Checking Datasets before Submitting Code

Mark Tabladillo, Ph.D., Atlanta, GA

ABSTRACT

This presentation will focus on the specific two-level dataset validation used throughout the application before any code is submitted. The first level checks for dataset availability, and the second level checks for integrity within and among datasets and attributes (variables). Eight classes (based on the Strategy design pattern) use this two-level methodology. Specific examples from the SAS/AF® application will be presented, but the checking methodology is of generic value to any application with submit blocks, and could be implemented using SAS® Macro Language and %SYSFUNC.

Extensive experience with classes or objects or design patterns is not necessary for this talk, but the presentation assumes basic knowledge of the SCL language.

INTRODUCTION

To assist states and countries in developing and maintaining their comprehensive tobacco prevention and control programs, the Centers for Disease Control (CDC) developed the Youth Tobacco Surveillance System (YTSS). The YTSS includes two independent surveys, one for countries and one for American states. A SAS/AF® application was developed to manage and process these surveys. During a four year period, over 1,000,000 surveys have been processed for 35 states and 100 international sites (from 60 countries).

This presentation will focus on checking datasets before submitting code within a SAS/AF application. The SAS/AF application supports information integrity and manages data flow by using SAS/AF procedures (presented in a frame), Windows native procedures, and one graphical control. Whether applied in classes or in Frame-related code, the SCL language provides the power to access and manipulate data.

The next two introductory sections will introduce how to move data from datasets to the SCL environment, and outline: 1) how to access datasets, 2) how to check dataset integrity. After these two introductory sections, the presentation focuses on checking datasets before submission. The checking methodology presented is of general utility for any application used to validate SAS datasets.

ACCESSING SINGLE DATASETS

The application uses the control datasets (either non-modifiable or modifiable) to generate both SCL and submitted SAS code. The term “control” refers to the ability to affect or create either SCL or SAS code. A “control dataset” therefore will affect or create SCL or SAS code from a SAS dataset.

The application uses SCL to read information from the dataset into memory. From that point, the data can be directly used in SCL, or optionally, submitted with base SAS code. Code submission is required for many types of commands, like the data step or most procedures, both of which unfortunately have no direct SCL equivalent. Life would be different if the SAS procedures were made available as inheritable classes.

Creating customized base SAS code from SCL involves sending portions of the programming code to the preview buffer, and then releasing the preview buffer using the SUBMIT CONTINUE and ENDSUBMIT commands. The technique is similar to running a SAS macro; however, there are some differences. First, SCL distinguishes between numeric and character substitution, while macro variables are all assumed to be a character type. Second, the way SCL encapsulates variables, especially in classes, is typically more complex that the two choices of global and local for SAS Macro variables.

Instead of providing a specific example, the following table lists key SCL commands repeatedly used to read SAS datasets and submit customized blocks of code.

Command	Use
SUBMIT	Allows sending only part of a command to the preview buffer, even allowing you to send part of a line (without the semicolon)
ENDSUBMIT	Marks the end of a block of text sent to the preview buffer
OPEN	Opens the dataset
ATTRN, ATTRC	Obtains information about the dataset, specifically NLOBS, the number of non-deleted observations
VARNUM	Determines the variable number (order) given the variable name
FETCHOBS	Obtains one observation from the dataset
GETVARN	Obtains the numeric value
GETVARC	Obtains the character value
CLOSE	Closes the dataset (allow the dataset lock to be released and allowing the LIBNAME to be cleanly reset)
SUBMIT CONTINUE	Releases all the code in the preview buffer for execution

Typically, the information was read (by a non-visual object) directly from the dataset and sent straight to the preview buffer. However, sometimes it was efficient to save dataset information or values in one or more SCL lists for repeated use throughout the SCL code. Within a class, the preferred saving method was multiple SCL lists, rather than attempting to create an XML-type nested SCL list (which is more prone to coding errors and maintenance headaches); rather, SAS already provides a way to easily access and reference a nested SCL list, and that way is called a class, and sometimes classes save dataset information in other classes.

SINGLE DATASET INTEGRITY OVERVIEW

The last section outlines how to access a single dataset. Another aspect of control is control dataset integrity, referring to having valid values inside the fields of a single dataset. Validity checks were all performed in SCL, as prerequisites to running processes, or in the Frame-related code.

