MWSUG 2017 – Paper TT04

Code Like It Matters: Writing Code That's Readable and Shareable

Paul Kaefer, UnitedHealthcare, Minnetonka, MN

ABSTRACT

Coming from a background in computer programming to the world of SAS® yields interesting insights and revelations. There are many SAS programmers who are consultants or work individually, sometimes as the sole maintainer of their code. Since SAS code is designed for tasks like data processing and analytics, SAS developers working on teams may use different strategies for collaboration than those used in traditional software engineering.

Whether a programmer works individually, on a team, or on a project basis (delivering code and moving on to the next project), there are a number of best practices from traditional software engineering that can be leveraged to improve SAS code. These practices make it easier to read, maintain, and understand/remember why the code is written the way it is.

This paper presents a number of best practices, with examples and suggestions for usage in SAS. The reader is encouraged not to apply all the suggestions at once, but to consider them and how they may improve their work or the dynamic of their team.

INTRODUCTION

For someone with a background in software engineering, it's easy to see the differences that exist in the SAS field. While SAS is an older language than many languages currently used in software development, the kinds of positions that require SAS tend to be quite different. Consultants and single-person teams tend to have a very different workflow than teams collaborating on large software projects. Even a large team of SAS developers that work together are typically not delivering a product, but rather a set of processes that are expected to be updated over time.

The aim of this paper is to bridge the gap between best practices in software engineering and the field of SAS programming, where these practices can greatly benefit developers who may not have experience in traditional software programming.

To be clear, the suggestions in this paper are exactly that – suggestions. Your organization or team may have different opinions or standards for a variety of reasons. These suggestions are provided with logical reasoning, and should be contemplated and applied only as makes sense. If this paper (and the associated presentations) is fuel for beginning the discussions, then it has been a successful endeavor to write.

WHY WRITE READABLE AND SHAREABLE CODE?

Software developers often have to share their code with others. These others may be other members of the team, the end users who will run the code, or another developer who is called upon to take up the mantle of maintaining the code. Another future user is the developer themselves, who may have to revisit the code months or years after it was written. Code written long ago and long-since forgotten can seem like it originated with someone else.

And thus, we aim to make our code readable. It should be immediately clear why things were written the way they are, whether to someone new to the project, or to your future self, trying to remember why you originally did what you did.

In the above explanation, "readable" and "shareable" have similar meanings. The difference in meaning here is "how the code reads" vs. "how well someone new can understand the code." You should aim for both: to make your code look pretty or nice, [1] and so that someone new could start on your code and hit the ground running for any changes they want to make.

BEST PRACTICES FOR REUSABLE CODE

The idea of reusability is forward-thinking. A programmer is likely to encounter similar problems in the future, so code should be written with the knowledge that it may be reused. Similarly, we know that systems change and data tables grow, so we want our code to be able to adapt to these changes. We want our code to be written in such a way that it can adapt with the times. A question that can help us ponder how to write code for the future is: what might be expected of this code someday? Do we really expect it will always do the same thing, or might there be some future expansion in functionality?

We want to generalize and modularize our code, and ensure that the documentation explains what the code does in a clear and concise way. In this way, the future developer(s) (whether that be someone else or the original developer long after they remember what they've done) can adapt the code, rather than scrap indecipherable code and have to begin again from scratch.

In SAS, a great way to both generalize and modularize is to write macro functions. This helps avoid having duplicate code in your programs, which is an undesirable practice. [2, p. 186] We can easily write a macro function that will adapt to different filenames, dates, etc. This is an example of a macro function that can be called for Excel files of different names, but to be imported in the same way:

```
%macro import_dataset(filename=, output_dataset=);

    proc import datafile = "&filename."
                    out = work.&output_dataset.
                   dbms = xlsx
              getnames = yes;
    run;

 %mend import_dataset;
```

Making macro functions that can be dynamically reused is a fundamental part of automating your SAS code, and process automation is widely considered a best practice. [3, p. 329] [4] Techniques like this can be very useful as things like database schemas or URLs might change, columns or variables might change names, order, or format, and structure of input or configuration files might change. Having code that can adapt helps avoid having to recode for any small change.

To add to this, when a function is written that may be of use across projects, you can save it. You can either save it to a file with your own collection of such functions, or to something like a keyboard abbreviation/macro. [5]

Comments are very important to consider. Ideally your code can explain itself, [6, p. 55] but often we flatter ourselves with the assumption that our code is perfectly clear and easy to be reused. Yet this is proven to be untrue when we ourselves revisit our code months or years later. By considering what assumptions need to be made in order for the code to work properly, we can better comment our code so if we forget the details, we can quickly remember them.

