Paper TU-11

Toward Object-Oriented Macros in SAS®

Mark Tabladillo PhD, markTab Consulting, Atlanta, GA

ABSTRACT

The qualities which SAS macros share with object-oriented languages account for the power of macro programming. This presentation will attempt to describe the extent to which SAS macros are object-oriented (defined as encapsulation, polymorphism, and inheritance) with the goal of extracting best practices for SAS macro programming.

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 evaluate the SAS Macro Language 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. Additionally, by making a connection between the SAS Macro language and object-orientation, then best practices in refactoring can be applied.

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.

The goal of this paper is to design structurally powerful SAS macros. The assumption is that SAS Component Language (SCL) classes or objects will NOT be used. The deliverable is seven specific tips on SAS macro structure.

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.

TIP ONE: MAKE SAS MACRO VARIABLES AS LOCAL AS POSSIBLE

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.

Good	Better
%let number=30; **%macro *tip1***; %do counter = **1** %to &number.; value2004&counter.1 %end; **%mend** tip1; %let number=50 Person Population; Title1 "&number."; **proc print** data=work.mark; var %***tip1***; **run**;	%let number=50 Person Population; **%macro *tip1***; **%local number;** %let number=30; %do counter = **1** %to &number.; value2004&counter.1 %end; **%mend** tip1; Title1 "&number."; **proc print** data=work.mark; var %***tip1***; **run**;

In the better example, the macro variable number is both a local and a global variable, perhaps making it available (by name) for use in other parts of the program. Making variables local also prevents potential collisions with common macro variable names with other portions of the program, or even global variables previously used during the SAS session.

TIP TWO: NEST SAS MACROS WHEN LOCAL BINDING MAKES SENSE

In object-oriented programming, using nested definitions for methods is the generally-accepted way to leverage the power and efficiencies of encapsulation, namely by specifically hiding the definition of methods from the global SAS session. The SAS documentation from 1997 (SAS Institute, 1997) and the online documentation for version 9 (SAS Institute, 2002) reiterate the same 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, namely when the programmer desires to hide a method's implementation inside a macro. The SAS documentation does not provide an explanation of when to not avoid nested macro definitions, but this paper provides a specific guideline: use nested macro definitions in cases when late-binding would make sense.

The following table summarizes the advantages of early versus late binding, and these guidelines provide the full scope of wisdom of why a programming technique which technically takes longer to accomplish may actually be overall judged more efficient.

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

As a final point, even the SAS documentation admits that the processing difference may be minimal:

> With only a few statements, this [nested macro code] takes only micro-seconds; but in large macros with hundreds of statements, the wasted time could be significant. (SAS Institute, 1997, 2002)

The advice from SAS Institute remains appropriately the same from 1997 to 2002, even though a “micro-second” has been substantially reduced given to cheaper access to faster and multiple processing and hardware systems. The benefits of speed efficiency today diminish as hardware improves tomorrow. It could be argued that object-oriented software development is either justified by, or perhaps justifies, more efficient hardware.

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)
```

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.”

TIP THREE: FEEL FREE TO PASS BLANK PARAMETERS

Object-oriented languages can distinguish between types of variables passed as parameters. A SAS macro can only pass character variables, even though SAS Component Language can pass the following types: character, numeric, lists, and objects. An object-oriented language can distinguish the difference in multiply-defined methods (or objects) by looking at the order and types of parameters, which define a specific *signature* for that method or object. SAS macros are simpler, using only character types all the time.

Even though a macro may have n parameters defined, it is not required to pass a value for each parameter in the definition. By using different parameters, it is possible to have the same named macro perform two very different types of operations.

Being an engineer, I have a personal theory that technical people generally attempt to fill in each and every blank when they fill out a form. This tip frees those types of people from their natural instinct.

Blank Parameter Example

```
%global age;
%let income=income;
%let yrs_educ=yrs_educ;
data work.mark;
        income = 75000;
        yrs_educ = 20;
        age = 40;
        run;

%macro plot(yvar= ,xvar= );
%if %length(&xvar) and %length(&yvar) %then %do;
  proc plot data=work.mark;
    plot &yvar*&xvar;
    run;
%end;
%else
%if %length(&yvar) %then %do;
  proc means data=work.mark;
    var &yvar;
    run;
%end;
%mend plot;

