CONTROLLING YOUR PROGRAMS WITH DATA: AN EXAMPLE IN PHARMACEUTICAL LABORATORY DATA REPORTING

Paul Wehr, StatProbe Inc., Ann Arbor, MI
Albert Mo, Parke-Davis Pharmaceutical Research, Division of Warner-Lambert Company,Ann Arbor, MI

ABSTRACT

Two of the arguments for object-oriented programming are: 1) The distinction between data and the functions that control data is largely an artificial one, and 2) The real-world is not static, so neither should the programs that attempt to manage it be static. While an object-oriented base language is not available, we can still make use of this object-oriented philosophy.

Using a limited amount of macro language programming, we moved a great deal of our programming logic into data sets. This may seem to be an unnecessary level of complexity at first glance, but especially for dynamic applications, the benefits of this approach will become clear. We use clinical laboratory reporting in a pharmaceutical research environment as an example. This environment is constantly changing, with the introduction of new assays, equipment, units and accuracy. Any program that is entrusted to manage this environment must be as active as its real-world counterpart. This paper explores the data driven programming we used to manage this dynamic environment.

INTRODUCTION

The pharmaceutical industry is closely supervised by the Food and Drug Administration (FDA). The FDA would like to see all computer applications go through a rigorous validation protocol. This approach is only practical when applications reach a final, fixed state. Unfortunately for programmers in the pharmaceutical industry, this environment is one that is unusually difficult to standardize. Programs are rarely developed for more than a few clinical studies, and no two studies are exactly alike. This creates an environment where application programmers are constantly programming "exceptions" into an otherwise standard application.

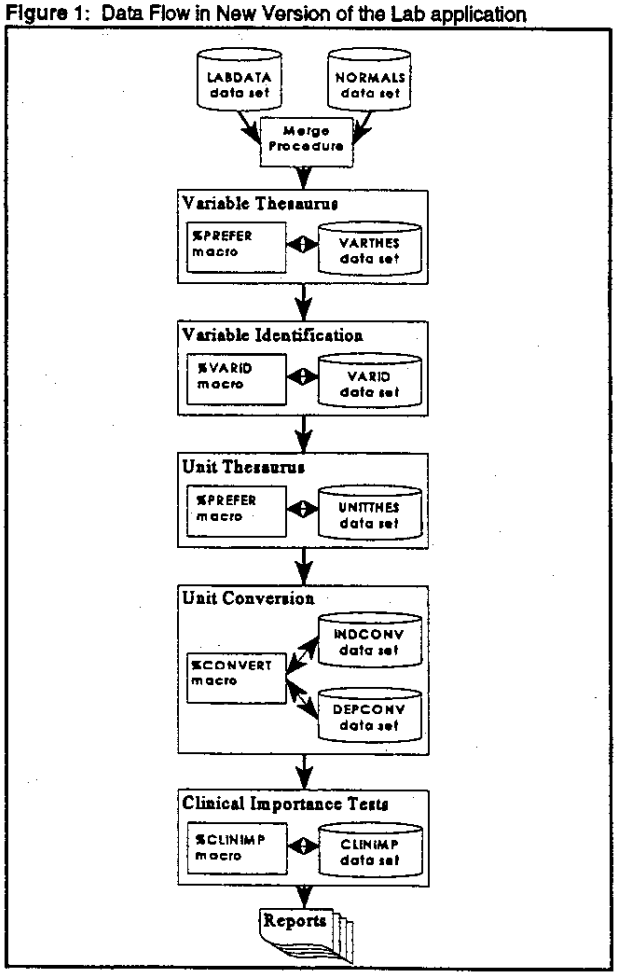
To address the conflict between regulatory requirements for validation, and increasing efficiency in development, we separated the dynamic programming information from the otherwise fixed information. All of the variable information extracted was stored in data sets, which are, after all, designed to manage variable information. These data sets are integrated into the application through corresponding macros. To implement this approach, we separated the application into logical operations. For each of these logical operations, we created an object consisting of a macro, and one or more data sets, that were designed to work together (see figure 1).

We applied this approach to the re-development of an existing application. This application is generically referred to as "The Lab Macros," and manages all aspects of lab data analysis and reporting. It does not, however, involve itself with the data collection or database building process. Using the re-development of this application as an example is useful for two reasons: 1) having the application already programmed allows us to concentrate on implementing the new approach while being able to compare and contrast with the previous version and 2) the environment is replete with examples of the dynamic environment this paper addresses.

