Paper GH-05

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

A multi-tier application architecture separates data management, application processing, and presentation. This paper shows how to facilitate this separation of concerns by producing loosely coupled macro components with specified purposes that also interact with one another. It examines an ETL batch file written so that one macro extracts spreadsheets from a web server, a second macro transforms the spreadsheets, and yet another loads them into datasets. This data processing is extended with a presentation layer macro which integrates SAS© with the Google Visualization API to produce a highly interactive motion or bubble chart that "plays" a dynamic "movie" to explore several U.S. federal government indicators over time. Ultimately, all of these SAS components can be housed in a repository in order to be reused by developers in an organization.

INTRODUCTION

Writing SAS code is difficult, and writing reusable SAS code is even more difficult. The SAS Macro language is a text-substitution language to facilitate the development of reusable code. Before implementing this code, however, programmers 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).

A MULTI-TIER ARCHITECTURE FOR SAS MACROS

Tiers are generally divided into a bottom tier for data (Model), a middle tier for logic (Controller), and, finally, a top tier for 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:

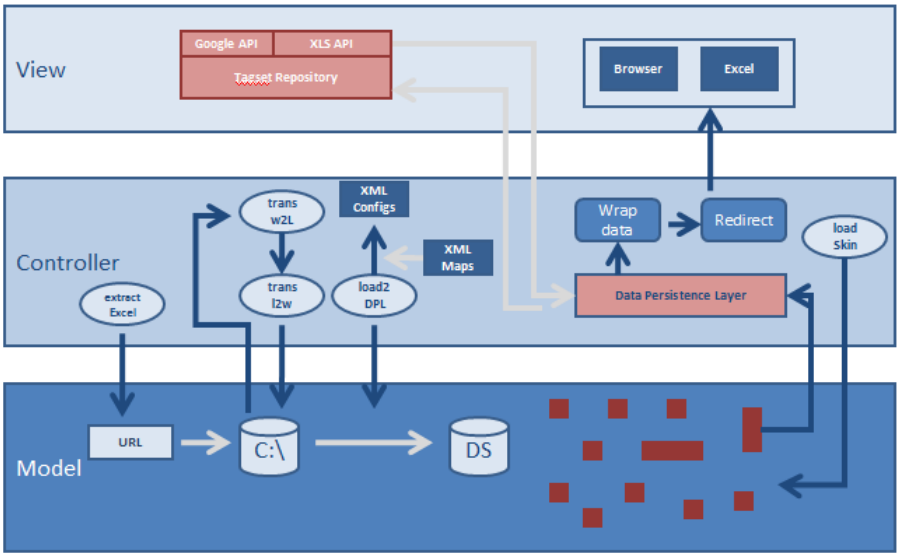

Figure 1. A Model View Controller Architecture for ETL and Visualizations. This paper examines the extractExcel, load2DPL, and loadSkin macros in detail.

Decomposition is the breaking up of a problem into smaller components. This is especially well suited for ETL (Extract, Transform, and Load) processes, which can be complex. The advantage of decomposing a process is that

it highlights the SAS macros that need to be developed to facilitate automation or uniformity. For example, one of the first ETL steps in the diagram above involves extracting data from an HTTP server. This is demonstrated by the following diagram:

Figure 2. Model and Controller Interactions with the extractExcel macro

Note that data is moved from a web server to a local file system, as demonstrated by the diagram. 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 which can also be referred to as a "Controller". Note, however, that software components in the top tier and bottom tier never interact with one another directly.

PROGRAMMING TO AN INTERFACE

A macro that downloads or extract data from an HTTP webserver (URL in Figure 2) 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:

```
%let mySources="http://gbk.eads.usaidallnet.gov/data/files/us_economic_assistance.csv";
%let myTargets="C:\temp\us_economic_assistance.csv";
%extractExcel(remoteURL=&mySources, localFilename=&myTargets)
```

