Paper 10

Taming the Chaos: Managing Large SAS/AF Applications Using Programming Standards and the Source Control Manager of Version 7 of the SAS System

C. Michael Whitney, Motorola, Austin, TX

ABSTRACT

The use of programming teams offers both advantages and disadvantages when compared with individual programming efforts. The team approach allows a wide range of programming skills and problem-solving perspectives to be applied to a project, and may shorten development time. On the other hand, team-developed projects are often marred by differences in programming styles among developers, resulting in inconsistencies in the use of variable names, levels of documentation, and user interface design. Further, without some form of code management, the risk of inadvertently overwriting the work of other team members is always present. All of potential problems can make development and maintenance a chaotic and frustrating experience.

This paper will discuss how these problems can be minimized, if not eliminated, by applying some basic programming standards, and by making use of the new Source Code Management system that has been released with SAS version 7.

SAS products discussed in this paper include SAS/AF, SCL, and the Source Control Manager (SCM). The coding practices discussed in this article are applicable to all versions of the SAS System, on all platforms, and are aimed at developers with moderate to advanced SAS/AF & SCL experience. The code management portion is applicable only to the version of the Source Control Manager (SCM) that was released with SAS version 7. Earlier experimental versions of the SCM are not covered.

INTRODUCTION

Programming standards are essential for producing high quality, easily maintained software applications. The larger the application, the more this holds true. Using a code management system, such as the Source Control Manager provided with SAS version 7, further enhances maintainability. Such a system enables a programmer to check code modules in and out of a software library during development, or when the code needs updating, in much the same way a book is checked out from a library. Modules that have been “checked out” cannot be modified by others until they have been “checked in” by the current programmer.

The purpose of this article is to provide a framework for high quality, structured, and easily maintained SAS software. Uniformity of software development is essential in creating and maintaining computer systems that operate at peak efficiency. Due to space constraints, only the Top Down development approach is discussed. This is the method most applicable to the programming practices employed at most organizations. However, the principles discussed here are equally valid when applied to the Object Oriented development.

Recently, the Motorola Semiconductor Products Sector (SPS) unified its various Information Technology (IT) divisions into a global unit, rather than each SPS factory having its own IT team working independently of the others. One of the many benefits of this has been a pooling of knowledge as programmers from differing IT teams have been brought together. This has allowed us to take the best programming techniques used at the formerly independent sites, including standards, and come up with a good unified set of programming practices for our teams.

The guidance presented here has been collected by the author over the past eleven years, working for the US Air Force, a consulting firm, and the Motorola SPS IT Engineering Analysis Tools (EAT) team. It is applicable to any organization, be it governmental, corporate, or educational.

To be effective, programming standards should apply to all applications created in your organization. Applications created without standards applied to them should be brought up to the standard as maintenance is performed, if possible.

SOFTWARE DEVELOPMENT

SOFTWARE DEVELOPMENT LIFE CYCLE

The software development life cycle begins with a determination that a software application is required for customer support and ends with the cataloging of the application into the production libraries. In the SPS Engineering Analysis section, most projects begin when a device or product engineer at a factory requests a new statistical analysis tool, or the addition of new features to existing applications.

The development life cycle identifies important phases associated with the process of developing software. When completed, each phase should significantly reduce future maintenance costs and minimize the chance for software errors in the final product. The amount of detail required for each phase should be directly related to the size and complexity of the project. Large, complex projects require more extensive documentation than do small projects.

OVERALL LIFE CYCLE

There are several major phases in the software development life cycle. Figure 1 provides a summary of this cycle. The following is a brief description of the major phases:

1. **Gather and Define Requirements** - The first task is to determine if existing software can be used to fully or partially meet the customer's requirements. Analysis and research are the foundation on which the rest of the project lies.

2. **Prepare Initial Design** - If software development or modifications are necessary, the programmer formulates the initial requirements and designs documentation. The more detail that can be provided, the more easily understood the design will be. This design should be presented to the customer for approval, ensuring that it meets the customer's requirements.

