Good Programming Practices in SAS  
Jason Ford, Bureau of Labor Statistics

Introduction

Why is it so difficult to write code that does exactly what it is intended to do? Even programmers with knowledge of the most complex SAS® functions struggle with writing code that is free of errors.

One good answer for the difficulty of programming comes from Frederick Brooks, a noted software engineer and author of the essay “No Silver Bullet: Essence and Accidents of Software Engineering”. This essay can be found at http://www.lips.utexas.edu/ee382c-15005/Readings/Readings1/05-Broo87.pdf. In this essay, Brooks discusses the complexity of most computer programs. This complexity makes programs difficult to design and test.

Brooks gives a number of reasons that computer programs tend to be complex and thus difficult to code correctly. These reasons include:

- Each subroutine in a program is going to be different, at least above the statement level. (If the subroutines were the same, the programmer would write a macro.) The result is that most programs have a large number of states that need to be enumerated and tested.

- Programs have to conform to the existing systems and rules that have been set up by organizations. These systems and rules can often be complicated.

- Programs change over time as people find new uses for the code and new systems replace old systems. These changes increase the complexity of code.

- Because a program is filled with abstract logic, it is difficult to visualize what the program is doing. If we attempt to do a flow chart on most programs, we find that the logic is too complex to visually lay out.

Furthermore, Brooks argues that for most programs, it is impossible to write correct requirements without first seeing some output. The implication of Brooks’ argument is that in the initial stages of development, nobody has a perfect understanding of what the program is supposed to do. During the process of developing the program, the customers, designers, programmers, and testers need to arrive at a shared understanding of what the program is supposed to do. Success is dependent on many people with varying types of expertise communicating with each other.
Programmers need to follow practices that make communication of complex concepts as easy as possible. Three key practices for programmers are:

1. Understand the subject matter behind the program.
2. Write the code in a way that other programmers can read it easily.
3. Write the code in a way that it can be updated easily.

**Domain knowledge: understanding the subject matter of what you are coding**

“Domain knowledge” is knowledge about the subjects related to the problem that the programming is addressing. If a program compiling statistics on children’s health care, for example, domain knowledge would be knowledge on the subject of children’s health care. Programmers should try to gain as much domain knowledge as possible so as to insure effective communication with the customers, designers, testers, and other programmers.

Domain knowledge can help in every phase of the project. During the design phase, for example, a programmer who has domain knowledge is more likely to interpret requirements correctly. During user acceptance testing, domain knowledge allows errors to be diagnosed and corrected quickly. The programmer and tester can understand each other when discussing the nature of errors and possible fixes.

One way to consider the matter is the difference between someone receiving directions in their home town versus receiving directions in a town they had never before visited. A person receiving directions in their hometown would be much more likely to catch any minor mistake, such as mistaking “Park Street” for “Park Road.”

Likewise, the programmer with domain knowledge will be more likely to understand the gist of the requirements. If a requirements writer accidentally lists a wrong variable in an equation, for example, the programmer with domain knowledge will be likely to catch the error before starting programming. Fixing errors at that stage is cheap and easy. If the programmer does not have domain knowledge, such an error might not get caught until testing or even deployment. The error thus becomes more costly.

**Formatting the code**

To insure accuracy, multiple programmers should review all code. Programmers should therefore write code in ways that makes review as easy to read as possible. Good formatting of the code is essential.

The following are some basic tips on good formatting:
1. Putting multiple statements on one line hurts readability, as demonstrated by the following example:

    Data Urate_ny; Set Urate_US; if state='NY'; Run;

A good practice is to put each statement on a different line:

    Data Urate_ny;
    Set Urate_US;
    If State='NY';
    Run;

2. When having a list of variables, put each variable on its own line:

    Proc sql;
    CREATE   table health_plan_choices as

        SELECT Company,
               Job,
               Health_plan
        FROM     library.occ_source
        WHERE    quarter_begin <= &Mquarter and
                  quarter_end   >= &Mquarter;
    quit;

With this approach, if the programmer needs to add or delete variables, the programmer can mark the change with a right-side comment:

    Proc sql;
    CREATE   table health_plan_choices as

        SELECT Company,
               Job,
               Health_plan,
               Worker_id  /*added March 4th
                2009 by Abigail Hammond*/
        FROM     library.occ_source
        WHERE    quarter_begin <= &Mquarter and
                  quarter_end   >= &Mquarter;
    quit;