In some cases the application applies checks when the data is input into SAS format from (for example) ASCII or Excel. A datasets tab (on the Frame) was created for importing and exporting control datasets to and from various formats. That tab also has a visual SAS data grid component, which can be used to modify the dataset. However, our experience has been that Microsoft Excel is generally less prone to crash (since touching some spots around the legacy data table component may crash SAS version 8 for Windows). Excel is also more useful because it is not an inherent database software, and therefore does not have the size and type (character or numeric) specifications (which become restrictions) that Microsoft Access or SAS would have. The Excel interface allows for easily reordering variables, renaming columns, or easily resizing character fields.

At several points, individual datasets are checked for integrity, and specifically that certain values and dataset characteristics are valid. For example, a field may be checked for valid codes. Another example is that a code fragment variable is checked to see that the parentheses are balanced (named “code fragment” because the information is submitted behind an IF statement in base SAS). During the initial design phase it's possible and prudent to build in many checks. However, exceptions and anomalies continue to arise, specifically because each survey questionnaire and analysis could potentially be different.

These above checks help insure, when the data are brought into a SAS dataset, that the information is valid. However, this design has an inherent flaw since it only is checking one specific dataset, rather than checking datasets together. Also, it only runs when someone has gone through the process of putting something into SAS format, and it is possible that something should have been brought into a SAS dataset which was not. Therefore, while it is important and essential to have these single dataset integrity checks, there is no way under this design (alone) to catch all possible run-time errors, and the point of this presentation is to present how to check datasets (plural) together.

MULTIPLE DATASET INTEGRITY OVERVIEW

Since the control datasets were used to send information straight to SAS execution, there are many opportunities for the application to crash. Given that the dataset integrity has already been addressed, there are some important checks to make before sending anything to a preview buffer, and these checks focus on what needs to be available for specific processes to run.

The original six base SAS programs were consolidated into five processes (a “process” is defined here as SCL code combined with run-time submitted SAS code). There are five processes because each step represents a stopping point where some amount of analyst checking needs to be done to insure that the final output is as expected. Those analyst checks include reports intentionally created to look for potential anomalies in the survey data, and for conditions under which the original sample design might be violated. For example, the users check that the expected number of surveys had indeed been scanned in (typically the survey is administered on bubble scantron-type forms) and match to the original school roster; this type of check needs to be done manually and not automated because there are numerous ways to verify whether the expected or scanned data numbers are wrong, including possibly recounting original scantron sheets.

Internally, each of the five core processes has a similar SCL checking structure for possible early termination, and this paper focuses on that preprocess checking structure.

First, the process checks that all accessed datasets exist and can be exclusively opened, as tested by the SCL OPEN function. If the dataset does not exist, sometimes the program will copy the standardized master copy; in other cases, there is no standardized dataset, and the program will terminate early with an error which indicates what datasets are either missing or unavailable. However, the expected situation will be that the dataset does exist and is available for exclusive use, and therefore can be locked. This locking only affects the regional level datasets, and therefore two separate analysts could be working on different regions of the same survey without encountering a locking error.

Second, the program checks for the existence of certain variables within each dataset, as tested by a nonzero return code from the VARNUM command. Each type of dataset has to have standardized names for this type of check to work. For example, the questionnaire layout file needs to have the variable "QUESTION" as a character type. The program looks for assigned standardized variable names and expected variable types (numeric or character). If any expected variable is not present or is of the wrong type, the program terminates early with an error.

Over time, variables were sometimes added to the control datasets in two categories, either required or optional. New required variables cause early termination if not present (just as any required variable would).

A new optional variable, if present, will trigger perhaps a block of SCL or base SAS code. For example, the survey questionnaire layout file includes an optional table numbers field (or variable). The original code automatically creates table numbers (for the TITLE statement) starting at one and increasing by one. However, sometimes cross-regional comparisons are more easily done when standardized table numbers represent specific variables, and those numbers may not reflect the file's (or sorted file's) variable order. Thus, the code now looks for a optional "TABLENUM" variable, which is not required to execute the code, but when present will be used instead of the standardized counter.

