Paper BB-11-2014

Documentation Driven Programming: How Orienting Your Projects Around Their Data Leads to Quicker Development and Better Results

Marcus Maher, Ipsos Public Affairs, Chicago, IL
Joe Matise, NORC at the University of Chicago, Chicago, IL

ABSTRACT

In this paper we show a modular, data-driven method of programming using a data dictionary based approach. Constructing a data dictionary in a consistent, tabular fashion, one can write an application that reads in data, adds formats and labels, performs data cleaning, and produces reports without almost any additional coding. This leads to efficiency gains, accuracy gains, and more clear code. We explain the fundamentals of a data dictionary that can be used for this purpose, with examples for different specific purposes, and show one implementation of a simple system. We invite the reader to adapt the base concept to their specific needs, and suggest a few things to consider when adapting this to more specific purposes.

The process applies to Base SAS version 9, and examples will be presented assuming the presence of Access to PC Files, although the concept does not require it. Users will be expected to have an intermediate skill level with working with SAS files, and some understanding of macros and automated code generation (using PROC SQL Select Into or similar).

INTRODUCTION

SAS is a great product for automating data and reporting related tasks for many reasons, one being the ability to write programs that are driven by data sources. Driving your program from data means identifying portions of your program that are specific to some pieces of data – whether it be the primary dataset you are working with, or that project's metadata, or something you create specific to the purpose – and writing the program as a generic piece of code that uses that data source as input parameters, rather than including that information in the code itself. This leads to shorter codebases, more organized programs, more reusable code, and simpler updates. (If this sounds suspiciously like Object-Oriented Programming to you, that's because it comes from a similar mindset: structuring code around data. If you are familiar with OOP techniques, you will see many parallels in this method.)

This paper looks at a specific type of data driven programming, which we call documentation or metadata driven programming, covering its benefits and discussing implementation. It is geared towards projects that consist of performing analyses or reporting on sets of data, but many of the concepts can be utilized in other areas. We will discuss the concepts at a higher level during the paper, and present a complete program in the appendix implementing this approach on a sample data source (the 2010-2012 ANES Evaluations of Government and Society Study).

WHAT IS DOCUMENTATION DRIVEN PROGRAMMING?

A high level definition: It is an approach to programming that first creates metadata in a tabular format for the fields in your data that describes these fields and contains instructions for a series of actions that you wish to perform. The programmer will then implement a generic SAS program to interpret the metadata and generate SAS code from it.

WHY SHOULD YOU USE DOCUMENTATION DRIVEN PROGRAMMING?

Documentation driven programming is not only useful as a method of keeping your manager off your back to document your process. Organizing your documentation in a way that allows you to drive your programming with it enables you to work with others more effectively (in particular with non-programmers), and leads to greater transparency, writing fewer lines of code, increased accuracy with less QA required, and greater reusability of code.

COLLABORATION AND EFFICIENCY

Documentation driven programming helps collaboration in two key ways: it clarifies communication, and it enables more efficient utilization of resources. Not every programmer knows SAS, and not everyone can read programming syntax. By driving your programming with documentation that does not require a programmer to fill out, you can move a substantial portion of the work to project staff and other non-programmers. Ideally it can be the method by which your requirements are delivered to you, allowing you to turn requirements into production code with much less effort.

TRANSPARENCY AND READABILITY

If your documentation set-up is clear enough that it can be explained to someone with a non-SAS or non-technical background, it also makes your work more transparent. If other researchers, supervisors, or collaborators want to know how you did something, for example dichotomizing a variable, you can send them your documentation, which includes complete information for that process. It is also easier for you to answer questions after the analysis is complete (when you have moved on and forgotten the specifics) because spreadsheets are more organized and easier to read than programming code.

Your programs will be much clearer and easier to read as a result of moving the data definitions to a spreadsheet. This takes the clutter out of the program, which then makes them much more focused on the logic of the process you are developing. Removing much of the code needed to read in data and perform normal cleaning, validation, and preparation from your main program makes it easier to focus on the interesting elements of your program.

For example, compare these two short programs.

```
data my_data;
  infile "myfile.txt" dlm=',';
  informat
    x y $10.
    a b $15.
    c  datetime17.
    d  date9.
    e f $20.
    g $35.
  ;
  format
    c datetime17.
    d date9.
  ;
  input
    (x y a b ) ($)
    c
    d
    (e f g) ($)
  ;
run;

data my_data_cleaned;
  set my_data;
  if x in ('BADDATA','MOREBAD')
    then delete;
  else if x in ('BADVAL','BADVAL2')
    then x=' ';
  if datepart(c) le '01JAN2010'd
    then  delete;
  else if datepart(c) ge '01JAN2014'd
    then delete;
run;
```

```
data my_data;
  infile "myfile.txt" dlm=',';
  &informatlist.;
  &formatlist.;
  &inputlist.;
run;

data my_data_cleaned;
  set my_data;
  &cleanrulelist.;
run;
```

Figure 1. Two Examples of Input Code.

Okay, we cheated some with the second program, but that is the point: all of that code was removed from the body of the core program, and can be located either in other program file(s) or in an earlier portion of the main program, easily skipped over while browsing the code as it has clear purpose and in a production environment would be oft-reused code. The core portion of the data step is stripped clean, allowing users to read exactly what is happening (of course, using well-named macro variables).

Now if you were to include a few lines of code in the data for a specific purpose, those lines would be obvious to any reader and easy to spot, not hidden in the middle of dozens of lines of input statements, formats, informats, and labels. Data driven programming reads like a story, and the details are tucked away in footnotes.

ACCURACY AND REUSABILITY

Utilizing a spreadsheet driven approach makes your program more generic, meaning it is more likely that you can reuse sections of code or even entire programs with nearly no changes other than the spreadsheet. Moving definitions to another source or data set removes much of what is project or task specific from the program, so if you have tasks that are superficially identical you can drop in a new spreadsheet, update the output file and input file names, and run. Your code is less likely to have errors, as it is tried-and-tested code from other projects. This also is beneficial for regularly scheduled programs, as you can set them up to look for the correct (or newest) spreadsheet and programmatically define the input/output files, and not need any changes to the SAS code under normal circumstances for your regularly scheduled runs - even if the data changes some.

All of this accumulates to make it much easier to develop accurate programs. If you have a series of similar actions to perform, you can easily structure your documentation to allow you to re-use code for those actions. Updating the process from run to run is easier too, as the organizational structure provided by a spreadsheet or table makes it easier to see where changes need to be made. Updating a series of hardcoded macro calls is not hard, but it's still possible to miss one. Not having to page through many lines of hardcoded formats, labels, macro calls, and input statements saves times and reduces errors.

