Beginning Macro Design Issues

Toby Dunn, Idea Integration, San Antonio, Tx
Ian Whitlock, Whitlock Consulting, Kenneth Square, Pa

ABSTRACT

Macro!!! Why does this word strike fear into even the most battle hardened SAS® programmers? The word macro conjures up visions of complex and convoluted programming which leads to hours of confusing debugging and mind-twisting logical problem solving. Does macro programming have to be this hard? No, it does not. With a little forethought and some careful planning, practically anyone can follow a few simple design principles and turn a complex and seemingly confusing macro problem into a simple, easy-to-follow macro. Although this paper is not a comprehensive discussion of macro design issues, the instructions provided will give the reader a basic foundation on which to start writing clear and concise macros. While programmers of all levels can learn something from this paper, the beginner and intermediate skilled SAS programmers are the intended audience.

INTRODUCTION

What exactly is Macro Design? What constitutes a good design? The answers to these questions require the programmer to consider the many users of the macro as well as the many issues that these users raise. Design is everywhere in our daily lives, from the tallest skyscrapers to the computer used to write this paper. Things that are well designed provide flexibility and robustness while still retaining the quality of being easy-to-use. It is natural for most people to avoid giving design a second thought. It is not until we encounter programming that is badly designed that we take notice and formulate a few choice words about the design and the creator. The importance of design to the SAS programmer can be found by looking no further than in the macros they write. The abstract nature of macros and their use by multiple users necessitates that macros be designed well. If designed poorly, they can cause not only delays but wasted time and money.

MACRO DESIGN

Macro Design, in its simplest form, is the process of applying a set of best practice principles to balance the quality characteristics of a macro. These characteristics fall into two broad categories of User and Programmer. User characteristics include correctness, usability, efficiency, reliability, integrity, and adaptability. The Programmer's characteristics are accuracy, robustness, maintainability, flexibility, portability, reusability, readability, testability, and understandability. While many of these characteristics compliment one another, others may contradict. An example would be to increase the flexibility of the program, increases the complexity to a point which decreases the usability. It is

left to the programmer to find the correct balance between these characteristics which best satisfies all of audiences of the macro.

Jack Reeves once said “the code is the design.” This is only partially true as design can be both a verb and a noun. Thus macro design can be both something you actively do and something you get from doing the activity. The process by which a macro is designed can be as important as the final design of the code. The general process is as follows:

1.) Define the problem to be resolved.
2.) Create a solution to the problem.
3.) Implement the solution.
4.) Verify results.
5.) Move forward if the solution is correct, else start over, and
6.) Reflect upon the solution(s) for future use.

Looking at the above steps it can be said that Macro Design is heuristic in nature. That is to say it often takes multiple attempts to get the best design possible under your current working conditions. Good design is created through hard-won experience and lessons learned from other experienced programmers.

GETTING THE PROBLEM RIGHT

During World War II the British realized that many of their bombers where shot down. These planes were grounded not from losing a wing or some other drastic issue, but rather from bullet holes in the fuel tank. Meanwhile, German planes did not suffer this fate, as an investigation revealed that German bombers had self-sealing fuel tanks. The British quickly decided they needed to do something to improve their fuel tanks. British scientist R. V Jones investigated and found that Britain had actually developed a bullet-proof fuel tank at the end of World War I, but never implemented it because someone decided that the fuel tank should not only be bullet proof but crash proof, since crashes where the major cause of death to pilots. The British were never able to develop a fuel tank which was both bullet proof and crash proof, so the whole concept was abandoned. Along these lines another scientist proposed the idea that plate glass is the best material to use for motorcycle crash helmets. It does not take a rocket scientist to deduce that although plate glass is great for visibility and would do a good job at keeping the elements and the bugs off riders, it would not fare so well when the rider crashed and needed protection.

