Paper 003-2012

Macro Design and Usage in a Multi-Tier Architecture for ETL and Google Visualization API Integration

Manuel Figallo-Monge, DevTech Systems, Inc., Arlington, VA

ABSTRACT

Macros are useful for creating reusable, tried-and-true components that are part of an Extract, Transform, and Load (ETL) process. This paper examines several SAS® macros that interact with one another to form a base ETL architecture, which consists of a macro to extract XML files from a web server, another to transform the XML files into SAS data sets, and yet another to load the data into a data visualization made available by the Google API for Data Visualizations. This data visualization is called a Motion Chart and can be integrated with SAS data sets, as shown by the architecture presented in this paper. The upshot is a highly interactive visualization that plays a dynamic "movie" and allows *end-users* to explore several World Bank indicators over time. These three primary macros either inherit other macros or can be extended with other macros to form larger applications. Ultimately, all of the SAS components in this paper can be housed in a repository to be reused by *macro users* in an organization. Finally, this paper explores baseline macro concepts related to usage, such as macro naming conventions, parameter validation, error handling, and macro parameter data types. These concepts are especially applicable to *macro developers*.

INTRODUCTION

Writing SAS code is difficult, and writing reusable SAS code is even more difficult. The benefits of reusable code, however, include decreased time and errors in the implementation phase of a project as well as increased satisfaction when using them during SAS applications development. The SAS Macro Language is a text-substitution language to facilitate the development of reusable code. Before implementing this code, however, macro developers need to be aware of good design principles, or else, they may produce code that is non-deterministic, inefficient, and, worse, code that is difficult to maintain (let alone reuse). These principles include layering into tiers and decomposition, encapsulation, and inheritance.

A MULTI-TIER ARCHITECTURE FOR SAS MACROS

Tiers are generally divided into a bottom tier for data (Model), a middle tier for driver processing logic (Controller), and, finally, a top tier for data visualizations or presentation (View). These tiers can further be decomposed into functions and variables. This is represented by the following diagram, which will be explained in this paper:

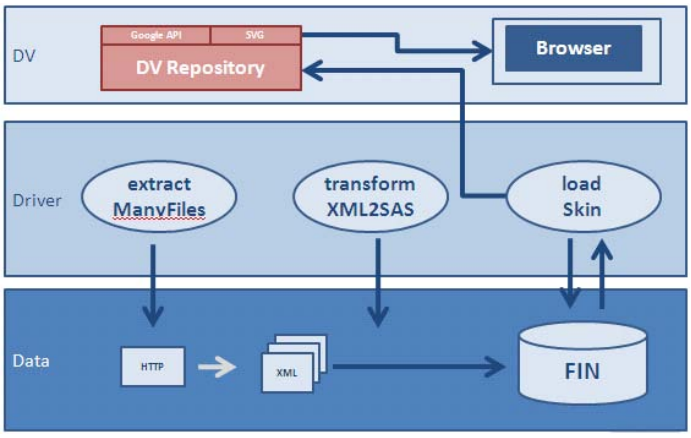

Figure 1. The base macro architecture for ETL and DataVisualizations (DV). This paper examines the %extractManyFiles, %transformXML2SAS, %loadSkin macros in detail. Note that the middle "Driver" tier is where processing logic is maintained and macros are managed.

Decomposition is the breaking up of a problem into smaller components. This is especially well suited for ETL processes, which can be complex. The advantage of decomposing a process is that it highlights the SAS macros that need to be developed in order to facilitate automation or ensure uniformity.

Paper 003-2012

One of the first steps in the ETL processing described in this paper is extracting data from an HTTP server. Figure 1 demonstrates this. The %extractManyFiles macro automates the extraction of files (in one case over 350 files in under 10 minutes) using a RESTful web service or API[1]. This macro's objective is to move data from a web server to a local file system, as demonstrated by Figure 1. Also important to note is that this is accomplished by a macro in the middle tier. In fact, all macros reside in the middle tier, because they manage data between the bottom tier and top tier. They are, moreover, managed and called by a driver file. Note also that software components in the top tier and bottom tier never interact directly with one another directly. It's also the case that a macro can reuse or inherit another macro as demonstrated in a later section. First, it is important to examine a chief advantage of macros, encapsulation.

ENCAPSULATION AND PROGRAMMING TO AN INTERFACE

A macro that downloads or extracts data from an HTTP web server and downloads it locally for processing is a good example of a macro that can be used and reused for ETL processing, since this is a common task. No longer does a SAS programmer need to write code from scratch to accomplish this; instead, all he needs to know is the correct SAS macro function call and its parameters (i.e., a RESTful API call to retrieve indicator data, in this case SP.DYN.TFRT.IN, and the local drive output location):

```
%let myURL="http://api.worldbank.org/countries/all/indicators/SP.DYN.TFRT.IN?per_page=13000";
%let myXMLOutput="C:\xxx\SUG\2012\SAS Global Forum\outputs\SP.DYN.TFRT.IN.xml";
%extractFile(remoteURL=&myURL, localFilename=&myXMLOutput)
```

This highlights one of the primary advantages of using macros: they can "encapsulate" or, that is, hide complexity. The %extractFile macro in this example uses the filename statement (URL access method as shown in Appendix A); this, however, is not important to SAS users who *use* this macro. It is only important to the macro developer who *created and maintains* the macro. A macro *user* only needs to understand what input parameters are required to run the macro and what output to expect. This is otherwise known as programming to an interface.

INHERITANCE

The %extractFile macro can only download one file at a time. It is a very useful macro for downloading either binary or text files, and in the case of text files will issue an HTTP request at least two times to capture HTTP errors and proceed when execution has completed successfully and without warning messages. Despite its utility, %extractFile would be difficult to use when several dozen files have to be downloaded. To handle this situation, SAS makes it possible to effectively pass a data set of values as a macro input parameter and then iterate through it to generate a lot of output in one fell swoop. Whatever the case, note that %extractManyFiles inherits completely from %extractFile. The code for %extractManyFiles is available in Appendix B, and it uses CALL EXECUTE. Inheritance is a way to reuse code of existing macros (i.e., %extractFile) to establish a "submacro" or "child macro" (i.e., %extractManyFiles). SAS, in other words, provides the programming language support that makes this possible. As a result of this feature, only one line is required to download several files:

```
%extractManyFiles(FilesList=wps_in.wb_indicators)
```

This is discussed in further detail below in the section about parameter data types.

