MWSUG 2016 - Paper SY04

Avoiding Code Chaos - Architectural Considerations for Sustainable Code Growth

David L. Ward, Nashville, TN

ABSTRACT

As projects grow in complexity – both in terms of code and personnel – source code can quickly become very difficult to maintain, extend, and test. While this problem certainly occurs across programming languages, SAS® seems uniquely suited for unmanageable complexity. First, SAS programmers are often trained more as analysts than as software engineers, so they tend to think more about organizing data than a complex code base. Second, the procedural nature of the language itself can contribute to code chaos by allowing for extremely long programs without much re-usable code. Third, though the macro language is powerful, it can often be used to create code that is very hard to understand.

This paper will present the essential components that must be considered when designing a SAS application that will thrive under growth and make concrete recommendations for their implementation. This outline can be taken back to your programming team to foster discussion and help work towards the adoption of design and coding standards that will make development more efficient, effective, and maintainable, and more importantly – make programming more enjoyable. The following topics will be treated: readability, syntax, clarity, headers, file names, modularization, inputs/outputs, directory structure, entry/exit points, handling log and output files, automation/looping techniques, and version control.

INTRODUCTION

Most of us have been down a code rabbit hole before. We were handed what appeared to be a simple assignment – document inputs and outputs of a legacy SAS application so that portions of it could be re-written or moved to other locations. After an embarrassing amount of time spent following nested macros, sysexecs, %includes, and turning on various SAS system options in debugging desperation, our resolve to never write code like that was steeled. But without a clear strategy – and not just for ourselves but for our whole team – we might inadvertently end up with the same mess years later, and horrifyingly it may have our own names on it.

Learning to create code – and beyond code to entire applications – that is easy to understand and manage is a vital skill that will benefit both yourself and your organization. Learning not only to code in this way but also to motivate others to do so will help you develop as a leader and manager and further benefit your organization and reputation for years to come.

READABILITY

Perhaps the most important consideration when attempting to write code that will be easily understood and supported by others is the *readability of the code itself*. The SAS compiler imposes no syntax restrictions and is case insensitive, so without careful thought and discipline, SAS code can be very difficult to read. For example, the following code is difficult to read:

```
data two[1] (keep=var1 var2 [2]compress=yes); [3]set one end=last;
by var1; [4]* var1 is important *;
length obs 8; retain obs;
obs = _n_;
if first.var1 then do;
[5]var2=var1;
```

[1] Data set names are meaningless
[2] Multiple data set options in one line make it difficult to see when one ends and another begins
[3] Multiple statements on one line make it easy to miss statements and commenting difficult
[4] This comment is meaningless and does not stand out from the code
[5] Lack of indenting in a DO … END block make it difficult to see what is in the block

```
    obs=obs+1; [6]end;
 [7]proc sort[8]; by var1 var2;
```

Compare the readability of this block to the following one which contains the same code but with better formatting:

```
DATA two (keep = var1 var2
          Compress = yes);

    SET one end = last;
    BY var1;                  * SO WE CAN USE first.var1 SYNTAX *;

    LENGTH obs 8;             * ANOTHER COMMENT HERE *;
    RETAIN obs;

    obs = _n_;
    IF first.var1 THEN DO;
        var2 = var1;
        obs = obs+1;
    END;
RUN;
PROC SORT;
    by var1 var2;
run;
```

While you may disagree with the particular style choices made above, it should be evident that this code is much more readable. A set of syntax conventions should be created and followed across teams and ideally across an entire organization or organizational unit. This allows programmers to read and understand code much more quickly. Some programming languages take common syntax conventions so seriously that they enforce them at the compiler level, like Python. Python uses strict indenting rules such that code blocks like IF THEN DO will not even compile if they are not indented correctly. While this has drawn ire from some, the author is in agreement that the importance of readability trumps the opinions of some coders about their personal formatting style being cramped.

The particular aspects of your organization's syntax conventions are not important, so don't agonize too much about the particulars. What is more important is that your programmers actually follow the conventions, so peer code review and personal investment is important. The following suggestions may prove helpful in developing your own standards, even if only the *categories* are employed.