MODULAR PROGRAMMING

One key concept to documentation driven programming is that of modular programming. Modular programming means writing programs that accomplish a particular small, self-contained task as a stand-alone program. That module is then one recipe in your cookbook; beginning a new project is as simple as grabbing the recipes that you need from the book and putting them into practice. This is part and parcel of many of the above objectives; it increases accuracy by virtue of re-using tested code, it improves readability by separating the tasks into self-contained modules, and it improves efficiency by reducing the start-up time for new projects. Modular programming also simplifies the development of new modules themselves, as they can be easily tested alongside other modules known to work for comparative purposes.

In the end, we utilize programming from metadata or documentation because it lowers risk and increases productivity. Non-programmers can more easily understand what you did, you can transition from specs to completed program in less time, and your program will be much more portable and focused on the process flow.

In the rest of the paper we will demonstrate how documentation driven programming would be applied to reporting, first at a general level and then working through a specific example. The paper authors come from a survey research background, so these implementations will reflect that industry experience; however, the general process is certainly applicable in a variety of contexts given appropriate customization.

THE CONTROL TABLE (DATA DICTIONARY)

The automated reporting solution is driven by the control table, which is both a data dictionary and the program driver. It contains the information necessary to input or subset your data, format and label the variables, clean or validate your data, as well as to generate reports from your data. It is often an Excel spreadsheet, as it will be in the examples in this paper, and references to "control spreadsheet" or "control sheet" can be assumed to refer to the same thing.

A1 fx Variable

	A	B	C	D	E	F	G	H	I	J
1	Variable	Label	Start	Length	Format	Values	Base	Range	Table	Statistic
2	version	version. Dataset version	1	30	$					
3	c2_caseid	c2_caseid. Case ID	32	4						
14	c2_xgrp	c2_xgrp. Group assignment	169	1				1:6		
15	c2_b1	Voted in 2008 election	251	2		1: 1. Definitely voted 2: 2. Definitely did not vote 3: 3. Not completely sure		1:3	1-1	freq
16	c2_b2	Do you think you probably voted or probably did not vote?	254	2		1: 1. Probably voted 2: 2. Probably did not vote	c2_b1=3	1:2	1-2	freq
17	c2_b3	In the 2008 presidential election, who did you vote for?	257	2		1: 1. Barack Obama 2: 2. John McCain 3: 3. someone else	c2_b1=1 or c2_b2=1	1:3	1-3	freq
18	c2_c1	How much do government officials care what people like you thinl	260	2		1: 1. A great deal 2: 2. A lot 3: 3. A moderate amount 4: 4. A little 5: 5. Not at all		1:5	3-1	top 2 bo

Figure 2. Example Control Table as Excel file

The basic structure of the table is one row per dataset variable. The first column is the SAS variable name. Other than that column, you can pick and choose what columns you need depending on what your application needs; we will describe the most common possible columns here, but there are many different potential combinations depending on your exact needs. One of the great advantages of a solution like this is the flexibility to add or subtract functionality based on need.

The control table does not have to be a spreadsheet; depending on your needs and comfort level with different applications, you might store the information in Access table(s), SQL Server tables, SAS datasets, or text files. Depending on your needs, you might prefer to contain all information in a single sheet or table, or you might prefer to split up different groups of information that are used in different places, or perhaps have different responsible parties.

We group the column types into four basic concepts. There are the input columns, the formatting and metadata columns, the data cleaning columns, and the reporting columns.

INPUT COLUMNS

While your data may come to you in a SAS dataset with exactly the columns and rows you are interested in, the odds are it usually will not. In these cases you need to input your data.

One common data source is the flat fixed-column text file. To read this in, you need to know the position, length, and informat of the data elements. You put each of those values in a separate column, as below.

	A	C	D	E
1	Variable	Start	Length	Format
2	version	1	30	$
3	c2_caseid	32	4	
4	c2_sampwt	37	12	
5	c2_weight	50	12	
6	der08c2	63	2	
7	der09c2	66	2	
8	derteac2	69	2	
9	derapp1c2	72	2	

Figure 3. Control Table Input Cells

These will later be used to construct input statements of the form

```
@[position] variable [informat][length]
```

This will be a consistent theme throughout this solution: deconstructing SAS statements and loading their contents into columns of the control table.

If your data is in a delimited text file, you may want to include a column that indicates character variables with a $ or blank value and a variable order column. You could also use an informat column if you need to (rather than the character indicator column).

Another type of input may be SAS dataset(s). You might download public use SAS datasets that have hundreds of variables in them, of which you need a handful. If this is the case, you can include a column that indicates variables you wish to keep (a simple 1/0 flag). You might also need to combine multiple datasets together; if this is the case, you might have a “source” column that defines which dataset a variable comes from, with “all” indicating all datasets (for your merge variable).

Further, in some cases you may need some variables for processing your data (for example, flags that are useful in data cleaning or validation) but are not desired in the output dataset (particularly if you are creating a dataset to deliver to an end user or client). The column flag can be expanded to have multiple values; in one project, we had “Input” variables and “Output” variables, where “Input” meant we brought them in but did not save them in the final SAS dataset. This can be combined with any of the other input types.

FORMATTING AND METADATA COLUMNS

Using formats is often two steps: creating custom formats, and applying formats to variables. Depending on the complexity of your data and your needs, you may accomplish the creation in a few different ways; we show two here.

The simplest way to define and apply your formats is to include them in a single column with equalities separated by a delimiter. In the example below, we have the formats defined in an excel column, separated by newlines (alt+enter to key this in excel). We will use this both to create a format for the variable, and to apply that format. If you are using a built-in SAS format, you can simply include it in this column without values.

	A	F
1	Variable	Values
6	der08c2	0-1: Democrat 2-4: Independent 5-6: Republican
7	der09c2	1-2: Liberal 3-5: Moderate 6-7: Conservative
8	derteac2	1-3: Support Tea Party 4: Neither support nor oppose 5-7: Oppose Tea Party
9	derapp1c2	1-3: Approve of Obama 4: Neither approve nor disapprove 5-7: Disapprove of Obama
10	derfreec2	1-3: More free today 4: The same amount of freedom 5-7: Less free today

Figure 4. Example Values column

This is appropriate when your formats are fairly simple (either one value to one value, or one range to one value) and are either very different from each other or you do not have too many different variables. This approach does not lend itself as easily towards reusing custom formats, so in cases where reuse is preferred it is often easier to separate the application from the creation of the formats. This has the advantage of showing the values for the variable directly in the data dictionary’s main sheet, making it easier to see what values a variable can take at a glance.