You may be wondering what bullet proof/crash proof fuel tanks and plate glass motorcycle helmets have to do with macros. These are perfect examples of improperly defining the problem. Macros perform best when they do one thing and consistently do that one thing well. When the problem is incorrectly identified, the macro may attempt to do too much, leading to the macro becoming overly complicated and inflexible. This scenario is much like the fuel tank

example; the real problem was the need for a bullet proof fuel tank. Adding the necessity to be crash proof over complicated the problem and the project was dropped when a solution could not be found. Failure to correctly identify a problem can lead to a macro providing the incorrect solution, similar to the crash helmet example. While the plate glass helmet does provide great visibility, it fails to protect the rider when a crash occurs.

Data structure has a strong influence on the nature of the macro design. Often, a change in the data structure is a far better solution than writing a macro to preserve it. For example, if the data variable names contain important information, it is quite likely that the macro code will need to extract and manipulate the information in code. To be specific, suppose we have monthly expenses for some entity stored in variables EXP_FEB2003, EXP_MAR2003, … , EXP_SEP2006. Each month a new variable is added to the file with a new name. Each quarter we need a program to search for extreme expenses, etc. Arrays and macros with parameters will make these tasks less repetitious, but consider how much easier it would be with the variables EXPENSE, MONTH, and YEAR in a long narrow data structure. Now you can add a new variable by simply concatenating a new data set, and the extreme values can be found with standard procedures like MEANS or UNIVARIATE.

Given the thousand of possibilities that can be factored into your definition of the problem, we can not conjure an exact check list of questions to ask yourself. We can offer some general processes to follow in order to help you to accurately define the problem. When contemplating how to solve the problem, forget about any attempts to design specific solutions. Those can be thought about in the next step. For now, just formulate what are the specific issues which cause the problem. Look at it from different angles and perspectives. Try looking at it from the user's perspective and then from the programmer's. You will find each view requires thinking about the problem a little differently. At first this process is slow, but the more you practice it the more you will find that you can identify the problem.

Let us look at an example of how to define a problem. Assume you have 50 datasets named Data1 through Data50, each data set is one month's worth of data. You need to combine these data sets into one dataset to do statistical work. Given that there are quite a few datasets that need to be combined, it is easy to make a mistake if done manually, therefore a macro could improve the readability and reliably of the code.

Here are some questions to consider:

Is the problem that the data should not be structured in this fashion?
Is the immediate problem to create the data the most important issue?
Is the immediate problem to create the list of data sets more important?
Is compiling a general list more important than this specific list?

Since the programmer rarely has control over how the data is delivered, we will dispense with the first question noting that programming problems are often created by previous decisions and perhaps the program should have been of the form:

```
Data Temp ;
...
Run ;

Proc Append
  Base = Need
  Data = Temp ;
Run ;
```

If the answer to the second question is yes, then the programmer might write:

```
%Macro CatData ;
%Local I ;

Data Need ;
Set
%Do I = 1 %To 50 ;
  Data&I
%End ;
;
Run ;
%Mend CatData ;

%CatData
```

While this code solves the immediate problem, the macro needs modification for the slightest changes in the scope of the problem, i.e. the macro lacks the qualities of reusability and maintenance. Developing macro code is hard enough, and it is worth paying a lot of attention to these qualities. For example, what would happen if later on you would need the program to loop through 20 datasets, or datasets 20 through 36? What if the library where the data sets are stored changes, what would happen if a different root is needed to name the input data sets? The macro would have to be rewritten. Simply put, the macro's code is tied too closely to the specific requirements and therefore does not provide the flexibility required to handle any of the suggestions raised.

```
%Macro   Cat   (Lib=,   Mem=   ,   Out=   ,   Start=1,   Num=1   )   ;
%Local                              I                                ;

Data &Out ;
Set
%Do        I        =        &Start        %To        &Num        ;
                                                      &lib..&mem&I
%End                                                              ;
```

```
;
Run ;

%Mend Cat ;


%Cat( Lib = work , Mem= Data , Out = one, Num = 50 ) ;
```

In this example, the parameters provide the flexibility to overcome many of the previous objections, but it does not solve the reusability question in a completely satisfactory manner. Typically, the user of this code will want not only to concatenate data sets, but will also have other requirements such as the creation of some new variables. Now, each re-use requires new modifications to the macro that has little to do with the basic concatenation problem. The issue here is the macro attempts to control too much because it did much more than attack the basic problem of making the list of input data sets.