Beyond the first five processes, other processes have been added, and there are now a total of eight distinct processes. Also, the application has been refactored to make these processes follow the Strategy design pattern, and they are therefore now eight distinct classes which are subclasses of the surveyYearAnalyzer class. Previously, each process was an SCL method. The specific class structure is incidental to this presentation, but is available for reference (Tabladillo, 2003a).

In summary, before any code is sent to the preview buffer, SCL checks the existence and availability of datasets, and the existence and correct variable types of variables. This section outlines how to check datasets before submitting code, but the next sections present a specific example.

AN EXAMPLE: PREFERRED TABLES

One of the key processes produces what are called “preferred tables”. The term “preferred” refers to the binary variables which collapse survey results, often in multiple categories, to two specific categories (a “yes”/”no” response). This process produces a set of tables (using proc tabulate) for each region-specific preferred variable. What this process does is less important than the fact that four specific datasets are required to make this preferred table process run. If any of these four datasets is missing or has bad values, then bad results will be produced. This example is presented in three sections: 1) Setup, 2) First Round: Dataset Availability, and 3) Second Round: Variable Availability.

SETUP

First, the datasets are accessed through the dataset attribute classes, so these four classes are linked to four (highlighted) specific objects. The definition code here does not use the DCL statement because they are in classes.

```
* Working Classes;
protected dataset_attr_prefSummary.class
      prefSummaryObj/(
         Category='Preferred Working Class',
         Description='Dataset Attributes for
Preferred Summary',
         AutoCreate='Yes',
         Editable='NO',
         ValidValues=''
         );

protected dataset_attr_preferredvars.class
```

```
      preferredvarsObj/(
         Category='Preferred Working Class',
         Description='Dataset Attributes for
Preferred Variables',
         AutoCreate='Yes',
         Editable='NO',
         ValidValues=''
         );

protected dataset_attr_layout.class
      layoutObj/(
         Category='Preferred Working Class',
         Description='Dataset Attributes for
Layout',
         AutoCreate='Yes',
         Editable='NO',
         ValidValues=''
         );

protected dataset_attr_data.class
      dataObj/(
         Category='Preferred Working Class',
         Description='Dataset Attributes for
Dataset',
         AutoCreate='Yes',
         Editable='NO',
         ValidValues=''
```

Note that the objects are all declared as "protected" which helps prevent the objects from being accessed from other parts of the application without going through this checking procedure. Accessing the datasets is completely encapsulated within the object, and other objects can only request information from this class (as opposed to directly accessing the process' control datasets). The "protected" status was chosen so that subclasses could also access datasets, and the term "private" would have prevented subclasses from accessing these objects.

Next, the objects are instantiated.

```
      * Initialize Objects;
      preferredvarsObj = _new_
dataset_attr_preferredvars.class();
      prefsummaryObj = _new_
dataset_attr_prefsummary.class();
      layoutObj = _new_
dataset_attr_layout.class();
      dataObj = _new_
dataset_attr_data.class(sex,level,age,special);
```

Three of the classes do not require any information to be instantiated. The dataObj requires the SCL variables sex, level, age and special. Instantiation means the object now exists and is ready for method calls. The number passed back to the object is the unique identifier of the SCL list containing the instantiated class information.

FIRST ROUND: DATASET AVAILABILITY

In the first round, the software will check whether or not process can open the datasets. It is possible that these checks will fail, and the two most common reasons are that the dataset does not exist at all, or that the dataset is locked (marked for exclusive use) by another process. The code does not attempt to determine the reason for failure, but simply will pass on an error message through setting the variable called "systemMessage". The error messages will result in a normal program termination without completing the process.

The "openDataset" method was created to capture the following steps each time the application attempts to open a dataset in the "input" mode (meaning read-only access), which is the most common open mode for this application:

```
OPENDATASET:public method
      inputDataset:INPUT:CHAR
      return=num
      /(
         Description='Opens dataset for input'
      );
      DCL
         num
            returnCode
            ;
      returnCode = 0;
      returnCode = exist(inputDataset,'DATA');
      if returnCode = 1 then do;
         datasetID = open(inputDataset,'i');
         if datasetID le 0 then
            systemMessage = '*** NETWORK ERROR:
DATASET CANNOT BE OPENED IN INPUT MODE -- ' ||
inputDataset;
      end;
      else do;
         systemMessage = '*** NETWORK ERROR:
SAS DATASET DOES NOT EXIST -- ' || inputDataset;
      end;
      return(systemError);
ENDMETHOD;
```

Instead of doing an outright OPEN, first the method checks for dataset existence, and then the dataset is attempted to be opened in "input" mode. If these conditions are not true, then an error is returned.

