Paper 168-30

Macro Architecture in Pictures

Mark Tabladillo PhD, markTab Consulting, Atlanta, GA
Associate Faculty, University of Phoenix

ABSTRACT
The qualities which SAS macros share with object-oriented languages account for the power of macro programming. This picture guide will graphically model and introduce how SAS macros are best built, and use these graphics to illustrate how macros share some qualities of object-oriented programming (defined as encapsulation, polymorphism, and inheritance).

INTRODUCTION
Experienced SAS programmers use SAS Macro language all the time. Their combination of relative simplicity and power demonstrate the continued popularity of this language. The SAS Macro language is powerful in several partially overlapping ways:

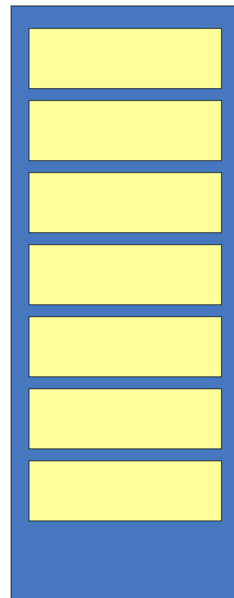
- Functionally – SAS has added specific macro functions
- Implementationally – the SAS System defines ways for macro variable and macros to resolve
- Structurally – the SAS Macro language has some characteristics of object-oriented languages

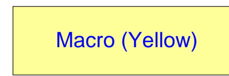
Focusing on structural power, this paper will describe SAS Macro Language features against a specific definition of object-oriented languages, defined as *encapsulation*, *polymorphism*, and *inheritance*. This paper will define each of these three terms, and then provide examples to support the conceptual reason why certain best practices in SAS macro language programming are justified.

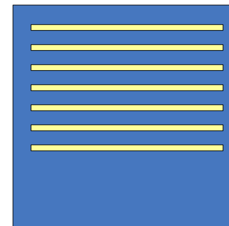
To be clear, this paper does NOT claim that the SAS Macro Language is inherently object-oriented, nor that the SAS Macro Language can or should be made to be completely object-oriented. Instead, the SAS macro language possesses many object-oriented language characteristics which can be optionally applied depending on how the code is written. This paper's premise is that just because the implementation of macro variables and macros has already been determined, the structural power has to be designed.

ENCAPSULATION
Encapsulation has been defined as "any kind of hiding, whether hiding variables or hiding methods, inside a class structure." (Tabladillo, 2003). The same paper describes a "class" as a "build-time repository for the declared attributes (variables) and the methods (functions or procedures)". Most closely, the class structure resembles a SAS Macro, which can indeed have its own local macro variables and methods. Also, a SAS Macro can be compiled (or built). The default location for this compiled SAS Macro is the catalog "work.sasmacr", and using the SAS explorer, you can see a graphical representation of compiled macros. Encapsulation describes the ability of the SAS macro to hide the definition of variables and methods inside a macro, and is (arguably) the strongest conceptual reason for using the SAS Macro Language. The following diagram pictures how encapsulation works. The diagram will be presented first, and then the explanation will show how to interpret the diagram.

BEFORE AFTER







Code (Blue)
With 7 One Line Macro
Calls (Yellow)

Code (Blue)
With 7 Blocks of
Repeating Code (Yellow)

Figure 1. Encapsulation

In Figure One, each column represents code, as if it were printed out on a long piece of paper. The start of the code is at the top, and it continues to the bottom. Yellow areas of the code represent sections which have been repeated. On the right, these repeated sections have been encapsulated into a macro, represented by the Macro (Yellow) box. The main code on the right (in blue) has seven one line macro calls, and therefore the overall code has been reduced in size.

The obvious advantage of encapsulation comes when code is repeated, and in this case it was code which was used seven times. Size reduction is also an advantage, but a further encapsulation advantage includes easier maintenance. In the left-hand example, something would have to be changed seven times. On the right, only one section of code would have to be changed. The following table presents code illustrating the above figure.