One could conclude that it is better to decouple the immediate data problem from building the list of data set names needed in a SET statement. This means more responsibilities for the user, but better control.

```
%Macro   Cat   (Lib=,   Mem=   ,   Start=   1,   Num=   1   )   ;
%Local                              I                              ;

%Do        I        =        &Start        %To        &Num        ;
                                                      &lib..&mem&I
%End                                                               ;

%Mend Cat ;

Data One ;
Set %Cat( Lib = Work , Mem= Data , Num = 50 ) ;
Run ;
```

The example above solves the dilemma without being so closely tied to the immediate problem, but it still limits list making ability for SAS dataset names. This limits the reusability of the macro code for anything else. The problem here can be summed as too specific to a particular issue, rather than to a more general problem. Of course, there are times when it is wise to avoid coding for a more general case, but generalization actually simplifies the usage and makes clearer the usage intent.

The problem can be defined as the need for a macro to create a list of values with a constant prefix and ending suffix which ends in a number; the range of this number as well as the prefix determined by the user.

```
%Macro   Enum(   Pref=   ,   Start   =   1   ,   Stop=   1)   ;
%Local I  ;

%Do       I       =       &Start       %To       &Stop       ;
     &Pref&I
%End ;

%Mend  Enum ;

Data One ;
Set %Enum( Pref = Work.Data, Stop = 1 ) ;
Run ;
```

This example still does not provide the most general list maker, however this solution does solve all of the problems in the previous examples and presents a simple solution. This code is general enough that if a requirement changes, it could be altered without modifying the inner workings of the macro and does not place too much of a burden on the user. The code is clearer because user issues can now be part of the code rather than in the province of the macro author. If the user's task is considered too burdensome, then this problem should be attacked in another macro, i.e. the user's other problems should not be confused with the list making problem.

This example still does not provide the most general solution to the list making problem, but to go further would take us too far from the original issue and would be out of the scope of this paper. As illustrated, getting the problem right is not always the first thing thought of. Spend some time getting this right, if you have to ask someone else for their opinion, create pseudo code to test out your interpretation of the problem. If you do not get the problem right on the first try, do not fret; the important thing is that you are able to recognize as soon as possible that the problem you are trying to solve is not the correct one so that you can revisit and solve the right problem.

USER INTERACTION

Consider the following paragraph interpretations from The Design of Everyday Things…"I visited a home in England that had a fancy new Italian washer-drier combination with multiple controls to do everything you ever wanted to do with the washing and drying of clothes. The husband (an engineering psychologist) said he refused to go near it. The wife (a physician) said she simply memorized one setting and tried to ignore the rest.

I read the instruction manual, thinking someone went to a lot of trouble to create that design. This machine took into account everything about today's wide variety of synthetic and natural fabrics. The designers worked hard; they really cared. But obviously they had never thought of trying it out, or watching someone using it."

It is obvious from this story that user interaction matters. For SAS Macros, this interaction usually takes place with the user specifying the parameter values. Far too often the parameters are cryptically named, nonexistent, something totally different than what the user expects, or just plain redundant. It matters not how much care was taken by the developer of the macro to ensure the efficiency of the macro's inner workings, nor the elegance of the macros, if the user can not figure out how to use it.

To insure that the user can use a macro, consider parameter mapping and affordance. Mapping refers to two parameters, Keyword vs. Positional, and the order in which the parameters are called in the macro. Affordance is concerned with the naming scheme and how that reflects some sense of meaning to the user.

