

SAS® Macro Design Patterns

Mark Tabladillo Ph.D., MarkTab Consulting, Atlanta, GA
Associate Faculty, University of Phoenix

ABSTRACT

The qualities which SAS® macros share with object-oriented languages account for the structural power of macro programming. This paper illustrates some examples of specific design patterns which can be partially or fully implemented with the SAS macro language. The material is intermediate to advanced, and assumes knowledge of macros and macro variables. The goal is to systematically illustrate best practices for SAS macro programming.

INTRODUCTION

Experienced SAS programmers use the SAS macro facility all the time. The macro facility is composed of two components, the macro processor (the portion of SAS that does the work) and the macro language (the syntax used to communicate with the macro processor) (SAS Institute, 2005). The SAS macro language is powerful in several partially overlapping ways, as illustrated in the following table.

Power Source	Definition	Examples
Function	The SAS System has added specific macro functions.	• %LENGTH • %EVAL • %SYSFUNC See SAS Institute (2005)
Implementation	The SAS System defines specific ways that macro variables and macros resolve.	• Global versus local macro variables • %DO inside macros See SAS Institute (2005)
Structure	The SAS Macro language has some characteristics of object-oriented languages.	To some extent: • Encapsulation • Polymorphism • Inheritance See Tabladillo (2005)

Table 1. Power sources of the SAS macro facility

Focusing on structural power, this paper will extend concepts from an earlier paper (Tabladillo, 2005) and provide specific examples of how macros can be used to implement some aspects of object-oriented *design patterns*, defined as a structural solution in context for solving a coding problem.

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 SAS macro language is a character-based scripting language which can take advantage of the base SAS System as well as many additionally licensed products which can be submitted from the SAS Windowing Environment. The base SAS System itself, a combination of functions, statements, and procedures, already simplifies many tasks operations on variables and datasets. These simplifications, especially seen in the SAS procedures, already contain many object-oriented features because base SAS is written in an object-oriented language. Simply using the SAS Windowing Environment allows programmers to leverage the power of object-oriented languages without knowing any object-oriented concepts (such as encapsulation, polymorphism and inheritance). The SAS macro language allows a programmer to additionally customize blocks of code.

The SAS macro language continues to be popular because with each new version, base SAS and optional additional SAS modules continue to grow more powerful, resulting in a reduced need to code complex applications or functions. Object-oriented languages excel in being able to create and readily maintain complex applications, and SAS Institute continues to listen to the community by continually adding many functions into the base and additionally licensed modules, improving the value delivered to SAS users. Since many of the common functions (such as INDEX) have

already been coded, the role of the SAS macro language is relegated to controlling how, which, and when these base and licensed functions are called.

The goal of this paper is to provide specific design ideas for structuring macros in the most powerful way possible. The design pattern applications are based on definitions provided by the experienced object-oriented development community, but the concepts can apply to some extent to the SAS macro language.

This paper will proceed through four sections:

- First, the paper will illustrate three specific ways to trick the SAS macro language to emulate some properties of object-oriented languages.
- Second, the paper will answer what a *design pattern* is and why design patterns are useful.
- Third, the paper will list seven specific design patterns and provide SAS macro language examples.
- Finally, the paper will prove some general application advice.

HOW TO TRICK THE SAS MACRO LANGUAGE

Tabladillo (2005) presents the limitations of the SAS macro language, which is a scripting language and not a full blown object-oriented language. However, scripting languages such as the SAS macro language continue to have popularity because of their relative simplicity and low memory and resource overhead compared with many object-oriented counterparts. Some scripting languages like PHP are free, and because the SAS macro language comes with the base SAS System, it has the advantage of being essentially free to anyone who licenses that base product.

This section will provide a list of three specific tricks which allow the SAS macro language emulate some features of object-oriented languages.

TRICK ONE: EMULATE INHERITANCE BY CUTTING AND PASTING TEXT

The SAS macro language, unfortunately, does not have direct inheritance. It would not violate the SAS macro facility to add a %INHERIT command which would allow inheritance, and we may see such a feature in future SAS versions. However, Tabladillo (2005) provided a way to simulate inheritance using cut-and-paste.