Another option is to put the formats onto a separate tab. Separating the format value list from the main data dictionary is useful because you can define a single format and then reuse it. We show an example here where we create a second sheet in the workbook labeled “Formats”, and define our formats in this sheet. We then use the format names from this sheet in the “format” column on the first sheet, which will then apply the formats we create on the other sheet. This also means that the format name can be more descriptive (in the single column version, the name will typically be derived from the variable name rather than the options).

	A	B	C	D	E
1	Fmtname	Start	End	Label	Multilabel
2	yesnof	1	1	Yes	
3	yesnof	2	2	No	
4	Agesf	18	34	18-34 Total	Yes
5	Agesf	18	24	18-24	
6	Agesf	25	34	25-34	
7	Agesf	35	54	35-54 Total	
8	Agesf	35	44	35-44	
9	Agesf	45	54	45-54	
10	Agesf	55	64	55-64	
11	Agesf	65	high	65+ Total	
12	Agesf	65	74	65-74	
13	Agesf	75	high	75+	

Figure 5. Example of separate Values tab

Variable labels should be stored in another column, which often will serve the data dictionary purpose of describing the variable for the user.

Other metadata fields may be useful, such as variable lengths (particularly for numeric variables), an output order column for defining the order the variables will be in the output dataset, or variable classes (which could be useful in reporting, to pull groups of questions that are related, or to pull questions with a similar scale).

DATA CLEANING AND VALIDATION COLUMNS

Data cleaning and validation can encompass a number of things, from very simple operations to very complex. We will cover this to a medium depth, and point to where this could be expanded.

One of the major tasks you must undertake in taking data from another source and using it in your analyses or reports is to ensure the data conforms to the rules you expect. This includes range checks for individual variables, skip pattern checks for variables that might be contingent on the value in another variable, and more in depth logical checks. Some of this will inevitably be project-specific, and may encompass the majority of the code you write for a project, but some of it can be generalized.

In general, we will be creating columns indicating particular checks to be performed on particular variables, and then in the methods section we will explain how to perform those checks. Typically, dataset(s) will be created containing observations and/or variables that fail these checks, with indicators of how they fail those checks.

	A	G	H
1	Variable	Base	Range
14	c2_xgrp		1:6
15	c2_b1		1:3
16	c2_b2	c2_b1=3	1:2
17	c2_b3	c2_b1=1 or c2_b2	1:3
18	c2_c1		1:5
19	c2_c2		1:5
20	c2_jl1	c2_xgrp in(2,5,6)	1:5
21	c2_jl2	c2_xgrp in(2,5,6)	1:5
22	c2_jl3	c2_xgrp in(2,5,6)	1:5
23	c2_jl4	c2_xgrp in(2,5,6)	1:5
24	c2_jl5	c2_xgrp in(2,5,6)	1:5
25	c2_jl6	c2_xgrp in(2,5,6)	1:5
26	c2_zeg1		1:2

Figure 6. Example Columns for implementing edit checks

Range checks often sync up with custom formats for categorical variables. As such, we might create a column that indicates whether a variable should be flagged if it does not have a value defined in its format.

You also could include in that column the valid range for a variable (in the form [start..end start..end], so if 1..5 7,8,9 are valid (1 through 5 for normal answers, and then 7,8,9 for refused/don't know/NA), then [1..5 7..9] or [1..5 7,8,9]). The exact syntax is unimportant, so long as you have a consistent set of rules and communicate them to your users.

Skip pattern shows when a variable is either required to have a value for every record, is required to have a value if another variable has a particular value, or is required NOT to have a value if another variable has a particular value. You might create a column in your data dictionary that indicates this through a few different means.

First, you might have a value "All" that indicates a variable is all-based (should have a value for every record). Then you might include values like "Q52=1" that indicate a question should have a value if Q52=1 but not otherwise (so Q52 might be "Did you lose your luggage", and Q53 might be "Did you eventually find your luggage?", only asked if Q52 is yes). You might indicate "not(Q53=1)" if it should have a value only of Q53 does not equal 1 (Q53 might be "Did you eventually find your luggage?" and Q53a might be "Did you file an insurance claim?", depending on Q53 not being 'yes').

You could conceivably allow any legal SAS syntax in this column, although it is probably more readable if you typically limit this usage to fairly simple syntax (ie, something that would fit in a 20 character wide column or so). Commonly using complex syntax makes the document harder to read, which makes it harder for users to get value out of it; and it may be easier to write that in a SAS program rather than in an excel column or similar. However, if it is unlikely that non-programmers will use your documentation, including all qualification logic will help keep your program cleaner and more focused on the logic of the overall process. Both approaches have benefits and shortfalls and we advise you to pick the path that makes most sense in your environment.

For some projects, you likely could come up with more in-depth logical checks, such as record counts for relational datasets (for example, a respondent can have up to 5 rows, one for each store the respondent shopped at). These would be custom implementations for each project, likely, but you could easily develop some generalized versions that could easily be modified for each project.

REPORTING COLUMNS

The contents of the reporting columns will vary depending on what kind of reporting you typically do, and how standardized it is; but the general format will be similar. You may have some column(s) that define what kind of statistic is associated with a particular column (Yes/No, 5 on a 1-5 scale, 8-10 on a 0-10 scale, >50, mean, etc.); you may have some column(s) that define composites of variables; and you will have some column(s) that define which variables are used in what report (probably one column per report). Often this needs to be a separate spreadsheet that is report-oriented rather than variable-oriented, but for simple applications (such as where a report consists of a frequency table for each variable), it may be contained on the variable sheet. We present one simple example here, but encourage you to think outside the box when using this in your own environment.