Should keywords or positional parameters be used? To demonstrate this need, have someone read the following numbers to you: 1, 2, 7, 24, 5, and 91. Wait five minutes and recite those numbers from memory. This is not too hard to do, but what if I asked you what you had for lunch three weeks ago? Ah…, now that takes considerably more effort. What if the time period was extended to three months, now the matter becomes virtually impossible. The human mind handles short term memory much better than long term memory and by using positional parameters the macro designer is asking the user to remember which parameter goes where in a macro call. As the time period grows from one use of the macro to the next, this information becomes harder and harder to remember. Using keyword parameters eliminates this problem. If the user needs to recall which parameter goes where, they merely have to look at one of the programs they last used it in. The readability and understandability of the code increases because now anyone can clearly see what values where passed to what parameters in the macro. This also helps anyone during the macro debugging process.

When creating a macro, keyword parameters are always superior to positional parameters because they help the reader see the role played by the value given. Keyword parameters are the only way in SAS to pass a true default value. If default values are well chosen then macro will be simple to use while providing user control when necessary. While positional parameters can contain the null default value, they can not hold a general value without changing the parameter or using a substitute variable. This makes it harder for the macro reader to understand, since the intended default is buried in the code rather than in the

%MACRO statement. Positional parameters require the user to remember where and what that parameter is. For example, consider:

```
%Tough(,,a,,,,b,,c)
```

Is A the name of the input dataset or the output dataset? Is B the By-variable or the ID-variable? Nothing in the code helps to answer these questions; the reader

needs intimate knowledge of both the macro code and the consuming code. Compare this issue with:

%Report(Out = a , Id = b , Errors = c)

You may not know the name of the input dataset since a default was used, but it can easily be determined from the %MACRO statement. Moreover, if the defaults were wisely chosen it is either known to everyone connected with the project, or it is known to all SAS programmers as the last created dataset.

Parameter names should describe the value that is being passed to the macro. They should not be too long or too short, and remember to be consistent with the naming scheme within, as well as between macros. Names such as In and Out and are too vague and should be expanded to DataIn and DataOut. Names such as BankXXXMonthlyDataIn and BankXXXMonthlyDataOut are too long and specific and should be shortened to MonthlyDataIn and MonthlyDataOut.

The position of the parameters should be considered, in addition to using proper names. In a complex macro it helps to organize the parameters into groups. For example, it is common to have all input parameters before all output parameters. Adding such a structure to the parameters helps both the reader of the macro and the reader of the code. If such structure is attempted with positional parameters then adding new parameters requires you to break the implied structure or to break existing code. Hence, positional parameters not only make it hard to see what the code means, they also make it hard to extend the functionality of an existing macro. Essentially, the only good reason for using positional parameters is because the usage is intended to mimic a function call. Even here, it may be wise to break with tradition.

Another mapping issue is the relative order in which the parameters are used in the macro call. Suppose someone was to give you driving instructions to a restaurant. However, the instructions are not in order. Finding how to get to the restaurant would be rather difficult, if at all possible. Ordering the parameters such that they match the flow of the program gives the user and reader a sense of order and can help with understanding what values these parameters are supposed to have. The most common order is parameters that specify input only values such as an input dataset, next are those that specify values that are input-output, and finally output-only values.

Remember, part of the purpose in writing macro is to allow a user to gain the service of your macro while knowing nothing about the internal workings of the macro. Even when you are the user this means easier coding and debugging because you can concentrate on the current code and the current point of view.
Using keyword parameters, naming the parameters well, and watching the placement of parameters in the macro call in conjunction with the macro's name will allow the user to understand what the macro does and provides a hint as to

how it does it. An important, although not well documented fact, is you can always use keyword parameters in calling a macro even when the macro author did not write the %MACRO statement that way.

LOCAL VS GLOBAL MACRO VARIABLES

One of the first steps I learned in the macro language was to create a global macro variable with a %let statement. The power that I could generate with such a simple little statement was awe inspiring, so much so that I used global macro variables extensively in most of my programs. I had noticed many experienced programs avoided using global macros but I did not understand why. It was not until a sunny afternoon when my boss asked me to debug a program heavily laden with poorly written macros that the reason others avoided global macro variables began to sink in.