Comments should come with a caveat: Don't comment out a huge block of code. Consider making it a macro function if it may be used someday. You can simply surround it with a macro definition, but not call the macro function if it is not needed at this time. However, if the block of code is something you no longer need, use **version control** (to be discussed later).

CODE STANDARDS

Organizations with large teams of programmers typically have agreed-upon standards to which code should conform. The idea is that by following a set of standards, code will look uniform and thus be easier to edit.

Consistency is the most important part of code standards. The details will differ by organization, programming language, and personal philosophy. Whatever they are, the best standards will involve writing code in a consistent matter.

WHY CONSISTENCY?

The answer to this question can be boiled down to one example: imagine two code files in the same project that are written by two individuals with vastly different coding styles. If they do not adhere to consistent standards, their code will look very different, even if they do similar things. This will make it difficult to understand how the code works together, and give the project an appearance of being unorganized.

Consistency is also important in fields with existing process standards. Or in organizations that follow ISO standards. One example that many organizations follow is the ISO 8601 date standard. [7] Applicable standards may impact how code should be written, or variables like dates should be formatted.

Another benefit of consistency is the personal discipline. As you refine your own skills and learn to be a better programmer, consistently adhering to best practices will help you make them routine. Once they are routine, they are no longer tedious to think about and practice.

Your coding style is part of your personal brand, and should be treated and developed as such.

EXAMPLES OF CODE STANDARDS IN SAS

One of the most obvious purposes of standards in SAS is for table and variable names. Names should always be descriptive, revealing information on what the purpose of the table or variable. [6, p. 18] Style of the name should be either `camelCase`, where each successive word begins with a capital letter (`tableNameWithCamelCase` is easier to read than `tablenamewithoutcamelcase`), or `using_underscores_for_spaces`. It doesn't matter what standard you use – just pick one, and stick with it. One option is camelCase for variables, and underscores in table names. This enables naming tables with a strategy like:

```
<domain/data source>_<subset/area>_<specific purpose>_<datestamp>_<version>
```

For datasets, SAS enables up to 32 characters to be used. Take advantage of this fact. Macro variable names do not have the same description, but they should not be excessively long.

Even work datasets, which are temporary, should be named descriptively. This helps communicate their purpose and function. In some cases, temporary datasets are saved to a folder separate from the main program outputs, so a descriptive name helps identify them.

SAS ignores case, so there is no need to use all capitals (this applies to both variable names and language keywords, for example in proc sql queries). In text messaging, use of all capitals can signify heightened emotion, something rarely desired to be communicated in code. Use all capitals sparingly, or perhaps only for dataset names.

Use of whitespace can greatly improve readability of your code. Consider the following example:

```
1  proc sql;
2      create table new_table as
3      select e.firstName
4            ,e.lastName
5            ,e.age
6            ,s.salary format=comma18.2
7      from employee e
8      inner join salaries s
9          on e.employee_ID = s.employee_ID
10     where e.employee_ID is not missing
11         and s.salary not in (., 0);
12 quit;
```

Note how the proc sql block starts and end the furthest to the left. Each of the query's clauses [8] are indented once (four spaces), and sub-units like the join criteria and where clause filters are indented one level further. The variables in the select statement are aligned, as would be with any group by variables.

In the above example, you may notice that I put commas *before* the variables in the select statement.

There is some debate on whether to put commas before or after. [9] I have learned that I prefer to place them before because (1) they align vertically, (2) it's easy to see where a comma might be missing, and (3) in the situation where a CASE WHEN is used, it is easy to denote that potentially several lines all yield one variable. Whatever style you use for commas, be consistent.

Another standard relating to queries is to avoid listing all variables on one line. A bad example is:

```
proc sql;
    select a format=date9.,b,c as different_name,d label="test label" format=$50.,e
    from dataset
    group by a,b,c,d,e;
quit;

```

Note how the user is required to scroll horizontally to read the entire query, and it is not immediately clear how many variables are being selected. Reformat such queries like so:

```
proc sql;
    select a format=date9.
          ,b
          ,c as different_name
          ,d label="test label" format=$50.
          ,e
    from dataset
    group by a
             ,b
             ,c
             ,d
             ,e;
quit;
```

…enabling much faster comprehension. The motivated programmer may even align labels and formats vertically.