The code below shows how the process object calls the openDataset method. The openDataset method is coded in an abstract parent class, and therefore is inherited by all the different dataset attribute children (named DATASET_ATTR_*child*.class).

```
* SETUP THE CONTROL DATASETS;
returnCode =
preferredvarsObj.openDataset('SASDATA.'||_self_.
selectedRegionObj.PREF);
if returnCode then systemMessage = 'ERROR:
CANNOT ACCESS REGIONAL PREFERRED DATASET';
returnCode =
prefsummaryObj.openDataset('SASDATA.'||_self_.se
lectedRegionObj.PSMDATA);
if returnCode then systemMessage = 'ERROR:
CANNOT ACCESS REGIONAL PREFERRED SUMMARY
DATASET';
returnCode =
layoutObj.openDataset('SASDATA.'||_self_.selecte
dRegionObj.LAYOUT);
if returnCode then systemMessage = 'ERROR:
CANNOT ACCESS REGIONAL LAYOUT DATASET';
```

```
returnCode = dataObj.openDataset(sourceData);
if returnCode then systemMessage = 'ERROR:
CANNOT ACCESS REGIONAL DATASET WITH WEIGHTS';

* Determine Fact Sheet Production;
returnCode =
factSheetObj.openDataset('SASDATA.'||_self_.sele
ctedRegionObj.FACTSHEET);
if returnCode then do;
      returnCode =
factSheetObj.clearErrorList();
      produceFactSheet = 0;
      logMessage = 'REGIONAL FACT SHEET DATASET
NOT FOUND -- NO FACT SHEET WILL BE PRODUCED';
end;
else do;
      produceFactSheet = 1;
      logMessage = 'REGIONAL FACT SHEET DATASET
FOUND -- FACT SHEET WILL BE PRODUCED';

      * Response rates dataset;
      returnCode =
responseObj.openDataset(sourceRates);
      if returnCode then do;
         systemMessage = 'ERROR:  CANNOT ACCESS
REGIONAL RESPONSE RATES';
         produceFactSheet = 0;
      end;
      else do;
         returnCode =
responseObj.closeDataset();
         if returnCode then systemMessage =
'ERROR:  CANNOT CLOSE REGIONAL RESPONSE RATE
      end;
end;
```

Also included in the above code is a check for a regional factsheet. This factsheet dataset is a fifth dataset possibly available but not required to run the process. If present, the software will produce a "factsheet" which is a one-page summary of certain preferred table results. Note also that the response rate dataset is required if the factsheet is present. This code is included to illustrate that conditionally available datasets can be determined during this first round.

If there are any errors, then the variable "systemMessage" is set to a value, and those values are automatically stored in an SCL list. There is another variable called "systemError" which is simply the length of the SCL list holding the system messages. So, if there are no errors, then "systemError" will have a value of zero. Otherwise, this numeric variable will have the value of the number of errors. The "systemError" variable can then be used as a qualification to continue, as in "if not(systemError) then do…". If any errors appear at this stage, the program terminates, and the SCL list holding the errors prints to the log, allowing the analyst to fix whatever problems were flagged. Over time, the specific wording of errors has been improved based on empirical use, and in some cases, code added to help the analyst take action. As with any application, this SAS/AF application uses many terms which have specific and consistent meanings, and learning those terms are a part of the initial analyst training.