This trick involves using code from a SAS file, or stored in a catalog, and simply cutting and pasting the text to a new location. The trick does not provide the same features as an object-oriented language (such as the optionally licensed SAS/AF® software), which would dynamically reflect changes in the parent code everywhere the inherited code was referenced. Nevertheless, if there are only a few places where inheritance is needed, the cut-and-paste solution can provide a way to emulate inheritance.

TRICK TWO: EMULATE ALL TYPES OF VARIABLES THROUGH SAS MACRO VARIABLES

The SAS macro language only accepts characters in its interface, defined as the collection of elements passed to a SAS macro. An object-oriented language will often accept characters, numbers, arrays, and objects as specific elements, and can possibly test whether or not the information passed matches the type of variable expected. The advantage of object-oriented languages is being able to test for potential errors during build time (when the code is being written) instead of discovering them at run time (during code submission or execution).

Still, the SAS macro language can be tricked to simulate passing other than just character variables. Many programmers know how to pass a number, but the following code provides some ideas on how to pass an array or an object (in this case, implemented as a nested macro):

```

%macro interface(char, num, array, macro);
%local major;
%macro breakarray(m, n, p, q);
data _null_;
    call symput('major',&m);
run;
%put &m &n &p &q;
%mend breakarray;
%macro submacro(a, b);
%put &a &b;
%mend submacro;
%put &char &num &array &macro;
* Break up the array;
%breakarray&array.;
%put &major;
* Call the named macro;
%&macro.;
%mend interface;
%interface (char=character, num=2005, array=(1,2,3,4), macro=submacro(a=5,b=c));

```

Figure 1. Passing different data types through a macro

This example can be submitted in a base SAS session. In the above code, the CHAR macro variable contains character information, the NUM macro variable includes a number, the ARRAY macro variable contains a list or array of numbers and/or characters, and the MACRO macro variable references a call to a specific nested macro.

The BREAKARRAY nested macro uses a CALL SYMPUT to populate the local MAJOR macro variable with the value. Note that the data step resolves inside the INTERFACE macro, and therefore populates the local MAJOR macro variable. Also, the data step needs to be inside the nested BREAKARRAY macro because it requires knowledge of the local M macro variable. The nested SUBMACRO is called within the INTERFACE macro, and the technique illustrated allows for named parameters to be used in this nested call.

This code merely provides illustrations of ways to pass more than just characters, and depending on the application, you could customize your own code to do more sophisticated things which object-oriented languages automatically do such as determine the number of elements in the array, or determine whether the elements of the array are numeric or character.

TRICK THREE: EMULATE PUBLIC METHODS (OR FUNCTIONS) BY APPLYING NESTED MACROS

The SAS macro language does not have *public* (externally callable) methods (or functions), as would be true in an object-oriented language. However, the SAS macro language does allow for nested macros, which provides a way to potentially simulate method calls.

```

%macro functions(function, array);
%macro subtract(p1,p2,p3);
data _null_;
    result = &p1 - &p2 - &p3;
    put result=;
run;
%mend subtract;
%macro add(p1,p2,p3);
data _null_;
    result = &p1 + &p2 + &p3;
    put result=;
run;
%mend add;
%if &function. = subtract or &function. = add %then %&function.&array.;
%mend functions;
%functions (function=add, array=(p1=45,p2=34,p3=34));

```

Figure 2. Emulating a public method (or function) with a nested macro

This example can be submitted in a base SAS session, and could be considered an application of trick two. Macros can be explicitly tricked to allow their internal nested macros to be called from the outside as a function would be called. The call itself is NOT an application of %IF and %THEN (which could internally decide what to call) but instead is a direct reference to a known nested macro. Object-oriented languages often allow objects to declare their functions as *public* (callable from the outside) or not, increasing the level of security and preventing potentially unwanted results. Having covered these three tricks, this paper will now focus on the what and why of design

patterns.

