PharmaSUG 2021 - Paper EP-077

Introducing and Increasing Compliance of Good Programming Habits

Alan Meier, Cytel, Inc.

ABSTRACT

SAS code is often shared within a task, project and organization. The better written the code is, the easier it is to understand, use, modify and validate. Improving programmers' coding habits increases the productivity of the entire team. The challenge is getting programmers to learn and use good habits.

To achieve improvement, a manager first needs to define what good coding habits are. These include in-program documentation, style requirements, use of modular programming techniques, rules on hardcoding, and naming conventions, to name a few. These habits then need to become part of the organization's expectation, usually through SOPs or Work Practice documents.

Most programmers have their own styles. Getting them to learn new habits can be challenging. Like any change, the manager needs to roll them out to the staff by explaining the benefits, building excitement around them, educating staff, allow time to practice and then require their usage. Finally, the manager needs to ensure compliance to the new requirements

INTRODUCTION

Everyone in this industry has used other peoples code, either entire programs or segments. It's obvious to the casual observer that the more readable the code is, the easier it is for someone picking up a program or section of code to understand the logic and correctly make necessary modifications. This results in reduced development time, more accurate deliverables and cost savings. ***Practices of an Agile Developer*** notes "*you should always choose readability over convenience. Code will be read many, many more times than it is written;*".

What makes code more readable includes use of modular programming techniques, internal documentation and a consistent style. As managers, we are called upon to provide guidance to staff. In my career, I've had several opportunities to write or consult on department standards that include style guides. All of them have included the three aspects listed above.

Developing a style guide is only the first step. Getting staff to accept and consistently use it is a significant challenge as most have their own personal preferences when it comes to writing code. There are several change management techniques that can be employed to increase compliance in addition to making usage a requirement for annual performance reviews

STYLE GUIDE

A style guide is usually written as a departmental work practice/guidance or operational procedure. This makes them official company documents, requiring their use and usually requiring staff training. They can be written by one person or by a team. They need to offer enough guidance without unnecessarily restricting individual preferences.

MODULAR PROGRAMMING TECHNIQUES

The Pragmatic Programmer defines **orthogonality** in code as "*Two or more things are orthogonal if changes to one do not affect any others*.". This concept directly addresses things like using the same variable name or macro variable name with different definitions/purposes within a program. It also encourages the use of macros for repeated code. Should the logic change and the change is made in only 1 set of code, the other sets are affected as they are no longer using the up-to-date logic.

Matt Bishop defines **robust** as "*Robust programming, also called bomb-proof programming, is a style of programming that prevents abnormal termination or unexpected actions. Basically, it requires code to handle bad (invalid or absurd) inputs in a reasonable way. If an internal error occurs, the program or library terminates gracefully, and provides enough information so the programmer can debug the program or routine.*". What this means is that programmers should always programmatically check their assumptions and print warnings or reports when those assumptions are violated.

Robustness also encourages use of simple code that can be easily understood and maintained or extended. David N. Welton wrote "*The Rule of Robustness in the Unix philosophy states that robustness results from transparency and simplicity. Software is transparent when a skilled programmer can examine its source code (i.e., the original version written by a human in a programming language) and soon comprehend how it works. It is simple when its operation is sufficiently uncomplicated that a programmer can visualize with little effort all of the potential situations that it might encounter. The more that programs have both of these qualities, the more robust they will be.*". Advanced programmers sometimes like to write complex code to reduce resources or lines of code, but less-advanced programmers will have trouble or be unable to understand what the code does. That makes maintaining the programs difficult, especially when the advanced programmer moves on.

The Free On-line Dictionary of Computing defines **scalability** as "*How well a solution to some problem will work when the size of the problem increases.*". When writing programs, thought should be given to how the request might change in the future. For example, a table may only be reporting Serum Sodium levels for baseline and Week 12, but as the study continues, Weeks 24, 36 and 48 may be added. The programmers should try to write code in such a way as facilitate these changes.

As for **extensible** programming, ***The Free On-line Dictionary of Computing*** says "*Said of a system (e.g., program, file format, programming language, protocol, etc.) designed to easily allow the addition of new features at a later date, e.g. through the use of hooks, an API or plug-ins.*". In the example above, the table might also need to be extended to include Serum Potassium levels. If the table code uses a macro with the analyte code as an invocation parameter, adding addition analytes would require minimal changes.

I call this the MORSE programming technique (M=Modular, O=Orthogonality, R=Robust, S=Scalable, E=Extensible). David N. Welton wrote "*Naturally, we can't forget the "scalability" part of the equation while focusing on "simple". It's not that hard to make something small and easy to use. The trick is that it's also got to be extensible, in order to grow. The system should be easy to pick up, but it should also be easy to add to, and should also provide tools to modularize and otherwise compartmentalize the growing system in order to not end up with what is known in technical terms as a "godawful mess". And of course it should still be fun to work with for the more experienced programmer.*". While he was writing about larger systems, the same holds for the programs that make up the systems.

DOCUMENTATON

All programs need documentation. They should all begin with a header that includes the following items:

- Project and protocol
- Program name
- Purpose of the program
- Author (full name – not initials)
- Inputs and outputs
- Date of initial writing
- Modification log including date, author and a description of changes

Other items that might be included are macro calls, platform and SAS® version where the program was run. There should be some type of frame to offset the header from the body of the program.