1. All indenting should use exactly four spaces, NOT tab characters. When supplying multiple options or arguments inside of parentheses, separate each option on a separate line.
2. Indent all code between DO blocks, NOT including the final statement
3. Use in-line comments (* … ;) when a group of statements may need to be commented out together. Most comments should be in all caps to help distinguish from code. Use two asterisks when commenting above code and one asterisk when commenting beside code. Line up comments beside code in the same column within the same step. Use block comments for larger sections (/* … */). Always start and end the comment block on separate lines.
4. Utilize upper case for SAS keywords (procedure names, data step, functions, etc.) and lower case for the rest (data set names, variable names, etc.).

[6] END of DO block is not indented or on its own line, making it easy to miss entirely
[7] No RUN statement is used for the first data step or the sort, a bad practice that can lead to undesired results
[8] Implicit use of &SYSLAST for the data set is unclear and can lead to unexpected results

5. Utilize underscores to separate logical phrases in names. E.g. DATA car_makes;
6. Always supply a final RUN; statement to data steps and procedures.
7. Always supply the data set name explicitly to procedures.
8. Always use the macro name with the %MEND statement
9. When mixing macro and base SAS code, use all uppercase for macro code to help distinguish it
10. Pad equals signs and other operators with spaces on either side

CLARITY

While readability refers to being able to quickly see what code is being executed, clarity goes a step further to helping the programmer understand the business purpose and logical flow of the program. A program can be extremely readable but still hard to actually understand. Consider the following ways in which you can write code that is clearer.

DATA AND VARIABLE NAMES

Choose your names descriptively. It has been many years since SAS only allowed 8 characters for data set and variable names. Gone should be the days of variable names like VAR1, X, or even CMNTKT. Even if a variable will only be used in one or two lines, resist the lazy temptation to use shorthand and meaningless names, except perhaps in the case of DO loop counters, where is almost universally accepted to use I, J, K, etc. Even temporary variables can have the prefix TEMP_ added, which will make it clear every time they are used that they are temporary and will not be included in the final output data set. Since SAS is case insensitive, consider using underscore to separate phrases in variable names, like NUMBER_OF_EPISODES. A little bit more typing now will make your code much easier to understand later.

README FILE

It has been a common practice in the computer programming field to include a text file named README (README.txt is also acceptable and more useful for Windows programmers) in the *root* directory of your application. This file is free-form and sometimes contains detailed outlines and diagrams of an application, entry-point scripts, author contact info, etc. It is *highly recommended* to include this file in your application.

HEADERS

All programs should have a header. In many cases when a program is short and obviously included within a larger framework, an abbreviated header may be used effectively. We have all been faced with trying to identify a random program that some coder has copied into a temporary location for you to examine. Without a header we don't (easily) know who wrote it, what project it was for, when it was written, or anything about what it is trying to do (without a detailed study of the code). Even if revision control software is utilized, most of this information isn't automatically tracked. Consider providing standard full and abbreviated headers with common formatting, with the following information mandatory:

```
Program or macro name
Project name
Program description
Purpose
Author
Creation date
Inputs & Outputs
```

Consider also making a configuration section obvious with comment blocks, immediately following the header, for anything that might need to be updated on a regular basis. For example:

```
/*-------------------------------------------------------------------------
BEGIN CONFIGURATION SECTION
```

```
---------------------------------------------------------------------------*/

** LOCATION OF INPUT DATA **;
%let input_data=/data/20160608/input;

/*---------------------------------------------------------------------------
END CONFIGURATION
---------------------------------------------------------------------------*/
```

COMMENTS

Some programmers believe that if code is readable and clear, there should be no need for comments. While this may be acceptable on a detailed level, comments do wonders to help the larger structure of your programs become more evident. Consider placing an outline of your code in the header, then marking each numbered section with comments to help make them easy to find and identify. Also consider adding extra blank lines and/or comment lines between groups of steps which are more closely related. When in doubt, add comments. Even if you think comments are unnecessary, after months or years go by others (or even yourself!) may be glad that you have them.

```
**************************************************************************
STEP 1: READ AND COMBINE INPUT DATA
**************************************************************************;
```

