AD003

User Implementation and Revision of Business Rules
Without Hard Coding: Macro-Generated SAS® Code

By Michael Krumenaker, Sr. Project Manager, Palisades Research, Inc.
and
Jit Bhattacharya, Manager of Decision Support,
Prudential Property and Casualty Insurance Company

Computer programs often encode “business rules” prescribed by business users or other non-programming professionals (actuaries, tax specialists, etc.). Ideally, changes in business rules and related parameters would not require changes in the program code.

But the program implementing a set of business rules may be very intricate. A classic example is the definition of a “Rate Class Code” to represent a complex collection of car and driver characteristics (class code variables) to determine insurance premiums. Since insurance is a state-regulated business, the business rules governing the Rate Class Codes vary by state, and any given state’s premium rating methodology involves hundreds of different Rate Class Codes. Furthermore, the structure of Rate Class Codes varies substantially by state. For example, in some states, age, gender and marital status are permitted rating variables, while other states prohibit the use of some or all of these variables in insurance rating. For a company doing business in many States, hard coding all of this complexity could require the creation of dozens of separate programs or one very complicated program.

This complexity may leave the user with a mass of hard code which only a professional programmer can edit. Or the application could include a graphical interface through which the user can enter the details to interpret Rate Class Codes, perhaps even new class code variables, without having to modify the underlying code. But creating such a graphical interface could be cost prohibitive.

Here is another option. The user interface can be a familiar tool, such as Microsoft Excel®. The calculation and presentation engine can and should remain a powerful programming language, with appropriate database, data warehouse, statistical and/or computational capabilities, such as SAS®. The trick is to make the application program code flexible enough so that the user can easily add or change parameters, formats and business rules, and even to add new variables, through the familiar user interface rather than changes in the program code.

One of the co-authors of this article created a sophisticated Excel workbook containing dozens of groupings of Rate Class Codes and associated values. He asked Palisades Research to design and build a unified program code for all States that would interpret any Rate Class Code and produce reports on performance metrics by grouping of class code variables. Palisades Research implemented this aspect of the overall application by (1) adding new worksheets to the Excel workbook to set or change allowable values and formats for each class code variable, and (2) using SAS macros to read and interpret the information from the workbook and automatically generate the program code during each month’s record processing. The business user can now change the way the program interprets the class code, even adding a new class code variable, without changing any lines of the program.

The Rate Class Code and Class Code Variables

1. The Rate Class Code and Class Code Variables

The Rate Class Code is a six-character text string. The characters are usually numeric, but can be alphas. Interpretation of the six-character Rate Class Code, in conjunction with the identity of the State and certain other items, provides the values of certain factors (class code variables) used in insurance policy rating. To avoid divulging rating methods, we will call the class code variables CLASS_CODE_VAR_1 – CLASS_CODE_VAR_10.

The value of each class code variables is determined by all of the following:

(1) The State (for the majority of States) or, where the State is not a factor, the insurer’s subsidiary involved.

(2) A single character, or ordered sets of two or three characters, from the 6-character Rate Class Code.
(3) In some cases, the value of a particular variable (\texttt{CLASS\_CODE\_VAR\_4}) which itself depends on the class code.

2. The Rules Worksheets

Figure 1 shows the top 19 rows of the Excel worksheet ("rules worksheet") which completely describes how to interpret the class code variables in States where the identity of the State (rather than the subsidiary) is a factor. We have substituted State1, State2, etc. for the actual states. \texttt{class\_code\_var\_4} has no effect on the first two class code variables, so all the values in Figure1, column C are "N/A". Compare Figure 3, which shows a portion of the same rules worksheet for a variable that is affected by the value of \texttt{class\_code\_var\_4}. A similar worksheet (not shown here) contains the rules to interpret class codes in which the subsidiary, instead of the State, is a factor.