EXTENSIBILITY

The %extractManyFiles is very useful; however, it would be better if more could be done with the XML output. To do this, extend %extractManyFiles and its output (in other words, several XML files) with %transformXML2SAS. This latter macro will take an XML file and MAP as input, and then create a SAS data set as output. Creating a MAP is greatly facilitated by the SAS XML Mapper[2], a Java-based, stand-alone application that removes the tedium when creating and modifying an XML Map. Given a list of XML files and their corresponding MAPS, therefore, will easily allow for the creation of SAS data sets from the many XML files:

[1] A RESTful web API is a simple web service implemented using HTTP with a set of operations or methods supported by the web service to produce, as output, an Internet media type such as XML.
[2] To download the latest version of SAS XML Mapper, visit: http://support.sas.com/demosdownloads/setupcat.jsp?cat=Base+SAS+Software

Paper 003-2012

```
data _null_;
    length macrocall $2000.;
    set wps_in.wb_indicators end=final;
    macrocall=catt('%nrstr(%transformXML2SAS(LocalXML="', local, '", XMLMap="', map, '", DSOut=', dsout,'))');
    call execute (macrocall);
run;
```

This call to %transformXML2SAS is from the driver file and CALL EXECUTE will work as seen earlier with %extractManyFiles. Creating a macro that primarily inherits from a base or parent macro and iterates through a data set containing many values is up to the discretion of the macro developer. Using CALL EXECUTE within the %extractManyFiles macro ensures widespread adoption of a frequently used macro, although CALL EXECUTE can just as well be used in a driver file.

Once the XML files have been transformed into SAS data sets, they are concatenated, and transposed long to wide, a point which will be discussed later. Figure 2 shows the data set that is created after a transpose. The %loadSkin macro will use this data set as input and wrap the data set inside the Google API code that is necessary to produce a Motion Chart skin. Thus, the SAS data set output resulting from %transformXML2SAS is extended with a macro that will load a skin and wrap the data with it (i.e., %loadSkin).

	Country	Year	Fertility Rate	Life Expectancy	GDP_growth__annual___	Region
1	Australia	1966	2.881	70.81951	2.370479	East Asia & the Pacific
2	Australia	1967	2.848	70.86927	6.266851	East Asia & the Pacific
3	Australia	1968	2.888	70.91902	5.149561	East Asia & the Pacific
4	Australia	1969	2.886	70.96878	7.046072	East Asia & the Pacific
5	Australia	1970	2.859	71.01854	7.15898	East Asia & the Pacific
6	Australia	1971	2.961	71.06829	4.022716	East Asia & the Pacific
7	Australia	1972	2.744	71.45756	3.938312	East Asia & the Pacific
8	Australia	1973	2.491	71.84683	2.624129	East Asia & the Pacific
9	Australia	1974	2.397	72.2361	4.051269	East Asia & the Pacific
10	Australia	1975	2.148	72.62537	1.2551	East Asia & the Pacific
11	Australia	1976	2.06	73.01463	2.634088	East Asia & the Pacific
12	Australia	1977	2.007	73.34439	3.48312	East Asia & the Pacific
13	Australia	1978	1.949	73.67415	0.903508	East Asia & the Pacific
14	Australia	1979	1.907	74.0039	4.125993	East Asia & the Pacific
15	Australia	1980	1.891	74.33366	3.044066	East Asia & the Pacific
16	Australia	1981	1.935	74.66341	3.403618	East Asia & the Pacific
17	Australia	1982	1.929	74.90488	3.215094	East Asia & the Pacific
18	Australia	1983	1.924	75.14634	-2.31489	East Asia & the Pacific
19	Australia	1984	1.84	75.3878	4.645282	East Asia & the Pacific
20	Australia	1985	1.923	75.62927	5.151181	East Asia & the Pacific

Figure 2. The final data set resulting from extract and transform processing in the Macro Architecture described in this paper

Appendix D show a portion of the %loadSkin macro with several key sections:

- Line 2 inherits %getVariableNames in order to get meta-data from the data set in Figure 3. This information is important since numeric variables are treated differently from character variables in the Google API.
- The output from %getVariableNames is also important for getting variable names and labels. Note that if a macro user loads a Motion Chart and selects the parameter USELABELS=YES but some labels are missing from the data set, an error will be produced (line 31).
- Line 55 contains a macro variable that is derived from a persistence layer described later in this paper. Noticed that line 120 also uses data objects from a persistence layer, discussed later.
- The header for the Motion Chart is produced by the code starting in line 63, and the body starts in line 89.

There are several dependencies between macros in this architecture, as demonstrated by %loadSkin (e.g., %getVariableNames). To download them all, please refer to the end of this paper for an HTTP link.

With additional macros, the architecture described earlier can be extended as in Figure 3. In this new diagram, a macro to transpose long to wide, %transposeL2W, has been added as has the %loadDPL (Data Persistence Layer) macro to demonstrate extensibility (the design principle where the implementation takes into consideration future growth).

Paper 003-2012

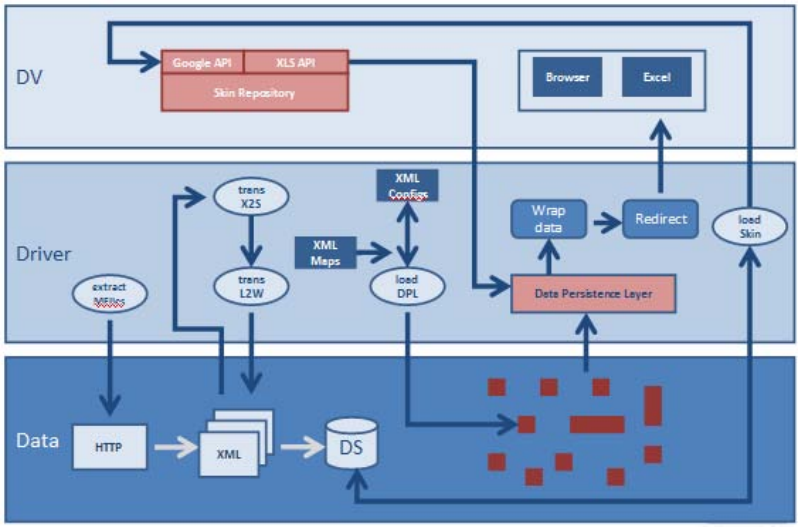

Figure 3. In a multi-tier architecture, 1) macros can be extended through reuse or inheritance, 2) and the architecture can also be extended to include new macros. The light-blue ovals are macros and the blue rounded-squares are functions.