FILE AND DIRECTORY NAMES

It may seem painfully obvious to instruct the reader to choose program names that help identify their purpose, but far too often the author has seen program names like READ1.SAS, TESTPROG.SAS, etc. Directory names should also be clear and ideally follow a convention, making their purpose immediately obvious. Consider the following directory structure for a SAS application:

```
/etc – configuration files (text files with parameters)
/run – programs that are directly executed via sysin
/macro – macros added to sasautos
/in and /out – input (e.g. *.csv) and output files (e.g. *.lst)
/test – programs to test various portions of your application
/doc – documentation
/log – logs from programs in the run directory
```

It is also wise to use sub-directories to group files logically. Once there are more than thirty or so programs in one directory, it can be difficult to quickly find a particular program. Consider sub-directories for the run and macro directories like this:

```
/run/extract – extract data from host
/run/transform – transform (merge, collapse, etc.) data
/run/load – load data to final destination
/macro/core – core utility macros
/macro/files – working with files
/macro/functions – macro functions
/macro/odbc – macros related to the ODBC data source
```

Here is a sample recursive macro to add a directory and all sub-directories to the SASAUTOS option. Using a macro like this will allow you to make all macros available to your code regardless of the sub-directory organization you use to maintain clarity.

```
%macro add_sasautos(dir,subdir);
    DATA _null_;
        LENGTH file $500 sasautos $32767;
        rc = FILENAME('_sasaut',SYMGET('dir'));
        did = DOPEN('_sasaut');
```

```
            IF did THEN DO;
                anysas = 0;
                DO i = 1 TO DNUM(did);
                    file = DREAD(did,i);
                    IF INDEX(UPCASE(file),'.SAS') THEN anysas = 1;
                    ELSE DO;
                        ** IF DIRECTORY THEN QUEUE UP MACRO **;
                        rc = FILENAME('_sasaut2',"&dir/"||file);
                        did2 = DOPEN('_sasaut2');
                        IF did2 THEN DO;
                            CALL EXECUTE('%nrstr(%%)add_sasautos(' ||
                                         SYMGET('dir') || '/' || TRIM(file) ||
                                         ',1);');
                            did2 = DCLOSE(did2);
                        END;
                        rc = FILENAME('_sasaut2','');
                    END;
                END;
                did = DCLOSE(did);
                rc = FILENAME('_sasaut','');
                IF anysas THEN DO;
                    sasautos = GETOPTION('sasautos');
                    IF ^INDEX(sasautos,QUOTE(symget('dir'))) THEN
                         CALL EXECUTE('options append=sasautos=(' ||
                                      quote(symget('dir')) || ');');
                END;
            END;
            ELSE PUT "ERROR: Unable to open SASAUTOS directory &dir";
        run;
    %mend add_sasautos;
```

PROGRAM LENGTH

Just like having too many files in one directory can make quickly locating and understanding a program's distinctiveness difficult, having too much code in one file can make understanding the contents difficult. A better design would be to make re-usable or repeated code into macros or split up your program into separate steps using %include. Writing large volumes of code used to be a badge of honor among some old-school programmers; the author has even heard of programmers who got paid by the line, not the hour. But today, with the power of the SAS macro language, there is little excuse for writing voluminous programs unless you are intentionally obfuscating your code for job security or sadism. On the other hand, learn a lesson from Perl and don't let the drive for brevity of code create genius-but-unintelligible one-line or one-screen monstrosities. A rule of thumb is to write programs that are no more than about 8 screens long.

MODULARIZATION

A module is defined as a part that can be connected with other parts to build something. Writing code that is modular happens when you combine brevity, file and directory structure clarity, and concrete (documented) inputs and outputs. One of the main advantages of writing modular code beyond clarity is that it allows you to do *unit* testing rather than just *functional* testing. Modularization is a popular concept in the object-oriented paradigm, which SAS does not support overall (syntax for proc DS2 being an exception). But even though SAS does not have a well-defined construct for modules, we can use macros and %includes to function in a similar way. If used properly, most macros are by nature modules – they have a well-defined purpose that is usually clear by their file name, and have defined inputs (parameters) and outputs (data sets or other macro variables returned).