Within this paper, we illustrate the technique of storing program-dependent variable data in data set/macro objects. We will also illustrate the advantages gained by using such an approach. Some of these advantages include:

- Maintenance of the application is simplified
- Tracking of modifications is simplified
- Reporting and analysis of parameters is available
- No need to modify source code for enhancements
- Application enhancement is more tightly controlled
- Internal Consistency verification is possible
- The object database facilitates user inquiry/reporting
- Objects can be shared with other applications

Figure 1: Data Flow in New Version of the Lab application



While the pharmaceutical industry is most profoundly effected by the validation requirements imposed by the FDA that drove the development of this approach, many other industries may feel similar or identical pressures within their own environments, and therefore may be able to apply the concepts illustrated here in their own applications.

ASSAYS AND THEIR UNITS

Assays are the fundamental objects in a laboratory analysis. They may include anything from the percentage of whole blood that consists of red cells, to the amount of nitrogen in the urine, to the presence or absence of antibodies, and any combination of dozens of other useful analysis quantities. Each of these assays may be represented in one of many choices of units appropriate to that test, and each of these units has a mathematical relationship to the other units in its family.

During the re-development of the Lab Macro system, we identified three logical processes necessary to identify each assay and its units. These processes are 1) identifying the preferred assay name, 2) identifying the assay to which the preferred name refers, and 3) identifying the preferred unit name. It is interesting to note that the processes we defined actually mimic the processes that go on in our own minds, if we were to perform this task manually. The first two processes answer the question "What variable name is this, and what test is it referring to", whereas the last answers the question "Can these units be represented more simply?"

In the current environment, the data is stored in tables where each assay's results are recorded in its own variable. After transposing this data, each assay has its own unique 8-character identifier--the variable name. Unfortunately, collecting data from more than one testing facility has often given us more than one 8-character name for the same test (see figure 1.1). This creates an environment where there may be more than one name for the same test. These names are effectively synonyms of one another.

Figure 1.1: Old approach to synonyms

```
PROC FORMAT;
VALUE $NAME  'HEMO'   ='Hemoglobin'
                .
                .
                .
             'UNA'    ='Urine sodium'
             'UNAN'   ='Urine sodium'
             'UK'     ='Urine potassium'
             'UKN'    ='Urine potassium'
             'KPLAS'  ='Potassium plasma'
             'RENIN'  ='Renin'
             'FT4I'   ='FT4I'
             'TSH'    ='TSH'
             'APPEAR' ='Urine appearance'
             'SCIAPP' ='Urine appearance'
                .
                .
                .
```

In the original application each new name was treated as a separate test, even if it was simply another name for a test that has already been defined. Although the application does not have any problems processing data that use more than one 8-letter identifier for the same test, the application does fail to identify and report identical tests as being identical. The identification of identical tests is critical when using the data to compare across multiple drugs, studies, or even patients. In the past, if two studies are to be compared, a programmer would have to edit the structure of the data set, according to their best judgement as to which tests were or were not synonyms of one another. Comparisons between the conversion factors of similar units would require the programmer to examine the source code, and produce the report by hand.

To be able to address the problem of various names for the same test, we have implemented a "test-name thesaurus," (see figure 1.2) which contains simply the "preferred name" of the test, and a list of synonyms that have been used in the past to identify that test. This gives us the flexibility to allow for any field name the user chooses, while giving us a facility to easily report a complete list of assumed equivalent names. Any names which are new to the system need only be added as a synonym to a previous name, or added as a new name itself, although the latter parallels a number of other modifications that are appropriate to a new assay. This thesaurus data set is implemented with the same macro that is used for the unit thesaurus (see figure 1.3). The data set and the macro, as a pair, make up the "Variable Thesaurus" object, as described above.

Figure 1.2: Object Method, Variable Thesaurus Data set

labs.varthes

OBS	SYNONYM	NAME
20	APPEAR	UAPPEAR
128	LYMPHO	LYMPHO
131	LYMPHOS	LYMPHO
178	SCIAPP	UAPPEAR
210	UAPPEAR	UAPPEAR
216	UCHLOR	UCHLOR
217	UCHN	UCHLOR
218	UCOLOR	UCOLOR
.	.	.
.	.	.
.	.	.