Note the naming convention and "mixed case":

```
%extractExcel
```

NAMING CONVENTION

Choosing a naming convention and reinforcing it within teams is a difficult task and requires support from SAS developers. There are advantages to having a naming convention: chief among them is breeding familiarity with how code is defined and utilized throughout a team or group. This is also true for programmers outside of the SAS population. This is one very good reason to choose the mixed case---for example, extractExcel—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 mySources="http://gbk.eads.usaidallnet.gov/data/files/us_economic_assistance.csv";
%let myTargets="C:\temp\us_economic_assistance.csv";
%extractExcel(remoteURL=&mySources, localFilename=&myTargets)
```

Notice that %let statements define the values for variables before the variable reference is passed. This is done for several reasons:

- It improves readability
- 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 by "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 Explorer window as well as the Log output in SAS 9.2.

In sum, the convention is as follows:

- Macro functions use mixedCase: e.g., extractExcel
- Macro variables should be prefaced with "my" and "this" when used in the Controller (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.

PARAMETER DATA TYPES

Consider the extractExcel macro once again:

- extractExcel(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 C: drive for this file. 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 object or dataset. The dataset is effectively passed as a parameter as such:

- extractManyExcels(<string1>, <string2>, List)

The list object appears as follows:

location	outputfile
http://esdb.eads.usaidallnet.gov/query/do?_progran	C:\temp\output\pink_sheet.xls
http://esdb.eads.usaidallnet.gov/query/do?_progran	C:\temp\output\life_expect.xls
http://esdb.eads.usaidallnet.gov/query/do?_progran	C:\temp\output\fert_rate.xls
http://esdb.eads.usaidallnet.gov/query/do?_progran	C:\temp\output\gdp_per_cap_constant.xls
http://gbk.eads.usaidallnet.gov/data/files/us_econo	C:\temp\output\us_economic_assistance.csv
http://esdb.eads.usaidallnet.gov/query/do?_progran	C:\temp\output\population_total.xls

Figure 3. A List object that can be passed as a parameter

Notice that the string variables are optional. That is because they now represent the names of the fields in the List object containing: 1. all of the source HTTP locations; and, 2) their corresponding output location on the local file system. In the example in Figure 3, that would be "location" and "outputfile"

To iterate through the List, simply use the call execute statement as follows:

```
%macro extractManyExcels(ExcelsList=, Sources=, Targets=);

options mprint mlogic;

                data _null_;
                  set &thisExcelsList end=eof;
                  IF _n_=1 THEN call execute('%macro extractExcels2;');
                  call execute('%extractExcel(remoteURL=');
                  call execute('"'||strip(&thisSources)||'"');
                  call execute(', localFilename=');
                  call execute('"'||strip(&thisTargets)||'"');
                  call execute(');');
                  IF eof THEN call execute('%mend extractExcels2;');
                run;
%extractExcels2
%mend extractExcels;
```

Figure 4. call execute iterates through a List object. Note that "this" is prepended to variables used inside a macro.

Interestingly, extractExcel is left completely intact, and instead all values in the LIST are passed as a parameter as a result of using call execute. In effect, extractManyExcels inherits extractExcel which serves as the base macro.

```
%macro extractExcel(remoteURL=, localFilename=);
    filename remote url &remoteURL recfm=s
    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 _all_ clear;
%mend extractExcel;
```

Figure 5. extractExcel serves as the base macro for more advanced macros.

If macro developers 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, extractExcelFile(string1, string2, List, Boolean) would have a Boolean used to conditionally execute extractExcel or extractManyExcels as demonstrated by this pseudocode:

IF Boolean="TRUE" THEN extractManyExcels()
ELSE extractExcel

If the Boolean is "TRUE", then it is imperative to use the List object. 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, extractExcel, the changes will propagate to all variations of that macro that utilize the core one. Of course, mistakes to the core macro would also propogate, 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)
```

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", 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), this code will work:

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

  This is the log output:

```
      %put &thisstaticvarslist;
"Region","country_name","program_name"
```

INPUTS AND OUTPUTS

In a multi-tier architecture, keeping track of variables, particularly those passed as parameter input, can be difficult. This is one reason naming conventions are used. It's also why keyword parameters in this paper use variable references (instead of hard-coded values) that are declared by %LET statements. Recall that this is particularly relevant regarding the "Controller". The "Controller" plays a particularly important role in managing macro inputs, that is because the architecture described thus far is also a framework in its nascent stages

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 Controller.sas 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 customizations for new and changing requirements. In any framework, therefore, 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."

The "Controller", therefore, is a good place for new code as well as changes to variables' values; however, it's not the only place in this architecture. Specifically, changes in variable *values* can also occur in configuration files, and that's one of the goals of the Data Persistence Layer (DPL) in this architecture. A DPL moves data values from a structured and permanent store, such as an XML file or dataset, to its most natural form (in memory object) to be assigned to a variable when it is used in generating output.

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 so that SAS macros remain a black box. This means that end-users can participate in customizing their report output or data visualizations, 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 that is used to control changes to a variable.