	A	H	I	J
1	Variable	Range	Table	Statistic
14	c2_xgrp	1:6		
15	c2_b1	1:3	1-1	freq
16	c2_b2	1:2	1-2	freq
17	c2_b3	1:3	1-3	freq
18	c2_c1	1:5	3-1	top 2 box
19	c2_c2	1:5	3-2	top 2 box
20	c2_jl1	1:5	2-1	mean
21	c2_jl2	1:5	2-2	mean
22	c2_jl3	1:5	2-3	mean
23	c2_jl4	1:5	2-4	mean
24	c2_jl5	1:5	2-5	mean
25	c2_jl6	1:5	2-6	mean
26	c2_zeg1	1:2	1-4	freq

Figure 7. Example of Reporting columns

Here we have three reporting columns. Each have the name of a report as the column header, and in the body each column that is defined in the report has some text here identifying what statistic you would include in the report based on this column. In our industry much of our reporting is on scale data, where we typically take the top X values out of Y. For example, we might have a likert scale on 1-5 and then we take % of 5 out of 1-5, or a scale of 1-10 and we take % (8,9,10) out of 1-10, to show the percentage of "highly favorable" responses. We also often would take the percentage of "highly unfavorable" responses, and we may frequently take the mean or median of a column.

For a more complicated report featuring multiple statistics per variable, there are several options. For example, suppose that we desire to report the proportion of highly favorable responses, the proportion of highly unfavorable responses, the mean, and the median. We could add multiple statistics in the single Statistic column, separated by a delimiter (say, a semicolon).

Another option would be to add a total of four columns, one for each statistic required. The mean or median column would be marked with an 'X' or a '1' if you desire to see them reported. In the columns for the favorable and unfavorable responses you would input the range of responses that fit that category.

	A	B	C	D	E	F	G
1	Variable	Table	Favorable	Unfavorable	Median	Mean	Composite
2	Q4	2-2	5	1:2			
3	Q5	3-1	5		x	x	DoctorSatisfaction
4	Q6	3-2	5				DoctorSatisfaction
5	Q7	3-3	5				DoctorSatisfaction
6	Q8	2-1			x	x	

Figure 8. Example of Reporting columns with multiple statistics and a composite

Here we add another level of complexity, a column defining which composite(s) a variable is a part of, and then add a row for the composite itself which allows us to more easily request it on a particular report. Q5, Q6, and Q7 are combined into a composite "DoctorSatisfaction", which is then used in a report using the proportion of highly favorable responses (in this case '5' on a 5 point scale).

IMPLEMENTATION MACROS

Now that we have a completed data dictionary, we need to have some SAS code that performs the necessary operations on the data. This comes in the form of macros that are parameterized based on the information in the data dictionary.   We will not go into every type of macro here - and in particular, this is where your implementation will most likely differ from the base concept, given your needs will be unique - but we will cover several of the most common types of code needed.

In this section, we assume you are familiar with automated code generation, such as using PROC SQL and SELECT INTO to convert data into macro variables. There are several other methods that might be preferable depending on your needs and preferences; use whichever fits best and you are most comfortable with.

DATA INPUT

The most basic utilization of the data dictionary is to input your data from an external source such as a text file. In this example we will show an implementation using a fixed width column file (as opposed to a delimited file, such as a CSV). This uses the columns containing the variable name, the position the variable starts in, the length of the column, and the informat for the column. Here, again, is the screenshot of the control file from above:

Variable	Start	Length	Format
version	1	30	$
c2_caseid	32	4	
c2_sampwt	37	12	
c2_weight	50	12	
der08c2	63	2	
der09c2	66	2	
derteac2	69	2	
derapp1c2	72	2	

Figure 9. Input section of Control Dataset

To utilize this information, we need a basic macro that takes as parameters variable, start, length, and format:

```
%macro read_in(var,start,fmt,length);
  @&start. &var. &fmt.&length.
%mend;
```

Then we use PROC SQL to generate a list of calls to this macro which we will later use in our input datastep.

```
proc sql noprint;
  select cats('%read_in(',variable,',',start,',',format,',',length,')')
    into :read_in_flat separated by ' '
    from prepped_control
    where start ne ' ';
quit;
```

This generates calls that look like the following. During development, you can remove the NOPRINT from the PROC SQL statement, which will cause the macro calls to appear in your results window for ease in debugging.

```
%read_in(version,1,$,30.)
%read_in(c2_caseid,32,,4.)
%read_in(c2_sampwt,37,,12.)
```

LABELLING AND FORMATTING

Generating the code to label your variables is a similarly straightforward process, with two slight differences to accommodate the more complex strings you might see in a variable label. The macro is simple:

```
%macro labelcode(var,label);
  &var.="&label."
%mend;
```

The PROC SQL code, however, is slightly different:

```
proc sql noprint;
  select cats('%labelcode(',variable,',%nrstr(',label,'))') length=1024
    into :labelcode separated by ' '
    from prepped_control
    where label ne ' ';
quit;
```

Note the length option on the select statement, which overrides the default length of CATS in a PROC SQL statement (200), which might be less than our variable labels' lengths. Also differing is the addition of the %NRSTR wrapper around the label, which protects us from special characters causing unintentional macro variable resolution. (If you intentionally include macro variables in your labels, leave this off.)

The last step in our initial data input is to generate some basic formats. At this stage we assume that you have loaded your formats from your control file into your format catalog using PROC FORMAT; full code for this step is included in the appendix.

```
%macro apply_formats(var,fmtname);
  &var. &fmtname.
%mend;
```

Now we are ready for a data step to input, label, and format our data. Notice that we call the three macro variables that we generated above in our data step. Recall the data step in Figure 1 here; this is nearly identical.

```
data input;
  infile datafile lrecl=32767 pad;
  format
    &formatcode.
  ;
  input
    &read_in.
  ;
  label
    &labelcode.
  ;
run;
```

At this stage, we have a complete dataset with labels, formats, and of course data. It is ready for the next step, which in our case is data validation.

DATA VALIDATION

The two data validation methods we present here are fairly basic, validating ranges and bases, and hopefully denote the minimum one would do to validate a dataset for delivery or analysis. Further validation methods could be included, depending on the needs of the project and the researchers; this might include logical validation, referencing an external lookup table, or foreign key validation, among others.

First we check that all of the data is in our defined range using the range_check macro. Here we would only pull into the macro variable rows that have valid ranges listed.

```
%macro range_check(var,range);
  if not(missing(&var.)) and not(&var. in(&range.)) then do;
    type='RANGE';
    question="&var.";
    output;
  end;
%mend;
```

Any instance where a variable has data and is not in our pre-defined range will be output to our checking dataset along with a flagging variable, which we are calling ‘type’ stating which check it failed.

Since this is survey data, the allowable responses on the questionnaire provide our constraints. In other instances one could flag data based on prior knowledge of what is “reasonable” and follow-up on outliers. One could also put a keyword into this field to trigger an execution of PROC UNIVARIATE that flags your most extreme cases or gives you a histogram to show you the distribution of the data.