3. Worksheet Listing the Class Code Variables

Every variable in column B (\texttt{class\_code\_var\_[1-10]}) is listed in another worksheet named cc_vars (figure 2). The authorized business user can create a new class code variable by adding it to cc_vars and describing the interpretation rules in the State-dependent (see Figures and 3) and/or Subsidiary-dependent worksheets. To prevent input errors, the rules worksheets use drop-down boxes in column B in figure 1, the list input for which is from column A in cc_vars. The cc_vars sheet also includes information used only by the SAS macro. Column B of cc_vars holds the actual variable names to be used by the program. This sheet could be used by SAS 6, since the variable names are eight characters long. The variable length – which we control so as to avoid wasting disk space – is prescribed in column D.
A SAS macro uses the 4_char values to create format names and certain macro variables. The SAS program first reads a csv file created from cc_vars into a temporary data set called “dims,” whose variables correspond to the columns in Figure 2. Similarly, the subsidiary names are limited to a list found on its own worksheet (Companies). The authorized user can add companies, as described above for cc_vars. The SAS program will read them.
4. Worksheets Listing Allowable Values and Formats for Each Class Code Variable

Each individual class code variable is described in its own worksheet (see Figure 4). Formats are in column A, values are in column B. The user can add, remove or change these values and formats.

![Figure 4: Worksheets cc_vars_1, _2 and _3](image)

We create individual .csv files for the State-dependent rules worksheet, the subsidiary-dependent rules worksheet, cc_vars, the subsidiary listing, and each individual class_code_var. We automated this with VBA, but that it not our focus. The lists in figures 2 and 4 can also be used to populate Excel drop-down boxes to prevent user input error (also an Excel issue).

The SAS Program Using Macros to Create Code from the Workbook

1. Creating and Counting the Class Code Variables in cc_vars

The program starts with no “knowledge” about any of the variables. It obtains this information – even the number of such variables - by reading the csv file derived from worksheet cc_vars (Figure 2) into a data set `dims` which has the following variables (corresponding to the four columns in figure 2):

<table>
<thead>
<tr>
<th>var_lbl</th>
<th>Variable label (e.g., CLASS_CODE_VAR_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>Variable name for SAS code (e.g., CC_VAR1)</td>
</tr>
<tr>
<td>char_4</td>
<td>4-character abbreviation of variable name</td>
</tr>
<tr>
<td>len</td>
<td>Variable length</td>
</tr>
</tbody>
</table>

Table 1: Information from cc_vars worksheet

Each of the 10 rows in cc_vars becomes a record in `dims`. A data step derives five global macro variables from each record in `dims`, using N (derived from the SAS automatic variable `_N_`) to distinguish between variables from each record. `dims` has 10 records, so N is 1, 2, 3...,10. A simple Call Symput to create each of the fifty (5 x 10) macro variables does the job:

<table>
<thead>
<tr>
<th>cc_vars column (Fugure 2)</th>
<th>Type</th>
<th>Description</th>
<th>Example: For N=1</th>
<th>Value for N=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>var_N</td>
<td>Variable name for SAS code (e.g., CC_VAR1)</td>
<td>&amp;var_1</td>
<td>CC_VAR_1</td>
</tr>
<tr>
<td>A</td>
<td>var_lbl_N</td>
<td>Variable label (e.g., CLASS_CODE_VAR_1)</td>
<td>&amp;var_lbl_1</td>
<td>class_code_Var_1</td>
</tr>
<tr>
<td>C</td>
<td>char_4_N</td>
<td>4-character abbreviation of variable name</td>
<td>&amp;char_4_1</td>
<td>VAR1</td>
</tr>
<tr>
<td>D</td>
<td>len</td>
<td>Length of variable var_N</td>
<td>&amp;len_1</td>
<td>1</td>
</tr>
<tr>
<td>derived from B</td>
<td>low</td>
<td>Lowercase var_N</td>
<td>&amp;low_1</td>
<td>cc_var_1</td>
</tr>
</tbody>
</table>

Table 2: Macro Variables and Their Values from cc_vars

We also use the counter `_N_` to set the value of a single global macro variable (`&num_vars`) to the number of class code variables (10). We use the “low” macro variable much later in the process, because the csv files use lowercase names.
2. Allowing for all Possible Combinations of Factors Determining Values of Rate Class Code Variables

As explained earlier, the value of each of the ten class code variables is determined by the State or company (as the case may be), the value of CC_VAR_4, and the value of one or more of the six characters which make up the class code. Referring to Figure 1, note that each row contains values for only one or two class code characters. This means that the other characters do not affect the value of the particular class code variable (column B) in the States in column A given the value of CC_VAR_4 (if not N/A) in column C. In the rule in row 3, only the second of the six class code characters is relevant. In other rows (rules), other characters in columns F-K are relevant. Class code character #2 (column G) is also used to determine the value of other class code variables in other rows of the worksheet, alone or in combination with other columns.