3. Generally, all the statements in a data step or proc step should be indented except for the first and last one. Related “do” and “end” statements should be indented by an equal amount:

    Data hours_20;
    Set hours_10
if paid_hours ne . then
do;
  if conversion_flag = 'BP' then
do;
    ann_cost = hours * base_wage;
    ann_cost_alt = alt_hours * base_wage;
  end;
else /* Conversion_Code not equal to 'BP' */
do;
  ann_cost = hours * wage;
  ann_cost_alt = alt_hours * wage;
end;
end;
run:

One question is the ideal amount of an indent. Five spaces make for good
readability, but if the logic is nested to any degree, the programmer will run
out of room. Indentation of two spaces should be the absolute minimum,
since one space of indentation is difficult to follow.

4. Equal signs should be lined up, even when it means additional spacing
within a single line. The following is an example:

```
proc sql;
  create table all_bens_5 as
  select o.*,
    b.benefit as prior_benefit,
    b.cost_level
  from all_bens_4 as o left join
    benefit_table as b
  on o.company = b.company
  and o.job = b.job
  and o.benefit = b.benefit;
Quit;
```

We can easily see that the three lines beginning with “on” and “and” are
related.

5. Include the “run” or “quit” statements at the end of all statements. Doing so
makes it clear where one step of the program ends and another step begins.

**Commenting**
Commenting serves two important functions. First, it makes the code easier to follow. Second, it forces the programmer to review the logic of the code. If the programmer is unable to describe what the code is doing and the reasons for the coding choices, then the programmer almost certainly has insufficient knowledge to write reliable code for the program.

Every program should start with a “header” that contains comment code that lists information about the program. This information includes:

- The name of the programmer and the program
- The date created
- The function of the program
- The history of changes
- The SAS Version
- The input and output data

Comments should be included before each data step, proc step, and macro. The comments should explain why the coding was done a certain way. If your organization numbers the specifications and requirements, consider including that information as well.

```
/***************************************
*****
/***** Calculating straight time annual earnings */
/*/ (Straight_time_annual_earnings). The straight time */
/*/ annual earnings is the sum of the earnings for */
/*/ straight time worked and the earnings for time */
/*/ while on leave. This code fulfills specification 5.1 */
/*/ which refers back to requirement 82.8. */
╚══════════════════════════════════════════╝
```

One mistake is to have comments that do not explain the reason for the coding but merely explain what is in the code itself:

```
/******************
/* This macro will assign the value of */
/*/ Z45 and LR based on the value of */
/*/ data_type (dt) and abort if dt is not */
/*/ equal to 2 and excluded from JrsCd. */
╚══════════════════════════════════════════╝
```

In this case, another programmer reading the commenting would gain very little additional knowledge beyond looking at the code itself.

When making a change to the code, commenting should be made in the following places:
• The header at the beginning of the code.
• Above each data or proc step that gets changed.
• If possible, to the right of one-line changes. These comments should be brief.

One argument put forth by those who put in little commenting is that code can be “self-commenting.” The idea is that dataset names, variable names, and formatting should explain what the code is doing. These are indeed good practices. However, programmers often overestimate the ability of other programmers to follow code.

The English language has many cues to help people process information easier, from common grammatical rules to punctuation to capitalization. Programming has far fewer cues. To take one example, when we write a list of items in a sentence, for example, we put an “and” before the last item, as with “I am going to the store to get bread, eggs, and milk.” The writer subtly alerts the reader that the list is coming to an end.

SAS, like most programming languages, is designed for the convenience of the coder. If we list variables, for example, we don’t put an “and” before the last item. Since we don’t have all the cues that are found in the English language, a reader of code will have more difficulty processing the information than a reader of English prose. This increased difficulty can lead to misunderstanding of what the code is doing. Code can therefore never be entirely “self-commenting” and programmers should always put in additional commenting.

**Names of datasets and views**
Dataset names should describe as best as possible what is in the dataset. To allow for traceability, it is best to give each dataset a different name. If we are trying to debug a program, we can print out rows from any dataset and see how the various programming steps affect the records.

In many programs, however, different datasets have very similar properties, so giving unique descriptive names can be difficult. If a dataset with 50 variables gets modified to add one more variable from a database table, for example, the old and new dataset would have essentially the same properties. Giving the two datasets significantly different names would be confusing.