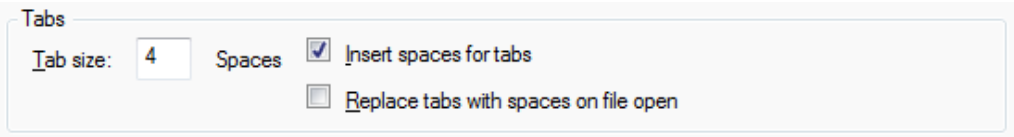
Others have developed coding standards for SAS, for specific fields or general-purpose programming. See [10] and [11], for example.

TABS VS. SPACES

If you've done much coding - especially across multiple languages - you will be familiar with reasons this question is important. Whitespace is important for readability (aesthetics) of your code, and in cases like python, it actually matters for execution.

Reasons for switching from tabs to spaces are as follows. If you edit your code in more than one editing environment, [12] the format will remain the same. Even within the same environment, if you change text before or after a tab, it can often adjust or "snap", changing the formatting of that line. Spaces really are the way to ensure your code will look the same every time it is opened, across systems or platforms. Likewise, if you do use multiple languages, you won't have to change your style from one language where whitespace doesn't matter (SAS, Java, R) to a language where it does (python).

It's easy to change this setting once and forget about it. In SAS Enterprise Guide®, go to Program >> Editor Options… and set the following:



If you're not convinced, consider the data showing that programmers who use spaces may make more money. [13]

NESTING

Sometimes conditional expressions (if statements and the like) are nested. Whitespace should be used to help illustrate the level of such nesting. Each successive level of nesting (or subquery, perhaps) should be indented one level further, like so:

```
10 data nesting_example;
11     if (a = 4) then do;
12         /* ... */
13         if (b = 7) then do;
14             /* ... */
15         else do;
16             /* ... */
17     end;
18 run;
```

ORDER OF OPERATIONS

Equations in SAS follow the standard mathematical order of operations, with equations in parentheses being evaluated first. Likewise, parentheses can be used in logical equations (if/then/else logic).

It doesn't hurt to add more parentheses if it makes the equation or logic easier to read or to appear less complicated. For example:

```
1 data order_of_operations;
2
3     set numbers;
4
5     if (a = 4 or
6            (b = 3 and
7                (c = 5 and d = 7)));
8 run;
```

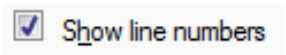
The parentheses around (c = 5 and d = 7) are not necessary, but may make it more clear to the reader.

MISCELLANEOUS TIPS

Semicolons should be placed at the end of every [relevant] line. Having a line with only a semicolon is useless, and some programming languages will throw an error if a line is not terminated with a semicolon.

While on the subject of lines of code, try and keep your lines so they fit within the window on a standard screen. It will be easier to read if you only have to scroll vertically and not horizontally.

Another editor setting that can be helpful for debugging and reading code, though will not impact the code itself, [12] is to enable line numbers. In SAS Enterprise Guide, go to Program >> Editor Options… to enable:



Another tip that should go without saying is to avoid having personal or sensitive information in your code. If you need to make connections to databases that require passwords, use SAS' built-in encryption rather than including passwords as plaintext. [14]

FONT

It almost goes without saying that a `monospace` font should be used. In such fonts, all characters have the same length, meaning that an **i** is not skinnier than an **o**. This enhances readability of code on a screen. Integrated development environments (IDEs) like SAS Enterprise Guide, use monospace fonts by default. So this standard really only applies to publishing code in external documents, for example when

embedded in documentation or a conference paper.

BEST PRACTICES FOR SHARING YOUR CODE

VERSION CONTROL

When this paper is presented, the audience will be asked by a show of hands whether they use version control. If all hands are raised (which hasn't happened yet), it would not be important to spend much time here. **All** programmers should **always** use version control, though I know this hasn't always been the case for SAS programmers.

To give a quick introduction, version control is how we track changes or iterations of our code. As we develop the code, consecutive versions can involve minor tweaks or massive overhauls. The idea is to track all of said changes, so we can go back and see those differences. In a very basic way, this can be done by renaming successive versions (code_1.sas, code_2.sas, or something with the modification date), but this is not the best practice.

Not only should all programmers be using version control, but all programmers should be using some version control software tool(s). Whether they be git, Subversion, Mercurial, or something else, this tool should be something the programmer uses consistently and with ease. Though there is a learning curve, the basics can be taught in under an hour, and even the basics will prove extremely useful.