The program creates one macro variable for each possible combination of eight relevant factors (subsidiary, CC_VAR_4, and the 6 class code characters or “digits”). The factors include the company (even where irrelevant), but State is considered separately from these eight factors. For each of the 10 (&num_vars) class code variables, we create 255 global macro variables by concatenating its char_4 (short) name (e.g., VAR1, VAR2,...) with every number between 1 and 255 (2 x 2 x 2^6 - 1), for a total of 2,550 such macro variables, named VAR1_1, VAR1_2,...,VAR2_1,...,VAR10_255:

%global activnum activbin;
%macro doglobal;
%do i = 1 %to &num_vars;   /*&num_vars = number of class code vars;*/
  %do j = 1 %to 255;
    %let tempmac = &&chr4_&i;
    %global &tempmac.&j;
    %let &tempmac.&j = 0;   /*Initialize each one at 0;*/
  %end;
%end;
%mend;
doglobal;

Ultimately, not all 2,550 variable will be used - only whose representing combinations of the eight characteristics which actually occur in the rules worksheets. The term “combinations” refers to which of the eight columns is relevant in a particular row (rule), not the actual values (such as >4 or ALL) in each such column. For instance, the first row of figure 1 says that of the aforementioned eight factors only class code character #2 affects the value of Class_Code_Var_1. The company has no effect in any of the determinations governed by the rules in Figure 1. Expressed as eight bits, the first row of figure 1 corresponds to the binary code 00001000 (16), in which the eight bits represent the following:

Whether Company has effect (bit 1)
Whether CC_VAR_4 has effect (bit 2)
Whether characters 1, 2, 3, 4, 5, and/or 6 of the Rate Class Code has effect (bits 3-8, respectively)

Later we determine which of these 2,550 macro variables represent combinations which are actually used. For now, %doglobal initializes every one of them to 0.

In a separate data step, we create an additional 255 generic macro variables to correspond to 255 possible combinations of 1/0 in an 8-character string (except for 00000000, which would mean no info is used) represented as binary numbers, using the expression bin = put(i,binary8) to generate each of 255 string values for bin and ultimately for macro variables &bin1, &bin2,...,&bin255. For example, &bin16 = '00010000'.

3. Formats and informats

The program creates all the code to create and run proc format for each of the class code variables. Worksheets such as those in figure 4 provide the values; the rest is basic macro coding.

The Excel workbook provides the program with all values allowed for each class code variable and each factor affecting it, including the names of companies where company identity (not State) is a factor. The “long form” of each company name is used in the rules worksheet, while the SAS data sets store a shorter form to save storage space. One column of the company worksheet (which has the same layout as any class code variable worksheet in Figure 4) contains the long names, which the program uses as informats to obtain the short names (the other column). This is needed to separate multiple company names in the same cell by commas, and also allows the rules worksheets to have spaces within a given name (Apex Manufacturing).
4. **Reading The Rules**

One data step reads the csv file created from the state-dependent business rule worksheet. Each row is a rule and each column of columns is represented by a variable:

<table>
<thead>
<tr>
<th>Column</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>states</td>
</tr>
<tr>
<td>B</td>
<td>cc_var (long form)</td>
</tr>
<tr>
<td>C</td>
<td>CC_VAR_4 (value of CC_VAR_4 or N/A)</td>
</tr>
<tr>
<td>D</td>
<td>desc (description)</td>
</tr>
<tr>
<td>E</td>
<td>actual value of variable cc_var per this rule (this row)</td>
</tr>
<tr>
<td>F-K</td>
<td>class code digit1, digit2, digit3, digit4, digit5, digit6</td>
</tr>
</tbody>
</table>

Table 3: Review of Contents of Each Column in State-Dependent-Rules Worksheet

The variable company is created, but has the value missing in each rule described in the State-dependent rules worksheet.

A similar data step reads the company-dependent sheet, using the same variable names, and reading company with the aid of the informat described in section 4, above. This time the company variable is populated in each record of the data set.