SECOND ROUND: VARIABLE AVAILABILITY

Given that there are no errors, the program then proceeds to check the integrity of the four required datasets (code will not be included here for the optional datasets). The following code is in four sections, each of which checks that the number of nondeleted observations (NLOBS) is greater than zero, and that certain named variables are present (they are present if their VARNUM is greater than zero) and of the correct type (either character or numeric – at present, there are no list types in SAS datasets, but if there were it would be like XML).

```
* Determine Possible Runtime Errors;
if preferredvarsObj.nlobs le 0 then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED DATASET HAS INSUFFICIENT
OBSERVATIONS';
if preferredvarsObj.varnum_name le 0 then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED DATASET HAS NO <NAME> VARIABLE';
if preferredvarsObj.varnum_yea1 le 0 then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED DATASET HAS NO <YEA1> VARIABLE';
if preferredvarsObj.varnum_nay2 le 0 then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED DATASET HAS NO <NAY2> VARIABLE';
if preferredvarsObj.varnum_question le 0 then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED DATASET HAS NO <QUESTION> VARIABLE';
if preferredvarsObj.vartype_name ne 'C' then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED DATASET <NAME> VARIABLE SHOULD BE
CHARACTER';
if preferredvarsObj.vartype_yea1 ne 'C' then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED DATASET <YEA1> VARIABLE SHOULD BE
CHARACTER';
if preferredvarsObj.vartype_nay2 ne 'C' then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED DATASET <NAY2> VARIABLE SHOULD BE
CHARACTER';
if preferredvarsObj.vartype_question ne 'C' then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED DATASET <QUESTION> VARIABLE SHOULD BE
CHARACTER';

if prefSummaryObj.nlobs le 0 then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED SUMMARY DATASET HAS INSUFFICIENT
OBERVATIONS';
if prefSummaryObj.varnum_name le 0 then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED SUMMARY DATASET HAS NO <NAME>
VARIABLE';
if prefSummaryObj.varnum_question le 0 then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED SUMMARY DATASET HAS NO <QUESTION>
VARIABLE';
if prefSummaryObj.varnum_yeacrit le 0 then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED SUMMARY DATASET HAS NO <YEACRIT>
VARIABLE';
if prefSummaryObj.varnum_naycrit le 0 then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED SUMMARY DATASET HAS NO <NAYCRIT>
VARIABLE';
```

```
if prefSummaryObj.varnum_freqstr le 0 then
      systemMessage = 'ERROR:  REGIONAL
PREFERRED SUMMARY DATASET HAS NO <FREQSTR>
VARIABLE';

if layoutObj.nlobs le 0 then
      systemMessage = 'ERROR:  REGIONAL LAYOUT
DATASET HAS INSUFFICIENT OBSERVATIONS';
if layoutObj.varnum_newname le 0 then
      systemMessage = 'ERROR:  REGIONAL LAYOUT
DATASET HAS NO <NEWNAME> VARIABLE';
if layoutObj.varnum_question le 0 then
      systemMessage = 'ERROR:  REGIONAL LAYOUT
DATASET HAS NO <QUESTION> VARIABLE';
if layoutObj.vartype_newname ne 'C' then
      systemMessage = 'ERROR:  REGIONAL LAYOUT
DATASET <NEWNAME> VARIABLE SHOULD BE CHARACTER';
if layoutObj.vartype_question ne 'C' then
      systemMessage = 'ERROR:  REGIONAL LAYOUT
DATASET <QUESTION> VARIABLE SHOULD BE
CHARACTER';

if dataObj.nlobs le 0 then
      systemMessage = 'ERROR:  REGIONAL CLEANED
DATASET HAS INSUFFICIENT OBSERVATIONS';
if dataObj.varnum_sex le 0 then
      systemMessage = "ERROR:  REGIONAL CLEANED
DATASET HAS NO <"||SEX||"> SEX VARIABLE";
if dataObj.varnum_level le 0 then
      systemMessage = "ERROR:  REGIONAL CLEANED
DATASET HAS NO <"||LEVEL||"> LEVEL VARIABLE";
if dataObj.varnum_age le 0 then
      systemMessage = "ERROR:  REGIONAL CLEANED
DATASET HAS NO <"||AGE||"> AGE VARIABLE";
if dataObj.vartype_sex ne 'N' then
      systemMessage = "ERROR:  REGIONAL CLEANED
DATASET <"||SEX||"> SEX VARIABLE SHOULD BE
NUMERIC";
if dataObj.vartype_level ne 'N' then
      systemMessage = "ERROR:  REGIONAL CLEANED
DATASET <"||LEVEL||"> LEVEL VARIABLE SHOULD BE
NUMERIC";
if dataObj.vartype_age ne 'N' then
      systemMessage = "ERROR:  REGIONAL CLEANED
DATASET <"||AGE||"> AGE VARIABLE SHOULD BE
NUMERIC";



* Obtain DataVarsList;
if not(systemError) then do;
if dataObj.datasetID then do;
      if dataObj.nvars > 0 then do;
         dataVarsList = makelist();
         do counter = 1 to dataObj.nvars;
            dataVarsList =
insertc(dataVarsList,varname(dataObj.datasetID,c
ounter),-1,putn(counter,'best3.'));
         end;
         dataVarsList = sortlist(dataVarsList);
         CALL PUTLIST(dataVarsList,'Data File
Variables',1);
      end;
      else systemMessage = 'ERROR:  REGIONAL
DATASET WITH WEIGHTS HAS INSUFFICIENT
VARIABLES';

      returnCode = dataObj.closeDataset();
      if returnCode then systemMessage =
'ERROR:CANNOT CLOSE REGIONAL DATASET WITH
WEIGHTS';
end;
else do;
      systemMessage = 'ERROR:  UNABLE TO ACCESS
OPEN DATAOBJECT';
end;
end;
```