3. **Formal Design and Design Review -** The detailed design is an effort to construct a concise, logical solution to the problem, which the programmer can easily translate into code. The design should be approved before coding begins.

4. **Test Plan, Coding, and Testing –** The appropriate coding standards should be followed for all production software. The development and execution of a test plan is essential to ensure that the customer receives a quality product (both software and data). Testing should occur any time that changes are made to the code. After testing is completed,

the software should undergo a review or walk-through prior to production.

5. Final Code Inspection - This is an intense, line-by-line review of the software prior to cataloging.

6. Cataloging - This process places approved software into the Production Software Libraries. It is extremely advantageous to place production code in libraries where only the software librarian has write authority. This will prevent accidental corruption of the source code and ensure that everyone is running the same version for production.

Top-Down Structured Design

Software should be developed in a top-down structured manner. Top-down design begins with formulating the solution in terms of generalized statements. After the general algorithm is developed, it can be refined by adding the details that are necessary to perform the general actions. For a complicated problem, this refinement process may be repeated several times, with each version containing more detail than the last. The design proceeds from the top (most general) to the bottom (most detailed), with the resulting design reflecting the nature of the problem. Top-down structure makes the program highly readable and easier to follow.

Structured software consists of separate functional modules. A module is a subportion of a program and is composed of a bounded group of instructions with a single identifier. A module may be a subroutine, function, or driver. In SCL terms, a module could be a either an SCL program, or a labeled section within an SCL program. Labeled subsections provide great modularity! Modules may call other modules, and in many cases, several modules may be required to complete a single function. Modules are designed with control flowing from the top to the bottom.

Top-down structured software should adhere to the following characteristics:

- **Cohesion** - All modules should perform single functions or small, related functions that require common data and pass information from one step to another.
- **Coupling -** Connections between modules must be obvious and should be minimized as much as possible for simplicity. Data will be passed as arguments between modules, when applicable.
- **Limited data exchange** - A minimum amount of data should be passed between modules. If only two variables out of a 100 variable data set are needed by a particular DATA step, use the DROP= or KEEP= data set option to eliminate those variables not needed. Similarly, only those SCL variables and list pointers that are required for a particular method to perform its given function should be passed between methods.
- **Exit and entry points** - Modules should have one entry point and one exit point.
- **Span of control** - Never let a module (except the program driver) directly control or call more than seven subordinates.
- **Scope of effect** - Subordinate modules are modules that make decisions so that other modules can complete a function. The scope of effect of any module includes all subordinate modules that are necessary for the completion of the module's function.
- **Module size** - Modules should consist of no more than four pages (200 lines) of executable code, minus comments. Modules should not perform more than one unrelated function, but may handle more than one related function for the program. If a module grows larger than 200 lines, check to see if the module is accomplishing more than a single function. If so, attempt to break it down into single-function modules. However, do not cut a single-function routine into multiple pieces simply for the sake of module size.
- **Independence -** The execution of the module is completely independent and does not depend on anything that occurred in previous invocations.

Top-down structured software is a direct result of the use of structured techniques and tools such as data flow diagrams, flow charts, pseudo code, structure charts, data dictionaries, and organized documentation. It requires forethought and work from everyone.

Top-down structured design has the following advantages:

- It allows for reusable modular coding, testing, and implementation.
- Design problems are detected early, when they are cheaper and easier to correct.
- Module development allows the programmer to concentrate solely on individual modules while treating other modules as black boxes.
- Fewer programming errors are likely to be made due to the module's significantly reducing program complexity.
- Modularity makes debugging easier. Problems are quicker to isolate in any particular module, shortening debug time.
- Modules are used more easily by other programs.

CODING STANDARDS

DOCUMENTATION

All program modules should be documented. The documentation must be standardized to promote uniformity in the program library.

At the top of each SCL or SAS program there should be a main program documentation section, that contains the following information in order to provide a complete reference of what's been done to the code in its lifecycle. Make sure that any additional information in the documentation section is pertinent.

