Coding for Posterity

Rick Aster

Some programs will still be in use years after they are originally written; others will be discarded. It is not how well a program achieves its original purpose that makes this difference. A program may work perfectly, but if it is not easy to maintain — if it cannot be modified, usually by different programmers, to meet changing purposes and requirements — it will likely be replaced by a different program as soon as circumstances change.

If you want the SAS programs you write to stand the test of time, you need to build maintainability into them. The way you write SAS code and the descriptive comments that accompany it can help a maintenance programmer read and understand it and make any changes that new circumstances require.

Maintenance

Equipment of any kind may require a maintenance effort from time to time to stay operational and useful, and computer programs are no exception. Revisions to programs add new capabilities and flexibility, correct design errors, adjust to changed interfaces with other programs, port the program to another computer or operating system, improve performance, and combine several programs into a coordinated system. It is safe to assume that any program designed to be run regularly or repeatedly will eventually be considered for revision.

Depending on how maintainable the original program is, such revisions may be made easily; accomplished with some difficulty; completed, but with errors; attempted unsuccessfully; or not attempted at all, as the program is discarded and a replacement program is written.

The same concerns can apply to a program that is intended to be run only once for a special purpose. There are three ways this can happen. It occasionally happens that such a program ends up being run regularly. This may be because the situation for which the program was written, which at the time seemed a unique circumstance, turns out to recur on a regular basis. Or it may occur when a program developed hastily as a quick fix is added to a production system. Alternatively, a program may be used as a model for a new program needed to handle a similar circumstance that arises later. As one common example, a program that projects the effects of a planned merger between two companies may be used later as the starting point for a new program to project the effects of a merger between the resulting merged company and yet another company. Finally, it may be discovered some time after a program is written and run that it did not fully meet its intended purpose. Then, the program must be revised and run again.

In any of these cases, the ease of working with the original program depends on its being written in a way that would make it easy to maintain. It happens so often that "one-time" programs are reused in one way or another that many programmers find it advantageous to write virtually every program with the assumption that it may be used again later.

Yourself or Another Programmer

Writing a program for easy maintenance is the same concern regardless of whether you or another programmer will later maintain it. The same strategies you use to make it easy for you to find, remember, and revise your own program also make it easier for a subsequent programmer to discover, understand, and revise the program.

Guesswork and Reverse Engineering: the Sad Reality of Maintenance Programming

The ideal case for maintenance programming occurs when a programmer has compared new business requirements with an existing program to reach the conclusion that the requirements can be met more easily by modifying the existing program than by developing an entirely new program. The unfortunate reality of most maintenance programming is just the opposite. The business requirements may be largely unknown and not formally stated, except that it is known that most of the functionality of the existing program must be maintained. The programmer is forced to work with the existing program just to learn (or at least not to upset) the business and technical requirements of the task. Effectively, the programmer must reverse-engineer the existing program. Reverse engineering is difficult by nature, so a maintenance programmer can easily expend a greater effort to complete minor modifications to a program than the effort the original programmer took to develop the entire program in the first place.

This sort of maintenance programming is more often done by guesswork than by a complete reverse engineering of the existing program. The maintenance programmer studies the existing program long enough to guess what changes are required, makes those changes, and attempts to run the program with the changes. If the changed program appears to run correctly, the programmer considers the changes to have been made. Otherwise, the programmer makes another guess and repeats the process. The potential for error in this programming methodology is enormous. It is likely that the majority of errors in business software are introduced in this fashion.

Both factors — the disproportionate effort required to do maintenance programming and the potential for error in

the process — argue for programmers to write programs that are more maintainable.

Low-Maintenance Design

The single overriding principle that makes any equipment maintainable is design. The strategies that make a low-maintenance design for a computer program are not so different from those that make a low-maintenance car or computer.

Most of the principles that make for low-maintenance design are intuitive, if not self-evident, when you take on the perspective of the person doing the maintenance work. For most programmers, this is not a difficult perspective to call up — most programmers' work every year is divided between new programming and maintenance.

If you never done maintenance programming, your experience in maintaining anything else may be helpful. Whenever you have to fix a program or any device that's broken, these are among the qualities you hope it has:

• *Components.* It helps when a device neatly disassembles into pieces: components, and perhaps combinations of components that form subsystems. The advantage is that you only need to repair or replace the broken parts, without having to consider all the details of the other parts. It helps even more if the components and the interfaces that join them are standardized, so that you can simply drop in a new standard component to replace the one that no longer works.