Also included in the above code is a routine which accesses dataObj, and attempts to make an SCL list with “Data File Variables”. Populating this SCL list is considered a requirement before running the program. In general, any of the accessed datasets could have information pulled from them and stored in SCL lists, or character or numeric variables, and then optionally checked for internal integrity or against each other for referential integrity. The information could also be checked with other variables within the SCL environment (namely information entered on the frame). The point is that reading and validating information can and should be done before any SUBMIT calls are made. Particularly in this second round, you can see that referential dataset integrity checks are logically an immediate precursor to submitting blocks of SAS code.

If any errors are present after the second round of checking, then the system message SCL list will be populated, and the process will terminate normally with error messages printed to the log. The analyst can then fix whatever errors are present and run the process again.

BEYOND ROUND TWO

Other types of validation can be done during the SUBMIT process (by using the SUBMIT CONTINUE command, then proceeding with SCL code), and this technique is done in certain places in the application. However, those intermediate stopping points are for choosing between run paths, and none of the possible choices are expected to cause a run-time crash (that has been proven true throughout the application’s life). Anything which could crash the submitted program during run-time should be rolled into the first or second round of checking, before anything is submitted.

The developer needs to know the submitted code well to anticipate the types of things which will cause submitted code to crash. In this example, the four datasets chosen were called during the submitted code. The conservative approach, the one applied here, is to check for all the datasets called and choose all the variables accessed. There is, therefore, an intrinsic coupling between these dataset checks and the submitted code, and in the future, when the submitted code changes, then these dataset checking rounds also need to be examined for potential changes too.

CONCLUSION

Managing a large file structure is very doable with SAS/AF, but requires forethought and planning. While a standard SAS/AF project is complex enough, a highly customized application, such as the one presented, presents unique challenges which can be best handled with not only the standard SAS/AF interaction, but also the intentional extension of the “Analysis Matrix” design

pattern (Shalloway and Trott, 2002). Though single datasets can and should be checked for internal consistency upon bringing them into the SAS application environment (whether in SAS native format or another format), multiple dataset checks most logically belong immediately preceding specific processes. These multiple checks include required and optional datasets, required and optional fields, and required and optional data.

Further information is available on this application's class structure (Tabladillo, 2003a) and development (Tabladillo, 2003b).

REFERENCES

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

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

Tabladillo, M. (2003a), "Application Refactoring with Design Patterns", *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*, Cary, NC: SAS Institute, Inc.

Tabladillo, M. (2003b), "The One-Time Methodology: Encapsulating Application Data", *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*, Cary, NC: SAS Institute, Inc.

ACKNOWLEDGMENTS

Thanks to all the great public health professionals at the Office on Smoking and Health, Center for Chronic Disease.

TRADEMARK CITATION

CONTACT INFORMATION

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

Mark Tabladillo
Email: marktab@marktab.com
Web: http://www.marktab.com/