INPUTS AND OUTPUTS

A key component of writing modular code is to clearly identify inputs and outputs. Global inputs to SAS programs are made either through files which are read, or through the use of the –SYSPARM system option. SYSPARM allows passing a string to SAS from the command line which is made available in the &SYSPARM automatic macro variable. In order to pass multiple parameters, a convention must be adopted for passing multiple values in one string. One such convention that has been around for quite a while and is fairly simple to implement manually is URL encoding. The following code parses a URLencoded &SYSPARM into separate global macro variables:

```
%macro parse_sysparm;
    DATA _null_;
        LENGTH name $32 sysparm value $32767 rc $1 i 8;
        sysparm = SYMGET('sysparm');
        i = 1;
        DO WHILE (SCAN(sysparm,i,'&')^='');
            value = SCAN(sysparm,i,'&');
            name = URLDECODE(SCAN(value,1,'='));
            value = URLDECODE(SCAN(value,2,'='));
            IF NVALID(name) then CALL SYMPUTX(name,TRIM(value));
            ELSE PUT 'WARNING: Invalid sysparm parameter name ' name +(-1)
                     '. Unable to create macro variable.';
            i = i + 1;
        END;
    RUN;
%mend parse_sysparm;
```

Programs which are called from *within* other programs also have inputs and outputs. Consider the following calls to modular programs:

```
%let data_csv_file=/data/in/sales.csv;
%let data_set_directory=/data/out;
%let data_set_name=sales;
** THIS PROGRAM RECEIVES THIS MACRO VARS AS INPUTS **;
%include 'read/csv_to_dataset.sas';
** OUTPUTS WOULD BE THE sales.sas7bdat FILE AND
   POSSIBLY OTHER MACRO VARS **;

%csv_to_dataset(
    csv_file=/data/in/sales.csv,
    directory=/data/out
    data_set_name=sales);
```

Global Macro Variable Inputs and Outputs

Global macro variables can be a convenient way to pass parameters between modules. For example, in the CSV_TO_DATASET macro above you might want to return the number of observations read from the csv file. To do this you could define a variable &CSV_TO_DATASET_OBS as global, then assign it within the macro. Then after the macro executes you could access the contents of that variable. While this is convenient and powerful, it lends itself to undocumented inputs and outputs which would not be apparent to programmers trying to understand the programs later on.

The author had to debug an application once with thousands of global macro variables which were constantly read and/or written randomly throughout hundreds of macros. A much better practice would be to clearly list which variables are read and written to in the header of each macro.

Validating Inputs

Another component to successful implementation of clear inputs and outputs is throwing meaningful errors when required inputs are not supplied. Consider printing a note to the log containing which parameters were not successfully supplied and then aborting SAS in the case of an error condition which prevents the program from continuing:

```
%if %length(&required_parameter)<=0 %then %do;
    %put ERROR: Required parameter REQUIRED_PARAMETER not supplied.;
    %abort abend;
%end;
```

CONFIGURATION FILES

A configuration file is really just a convention for how to specify inputs, usually global inputs for an entire application or portion of an application. The main benefit of using config files is that the approach pulls parameters out of code files and puts them into a centralized place, making them easier to find and update. It also gives you a more straightforward way to control user permissions and allow some users to update configuration options but not source code. There are a few standards for configuration file syntax, but the author has favored using Linux Bash script syntax so that they are easy to use from Bash scripts and can be shared between coding languages within the same application.

Consider having a main.cfg file in the root directory of your application, then if there is a need, adding additional configuration files into the /etc directory that correspond to various functional parts of your application. A main.cfg might have something like this:

```
# Application name
app_name =”Sales data processing”
db_engine=Oracle
db_name=oracle_sales
db_user=oracle_uid
log_dir=/log/sales
```

Once parameters like this are added to a config file, your programs need some way to access them. Here is a sample macro that parses configuration files into global macro variables:

```
%macro config_read(file);
    FILENAME _config "&file";
    DATA _null_;
        LENGTH name $32 value $32767;
        INFILE _config LRECL=32767 TRUNCOVER;
        INPUT value $32767.;
        IF SUBSTR(value,1,1)^='#' and INDEX(value,'=') AND
            NVALID(SCAN(value,1,'=')) THEN DO;
            name = LOWCASE(SCAN(value,1,'='));
            value = LEFT(TRIM(SUBSTR(value,INDEX(value,'=')+1)));
            IF SUBSTR(value,1,1) IN ('"',"'") THEN value = DEQUOTE(value);
            CALL SYMPUTX(name,TRIM(value));
        END;
    RUN;
%mend;
```

