If You Have Programming Standards, Please Raise Your Hand: An Everyman's Guide

Dianne Louise Rhodes, Connect International, Inc., Washington DC

ABSTRACT

When a new project starts, or a new manager takes over an old project, the programming team is faced with a new culture. Often, this means that they are asked to adhere to yet another set of programming standards, use style sheets, and are subject to peer reviews. This process is usually short lived, because of the lack of a practical approach to applying standards and enforcement and failure to budget time and resources in the project schedule. This paper goes through a step by step process of developing programming standards, classifying them and entering them into a database. This database can then be used to develop style sheets and check lists for peer review and testing. Through peer reviews and in preparation for them, programmers learn good programming practices. We describe in detail the most common standards; why and how they should be applied.

INTRODUCTION

In researching this paper, I found a number of sources of programming standards, some specific to SAS® and others more general (see References). There is agreement that standards are a good thing, but rarely did I find mention of a practical approach to the use and enforcement of standards. No one wants to police programmers; that has been compared to "herding cats." Given explicit conventions to follow, programmers can easily review each other's work and apply these standards. When the corporate culture encourages peer reviews by allocating time to this activity it is an opportunity for professional growth. The process for developing explicit standards and checklists for peer reviews are detailed in this paper.

THE LOGIC BEHIND STANDARDS AND STYLE SHEETS

In a maintenance environment, a programmer should be able to make changes to a program without fully comprehending the entire process. If the original programmer has followed standards, which always include comments, it will be easier to understand the code (Lavery, 2009).
The majority of standards are rules that make the code easier to read on the page, easier to follow the logic and logical branches, and leave less room for interpretation. Code that meets these requirements is easier to maintain. Standards are useful working in a teamwork development environment because they set minimum requirements, which in turn insure some uniformity from programmer to programmer. They are imperative in managing a large project, where source code control is also an issue. (See Mitchell, 2005).
Standards for the sake of having standards are a good idea as well. Aster (1999) says: "A good style is simple, clear, and consistent... The main point, however, is that just about any style that you follow consistently is better than no style. Even a bizarrely idiosyncratic coding style can eventually start to make sense to the reader. But if your code has no style, a maintenance programmer can never quite figure it out without reading every detail."

If you are the only one who will be reading your code, you may be asking yourself, "Why bother?" As Lavery (2009) points out in his Animated Guide, the goal is to produce code that is eary to understand, maintain, and produces high quality, low cost outputs. One of Murphy's laws no doubt is that you will forget why you did what you did, and when you or someone else revisits the code you will be able to reproduce your work (Axelrod, 2009).

JUSTIFICATION FOR PEER REVIEWS

Peer reviews and formal walkthroughs are a valuable exercise in quality control. These processes help to ensure that code complies with programming standards, meets specifications, and is error-free. They are critical in managing a large project to ensure uniformity in programming style and use of variable names (Mitchell, 2005) Often this process breaks down as a project progresses and deadlines loom large. This is precisely the situation that benefits the most by having a second or third pair of eyes examining your work. A good Peer Review Process demands a corporate culture that adopts it and embraces it.

FACILITATING PEER REVIEWS AND WALKTHROUGHS

An informal walkthrough requires preparation time on the part of the reviewer, the programmer, and others critical to the review process. The “others” will vary as different corporate cultures may include managers, for example, while others specifically exclude them. Once time is allocated, a helpful tool is to have a checklist. This gives the reviewer an objective set of criterion to apply against the program or code being reviewed. The checklists will vary depending on the lifecycle stage of the programming, e.g., analysis, design, implementation, testing, maintenance.

THE DILEMMA

Despite the promises of a “paperless office” made in the ‘80’s, I have accumulated a variety of standards, style sheets, testing procedures, hints and tips. Mostly these are a rag-tag bunch of paper documents, with something of value lurking in their depths. Some were developed for a particular project and are therefore very specific; others are generic “wish lists” of programmer behavior. The dilemma – how to translate all these piles of paper into “useable” standards?