Figure 1.3: Object Method, Thesaurus Macro

```
%macro prefer(data= last ,thes=,rename=,var=name,drop=yes);
%if &rename= %then %let rename=&var;

proc sort data=&data;
  by &var;

data &data
    %if %upcase(&rename)=%upcase(&var) %then (drop=synonym);
    %else (rename=(&var=&rename synonym=&var));
      error;
  merge &data(in=innorm rename=(&var=synonym))
        &thes(keep=&var synonym in=inthes);
  by synonym;
  if innorm or synonym='';
  if synonym='' then &var='.';
  else if not inthes then output  error;
  %if %upcase(&drop)=NO %then
    output &data;
  %else else output &data; ;

data  null ;
  set  error;
  by synonym;
  if first.synonym then do;
    %if %upcase(&drop)=NO %then
    put "Warning:  The &var " synonym 'is unknown.';
    %else put "Warning:  All variables using the unknown &var"
              synonym 'will be dropped.'; ;
  end;
run;
%mend;
```

Once a variable has been converted to its preferred name, it makes identification easier. In fact, the implementation of the thesaurus allows each assay name to "inherit" the properties of its preferred name. Identification of a test supplies us with information about the variable (see figure 1.4). This information includes the variable's type; character or numeric (VARTYPE), and the variable's domain, whether it is a categorical variable, or continuous variable (CATEG). It also supplies the test's preferred Conventional and SI units (CONVPREF and SI respectively), and finally, its descriptive label (LABEL).

Figure 1.4: Object Method, Variable Identification data set

labs.varid

OBS	NAME	CONV	SI	VARTYPE	CATEG	LABEL
9	ALBUM	g/dL	g/L	1	0	Albumin
10	ALHYDEH	U/L	U/L	.	0	Alpha Hydroxybut..
11	ALP	U/L	ukat/L	1	0	Alkaline Phospho..
12	ALT	U/L	ukat/L	1	0	ALT
13	AMM	umol/L	umol/L	1	0	Ammonia
14	ANA	+/-	+/-	1	1	Antinuclear Ant..
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.

As is the case with all of the data set objects we created, this data set interfaces with the application through a macro designed for that purpose (see figure 1.5).

Figure 1.5: Object Method, Variable Identification Macro

```
%macro varid(type=);

%if %upcase(%substr(%str(&type  ),1,1))=C %then %do;
  %let type=1;
  %let right=character;
  %let wrong=numeric;
%end;
%else %do;
  %let type=2;
  %let right=numeric;
  %let wrong=character;
%end;

data discvar contvar  nonorm;
  merge numvar(in=invars)
        labs.varid(in=inid);
  by name;
  if invars;
  if not inid then put '*** ' name;
  if vartype=&type and first.name then
    put "Warning:  The &wrong variable " name
        "was expected to be &right.." /
        '              The variable will be ignored.';
  else do;
    origval=value;
    origunit=unit;
    if categ then output discvar;
    else
      if ulim=. or llim=. then
        output  nonorm;
      else output contvar;
  end;

%numobs( nonorm);
%if &num obs > 0 %then %do;
  proc sort data= nonorm;
    by &key;

  proc freq data= nonorm;
    by &key;
    tables name/out= nonorm noprint;

  data  null ;
    set  nonorm;
      put "&key=" &key ':  has no normal range for the '
          'continuous variable ' name '.
          ' count 'records will be deleted.';
  run;
%end;
%mend;
```

Just as there are a number of names given to each assay, there are also a number of synonyms for each unit assigned to a particular test. For example, mmol/µL is synonymous in meaning to mol/mL and mmol/mm^3. For simplicity in conversion factors and clinical significance checking (shown below), we have also implemented a "unit thesaurus" (see figure 1.6) which will map any of an assortment of units to their "preferred" equivalent. In our implementation, we use SI units (Système International d'Unités) as a standard, but any convention is possible.

As we found with the synonyms in the variable names, the old version of the program makes no effort to distinguish between various names for the same units. The synonyms for units were only addressed at the point that the conversions were made (see lines 2-5 in figure 2.1, and lines 4 and 5 in Figure 3.1).

In the new version of the program, we also implemented a unit thesaurus which contained a list of the synonyms for any given unit that has been encountered in the past.