The second check we will perform verifies that you have data if and only if you expect to have data. Here there are two distinct possibilities. You could expect the field to have data always, then all one needs to do is leave the ‘base’ column blank in the control file and the program will catch blank records. The second is that you expect to have data in particular situations. Input your constraints in the ‘base’ column and the program will check in both directions. Do you have data when you expect it, and is it missing when you don’t expect it?

```
%macro base_check_all(var);
  if missing(&var.) then do;
    type='BASE';
    question="&var.";
    output;
  end;
%mend;

%macro base_check_restricted(var,base);
  if (&base.) and missing(&var.) then do;
    type='BASE';
    question="&var.";
    output;
  end;
  if not(&base.) and not(missing(&var.)) then do;
    type='BASE';
    question="&var.";
    output;
  end;
%mend;
```

Remember that &BASE. here is a logical SAS statement (that evaluates to TRUE or FALSE). When we create the two lists of macro calls, we use the presence or absence of data in the Base column to determine which list a variable falls into. Here we assume that all questions should be checked; if your project expects some variables to have missing values, and considers that acceptable, you may have a different call.

```
proc sql noprint;
  select cat('%base_check_all(',variable,')')
    into :base_check_all separated by ' '
    from prepped_control
    where missing(base) ;
  select cats('%base_check_restricted(',variable,',%nrstr(',base,'))')
    into :base_check_restricted separated by ' '
    from prepped_control
    where not(missing(base));
quit;
```

You can call these three macro lists in a later data step. We choose to create a vertical dataset with all of the errors and an identifier showing which error check it failed (base or range) – thus the output statements in the macros.

```
data checking;
  format question type $32.;
  set input;
  &base_check_all.;
  &base_check_restricted.;
  &range_check.;
  keep c2_caseid type question;
run;
```

This dataset could then be used to create an error log for your users to check, either using a PROC FREQ to identify problematic variables (particularly during development) or by viewing an export of this dataset directly to a file if there are few errors (during production in particular). In the appendix we show a simple error log as an example.

AUTOMATIC CORRECTION

In the case of our data source, we have a lot of data to clean. People who did not respond to questions were marked with a negative number of varying values depending on the reason why the person did not answer the question. We can take all of our entries into the control file for validation and just as easily turn them into cleaning statements, if needed. We only recommend doing this in conjunction with verifying the errors in the validation step, as this could cause serious issues, either by masking a more serious error (such as a failure point/break off in the questionnaire) or could be a result of improperly defined validation specifications.

If you analyze your errors and decide you want to automatically clean them, as we do, you could use a macro like this. You could also supply in another column more complex cleaning instructions, such as different special missing values for different reasons for invalid data. In our case we are comfortable with simply setting these invalid data points to missing.

```
%macro range_clean(var,range);
  if not(missing(&var.)) and not(&var. in(&range.))
    then call missing(&var.);
%mend;

%macro base_clean(var,base);
if not(&base.) and not(missing(&var.)) then  call missing(&var.);
%mend;
```

We then utilize these macros (pulled into macro variable lists) in a data step to generate our final cleaned dataset. In a more complex project we might have additional cleaning instructions included in this data step that were not generated automatically, but instead were either listed in a SAS program to %INCLUDE, or stored in an editing database to be applied in a similar fashion here.

```
data cleaned_data;
  set input;
  &range_clean.;
  &base_clean.;
run;
```

REPORTING

Reporting needs vary significantly from project to project, so our reporting solution is geared around separating the calculation and report production into separate modules, allowing users to build a cookbook of reporting solutions over time that can be mixed and matched to produce reports as needed. Here we will show a fairly simple example, with two calculation macros and one report production macro.

The basic shell of a calculation macro consists of a call to a summarization procedure, whether that is PROC TABULATE, PROC SURVEYFREQ, PROC SUMMARY, or similar procedures. Following that is a data step that converts the output from the summarization procedure into a generic format that can be consumed by the report production macro(s) without having to know how the data was summarized.

In our approach, the calculation macro is responsible for providing five variables: the table the row of results is for (from the control spreadsheet); the statistic type, useful when a variable is summarized in multiple ways; the stub, which is identifies which row in the report the result belongs on; the column variable, which stores which classification variable created that row and which determines which column in the report the data is intended for; and the score, which is the result itself.   More complex reports might include additional columns, such as a statistical test result or a standard error.

We also take an approach here in this example that is useful for allowing extensive customization and extremely simple code, but comes at the cost of additional runtime when used with larger datasets. That approach is to run the summarization procedure (in this case PROC TABULATE) once for each report row. This is generally insignificant when running reports from small or medium sized datasets, but when run on very large datasets or with a very large number of report rows it may be impractical to take this approach. In that case, a similar module might be created that performs the tabulations in larger batches to avoid multiple runs against the dataset.

CALCULATIONS

The first tabulation macro (below) is used to report proportions. This might be the proportion of responses in a particular range, or to report the distribution of responses across all values. It takes three parameters: VAR, FMT, and TABLE. VAR indicates the variable that is being summarized, while TABLE includes which table the response will be included in and the order in that table (for example, Table 3.01 would be Table 3, row order 1).

FMT is how the macro is able to flexibly calculate many different proportion types, even of a single variable. FMT provides the name of a format that groups the values needed for reporting together, and can even be used to remove unwanted values (such as if you want to see proportion of positive responses, and do not want to report negative or neutral responses) by using '.' to label those values.

In our case, we use two format types; one is a simple distribution, using the default format for the variable, and one is a "top box" format, where we show the proportion of positive responses and the proportion of other responses. The top box format is an example of a custom module here; it is dropped in to create a custom format that meets our needs, but doesn't require modifying the remainder of the program beyond adding the additional tabulation step, which could be included as part of the module or built as a separate module if it is more generally useful.

```
%macro tab_prop(var,fmt,table);
  proc tabulate data=cleaned_data out=_table_&table.(rename=&var.=stub);
    class &var. &class./mlf missing;
    var weight_inv;
    table (&var.*colpctn weight_inv*sum),(all &class.);
    freq tabulateweight;
    format &var. &fmt.;
    label weight_inv='Total Respondents';
    where not(missing(&var.));
  run;

  data _table_&table.;
    format stub $256.;
    set _table_&table.;
    array classes &class.;
    do _i = 1 to dim(classes);
      if substr(_type_,_i+1,1)='1' then columnvar=classes[_i];
    end;
    if (substr(_type_,1,1)='1' and length(compress(_type_,'0'))=1)
      or compress(_type_,'0')=' '
      then columnvar='All';
    Score=round(coalesce(of pct:,weight_inv_Sum),1.0);
    Stattype=ifc(not(missing(weight_inv_Sum)),'Total Respondents','Percent');
    Table="&table.";
    if stattype='Total Respondents' then stub=vlabel(stub);
    if strip(columnvar)='.' then delete;
    keep table stattype stub columnvar score;
  run;
%mend;
```

The second tabulation macro calculates mean values, but otherwise operates similarly to the proportion macro. The only significant external difference is that instead of the FMT parameter, a VARLABEL parameter is included to allow the column to be given a custom label. This is specific to the means calculation, as a proportions calculation has a logical column label derived from the formatted value. The %tab_mean macro is available in the appendix.