The problem with this program I had to debug was that it would not stop running. I spent hours reviewing the code, but the fact that the program would not complete made debugging it hard, as I was not able to study a complete log file. Finally, I manually terminated the program and discovered that it was caught in an endless loop. This loop was the result of one macro calling upon another macro in a %do-loop where the called macro performed another %do-loop with the same index variables names declared as a global macro variable. Since the called macro %do-loop terminated with a value less than the stop condition of the calling macro %do-loop the program never stopped and caused an endless loop. It was at this time I realized that one should use global macro variables with caution.

When a macro is executed it creates its own environment for storing local variables. Any variables outside of that execution are independent of those variables and do not change or clash with variables inside the macro. However, any call to a macro can include the variables created outside the macro environment. This means a macro can protect itself from outside influence by declaring inner variables local with a %LOCAL statement, but it can not protect itself from the macros that it invokes. Consequently, each macro author has a responsibility to declare all macro variables. The SAS system also sets up an environment that can be accessed from anywhere. These variables are called global variables. If they must be used inside a macro then it is best that they be declared in that macro with a %GLOBAL statement so that the user understands

that either the source of this variable lies outside the macro or the macro is creating this variable for use by an outside agent.

While there are many arguments as to why global macro variables should be avoided, for the sake of brevity we will review three reasons; code comprehension, code reuse, and inadvertent value changes. When a macro uses a global macro variable that must be assigned by the user, it increases the

amount of information required to use the macro. Even if the user need not assign the value, they must be aware of the name of the variable in order to prevent inadvertently using that name. Local variables hide the information from the caller and protect them from the need to know about the macro's internal code.

Another problem with using global macro variables is it makes the code difficult to reuse. Let's assume a macro requires the user to assign a global macro variable named X. Anyone who uses that macro must know the name X and avoid using that name for any other purpose. It also makes the consuming code difficult to debug because X might be assigned anywhere in the preceding code. Such variables are essentially acting as parameters for the macro. Thus the function should be assigned to a parameter with a meaningful name. Now the assignment can be made in only one place, at the point of macro invocation.

```
%Macro PrintIt ;
    Proc Print
       Data = &MyDataSet ;
    Run ;
%Mend PrintIt ;
```

Looking at the above example it can be seen that the user would need to know that the macro uses the global macro variables MyDataSet. This limits the macro use and the user must remember that they have to create a global macro variable before they call the macro PrintIt. While this might not be too much of a hardship it definitely makes using the macro more difficult, especially if the user already has a macro variable with the same name. As mentioned before a better way to pass this information to the macro is through a parameter.

```
%Macro PrintIt( DataSetIn= ) ;
   Proc Print
      Data = &DataSetin ;
   Run ;
%Mend PrintIt ;
```

Finally, when a global macro variable is utilized, the user, not the macro author, must be responsible for the integrity of that variable, i.e. that the value contained in the global macro variable does not change between the time when the value is assigned and when the macro is invoked. If the value is altered on purpose, it makes the code more complex, since a larger chunk of the program must be considered in order to insure correct execution of the macro. If the value is altered inadvertently, then it simply causes the code to fail.

```
%Let InputDataSet = Data1 ;

< More SAS Code >
```

```
Data _Null_ ;
Set DataSetNames ;
  Call SymputX( 'InputDataSet' , DSN ) ;
Run ;

%PrintIt
```

To summarize, most experienced macro writers have discovered that the problems with global macro variable usage do not outweigh the benefits. If you find you are in need of global data, then reconsider your approach. One common usage for global macro variables is to pass information to the macro, but this should be done via parameters. The other common need arises when one macro must create variables for another macro. In this case, you should consider a redesign where the first macro calls the second macro, or a third macro is introduced to manage the whole process and provide the environment for the required variables.

If, after serious consideration, you still think you require a global macro variable, then you probably need a parameter for the whole program rather than the macro. To insist that the macro must use this parameter means that the macro's usage will be restricted to programs that require those parameters. In this case, it is better to isolate the macro from such usage. For example, LIBREFs are often treated as program parameters. Suppose you have a program that needs a flexible LIBREF, then you might use this statement at the top of the program:

```
%Let PgmLib = MainLib ;
```

This does not mean that any called macro must know about this convention for this program! Instead you might try the following code:

```
%GetData( Lib = &PgmLib ) ;
```

In this case the user must know the global parameter, PGMLIB, but the macro author does not need to know and should not need to know about this program parameter.