A Data Persistence Layer (DPL) moves data values from a structured and permanent store, such as an XML file or data set, to its most natural form (in memory objects) in order to be assigned to a variable when it is used in generating output. One of the goals of the DPL in this architecture is to localize changes in variable *values* so that they occur in XML configuration files and then made, as needed, into persistent objects.

An advantage of the DPL is that it facilitates management of variables so that any changes to its values occur only to the XML file and SAS macros remain a black box. This means that end-users can participate in customizing their report output or data visualizations through XML files, as discussed later, without involving SAS developers.

A DPL is also very useful for moving data objects from a DPL layer to anywhere in output without using procedures, such as PROC TEMPLATE, etc. The %load2DPL macro is responsible for loading configuration values and mapping them to persistent objects from a configuration XML file and mapping XML file, respectively. Persistent objects are retrieved using the standard ampersand (&); this, in effect, behaves like an accessor function.

For example, the data visualization discussed later in this paper, has a configurable title. It is defined in an XML configuration file as such:

```
<CONFIG>
   <REPORT>mylabreport2_v1</REPORT>
   <METADATA>report_title</METADATA>
   <VALUE>My Motion Chart Title Goes Here</VALUE>
   <NOTES>This title appears in the top of the map</NOTES>
</CONFIG>
```

The %loadDPL macro takes this value from the XML file, after it has been transformed into a data set, and maps it to a persistent object named r_rpttitle2. This is defined in an XML file for mappings as follows:

```
<MAPS>
   <SELECT>Value</SELECT>
   <GVAR>r_rpttitle2</GVAR>
   <DBTABLE>work.Configs2</DBTABLE>
   <FILTER>Report='mylabreport2_v1' AND MetaData='report_title'</FILTER>
</MAPS>
```

As this mapping object shows, the persistent object named r_rpttitle2 is retrieved, as required, from the work.Config2 data set where Report='mylabreport2_v1' and MetaData='report_title'. This persistent object as well as other persistent objects are used extensively by the %loadSkin macro to produce data visualization output. For example:

Paper 003-2012

```
put '    <table width=800>';
put '    <tr>';
put '    <td align=center>';
put '    <font color=gray><b>'"&r_rpttitle2"'</b></font>';
put '    </td>';
put '    </tr>';
put '    </table>';
```

The value of a DPL is immeasurable: data can now be stored and retrieved as required and for a variety of purposes. This very complex Excel spreadsheet, for instance, was created using a DPL implemented in SAS:

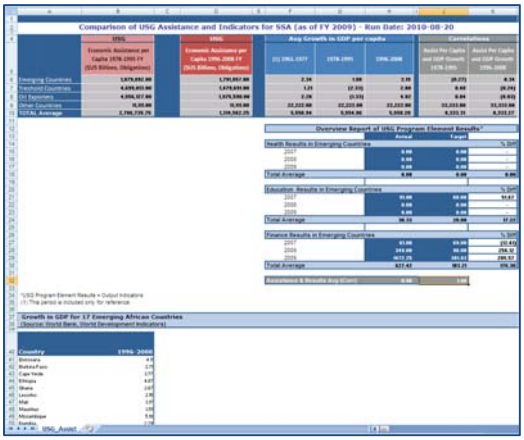

Figure 4. A complex Excel report produced using a DPL.

Demonstrating the value of a DPL is arguably one of this paper's most important contributions. It is one of the key components that enabled and facilitated the integration of SAS with the Google API for Data Visualizations, as described in the next section, whose main output is an HTML file.

DATA VISUALIZATION OUTPUT

The Google API for Data Visualizations is "designed to make it easier for a wide audience to make use of advanced visualization technology, and do so in a way that makes it quick and easy to integrate with new visualizations." One of these visualizations is a Motion Chart. It can show up to 5 dimensions over time and is an excellent tool for exploring data and finding patterns and developing new insights with it. In sum, it makes an enormous amount of data "easily digestible" as shown in Figure 5:

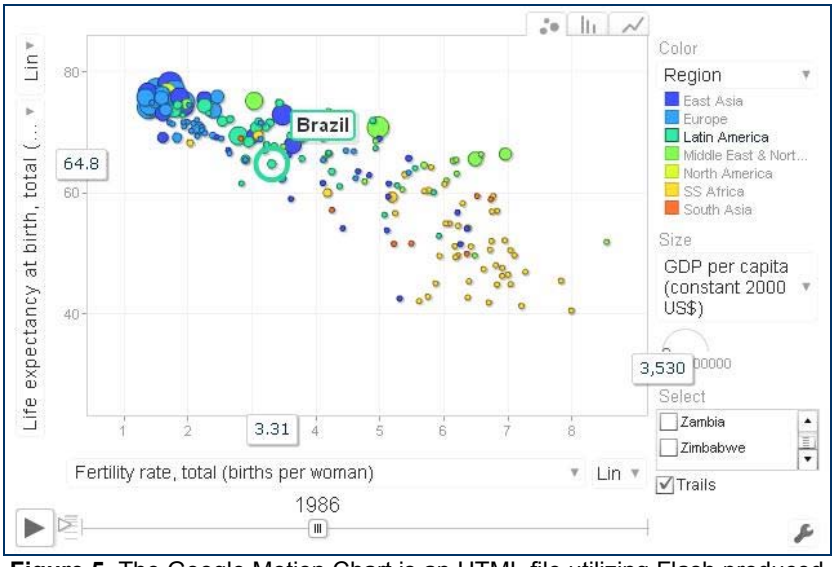

Figure 5. The Google Motion Chart is an HTML file utilizing Flash produced by the SAS macros described in this paper

This is particularly true when it comes to analyzing government indicators, as shown in Figure 6.