These macros are then called based on the data in the control sheet. First, the CLASS variables are brought into a macro variable for use in the tabulations. Then, we generate our calls to the tabulation macros themselves. Here we have two basic calls (one to %tab_prop and one to %tab_mean), and then a third custom call that comes from our top box module.

```
proc sql noprint;
  select variable into :class separated by ' '
    from prepped_control
    where not(missing(class))
    order by class;
```

```
  select cats('%tab_mean(',variable,',',table_sas,', %nrstr(',label,'))')
    into :tab_mean separated by ' '
    from table_control
    where statistic='mean';

  select cats('%tab_prop(',variable,',',var_fmtname,',',table_sas,')')
    into :tab_prop1 separated by ' '
    from table_control
    where statistic='freq' ;

  select cats('%tab_prop(',variable,',',reporting_fmtname,',',table_sas,')')
    into :tab_prop2 separated by ' '
    from table_control
    where substr(statistic,1,4)='top ';
quit;
```

These calls are then executed, generating our table row datasets. Finally, these table row datasets are accumulated into a single reporting dataset, called ALL_TABLES. This reporting dataset may be saved as an external table to assist in quality assurance activities, as it contains all values that will be reported out.

REPORT GENERATION

Once the reporting dataset is generated, the appropriate reporting macro is called to generate the report. In the implementation we show here, we first transpose the ALL_TABLES dataset so each columnvar is turned into a column (rather than a row, as it is initially).

```
proc transpose data=all_tables out=for_report;
  id columnvar;
  idlabel columnvar;
  var score;
  by table descending stattype stub;
run;
```

Then, we generate three macro variable lists for PROC REPORT: the column variables (for the COLUMN statement), the DEFINE statements, and the report macro calls (defined in the reporting dataset itself).

```
proc sql noprint;
  select  label_sas
    into  :columnord separated by ' '
    from format_cntlin
    where not(missing(class));

  select cats('%definecol(', label_sas,')')
    into :definecol separated by ' '
    from format_cntlin
    where not(missing(class));

  select distinct cats('%report(',scan(table,1,'-'),')')
    into :report separated by ' '
    from table_control;
quit;
```

Finally, the reporting macro, which is very simple. Most of the code has been pulled from the control file. Any style options would likely be located in the %DEFINE macro (either as parameters or defaults for all rows). The report calls are then placed inside an ODS statement based on which destination you want to output the report to.

```
%macro report(page);

  proc report data=for_report nowd spanrows;
    where substr(table,1,1)="&page.";
    columns
      (stub stattype)
      (All &columnord.)
    ;
    define stub/' ' display style={width=1.5in};
    define stattype/' ' display;
    define all/display;
    &definecol.;
  run;

%mend;
```

ADDITIONAL REPORTING OPTIONS

The reporting macro included here is very bare-bones in order to show the key concepts effectively. There is a lot of room for improvement and/or additional options that is left to the reader. For example, details like titles and footnotes could be added to tables, statistical tests or standard errors could be added, multiple files could be produced. Many of these could be accomplished by adding a new column to the control table, and then adding a small bit of code to add the functionality.

Adding titles, as an example, would require adding a column to the control table, in that column entering a report title to each of the table rows (just the first of each overall table would suffice), and then modifying the report macro and the call to that macro to add a title parameter and to use it in the title statement.

CONCLUSION

Documentation driven programming can be an effective way to improve efficiency and the accuracy of your results, while maintaining flexibility for custom work. Developing a large cookbook of modules allows you to respond quickly to requests while having to code very little. Further, using the data dictionary format as a requirement definition allows the programmer to further reduce project-specific work while allowing the more project-specific work to be done by those with the business or project knowledge.

REFERENCES

Data for the example in this paper was obtained courtesy of the ANES data repository at http://electionstudies.org/ . Dataset documentation is located at http://www.electionstudies.org/studypages/2010_2012EGSS/2010_2012EGSS.htm (accessed 9/7/2014), and data may be obtained with (free) registration from their study database at http://www.electionstudies.org/studypages/data/2010_2012EGSS/anes2010_2012egss2.zip

DeBell, Matthew, Catherine Wilson, Gary Segura, Simon Jackman, and Vincent Hutchings. 2011. Methodology Report and User's Guide for the ANES 2010-2012 Evaluations of Government and Society Study. Palo Alto, CA, and Ann Arbor, MI: Stanford University and the University of Michigan.

RECOMMENDED FURTHER READING

For a good reference on list processing (the concept largely used in this paper), see Ron Fehd and Art Carpenter's 2007 SGF paper, "List Processing Basics: Creating and Using Lists of Macro Variables", found at http://www2.sas.com/proceedings/forum2007/113-2007.pdf . We prefer the variation "Write Calls to a Macro Variable".