DESIGN PATTERNS – WHAT AND WHY?

Shalloway and Trott (2002) cite architect Christopher Alexander's definition of a *pattern* as “a solution to a problem in context.” (Shalloway and Trott, 2002, p. 74) Moreover, they propose that patterns provide a way to solve commonly recurring software design problems, and they provide two compelling reasons to use design patterns at all (Shalloway and Trott, p. 80):

- **Reuse solutions** – programmers learn efficiencies through code reuse, and *design patterns* provide a way to copy coding frameworks from one project to the next. Additionally, most coding problems follow the design patterns described in the software development literature, and developers who know these patterns have a conceptual advantage in both recognizing and solving similar problems in the future.
- **Establish common terminology** – By attaching the same terms to the same types of problems, teams of developers can collaborate more efficiently. Additionally, there is an opportunity to learn and collaborate with the larger software development community, even people who program in different languages.

The reusing solutions rationale provides a way for software developers to systematically gain more experience in authoring complex systems. The common terminology rationale allows developers to continue to read books and articles on design patterns, and use that knowledge to build conceptual experience based on the experience of other developers. In this author's opinion, the terminology associated with object-oriented languages provides the main way that application developers can generalize their experience whether using the SAS macro language, SAS/AF® software, or any scripting or object-oriented language.

DESIGN PATTERNS USING THE SAS MACRO LANGUAGE

Anyone who has been already using the SAS macro language will probably discover that they have already been using some of these patterns. What follows is a list of specific patterns which can be fully or partially implemented with the SAS macro language.

FACADE: SIMPLIFY A COMPLEX SYSTEM

From Shalloway and Trott (2002):

	Facade Pattern Key Features
Intent	You want to simplify how to use an existing system. You need to define your own interface.
Problem	You need to use only a subset of a complex system. Or you need to interact with the system in a particular way.
Solution	The Facade presents a new interface for the client of the existing system to use.
Consequences	The Façade simplifies the use of the required subsystem. However, since the Façade is not complete, certain functionality may be unavailable to the client.

Table 2. Façade Pattern Key Features (Shalloway and Trott, 2002, p. 90)

The façade pattern is perhaps one of the most used ideas in the SAS macro language documentation. The idea is to put complex code inside a macro, and simply present a macro variable interface which only contains certain unique variables.

The following SAS macro language example shows how the Façade pattern might work. In this case, the variables passed to the macro represent the *interface*. We earlier saw a way to pass character, numeric, array, and macro call information through the macro interface. In this example, we are going to pass the name of a dataset through the macro, which will contain a relatively complex series of commands to process that dataset.

```
%macro facade(dataset);  
proc freq data=&dataset.;  
    tables year*sales/list missing;  
run;  
  
proc means data=&dataset.;  
    var annualSales totalEmployees sumTurnover;  
run;  
%mend facade;  
%facade(dataset=sales.ca200503);  
%facade(dataset=sales.ca200510);  
%facade(dataset=sales.ca200511);
```

Figure 3. Façade pattern example

Because the details of the operations are not presented through the interface, this example illustrates that the macro call asks only for the minimum information required to make a move, specifically, what dataset is involved, thus

satisfying the best practice number one of programming to as simple an interface as possible. The *façade* above is specifically only the dataset name, and the macro itself does the complex work of completing several procedures based on this simple one-variable interface.

STRATEGY: USE DIFFERENT BUSINESS RULES DEPENDING ON CONTEXT

From Shalloway and Trott (2002):

	Strategy Pattern Key Features
Intent	Allows you to use different business rules or algorithms depending on the context in which they occur.
Problem	The selection of an algorithm that needs to be applied depends upon the client making the request or the data being acted upon. If you simply have a rule in place that does not change, you do not need a Strategy pattern.
Solution	Separates the selection of the algorithm from the implementation of the algorithm. Allows for the selection to be made based upon context.
Consequences	• The Strategy pattern defines a family of algorithms. • Switches and/or conditionals can be eliminated. • You must invoke all algorithms in the same way (they must all have the same interface).