Paper 003-2012

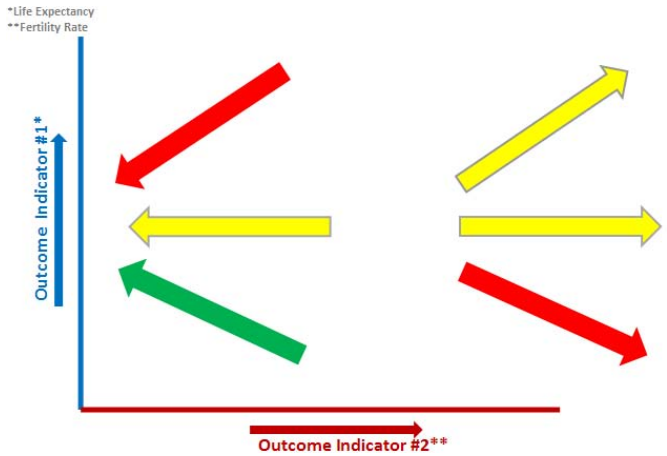

Figure 6. Two government indicators from the Motion Chart above and their relationships (green denotes a desirable outcome, red an undesirable outcome, and yellow value judgments)

As demonstrated in the architecture diagram at the beginning of this paper, the %loadSkin macro is responsible for taking a SAS data set, using the SAS data set's variables in the Google API function data.addColumn, and wrapping it in a presentation layer in order to produce the Motion Chart visualization. It embeds data from the data set as well as the DPL to produce the Google API code partially shown in Figure 7.

```
data.addColumn('string', 'Country');
data.addColumn('number', 'Year');
data.addColumn('number', 'Fertility_rate__total__births_pe');
data.addColumn('number', 'Life_expectancy_at_birth__total');
data.addColumn('number', 'GDP_growth__annual___');
data.addColumn('string', 'Region');
data.setValue(0 , 0, 'Australia' );
data.setValue(0 , 1, 1966 );
data.setValue(0 , 2, 2.881 );
data.setValue(0 , 3, 70.81951 );
data.setValue(0 , 4, 2.370479 );
data.setValue(0 , 5, 'East Asia & the Pacific' );
data.setValue(1 , 0, 'Australia' );
data.setValue(1 , 1, 1967 );
data.setValue(1 , 2, 2.848 );
data.setValue(1 , 3, 70.86927 );
data.setValue(1 , 4, 6.266851 );
data.setValue(1 , 5, 'East Asia & the Pacific' );
data.setValue(2 , 0, 'Australia' );
data.setValue(2 , 1, 1968 );
data.setValue(2 , 2, 2.888 );
data.setValue(2 , 3, 70.91902 );
data.setValue(2 , 4, 5.149561 );
data.setValue(2 , 5, 'East Asia & the Pacific' );
```

Variable names, not labels, are used

The first item must be a string data type, or "Country" in this case, and the second item a Year data type

Figure 7. The format for data required by the Google API, and two functions in the Google API: 1) data.addColumn for variable names; 2) and data.setValue for values

Note that the first "setValue" row in this format must be "Country" and the second "Year". Many other Google API Data Visualizations use this file format. All of this API code, however, remains hidden from the macro user since he or she is only concerned with the macro's interface.

MORE ON THE DRIVER FILE (INPUTS AND OUTPUTS)

The driver is where macros are used and interfaces, especially input parameters, are defined. It is where macro users can, for instance, make a call to %loadSkin:

```
%let myDS=my_wb_data;
%let mySkin=MOTION_CHART;
%let myGAPIOutput="C:\xxxxx\SUG\2012\SAS Global Forum\outputs\motion_chart_out_v1.html";
%loadSkin(dsinput=&myDS, output=&myGAPIOutput, skin=&mySkin)
```

This call creates a Motion Chart; however, it uses variable names from the data set in the output, as demonstrated in Figure 7. To use the labels from a data set, make this call instead (using &myDs and &mySkin from above):

```
%let myGAPIOutput="C:\xxxxx\SUG\2012\SAS Global Forum\outputs\motion_chart_out_v1_1_labels.html";
%let myUSELABELS=YES;
%loadSkin(dsinput=&myDS, output=&myGAPIOutput, skin=&mySkin, USELABELS=&myUSELABELS)
```

Paper 003-2012

The header for %loadSkin provides more details on the input parameters:

```
/*@parameters:                                                                    */
/*      dsinput (string) - The name of the input dataset.                         */
/*      output (string) - The output filename                                     */
/*      skin (string) - The name of the skin                                      */
/*      USELABELS (boolean/optional) - By default, the skin output uses the variable */
/*                      names, USELABEL=YES will instead use labels               */
/*                                                                                */
/*@end                                                                           */
```

By changing the value for the parameter "skin", other output is also possible. A macro user will find it easy to change from a Motion Chart to a Dynamic HTML Table with moveable columns and sortable rows, for instance, with this very similar call:

```
%let myDS=my_wb_data;
%let mySkin=DYNAMIC_HTML;
%let myGAPIOutput="C:\xxxxx\SUG\2012\SAS Global Forum\outputs\dhtml\dhtml_out_v1_1_labels.html";
%let myUSELABELS=YES;
%loadSkin(dsinput=&myDS, output=&myGAPIOutput, skin=&mySkin, USELABELS=&myUSELABELS)
```

The driver plays a particularly important role in managing macro inputs, which would become more important as the architecture described thus far evolves into a framework.

The purpose of a framework is to produce "an abstraction in which software providing generic functionality can be selectively changed by user code, thus providing application specific software". Thus, if implemented correctly, all code changes for the architecture will only occur in the driver file whose purpose is to: 1) define inputs; 2) call reusable macros; 3) and manage macro output. This is rarely the case, however. Most multi-tier architectures still do require code customizations for new and changing requirements, although this should be kept to a minimum.