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 and maps it to a persistent object named r_rpttitle2. This is defined in an XML 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>
```

It, along with all of the other persistent objects are retrieved as required. This is how it is used by the loadSkin macro before it produces the visualization output:

```
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:

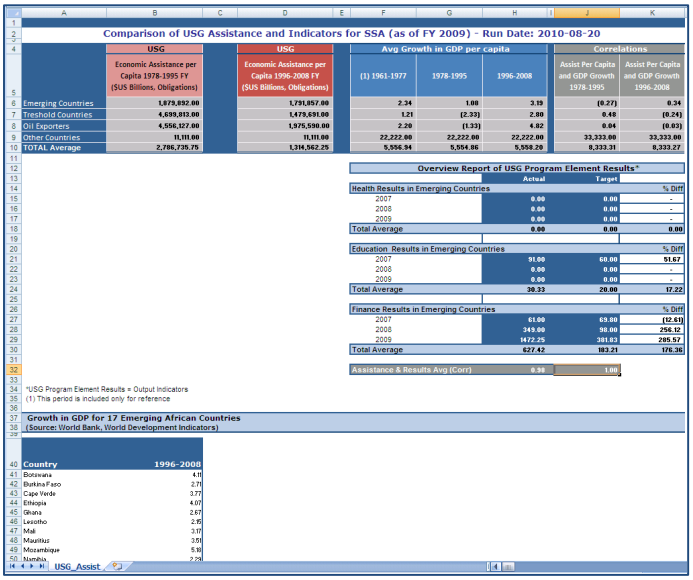

Figure 6. A complex Excel report produced using a DPL

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 7:

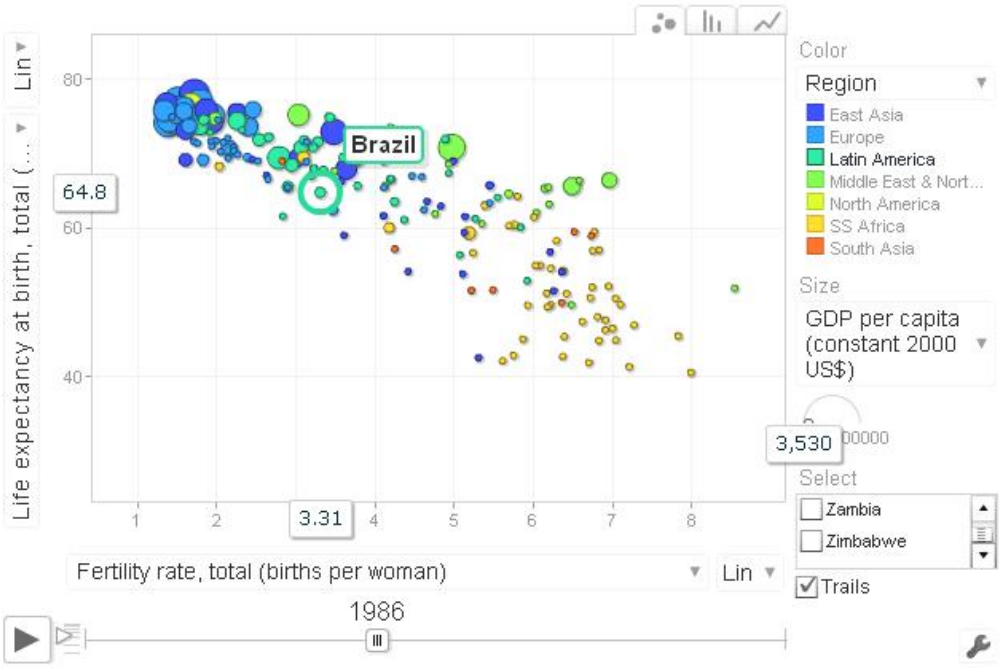

Figure 7. The Google Motion Chart is an HTML file produced

by the SAS macros described in this paper

This is particularly true when it comes to analyzing government indicators, as shown in Figure 8