Table 3. Strategy Pattern Key Features (Shalloway and Trott, 2002, p. 238)

The strategy pattern allows algorithm variation to happen based on the parameters. In this pattern, the implementation of the algorithm is completely within one macro, and the selection of the implementation is passed to the macro through the macro parameters (which collectively form the *context*). The *family of algorithms* are implemented as a series of nested macros, and by using the macro parameters to choose the correct algorithm, there is no need for IF/THEN statements. The following example illustrates how to use the strategy pattern based on the example of choosing tax processing algorithms.

```
data work.taxExample;
    length grossAmount totalTax 8;
    grossAmount = 34000;
    run;
%macro calculateTaxes(dataset, county);
%macro taxStrategy(county);
%macro DeKalb;
    totalTax = grossAmount * 0.06;
%mend DeKalb;
%macro Fulton;
    totalTax = grossAmount * 0.075;
%mend Fulton;
%macro Gwinnett;
    totalTax = grossAmount * 0.06;
%mend Gwinnett;
%if &county. = DeKalb or &county. = Fulton or &county. = Gwinnett %then %&county.;
%mend taxStrategy;
data &dataset.;
    set &dataset.;
    %taxStrategy(&county.);
    run;
proc print data=&dataset.;
    run;
%mend calculateTaxes;
%calculateTaxes(dataset=work.taxExample, county=Fulton);
```

Figure 4. Strategy pattern example

The strategy pattern example can be submitted in base SAS. The family of algorithms are named *DeKalb*, *Fulton*, and *Gwinnett*, after the three metro Atlanta counties, and these three macros are chosen outside the *calculateTaxes* macro. Therefore, there are no IF/THEN statements involved in choosing the specific strategy.

In actual use, the nested tax calculation macros could be quite complex, and additionally, it is possible to introduce identical parameters to these simplified macros to make them more realistic to actual business problems. The idea is that nested macros can be strategically chosen from outside the containing macro.

SINGLETON: HAVE ONLY ONE OBJECT AVAILABLE

From Shalloway and Trott (2002):

	Decorator Pattern Key Features
Intent	You want to have only one of an object but there is no global object that controls the instantiation of the object.
Problem	Several different client objects need to refer to the same thing and you want to ensure that you do not have more than one of them.
Solution	Guarantees one instance.
Consequences	Clients need not concern themselves whether an instance of the singleton object exists.

Table 4. Singleton Pattern Key Features (Shalloway and Trott, 2002, p. 258)

By definition, the SAS macro language only allows for one macro object to exist. There cannot be multiple *instantiations* (uniquely named copies), because instantiation is not a SAS macro language feature. Instantiation implies *classes* (a single master source to copy and name), and there are no SAS macro classes. Perhaps a purist language analyst might view the macro as a class and the macro call to be a singleton instance (copy). However, this author prefers to consider the macro as a single object because the macro call does not allow for explicitly naming the new instantiation with any arbitrary unique name, but only allows the original name.

Putting these complex subtleties aside, the point is that the SAS macro language only allows for one instance of a macro, period. Therefore, you don't need to code anything additional because your SAS session counts as a the global object (where, for example, a global macro variable might exist) and that global SAS session will only allow for a single operative copy of your macro. Recalling the earlier strategy example with tax tables, a singleton macro inherently could be a good place to store tax calculation rules (say for a particular year, or a particular year and month combination) because there would be one and only one place where those rates could be set.

The base SAS session is a place for global macro variables to be defined, and it is a place for a single macro definition to reside too. However, since macros can be called from several locations, it is important to note what the SAS documentation says about the macro hierarchy, and specifically which single definition takes precedence:

When you call a macro, the macro processor searches for the macro name using this sequence:

1. the macros compiled during the current session
2. the stored compiled macros in the SASMACR catalog in the specified library (if options MSTORED and SASMSTORE= are in effect)
3. each autocall library specified in the SASAUTOS option (if options SASAUTOS= and MAUTOSOURCE are in effect).