If the reader is new to version control, I recommend reading the article on sasCommunity.org. [15] Several references are provided on that page for further reading. I will only give a brief explanation.

The idea of these software tools is to track changes made to code. This is done by comparing differences (or **diffs**) between the updated version and the previous version, and saving those differences, rather than saving new copies of the full code in the **repository**, to which each user has access. When some changes have been made and saved by the programmer, the programmer stages a **commit** (set of changes, whether they be to one or more files), and **pushes** those changes to the repository. Anybody with access to the repository can then **pull** the latest version with whatever changes have been made, and continue development themselves. Thus, version control also facilitates cases where multiple people are working on the same codebase.

STANDARDS FOR VERSION CONTROL

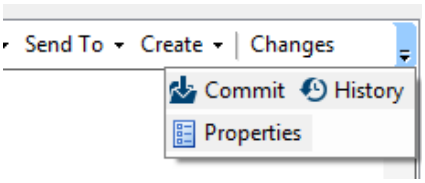
One standard for version control is to make a commit with whatever changes have been made, typically at the end of the day, though multiple times a day may be necessary for periods of high development activity. Even if the changes are not "finished" or "production-ready", commits can be made to a development **branch** in the repository, or a branch for a specific developer. Once a given issue has been resolved or feature completed, it can be **merge**d with the main branch (sometimes called master or trunk).

Another standard is to have descriptive commit messages. When a programmer has prepared a set of changes (or commit) that they are ready to be tracked, they must provide a message to accompany that commit. Blank messages are possible, but highly discouraged. Rather, this message should explain what was changed and why, in under approximately 80 characters. [16] If necessary, the developer can go into more detail on successive lines in the commit message. The first line should describe the important aspects of what was done. If bug tracking software is used, it may mention an issue number that was resolved. Otherwise, a high-level summary with descriptive terms that may be searched in the future should be used.

A final standard is to use .sas files instead of .egp (Enterprise Guide Project) files. Because they are text files [1] (as opposed to the binary .egp files that are essentially .zip files), version control software can track the changes made to them. Also these are not specific to SAS versions or platforms.

VERSION CONTROL INTEGRATION IN SAS ENTERPRISE GUIDE

There is existing support in SAS Enterprise Guide for version control using git: [17]



While the available features are limited, more integration is likely on the way. [18]

CLEAR, CONCISE DOCUMENTATION

If this paper appears to be repetitive by mentioning documentation again, the point has been reached. It is of the utmost importance to have clear and concise documentation.

COMMENTS, COMMENTS, COMMENTS

Comments should not describe exactly what the code is doing it. For example:

```
data sample;
    /* set variable x equal to one */
    x = 1;
run;
```

…is unnecessary and obvious. If the code is complex or doing something that may not be obvious, a comment should be used. However, comments should go further and explain *why* the code is doing what it is doing. Is the variable being set to the value of one to be an important flag used later? Will this macro variable be used to determine names for output datasets and reports? Document these things, so they do not come as a surprise to the reader later.

Looping back to version control, your comments can mention specific commit(s) as are relevant. As an example, if a major feature was added or removed, it may be useful to point to when that happened, so someone reading the code could consult those commits for more information. Something like a header/audit trail at the top of the program can be used to identify dates of major changes, and even reference specific commits. Here is a sample header:

```
/*************************************************************************
***************  PROGRAM NAME (not necessarily file name) ******************
*************************************************************************
Program Name          :      code_name.sas
Owner                 :      author/team that developed/maintains this code
Requestor             :      business user/team/purpose
Approximate Run Time  :      27-30 minutes
Program Description   :      Description of the program.
                             This can extend to multiple lines.
Input                 :      LIB.INPUT_DATASET
                             ...
Output                :      LIB.DATASET_NAME
                             ...
Dependencies          :      other_code_file.sas
                             (typically if required to run first)
Macro usage           :      brief description of macro variables/functions
                              provided
Audit Trail           :
 2016-04-11  Paul Kaefer   created the program
 2017-05-02  Paul Kaefer   restructure output datasetS according to updated
                            spec
 2017-08-22  Paul Kaefer   removed obsolete code (see git commit
                            3668a399e176c6a0276fa0eeb13c5a82fdab6fbd)
 2017-08-23  Paul Kaefer   updated according to documentation
                            (/.../file/path/); see Subversion commit 92
```

```
**************************************************************************
**************************************************************************
**************************************************************************/
```

Note that it is typical to use one version control software, but this example also illustrates the difference in how commits are named between git and Subversion, two of the more common tools.

