Client Communication: Transform Specifications into Code with Spreadsheets
John Cantrell, University of North Carolina, Chapel Hill, NC

ABSTRACT:
Spreadsheets are a popular tool for communication among team members, often used to define SAS program specifications. These specifications, if properly formatted by the specifier, can be used by SAS programs to generate program logic. Making the program logic and specifications come from the same source document offers several advantages, including: making the specifier more responsible for accurate, clearly defined specifications, increasing the specifiers involvement in program development by giving them more control over the program’s results, requiring specifiers to write specifications before program is written, and providing all team members with an completely accurate record of what was done by the program.

PURPOSES:
• The reader should understand how spreadsheets containing specifications can enhance communication, especially if they are integrated into SAS program logic.
• The reader should understand a simple, detailed example of how to use very basic SAS tools to turn specifications into SAS program logic.

THE COMMUNICATION ISSUE:
A major challenge in information technology work is communication between team members. Communication between team members who work in non-technological roles and those who work in technological roles is often difficult because of differences in how we think and talk about our jobs.

Since our jobs require that we communicate with each other, managers look for ways to improve communication between team members, including periodic project meetings, team building exercises, and written documentation. Written documentation, for those of us who write SAS programs, nearly always includes program specifications.
• In very structured workplaces, program specifications are often clearly defined with a common structure and language.
• In unstructured workplaces, specifications may be written by the programmer because no one else will write them.
• In the worst case, specifications are not written at all, leading to programs that may or may not produce the desired results. However, since the desired results were never put into writing, it may be argued that the desired results are unknown, making it impossible to tell if the program produced them.

THE NEED FOR SPECIFICATIONS:
To build programs that produce intended results, good specifications are an essential tool. Good specifications have other, very important benefits including:
• less development time
• reduced errors
• a record of changes to program requirements (in situations where some people seek to assign blame, such records can save programmers jobs!)
• documentation for the program.

Documentation outside of the program is extremely valuable because all team members can read documentation even if they do not understand SAS code and do not have access to the program.

Documentation is an essential part of any program; without documentation, programs are expensive junk because once the programmer/author leaves the organization, the program’s useful life is over unless another programmer analyzes the program. Analyzing another programmer’s code is, in the author’s experience, is a most unpleasant and time-wasting task. To avoid this unpleasantness and time-wasting, documentation and specifications must accurately reflect the actions of the program.
CONCORDANCE BETWEEN SPECIFICATIONS AND PROGRAM:

Specifications can accurately reflect a program’s actions, but often there are discrepancies between programs and specifications, sometimes very significant discrepancies. To eliminate discrepancies and insure complete agreement between specifications and programs requires painstaking attention by the programmer, both to follow the specifications to the letter and to revise them when the program deviates. Another way to insure that specifications and programs agree completely is to use the specifications as a part of the program logic. Incorporating the specifications as a part of the program logic insures agreement between program and specifications. Specifications are often defined by using spreadsheet software such as Microsoft Excel. Spreadsheets are good for laying out specifications in the form of lists containing discrete blocks of information, since each row can form an item in the list and each column can form one discrete block of information within the item. Since SAS can easily read spreadsheets, it is possible to use the information in the specification spreadsheet to create program logic.

The process of extracting program logic from spreadsheet specifications is outlined in five steps, as follows:

1) Persuade clients to write specifications in a spreadsheet.
2) Persuade clients to document all changes to specifications in new versions of the spreadsheet.
3) Meet with clients and ‘design’ the format of specifications in the spreadsheet.
   a) Adapt the format to fit your clients’ needs so that they’ll use it!
   b) If they understand the format, they’ll understand what your program does.
   c) Your program can use any format, as long as it’s consistent.
4) Write your program to read the spreadsheet and translate it into SAS code.
5) When your clients want changes, remind them of their commitment to describe their changes in a new version of the spreadsheet in step 2 according to the format that they devised in step 3.

IMPLEMENTATION USING SAS:

SAS tools for turning specifications into SAS program statements include the following:

1) Put statements in data _null_s, which may be used to write code in an external file.
2) Proc SQL, which can be used to write data from datasets into macro variables and to include the data in SAS code.
3) SAS functions which operate on character strings, like:
   a) scan: selects a word
   b) index: searches for a substring
   c) substr: extracts a substring
   d) translate: replaces characters
   e) ||: concatenates character strings

Tools 1 is included for completeness and will not be discussed in this paper.
Tools 2 and 3 are discussed in a detailed example below.