%plot(yvar=&income,xvar=&age)
%plot(yvar=&income,xvar=&yrs_educ)
```

In the above example, the macro variable “age” is blank. Note that the macro named “plot” has conditional code (highlighted) to execute different processing depending on the “xvar” macro variable. In this case, because “age” is blank, the macro will, in one case, execute a proc means, and then execute a proc plot.

Passing blank parameters is not the same as having two independent macros, callable depending on their signature, but the blank parameter technique leverages some of the polymorphic advantages.

TIP FOUR: OVERRIDE SAS MACROS AT BUILD TIME

Earlier, this presentation claimed that the SAS macro was similar to a class structure. More specifically, a SAS macro cannot be a class because it is not possible to multiply instances of the macro into specifically defined objects. The class would be the conceptual model, and the object would be the named real model. A SAS macro more closely resembles an object because it has a specific single name which can be called.

What happens if there are macros with the same name? The SAS session recognizes only one name of a specific SAS macro, and from the documentation, SAS will resolve named macros in the following order (SAS Institute, 2002, see the subsection titled “Saving Macros in an Autocall Library” under the section “Storing and Reusing Macros”):

1. The most recently defined session compiled macro definition
2. The permanently stored compiled macro
    a. Compiled stored macros must be enabled with the MSTORED option
    b. The macro processor opens the single macro catalog in the library specified in the SASMSTORE option
    c. Macro names are unique in the single compiled macro library
3. A macro saved in a library
    a. Libraries must be specified by the SASAUTOS option
    b. The macro processor searches for libraries in the order in which they are specified in the SASAUTOS option
    c. Each library is searched for a member with the specified name, since different libraries may have the same named macro

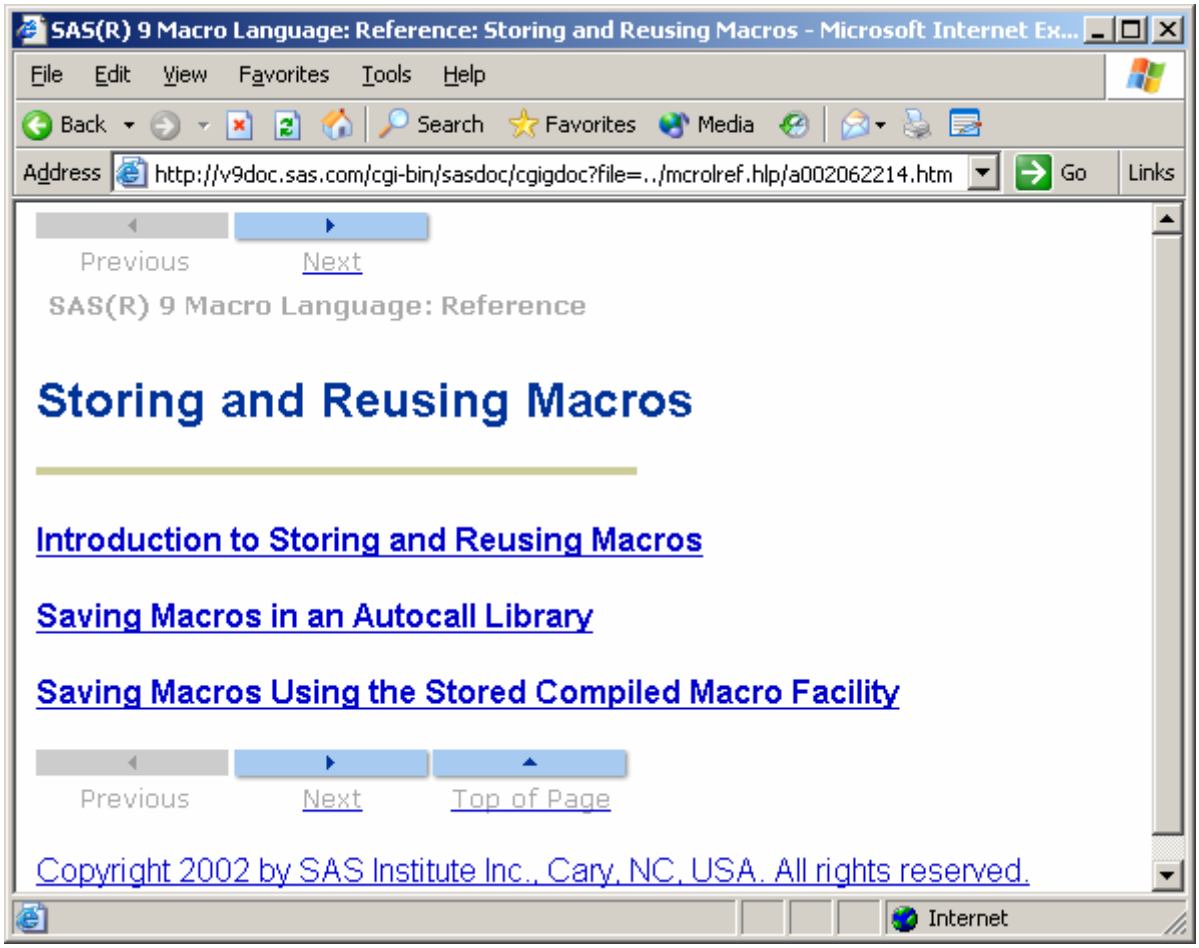
The following table summarizes the three places to save macros:

Current Session Complied Macros	Most Local
Stored Compiled macros (one user)	Still Local
Autocall Library (shared)	Most Global

If the macro is found, it is then compiled, then any nested macros are compiled, and finally the macro is run. Because the macro processor will use one and only one definition of a macro, the use of macros resembles properties, restrictions, and advantages of the Singleton design pattern (Shalloway and Trott, 2002). All this discussion is directly pertinent to polymorphism, because in pure object-oriented languages, there is the ability to have multiple classes or objects with the same name, even though they are defined differently. In the above diagram, there are three specific places to put callable macros of the same name.

The tip for this step is to use this structure to the developer's advantage, and know that the order of execution allows macros with the same name to be simultaneously available in different places.

Examples are online (as of this writing), at the URL listed in the picture below – in general, you can find examples and instruction in the online documentation under the SAS Macro Language Reference document.



TIP FIVE: OVERRIDE SAS 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.

Note that 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;
```

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.