Figure 1.6: Object Method, Unit Thesaurus Data set

labs.unitthes

OBS	UNIT	SYNONYM
42	ng/Ls	ng/L-sec
44	ng/Ls	ng/L/s
45	ng/Ls	ng/Ls
116	1e3/uL	X 10E3/UL
118	1e3/uL	X10E3/MM
119	1e3/uL	X10E3/MM3
120	1e3/uL	X10E3/UL
129	1e3/uL	1e3/mm
130	1e3/uL	1e3/mm3
131	1e3/uL	1e3/uL
137	1e3/uL	1e9/L
140	1e3/uL	1E9/L
138	1e3/uL	1E3/mm3
125	1e3/uL	X10E9/L
.	.	.
.	.	.
.	.	.

The unit thesaurus data set is implemented into the application through the use of the %PREFER macro, the same macro that was used to implement the variable name thesaurus. Again, it is interesting to note that because the processes are largely identical, we are able to use exactly the same programming code to implement these two different processes. We would equate this with the concept of two different instances of the same object--same logical process, different data.

Conversion Factors

The ability to convert a lab test's value to a different set of units was one of the most complex and most difficult to maintain features of the original version of the Lab Macros. Each test had its own set of valid "original" units and "destination" units, with the appropriate conversion factor for each, and because these units were needed for various sections of the application, the conversion routines were often repeated in different locations throughout the code.

In our implementation, we separated the unit conversions into two groups: 1) those that depended on the actual test being converted, and 2) those that were independent of the test being converted. An example of the first type is converting mmol/mL to g/L, where the number of grams depends on the molecular weight of the substance being measured. An example of the second type is converting from mg/dL to g/L, for which a fixed conversion factor can be identified for any substance, namely 0.01.

These two conversion types are stored in two separate databases (see figures 2.2 and 2.3). Once the conversion information is stored in data sets, the programming that is required to perform the conversion from any given set of units to any other set is reduced to a few dozen lines (see figure 2.4). This is in sharp contrast to the 2,546 lines of code required for unit conversion in the old system (example 2.1 presents only the unit conversion for white blood cell count, and only to SI units).

In the interest of database integrity, the two conversion tables are actually generated from a master table of conversion factors in one direction. This master table is then separated into the two tables above, and each observation conversion factor is reversed, so that units can be converted in both directions (observations 8 and 9 in Figure 2.3 are generated by a single observation in the master table). Although the master conversion factor database is not an integral part of the lab application, we reference it to illustrate the level of control that can be gained by maintaining application information in data sets.

Figure 2.1: Old approach to Unit conversion

```
IF  NAME ='WBC' THEN DO;
  IF (UPCASE(UNIT)='X10E3/UL' | UPCASE(UNIT)='X10E3/MM' |
      UPCASE(UNIT)='K/CMM'    |
      UPCASE(UNIT)='X 10E3/UL'|
      UPCASE(UNIT)='X10E3/MM3') THEN DO;
        UNIT='X10E9/L';
        ULIM=ROUND(1*ULIM,.1 );LLIM=ROUND(1*LLIM,.1 );
        DO K=1 TO 32;
          IF  COL(K)>. THEN   COL(K)=ROUND(1* COL(K),.1 );
        END;
  END;
  ELSE IF (UPCASE(UNIT)='/MM3' |
           UPCASE(UNIT)='X10E6/L') THEN DO;
        UNIT='X10E9/L';
        ULIM=ROUND(0.001*ULIM,.1 );LLIM=ROUND(0.001*LLIM,.1 );
        DO K=1 TO 32;
          IF  COL(K)>. THEN   COL(K)=ROUND(0.001* COL(K),.1 );
        END;
  END;
  ELSE IF (UPCASE(UNIT)='/UL') THEN DO;
        UNIT='X10E9/L';
        ULIM=ROUND(0.001*ULIM,.1 );LLIM=ROUND(0.001*LLIM,.1 );
        DO K=1 TO 32;
          IF  COL(K)>. THEN   COL(K)=ROUND(0.001* COL(K),.1 );
        END;
  END;
  IF UNIT^='X10E9/L' THEN DO;
        ERRMSG1=1;
        LINK ERRMSG;
  END;
END;

IF  NAME ='FE' THEN DO;
      .
      .
      .
```