For a program, you hope that the functionally separate sections of the program are also logically separated so that you can modify or replace one section without affecting the others. In SAS programs, it helps that the interface between two sections is often a SAS dataset, a standardized file format with contents that you can easily study and understand. This makes it possible to revise one or a few steps in a SAS program while being assured that the revisions do not have some unexpected effect on other steps.

• *An understandable design.* When you set out to repair or modify something, you hope to discover and understand its design. The design lets you know what the components and interfaces are supposed to be. This information first helps you identify components that you want to change. The design also tells you how the components you add or revise have to be to fit in. A SAS program's design is usually explained in part in comments contained in the program. Other details of the design may be found by reading the code and any separate data flow diagrams, record layouts, report descriptions, and other design documents.

It is impractical to completely specify the design of a computer program with diagrams, tables, and descriptive text, so you have to find many details of the design in the code. That means that the readability of the program is an important factor in your ability to understand its design.

• *Ease of modification.* A design can facilitate some repair and maintenance. It can even state exactly how to make certain modifications. As a tangible example, it is easy to add memory to most computers, because they are built with connectors specifically designed for standard memory components. Similarly, a computer program can contain "hooks," designed interfaces for making certain anticipated modifications.

Priorities and Principles

Maintainability is not a high priority in the management of most new computer programming. Often the only operant concern is that a program run correctly before the stated deadline, or as soon as possible thereafter. Programmers in this context cannot spare any significant amount of time to make the programs they write more maintainable. For all practical purposes, this means that maintainability has to come from the coding process itself.

Fortunately, when you understand the principles of maintainability, it takes no more time to write SAS code with good maintainability than it does to write unmaintainable code. The most important principles can be organized around four qualities of a program: purpose, context, scope, and style. Objectives include making it easier for the maintenance programmer to read and understand the program and coding in such a way that fewer code modifications are necessary.

✪ Purpose

The first thing a maintenance programmer needs to know when considering a program is its purpose: what it is, what it is intended to do, what output it produces from what input, what requirements it is intended to fulfill. With an understanding of the purpose of the program, a maintenance programmer can consider its potential uses. But a maintenance programmer cannot be expected to spend more than a few seconds studying an unknown program to try to ascertain its purpose.

It is easy enough to make a short statement of the purpose of a program in a comment at the beginning of the program file. After you develop this habit, you'll wonder how you ever wrote a program without first stating what the program was supposed to do. If it is difficult to state the purpose of a program you want to write, either you don't really understand what the program is supposed to do, or you have what should be two or more programs run together. To maximize maintainability, identify each independent purpose and develop it as a separate program.

You should be able to state the primary purpose in no more than three sentences. This is an example of a statement of the purpose of a program: "This program combines a customer's mailing database records after it has been discovered that the same customer has more than one record."

The same ideas of purpose may apply to the sections of a program that may be of interest to a maintenance programmer. The relevant sections of a SAS program are

usually steps, but it would be a mistake to always consider each step separately. A sort-merge process, for example, often spans three steps: two PROC SORT steps followed by statements in a data step. When it takes a combination of steps to carry out a single task, putting these steps consecutively in the program and describing what they do collectively helps the reader understand how the program fits together.

The following example shows how a program section might be described with a comment:

```
*
  Merge history with inquiries, matching on
  subject and date.
*;
proc sort data=main.history;
   by subject date;
run;
proc sort data=main.inquiry;
   by subject date;
run;
data research;
   merge main.history (in=in1)
       main.inquiry (in=in2);
   by subject date;
   if in2;
. . .
```

The purpose of an individual step is often obvious enough from the code, but only if the reader can pick out the separate steps easily. It is a good practice to put a RUN statement on a separate line at the end of each step, except for those few steps (such as a PROC SQL step) where RUN statements are not appropriate. RUN statements make a clear visual delineation between steps. Even programmers who are not familiar with the SAS language can see the significance of RUN statements and the steps they mark off.

When you write extensive data step code to implement a particular algorithm, a maintenance programmer reading the program needs to know what statements are involved and what they are supposed to do. You can get this point across with a brief description at the beginning of the task and a blank line at the end.