RULES OF DECLUTTERING

I approached this problem using some of the tricks I have learned to “declutter” my living and working spaces. The **first rule**: get everything out in the open. The **second rule:** group similar items together. Eliminate duplicates. The **third rule**: find homes for everything. And the **fourth rule**: throw out or give away anything you haven’t used in over a year. That might be harder to do with standards than with old clothes, but the idea is the same – if you aren’t using it, it’s probably obsolete.

APPLYING THE RULES TO STANDARDS

The **first task** was to locate copies of all the standards, rules, and tips I wanted to include. This meant I had to dig through files and piles of papers to find all the goodies I wanted to include in this project. I also printed out various SUGI and NESUG papers appropriate to the subject (see References). The **second task**: put like objects together, i.e. Classify them. I initially defined four groups of rules: documentation, efficiency, maintenance, and testing (with some overlap). The **third task**: put them into a database. I elected to use MS Access® because it was quick and easy and I could give the data entry work to someone who wasn’t familiar with SAS. SAS was used to develop more complex reports and checklists. I also tried to operationalize the rules, that is, give working examples. The **fourth task**: I eliminated some rules that were vague or subjective such as “avoid unnecessary branches.”

CORPORATE STANDARDS

Does your company have corporate standards? When I asked this question of my colleagues, the answer was no. Did I want to be the one to try to create corporate standards? No. For this reason, I decided to keep most of my styles and tips pretty general, instead of implementing specific rules. For example, I suggest that it is useful to indent code to show logical flow. But I don’t specify a style, since I don’t really care what style is used as long as it is used consistently. I once worked on a project that dictated you could not nest IF THEN ELSE statements more than five deep without review. Ask yourself, do you want to be the IF THEN ELSE police?

THE TIPS DATABASE

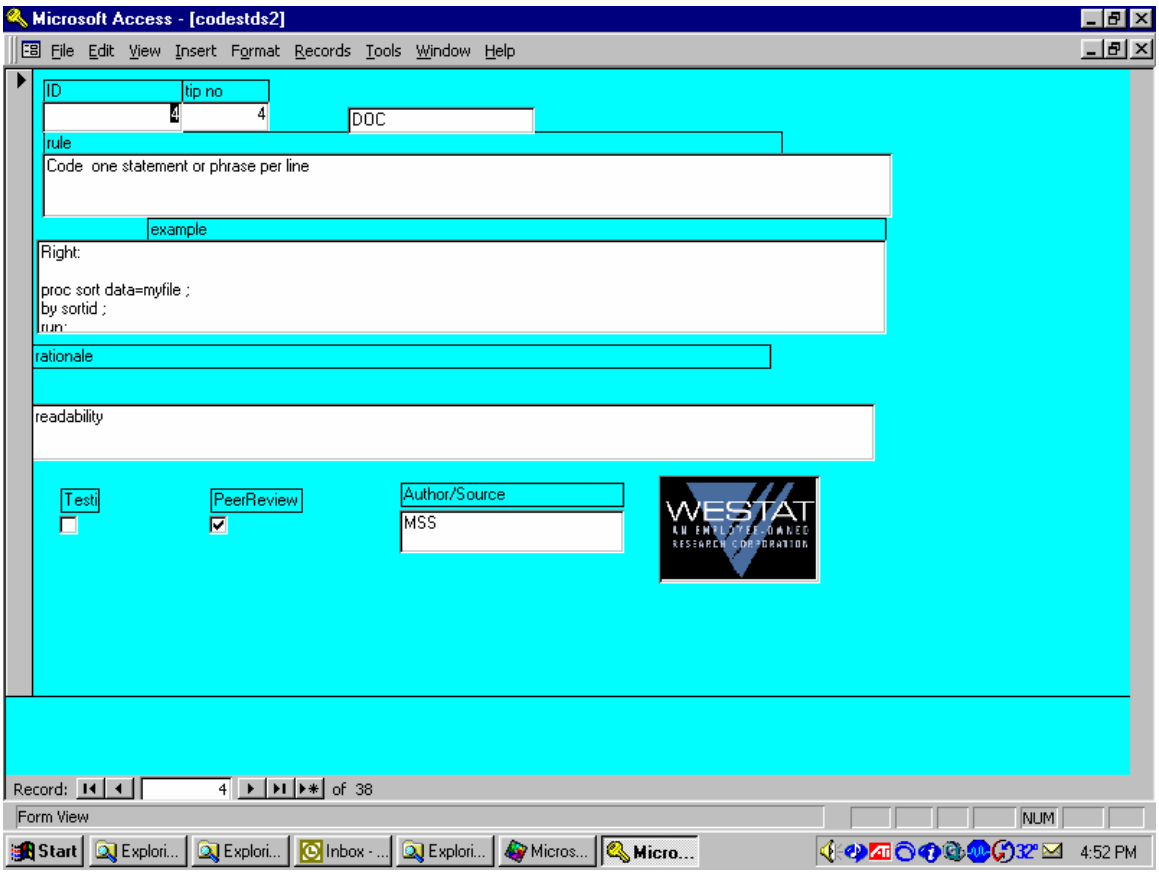
Working from various paper documents, I roughed out a database. The fields were tip number, type, the rule, an example, and the rationale for the rule. I quickly added check boxes for Peer Review and Testing, and a field to store the author or source. This allows for the generation of various checklists for peer reviews, testing procedures, etc. and customization by selecting certain subgroups. The Appendix contains examples of a document to present tips and standards, checklists used for peer reviews and testing, and a report that can be used as a style sheet.