Figure 2.2: Object Method, Parameter-Dependent Conversions

labs.depconv

OBS	NAME	FACTOR	FROM	TO
1	FSH	1.00	.	U/L
2	MCHC	0.62	%	mmol/L
3	MCH	0.02	amol	pg
4	MCH	15.87	fmol	pg
5	UCREAT	8.84	g/day	mmol/day
6	ALBUM	1.00	g/dL	g%
7	HEMO	0.10	g/dL	mg%
8	HEMO	0.62	g/dL	mmol/L
9	MCHC	0.62	g/dL	mmol/L
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Figure 2.3: Object Method,Parameter-Independent Conversions

labs.indconv

OBS	FACTOR	FROM	TO
.	.	.	.
.	.	.	.
.	.	.	.
5	0.010	%	.
6	1.000	%	g/dL
7	10.000	%	g/L
8	0.001	amol	fmol
9	1000.000	fmol	amol
10	1.000	fL	L
11	1.000	g/dL	%
12	10.000	g/dL	g/L
13	1.000	g/dL	g%
14	1000.000	g/dL	mg/dL
15	1000.000	g/dL	mg%
16	1000.000	g/mL	g/L
.	.	.	.
.	.	.	.
.	.	.	.

The conversion factor database is where the object-oriented code really begins to demonstrate its effectiveness. Whereas the original application had various conversion factors repeated numerous times across many sections of the application, and dozens of separate assays, the new application records each conversion factor only once, in the CONVERT data set. The generated data sets INDCONV and DEPCONV are implemented through the %CONVERT macro. Any time a user needs to convert from one set of units to another, even outside the Lab application, they may make use of the CONVERT object that the data sets and the macro create.

The advantages to this implementation are extensive. Any corrections to a particular factor can be made in one place, and will automatically be propagated throughout the application. Any additions to the conversion factor database can be made in the data sets, requiring no modification of the application code, which, more importantly, keeps the application from having to be re-validated. Furthermore, for those conversion factors that are independent of the tests that they convert, the application has a certain "learning" ability regarding these conversion factors; i.e., once the application has been trained to convert between two units on one test, it will be able to apply the same rule to any other test that requires the same conversion.

Figure 2.4: Object Method, Unit Conversion macro

```
%MACRO CONVERT(data=,to=SI,convert=value,usethes=yes);

proc sort data=&data;
  by unit &to name;

data &data  indconv(drop=factor);
  merge &data(in=indata)
        labs.depconv(in=inconv rename=(from=unit to=&to));
  by unit &to name;
  if indata;
  if not inconv then output  indconv;
  else do;
    %do i=1 %to %wordcnt(&convert);
      %scan(&convert,&i)=%scan(&convert,&i)*factor;
    %end;
    unit=&to;
    output &data;
  end;
run;

%let  eindcnv=0;
data  indconv  eindcnv;
  merge  indconv(in=indata)
         labs.indconv(in=inconv rename=(from=unit to=&to));
  by unit &to;
  if indata;
  if unit ne &to then do;/*conv. necessary (different units)*/
    if not inconv then do; /*no conversion is defined, error*/
      call symput(' eindcnv','1');
      output  eindcnv;
    end;
    else do;                    /* conversion is defined, use it */
      %do i=1 %to %wordcnt(&convert);
        %scan(&convert,&i)=%scan(&convert,&i)*factor;
      %end;
      unit=&to;
      output  indconv;
    end;
  end;
  else output  indconv; /*no conversion necessary, same unit*/
run;

%if & eindcnv %then %do;
proc freq data= eindcnv;
tables unit*&to*name /noprint out= eindcn2;

data  null ;
  set  eindcn2 end=eof;
   put 'Error:  The unidentified conversion from ' unit 'to '
       &to 'for the variable ' name 'was found on' count
       ' records.';
    if eof then put '        The tests will be ignored.';
run;
%end;

data &data;
  set &data(drop=factor)  indconv(drop=factor);
run;

%MEND;
```

CLINICAL SIGNIFICANCE GUIDELINES

At Parke-Davis, as is probably the case at many pharmaceutical research organizations, the study authors, reviewers, and medical writers have a list of assay results that are of particular interest. These conditions are identified as being "clinically significant." Examples of clinical significance guidelines include: "any value of creatinine clearance that represents a 20% increase over baseline," or "any value of urine sodium which is 30% below normal values."