1. **Program or Method Name -** name of the module.
2. **Support** - who, or what organization, 'owns' this module.
3. **Product -** which application does this code belong to?
4. **Purpose** - a brief, one-sentence description of what the module does.
5. **Usage** - how the module is called.
6. **Parameters** - what is passed into and out of the module.
7. **History** - details when the code was modified, who did it, and what was changed. The history section is the key to modifying or repairing the program properly. All entries in the history section must be as complete as possible.

   Each time the code is modified, an entry should be made in the history section. History entries should be completed for each re-cataloging of the program and should include the following information:

   - The date of the modification and project number. In the IT section of Motorola SPS, we use Rational's ClearDDTS™ defect tracking system for documenting bugs and enhancements requests. Each entry in that system has a unique tracking number. If your organization has a similar method of tracking requests, that number should go here, as well as the name of the person or persons who worked on the program.
   - A complete description of why the program was modified, what and where changes were made, and the results of the changes. Also, include any differences between the old and new versions.

- Additional information may be included in the documentation as needed, but a good rule of thumb is to limit the documentation to eight printed pages (400 lines). A general overview of the program is all that is needed to fulfill documentation requirements.

8. **Notes** - helpful notes for future maintenance. Typically you'll see warnings, or comments about future changes that should be made to this code. This is also a good place to list any references used for making the program. Notes may be entered anywhere in the documentation that is deemed necessary, but a grouping of the notes is easier to follow.
9. **Labeled Code Sections** - each labeled section of the SCL program should be listed here, in a logical order. Sections should be listed in the order in which they occur in the program. Whether it's alphabetical, or broken down as alphabetical for Frame widgets and non-widgets, or some other method, is up to you.
10. **Data Dictionary** - an alphabetical listing of all variables (including macro variables) used in a program is extremely useful. The SCL LENGTH code statements can often double for this purpose, if in-line comments are used after each LENGTH entry.

Within the body of the program, individual sections of code should be documented as well, describing the what and why of a particular code segment. Be as precise as you can -- you may be the one maintaining that code a year from now, when you've forgotten just exactly what that segment was supposed to be doing. Often, an explanation of **why** something was done is much more important than **what** was done – the **what** may be readily apparent from the code, but **why** a code section exists is not.

Without good documentation, maintenance becomes much more difficult. We have probably all heard the saying about not documenting the code being good job security, but even the best programmers cannot remember every detail about what they themselves wrote just a few months before. Poor documentation hurts everyone.

Please see Example 1: Sample Documentation Section

COMMENTING THE PROGRAM

Comments should precede the executable code and be set off in a uniform manner. In-line comments should be right-justified for readability. Each block of code should have a preceding block of comments pertaining to the workings of the code. Blocked comments ought to be set off with noticeable borders or blank lines. A continuous line of asterisks is common practice and provides unmistakable border. Single line comments should be avoided because they are easy to overlook; however, if necessary, they should also be set off with noticeable borders or blank lines to prevent code and comment confusion. Comments must be clear and concise with consideration for those who have to maintain or use the program. Having too many comments in a program is as bad as having too few. Use only enough comments to make the program understandable.

Please see Example 2: Sample Comments

CODE INDENTION AND COLORING

The purpose of code indention is to improve the readability and the logical structure of programs through a format that reflects the logic of the program:

1. Each base SAS DATA and PROC statement, and SCL entry labeled section should start in column 1.
2. Each subsequent line should be indented evenly.
3. Each DO group level should be indented, with DO statements starting on a new line for easy visibility.
4. The END statement used to terminate a DO loop should be indented at the same level as the starting DO statement.
5. Comments should not be indented to match corresponding code, and must precede the relevant code.

Coloring the SCL code can also improve readability. Our EAT team uses a gray background color, with code in black text, comments in blue, and green text for code in submit blocks. Comments in submit blocks are green as well. One of our factories IT teams used a black background, yellow text, cyan comments, and white submit blocks. Any set of contrasting colors that your programmers can agree upon will work.

NAMING CONVENTIONS

Without a standard set of variable names, code reusability means that each SCL program must often rename the variables or parameters of the calling program or macro in order to meet the naming convention of the calling program. It is much simpler to use a standard set of names for all programs written in your organization.