(SAS Institute, 2005)

Finally, note that if you define a macro in a SAS session, then define it again, the SAS macro facility will consider the last submitted definition to be the current definition. Multiple macro submissions are a way to *override*, or change the definition, of a macro.

DECORATOR: ATTACH ADDITIONAL RESPONSIBILITIES DYNAMICALLY

From Shalloway and Trott (2002):

	Decorator Pattern Key Features
Intent	Attach additional responsibilities to an object dynamically.
Problem	The object that you want to use does the basic functions you require. However, you may need to add some additional functionality to the object occurring before or after the object's base functionality.
Solution	Allows for extending the functionality of an object without resorting to subclassing.
Consequences	Functionality that is to be added resides in small objects. The advantage is the ability to dynamically add this function before or after the core functionality.

Table 5. Decorator Pattern Key Features (Shalloway and Trott, 2002, p. 251)

The decorator pattern can only be fully implemented using a full object-oriented language. Though inheritance can be simulated with the SAS macro language, there is no class structure, only objects (the macros). However, the following example demonstrates a partially implemented decorator pattern which provides some of the main features.

```

%macro messyMacro(num,char);
* Messy Macro;
%put MessyMacro &num. &char.;
%mend messyMacro;

%macro decoratorBefore(parameters);
%put Before;
%messyMacro&parameters.;
%mend decoratorBefore;

%macro decoratorBeforeAndAfter(parameters);
%put Before;
%messyMacro&parameters.;
%put After;
%mend decoratorBeforeAndAfter;

%macro decoratorAfter(parameters);
%messyMacro&parameters.;
%put After;
%mend decoratorAfter;

%decoratorBeforeAndAfter(parameters=(num=1,char=SAS));

```

Figure 5. Decorator pattern example

The decorator pattern example can be submitted in base SAS. The *messyMacro* has only a few lines, but represents what could be a very complex macro. Perhaps this complex macro represents a macro stored in a shared SAS catalog, and has been used successfully for years, and is currently being used by many people.

The decorator pattern provides a way to add code either before or after this *messyMacro*. Three variations are provided, though in the general decorator pattern, there could be any number of macros which had code *before*, and any number of the other two variations (named *before and after*, and *after*). Again, this implementation is not the full decorator pattern since true object-oriented languages can have uniquely named multiple objects, while the SAS macro language only allows for one object (or macro) to be available.

TEMPLATE METHOD: APPLY DISCIPLINE TO THE FUNCTIONS AVAILABLE INSIDE MACROS

From Shalloway and Trott (2002):

	Template Method Pattern Key Features
Intent	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Redefine the steps in an algorithm without changing the algorithm's structure
Problem	There is a procedure or set of steps to follow that is consistent at one level of detail, but individual steps may have different implementations at a lower level of detail.
Solution	Allows for definition of substeps that vary while maintaining a consistent basic process.
Consequences	Templates provide a good platform for code reuse. They are also helpful in ensuring the required steps are implemented.

Table 6. Template Method Pattern Key Features (Shalloway and Trott, 2002, p. 283)

The template pattern can only be fully implemented using a full object-oriented language. Though inheritance can be simulated with the SAS macro language, there is no class structure, only objects (the macros). However, the following example demonstrates a partially implemented template pattern which provides some of the main features.

```

%macro accessSAS(connectParameters,selectRecordParameters);
%macro connectDB(num,char);
%put SAS ConnectDB;
%mend connectDB;
%macro selectRecords(num,char);
%put SAS Record Selection;
%mend selectRecords;
%macro doQuery;
* doQuery is the same for all macros;
%connectDB&connectParameters.;
%selectRecords&selectRecordParameters.;
%mend doQuery;
%doQuery;
%mend accessSAS;

%macro accessSQLServer(connectParameters,selectRecordParameters);
%macro connectDB(num,char);
%put SQL Server ConnectDB;
%mend connectDB;
%macro selectRecords(num,char);
%put SQL Server Record Selection;
%mend selectRecords;
%macro doQuery;
* doQuery is the same for all macros;
%connectDB&connectParameters.;
%selectRecords&selectRecordParameters.;
%mend doQuery;
%doQuery;
%mend accessSQLServer;

%accessSAS(connectParameters=(num=1,char=SAS),
selectRecordParameters=(num=2,char=SAS));
%accessSQLServer(connectParameters=(num=1,char=SQLServer),
selectRecordParameters=(num=2,char=SQLServer));