LOOPING AND AUTOMATION TECHNIQUES

So far the need for writing concise, commented, and well-organized code has been emphasized. One important way to improve programs in this way is to write table-driven code that uses input files to drive loops. Not only can this reduce the amount and complexity of code, but the table files can be moved into a directory like `in` (suggested above) so that their location and maintainability is clearer. For example, let’s say that you needed to read the following files into SAS data sets:

```
data sales;
    length mfg desc model $100 price 8;
    infile "/data/sales.csv" truncover dsd;
    input mfg desc model price;
```

```
run;
proc sort data=sales;
    by mfg model;
run;
data inventory;
    length mfg desc model $100 qty 8;
    infile "/data/inventory.csv" truncover dsd;
    input mfg desc model qty;
run;
proc sort data=inventory;
    by mfg model;
run;
data returns;
    length model_id $100 returns 8;
    infile "/data/returns.csv" truncover dsd;
    input model_id returns;
run;
proc sort data=returns;
    by model_id;
run;
```

USING MACROS TO RE-USE CODE

A pattern should be self-evident in this code. For each of three different data sources, a csv files is read and a sort is performed. For each source, the length statement, input statement (really just length statement without the length qualifiers), file name, and sort by values change. The code can be simplified like this:

```
%macro process_data_source(filename,length,sortby);
    %let input=;
    %let i=1;
    %do %while (%scan(&length,&i,%str( ))^=);
        %* CHECK EACH WORD AND ONLY USE VARIABLE NAMES *;
        %if ^%index($0123456789,%substr(%scan(&length,&i,%str( )),1,1))
            %then %let input=&input %scan(&length,&i,%str( ));
        %let i=%eval(&i+1);
    %end;
    data &filename;
        length &length;
        infile "/data/&filename..csv" truncover dsd;
        input &input;
    run;
    proc sort data=&filename;
        by &sortby;
    run;
%mend;
%process_data_source(sales,mfg desc model $100 price 8,mfg model);
%process_data_source(inventory,mfg desc model $100 qty 8,mfg model);
%process_data_source(returns,model_id $100 returns 8,model_id);
```

TABLE-DRIVEN CODE

To make this code table driven, a file named in/data_sources.csv can be created:

```
filename,length,sortby
sales,mfg desc model $100 price 8,mfg model
inventory,mfg desc model $100 qty 8,mfg model
```

```
returns,model_id $100 returns 8,model_id
```

Then, instead of the three macro calls above, we would use this code to read in each line of the file and execute our macro using the parameters in the file:

```
data _null_;
    infile "in/data_sources.csv" truncover dsd;
    length filename length sortby $200;
    input filename length sortby;
    call execute('%nrstr(%%)process_data_source(' || strip(filename) ||
                 ',' || strip(length) || ',' || strip(sortby) || ');');
run;
```

FILE-DRIVEN CODE

In addition to using tables (like *.csv files), we can also execute macros or %include programs once per file which exists in a directory. Let's say we have a directory of csv files to read into SAS data sets in the `in/static` directory. To call a macro once for each csv file in this directory, we can use the following code:

```
data _null_;
    length filename $200;
    rc = filename('_dir','in/static');
    did = dopen('_dir');
    do i = 1 to dnum(did);
        filename = dread(did,i);
        if index(filename,'.csv') then
            call execute('%nrstr(%%)read_file(' || strip(filename) ||
                         ');');
    end;
    did = dclose(did);
run;
```

ESSENTIAL LOOPING TECHNIQUES

Three techniques are worth understanding in order to build concise code that uses loops to reduce redundancy. One of these has already been demonstrated in the above examples but without explanation.

Call Execute

Inside of a data step, call execute('code') can be utilized to queue up Base SAS statements that will execute after the data step finishes. Since the data step is built for looping, this technique is useful to read files or data sets and then generate code (macro calls or direct code) using input values. The reason that %nrstr is being used is to "double" mask the percent sign so that it is not resolved until after the data step executes and the statements are actually submitted. Any macro statements generated by a call execute will run immediately (meaning before the data step finishes), which could produce unexpected results. In general, if you are doing looping like we are demonstrating in these examples, you will probably want to mask the percent sign so that during data step compilation and execution the macro name is not resolved.