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td>

```
libname input 'c:/data/1972';
proc contents data=input.sales;
        run;
proc means data=input.sales;
        by year;
        var netSales grossSales margin;
        end;
proc freq data=input.sales;
        tables region*netSales/list;
        run;
libname input 'c:/data/1973';
proc contents data=input.sales;
        run;
proc means data=input.sales;
        by year;
        var netSales grossSales margin;
        end;
proc freq data=input.sales;
        tables region*netSales/list;
        run;
libname input 'c:/data/1979';
proc contents data=input.sales;
        run;
proc means data=input.sales;
        by year;
        var netSales grossSales margin;
        end;
proc freq data=input.sales;
        tables region*netSales/list;
        run;
libname input 'c:/data/1980';
proc contents data=input.sales;
        run;
proc means data=input.sales;
        by year;
        var netSales grossSales margin;
        end;
proc freq data=input.sales;
        tables region*netSales/list;
        run;
libname input 'c:/data/1999';
proc contents data=input.sales;
        run;
proc means data=input.sales;
        by year;
        var netSales grossSales margin;
        end;
proc freq data=input.sales;
        tables region*netSales/list;
        run;
libname input 'c:/data/2000';
proc contents data=input.sales;
        run;
proc means data=input.sales;
        by year;
        var netSales grossSales margin;
        end;
proc freq data=input.sales;
        tables region*netSales/list;
        run;
libname input 'c:/data/2001';
proc contents data=input.sales;
        run;
proc means data=input.sales;
        by year;
        var netSales grossSales margin;
        end;
proc freq data=input.sales;
        tables region*netSales/list;
        run;
libname input;
```

</td><td>

```
%macro salesAnalysis(year);
libname input "c:/data/&year.";
proc contents data=input.sales;
        run;
proc means data=input.sales;
        by year;
        var netSales grossSales margin;
        end;
proc freq data=input.sales;
        tables region*netSales/list;
        run;
libname input;
%mend salesAnalysis;

%salesAnalysis(1972);
%salesAnalysis(1973);
%salesAnalysis(1979);
%salesAnalysis(1980);
%salesAnalysis(1999);
%salesAnalysis(2000);
%salesAnalysis(2001);
```

</td></tr>
</table>

Table 1. Encapsulation Code

ENCAPSULATION ADVICE: MAKE SAS MACRO VARIABLES AS LOCAL AS POSSIBLE

Macros also have the ability to encapsulate variables, called *macro variables*. Encapsulated (or "local") macro variables are callable and changeable within the macro but not outside the macro. A common example includes a looping variable which would produce code based on the loop value. Macro variables can also be created outside of macros, and are referred to as global macro variables.

Some have offered the advice to never use global SAS Macro variables. However, technically, all global SAS Macro variables are encapsulated inside the SAS session, and therefore because they are encapsulated, there is a conceptual reason for using global variables. In addition to any user-defined global SAS macro variables, the SAS session provides many global macro variables (some are read-only) which provide dynamic information on the session. Conceptually, encapsulation pushes toward nesting macro variables as low as they can go. In other words, if a certain variable is only needed inside a specific macro, then it is a better programming practice to put that variable inside a macro as a local variable instead of a global variable. The following example contrasts a global variable definition with a local variable definition.

<table>
<tr><th>Good</th><th>Better</th></tr>
<tr><td>

```
%let yearStart = 1990;
%let yearEnd = 2004;

%macro tip1;
%do counter = &yearStart. %to &yearEnd.;
       Sales&counter.
%end;
%mend tip1;

proc print data=work.mark;
       var %tip1;
       run;
```

</td><td>

```
%macro tip1;
%local yearStart yearEnd;
%let yearStart = 1990;
%let yearEnd = 2004;
%do counter = &yearStart. %to &yearEnd.;
       Sales&counter.
%end;
%mend tip1;

proc print data=work.mark;
       var %tip1;
       run;
```

</td></tr>
</table>

Table 2. Global versus Local Macro Variables

In the better example, the macro variables *yearStart* and *yearEnd* are both a local and global variables, perhaps making them callable in other program areas. Making variables local prevents potential collisions, and makes maintenance easier in programs with many variables.

ENCAPSULATION ADVICE: NEST SAS MACROS WHEN LOCAL BINDING MAKES SENSE

Macros also have the ability to encapsulate macros, which would be called *nested macros* because they are contained within a defined macro. Nested macros are callable within the macro where they are defined but not outside the macro. Nesting macros makes sense when a macro will be called inside a specific macro, and will be called multiple times.