The types that I defined were:
Documentation. Tips or standards that help document the purpose of the code and make it easier to read.

Efficiency. Tips to make the code more efficient; that is to run faster.
Maintenance. Tips to make the code easier to maintain. These sometimes are contrary to the efficiency tips!
Testing. These tips mainly concerned testing issues, or things to look for when testing. This is an area I would like to continue to enhance.

FORMS

The main form was a simple data entry form. Here you transcribe from a paper cheat sheet the key elements of the tip, trick, or standard.



REPORTS

The first report is basically a hard copy of the database, as shown in Programming Tips and Tricks in the Appendix. These are listed by tip number, which is the arbitrary order in which they were entered into the database. I included another tip number field in the database, so that the order can be changed.

The Style Sheet is a list of the tips, primarily those related to documentation.
The Peer Review checklist was produced from a query that selected the tips where the Peer Review check box was checked. It is a quick list for programmers to review when they are desk checking a program.
The Testing checklist was produced from a query where the Test check box was checked. It is a list for review when testing a program, which could be stress testing, parallel (regression) testing before putting a program into production, etc.

STANDARDS IN DETAIL

THE HEADER OR DOCBLOCK

Every set of standards for SAS starts with a documentation block at the top of the program. Here's a minimal version (adapted from Axelrod, 2009):

```
/*************************************************************/
/*   program:    mp003.sas                                    */
/*                                                            */
/*                                                            */
/*   path:           h:\projects\my_projects                  */
/*                                                           */
/*   author:        D. Rhodes                                 */
/*  Date:           March 2009                               */
/*   input           2010  project data file – ASCII          */
/*                                                           */
/*   output:        project_data.sas7bdat                    */
/*   purpose:   Read in raw data and save as a SAS file      */
/*   usage  :   %mymacro(dsn=mydata, version=reltest ;       */
/*************************************************************/
```

When the program is modified, add a note. For macros, it is a good idea to include usage, eg. To call this macro you need to specify dataset name and version

```
%mymacro(dsn=mydata, version=reltest) ;
```

The main drawback is laziness. For example, an analyst copies a program to use as a starting point, but fails to update the doc block.
Rick Mitchell gives a cute example of how a job security specialist would use the program header:

```
Program name:      Rick57.ABC
Owned by :         Rick
Date Written:      2002
Purpose:           To be provided on a need to know basis
(Mitchell, 2005)
```

At Freddie Mac, I saw a version like

```
Program name:       roll_over.sas
Author:             C12349
Date Written:       March 2, 2010
Purpose:            roll over to the next year
```

Not really very helpful, is it? Neither of these examples provide you with additional information beyond the name of the program. In the FM example, the author is given as their employee id, which couldn't be easily traced.