The State-dependent and company-dependent rules data sets, which have the same structure, are then concatenated into a single data set named classall.

5. **Controlling the Process**

Recall that Table 2 showed how we used the information in cc_vars to convert variable names, lengths and labels into macro variables &VAR_1-10, &char_4_1-10, etc. A controlling macro (%all_var) calls a macro (%dovar) once for each of the 10 class code variables (iCount 1 to 10, below). %dovar takes the values from Table 2 as parameters:

```sas
%let tempmac1 = &&var&iCount;
%let tempmac2 = &&chr4_&iCount;
%let tempmac3 = &&low&iCount;
%let tempmac4 = &&vlbl&iCount;
%put Doing &tempmac2;
filename &tempmac1 %str("&csvdir./&tempmac3..csv");
%if %sysfunc(fexist(&tempmac1)) %then %do;
   %dovar(&tempmac1,&tempmac4,&tempmac2);
%end;
```

%all_var controls the process of creating final records from the business rules worksheets and formats for each variable. It calls %dovar 10 (&num_vars) times. %dovar takes as parameters the long variable name, label, and short name as parameters, e.g., %dovar(CC_VAR_6, class_code_var_6, var6).

IMPORTANT: %dovar is called for each class code variable in turn, so to create the final records corresponding to permutations of class code values – described below – we perform all the steps on a given class code variable before proceeding to the next class code variable. We will refer to the number of each class code variable as N (e.g., VARN).

6. **Handling Multiple Rules For Multiple States and Multiple Combination of Class Code Characters**

Recall from Figure 1 that a given worksheet has several rows (rules) for each of the 10 class code variables, and that a given rule can apply to multiple states and involve multiple combinations of values of class code characters. For instance, row 15 applies to three states and has 20 (2 x 10) possible class codes resulting in “Liability+Collision”. So, we must “explode” row 15 “ into 60 separate records in a SAS data set. %dovar calls all of the macros needed to explode these rows, starting with step (c), below.

a. **Separate the Concatenated Rules Dataset by Class Code Variable (10 data sets)**
Before dealing with individual rows, macro \%subfile(xlvar,sufx) breaks the single concatenated business rules data set (classall) into 10 data sets, one for each class code variable. \%subfile is run once for each class code variable (xlvar). The values of sufx (VAR1, VAR2, ..., VAR10) are used to name the ten output data sets: clasvar1, clasvar2, ..., clasvar10.

b. “Turning on” Relevant Binary Macros

\%subfile also determines which of the 2,550 macro variables representing all possible combinations of factors (not their values) should be set to 1; i.e., which of the combinations is actually used in a business rule for a given class code variable. For instance, if a given row in the state-dependent rules worksheet (for which company is irrelevant) describes a rule for class_code_var_6 in which the value of class_code_var_4 is not N/A and only digits 2 and 3 are not blank, \%subfile, sets a variable called holdchars to “01011000” (=88). If this were a company-dependent rule, the first character (digit) of holdchars would be 1. The holdchars string is read into variable holdnum as if it were the binary number 88. If and only if – the particular rule uses at least one of the 8 factors (not including State), then the VALUE of the macro variable named by concatenating the short class code variable name and the decimal representation of the relevant factors is set to 1. Put simply, if the data step evaluates a rule for the class_code_var_6 and the relevant factors for a particular rule is represented by 1011000, then \%subfile changes the value of the global macro variable &var688 from 0 to 1:

```plaintext
holdnum = left(trim(input(holdchars,binary8.))); if holdnum ne "0" then do;
call symput("sufx"||holdnum,'1');
```

At this point, each record in the data set classvarN (where N is 1, 2, ..., or 10) may still apply to more than one state and/or more than one combination of values of class code digits (the actual values in columns F-K in figure 1).

c. Explode multi-state records

\%dovar calls macro \%xpnd_st(ratfile), in which ratfile is, in turn, classvar1 – classvar10. We “explode” each multi-state record using simple looping and scanning, so that a single record with 20 states explodes into 20 records:

```plaintext
do iCount = 1 to 100; *More than enough for all states;
state_cd = left(trim(scan(states,iCount,',')));
if state_cd = "" then iCount = 101;
else output xpnd_st;
end;
```

The data set produced by this step is called xpnd_dg0. “dg” refers to class code digit (class code characters, column F-K).

d. Explode permutations of class code variable values

For a given class code variable, \%dovar now calls macros which analyze the six class code character fields in each record of xpnd_dg0, exploding records with multiple permutations of values for a given class code character in the worksheet row. The inventor of the original Excel workbook created the syntax for columns F-K to represent the possible values of each class code character, as per Table 4. The syntax is illustrated in columns F-K in Figures 1 and 3.