The operation of a step or section of code may be obvious enough from the code itself, but you may still need a descriptive comment if the effects of the step are not so obvious. When you read the following example, the comment helps you recognize that the SORT proc is being used to create a list.

```
*
  Make a list of the states that have had
  quakes of 3.2 or greater in 1998 so far.
*;
proc sort data=main.seismic
     (where=(year=1998 and scale >= 3.2)
     keep=state year scale)
     out=list (keep=state compress=no
     index=(state)) nodupkey;
   by state;
run;
```

On occasion it may be appropriate to describe the purpose of an individual statement. Use comments to clarify the intended effects when a statement is being used for something different from the usual. One such situation is the use of an assignment statement with automatic truncation to assign the beginning of a character value to a shorter character variable:

```
length region $ 2;
* Region code is first two digits of
  customer code.;
region = customer;
```

Without the comment, someone reading the program would be likely to get the mistaken impression that the entire value of the customer code had been assigned or was intended to be assigned to the REGION variable.

Code With Purpose

The way you write program statements can also help clarify their purpose. These are examples of code strategies that can make the code itself more clear of purpose:

• *SELECT blocks.* Often a section of code selects one of a set of actions in each case, depending on a set of conditions. To make this logical structure clear at a glance, use a SELECT block to code it.

• *Discarding observations.* Sometimes it is more clear to discard observations with a DELETE statement. In other situations, a subsetting IF statement is better. Ask yourself how you would describe the selection process in ordinary words — is the objective to keep one set of observations or to discard the other set? Then code the selection logic in the way that follows your description.

• *Index ranges.* Use meaningful values for array subscripts if you can. This code:

```
array att{5};
do i = 1 to 5;
   year = i + 1998;
   put year att{i};
   end;
```

is easier to understand when rewritten this way:

```
array att{1999:2003};
do year = 1999 to 2003;
   put year att{year};
   end;
```

• *Titles.* If TITLE statements help explain what a program does, put them near the beginning of the program so they'll be easy to find. Similarly, put TITLE statements for a step at the beginning of the step.

• *Parentheses.* If parentheses would clarify the order of evaluation of an expression, use them, even if the syntax rules don't require them.

• *Variable list.* A variable list, as in the VAR statement or the KEEP= dataset option in the SET statement, can help clarify what variable or variables from a SAS dataset the program is actually using.

Purpose in Macro Programming

Purpose of code becomes three times as complicated in a program based on macros. The reader needs to know the purpose of each macro, the purpose of code generated

each time the macro is called, and the connection between the two. This is hard to keep straight and even harder to explain clearly if you are not an expert macro programmer.

For maintainability, the best approach is to reserve macro programming for those situations that really require it. The time saving that macros may offer when you originally develop a program can be more than offset by the additional effort required to maintain the program and the macros it uses.

Another factor to consider is that macros are not universally accepted. Few maintenance programmers are expert macro programmers, and many will not consider maintaining a program that relies on macros if they have the option of writing a new program.

When you must code macros, take care to comment them correctly. Use macro comment statements, which do not become part of the generated code, to describe the macro code. Use regular comments to describe the code that the macro generates. The macro will then generate code that has appropriate comments.

✪ Context

A maintenance programmer who can put a program in context is more likely to understand the program's significance. At the most basic level, context means the time and place at which the program originated, who wrote it, and what project or system it was created for. In some projects, the SAS software version or products they rely on may be important. If it is not obvious what needs to be done to run the program, state that. Describe any relevant contextual information in a comment at the beginning of the program file.

Systematic storage of programs is another way to communicate context. If all the programs of a project are stored together, and separate from other, unrelated projects, the reader has a much easier time finding the programs that matter and understanding how they fit together.

If a substantial or interesting part of a program is copied from or based on another program, state that. Cite the paper or book that you took an unusual code fragment from. If there is anything unusual or special about the circumstances for which a program was developed, state that. Ideally, the reader can get a sense of a story of how the program was written and what it was used for.