If your desire to make a variable global is not based on the fact that it is really acting as a program parameter, then it is time to consider whether the information would not be better conveyed by data stored in a SAS dataset or some other form of data storage. One common argument for global variables is that since there are so many of them that it is inconvenient to list them all in parameters. But this is precisely the time when a dataset can offer better integrity for handling the information.

NAMING CONVENTION

A naming convention can save or hurt you. A good naming convention is quite possibly the cheapest, simplest, thing a programmer can do to increase code readability. Naming conventions that are too stringent can decrease the creativity of the programmer; those not stringent enough can be useless. The best naming convention strikes a delicate balance between the two extremes. If you do not have a naming convention then argue for adopting one and at the very least adopt one of your own and remain consistent with it.

In regards to macros, naming conventions have to focus on different sets of names, macro names, parameter names, and macro variable names. Since macros execute an action such as creating a report or generating value(s), they should begin with a strong verb like 'get' or 'create' and then be followed by a noun like 'xxxreport' or 'xyzdata'. A bad example of a macro name would be %Report or %GetData, as they are too vague to do any good. A better name would be %CreateQrtlyReport or %Get95thPctl.

Macro variables and parameters hold specific values and should therefore have names that are descriptive of the value they contain. a name such as NumberOfElementsInQue is descriptive but too verbose. A name such as CampaignQue is better because it is shorter and still remains descriptive. If a name is not descriptive enough it may be misinterpreted; if the name is too long it may cause a programmer to give up using it or simply mistype it. Finally, many programmers find it helpful to prefix macro variables to designate their scope. All global macro variables must begin with a G or Global while local macro variables start with L or Local. This helps to avoid confusion about the scope of the macro variable.

TRASH COLLECTING

All macros should clean up any values or steps created by the macro for internal use. This can include setting options, making dataset variables, and datasets. This clean up process should not be left to the macro user. Failure to remove working datasets can cause undue stress on the machine running it, or cause damage to the program or SAS session with unanticipated side-affects. Failure to clean up may cause problems for the macro if it is repeatedly called. For example, a macro could use a dataset from a previous execution when in fact it should not. If a macro opens a dataset and does not close it, then a following execution of the macro or another part of the program may not be able to access that file.

```
%Macro GlimMix( DataIn = , DependentVars = , ModelName= ) ;

ODS Output ParameterEstimates = __Parm CovParms = __Cov ;

Proc Glimmix
  Data = &DataIn ;
  Where Age15 = 1 ;
```

```
  Class State ;
  Model Dvexvgg( Event = '0' ) = &DependentVars
                               /  Solution
                                  CovB
                                  Link = Logit
                                  Dist = Binomial
                                  OddsRatios ;
  Random Int / Sub = State ;
  Random _Residual_ ;
  Nloptions Tech = nrridg ;
Run ;
Quit ;


Data __OR ( Keep = Effect OR LOR UOR Model ) ;
Length Model $200 Effect $32 ;
set __Parm ;

Model = "&ModelName" ;

OR  = Exp( Estimate ) ;
LOR = Exp( Estimate - ( 1.96 * StdErr ) ) ;
UOR = Exp( Estimate + ( 1.96 * StdErr ) ) ;

Run ;


Proc Append
  Base = Parameters
    Data = __OR Force ;
run ;


Proc Append
  Base = Covariates
Data = __Cov Force ;
run ;


/*************************/
/** Clean Up Work Space **/
/*************************/

Proc DataSets NoList ;
  Delete __Parm __Cov __OR ;
Quit ;

%Mend GlimMix ;
```

The macro shown above is designed to be called multiple times and it can be surrounded by an ODS destination. If the proc datasets statement was not

included in the macro and one of the calls failed because one of the dependent variables was not found in the input dataset, then the macro would append the dataset from the last execution of the macro to datasets Parameters and Covariates.