In the original system, this task was achieved by a series of conditional DO; END; blocks. These blocks included any unit conversions that had been necessary in the past, although these unit conversions are done independently of the unit conversion code that occurs earlier in the application (discussed in section 2).

Figure 3.1: Old Approach to Testing For Clinical Significance

```
.
.
.
IF  NAME ='GLUC   ' THEN
  DO;
    SEL=1;
    IF ( COL(K)>140*(.05551*(UPCASE(UNIT)='MMOL/L')
        +(UPCASE(UNIT)='MG/DL'))
        & .<INITIAL< COL(K)) THEN  PCOL(K)='++ ';
    IF ( COL(K)<60*(.05551*(UPCASE(UNIT)='MMOL/L')+
       (UPCASE(UNIT)='MG/DL'))) THEN  PCOL(K)='-- ';
    incdec=0;/*interest in significant increase and decrease*/
  END;
IF  NAME ='UA     ' THEN
  DO;
    SEL=1;
      IF ( COL(K)-INITIAL>3*(59.48*(UPCASE(UNIT)='UMOL/L' |
         UPCASE(UNIT)='MCMOL/L')+(UPCASE(UNIT)='MG/DL'))
         & ULIM< COL(K)) THEN  PCOL(K)='++ ';
      incdec=1; /* interest in significant increase */
  END;
IF  NAME ='TPROT  ' THEN
.
.
.
```

Putting the tests for clinical significance into a separate data set proved one of the more ambiguous cases we faced in this project. While we could certainly put all the values, operators, comparisons, and variables into variables in a table which, in turn, could be assembled into conditional expressions by the macro generator, we would not gain much, if anything, in simplicity. There was, however, enough redundancy to try to move some of the evaluation expressions into data sets. The compromise we came to was to develop a set of conditional expressions, which we could refer to by number (the TYPE variable). These expressions would be implemented in the macro code, while the actual numeric comparison values would be stored in a data set (figure 3.2).

Figure 3.2: Object Method, Clinical Significance Test Data Set

labs.clinimp

OBS	NAME	TYPE	P1	P2	P3	INTEREST
1	AEOS	8	0.05	0.5	.	1
2	AK	5	40.00	4.0	16	0
3	ALBUM	4	-1.00	.	.	-1
4	ALKPH	1	1.25	.	.	1
5	ALT	1	3.00	.	.	1
6	AST	1	3.00	.	.	1
7	BILI	1	1.25	.	.	1
8	BUN	1	1.25	.	.	1
9	CA	2	2.00	.	.	0
10	CL	2	5.00	.	.	0
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.

In our implementation, we simplified the clinical significance guidelines into nine "conditional functions" (see SELECT statement at the bottom half of figure 3.3). Each of these functions may employ between one and three bounds of comparison (P1-P3). It is these bounding values, as well as the assay itself, which we store in the database, with one record representing one clinical significance check.

The improvement that this approach holds over the original program is largely a result of the unit conversion and thesaurus objects. All unit conversions are done by this single object, and use the standard table we developed. Therefore, any changes and enhancements made to the conversions there will be echoed in the clinical significance tests.

Unlike the implementations of the other objects above, the implementation of clinical significance tests does not exclude modification of the source code. A situation may arise where none of the nine types of comparisons currently defined are appropriate for a newly created definition of clinically significant. Fortunately, this is of marginal concern, as clinical significance definitions rarely change, and, of the 30 tests currently defined, type 1 and type 2 (%increase and %change respectively) combine to address 66% of the definitions.