<table>
<thead>
<tr>
<th>Rules Worksheet Cell Syntax</th>
<th>Example</th>
<th>Created From Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single character</td>
<td>3</td>
<td>Single record (3)</td>
</tr>
<tr>
<td>Multiple class code digit values separated by commas</td>
<td>3,4,7,9</td>
<td>Four records, one for each value</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt; 7</td>
<td>Two records (one for 8, one for 9)</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt; 4</td>
<td>Four values (one each for 0, 1, 2, 3)</td>
</tr>
<tr>
<td>ALL</td>
<td>ALL</td>
<td>36 values (one each for 1,2,3,...,9 and A-Z)</td>
</tr>
</tbody>
</table>

Table 4: Syntax of Rules Worksheet Columns for Class Code Characters 1 – 6

i. Multiple Values Separated by Commas

There are up to 10 values (0,1,2,...9) in a comma-delimited cell, and there are six class code characters. So, we call on a new macro \%digcomma(num) six times, where num is 1-6 (one for each of column F-K), respectively. The data step in this
macro scans the class code character corresponding to num (the first, second,...,sixth), and outputs a separate record for each comma delimited value. If the value for the class code character (1, 2, 3, 4, 5 or 6, as the case may be) being tested has no comma (a single character, or “>5”, etc.), the record is output as-is; otherwise, explosion requires nothing more than a simple scan() within the data step and an output statement in a loop.

We call %digicomma(1), then %digicomma(2), through %digicomma(6). The output data set of %digicomma(1) becomes the input data set of %digicomma(2), and the output of (2) becomes the input of (3), and so on. This is easily achieved by looping through &i = 1 to 6, using the statement %let priornum=%eval(i-1), and using &priornum in the set statement. The final resulting data set (from %digicomma(6)) is the input data set for the next step.

ii. Multiple Values: Greater than or less than

As in step (i), we have six iterations of the macro (here called %xpnd_sym) corresponding to each of the six characters. The output of one iteration becomes the input of the next, just as in step (i). Simple substring and index functions find “>” or “<” if either occurs in a given class code character value. If the data step finds “>” or “<”, it is a easy to output the appropriate values, e.g., it finds “>6” in a record, it outputs three records with “>6” replaced by 7, 8, and 9, respectively. There are different approaches (assigning a macro variable such as &mydig1 to each variable such as digit1, or using an array such as mydig[6] with elements digit1-digit6). Records with no “>” or “<” in the class code character being analyzed are output as-is. As in step (i), the final resulting data set becomes the input data set for the next step.

iii. ALL

In the rules workbook syntax, ALL means every digit from 0 to 9 and every uppercase character from A-Z. As before, we do one character at a time, so we iterate six times (one for each class code digit), and the output from one data set becomes the input to the next. Records which do not have ALL in the class code character being analyzed are output as-is. If the value is ALL, we output 36 records (one each for 0-9 and A-Z, respectively), using simple do loops.

The output (which – remember – is for a single class code variable such as class_code_var_1) now contains all permutations of the actual values of State or Company, as the case may be, CC_VAR_4, and each of the class code characters for any rule describing values for that class code variable. For example, row 13 in figure 1 explodes from 1 record into a total of 576 records (2 states x 8 comma-delimited digits x 36 [0,1,2,3,4,5,6,7,8,9,A-Z from ALL] = 576 combinations). These 576 records are included with records derived from the other rules for class_code_var_1. After the sixth iteration, the data set goes through one more minor transformation (not described here) and emerges as xpnd&sufx, where &sufx is VAR1 (or VAR2, etc.).