NO ERRORS OR WARNINGS

This should be obvious but it's often skipped over. "It's just a warning, it doesn't affect the outcome." If you're smart enough to figure out that it does not effect the results, you are smart enough to fix it. (A pet peeve of mine).

NO NOTES FOR CONVERSION, UNINITIALIZED

These particular notes do not necessarily cause errors, but they can be confusing, especially when encountered by a new programmer working in legacy code. Use MSGLEVEL = I for additional information notes.

Notes that Character values have been converted to numeric (and vice versa) can be avoided by using put or input statements.

Do not allow NOTE: Variable XYZ is un-initialized. This could mean that a variable you are expecting is not there, or you spelled it incorrectly (Lavery, 2009). Always initialize variables, preferably to a special missing

value so you can tell when you've set to missing intentionally, and when it has been set to missing otherwise. I worked on a project at BLS where one programmer would use length statements to put variables on the PDV and then never set a value. It drove me nuts.

READABILITY

Liberal use of white space as indentation, leave a blank line between steps. Write one statement per line. Keep your text in columns 1-72. If code goes beyond column 150 the reader has to scroll back and forth. On some mainframe systems, code beyond cc 72 will be "lost" or not interpreted.

NAMING CONVENTIONS

These rules apply to variable names, names of datasets, and names of programs (Mitchell, 2005). Names should not be reused. Use a meaningful name. I is okay for an index, but I1 is just going to be difficult. At one job, I inherited a program named Final2. Shouldn't that be a clue that the name final is meaningless? But I couldn't change the name it had been institutionalized by management. If you are defining an array of variables, use a leading zero eg. Var01-Var10 so that the sort sequence will put them in logical order when used in reports or contents.

UNAMBIGUOUS MERGING

Use MSGLEVEL = I for additional information notes. This will tell you if the same variable is on both sides of a merge. The only common variables should be the merge by variables. You should never allow this situation as it can propagate missing values. Always use a BY statement in a merge. Some work sites may set NOMERGEBY to trip an error. The default for NOMERGEBY is a NOTE. If you don't need a BY statement, maybe it isn't a merge. Do not allow the message NOTE: MERGE statement has more than one data set with repeats of BY values. This can propagate missing values and have an effect similar to a Cartesian join.

USE SAS FUNCTIONS OR PROCEDURES INSTEAD OF ROLLING YOUR OWN

SAS® functions have been rigorously tested and optimized. They will be faster, easier to understand, and more accurate. Be sure to check out the "any" and "not" functions (introduced in version 9) for data cleaning.

MINIMIZE HARD CODING

Use a macro variable at the beginning of your program to set a value. This makes the code easier to reuse (Axelrod, 2009). For example, for a time period of quarter use %let qtr = 4 or for quarter and year, %let qyr = 20104. If there are a number of parameters, maybe you should wrap your code in a macro.

ENHANCEMENTS

The major enhancement I plan to make is to create another table and input form to allow programmers to interact with the Peer Review and Test checklists while desk checking a program. This would allow them to check off that a program is in compliance or needs work in a particular area.

CONCLUSION

Organizing standards and style sheets can be an onerous task, but it's a necessary requisite to implementing their use. Putting them all into one place, cleaning them up, and having them where they are accessible to the programming staff is part of the process of learning good programming practices.
DISCLAIMER: The contents of this paper are the work of the author(s) and do not necessarily represent the opinions, recommendations, or practices of Connect International, Inc. or the Bureau of the Census.

REFERENCES

Aster, Rick (1999).“Coding for Posterity.” In the 11th Annual Proceedings of the NorthEast SAS Users Group. (NESUG).
http://www.nesug.org/Proceedings/nesug98/appl/p131.pdf

Axelrod, Elizabeth (2009). “Boot Camp for Programmers: Stuff you need to know that’s not in the manual.” Proceedings of SAS Global Forum (SGF) 2009.

Lavery, Russ (2009) “An Animated Guide to Coding Standards for SAS Production Programs. “ Proceedings, NESUG , 2009.

Mitchell, Rick M. (2005). “Fast and Easy Ways to Annoy the Job Security Specialist.” Proceedings, SAS User Group International (SUGI) 30.