Figure 3.3: Object Approach, Clinical Significance Calculation

```
%macro clinimp(data=labdat,value=value,
               key=ci prot trial ptno,
               datetime=labdate);

proc sort data=&data;
  by name;

*-------------------------------------------------------------*
  Clinical significance data set must be added in advance,
  because baseline values and wbc values must be retained
  obtained using a different by statement;
*-------------------------------------------------------------*;
data &data;
  merge &data(in=indata)
        labs.clinimp(in=inclin);
  by name;
  if indata;

proc sort data=&data;
  by &key &datetime;

data wbc(drop=name rename=(value=wbc));
  set &data(keep=&key &datetime value name);
  if name='WBC';

%numobs(wbc);
%if &num obs > 0 %then %do;
  data &data;
    merge &data wbc;
    by &key &datetime;
%end;
%else %put  ERROR: No WBC observations were found. Clinical
      significance  tests may be inaccurate for EOS and AEOS;

proc sort data=&data;
  by &key name &datetime;

data &data(drop=wbc initial type p1-p3);
  set &data;
  by &key name;
  retain initial;
  if first.name then do;
    initial=value;
  end;
  if type ne . then select (type);
    when (1)
      clinimp=(&value > p1*max(initial,ulim));
    when (2)
      clinimp=sign(&value-initial) *
              (abs(&value-initial) > p1 and
              not (llim < &value < ulim));
    when (3)
      clinimp=((&value-initial) > p1 and &value > ulim);
    when (4)
      clinimp=-1*(. < (&value-initial) < p1);
    when (5)
      clinimp=sign(&value-initial) *
             (abs((&value/initial)-1) > p1 and
              not (llim < &value < ulim));
    when (6)
      clinimp=sign(&value-initial) *
             ((initial > p1 and
               1/p2 < &value/initial < p2) or
              (initial < p1 and
               &value/initial > p3));
    when (7)
      clinimp=(&value > p1 and &value > ulim) -
              (&value < p2);
    when (8)
      clinimp=(&value/wbc > p1 and
              &value > p2);
    when (9)
      clinimp=(&value > p1 and &value*wbc > p2);
    otherwise put 'Warning: No clinical significance logical '
                  'expression found for type ' type;
  end; /*select*/
  else clinimp=0;
%mend;
```

CONCLUSION:

Because the environment in which our applications are to survive is constantly changing, we must develop applications that can adapt to these changes. Although it is unrealistic to try to anticipate every possible event, we can develop our applications in a way that minimizes the amount of effort it takes to "update" the application. In the application environment, one approach is to remove all of the theoretically changeable values, and store them in data sets. Wherever the programmer codes a fixed value, either character or numeric, it should be examined with suspicion: "Is this value a universal constant, unchanging across the depths of space and time," or, more likely, is it a value that might be more appropriately stored in a database to be easily reported, documented, updated, and tracked.

The extraction of this information from the application, storing it in data sets, and creating the macros that would allow

these data sets to interact with the application, created logical objects. These objects had many of the properties that an object in an object-oriented programming language may have. Although it may not have some of the more technical features of a true object, we do find the spirit of object-oriented programming in each of the modules. Features such as re-usability and data integrity are clearly visible in our objects.

Besides the features of the objects themselves, the implementation of the object approach greatly simplified each task involved in the processing of clinical laboratory data. In every case, the number of lines of programming dropped considerably when implementing the object approach (see Figure 4). Furthermore, in all but one case, the amount of code and data in the objects combined was less than number of lines of source code that was required by the old system to accomplish the same task. In fact, the objects developed for the new application are generally equipped with greater functionality than their non-object counterparts, despite their smaller size. Although the number of lines of code is not an accurate measure of the quality of an application, an average code reduction of 94% goes a long way toward the goal of "keeping it simple."

By analogy, one could think of an application as a data-processing machine, and these objects seen as the controls of this "machine." These controls can modify the operation of the application by changing or adding data to the data sets that make up these controls. By taking the time to build these controls into the application, we can make common modifications to the machine without having to re-build it.

BIBLIOGRAPHY:

References:

The Tao of Objects: a beginner's guide to object-oriented programming Gary Entsminger, 1990.

Suggested Reading for Programming with Data sets:

Code-free Data Validation and Verification Samuel Berestizhevsky, Tanya Kolosova, Proceedings of the 19th Annual SAS User's Group International Conference, pg 18-21.

A Development Environment for Applications Based on SAS® Software Staff of SAS Consulting Services Inc. Proceedings of the 19th Annual SAS User's Group International Conference, pg 1347.

Suggested Reading for Managing Laboratory Data:

Managing Clinical Lab Data with SAS/AF Software--An Online Demonstration Eric Brinsfield, Brad Klenz, Proceedings of the 19th Annual SAS User's Group International Conference, pg 514-7.

Contacting the Authors:

Paul Wehr
2502 Traver
Ann Arbor, MI 48015-1249
wehrp@aa.wl.com
wally@eworld.com

Albert Mo
Parke-Davis Research
2800 Plymouth Road
Ann Arbor, MI 48105
moa@aa.wl.com

Figure 4: Comparison of complexity of Old System vs. Object-Oriented Approach