In object-oriented programming, using nested methods is an important way to leverage the power and efficiencies of encapsulation, namely by specifically hiding the definition of methods from the global SAS session. However, the SAS documentation from 1997 (SAS Institute, 1997) and the online documentation for version 9 (SAS Institute, 2002) reiterate the same wise message: "Avoid Nested Macro Definitions", because they are "usually unnecessary and inefficient". How should this efficiency discrepancy be resolved?

The rationale behind the documented advice has to do with how nested macros are processed. Nested macros are not compiled, but rather saved as text inside the macro definition, and then compiled (real-time) when the larger macro is invoked. In the object-oriented world, the difference can be described as the similar to the difference between "early binding" and "late binding". Because the SAS session does not know what parameters may be passed to the nested macro, it is therefore required (in general, and therefore applied to every case) to compile the macro during run-time instead of during build-time. Because nested macro compilation happens during run-time, the more "efficient" way to program may be (therefore) to only rely on build-time macros (that is, non-nested macros) which are entirely compiled no matter how many times they are called. The obvious efficiency therefore is processing time.

Still, there are times in object-oriented development when "late binding" (or run-time binding) is an advantage. The following table summarizes the advantages of early versus late binding.

Early Binding	Late Binding
◆ Allows for checking during compilation ◆ Typically faster	◆ Graceful adaptation to changes ◆ Flexible modification of existing code

Table 3. Early versus Late Binding

The following example illustrates what a nested macro looks like. Whether nested macros should be used is not typically apparent for any code example, but must be judged by the factors in the late versus early binding table, such as considering how the code fits into the entire development process (such as whether specific code is expected to be expanded or reused at some later date).

Nested Macro Example

```
%macro stats1(product,year);
   %macro title;
      title "Statistics for &product in &year";
      %if &year>1929 and &year<1935 %then %do;
         title2 "Some Data Might Be Missing";
      %end;
   %mend title;
   proc means data=products;
      where product="&product" and year=&year;
      %title
   run;
%mend stats1;

%stats1(steel,2002)
%stats1(beef,2000)
%stats1(fiberglass,2001)
```

Table 4. Nested Macro Example

An important caution: if you are going to code a nested macro, then it is recommended that you put the name of the appropriate macro on the %mend statements. In the above example, there are two illustration of named %mend statements (%mend title and %mend stats1). Many programmers never put the name on the %mend statement, and though code may work, it will be much easier to debug if you explicitly declare where a macro ends.

POLYMORPHISM

Polymorphism has been defined as “the ability to substitute objects of matching interface for one another at run-time.” (Tabladillo, 2003). The same paper describes an “object” as “a run-time entity which packages both a class structure and a specific state.” The word “polymorphism” has two important roots, that of “poly” meaning multiple, and “morph” meaning change. The term altogether means the ability to use the same macro for multiple uses. The following diagram pictures how encapsulation works. The diagram will be presented first, and then the explanation will show how to interpret the diagram.