Further, it is useful to adopt a common naming convention for all of the individual SCL and FRAME components that make up each application. We've implemented the following naming standards at Motorola SPS IT:

- **Product Abbreviations** – All major products are identified by a unique two-character product identifier prefix. For example, the Engineering Data Analysis System[1] (EDAS) product uses "ED", while the Data Analysis Reporting Tool[2] (DART) uses "DA". Further, a few two-character codes are used for non-product-related systems, such as "C_" for common code, and "UT" for utilities.
- **Library and Catalog Names** **-** All libraries and catalogs associated with a given product use that products prefix. For instance, the EDAS project code library is "EDASLIB", while the EDAS beta code goes into the "ED_BETA" library, and the EDAS data management library is "ED_MANAG". Catalog entries should have the product revision ID in them: "EDAS50" for the 5.x release of EDAS.
- **Entry Names** – Use of the SCL SEARCH statement allows an application to call SAS/AF entries stored in other libraries, thus increasing the functionality of the application by taking advantage of a modular design.

  Entry names may be specified without specifying which catalog they are in. By providing a search path, the application will search a list of specified catalogs one-by-one until it finds the first entry matching the name specified. This feature aids greatly in development, since an experimental entry may be substituted for testing by merely adding a test catalog earlier in the SEARCH path for the tester. (The SAS Source Control Manager takes advantage of this ability).

  The drawback is that if you having SCL entries with the same name in the search path, regardless of the catalog names, may result in one of the products failing to work correctly.

  For this reason, a portion of the entry needs to reflect the product to which it belongs, using the same product identifier as its parent application. For instance, the FRAME entry and its associated SCL for the EDAS scatterplot module are called "ED_SCATR.FRAME" and "ED_SCATR.SCL". Non-FRAME SCL is differentiated by not having the underscore: "EDSCATR.SCL". All of the

[1] EDAS was presented at SUGI 16 by Leslie Fowler. (See reference section)
[2] DART is the successor of DevIS, presented at SUGI 21 by Larry Worley. (See reference section)

EDAS product code is stored in the EDAS50 catalog, in the EDASLIB library.

- **Macro Names and Macro Variables** –
  - Run-Time and Pre-Compiled Macros - Macros or macro variables which will exist for a user, such as run-time or pre-compiled macros, should start with an underline, followed by the two-character product identifier. I.e., "_EDMACRO". The underline makes it easier to differentiate a macro variable from a normal one. Since users may also create macros, it is important that the application's macros not interfere with their macros, and vice versa. Most users are warned not to begin their macros with an underline. (This is an important point to add to user documentation for most applications.)
  - Compile-Time Macros - Macros or macro variables which will exist only for the developer to aid in code generation - but which will not exist in the user's SAS environment when the user runs the application -- may follow any naming convention, since only the developer's environment will be affected. The convention discussed above may be followed, of course, if there is any doubt about the impact on the user or other developers.
- **Variable Names** – Here are a few recommended guidelines for variable names:
  - Avoid using variable names that duplicate SAS keyword or function names. Although SAS usually deals with these correctly, it makes reading the code very difficult.
  - Because list manipulation is so important in SCL, variables which are list identifiers stand out as such when the last part of the variable is either ...list or ...lst. These suffixes should be avoided, where possible, for non-list ids.
  - Dataset identifiers associated with a SAS dataset stand out as such when the last part of the variable is either ...id or ...dsid. These suffixes should be avoided, where possible, for non-dataset ids.
  - Temporary or holder variables can easily be designated as such by the use of temp, tmp, hold, or hld as part of their names.
  - Single-character variables such as i, j, k, etc., are fine for loop counters. Try to use more mnemonic names for important variables.

TESTING THE PROGRAM

CREATING AND TESTING THE TEST PLAN

Create an initial test plan based on the requirements and design documentation. This plan is a list of the tests to be made to verify the correctness of the results. The test plan identifies the test data to be used. Update the test plan as necessary during testing to reflect the actual testing done. All software requires testing by both the programming team and the customer before it is run in production or cataloged. Testing ensures that customer requirements are met or exceeded as far as possible, that coding and logic errors are discovered and corrected, and that each routine does what it was designed to do.