CODE USED FOR DEBUGGING

As programmers, we all have code that is used for testing, whether it prints something to the log, runs a proc freq or proc summary on an intermediary dataset, or sets a variable to a predetermined value so we can verify if outputs meet expectations. These are good practices. Unit testing is better, but that is not the focus of this paper.[1]

Debugging code is often incredibly useful, and sometimes tedious to keep writing and removing. There are a couple of options for dealing with this: (a) commenting it out with a note:

```
/* Debugging:
%put &=variable_name;
%put &=dataset_name;*/
```

…or (b) setting a macro variable as a flag that will control running such code:

```
%let debug = on;
…
%macro function_name(params=);
    …
    %if "&debug." = "on" %then %do;
        …
        /* debugging code */
    %end;

%mend;
…
```

It is important to note that code used for debugging also has a place as part of the code's documentation. So it should follow the same standards as the rest of your code.

DEVELOPING A CULTURE OF STANDARDS

Code review can be very effective for developing a culture of standards at your organization. Typically code review is best for teams that collaborate on the same codebase, but even team members working on different projects can help review code and help ensure standards are implemented. Code review can be formal, where the team gathers in a conference room and individuals walk through their code via a projector. Or various code review software tools can be used so contributors can review team members' code at their own pace.

However it is done, code review should focus on *constructive* criticism, sharing ideas, and encouragement. We can all work together and learn to adhere to standards.

APPLICATION OF THESE CONCEPTS TO DATA

Given that SAS programmers are often data analysts/scientists, or are at least working with large datasets, it is appropriate to discuss best practices for data. Whether we are sharing our data (across

[1] For information on unit testing, see Robert C. Martin's *Clean Code*, [6] Kent Beck's *Test-Driven Development By Example*, Marc Clifton's *Unit Testing Succinctly*, or several SAS-specific implementations: FUTS, [20] SclUnit, [21] or methods explained in "Unit Testing as a Cornerstone of SAS® Application Development". [22]

teams or even publicly), planning to store it and use it in the future, or designing databases/data warehouses, we should consider the implications of our dataset design and structure. These methods are sometimes called data tidying. [19]

Like with variable names, column names should be descriptive. Similar standards should apply.

Data dictionaries should accompany datasets so new users can easily identify the purpose and description of each column.

CONCLUSION

The stated aim of this paper is to present best practices of software engineering in a way that may be used by SAS developers. While the amount of information and ideas may be overwhelming, the main takeaway should be a paradigm shift towards using standards to improve how code is read and shared.

When developing standards, discussion is necessary. The tips presented here are from the author's experience and research. The reader is encouraged to adapt as they see fit for their organization and/or project.

REFERENCES

[1] K. Bremser, "Maxims of Maximally Efficient SAS Programmers," communities.SAS.com, 24 April 2017. [Online]. Available: https://communities.sas.com/t5/SAS-Communities-Library/Maxims-of-Maximally-Efficient-SAS-Programmers/tac-p/370642?nobounce.

[2] D. E. Knuth, The Art of Computer Programming, Third ed., vol. 1: Fundamental Algorithms, Addison-Wesley, 1997. ISBN 978-0-201-03804-0. Available: https://archive.org/stream/camputorsinceeboocz2/Knuth%2C%20Donald%20E.%20-%20The%20Art%20Of%20Computer%20Programming%20Vol.%201%20-%20Fundamental%20Algorithms#page/n0/mode/2up.

[3] A. Carpenter, Carpenter's Guide to Innovative SAS® Techniques, SAS Institute Inc., 2012.

[4] fankaiqing, "What should be The Golden Rules for each SAS Developer to Follow?," communities.SAS.com, 21 August 2017. [Online]. Available: https://communities.sas.com/t5/SAS-Communities-Library/What-should-be-The-Golden-Rules-for-each-SAS-Developer-to-Follow/ta-p/389606. [Accessed 22 August 2013].

[5] "Abbreviations/Macros," 1 June 2017. [Online]. Available: http://www.sascommunity.org/wiki/Abbreviations/Macros. [Accessed 23 August 2017].

[6] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Boston, MA: Pearson Education, Inc., 2009.

[7] "ISO 8601," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/ISO_8601.