Whatever the case, in any framework, there are hot spots and frozen spots. "*Frozen spots* define the overall architecture of a software system, that is to say its basic components and the relationships between them. These remain unchanged (frozen) in any instantiation of the application framework [(e.g., SAS reusable macros)]. *Hot spots* represent those parts where the programmers using the framework add their own code to add the functionality specific to their own project."

Generally, a macro user should only update driver files and XML configuration files (hot spots). He can also build macros of his own, as a macro developer, by understanding the naming conventions for macros below, basic parameter validation, error handling, and parameter types beyond character and numeric.

NAMING CONVENTIONS

Choosing a naming convention and reinforcing it within teams is a difficult task and requires support from SAS macro developers. There are advantages to having a naming convention. Chief among them is breeding familiarity with how code, macro or otherwise, is defined and utilized throughout a team or group. This is also true for programmers outside of the SAS population, including web developers. This is one very good reason to choose the mixed case---for example, %extractFile—since it's familiar to web developers and those developers who program in Internet languages. Note the verbNoun syntax identifies what will be accomplished by the macro.

The interface also has a convention. If the parameter is positional, then the value is passed directly. However, in the case of keyword parameters, consider using only variable references as in this example:

```
%let myURL="http://api.worldbank.org/countries/all/indicators/SP.DYN.TFRT.IN?per_page=13000";
%let myXMLOutput="C:\xxx\SUG\2012\SAS Global Forum\outputs\SP.DYN.TFRT.IN.xml";
%extractFile(remoteURL=&myURL, localFilename=&myXMLOutput)
```

Notice that %let statements define the values for variables before the variable reference is passed. This is done for several reasons: it improves readability; and, it helps manage variables (that could potentially be passed to other macro functions) in one single place

Also notice the convention for the variable names. If the variable is used inside a driver or controller file and it is defined by a user, then preface it with "my". This is to distinguish it from other variables in the system such as variables inside the macro itself, which should be prefaced by "this" (to indicate that it is used within "this" macro) and variables that serve as return types, which should be preceded by "r_". While debugging, this naming convention is invaluable towards managing variables in the PC-SAS Explorer Window as well as the Log output.

Paper 003-2012

In sum, the convention is as follows:

- Macro functions use mixedCase or actionverbNoun: e.g., %extractFile.
- Macro variables should be prefaced with "my" and "this" when used in the driver file and inside macros, respectively.
- Return macro variables should be preceded by "r_".

The upshot of this is code with components that are readily identifiable not only by the individual who wrote it but also his or her team. This is especially the case in teams which may consist of developers who don't know SAS but are familiar with object-oriented languages where this type of naming convention is much more ubiquitous.

VALIDATING PARAMETERS

It's important to validate the parameters that are passed to a macro and used by a macro. This will prevent misuse of the macro and provide, moreover, valuable log information to SAS developers who use the macro.

```
%macro extractFile(remoteURL=, localFilename=, TYPE=);
%if &remoteURL= %then %do;
    %put ERROR: A null value is not valid.;
    %put ERROR- Please provide an HTTP location.;
    %return;
%end;
%if &localFilename= %then %do;
    %put ERROR: A null value is not valid.;
    %put ERROR- Please provide an destination location on your C: drive, S: drive, etc.;
    %return;
%end;
```

Figure 8. Basic validation makes a macro more "user-friendly".

SAS provides excellent ways of customizing notes, errors, and warnings in the log. As indicated in Figure 8, use put statements that begin with 'ERROR:', 'WARNING:', or 'NOTE:', and SAS will format the text as an ERROR, WARNING, or NOTE respectively.

ERROR HANDLING

Sometimes running a macro does not go as expected. In the case of the %extractFile, for instance, an "HTTP 500 Internal Server Error" may occur. This happens because the HTTP server can experience a deadlock situation, and one solution is to simply reissue the request for the file on the HTTP server. Dealing with this kind of error is essentially a two step process: first, check for the error; secondly, handle the error.

```
/*For BINARY files use this code*/
%IF &TYPE=BINARY %THEN %DO;
    FILENAME lcl url &remoteURL recfm=s DEBUG;
    FILENAME rmt &localFilename recfm=n;
    DATA _NULL_;
       N=1;
       INFILE lcl NBYTE=n;
       INPUT;
       FILE rmt ;
       PUT _INFILE_ @@;
    Run;
%END;
/*For a text extraction, try executing the extractFile macro only two times*/
%ELSE %DO i = 1 %to 2;
    filename remote url &remoteURL recfm=V debug;
    data _null_;
    nbyte=1;
    infile remote nbyte=nbyte end=done lrecl=999999;
    file &localFilename  lrecl=999999;
    do while (not done);
     input;
     put _infile_ @;
    end;
    stop;
    run;
    filename remote clear;
    %if &syserr=0 %then %return;
%END;
```

2

1

Figure 9. An "HTTP 500 Error" is handled by %extractFile by 1) checking for the error;

and, 2) handling the error. The ELSE statement will execute no more than two times.

Note that SYSERR[3] is an automatic macro variable, which contains a return code status set by the DATA step. It returns 0 if execution completed successfully and without warning messages and the RETURN statement, which causes execution to stop at the current point in the DATA step.

PARAMETER DATA TYPES

Consider the %extractFile macro once again:

- %extractFile(string1, string2)

String1 is the HTTP location of an Excel file (or query string, for that matter) and String2 is the output location on the local drive location for this file. As mentioned, this works very well if there is only one file to extract; however it becomes unwieldy when there are over 100 files to download. The solution is to use a list or data set as a parameter as demonstrated in Figure 10.

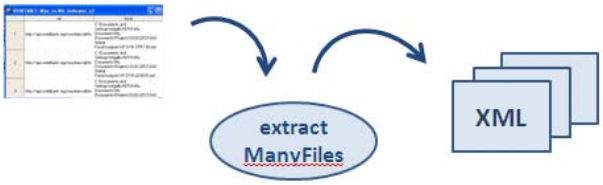

Figure 10. Passing a data set as a parameter to a macro. A macro can iterate through the data set in order to utilize several values as parameters. SAS macro users can also use SAS functions on the data set before it is passed as a parameter.