Testing should follow the test plan. Update the test plan when new or additional tests are required, or if tests described on the test plan become unnecessary or too difficult or impractical to perform. As a minimum, software should be tested as follows:

1. Test the program with both test and actual input data. Testing should include stressing the program with data both in and out of the testing parameters. This is done to ensure the program either stops or rejects the bad data.
2. Test all modules to the maximum extent possible with valid data. If possible, make sure each decision is executed at least once. Do hand calculations, if necessary, to verify that each module is functioning properly.
3. Have someone else test your program. Often, another person can discover awkward or cumbersome procedures or manage to break the program with erroneous data.
4. After modular testing, test the entire application as a whole to ensure that all of the modules work together properly. Be especially aware of the arguments and units being passed between the subroutines. Often, different arguments are required for different subroutines. Make sure the correct arguments with the correct units are passed to each subroutine.
5. Include an acceptance clause at the end of the test plan. This should be signed by the customer and the programmer at the conclusion of testing.

WALK-THROUGHS

A walk-through is a group evaluation of a product at various stages of its life cycle. Walk-throughs should be formal, properly structured, and well-documented. Proper structuring will make the walk-through more beneficial. Walk-throughs allow you to produce reliable, error free code. They can reduce the average from three to five errors per 100 lines to as few as three to five errors per 10,000 lines (Freedman 1990). They can help you correct design flaws and improve program documentation, as well as cut production time by as much as 50 percent. They also help increase the quality of system software. There are several types of walk-throughs:

- Design walk-throughs focus on the solution to the problem. This is critical in that it sets the guidelines on how a project is to be completed.
- Periodic walk-throughs are conducted whenever deemed necessary. Periodic walk-throughs will tell you where a project is.
- Final walk-throughs are necessary prior to submitting a program for a final code inspection. This walk-through will help find any discrepancies previously missed.

A minimum of three and a maximum of seven individuals should be involved in any type of walk-through. The size and scope of the project should determine the number of attendees. One individual moderates the walk-through. A second individual should record all pertinent information discussed during the walk-through, such as recommended changes to the material being presented. A third individual presents the material.

WALK-THROUGH STAGES

There are three stages in the walk-through process:

1. **Review stage -** This three to five day period prior to a walkthrough is used to acquaint each attendee with the product. Standards, checklists, material to be reviewed, and any relevant document from prior reviews or walk-throughs will be looked over during this stage.
2. **Walk-through stage** - In this stage, each detail of the product is reviewed. The presenter or moderator guides the meeting, which should last no longer than one to two hours.
3. **Follow-up stage** - This is where changes are implemented. All involved parties are informed of the changes and must agree to them.

HELPFUL HINTS

Here are a few very simple guidelines to remember while conducting a walk-through:

- The author is not on trial.
- The product is guilty until proven innocent.

- Choose walk-through participants carefully. Avoid personality conflicts if at all possible.
- Keep walk-throughs within the predetermined time limits. Schedule well in advance to ensure that everyone needed for the walk-through can attend.
- Create and follow a checklist of possible problems.

There are some people problems to watch out for in reviews and walk-throughs. Egos can play a factor in that people naturally do not like to be told they've made mistakes. Another problem area is inexperience at giving and receiving criticism. (Misdirecting comments at the creator rather than the code can turn the meeting into a defensive war.) The final problem is apathy - not trying hard to find errors. (Don't assume that others will find the same errors that you find.)

The team only has three decisions to choose from at the conclusion of a walk-through: accepting the product as is; accepting the product with revisions, trusting the creator to make the fixes; or determining that another walk-through is necessary after the errors have been corrected and the comments for improving the product have been implemented.

REVIEWS

A review is an informal check of a portion of a software program that can be conducted at any point in the development or maintenance processes. Very little documentation is needed, and structure is of no concern. Two or three individuals are sufficient for a review. Proper use of both reviews and walk-throughs will result in better software products and reduce long-range maintenance costs.

