):Void
```

```
{
	userAnswer = ua;
}

public function gtpTrack(paramKey:String, paramValue:String):Void
{
	// Insert parameter value into the list of parameter values.
	dcsParams.push(paramKey);
	dcsParams.push(paramValue);
}

// This function sends the tracking parameters.
private function gtpSend():Void
{
	if(ExternalInterface.available)
	{
		if(isInsideIFrame())
		{
			// If the containing entity is an iframe, send the parameters
			// to a function that passes the parameters to the iframe's
			// container.
			dcsParams[0] = "iframeDcsMultiTrack";
		}
		ExternalInterface.call.apply(null, dcsParams);
	}
	else
	{
		trace("GTPWebTrendsTracker: ExternalInterface is not available.");
	}
}

private function isInsideIFrame():Boolean
{
	// If this SWF is run from within an iframe, return true.
	var isIn:Object = ExternalInterface.call("isInAnIFrame",null);
	if(isIn == "true") return true;
	return false;
}

private function retrieveDcsURI():String
{
	// Retrieve the URI, i.e., the stem of the location of the window
	// containing the SWF.
	var dcsLoc:Object = ExternalInterface.call("getDcsLocation",null);
	if(dcsLoc == null) return "/esupport/article.jsp";
	var pathname:String = dcsLoc.pathname;
```

```
        var startIndex = pathname.indexOf("/urf");
        return pathname.slice(startIndex + 1);
}

private function retrieveDcsRef():String
{
        // Retrieve the reference of the window containing the SWF.
        var dcsref:Object = ExternalInterface.call("getDcsref",null);
        if(dcsref == null) return "dcsref unavailable.";
        return dcsref.toString();
}

private function retrieveLinkName():String
{
        // Retrieve the text of the link clicked to invoke the tutorial.
        var linkName:Object = ExternalInterface.call("getLinkText",null);
        if(linkName == null) return "Flash Tutorial";
        return linkName.toString();
}

private function completeParameters():Void
{
        // Set the Javascript function name.
        dcsParams = ['dcsMultiTrack'];

        // Insert the base tags required for WebTrends tracking.
        dcsParams.push('DCS.dcssip');
        dcsParams.push('www.att.com');

        dcsParams.push('DCS.dcsuri');
        dcsParams.push(retrieveDcsURI());

        dcsParams.push('DCS.dcsref');
        dcsParams.push(retrieveDcsRef());

        // Complete the collection of trend tracking parameters.
        dcsParams.push('DCSext.wtPN');
        dcsParams.push('Global Tutorial Player');

        if(flowCode != 'NONE')
        {
                dcsParams.push('DCSext.wtFlowCode');
                dcsParams.push(flowCode);
        }

        dcsParams.push('DCSext.wtNoHit');
```

```
                dcsParams.push('1');

                dcsParams.push('DCSext.wtLinkLoc');
                dcsParams.push('FT');

                dcsParams.push('DCSext.wtLinkName');
                dcsParams.push(retrieveLinkName());

                dcsParams.push('DCSext.wtMediaId');
                dcsParams.push(mediaID);

                dcsParams.push('DCSext.wtTutorialName');
                dcsParams.push(tutorialName);

                dcsParams.push('DCSext.wtEvent');
                dcsParams.push('Global Tutorial Requested');

                dcsParams.push('DCSext.wtUserResponse');
                dcsParams.push('Global Tutorial Requested');

                dcsParams.push('DCSext.wtSlideCount');
                dcsParams.push(slideCount);

                dcsParams.push('DCSext.wtMaxSlideViewed');
                dcsParams.push(maxSlideViewed.toString());

                dcsParams.push('DCSext.wtExitSlide');
                dcsParams.push(exitSlide);

                dcsParams.push('DCSext.wtSuccessFlag');
                dcsParams.push(successFlag);

                dcsParams.push('DCSext.wtStatusMsg');
                dcsParams.push(statusMsg);

                dcsParams.push('DCSext.wtUserResponse');
                dcsParams.push(userAnswer);
        }
}
```