APPENDIX

Below is the full code necessary to implement this procedure, using the ANES data previously referenced. A control spreadsheet will be distributed with this paper, or may be obtained from the authors. The examples above were largely taken from the code below, but when implementing this we recommend using the code presented here, as there may be minor differences.

```
%let datadir=d:\temp\DataDrivenProgramming;
              *specify the directory your data and control are stored in;
%let outdir=d:\temp\DataDrivenProgramming;
             *specify the directory your reports should be generated in;
%let reportname=DDPExample.pdf;
filename datafile "&datadir.\anes2010_2012egss2_dat.txt";  *the main datafile;


*Control file import and processing;
proc import out=control
  file="&datadir.\Control - Documentation Driven Programming.xlsx"
  dbms=excel replace;
run;
proc sql noprint;
  select cats('fmtval',max(countc(values,'0A'x))+1)
    into :max_distinct_fmt_vals
    from control;
quit;


data prepped_control;
  set control;
  format fmtval1-&max_distinct_fmt_vals. $64.;
  if not(missing(values)) then fmtname=cats(variable,'f');
  array fmtvals fmtval1-&max_distinct_fmt_vals.;
  if not(missing(values)) then do _i=1 to countc(values,'0A'x)+1;
    fmtvals[_i]=scan(values,_i,'0A'x);
  end;
  format reporting_fmtname $32.;
  if statistic=:'top ' then
    reporting_fmtname=cats('top',scan(statistic,2,' '),compress(range,':'),'f');
  format page $2.;
  page=scan(table,1,'-');
  table_sas = translate(table, '_', '-');

run;


*Standard variable format preparation section;
proc sort data=prepped_control(where=(not(missing(fmtname)))) out=format_control;
  by class fmtname;
run;

proc transpose data=format_control
  out=format_control_vert(where=(not(missing(col1))));
  by class fmtname;
  var fmtval:;
run;

data format_cntlin;
  set format_control_vert;
  format start end label $64.;
  start=scan(scan(col1,1,':'),1,'-');
  end=ifc(find(col1,'-')>0,scan(scan(col1,1,':'),2,'-'),start);
  label=strip(scan(col1,2,':'));
  label_sas = tranwrd(strip(label),' ','_');
run;

proc format cntlin=format_cntlin;
quit;
```

```
*Extra module adding Top Box and Bottom Box formats for tabulation;
data reporting_format_cntlin;
  set prepped_control(drop=fmtname label);
  format label $32.;
  where statistic =: 'top ';
  rename reporting_fmtname=fmtname;
  **Top Box portion;
  end=scan(range,2,':');
  start=put(input(end,8.)-input(scan(statistic,2,' '),8.)+1,2.);
  label=catx(' ','Top',scan(statistic,2,' '),'Box');
  output;



**Bottom (remaining) portion;
  start=scan(range,1,':');
  end=put(input(end,8.)-input(scan(statistic,2,' '),8.),2.);
  label=catx(' ','Bottom',end,'Box');
  output;
  keep reporting_fmtname start end label;
run;

proc sort nodupkey data=reporting_format_cntlin;
  by fmtname descending start;
run;

proc format cntlin=reporting_format_cntlin;
quit;


proc sort data=prepped_control;
  by page table;
run;

data table_control;
 set prepped_control(where=(not(missing(table))));
 var_fmtname = cats(variable,'f.');
 reporting_fmtname = cats(reporting_fmtname,'.');
run;


**Input section;

%macro read_in(var,start,fmt,length);
  @&start. &var. &fmt.&length.
%mend;

%macro labelcode(var,label);
  &var.="&label."
%mend;

%macro apply_formats(var,fmtname);
  &var. &fmtname.
%mend;

proc sql noprint;
  select cats('%read_in(',variable,',',start,',',format,',',length,'.)')
    into :read_in_flat separated by ' '
    from prepped_control where start ne ' ';

  select cats('%labelcode(',variable,',%nrstr(',label,'))') length=1024
    into :labelcode separated by ' '
    from prepped_control where label ne ' ';
```

```
  select cats('%apply_formats(',variable,",",fmtname,".)")
    into :formatcode separated by ' '
    from prepped_control
    where fmtname ne ' ';
quit;

data input;
  infile datafile lrecl=20000 pad firstobs=2;
  format
    &formatcode.
  ;
  input
    &read_in_flat.
  ;
  label
    &labelcode.
  ;
run;


**Error Checking and Cleaning section;

%macro range_check(var,range);
  if not(missing(&var.)) and not(&var. in(&range.)) then do;
    type='RANGE';
    question="&var.";
    output;
  end;
%mend;

%macro base_check_all(var);
  if missing(&var.) then do;
    type='BASE';
    question="&var.";
    output;
  end;
%mend;

%macro base_check_restricted(var,base);
  if (&base.) and missing(&var.) then do;
    type='BASE';
    question="&var.";
    output;
  end;
  if not(&base.) and not(missing(&var.)) then do;
    type='BASE';
    question="&var.";
    output;
  end;
%mend;

proc sql noprint;
  select cats('%range_check(',variable,',%nrstr(',range,'))')
    into :range_check separated by ' '
    from prepped_control
    where not(missing(range));

  select cat('%base_check_all(',variable,')')
    into :base_check_all separated by ' '
    from prepped_control
    where missing(base) ;
```

```
  select cats('%base_check_restricted(',variable,',%nrstr(',base,'))')
    into :base_check_restricted separated by ' '
    from prepped_control
    where not(missing(base));
quit;

*Vertical dataset containing one row per variable per respondent that fails checks;
data checking;
  format question type $32.;
  set input;
  &base_check_all.;
  &base_check_restricted.;
  &range_check.;
  keep c2_caseid type question;
run;

proc sql noprint;
  create table badids as
    select distinct c2_caseid
    from checking;
  select nobs into :err_recs
    from dictionary.tables
    where memname='BADIDS' and libname='WORK';
quit;

%macro grab_bad_dat(respid,var);
  if c2_caseid=&respid. then do;
    bad_val=vvalue(&var.);
    variable="&Var.";
    output;
  end;
%mend;

%macro error_reporting(err_recs);

  %if &err_recs.=0 %then  %do;
    data No_err;
      errors=&err_recs;
      output;
    run;

    title 'There are no errors, the data is clean. No error output forthcoming.';
    proc print data=no_err noobs;
    run;

  %end;

  %else %do;

    title;
    proc sql;
      select cat('There are ',count(1),
        ' records with errors. Error reports and datasets following.')
        from badids;
    quit;

    title 'Summary list of errors by question and by type within question';
    proc freq data=checking;
      table question question*type/list;
    run;
    title;
```

```
    proc sort data=prepped_control(where=(not(missing(variable))))
      out=control_for_el(keep=variable range base);
      by variable;
    run;

    proc sort data=checking out=err_list(rename=question=variable);
      by question;
    run;

    **Here we use a temporary file, because it is possible lthis may exceed;