Read the following comment from the beginning of a program file and consider how much it would help you read the program itself and understand its significance.

```
*
  ROBOTS5.SAS
  Robot System Customer Reports
  Inactive - Illinois - Indiana
  Rick Aster  August 27, 1998

  This program is an altered version of
  ROBOTS.SAS. It produces the same reports,
  but includes only inactive customers
  located in Illinois and Indiana. We are
  running thess reports to help Adam Selinem
  fill in gaps in his itinerary.
*;
```

When you can, use general terms to describe the purpose of a program or a section of code. Don't make the description more specific than the code itself is. As an example, a program you developed to prepare data for a specific report might also prepare the data for any other reporting or analysis that might be required. If so, the program is more accurately described as a data-cleaning process than as merely the preliminary work of one specific report.

When you describe the context of a program, you often need to refer to other programs. Usually you can refer to programs by file names. If not, give each program a short name and state that name in a comment at the very beginning of a program file.

Contextual information is especially important when a program is not complete in one program file. List all files required for the complete program in the main or first program file. Refer to that file in each secondary or subsequent program file.

✪ Scope

No program can do everything that its purpose and context may suggest. Its scope is limited by dependencies and design limitations. When limitations are clearly stated or shown, a maintenance programmer can more accurately determine the applicability of a program to a new situation and identify aspects of the program that may need to be changed.

When specific kinds of changes to a program are likely to occur, you may be able to code it or mark it in a way that makes it easier to change.

Dependencies

These are examples of the kinds of dependencies you may want to note about a program:

- Record layouts of input files
- Variable names and types and other required characteristics of input SAS datasets
- Limitations on the values of input data
- Memory and storage requirements
- Informats or formats stored in a format library
- Macros, macro variables, or system options that the program relies on, but doesn't itself set
- The program's place in a sequence of programs or actions
- Use of operating system or file system features or specific file or directory names
- Filerefs, librefs, or symbols that the program expects to have been defined externally, perhaps with operating system commands

Limitations

The limitations of a program's design may be hard to notice when they are based on conditions that you take for granted. The currently notorious case is the century limitation of programs that make date calculations based on two-digit year numbers. This was a design compromise

in the development of some programs, but represented nothing more than a thoughtless habit in other cases.

Consider a routine that generates random 5-digit customer ID codes. That sounds reasonable enough, but it would fail to generate unique codes if the number of customers increased beyond 100,000. When you identify such a limitation, the way to handle it depends on how significant you think the limitation is. If the limit is likely to be exceeded, redesign the program with a higher limit. If reaching the limit is a significant possibility, design a way to change the program to go beyond the limit. If there is only a slight possibility of reaching the limit, make a note of it as a comment in the affected section of the program. The comment for the 5-digit ID code generator could simply say, "Limited to 100,000 customers." Consider writing an error routine so that the program fails gracefully if the limit is reached.

The length of every variable and the dimension of every array represents some kind of limit, and it is sometimes hard to guess when you write the program how much space a data element requires. In that situation, consider using the program parameter strategy, described below, to set the size.

Displaying Dependencies

These are specific coding strategies that make a program's dependencies more obvious:

• *References to external files.* Declare libraries and files in LIBNAME and FILENAME statements. Collect these statements together near the beginning of the program. Use descriptive names or comments to describe each file or library.

• *Options.* Use an OPTIONS statement at the beginning of the program to explicitly set any required system options.

• *Declarations.* Use a LENGTH statement to declare and set the length of a character variable.

Flexibility

You can design code that has flexibility, to handle situations beyond the specific situation the code is developed for. These are three different degrees of code flexibility:

• Code that doesn't require a change to handle a different situation

• Code that requires only a minimal change to handle a different situation

• Comments that describe how to alter code to handle a different situation

When you anticipate contingencies that could lead to changes in a program, consider how the initial code could connect to code added later. After you determine where likely enhancements to the program would fit into its structure, use a comment that says, "Add additional data-scrubbing logic here," or something similar. Describe the added code in as much detail as necessary. Show commented-out code if it help to demonstrate the form of potentially added code.

SAS Software Features for Generality

SAS software offers numerous features that let you code without spelling out every detail. For example, you can omit the DATA= option from a PROC statement, and the proc will use the most recently created SAS dataset. This default lets you write proc steps that can work on SAS datasets that may have different names. The special abbreviated variable list _NUMERIC_ lets you select all numeric variables in a SAS dataset (or those defined so far in a data step). You can process a whole set of numeric variables without having to know what their names are or even how many there are. Such features make it possible to write code that has generality — code that doesn't need to change with every change in the details of its input data.