```

Figure 6. Template method pattern example

The template method pattern example can be submitted in base SAS. The main point is that the *accessSAS* and *accessSQLServer* macros have the same nested macros, called *connectDB* and *selectRecords*. Since nested macros emulate methods, the two nested macros are the template methods. If the SAS macro language had the ability to define parents, we would have put these two nested macros into the parents, making sure that all the children macros would have copies. The children macros could optionally choose to go with the default (parent definition) or override the definition with their own unique definition.

In the SAS macro language, we do **not** have parents, but we can manually apply genetic discipline by making sure that the nested macros have the same names and similar (but not identical) function. The advantage of using nested macros with the same name includes ensuring complete coding in complex situations. This example only has two nested macros, but more complex macros could have a larger group of nested macros, perhaps ten or twenty. If we were to add a new macro called *accessOracle*, then we could start with the template provided by the other access macros, and make sure all the nested macros genetically replicated previous work.

ANALYSIS MATRIX: HIGH-VOLUME MACRO PROCESSING

From Shalloway and Trott (2002):

	Analysis Matrix Method Pattern Key Features
Intent	Provide a systematic way to surface varying dimensions of change.
Problem	Variations in the system need to be made explicit so that applying design patterns becomes easier.
Solution	Create a matrix which documents the concepts which are varying. Matrix creation can help the developer discover the concepts that are varying, find points of commonality, and uncover missing requirements.
Consequences	The final design is optimized for the surfaced dimensions, and refactoring may be required if dimensions are added or removed.

Table 7. Analysis Matrix Pattern Key Features (Shalloway and Trott, 2002, pp. 291-302)

For example, Shalloway and Trott (2002) provide the following example analysis matrix which allows an analyst to enumerate combinations of processing rules, in this case applied to shipping:

	U.S. Sales	Canadian Sales	German Sales
Calculate freight	Combination	Combination	Combination
Verify address	Combination	Combination	Combination
Calculate tax	Combination	Combination	Combination
Money	Combination	Combination	Combination
Dates	Combination	Combination	Combination

Table 8. Analysis Matrix Example (Shalloway and Trott, 2002, pp. 298-299)

Note that the fifteen unique combinations (5 rows times 3 columns) are not necessarily one-line formulas or numeric multiplying parameters, but any of the combinations could be a series of complex IF/THEN statements which provide business rules dependent on the specific situation.

A systematic way to proceed would be to write the fifteen programs for each of the fifteen combinations, and from that point discover what could be put into macros and nested macros, and what minimal information would need to be used to distinguish the combinations from each other. That minimal amount of information becomes the macro parameters, which could be character, numeric, an array, or a macro (as shown under the tricks section).

Once the analysis produces that minimal amount of information, the table of varying parameters could be then stored in a SAS dataset, and used to call the macro(s). Storing the parameters in a dataset is an application of the analysis matrix as a creational design pattern, and therefore, in this author's opinion, deserves to be listed as a separate design pattern even though the text does not say so. Keeping the parameter information in a dataset allows the matrix structure to remain intact, and therefore the matrix becomes part of the overall code design. Integration would also result from using any type of array, including a PROC FORMAT.

In the example above, there are only three countries listed, and only five types of shipping considerations. However, using a SAS dataset to hold the parameters could allow for hundreds of countries, and even going to a more detailed level (in the United States, to the level of states, counties, and zip codes). Companies who do high-volume shipping have a complex matrix of rules based on carriers and all sorts of parameters beyond the five types illustrated. Design pattern examples typically show a small portion of code as part of a larger system, and in this case, it is possible to see how these concepts can easily expand to examples far beyond the presented fifteen combination example.