ACKNOWLEDGMENTS

I want to thank my colleagues at the Census Bureau for their support and timely feedback.

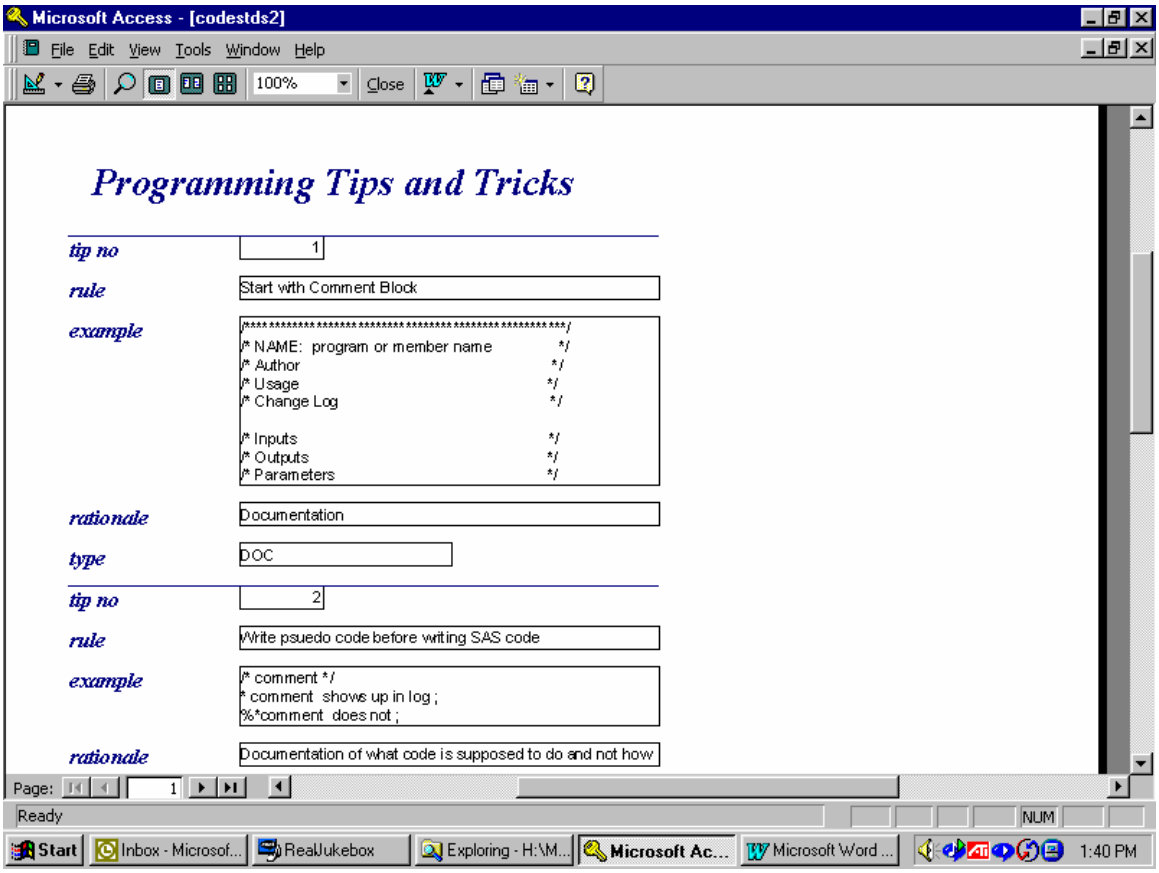
CONTACT INFORMATION

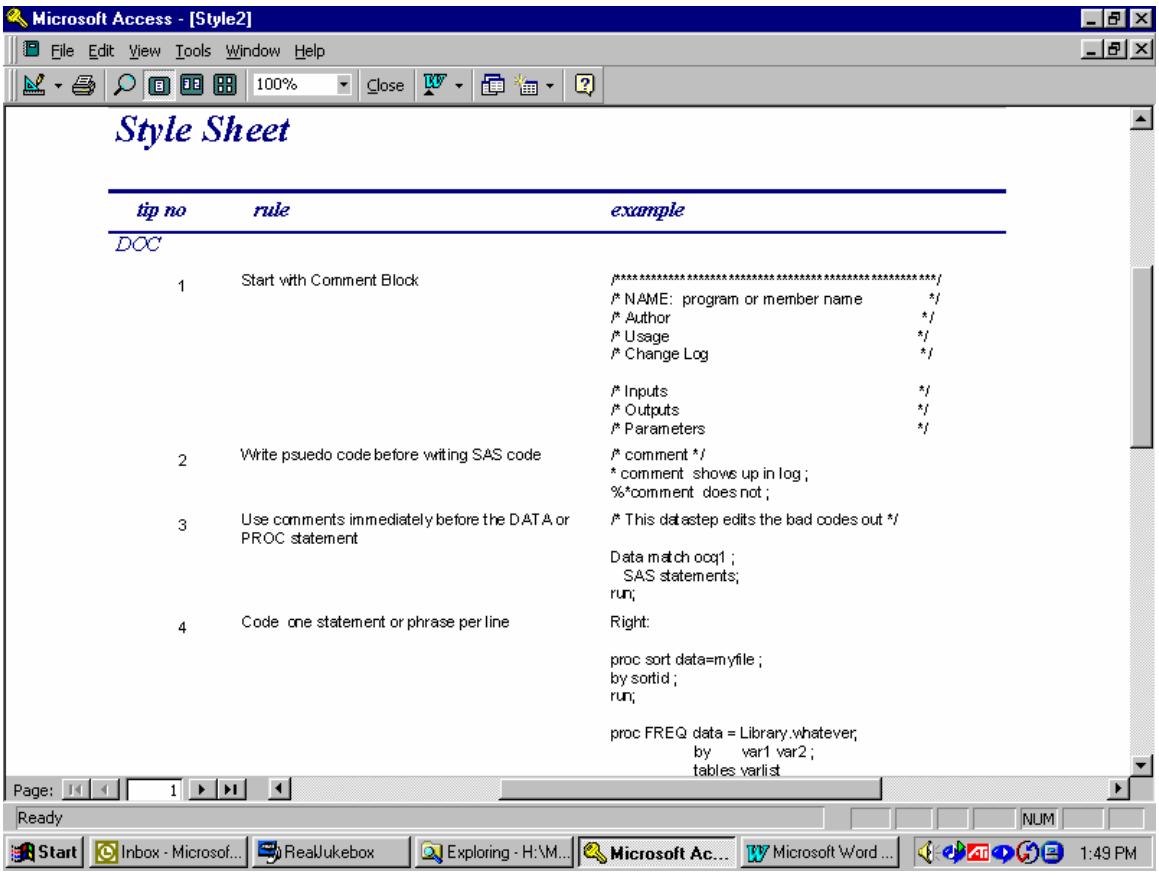
Comments, questions, and additions are welcomed.
Contact the author at:
Dianne Louise Rhodes
Connect International, Inc.
2275 Research Blvd., Suite 500
Rockville, MD 20850
Work Phone: (301) 763-2093
Email: diannerhodes@comcast.net

The tips database is an MS Access® database. If you would like a copy, contact me and I will provide it to you.

TRADEMARKS

Appendix of Reports




Microsoft Access - [Style2]
File Edit View Tools Window Help
100%
Close
Style Sheet
tip no
rule
example
DOC
1
Start with Comment Block
/**/
/* NAME: program or member name */
/* Author */
/* Usage */
/* Change Log */
/* Inputs */
/* Outputs */
/* Parameters */
2
Write psuedo code before writing SAS code
/* comment */
* comment shows up in log ;
%*comment does not ;
3
Use comments immediately before the DATA or PROC statement
/* This datastep edits the bad codes out */
Data match ocq1 ;
SAS statements;
run;
4
Code one statement or phrase per line
Right:
proc sort data=myfile ;
by sortid ;
run;
proc FREQ data = Library.whatever;
by var1 var2 ;
tables varlist
Page: 1
Ready
NUM
Start
Inbox - Microsof...
RealJukebox
Exploring - H:\M...
Microsoft Ac...
Microsoft Word ...
1:49 PM