Strategies for Flexibility

These are examples of coding strategies that can produce flexible SAS code.

• *Program parameters.* When you expect a name or value that appears in the program to change, make it a macro variable and assign the value to the macro variable in a %LET statement at the beginning of the program.

• *Read data values from files.* Hard-coding is writing data values into program code. The only way hard-coded values can change is by changing the program. Avoid hard-coding values that you expect will change frequently. Instead, have the program retrieve the values from a file.

• *Arrays.* When an array might change in size, use the DIM function as the iteration limit of a loop over the array:

```
do i = 1 to dim(array);
   sum + array{i};
   end;
```

This way, if the size of the array changes, you will only need to change the ARRAY statement. For even more generality, use the LBOUND and HBOUND functions.

• *Table output.* When you can, use procs such as REPORT to format tables. When you must use a data step to produce table output, use variables for the column locations. Set them in a RETAIN statement, like this:

```
retain c1 2 c2 8 c3 21;
```

Then, use these variables with the @ pointer control in PUT statements:

```
put @c1 code $char4. @c2 date yymmdd8.
    @c3 desc $char36;
```

With this approach, you can adjust the positions of the columns on the page just by changing the RETAIN statement.

• *References to external files.* Declare libraries and files in LIBNAME and FILENAME statements at the beginning of the program so that it is easy to modify the program to work with different files or file names.

• *Table-driven logic.* When a great deal of flexibility is needed, consider a table-driven development strategy, in which program actions are determined by values stored in SAS datasets.

• *Portability.* When possible, write programs using only portable SAS language features, avoiding those features that are available only in specific operating systems.

Efficiency vs. Flexibility

Sometimes, when writing a program, you need to decide whether it is more important to make it run efficiently or to write it in a way that can be easily modified. For example, a program might run faster if you calculate statistics in a data step, but be easier to modify (and probably easier to code in the first place) if the statistics are calculated in a subsequent PROC SUMMARY step.

Usually, your decision should to be to write maintainable code rather than efficient code. The cost of computer resources to run most SAS programs is minor, even trivial, compared to the cost of programmers' efforts to develop and maintain the programs. However, there are always exceptions, and in the case of interactive systems, the convenience of the programmer must give way to the convenience of the user.

When you must write more cumbersome code for the sake of efficiency, there are still things you can do make the code easier to maintain. Make an extra effort to identify the functionally separate aspects of the code, even as they overlap. Use comments to describe the segments of code that are essential for efficiency so that a maintenance programmer does not inadvertently compromise efficiency when modifying the program.

✪ Style

Your coding style is the way you use elements of the programming language that have no effect on the execution of the program. SAS style makes use of things like indentation, blank lines, spacing, the selection of variable names used within the program, and the use of optional statements and terms.

Neatness is another way of thinking of style. When you apply a style to a program, the program appears more orderly and tidy than when it is written without style. When a program looks neat, errors in it may jump out at you. Conversely, a sloppy program can hide errors.

A good style is simple, clear, and consistent. Many specific suggestions have been made about SAS coding style. The main point, however, is that just about any style that you follow consistently is better than no style. Even a bizarrely idiosyncratic coding style can eventually start to make sense to the reader. But if your code has no style, a maintenance programmer can never quite figure it out without reading every detail of the program.

A consistent style can also reinforce a maintenance programmer's confidence in the quality of your code. As in any kind of work, neatness of presentation gives the impression that some care was taken in the work itself, which may create the presumption that the product of the work is reliable and should be taken seriously.

Coding as Writing

Coding for maintainability makes sense if you think of coding as a kind of writing. It's true that writing a SAS program is different from writing prose or poetry. SAS syntax has requirements all its own that you must follow to have a successful SAS program. At the same time, though, a well-written SAS program also captures its message in a way that at least some people can read. And as with prose or poetry, the better the writing is, the more compelling a case it makes to the person reading it.

It may help to visualize a SAS programmer who, at some point in the future, continues the work that you have started on the program you are presently writing. When you write code with your reader in mind, the value of the program you write is more obvious, and you make it more likely that its value will remain apparent for some time to come.

Contact:
Rick Aster
Breakfast Communications Corporation
P.O. Box 176
Paoli, PA 19301-0176
finis@pond.com
http://www.pond.com/~finis/sas