The data set is effectively passed as a parameter:

- %extractManyFiles(<string1>, <string2>, Data set)

The list or data set appears as follows:

	url	local2
1	http://api.worldbank.org/countries/all/indicators/SP.DYN.TFRT.IN?per_page=13000	C:\SAS Global Forum\outputs\SP.DYN.TFRT.IN.xml
2	http://api.worldbank.org/countries/all/indicators/SP.DYN.LE00.IN?per_page=13000	C:\SAS Global Forum\outputs\SP.DYN.LE00.IN.xml
3	http://api.worldbank.org/countries/all/indicators/NY.GDP.MKTP.KD.ZG?per_page=13000	C:\SAS Global Forum\outputs\NY.GDP.MKTP.KD.ZG.xml

Figure 11. A list or data set that can be passed as a parameter. This entire process can be automated since, in addition to indicator data, indicator names (e.g., SP.DYN,TFRT.IN) used in the URL and local drive output location are also made available through a RESTful web service

Notice that the string variables are optional (hence the brackets "<" and ">"), and they now represent the **names of the fields** in the data set parameter. In the example in Figure 11, that would be "url" and "local2".

To iterate through the data set, simply inherit the %extractFile macro inside %extractManyFiles by using the CALL EXECUTE statement as follows:

```
%macro extractManyFiles(FilesList=, Sources=, Targets=);
    ...
    data _null_;
                set &FilesList end=eof;
                IF _n_=1 THEN call execute('%macro extractManyFiles2;');
                call execute('%extractFile(remoteURL=');
                call execute('"'||strip(&Sources)||'"');
                call execute(', localFilename=');
                call execute('"'||strip(&Targets)||'"');
                call execute(');');
                IF eof THEN call execute('%mend extractManyFiles2;');
    run;
    %extractManyFiles2
%mend extractManyFiles;
```

[3] SYSERR contains a return code status set by FILENAME URL and is oftentimes used as a condition to determine further action. It can also be used to detect major system errors. See: http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/viewer.htm#a000208995.htm

Figure 12. CALL EXECUTE iterates through a data set as shown in Appendix B.

Interestingly, %extractFile is left completely intact. In effect, %extractManyFiles inherits %extractFile which serves as the base macro, as mentioned earlier. If macro users do not like having several macros for similar purposes, one option is to create a super macro with a parameter list to accommodate several macros. So, for instance, %extractFiles(string1, string2, Data set, Boolean) would have a Boolean used to conditionally execute %extractFile or %extractManyFiles as demonstrated by this pseudocode:

IF Boolean="TRUE" THEN %extractManyFiles()
ELSE %extractFile()

If the Boolean is "TRUE", then it is imperative to use a data set as parameter. In other words, care must be taken to validate the parameter list appropriate to the selected macro.

This design can coexist with the earlier one; that is to say, all of these macros can be made available to developers in a repository, and if enhancements are made to the core macro, %extractFile, the changes will propagate to all variations of that macro that utilize the core one. Of course, mistakes to the core macro would also propagate, so enhancements must be tested thoroughly whenever they are made.

One other underutilized parameter data type is arrays. This can be created by passing a string with a delimiter, since the SAS array statement does not compile or execute when passed as a parameter. Here again, it's important to decide on a convention. Using a delimiter that's infrequently used is recommended. For example, here is an array named myxmlmap_arr that uses an '*' delimiter:

```
%let myxmlconfigfile="C:\temp\config\configurations2_v1.xml";
%let myxmlmapfile="C:\temp\config\mappings_v1.xml";
%let myxmlmap_arr=SELECT*GVAR*DBTABLE*FILTER;
%loadDPL(xmlconfig=&myxmlconfigfile, xmlmaps=&myxmlmapfile, xmlmaps_arr=&myxmlmap_arr)
```

As shown above, %loadDPL uses an array to describe and define the MAP that will serve as an intermediate or definition file for data objects as they are moved from a permanent data store to objects in memory.

When passed to a macro, it can be iterated in several ways:

- The standard method is as follows:

  1) First get the size of the "array" (i.e., &xmlmaps_arr whose value is SELECT*GVAR*DBTABLE*FILTER), using countw as such:

  ```
  %let thisXML_vararray=&xmlmaps_arr;
  %let total=%sysfunc(countw(&thisXML vararray1,*));
  ```

  2) Iterate using a do loop:

  ```
  %do i=1 %to &total;
          %let passarray=%scan(&thisXML_vararray,&i,*);
          %put &passarray;
  %end;
  ```

- The iterator can also increment twice (or more) within a single loop using the %eval macro:

  ```
  %do i=1 %to &total;
          %let passarray=%scan(&XML_vararray,&i,*);
          %put &passarray;

          %let i=%eval(&i+1);

  %end;
  ```

- If the array needs to be converted to a list (to be used with an IN statement, for instance), use the tranwrd function which replaces all occurrences of '*' with a comma and double quote:

  ```
  %let myStaticVarsList=Region*country_name*program_name;
  %let thisstaticvarslist="%sysfunc(tranwrd(&myStaticVarsList,*,%str(",")))";
  %put &thisstaticvarslist;
  ```

  The output of this is: "Region", "country_name", "program_name".

Armed with this knowledge, developers can write macros that are flexible and generate a variety of output based on the input parameters that are provided.

Paper 003-2012

CONCLUSION

As the result of concepts described in this paper, the SAS community can realize the many advantages of a multi-tier macro architecture: code that is easier to maintain, extend, and even reuse. If designed correctly, any application using SAS macros will be nothing more than a black box for an end user, configurable by XML files. Architects, moreover, will be better able to understand and communicate how SAS systems are used in their organizations, since a multi-tier design provides an intuitive and standard means of describing a system implementation. And, finally, macro developers or users will be able to extend and reuse existing macro code to add new behaviors, as needed, through the introduction of additional macros; this will ultimately allow SAS developers to more readily respond to new requirements and incorporate enhancements as needed.

REFERENCES

Carpenter, Art. Carpenter's Complete Guide to the SAS Macro Language, 2nd Edition. SAS Publishing, 2004.