The following code illustrates a way to use a SAS dataset to call macros. Again, the economies of scale are justified when there are many variations, and the amount of code below provides a visual indication of whether or not this type of solution would be useful.

```
data work.shippingMatrix;
  length freight address tax money dates country $64;
  freight = '(a,1,a)'; address = '(b,2,b)'; tax = '(c,3,c)';
  money = '(d,4,d)'; dates = '(e,5,e)';
  country = 'United States of America'; output;
  country = 'Canada'; output;
  country = 'Germany'; output;
run;

%macro processingMacro(freight,address,tax,money,dates,country);
%put Processing Macro &freight. &address. &tax. &money. &dates. &country.;
%mend processingMacro;

%macro analysisMatrix(dataset);
%local systemError datasetID returnCode totalObs;
%local matrixFreight matrixAddress matrixTax matrixMoney matrixDates matrixCountry;
%let systemError = 0;
%let datasetID = 0;
%if %sysfunc(exist(&dataset,DATA)) %then %do;
  * Open the dataset by obtaining a numeric datasetID;
  %let datasetID = %sysfunc(open(&dataset,i));
  %if &datasetID. > 0 %then %put Dataset Successfully Opened: &datasetID.;
  %else %do;
    %put ERROR IN OPENING DATASET &dataset. - DatasetID = &datasetID.;
    %let systemError = 1;
  %end;
%end;
%else %do;
  %put Dataset &dataset. does not exist;
  %let systemError = 1;
%end;
%if not(&systemError.) %then %do;
```

```

* Obtain the number of dataset observations (rows);
%let totalObs = %sysfunc(ATTRN(&datasetID., NLOBS));
%if &totalObs. > 0 %then %put Total Dataset Observations: &totalObs.;
%else %do;
  %put ERROR INSUFFICIENT OBSERVATIONS IN &dataset. -- DatasetID = &datasetID.;
  %let systemError = 1;
%end;
%end;
%if not(&systemError.) %then %do;
  %do counter = 1 %to &totalObs.;
    %let returnCode = %sysfunc(fetchobs(&datasetID., &counter.));
    %if &returnCode. = 0 %then %do;
      %let matrixFreight = %sysfunc(getvarc(&datasetID., %sysfunc(varnum(&datasetID., freight))));
      %let matrixAddress = %sysfunc(getvarc(&datasetID., %sysfunc(varnum(&datasetID., address))));
      %let matrixTax = %sysfunc(getvarc(&datasetID., %sysfunc(varnum(&datasetID., tax))));
      %let matrixMoney = %sysfunc(getvarc(&datasetID., %sysfunc(varnum(&datasetID., money))));
      %let matrixDates = %sysfunc(getvarc(&datasetID., %sysfunc(varnum(&datasetID., dates))));
      %let matrixCountry = %sysfunc(getvarc(&datasetID., %sysfunc(varnum(&datasetID., country))));
      * After the matrix variables have been populated, then call the processing macro;
      %processingMacro(freight=&matrixFreight., address=&matrixAddress., tax=&matrixTax.,
        money=&matrixMoney., dates=&matrixDates., country=&matrixCountry.);
    %end;
  %end;
%end;
%if &datasetID. > 0 %then %do;
  * Close the dataset;
  %let returnCode = %sysfunc(close(&datasetID.));
  %if &returnCode. = 0 %then %do;
    %let datasetID = 0;
    %put Dataset Successfully Closed;
  %end;
%else %do;
  %put ERROR IN CLOSING DATASET &datasetID.;
  %let systemError = 1;
%end;
%end;
%mend analysisMatrix;
%analysisMatrix(dataset=work.shippingMatrix);

```

Figure 7. Analysis matrix pattern example

The analysis matrix pattern example can be submitted in base SAS. The key functions such as OPEN, CLOSE, ATTRN, FETCHOBS, and GETVARC are normally associated with SAS/AF® software, but are included in the base SAS license. The simplified test dataset does not have any variation in the parameters other than country, but that type of variation could include a variable number of named macro parameters depending on the specific country. The code above includes a lot of error condition checking, which is always a good idea even if you expect every situation to pass. Included with the error checking are %PUT notification statements which provides processing messages in the SAS log.