One solution is to give each dataset a numbered suffix. If a dataset contains the immunization records from Ohio, for example, the initial creation of the dataset could be called Immunizations_Ohio_10. The next call of the dataset called could be Immunizations_Ohio_20. Each call would increase the number of the suffix by ten. When numbering, it is best to increase the suffix by at least ten with each call to allow for the possibility that revisions of the program will require more datasets.
The problem with renaming is that multiple datasets can fill all the available space on a server. One solution is to use the proc datasets option to periodically delete datasets. With this approach, however, the traceability gets lost.

One good solution in large programs with big datasets is to use SAS views. A view is not an actual dataset. A view merely instructs the program to retrieve data from the datasets or input files from which the view was created.

Consider this example:

```sas
data view_example;
  input a b;
  cards;
  1 2
  1 3
  2 4
;

data view_example_2 /view=view_example_2;
  set view_example;
  if a=1;
run;
```

View_example_2 doesn't actually exist. Nevertheless, we can call View_example_2, make changes, print it, sort it, merge it with datasets, and do just about all the commands we could do with a regular dataset. Each time View_example_2 gets called, the computer just goes to the “View_example” dataset and gets the records where a=1. The program calling the view is a type of macro.

If View_example is ever deleted, any attempts to call View_example_2 will result in an error. Since the attempt to call View_example_2 will cause the program to pull records from View_example, View_example needs to exist for the call to work.

The use of views allow for traceability, since we can always print data from a view. Because the views do not take up space on the server, we can avoid the problem of having the datasets fill all the server space. For more information on SAS Views, please see Robert Ray’s Paper “Save Time Today Using SAS Views.” This paper can be found at http://www2.sas.com/proceedings/sugi27/p019-27.pdf.

**Variable names**

As with dataset names, the key to choosing variable names is clarity. Variable names can be up to 32 characters long. Sometimes, it is necessary to make the
variable name long to make sure the meaning is clear. If a variable gives the amount of the annual salary including the bonus, for example, Annual_salary_plus_bonus is a better choice for a name than Ann_Sly_with_bns. The advantages of clarity outweigh the disadvantages of typing the longer name.

Using the keyboard macros on the “tools” menu in SAS can alleviate the problem of having to type a long name. The paper “Practical Tips to Customize a SAS Session” by Xiaohui Wang and Beilei Xu of Merck & Co., Inc. gives some tips on keyboard macros. See http://www.lexjansen.com/pharmasug/2003/tutorials/tu027.pdf.

When pulling fields from a database, it is best to use the same variable name as what is on the database. Those reviewing the code can then refer to database dictionaries to see what the variable means.

**Macro variables and code**
Always use macro variables when assigning values that are likely to change. If the program refers to a date field, such as the year, create a macro variable:

```%Let m_year=2009```

For the rest of the program, the year can equal &m_year.

One useful standard is to have a coding convention for the naming of macro variables. One possibility is to put an m and an underline before the name of each macro variable, as was done in the previous example with m_year.

A good practice is to put all the macro code at the beginning of the program. If we can create a macro variable at the beginning of the program (such as with the previous example with the year) we should do so. The actual code within macros should also go at the beginning of a program.

With macros, the price we pay as programmers is that we gain efficiency and ease of updatability at the expense of the clarity of the code. A programmer should ask if the benefit is worth the cost. The programmer should only write a macro if multiple executions of code are required or a particular task requires the logic of the macro language.

Once a system reaches a certain degree of complexity, use of macros is inevitable. Different programs may make use of the same macros. In such situations, reading the code becomes very difficult. Programmers should prepare external documentation explaining the flow of programs and use of macros.
Using good programming practices for all programs
Some programmers have made the mistake of not following good programming practices because of the assumption that a program would be run only one time. Plans for programs can change, however, so this approach can cause problems. A program may get used in ways that the programmer did not expect.

Conclusion
All programs are solutions to a problem. The programmer should view the problem that must be solved as an abstract idea that is difficult for any one person to comprehend fully. Good communication is thus essential to programming.

Acquiring domain knowledge and using good programming practices can maximize the amount of communication between all people involved in a project. A high level of communication means that errors have a good chance of being caught before the program is deployed.

Bibliography


Contact Information
Comments, questions, and additions are welcomed. Contact the author at:

Jason Ford
Bureau of Labor Statistics
Email: ford_j@bls.gov

Acknowledgements
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.