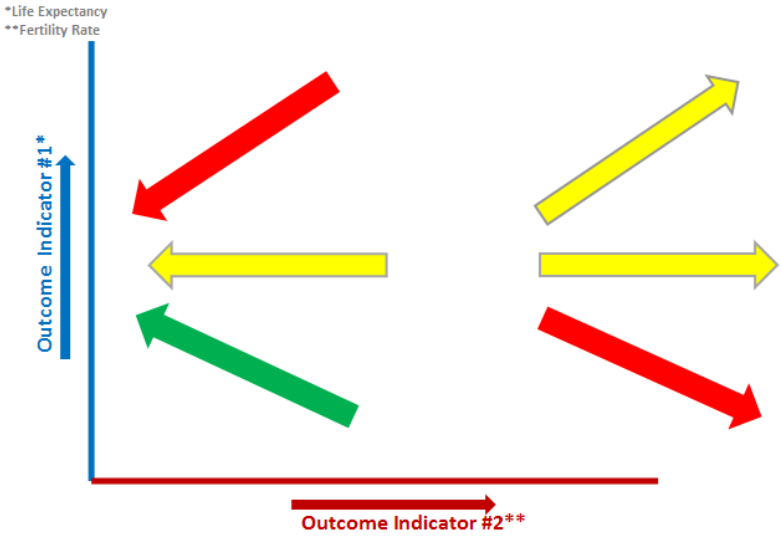

Figure 8. Two Government Indicators from the Motion Chart above and their relationships (green denotes a desirable outcome, red an undesirable outcome, and so on)

As demonstrated in the architecture diagram at the beginning of this paper, the loadSkin macro is responsible for taking a dataset with the following fields and wrapping it in a presentation layer in order to product the motion chart visualization:

	country_name	year	MRA_per_Capita	NC_per_Capita	PC_per_Capita	Region
1	Armenia	2004	0.1306402286	0.5693297895	0.5238673166	Europe & Central Asia
2	Azerbaijan	2004	0.0618960079	0.193557859	0.1596359434	Europe & Central Asia
3	Azerbaijan	2005	0.0279825068	0.0662734677	0.127327824	Europe & Central Asia
4	Azerbaijan	2006	0.0441470673	0.132398654	0.1628271387	Europe & Central Asia
5	Azerbaijan	2008	0.0980812433	0.0122833838	0.1559374892	Europe & Central Asia
6	Azerbaijan	2009	0.1619387093	0.2313164638	0.249399278	Europe & Central Asia
7	Bangladesh	2001	0.0011166277	0.0009770493	0.0063368624	South Asia
8	Costa Rica	1999	0.0606471546	0.3019343319	0.1056770162	Latin America & the Caribbean
9	Costa Rica	2005	0.0788751033	0.102057083	0.3124419815	Latin America & the Caribbean
10	Costa Rica	2006	0.0321676467	0.0369734567	0.3048794498	Latin America & the Caribbean
11	Costa Rica	2008	0.0581749657	0.0010521946	0.4003982186	Latin America & the Caribbean
12	Ecuador	2003	0.0138287927	2.4187587202	0.2343133367	Latin America & the Caribbean
13	Ecuador	2004	0.0099153526	2.4715202976	0.2318226469	Latin America & the Caribbean
14	Ecuador	2006	0.075740144	2.0222752521	0.206250107	Latin America & the Caribbean
15	Ecuador	2007	0.0966884795	1.3208637174	0.2123093878	Latin America & the Caribbean
16	Ecuador	2008	0.1802424581	0.0291912783	0.196618918	Latin America & the Caribbean
17	Ecuador	2009	0.4735246478	0.0328999435	0.2561454918	Latin America & the Caribbean
18	El Salvador	2006	0.0016442763	0.1154591074	0.4169901095	Latin America & the Caribbean
19	El Salvador	2007	0.0130846778	0.2375624329	0.4148831107	Latin America & the Caribbean
20	El Salvador	2009	0.0438094775	0.5020932817	0.509487997	Latin America & the Caribbean
21	Georgia	2004	0.3332689318	0.4197393487	0.3374992918	Europe & Central Asia
22	Georgia	2005	0.2253769311	0.6207016037	0.2578823857	Europe & Central Asia

Figure 9. The final dataset resulting from ETL processing in the Macro Architecture described in this paper