Note that when any functions are run behind the %SYSFUNC parameter, the parameter quotation marks (which you might see in SAS documentation examples) are dropped for enumerated constants, and additionally, variables are implemented using macro variables preceded by an ampersand. The table below illustrates the difference using the OPEN function.

	Enumerated Constant	Variable	Example of Final Form
Without %SYSFUNC	'constant'	Dataset	open(dataset, 'constant')
With %SYSFUNC	Constant	&dataset.	%sysfunc(open(&dataset., constant))

Table 9. Illustration of how to call functions inside %SYSFUNC

GENERAL APPLICATION ADVICE

Normally, someone would think that these seven patterns describe seven different types of programs. However, the accurate application of design patterns is to consider them as ideas which could be used in combination with other patterns. In other words, in a production environment, the patterns are not patterns of specific individual programs (though they could be, and the examples above show how), but instead represent conceptual relationships on how to handle variation through programming.

For example, all the SAS macro language examples used the façade pattern in presenting a simplification of code encapsulated inside a macro. Additionally, as argued, the SAS macro language automatically applies a singleton definition by default. This author believes that scripting languages have these types of restrictions which beneficially force everyone to use basic design patterns every time they code, proving how scripting languages have inherent structural power.

In the general case, it might not be desired to, by default, apply certain design patterns every time, and at that point, the programming challenge would call for more than the default pattern structure written into a scripting language. The optionally licensed SAS/AF® software provides a way to use full object-oriented programming with SAS, and there are other ways including the optional webAF™ software (which uses Java) and SAS® Integration Technologies (which exposes SAS to Windows .NET development).

The SAS macro language has been one of the most basic ways to program applications which use base SAS procedures, functions and statements. However, this author considers the SAS Business Intelligence tools as ways which apply GUI interfaces to control SAS functionality. These other optional products include:

- Enterprise Guide®
- SAS/Warehouse Administrator®
- Enterprise Miner™ software
- SAS Add-in for Microsoft Windows
- SAS Web Report Studio
- SAS Information Delivery Portal

All of these products allow for functionality which is not inherently in the SAS macro language or base SAS, but perhaps could be programmed with great effort. The cost of these additions would be weighed against the benefit of additional features, and the advantage of providing already constructed interfaces (think façade pattern) for common types of problems. Recalling the introduction, additional software adds advantages in three categories: functions, implementation, and structure. It is important to keep up with new software because these packages inherently contain built-in structural (meaning design pattern) advantages. In the fast-paced corporate world, the “make or buy” decision comes up all the time, and this illustration shows the application to software development.

In this paper, we only consider the SAS macro language. Illustrating design patterns proves that the SAS macro language is more than “just a scripting tool”, which can be intelligently used to create sophisticated applications, and this author would argue should be used that way.

CONCLUSION

This paper demonstrated specific design patterns which can be partially or fully implemented with the SAS macro language. The Shalloway and Trott (2002) book is specifically recommended to provide further understanding of how to apply object-oriented concepts, and the examples used in this paper were intended to complement the examples used in the book. This book excels in not only defining and describing patterns, but also providing extensive and detailed coaching, far beyond the advice contained in this paper, on how to best apply design patterns in a production environment. A larger challenge is the original *Design Patterns* (Gamma, et. Al., 1995), a comparatively difficult text because it reads more like a dictionary rather than a training manual. Finally, there are many other object-oriented resources available on the web, and by typing in the keywords design pattern or the specific names of patterns, you can find a lot of potentially useful documentation.

REFERENCES

- Fowler, Martin (1999), *Refactoring: Improving the Design of Existing Code*, Reading, MA: Addison Wesley Longman, Inc.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison Wesley Longman, Inc.
- SAS Institute Inc. (2005), *SAS OnlineDoc® 9.1.3*, 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. (2005), “Macro Architecture in Pictures”, *SUGI Proceedings*, 2005.

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.