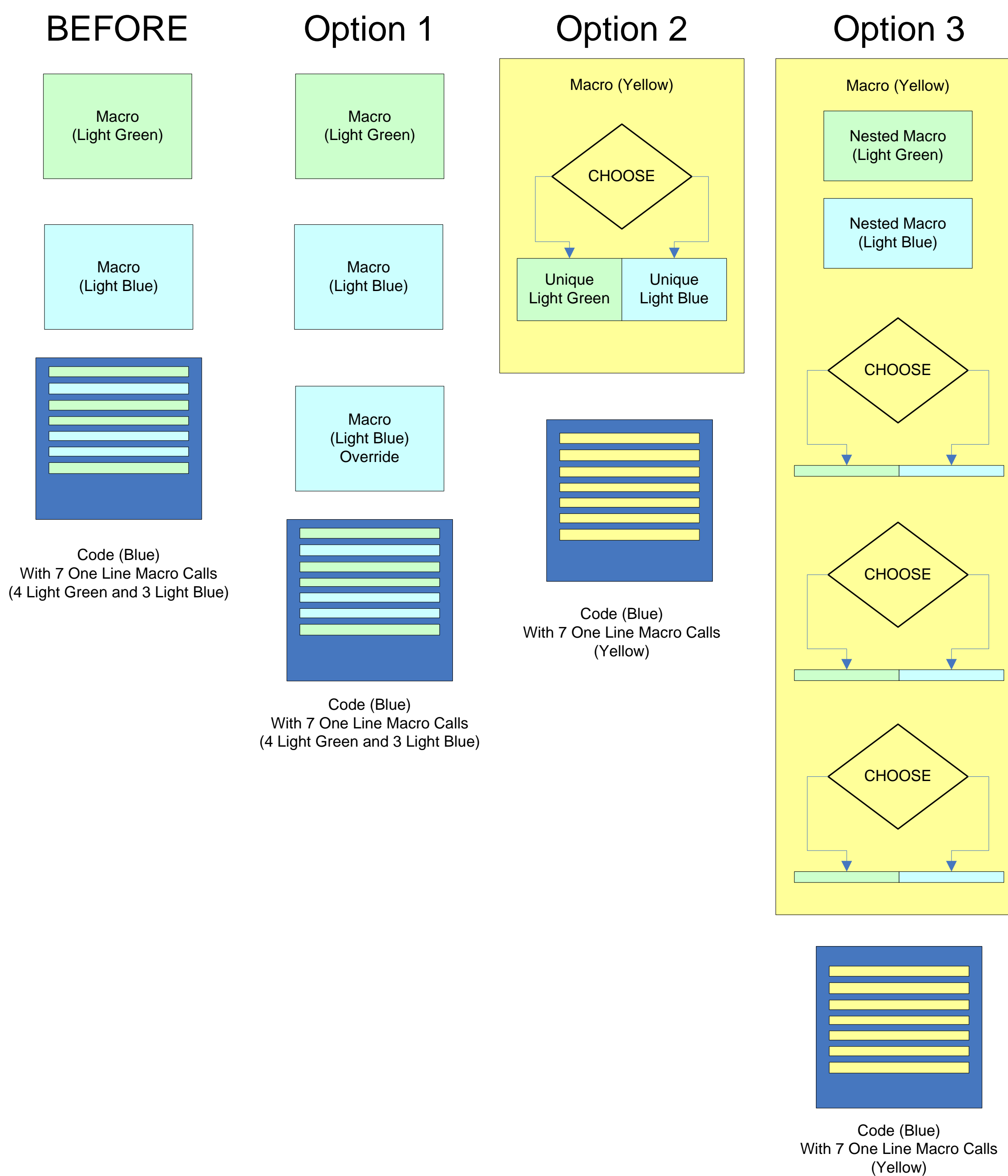


Figure 2. Polymorphism

In Figure Two, each column represents code as if it were printed out on a long piece of paper. The start of the code is at the top, and it continues to the bottom. The "Before" code on the left has two macros which are each called a number of times in the main code. The point of polymorphism is to have options in morphing to another form. Though there are many ways to morph code, only three types are illustrated in the picture. Option one represents overriding a macro at run time by redefining it again. The latest definition will be the only one which counts. Option two represents combining two macros into a single macro and using local macro variables to build that new macro. In this second option, the final macro could be defined by the parameters passed to it. Option three represents conditional nested macros, which could be determined by passing parameters to the macro.

The polymorphism need comes when specific projects may justify changing from one form to the other. If the polymorphism property were not true, there would only be one form or way to express code. However, the macro language enables programs to take on any number of forms. Experience would then prove when such a change is necessary. The rationale behind *refactoring* (changing forms of code expression) are beyond the scope of this

presentation. The next two sections will illustrate option one and option three (option two is similar to option three).

OPTION ONE: OVERRIDDING MACROS AT RUN TIME

There is an order for calling SAS macros (most local to most global). However, it is possible to redo definitions at run time. Overriding in object-oriented context typically means defining a method differently in another inherited class structure (at build time), though it could also mean a run time override. Without explaining what that statement means, there is a similar effect in the SAS macro language at run time when one macro is defined and then overridden (at the same level) by a later statement.

Overriding Macro at Run Time Example

```
%macro mark1(var=);
data work.&var.;
        length year 8;
        year=2003;
        run;
%mend mark1;

proc catalog cat=work.sasmacr;
        delete mark1.macro;
        run;

%macro mark1(var=,year=);
data work.&var.;
        length year&year. 8;
        year&year=&year.;
        run;
%mend mark1;
```

Table 5. Overriding Macro Example

In the above example, the macro named "mark1" is defined, then deleted using a proc catalog statement, and then redefined. This technique is used in a SAS/AF application where the design is to use a limited set of macro names throughout the application (to prevent collisions) and then delete the macros after they are used. Rerunning the program creates the macros again, and the new definition may or may not (usually not, in this case) be the same as before. The proc catalog statement is not absolutely required because a new macro will override the old one without the need to delete the macro. However, it is good to delete macros once it is determined that they are no longer needed.