Program reviews may be made at any point in the program development cycle. The best times to review a program are after the design is developed, before any formal walk-through or code inspection, and before cataloging and production. Periodic reviews of all programs, either being developed or modified, are essential to ensure adherence to programming standards, that errors are detected (as undetected errors will haunt you later), and that documentation is correct. Make entries in the project log for each review. A properly kept log may help trace any problems that may arise later on in the project.

SOURCE CONTROL MANAGER

FEATURES

With the release of SAS version 7, SAS Institute has provided a new tool, the Source Control Manager (SCM), for managing the SAS/AF source and data files that make up your applications. Previously released as an experimental tool, the SCM is now fully integrated with SAS/AF. It provides a robust environment for developing and maintaining your SAS/AF applications, allowing you to check code in and out of the library, test changes before checking code modules back in, carry out revision control and version labeling, and easily distribute your application. The SCM environment contains a number of tools for generating reports on the development library and comparing file differences. Among these tools, the SCL Static Analyzer tool is especially notable for its ability to provide a wealth of information about the SCL in a given catalog.

The Source Control Manager (SCM) features a point-and-click interface, with pull-down or pop-up menus through which you can issue commands. The interface gives you the ability to browse the software libraries associated with the SCM, and the various catalogs and entries contained within them.

The SCM creates a control database that is associated with a given SAS library. When a developer checks code out of the SCM, the file is copied to his or her specified work area. The SCM then updates the control database by placing a lock on the file so that no one else using the SCM associated with that library can check out the code until it has been checked back in, thus preventing overwriting accidents.

By using the new CATNAME function in conjunction with the SEARCHPATH function, the SCM allows most code modules to be tested before it is checked back into the library.

Each time a file is checked back into the SCM, the previous version is archived. This provides an easy method of backing changes back out of the application if needed. The SCM administrator determines the total number of archives kept.

Further, the SCM's version labeling feature allows a 'snapshot' to be taken of the revision numbers of all the modules that make up an application. This way, if you need to rebuild an earlier release, the SCM will pull the correct revisions out of the main library and the archives to build a given release.

Once a version label has been created, you can copy that version of your application to a central distribution point, or to remote computers via SAS/CONNECT.

USING THE SCM

The SCM is located on the SOLUTIONS pull-down menu, in the DEVELOPMENT AND PROGRAMMING sub-menu. Or, you can type '`SCM`' on a SAS command line. Optionally, you can specify the location of the control database when using the command line, as well by using '`SCM SCMDATA=`*`CDBLibref`*'.

The documentation for the SCM is provided in the form of online help screens, from the SAS help menu. At the time of this writing, no documentation was available via the SAS Online Documentation CD, other than a brief mention of it.

When you start the SCM for the first time, you will need to associate a software library with it. Once you have done so, the SCM will create several datasets that make up the control database in the library. In this database are stored the list of all the files that are a part of the project, the location of the preference files for each person, who has which files locked, where the archives are to be stored, and more. Each developer can set preferences as to where his or her work library is located for each project library. The location of these preference files is determined when the SCM control database is created, and defaults to SASUSER. SAS Institute recommends that SAS/SHARE be used to access the control database library.

More than one project library can be assigned to a given control database. To do this, start the SCM with the control database you want. Then, from the TOOLS pull-down menu, select ADMINISTRATION UTILITIES. On the SOURCE DATA tab, new libraries or catalogs can be registered. Select the library or catalog, provide an archival location, and click the REGISTER button. You can choose whether or not you want all development libraries assigned to one SCM control database, or just those used for a particular project. However, it would appear to be best to include all libraries needed for a given application if you plan on using the version labeling feature.

Once the control database has been set up and configured, it's ready to be used by the developers. The programmer will start up the SCM, and tell it which control database library to use. The project library associated with that control database will then be displayed, and if any files have been checked out, those will appear with an icon of a lock in front of them. Double clicking on a file name will bring up that file in browse mode. The developer will need to tell the system where to copy the checked out files using the OPTIONS item from the pop-up menu, or TOOLS > OPTIONS from the pull-down menu. This is at a library, rather than a catalog, level. The SCM will also ask for the developer's name, so that it can assign checked out files to the developer.