Macro Arrays

You can also loop over lists by using macro arrays, though this gets messier when more than one dimension is needed. In the case of our list of csv files, you could generate an array like this:

```
data _null_;
    length filename $200;
```

```
        rc = filename('_dir','in/static');
        did = dopen('_dir');
        n = 0;
        do i = 1 to dnum(did);
            filename = dread(did,i);
            if index(filename,'.csv') then do;
                n = n + 1;
                call symput('file'||compress(put(n,8.)),strip(filename));
            end;
        end;
        did = dclose(did);
        call symput('num_files',compress(put(n,8.));
    run;
    %do i = 1 %to &num_files;
        %read_file(&&file&i);
    %end;
```

Consult documentation on PROC SQL's :INTO statement as well, which provides a handy way to load values into a macro variable or array of variables from a SAS data set or database table via SAS/Access.

DOCUMENTATION

Documentation is an important part of creating applications that are understandable as a whole. If the techniques in this paper are followed, much vital information about the application's design and code will be evident without additional documentation. The invaluable information that documentation can provide which code and file structure cannot, however, is the overall architectural design, business objectives, or history of the application. It is important not to neglect these topics in order to give developers a holistic view of what they are working towards achieving. The author recommends that the following information be captured in a documentation system:

- Development team members
- Application history
- User requirements (may be supplied by other groups)
- Business objectives
- Components
- Key executable scripts
- Inputs from other applications
- Outputs to other applications
- Test scripts with instructions

VERSION CONTROL

Version control is an essential part of modern computer programming. If implemented properly, version control systems will provide not simply backup and recovery/restore options, but also manage releases and provide a historical log of how an application has grown and changed over time. In every version control system (VCS), when changes are committed or checked in, a central database is maintained with the changes, the change notes (often required) and the user. Thus, reports can be generated and exact changes (often called DIFFs because of the Unix tool "diff") viewed. This knowledge also helps programmers understand, debug, or extend existing code, as they can see what changes might have been implemented for specific purposes and see an angle of the code that the organization, structure, or even documentation do not provide. SAS does not provide built-in version control for sources files, so an external tool must be used.

VCS VERSUS DVCS

When choosing a version control system for your application, the first consideration is whether to use a *centralized* VCS (just referred to as VCS) or a *distributed* VCS (DVCS). A centralized system like the old CVS or Subversion uses a single master repository into which files or file changes are committed or checked in. One disadvantage of this model is that if the central server is down, no one can commit changes or track their own changes. Another disadvantage is that all version control happens in the shared environment of the server, making it less suitable for individual developers to track their own changes at a finer level. This was the main reason that DVCS systems were created – to give each programmer their own repository into which they can commit changes all day long, then at some point when they are ready, can shared their changes with the main repository and then other programmers. The most popular DVCS systems are Git and Mercurial. One disadvantage to DVCS's is their complexity; they typically have a much higher learning curve.

RELEASES

Another important consideration that version control systems provide is the ability to track releases. A release is a snapshot of an entire application which is usually promoted from a development environment into a test environment and then eventually to a production or deployed environment. Without a version control system it is very difficult to keep track of releases and easily obtain past revisions of files or entire directory structures for testing and debugging.

CONCLUSION

After reading through this paper you might be tempted to wonder if all of this attention to syntax, directory design, modularity, table-driven code, documentation and version control are all worth it. For most programmers, these activities can feel like chores that must be done to satisfy management – chores that get in the way of what we really enjoy, writing code. You might be tempted to think that you can manage the complexity of your growing application by the seat of your pants. Most applications start with one or two people, and over time the requirements, scope, and team grow. New team members join who do not have the same skill level and who start to make gradual changes which increase the chaos. And then one day your application will break, and break in such a glorious fashion that it will take you untold hours of anguish to pull out and re-write portions just to get to the point where you can debug it.

By disciplining yourself and your team to start implementing many of these architectural considerations now rather than waiting until it is too late, you will save yourself grief and your company resources. And it is the author's hope that you might come to see this type of work as not merely a discipline but as a rewarding part of the software development process itself.

CONTACT INFORMATION

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

David L. Ward
dward@axomsoftware.com