OPTION THREE: CONDITIONALLY DEFINE NESTED SAS MACROS

Nested SAS Macros are defined at run time, and were discussed earlier in this paper. Allowing these nested macros to be conditionally defined is a way to leverage polymorphism. It's important to state that if unconditional nested macros do not make sense, then conditional nested macros will not make sense either. Therefore, it's important to look over the earlier discussion on nested SAS macros first.

Conditional Nested SAS Macro Example

```
%macro stats1(product,year);
title "Statistics for &product in &year";
%if &year>1929 and &year<1935 %then %do;
   title2 "Some Data Might Be Missing";
%end;

%if &year >= 2001 %then %do;
%macro analysis(type=);
   proc means data=work.products;
      where product="&product" and year=&year and type=&type;
   run;
   proc freq data=work.products;
      tables year*type*product/list;
         run;
%mend analysis;
%end;
%else %do;
%macro analysis(type=);
```

```
   proc means data=work.products;
      where product="&product" and year=&year and type=&type;
   run;
%mend analysis;
%end;

%do counter = 30 %to 50;
       %analysis(&counter);
%end;
title;
%mend stats1;

%stats1(steel,2002)
%stats1(beef,2000)
%stats1(fiberglass,2001)
```

Table 6. Conditional Nested Macro

Note in the above example that the nested macro named "analysis" is conditionally defined based on the year. As in the earlier nested macro example, whether or not this macro structure makes sense involves a larger discussion of where this code fits into the big picture, and specifically what the developer would expect to happen over the long term. These factors are the most complex and involved of the polymorphism-related examples. The option two example would be similar except that there would be no %macro or %mend statements inside the conditional logic.

INHERITANCE

Inheritance has been defined as "the process of acquiring the characteristics of another piece of software code." (Tabladillo, 2003). More specifically, an object-oriented language will allow one piece of code to inherit specific attributes (variables) and methods (functions) from the designated parent(s). Calling a macro does not allow access to the internals, the local variables and perhaps the nested macro definitions. There is a way to share the guts of a macro in other macros. The diagram will be presented first, and then the explanation will show how to interpret the diagram.