7. Generate One or More Lookup Tables for This Class Code Variable

Recall that there are 255 flags for each of the 10 class code variables (e.g., VAR1_1 to VAR10_255). In step 6(b) we changed the value of flags representing “relevant” combinations of factors from 0 to 1. Here’s why. For each case in which the flag=1 we dynamically create a data step. We do not create 255 data steps (and data sets) for each class code variable. We create only those for which the flag (e.g., VAR624) is 1, generally no more than 4 data sets per class code variable. Note: In step 6(d)(iii) we described the explosion of one worksheet record into 576 SAS data set records. This provides the possible values for the rule in row 13 of Figure 1, but it is still only ONE RULE represented by ONE of the 255 flags.

If N is the number of the class code variable (1-10), then we loop through VARN1 to VARN255 and, where the value of that flag is 1 (from step 6(b)), we generate all the statements for a data step to create the lookup table:

1. A data statement (e.g. data var624).
2. A set statement (which refers to the data set which holds all business rule records for this VAR (step 6(d)(iii)).
3. Selection criteria (to determine which records we will output from the Set).
4. Drop statements (to drop class code characters and/or CO_CD (company code) and CC_VAR_4, if not relevant.)

Data statement: As an example, if &var624 is 1, then we create the data statement data var624.

Set statement: The input data set is xpnd&sufx. For the first class code variable, the set statement is set xpndVAR1.

Selection criteria: Remember that even after explosion, the data set xpnd&sufx (e.g., xpndVAR1) contains all the records for a given class code variable (and there will be 10 such data sets, one for each class code variable). From the data set for this
class code variable, the generated data step must choose those records having any value in the fields which track the 8-character binary representation of a number between 1 and 255. Example: If the data statement is data var624, 24 in binary is 00011000, so choose records for which there are non-blank values in the fields DIGIT2 and DIGIT3, N/A for CC_VAR_4, and blanks for CO_CD, DIGIT1, DIGIT4, DIGIT5 and DIGIT6. If we had flipped four out of the 255 flags for VAR1 from 0 to 1, we will break up the data set for VAR1 into four data sets, one for each flipped flag.

We use the generic binary global macro variables for the 8-bit string equating to 1-255 (&bin1, &bin2, ..., &bin255) created in Step 2 for the values for which the flag = 1. Remember, the value of each of these &bin variables is an 8-character string of zeros and ones. So, these &bin variables act as templates. Suppose the flag &VAR624 is 1. The string represented by &bin624 string is 00011000. We translate this concept into “if” statements to be included in the generated data set (using a state-dependent case as an example), matching each position in the binary string to the factor it represents:

```plaintext
%if %substr(&&bin&num,2,1) = 1 %then %str(if CO_CD ne "");
%else %str(if CO_CD = "");
%if %substr(&&bin&num,3,1) = 1 %then %str(if digit1 ne "");
%else %str(if digit1 = ");
```

Drop statement: Only the variables which are relevant in a particular rule will be used in matching to the monthly records file. The others must be DROPPED from the data set, otherwise they will overwrite the values of their counterparts in the monthly customer records data set being processed! We dynamically generate the appropriate drop statement for the data set being generated. Again we use the 255 generic &bin strings from step 2 in statements such as

```plaintext
%if %substr(&&bin&num,3,1) = 0 %then digit1;  *num is between 1 and 255.;
```

8. Generate indexes for the lookup tables

Another macro generates the index for the lookup table by creating a macro (%IDX) which includes (as text, since these are just variable names, not values) STATE_CD (always), and includes or excludes CO_CD, CC_VAR_4, DIGIT1, DIGIT4, DIGIT5 and DIGIT6. So, when the data step executes, the only variables left in this lookup data set are the ones being matched to the monthly file (State, DIGIT2, DIGIT3), the class code variable, and the value of that class code variable (see Column D of figure1). In the example described in step 6(d)(iii), this lookup data set has 576 rows, accounting for all the combinations of State, DIGIT2 and DIGIT3 in this rule.

9. Create Lookup Tables and Indexes for All Other Rate Class Code Variables

Remember that there are nine other class code variables (cc_var2, cc_var3, ..., cc_var10). The starting point for each class code variable is its separate data set created in Step 6a. The process is the same as in steps 6b-6d and 7-8.