    **the maximum length of a macro variable;
    filename bad2dat temp;

    data _NULL_;
      format grab_bad_dat $512.;
      set err_list;
      grab_bad_dat=cats('%grab_bad_dat(',c2_caseid,',',variable,')');
      file bad2dat;
      put grab_bad_dat;
    run;

    data checking_vert;
      set input;
      format variable $32.;
      %include bad2dat;
      keep c2_caseid bad_val variable;
    run;

    proc sort data=checking_vert;
      by variable;
    run;

    data abbrev_err_data;
      merge err_list(in=a) control_for_el(in=b) checking_vert(in=c);
      by variable;
      if c;
      label type='Type of Error'
      bad_val='Bad Value';
    run;
  %end;
%mend error_reporting;

%error_reporting(&err_recs);

%macro range_clean(var,range);
  if not(missing(&var.)) and not(&var. in(&range.))
    then call missing(&var.);
%mend;
%macro base_clean(var,base);
  if not(&base.) and not(missing(&var.))
    then call missing(&var.);
%mend;

proc sql noprint;
  select cats('%range_clean(',variable,',%nrstr(',range,'))')
    into :range_clean separated by ' '
    from prepped_control
    where not(missing(range));
  select cats('%base_clean(',variable,',%nrstr(',base,'))')
    into :base_clean separated by ' '
    from prepped_control
    where not(missing(base));
quit;
```

```
**Final cleaned datafile;
data cleaned_data;
  set input;
  &range_clean.;
  &base_clean.;
  tabulateweight=100000*c2_weight;
  if tabulateweight>0 then weight_inv=1/tabulateweight;
run;


**Reporting section;
proc sort data=prepped_control;
  by class;
run;

%macro tab_mean(var,table,varlabel);
  *First run PROC TABULATE to generate results;
  proc tabulate data=cleaned_data out=_table_&table.;
    class &class./mlf missing;
    var &var. weight_inv;
    table (&var.*mean weight_inv*sum),(all &class.);
    weight tabulateweight;
    label weight_inv='Total Respondents';
    where not(missing(&var.));
  run;

  *Then reformat those results for reporting;
  data _table_&table.;
    set _table_&table.;
    array classes &class.;
    do _i = 1 to dim(classes);
      if substr(_type_,_i,1)='1' then columnvar=classes[_i];
    end;
    if compress(_type_,'0')=' ' then columnvar='All';
    Score=ifn(not(missing(&var._mean)),
      round(&var._mean,0.1),
      round(weight_inv_Sum,1.0));
    Stattype=ifc(not(missing(weight_inv_Sum)),
      'Total Respondents',
      'Mean');
    Table="&table.";
    if stattype='Total Respondents'
      then stub="&varlabel.";
    if strip(columnvar)='.' then delete;
    keep stub table stattype columnvar score;
  run;
%mend;

%macro tab_prop(var,fmt,table);
  proc tabulate data=cleaned_data out=_table_&table.(rename=&var.=stub);
    class &var. &class./mlf missing;
    var weight_inv;
    table (&var.*colpctn weight_inv*sum),(all &class.);
    freq tabulateweight;
    format &var. &fmt.;
    label weight_inv='Total Respondents';
    where not(missing(&var.));
  run;
```

```
  data _table_&table.;
    format stub $256.;
    set _table_&table.;
    array classes &class.;
    do _i = 1 to dim(classes);
      if substr(_type_,_i+1,1)='1' then columnvar=classes[_i];
    end;
    if (substr(_type_,1,1)='1' and length(compress(_type_,'0'))=1)
      or compress(_type_,'0')=' '
      then columnvar='All';
    Score=round(coalesce(of pct:,weight_inv_Sum),1.0);
    Stattype=ifc(not(missing(weight_inv_Sum)),'Total Respondents','Percent');
    Table="&table.";
    if stattype='Total Respondents' then stub=vlabel(stub);
    if strip(columnvar)='.' then delete;
    keep table stattype stub columnvar score;
  run;
%mend;

proc sql noprint;
  select variable into :class separated by ' '
    from prepped_control
    where not(missing(class))
    order by class;

  select cats('%tab_mean(',variable,',',table_sas,', %nrstr(',label,'))')
    into :tab_mean separated by ' '
    from table_control
    where statistic='mean';

  select cats('%tab_prop(',variable,',',var_fmtname,',',table_sas,')')
    into :tab_prop1 separated by ' '
    from table_control
    where statistic='freq' ;

  select cats('%tab_prop(',variable,',',reporting_fmtname,',',table_sas,')')
    into :tab_prop2 separated by ' '
    from table_control
    where substr(statistic,1,4)='top ';
quit;

ods results=off;
ods html close;

*run calculation stage;
&tab_mean.;
&tab_prop1.;
&tab_prop2.;

*Collect tables together;
data all_tables;
  format stub $256.;
  set _table:;
run;

proc sort data=all_tables;
  by table descending stattype stub;
run;
```

```
*Transpose to make column variables into actual columns;
proc transpose data=all_tables out=for_report;
  id columnvar;
  idlabel columnvar;
  var score;
  by table descending stattype stub;
run;


proc sql noprint;
  select  label_sas
    into  :columnord separated by ' '
    from format_cntlin
    where not(missing(class));

  select cats('%definecol(', label_sas,')')
    into :definecol separated by ' '
    from format_cntlin
    where not(missing(class));

  select distinct cats('%report(',scan(table,1,'-'),')')
    into :report separated by ' '
    from table_control;
quit;


*This macro can be expanded to include style options for columns;
%macro definecol(var);
  define &var./display;
%mend;

*This runs the actual reports;
%macro report(page);
  proc report data=for_report nowd spanrows;
    where substr(table,1,1)="&page.";
    columns
      (stub stattype)
      (All &columnord.)
    ;
    define stub/' ' display style={width=1.5in};
    define stattype/' ' display;
    define all/display;
    &definecol.;
  run;

%mend;

options orientation=landscape;

ods pdf file="&outdir.\&reportname.";

  &report.;

ods pdf close;
```

ACKNOWLEDGEMENTS

Marcus would like to thank Alan Roshwalb at Ipsos Public Affairs for encouraging me to write this paper.

Joe would like to thank Rich Hebel for igniting the seed of data-driven programming on day one of his employment, and Paul Silver for challenging him with new and different ways of approaching the problem.

Finally, both Marcus and Joe would like to acknowledge John Vidmar, Chairman of Ipsos US Public Affairs, who in his zeal for methodological rigor in every aspect of survey research planted the idea many years ago, and gave us the opportunity to refine our approach on project after project.

CONTACT INFORMATION

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

Joe Matise
NORC at the University of Chicago
65 E Monroe
Chicago, IL 60604
(312) 759-4296
matisejoe@gmail.com

Marcus Maher
Ipsos Public Affairs
222 S Riverside Plaza
Chicago, IL 60606
(312) 526-4933
marcus.maher@ipsos.com