Dorfman, Merlin (Editor) and Thayer, Richard(Editor). Software Engineering. Wiley-IEEE Computer Society, 1996.

Droogendyk, Harry and Fecht, Marje. "Demystifying the SAS® Macro Facility - by Example" http://www2.sas.com/proceedings/sugi31/251-31.pdf

Few, Stephen. Show Me the Numbers: Designing Table and Graphs to Enlighten. Analytics Press, 2004.

Figallo-Monge, Manuel. "Macro Design and Usage in a Multi-Tier Architecture for ETL and Google Visualization API Integration". http://analytics.ncsu.edu/sesug/2011/GH05.FigalloMonge.pdf

Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

Jacobson, Ivar, M. Griss, and P. Jonsson. Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley Professional, 1997.

Pree, W. "Meta Patterns: A Means for Capturing the Essentials of Reusable Object-Oriented Design", Proceedings of the 8th European Conference on Object-Oriented Programming (Springer-Verlag). 1994

SAS Publishing. SAS 9.1 Macro Language Reference. SAS Publishing, 2004.

Wikipedia contributors, "Software Framework," Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Software_framework (accessed March 10, 2012).

ACKNOWLEDGMENTS

The author would like to thank the members of DevTech Systems who provided invaluable insights into the Motion Chart. SAS also provided considerable assistance and proved to be an indispensable and trusted partner in the evolution of ideas describes in this paper. Finally, this paper would not have been the same without the insights from Greg Nelson.

CONTACT INFORMATION

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

Manuel Figallo-Monge
Phone: 202 255-6879
E-mail1: mfigallo@devtechsys.com
E-mail2: mfigallo@alumni.carnegiemellon.edu
Code samples for this architecture: http://www.ocf.berkeley.edu/~mfigallo/sas/global_forum2012/

Paper 003-2012

APPENDIX A. ENCAPSULATION WITH THE EXTRACTFILE MACRO (SANS HEADER)

```
%macro extractFile(remoteURL=, localFilename=, TYPE=);
%if &remoteURL= %then %do;
    %put ERROR: A null value is not valid.;
    %put ERROR- Please provide an HTTP location.;
    %return;
%end;
%if &localFilename= %then %do;
    %put ERROR: A null value is not valid.;
    %put ERROR- Please provide an destination location on your C: drive, S: drive, etc.;
    %return;
%end;
/*For BINARY files use this code*/
%IF &TYPE=BINARY %THEN %DO;
    FILENAME lcl url &remoteURL recfm=s DEBUG;
    FILENAME rmt &localFilename recfm=n;
    DATA _NULL_;
       N=1;
       INFILE lcl NBYTE=n;
       INPUT;
       FILE rmt ;
       PUT _INFILE_ @@;
    Run;
%END;
/*For a text extraction, try executing the extractFile macro only two times*/
%ELSE %DO i = 1 %to 2;
    filename remote url &remoteURL recfm=V debug;
    data _null_;
    nbyte=1;
    infile remote nbyte=nbyte end=done lrecl=999999;
    file &localFilename  lrecl=999999;
    do while (not done);
     input;
     put _infile_ @;
    end;
    stop;
    run;
    filename remote clear;
    %if &syserr=0 %then %return;
%END;

%mend extractFile;
```

Paper 003-2012

APPENDIX B. INHERITANCE WITH THE EXTRACTMANYFILE MACRO (SANS HEADER)

```
%macro extractManyFiles(FilesList=, Sources=, Targets=);
    options mprint mlogic;
    %if &FilesList= %then %do;
        %put ERROR: A null value is not valid.;
        %put ERROR- Please provide a sas dataset with a source to target list.;
        %return;
    %end;
    %if &Sources eq %then %do;
        %LET Sources=sources;
    %end;
    %if &Targets eq %then %do;
        %LET Targets=targets;
    %end;
    data _null_;
                set &FilesList end=eof;
                /*use call execute to iterate through a dataset of parameter values*/
                IF _n_=1 THEN call execute('%macro extractManyFiles2;');
                /*inherit from extractFile*/
                call execute('%extractFile(remoteURL=');
                call execute('"'||strip(&Sources)||'"');
                call execute(', localFilename=');
                call execute('"'||strip(&Targets)||'"');
                call execute(');');
                IF eof THEN call execute('%mend extractManyFiles2;');
    run;
    %extractManyFiles2
%mend extractManyFiles;
```

APPENDIX C. EXTENSIBILITY WITH THE TRANSFORMXML2SAS MACRO (SANS HEADER)

```
%macro transformXML2SAS(LocalXML=, XMLMap=, DSOut=);
/*xml_root_tmp will scan the MAP file for the name of the ROOT node, used in the set statement
when the SAS dataset is created*/
    data xml_root_tmp;
        infile &XMLMap truncover end=done;
        input col1 $200.;
        length filen $20;
        filen=compress(filen,'09'x);
        if col1 =: "<TABLE name=" then filen=strip(translate(strip(scan(col1,2,'=')),'','">'));
        if done then output;
        retain filen;
        drop col1;
        call symputx('r_tbl_name',trim(filen),'G');
    run;

    %put &r_tbl_name;
    %put &r_tbl_name;
    %put &r_tbl_name;
    %put &r_tbl_name;


    filename  XML_FN &LocalXML;
    filename  MAP_FN &XMLMap;
    libname   XML_FN xml xmlmap=MAP_FN access=READONLY;

    data &DSOut;
        set XML_FN.&r_tbl_name;
    run;
%mend;
```