A simple example of specifications in a spreadsheet format is shown in the table (which was copied from a spreadsheet) below:

<table>
<thead>
<tr>
<th>source variable</th>
<th>destination variable</th>
<th>Sue's variable description</th>
<th>Stem</th>
<th>Label description</th>
<th>John Transform</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>q8a</td>
<td>q8aBin</td>
<td>OPINION, good pets</td>
<td>Alligators make good pets.</td>
<td>Doesn't like reptiles.</td>
<td>1,2=0; 3,4,5=1</td>
<td>they eat too much!</td>
</tr>
</tbody>
</table>

The Sample Spreadsheet Table contains instructions for collapsing a variable named Q8a. “Collapsing” here means turning multiple responses into one response. Q8a is an answer to an opinion question that states “Do you agree or disagree that alligators make good pets?” The answers are given on the Likert scale, where 1 means “strongly agree”, 2 means “agree somewhat”, 3 means “not sure”, 4 means “disagree somewhat”, and 5 means “disagree strongly”. The client, a researcher named Sue, wants to collapse the original (source) variable into a dichotomous, binary (destination) variable called q8aBin. In
the collapsed variable q8aBin, survey respondents who think that alligators make good pets are coded 0 and all others are coded 1. The **collapse instructions**, or coding of the binary variable, are shown in the column “John Transform”. Note that the agreed upon “syntax” for collapsing a variable consists of sets in the form:

```
<old values>=<new value>
```

where the old values “1” and “2” are collapsed to “0” and the old values “3”, “4”, and “5” are collapsed to 1. Each old value/new value set, constituting a set of collapse instructions, is separated by a “;”.

The Sample Spreadsheet Table also shows columns “Sue’s variable description”, “stem”, “Label”, and “Notes”, which are all for Sue’s use. Sue uses these columns to help her and her colleagues understand the spreadsheet contents. Sue insisted on calling the variable description column “Sue’s Variable Description” and the transform column “John Transform”.

I let Sue have her extra columns and funny column names because if I can persuade her to keep track of her changes with the spreadsheet, then in most cases I can avoid making revisions to the code in my program and she has a rock solid record of what the program does. With that rock solid record, when she inevitably shows up in my office at some later time, for example, six months after the program was last run, asking me to explain what the program does, I can respond with the question “Did you look at the specifications that you created to see if you can answer that question?” Such moments are truly satisfying.

**READ THE SPREADSHEET:**

Once Sue gives me a draft of the spreadsheet, I write some SAS to read the row from the spreadsheet into a dataset named “main1.” My SAS program uses SAS Access; in my opinion it is the easiest way to use SAS to read/write to spreadsheets and databases. The SAS access libname engine reads the data from the Excel spreadsheet. Note the Excel filename contains an embedded date “January 6 2006”. This date allows Sue and me to identify the time when the spreadsheet was last modified and prevents previous versions of the spreadsheets (whose names contain a different date) from being overwritten. The macro variable “&sfiles” contains a path.

```
libname xl "&sfiles.\Control Predictor Variables January 6 2006.xls" mixed=no;
data main1;
set xl.'main$'n;
run;
libname xl clear;
```

**PARSING THE CHARACTER STRINGS FROM THE SPECIFICATIONS:**

The data in dataset main1 can then be parsed to prepare it for use as SAS code, as shown below:

```
data main2 (drop=i);
set main1 (rename=(Sue_s_variable_description=varDescription
source_variable=source destination_variable=destination john_transform=transform));
array old{10} $20;
array new{10} $10;
if index(transform,“=”)>0 then do;
equalscount=count(transform,”=”);
di = 1 to equalscount;
if i=1 then old{i}=compress(scan(transform,i,”=”),” “);
if i>1 then old{i}=compress(scan(scan(transform,i,”=”),2,” “),” “);
new{i}=compress(scan(scan(transform,i+1,”=”),1,” “),” “);
end;
end;
run;
```

The datastep above does the following:

- Variables are renamed for the convenience of the programmer.
- Two character arrays, “old” and “new”, are declared. In the example above in the Sample Spreadsheet Table, “old” would contain “1,2” and “new” would contain 0 or “old” would contain “3,4,5” and “new” would contain “1”.

```
The index function checks for the presence of an “=” in the variable “transform”, ensuring that the row of data from the spreadsheet contains instructions for collapsing a variable. Sue and I agreed that the presence of an “=” would mean “collapse stuff to the left of the = into stuff to the right of the =”.

The count function counts the number of “=” contained in the variable “transform” (the number of collapse instructions) and stores this number in a variable named “equalscount.” A do loop counts from one to equalscount, parsing the string in the variable “transform.” The code inside the do loop works as follows:

• The line that starts “if i=1” uses scan to find the first “word” in the string. Words here are delimited by “=”, so scan puts the string to the left of the first “=” into array element old{1}. In the Sample Spreadsheet Table, old{1} would be “1,2.” The compress function removes blanks from the string.

• The line that starts “if i>1” uses the inner call to the scan function to find the i-th “word” delimited by an “=” (in the Sample Spreadsheet Table, this would be “0; 3,4,5”) then uses the outer call to the scan function to put the part of this string to the right of the “;” (the string “3,4,5”) into the array element old{i}. Again, the compress function removes blanks.

• The line that starts with “new{i}=” uses the inner call to the scan function to find the (i+1)th “word” delimited by an “=.” In the Sample Spreadsheet Table, the first instance of this, where i=1, would be “0; 3,4,5.” Then the outer call to the scan function puts the part of this string to the left of the “;” (the string “0”) into the array element old{i}. The compress function removes blanks.

For the Sample Spreadsheet table, the code above puts the following data into the following array elements:

old1: contains “1,2”
old2: contains “0”
new1: contains “3,4,5”
new2: contains “1”

BUILDING LINES OF SAS CODE:

Again, “Old1” and “new1” form a pair that constitutes a single collapse instruction and “Old2” and “new2” form a pair that constitutes another single collapse instruction. They will be used in a subsequent datastep to create lines of SAS code, as follows:

```sas
data collapse1 (keep=code);
set main2;
array old{10} $20;
array new{10} $10;
length code $200;
if old{1}^="" then do;
   code="";
   do i=1 to equalscount;
      code=compbl("if \"source\" in \"old{i}\" then \"destination\"=" || new{i} || ";");
      output;
   end;
end;
run;
```

The datastep above does the following:

• Two character arrays, “old” and “new”, are declared to allow array processing on variables created in the previous datastep.
• A character variable “code” is declared, which will contain the SAS code generated in this datastep.
• A do block is conditionally executed if the first element of the “old” array contains data, avoiding creating code for collapsing variables if no collapsing instructions were found in the row.
• Within the do block, the “code” variable is initialized to missing.
• A do loop is executed from one to the variable “equalscount” (which contains the number of collapse instructions in the row). For each collapse instruction, a string is created of the form:
  “if ||source|| in ||old{i}|| then ||destination||= || new{i}||;”
where source, old{i}, destination and new{i}, are all variables. For the first collapse pair in the Sample Spreadsheet Table, the resulting string:
  if q8a in (1,2) then q8aBin=0;
is assigned to the variable “code.”
The output of this datastep is a SAS dataset "collapse1" that contains a variable named "code." The variable "code" contains SAS code that will collapse the data in the variable named in the "source" column and put the results into a variable named in the "destination" column, according to collapse instructions in the "transform" column.

**USING THE SAS CODE FROM THE DATASET:**

After generating the dataset that contains the variable named "code", the following proc SQL puts these statements into a macro variable named "&collapseCode":

```sas
proc sql noprint;
select code into :collapseCode separated by " "
from collapse1;
quit;
```

Once the SAS statements are loaded into a macro variable, they can be included in a datastep as follows:

```sas
data allData2;
set allData1;
&collapseCode;
run;
```

Now, the original data, in the form of a Likert scale containing 1, 2, 3, 4, and 5 includes a new data element that contains 1 and 0. The data have been collapsed to prepare for an analysis that requires binary data.

**CONCLUSION:**

Once the specification spreadsheet format is defined and the program is written, clients have the means to change program logic within certain limits without asking the programmer to make revisions. Most importantly, since the specifications in the spreadsheet must be accurate for the program to work, the specifications will serve both as accurate documentation that is accessible to the entire project team and as a valuable project communications tool.

**REFERENCES:**


**ACKNOWLEDGMENTS:**

I would like to thank Gary Rozier, for inspiring me to do my best and Kavita Mathu-Muju, for her patient collaboration.

**CONTACT INFORMATION:**

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

John Cantrell
University of North Carolina
Health Policy & Administration
1107E McGavran-Greenberg
CB# 7411
Chapel Hill, NC 27599-7411
(919)843-6495
john_cantrell@unc.edu

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration.

Other brand and product names are trademarks of their respective companies.