TIP SIX: 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)
```

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 tips, and therefore has been presented last.

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).

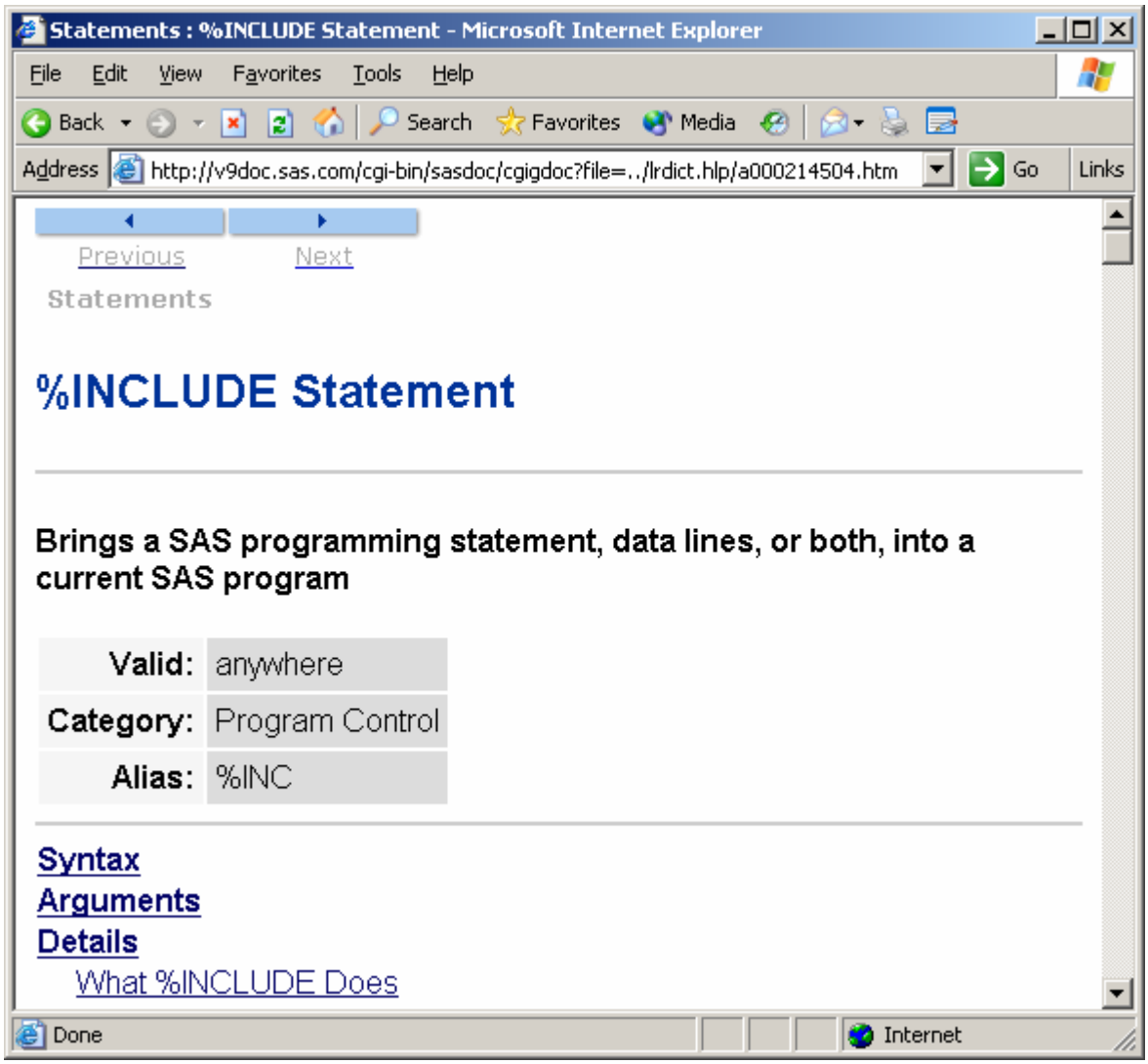
TIP SEVEN: USE %INCLUDE TO REUSE MACRO CODE

The SAS macro language has no direct means to inherit variables and methods. In other words, there is no %INHERIT. However, there is a %INCLUDE which is a Base SAS statement, and not inherently a SAS macro feature, making it an ideal tool for code reuse.

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)

Examples are online (as of this writing), at the URL listed in the picture below – in general, you can find examples and instruction in the online documentation under the list of SAS statements (don't look in the Macro reference).



CONCLUSION

This presentation demonstrated a conceptual framework to analyze the structural power of SAS Macros. SAS Macros are strongest in encapsulation, and weakest in inheritance. The seven tips provided specific examples of 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