The loadSkin macro takes this dataset (Figure 9) and produces data in the format required by the Google API. This format is shown in Figure 10.

```
data.addColumn('string', 'country_name');
data.addColumn('number', 'year');
data.addColumn('number', 'MRA_per_Capita');
data.addColumn('number', 'NC_per_Capita');
data.addColumn('number', 'PC_per_Capita');
data.addColumn('string', 'Region');
data.addRows([
['Armenia ',2004 ,0.1306402286 ,0.5693297895 ,0.5238673166 ,'Europe & Central Asia '],
['Azerbaijan ',2004 ,0.0618960079 ,0.193557859 ,0.1596359434 ,'Europe & Central Asia '],
['Azerbaijan ',2005 ,0.0279825068 ,0.0662734677 ,0.127327824 ,'Europe & Central Asia '],
['Azerbaijan ',2006 ,0.0441470673 ,0.132398654 ,0.1628271387 ,'Europe & Central Asia '],
['Azerbaijan ',2008 ,0.0980812433 ,0.0122833838 ,0.1559374892 ,'Europe & Central Asia '],
['Azerbaijan ',2009 ,0.1619387093 ,0.2313164638 ,0.249399278 ,'Europe & Central Asia '],
['Bangladesh ',2001 ,0.0011166277 ,0.0009770493 ,0.0063368624 ,'South Asia '],
['Costa Rica ',1999 ,0.0606471546 ,0.3019343319 ,0.1056770162 ,'Latin America & the Caribbean '],
['Costa Rica ',2005 ,0.0788751033 ,0.102057083 ,0.3124419815 ,'Latin America & the Caribbean '],
['Costa Rica ',2006 ,0.0321676467 ,0.0369734567 ,0.3048794498 ,'Latin America & the Caribbean ']
])
```

Figure 10. The format for data required by the Google API

The first field in this format must be string and the second Year. Many other Google API data visualizations use this file format in addition to a variety of API calls. A submacro used by loadSkin is getSkinData which consists of the following SAS code (and can be reused for other Google API visualizations):

```
%do i=1 %to &count;
 put "data.addColumn('" "%scan(&typlist,&i)" "', '" "%scan(&varlist,&i)" "');";
 %end;
put 'data.addRows([';
end;
    if not eof then do;
        put "[" %do i=1 %to &count;
         %if &i ne &count and %scan(&typlist,&i)=string %then %do;
             "'"     %sysfunc(trim(%scan(&varlist,&i)))"',"
         %end;
         %else %if &i ne &count and %scan(&typlist,&i)=number %then %do;
            %scan(&varlist,&i)","
         %end;
         %if &i eq &count and %scan(&typlist,&i)=string %then %do;
            "'"     %scan(&varlist,&i)"'"
         %end;
         %else %if &i eq &count and %scan(&typlist,&i)=number %then %do;
            %scan(&varlist,&i)
         %end;
        %end; "],";
    end;
    else if eof then do;
      put "[" %do i=1 %to &count;
         %if &i ne &count and %scan(&typlist,&i)=string %then %do;
             "'"     %scan(&varlist,&i)"',"
         %end;
         %else %if &i ne &count and %scan(&typlist,&i)=number %then %do;
            %scan(&varlist,&i)","
         %end;

         %if &i eq &count and %scan(&typlist,&i)=string %then %do;
             "'"     %scan(&varlist,&i)"']"
         %end;
         %else %if &i eq &count and %scan(&typlist,&i)=number %then %do;
            %scan(&varlist,&i)"]"
         %end;

        %end;;
      put "])";
```

Figure 11. A code sample from a utility macro which is used by the loadSkin macro, which is its parent. This code produces the Google API data format mentioned earlier.

As mentioned, persistent values are also used to handle configuration details. The rest of the output utilizing the Google API consists of API calls to Javascript hosted by Google.

CONCLUSION

As the result of concepts described in this paper, the SAS community can realize the many advantages of a multi-tier architecture: code that is easier to maintain, extend, and even reuse. If designed correctly, any application using SAS 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, developers 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.

REFERENCES

Committee on International Relations. Committee on Foreign Relations. The Foreign Assistance Act, 2003. http://www.usaid.gov/policy/ads/faa.pdf

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.

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.

SAS 9.1 Macro Language: Reference By SAS Institute, Publishing. http://books.google.com/books?id=aJ4HAOaV7aoC&pg=PA3&dq=SAS+Macro+Language:+Reference&hl=en&ei=ZevFTM25H4Wdlgf4mKzICw&sa=X&oi=book_result&ct=result&resnum=7&ved=0CE0Q6AEwBg#v=onepage&q&f=false

Zender, Cynthia L. "Creating Complex Reports" http://www.lexjansen.com/pharmasug/2008/sas/sa08.pdf

ACKNOWLEDGMENTS

The author would like to thank Frank Martin, the DTUG Study Group, and members of EADS 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.

CONTACT INFORMATION

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

Manuel Figallo-Monge
DevTech Systems, Inc.
1700 N. Moore St, Suite 1720
Arlington, VA 22209
Work Phone: 703 778-2664
E-mail1: mfigallo@devtechsys.com
E-mail2: mfigallo@alumni.carnegiemellon.edu
Code samples for this architecture: http://www.ocf.berkeley.edu/~mfigallo/sas/sesug2011/