Paper 003-2012

APPENDIX D. EXTENSIBILITY WITH THE LOADSKIN MACRO (SANS HEADER, MOTION CHART ONLY)

```
1         %let this_uselabels=&USELABELS;
2         %getVariableNames(DSIn=&thisDS)
3         data &thisDS2;
4             set &this_DSOut;
5             IF TYPE='Num' THEN TYPE_GAPI='number';
6             IF TYPE='Char' THEN TYPE_GAPI='string';
7             col_GAPI=Num-1;
8         run;
9         proc sql noprint;
10         select variable into :r_varlist separated by ' '
11         from &thisDS2;
12         select type_gapi into :r_typlist separated by ' '
13         from &thisDS2;
14         select NUM into :r_numlist separated by ' '
15         from &thisDS2;
16         select Label into :r_labels separated by '*'
17         from &thisDS2;
18        quit;
19        %IF &this_uselabels=YES %THEN %DO;
20            proc sql noprint;
21                 select count(*) into :r_labels_err
22                 from &thisDS2
23                 where Label=""
24                ;
25            quit;
26            data _null_;
27                if &r_labels_err = 0 then do;
28                    put "NOTE: All variables/columns have labels!";
29                end;
30                else do;
31                    put 'ERROR: Stopped processing because there are variables/columns with missing labels';
32                    abort cancel;
33                end;
34                stop;
35            run;
36        %END;
37        %let r_count=%sysfunc(countw(&r_varlist));
38        %let r_count_labels=%sysfunc(countw(&r_labels,'*'));
39        proc sql noprint;
40         select count(*) into :max_rows
41         from &thisDS;
42        quit;
```

1 Get variable names or labels and types

2 Validation. Stop processing if a user has chosen to use labels and some are missing from the data set.

Paper 003-2012

```
44          filename out_GAPI &output;
45          data _null_;
46             set &thisDS end=eof;
47             FILE out_GAPI LRECL=999999;
48             if _n_=1 then do;
49
50                put '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtm
51                put '<html xmlns="http://www.w3.org/1999/xhtml">';
52                put '<head>';
53                put '<meta http-equiv="content-type" content="text/html; charset=UTF-8'
54                /*TITLE*/
55                put "<title> " "%sysfunc(strip(&r_mctitle))" " </title>";
56                put '<script type="text/javascript" src="http://www.google.com/jsapi">
57                put '<script type="text/javascript">';
58                put "google.load('visualization', '1', {packages: ['motionchart']});";
59                put /;
60                put 'function drawVisualization() {';
61                put 'var data = new google.visualization.DataTable();';
62                /*HEADER*/
63              %IF &this_uselabels=YES %THEN %DO;
64                put "data.addRows(" "%sysfunc(strip(&max_rows))" ");";
65                %do i=1 %to &r_count_labels;
66                    %if &i ne &r_count_labels %then %do;
67                        put "data.addColumn('" "%scan(&r_typlist,&i)" "', '" "%scan(&r_labels,&i,'*')" "');";
68                    %end;
69                    %else %if &i eq &r_count_labels %then %do;
70                        put "data.addColumn('" "%scan(&r_typlist,&i)" "', '" "%scan(&r_labels,&i,'*')" "');";
71                    %end;
72                %end;
73              %END;
74              %ELSE %DO;
75                put "data.addRows(" "%sysfunc(strip(&max_rows))" ");";
76                %do i=1 %to &r_count;
77                    /*                      */
78                    %if &i ne &r_count %then %do;
79                        put "data.addColumn('" "%scan(&r_typlist,&i)" "', '" "%scan(&r_varlist,&i)" "');";
80                    %end;
81                    %else %if &i eq &r_count %then %do;
82                        put "data.addColumn('" "%scan(&r_typlist,&i)" "', '" "%scan(&r_varlist,&i)" "');";
83                    %end;
84                %end;
85              %END;
86             end;
```

3 Produce the API title and load the API

4 Generate the API code for data.addColumn using variables or labels from the data set

Paper 003-2012

```
87              /*BODY*/
88             /*iterate through dataset except last row*/
89             if not eof then do;
90                             /*get row counter - 1*/
91                             j=strip(input(_n_,12.))-1;
92                             put
93                             %do i=1 %to &r_count;
94                                  /*output format for strings*/
95                                  %if &i le &r_count and %scan(&r_typlist,&i)=string %then %do;
96                                      "data.setValue(" j ", %eval(&i-1), '" %scan(&r_varlist,&i) +(-1)"' );"/
97                                  %end;
98                                  /*output format for number*/
99                                  %else %if &i le &r_count and %scan(&r_typlist,&i)=number %then %do;
100                                     "data.setValue(" j ", %eval(&i-1), " %scan(&r_varlist,&i) +(-1)" );"/
101                                 %end;
102                            %end;;
103            end;
104            /*iterate through last row*/
105            else if eof then do;
106                            j=strip(input(_n_,12.))-1;
107                            put
108                            %do i=1 %to &r_count;
109                                 %if &i le &r_count and %scan(&r_typlist,&i)=string %then %do;
110                                     "data.setValue(" j ", %eval(&i-1), '" %scan(&r_varlist,&i) +(-1)"' );"/
111                                 %end;
112                                 %else %if &i le &r_count and %scan(&r_typlist,&i)=number %then %do;
113                                     "data.setValue(" j ", %eval(&i-1), " %scan(&r_varlist,&i) +(-1)" );"/
114                                 %end;
115                            %end;;
116                put /;
117                /*call variables from DPL for footer here:*/
118                put 'var motionchart = new google.visualization.MotionChart(';
119                put "document.getElementById('visualization'));";
120                put "motionchart.draw(data, {'width': " "%sysfunc(strip(&r_mcwidth))" ",
121                    'height': " "%sysfunc(strip(&r_mcheight))" ", 'showChartButtons': "
122                    "%sysfunc(strip(&r_mcshowChartButtons))" ", 'showSidePanel': "
123                    "%sysfunc(strip(&r_mcshowSidePanel))" "});";
124                put ")";
125                put "google.setOnLoadCallback(drawVisualization);";
126                put "</script>";
127                put "</head>";
128                put '<body style="font-family: Arial;border: 0 none;">';
129                put "<div id='visualization' style='width: " "%sysfunc(strip(&r_mcwidth))"
130                    "px; height: " "%sysfunc(strip(&r_mcheight))" "px;'></div>";
131                put '</body>';
132                put '</html>';
133            end;
134        run;
```

5

Generate API calls for data.setValue using data from a SAS data set

6

Produce the footer using variables from the DPL (Data Persistence Layer)

Paper 003-2012

APPENDIX E. MOTION CHART SCREENSHOTS