SCL STATIC ANALYZER

Another formerly experimental tool, the SCL Static Analyzer is now part of the SCM environment. It can tell you a great deal about the SCL that goes into your application, and works on a given catalog. When activated, it examines all the SCL in the catalog, and provides the following statistics: total lines, instructions, functions, attributes used, and the number of unique entries, labels and methods, variables, functions, etc. Further, it can detail the flow of your program, listing all the interrelations between the various modules that make up your application. All of the data the program collects is stored in datasets that can be viewed from within the system by clicking on the VIEW DATA button. All of the statistics and listings pop up in a compact tabular view window. Also, the Static Analyzer will provide a list of warning areas you should look at in your code…including areas of dead code and uncompiled entries, as well as areas where the Analyzer can't find the source entry being called, etc.

SCM NOTES

Several notable items:

- When checking out files, be sure not to have the catalog selected -- you **can** check out an entire catalog by accident.
- The current release of the SCM doesn't support creating copies of a version label if the version label contains multiple library references.

CONCLUSION

Programming standards provide a means of ensuring that software written at your organization is of high quality. But in order for standards to be effective, everyone must follow them.

The SCM provides an elegant solution to the problem of having program files accidentally overwritten, and provides additional functionality to the already excellent SAS development environment.

Unfortunately, due to size constraints, this article could offer only a brief overview of the software life cycle and the SCM. For more information on the life cycle and other aspects of programming standards, you might turn to the Freedman and Dunn's books listed below. For more information on the SCM, see the SAS Online Help files.

REFERENCES

Fowler, Leslie, et al (1991), *Data Analysis Applications for the Semiconductor Industry*, Proceedings of the Sixteenth Annual SAS User's Group International Conference, pg. 658-662.

Worley, Larry and Nelson, Jim (1996), *DevIS: Motorola's Near-Real-Time Device and Visualization System Using SAS/AF Software and SCL*, Proceedings of the Twenty-first Annual SAS User's Group International Conference, pg. 654-659.

Freedman, Daniel P. (1990), *Handbook of Walkthroughs, Inspections, and Technical Reviews*, New York: Dorset House Publishing.

Dunn, Robert (1982), *Quality Assurance for Computer Software*, New York: McGraw-Hill Book Company.

USAFCCC (1995), *Communications-Computer Systems Automated Data System Standards and Procedures*, Instructions 33-102, Volumes I-III, Scott AFB.

Motorola SPS IT Engineering Analysis Tools SAS Programming Standards (1998).

ACKNOWLEDGMENTS

Thanks go out to Les Bortner for a superb reviewing job and editing suggestions. Thanks also go to Chris Weyn, Leslie Fowler and Robert Smith for their reviews of the paper, as well.

CONTACT INFORMATION

(In case a reader wants to get in touch with you, please put your contact information at the end of the paper.)

Your comments and questions are valued and encouraged. Contact the author at:

C. Michael Whitney
Motorola SPS
2150 Woodward St.
Austin, TX 78744
Work Phone: (512) 373-3164
Fax: (512) 373-3171
Email: ra6952@email.sps.mot.com