CODE DOCUMENTATION

Code documentation, like a good naming convention, is an easy way to reduce the effort needed to maintain and use a macro. However, most programmers only give lip service to this process. Many programmers refuse to document their code, stating that the code speaks for itself or that they simply do not have the time to document. Code can reveal what is done only by inference, and often that is not enough. Statistical and technical macros may need to explain why a certain method was chosen over another. Documentation is the subject of numerous papers and books, so we will only cover the basics of commenting code. Comments fall in two forms - a header at the beginning of the macro to explain its purpose, usage, and methods, and more detailed comments about specific code segments.

Just like programs, macros deserve header information. The bare minimum of information that header information should contain is the program name, parameters and valid values, output produced, a general description of what the macro does, the date the program was created, and who created the macro. In addition, many programmers like to include sample data that proves the macro runs successfully.

```
/*************************************************************/
/** Macro Name : GlimMix                                    **/
/**                                                         **/
/** Parameters : DataIn       ~ Input Data Set name.        **/
/**                                                         **/
/**              DepedentVars  ~ A space separated list of  **/
/**                                Dependent Variables to be  **/
/**                              used in the GlimMix model. **/
/**                                                         **/
/**              ModelName    ~ The value that will        **/
/**                             designate the model the     **/
/**                             covariate values come       **/
/**                              From in the                  **/
/**                            Covariate.sas7bdat           **/
/**                            output data data set.        **/
/**                                                         **/
```

```
/** Outputs     : Parameters.sas7bdat data set which          **/
/**               contains the parameter estimates.          **/
/**                                                          **/
/**              Covariate.sas7bdat data set which contains **/
/**               the covariate information from the GlimMix **/
/**               model                                      **/
/**                                                          **/
/** Description: Runs a Proc GlimMix procedure, captures the**/
/**              parameter estimates and covariate measures **/
/**                                                          **/
/**              It then appends each of these these        **/
/**               together into two output data sets         **/
/**               Parameters and Covariates.                 **/
/**                                                          **/
/** Created By : Toby Dunn                                   **/
/** Created On : May 05, 2006                                **/
/**************************************************************/
```

Code comments are one of the most misunderstood aspects of documentation. Many programmers think that they should describe what the code is doing and therefore make the code easier to read. This clutters the macro with useless data and makes the code harder to read. Comments should explain why action occurs, why it was done this way and not another way, and rarely should explain a tricky hunk of code. A person reading

the macro should be familiar enough with the syntax to understand what the code does so explaining this in a comment is worthless. What the code cannot explain is why this procedure is done or why this method or procedure was chosen over some other one. In the rare case that a hunk of code cannot be made any simpler and is still tricky, then comments should help explain how the code achieves its objective. Finally comments should be in full complete sentences and should not (unless in the rare case of tricky code) refer to syntax.

An example of bad commenting:

```
/**********************/
/** Sort The Dataset **/
/**********************/
Proc Sort
Data = ABC ;
By Descending PValue ;
Run ;

/********************/
/** Print Five Obs **/
/********************/
```

All of the comments in the above code can be determined from the code itself. What the comments fail to do is inform the reader what is actually being done and why.
Another example of commenting:

```
/*************************************/
/**  Produce a Print Out Of The Top **/
/**  Five Pvalues For Final Report. **/
/*************************************/
Proc Sort
Data = ABC ;
By Descending PValue ;
Run ;

Proc Print
Data = ABC ( Obs = 5 ) ;
Run ;
```

This comment conveys more meaning to the code as a whole and requires substantially less typing.