10. Generate the Data Step That Applies the Lookup Tables to the Large Set of Customer Records

Finally, we generate the data step to match customer records to the generated, indexed lookup tables. Let’s call the output data set out_set. The data step creating out_set uses multiple SET statements to make use of the lookup tables. The macros referenced by this data step are described below. In order to match to each class code character (digit), the data step parses the class code in each monthly record into DIGIT1, DIGIT2,...
%do_sets(CC_VAR_4,varshort); %*Special call for CC_VAR_4;
%all_sets;
drop digit1 digit2 digit3 digit4 digit5 digit6;
run;

i. Initializing class code variables from data in cc_vars (figure 2)

%do_types: The 10 class code variables must be added to the customer record data set. Lengths, formats and labels are obtained from the macro variables created in Step 1 from cc_vars (Figure 2). %do_types generates statements such as:

Length CC_VAR_1 1;
Format CC_VAR_1 VAR1;
Attrib CC_VAR_1 Label="Class_Code_Var_1";

For safety, %do_types uses %sysfunc(fexist()) to check whether a csv file (see Figure 4) exists for each variable listed in cc_vars, skipping any variable for which the individual csv file is missing.

ii. The initial Set statement

The customer records (custrecs) is a large data set which includes the actual values for the State, Company Code, and Class Code. We use these values to generate the values of CC_VAR_1, CC_VAR_2, ..., CC_VAR_10 for that data set.

iii. Set statements for lookup tables

The macro %all_sets makes one call to %do_sets for each class code variable (in this case, 10). The initial call is to evaluate CC_VAR_4, since its value affects the other class code variables. As in (ii), %all_sets verifies that each class code variable really has a csv file. Parameters for %do_sets are the variable name (var) and the set name (e.g., VAR624 – see above). The set name pulls double duty, since it is also the macro variable name for the corresponding macro flag variable. So, selection of data sets for the SET statements by %do_sets is based on the macro flags since %if &&&setname&i = 1 is true only if the value of the flag (e.g., VAR624) is 1:

```
%let marker = 0;
%do i = 255 %to 1 %by -1;          *iterate downward;
  %if &&&setname&i = 1 %then %do;  *&&&setname&i: the flag macro variable;
    %*marker is 1 if there was a previous hit for this variable.;
    %if &marker = 1 %then %do;
      if &var = "" then do;
        set &setname&i key = X%upcase(&setname)&i / UNIQUE;
        if _IORC_ <> 0 then _ERROR_ = 0;
      end;
    %end;
  %else %do;
    set &setname&i key = X%upcase(&setname)&i / UNIQUE;
    if _IORC_ <> 0 then _ERROR_ = 0;
    %let marker = 1;    *Prevent reprocessing of records.;
  %end;
%end;        %*if setname_i = 1...;
%end;          %*do i = 255 to 1 by -1;
```

We iterate downwards from 255 to 1 to do more restrictive matches before less restrictive ones. For instance, the structure of the state-dependent business rules csv file may be such that the following two indexed lookup data sets exist for VAR6:

```
var648  Index: DIGIT1 DIGIT2
var6120  Index: ADLTYTH DIGIT1 DIGIT2
```

Were we to match to var648 before var6120, we might pick up matches which should really be to var6120.
CONCLUSION

The concept of separating business rules from code is more important than the actual techniques involved. Flexible code which can adapt to changes in business rules which the client can easily input adds significantly to the value of the application program. The user interface can be a tool well-known to business users, with an agreed-upon syntax adequate to describe all reasonably expected values and formats for variables affected by the business rules. SAS macros can provide such code flexibility relatively simply.

CONTACT INFORMATION

Michael Krumenaker
Sr. Project Manager
Palisades Research, Inc.
75 Claremont Road, Suite 312
Bernardsville, NJ 07924
908-953-8081
mkrumenaker@palisadesresearch.com

Jit Bhattacharya
Manager of Decision Support,
Prudential Property and Casualty
Insurance Company
23 Main Street
Holmdel, NJ 07733
732-816-8036
jit.bhattacharya@prudential.com

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 registered trademarks or trademarks of their respective companies.

1 The developer must provide the business user with simple rules for making such changes, e.g., the ordering of values and how to preserve the integrity of range names. These are Excel matters, not discussed here.

2 CC_VAR_4 is treated differently from other variables because while it is a class code variable it is also a factor (the third binary digit in the 8-digit string) affecting other class code variables.