EXAMPLE 1 – SAMPLE DOCUMENTATION SECTION

```
/*----------------------------------------------------------------------*/
/* Method:   EDASINIT                                                   */
/* Support:  Mike Whitney, SPS IT Solutions - Engineering Analysis      */
/* Product:  EDAS                                                       */
/* Purpose:  the method that starts EDAS50, site configuration          */
/* Usage:    call method('edasinit.scl','start','load_buffer');         */
/* Parameters:                                                          */
/*          what_to_do $20: method called by init file that starts EDAS */
/*          'load_buffer' : method called when build code for batch job */
/*          'script'      : method called when creating a script        */
/* History:                                                             */
/*    Sep 95 - M. Grover                                                */
/*           - Initial coding                                           */
/*                                                                      */
/*    Feb 97 - J. Nelson                                                */
/*           - added simple debug listing                               */
/*                                                                      */
/*    Oct 97 - E. Stokes                                                */
/*           - Added code to determine OS and pathname                  */
/*             Created _edstart macro.                                  */
/*                                                                      */
/* 20 Jan 98 - Mike Whitney - CIMcm0279                                 */
/*           - modified OS pathname building routine, removed           */
/*             non-functioning SymPut calls.  Modified VMS section.     */
/*                                                                      */
/* 23 Jun 98 - Mike Whitney - CIMcm0280                                 */
/*           - tweaked usrtr assignment to correctly reflect the VMS    */
/*             user path.                                               */
/*----------------------------------------------------------------------*/
/* Notes:                                                               */
/*----------------------------------------------------------------------*/
/* Labeled Code Sections: (listed in order of appearance)               */
/*  START – method that starts EDAS                                     */
/*  LET - Assign let statements for padas directories                   */
/*  SYMBOLN – build the SAS symbol statements                           */
/*----------------------------------------------------------------------*/

length
    ct            $3     /*  Session settings:  graphic char type   */
    dsname        $17    /*  dataset name                           */
    edasos        $8     /* edas' name for the current os           */
    fullpath      $80    /* physical path of 'padas' subdirectories */
    grcat         $17    /*  Session settings:  graphic catalog     */
    ht            $3     /*  Session settings:  graphic text height */
    let_libname   $200   /* Libname statement for padas libraries   */
    let_statement $200   /* %let macro definition statement         */
    listid        8
    ls            $8     /*  Session settings:  line size           */
    msg           $80    /*  Generic Return Message                 */
    OpSys         $35    /* substr of sysccp global macro           */
    parmtab       $17    /*  parameter table name                   */
    ps            $8     /*  Session settings:  page size           */
    rdsname       $17    /*  raw dataset name                       */
    sas_user      $80    /* physical path of the sasuser directory  */
    screenme      $17    /*  single node dataset name               */
    sys_command   $200   /*  Operating System Command               */
     ;
```

This is an older SCL entry, which wasn't documented as well as it could have been. The history was brought up to date during recent modifications

Note the issue tracking numbers.

Here the LENGTH statement does double duty as a data dictionary.

EXAMPLE 2 – SAMPLE COMMENTS

```
      control asis;

   START:   /* Start EDAS */

      method
          optional=what_to_do  $20;

   /*******************************************************************/
    The following submit block is here to make sure that SAM recognizes
    a new script from an older one. - 1/98 by Chris Weyn
   *******************************************************************/
```

An in-line comment used to briefly describe the purpose of the labeled section.

Commenting the code as to **why** something was done.

EXAMPLE 3 – CODE INDENTATION

Indented code is easy to read and follow.   Only one comment has been left to show how they should be indented.

```
   START:
      if symget('_eddebug')^='ON' then
         submit;
            options nosource nonotes nosource2;
         endsubmit;

   /* Get the physical path of the sasuser directory    */

     select (edasos);
       when ('VMS')  sas_user= scan(pathname('padas'),1,']');
       when ('UNIX') sas_user= pathname('padas');
       when ('WIN')  sas_user= pathname('padas');
       when ('MAC')  sas_user= pathname('padas');
       otherwise;
     end;

     sas_user=lowcase(sas_user);

     if what_to_do ne ' ' then do;
       if what_to_do = 'START' then do;

         submit;
            %let _eddsnm = &dsname   ;
            %let _edrdsnm= &rdsname  ;
            %let _edsrnme= &screenme ;
            %let _edprmtb= &parmtab  ;
         endsubmit;

       end;
     end;
     else do;

       submit;
         %let _eddsnm=  ;
         %let _edrdsnm= ;
         %let _edsrnme= ;
         %let _edprmtb= ;
       endsubmit;
     end;
   return;
```

Having the label in the first column, and the rest of the code indented, makes the labels more noticeable.

For the same reason, comments aren't indented, either.

Nested if-then-do loops

Submit block contents should be indented, as well.