CONSISTENCY

Writing code consistently the same way each time will increase the readability, understandability, and maintainability. It helps relieve some of the burden of the reader to understand a multitude of ways to solve a problem within one macro while not leaving them guessing as to why this method was chosen this time and another method was chosen another time. It also decreases the chances of making a mistake. The more ways a solution to a problem is used the more ways there are for those solutions to go wrong. A simple example of bad consistency is in the indentation style of the code. Consider multiple SQL statements:

```
Proc Sql ;
Create table New as
Select *
From Old ;
Quit ;

Proc Sql ;
Create table New as
  Select *
    From Old ;
Quit ;

Proc Sql ;
Create table New as Select * From Old ; Quit ;

Proc Sql ;
Create table New as Select *
```

```
From Old ;
Quit ;
```

Another example is using parentheses:

```
%let Percent = %Sysevalf(  (&Numerator / &Denominator ) * 100 ) ;

%let Percent = %Sysevalf(  &Numerator / &Denominator  * 100 ) ;
```

All of these will create the exact same dataset labeled New, but when they are all mixed up in the macro code it becomes hard to focus on what the code is accomplishing. The proper use of white space, indentation, or parentheses works on a subconscious level to help reader to better understand the code.

REFACTOR

After the macro is written it is important that the programmer review the macro to look for possible errors, ensure that the code is simple and clear as possible, that the macro addresses the correct problem at the right level, and that the macro provides flexibility and control for the user. Often in the rush to get the final result out the door this step is skipped entirely. Unfortunately, this results in less than optimal code turned into production code. In addition, this review step is important because it provides time to review and learn design patterns for developing future macros.

When Refactoring macro code, ensure that the code has both simplicity and clarity. Do not confuse simplicity and clarity, as they deal with two very different aspects. Simplicity refers to the functionality of the code while clarity is concerned with code readability.
The principles discussed in this paper are designed to help you achieve these goals. The macro development process can be confusing when attempting different approaches; requirements can change halfway through the build process, and programmers often do not see the overall big picture of the macro until the macro is created.

After your macro is written, go back and specifically look for confusing chunks of code and places that can be combined or eliminated. Any code that is slightly confusing to the writer of the macro will be twice as tricky to someone else reading the code. If a complex hunk of code is revealed, then think of another way to write the code and replace it. If there is not a simpler way to rewrite the code then double check that the comments exactly describe what process is happening.

Once simplicity of the code is confirmed, review the code again to make sure it is clear in its intent. Look at the code as a whole to make sure anyone reading the code will understand where the macro starts and ends. If the flow of the macro is

muddled up, think about ways that the code can be rearranged so that it flows as close to a linear line as possible. Then look at individual hunks of code making sure that the comments and the code work together to clearly state what is happening at any given point.
Once you have mastered the initial design principles of macro writing, the last step in the review process is to search for macro design patterns. Design patterns are common processes used in macros. Two common design patterns are list processing or text substitution macros. The more of these you recognize, the faster you will be able to spot common problems and create reliable solutions.

```
%Macro OutputVal( Template= , VarList= ) ;
%Local Stop I ;

%Let Stop = %Eval( %Sysfunc( CountC( &VarList , %Str( ) ) + (
%Length( &VarList ) > 0 ) ) ;

%Do I = 1 %To &Stop ;
    %Sysfunc( TranWrd( &Template , Var , %scan( &VarList , &I ) )
)
%End ;

%Mend OutputVal ;
```

CONCLUSION

This paper has demonstrated some of the basic principles of Macro Design. Just like addition, subtraction, multiplication, and division become the foundation of higher order math skills like algebra and calculus, these fundamental macro design principles are the foundation to more advanced macro design principles. There is a whole world of design principles left to discover. Macro coupling, cohesion between macros, and macro design patterns are but a few. I encourage you to discover and explore these and other design principles. With a little time, patience, and good design principles, anyone can create a well written macro that is robust, flexible, maintainable, and is still clear and readable.

REFERENCES

Donald A. Norman , ‘The Design of EveryDay Things’, Basic Books 1998
R. V. Jones, ‘The Wizard War’, Coward, McCann & Geoghegan, Inc. 1978

Thanks to: Paul St. Louis his kind words, technical review and editing abilities.

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

Toby Dunn
550 Heimer Rd
San Antonio , Tx 78232
tobydunn@hotmail.com

Ian Whitlock
29 Lansdale Lane
Kenneth Square , Pa 19348
IanWhitlock@comcast.com