[8] "Anatomy of a SQL query," sasCommunity.org, [Online]. Available: http://www.sascommunity.org/wiki/Anatomy_of_a_SQL_query.

[9] "mysql - Leading Comma or Trailing Comma?," Stack Overflow, 11 January 2014. [Online]. Available: https://stackoverflow.com/questions/21063036/leading-comma-or-trailing-comma. [Accessed 23 June 2016].

[10] "Good Programming Practice for Clinical Trials," sasCommunity.org, [Online]. Available: http://www.sascommunity.org/wiki/Good_Programming_Practice_for_Clinical_Trials.

[11] "Style guide for writing and polishing programs," sasCommunity.org, [Online]. Available: http://www.sascommunity.org/wiki/Style_guide_for_writing_and_polishing_programs.

[12] C. Hemedinger, "Ten SAS Enterprise Guide program editor tricks," The SAS Dummy, 3 July 2017. [Online]. Available: http://blogs.sas.com/content/sasdummy/2017/07/03/sas-program-editor-tricks/.

[13] R. Allison, "Tabs -vs- Spaces: Which coders make more money?," blogs.sas.com, 2017 20 June. [Online]. Available: http://blogs.sas.com/content/sastraining/2017/06/20/tabs-vs-spaces-which-coders-make-more-money/.

[14] L. Batkhan, "One deadly sin SAS programmers should stop committing," SAS Users Blogs, 13 April 2017. [Online]. Available: http://blogs.sas.com/content/sgf/2017/04/13/one-deadly-sin-sas-programmers-should-stop-committing/.

[15] "Version control," sasCommunity.org, [Online]. Available: http://sascommunity.org/wiki/Version_control.

[16] "Studies on optimal code width?," Stack Overflow, [Online]. Available: https://stackoverflow.com/questions/578059/studies-on-optimal-code-width.

[17] C. Hemedinger, "The SAS Dummy," SAS, 18 August 2017. [Online]. Available: blogs.sas.com/content/sasdummy/2017/08/18/debug-program-history/.

[18] C. Smith, "enable non-local git server integration," SASware Ballot Ideas, 29 November 2016. [Online]. Available: https://communities.sas.com/t5/SASware-Ballot-Ideas/enable-non-local-git-server-integration/idi-p/312736?nobounce. [Accessed 1 December 2016].

[19] H. Wickham, "Tidy Data," *Journal of Statistical Software,* vol. 59, no. 10, 2014.

[20] J. Wright, "Drawkcab Gnimmargorp: Test-Driven Development with FUTS," in *SUGI 2006*, 2016.

[21] D. Scocca, "Automated Unit Testing for SAS Applications," in *SAS Global Forum*, 2008.

[22] D. D. Tommaso, "Unit Testing as a Cornerstone of SAS® Application Development," in *PhUSE*, 2011.

RECOMMENDED READING

- R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship,* Boston, MA: Pearson Education, Inc., 2009.
- "Good Programming Practice for Clinical Trials," sasCommunity.org. Available: http://www.sascommunity.org/wiki/Good_Programming_Practice_for_Clinical_Trials.
- "Style guide for writing and polishing programs," sasCommunity.org. Available: http://www.sascommunity.org/wiki/Style_guide_for_writing_and_polishing_programs.
- Tricia Aanderud, “Plate of Spaghetti Anyone? Techniques for Learning Existing SAS® Programs”. SAS Global Forum 2011. Available: http://www.sascommunity.org/wiki/Plate_of_Spaghetti_Anyone%3F_Techniques_for_Learning_Existing_SAS%C2%AE_Programs

ABOUT THE AUTHOR

Paul Kaefer has a B.S. in computer engineering and an M.S. in computational sciences, both from Marquette University. While pursuing his master’s, Paul worked for Marquette University GasDay, a business housed in a research lab that develops software for natural gas forecasting. After earning his M.S., Paul spent five months volunteering with the Ifakara Health Institute in Tanzania, leading statistics/R and technical English communication workshops. Paul now works as an IT Data Analyst (mostly developing in SAS) for UnitedHealthcare in Minnetonka, MN.

You can view this paper and the associated presentations at http://www.sascommunity.org/wiki/Code_Like_It_Matters:_Writing_Code_That%27s_Readable_and_Shareable

CONTACT INFORMATION

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

Paul Kaefer
UnitedHealthcare
paul_kaefer@uhc.com
sasCommunity.com/wiki/User:paulkaefer
paulkaefer.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 trademarks of their respective companies.