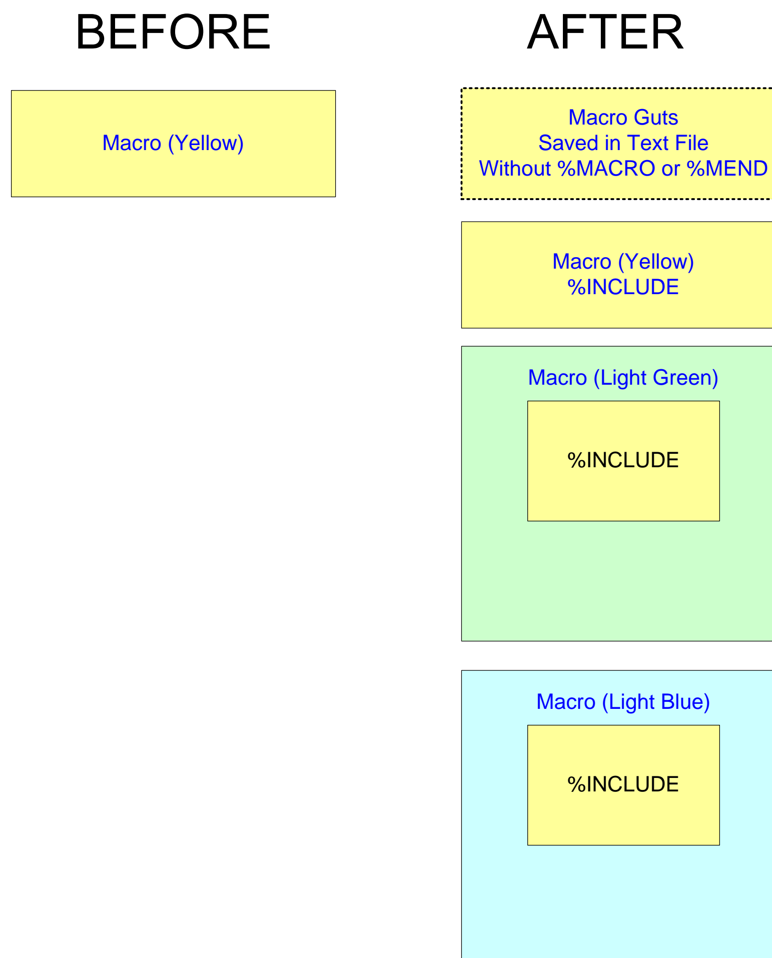


Figure 3. Inheritance

In Figure Three, each column represents code, as if it were printed out on a long piece of paper. The start of the code is at the top, and it continues to the bottom. On the left, we have a macro (not called) which is colored yellow. On the right, we have removed the yellow macro code by stripping off the %macro and %mend lines, and put the guts of this macro into a text file. These guts are then inserted into a new yellow macro using the %INCLUDE command. Because the %macro and %mend statements are absent, the encapsulation barrier has been removed, and the guts can be inserted, using %INCLUDE, into other macros. In Figure Three, we see these guts inserted into a Light Green Macro and a Light Blue Macro.

The advantage of inheritance includes being able to change the inherited parent once, while all derivative children will successively take on the new parent's characteristics. Using this technique would only be powerful if the block of code were reused iteratively in later sections of code, and the more the code is replicated through %INCLUDE, the more powerful this technique would be.

FAKING INHERITANCE: USE %INCLUDE TO REUSE MACRO CODE

The SAS macro language has no direct means to inherit variables and methods. In other words, there is no %INHERIT which would automatically strip off the encapsulation barrier (the %macro and %mend lines) and leave behind only the guts. However, using %INCLUDE, a base SAS statement, allows inheritance characteristics to be faked.

The %INCLUDE command can be used in the following contexts:

- Lines can be included from an external named text file
- Files can be included from a SAS Catalog
- Previously submitted lines can be included by number (feeling lucky?)
- Lines can be entered interactively (terminated with a %RUN command)

You can find examples and instruction for %INCLUDE in the online documentation under the list of base SAS statements (don't look in the Macro reference). The %INCLUDE would insert all the "guts" of the macro, everything between the %MACRO and %MEND statements. In making new macros, the %INCLUDE statement would need to be the first statement in the inheriting macro, even before local macro variable definitions. Putting the %INCLUDE first allows the option of overriding local macro variables or nested macro definitions. Since we don't have a %INHERIT, the following sample code shows how to reuse macro guts from a text file.

```
filename yellow "c:\files\yellowMacro.txt";

%macro yellow();
%inc yellow;
%mend yellow;

%macro lightGreen(year);
%inc yellow;
%let happy = Living Another &Year.;
%mend lightGreen;

%macro lightBlue(sugi);
%inc yellow;
%let happy = I Love &sugi.;
%mend lightBlue;
```

Table 7. Inheritance Example

Looking at the above code, it would seem that there is no need for a %INHERIT command. However, this technique is extremely limited because whatever is inside the included file is resolved in open code. Thus, you could define macro variables in the inserted yellow macro, but they would become global macro variables, and most all of the helpful functions and features specific to macros would be unresolvable. This including technique would only be of regular use for code not requiring macro-specific functionality. Therefore, the only current way to achieve full inheritance is to manually cut and paste text into your new macro.

CONCLUSION

This presentation demonstrated a conceptual framework to analyze the structural power of SAS Macros. SAS Macros are robust in encapsulation, most varied in polymorphism, and weakest in inheritance (since there is no direct inheritance feature). The specific examples show how to leverage qualities of object-oriented languages. Using these tips can help coders create more structurally powerful SAS macros.

REFERENCES

Fowler, Martin (1999), *Refactoring: Improving the Design of Existing Code*, Reading, MA: Addison Wesley Longman, Inc.

SAS Institute Inc. (1997), *SAS Macro Language: Reference, First Edition*, Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2002), *SAS OnlineDoc® 9*, Cary, NC: SAS Institute, Inc.

Shalloway, A., and Trott, J. (2002), *Design Patterns Explained: a New Perspective on Object-Oriented Design*, Boston, MA: Addison-Wesley, Inc.

Tabladillo, M. (2003), "Application Refactoring with Design Patterns", *SUGI Proceedings*, 2003.

CONTACT INFORMATION

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

Mark Tabladillo
markTab Consulting
Web: http://www.marktab.com/

TRADEMARK CITATION

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.