Comments should be frequent and with sufficient detail to allow the reader to understand what the code or procedure is doing by reading just the comments. Any complex logic and PROC SQL sub-queries should be described in comments. Also from the ***Practices of an Agile Developer*** "*One way to make code understandable is to make it obvious to see what's happening.*". I usually add a comment for each data step and procedure, even if the data step is just bringing a permanent data set into the work library.

Once a deliverable goes out, any modifications to the code should be identified with a comment where the code was modified as well as in the header..

CODING GUIDELINES

No one wants to be dictated to on how they write code, but in order to maintain readability, some structure needs to be provided. These can include:

- Indentation – let programmers decide how many spaces they want to use, but set requirements including:
  - All data and procedure statements except DATA, PROC, RUN and QUIT should be indented
  - All code in a DO-END block should be indented
  - Indentation should be consistent throughout the program
  - Spaces are used instead of tabs as tab widths can change across applications/user preferences
- Use of case – don't dictate when upper and lower case is used, just require consistency for keywords, data set names and variable names.
- Use of blank lines for separation – require at least one between each data step, procedure call, and SQL query.
- Putting more than one statement per line – normally there should only be one per line, but exceptions might be granted (e.g. proc sort data=SORTME; by sortv1; run;).
- RUN/QUIT statements – require all data steps and procedure calls to end with the appropriate RUN/QUIT statement.
- DATA= and SET/MERGE statements – nowhere should a programmer assume what the previous data set is/are and should be required to specify data set names.
- Placement of non-executable statements – I've seen some style guides that require these statements (e.g. KEEP, DROP, RETAIN, LENGTH, LABEL, FORMAT, ARRAY, etc.) be at the top of a data step.

NAMING CONVENTIONS

To enhance readability and understanding, programmers should use descriptive data set and variable names that reflect what is in the data set or variable. No one should ever see anything like:

```
**** When Harry met Sally ****;
data HARRY;
   set SALLY;
   var1 = 'Met';
   var2 = 'Love';
run;
```

Practices of an Agile Developer defines the PIE Principle as "*Code you write must clearly communicate your intent and must be expressive. By doing so, your code will be readable and understandable. Since your code is not confusing, you will also avoid some potential errors. Program Intently and Expressively.*".

There are also benefits to defining structure to program, permanent data set and output names that provide meaning and ease in finding files. Programs and outputs should share common elements. For example, ADLS data set is created by a program named ADSL.SAS and is validated by a program named Q_ADSL.SAS. Table programs and output can begin with "T", followed by the table number, data, analysis and population (e.g. T_1_3_1_AE_SUM_SAF.SAS is used for a summary table of AEs experienced in the safety population and is identified as Table 1.3.1). Note that if outputs get renumbered, this specific convention would require renaming programs and outputs.

HARDCODING

Sponsor companies should already have a policy on hard-coding (explicitly changing data values within a program, usually done during on-going trials where data is still dirty). Regulatory agencies usually frown on this, especially in final submissions. However, code sometimes gets inserted during development and forgotten. If hardcoding is authorized, comments should be placed in the program header and around the code (including identification of the authorization documentation) and warning messages included in the log to alert the programmer that the code is still present on the current run.

MACROS

The file header documentation of saved macros need to document the invocation parameters and any global macro variables that are created or modified.

The use of keyword parameters should be encouraged (if not required) over positional parameters. This makes it explicitly clear what values invocation parameters are being assigned and avoids mixing up positions.

ROLLOUT AND COMPLIANCE

There are a number of practices from Change Management that can be used to successfully rollout a style guide and ensure compliance.

UPPER-MANAGEMENT SPONSORSHIP

Before starting on any of this, managers should communicate with upper-management and ensure they have buy-in. Expect to defend your cause with a clear list of benefits and costs; benefits being better quality and quicker turnaround, resulting in cost savings, and costs being potentially unrest and dissatisfaction among staff.

Getting upper-management to actively and visually supporting the effort will greatly assist in getting a team to accept and comply with the new guidelines.

COMMUNICATE WITH STAFF

Once upper-management support is secured, let the staff know what’s coming. Explain what is planned and the benefits to them and the organization. Give them an opportunity to express their initial concerns. At the first discussion, don’t try to address the issues, just listen. Identify any good suggestions, areas of concern, individuals who are enthusiastic, and individuals that appear to be set against it.

If possible, select some of the enthusiastic supporters and detractors to help draft and review the style guide. These people can become early adapters.

Discuss the project frequently and openly. Focus on the benefits to the team members. Help them see what programming will be like with the new guidelines. Get the early adapters to discuss what changes they had to make to their coding style and any be open about any issues they encountered. The more staff hears about it, the more they become socialized to it and the rollout won’t be as traumatic, as they’ll have opportunities to influence the content.

Build excitement around it and make it fun. In the COVID-19 era and remote working, it’s hard to throw a party, but think of other ways to celebrate and reward successes. Make it fun – maybe have a contest to select a mascot or logo.

EDUCATION

Leading up to the rollout, provide a reference list of books, papers and WEB sites where staff can go to read about modular programming and other style guides. Give presentations on what is modular programming.

At the rollout, walk-through the style guide’s contents. If possible, record the rollout and other presentations as training for new staff. Include reading the guide as part of their compliance training if possible.

COMPLIANCE

With the rollout, discuss the compliance requirements. It's probably not feasible or cost effective to bring old code up-to-date, but any new programs or significant code additions to existing programs should follow the new guidelines. Compliance checks may include random code review by the manager who will provide feedback. A checklist can be developed to help staff and managers check for compliance.

Give staff a set period to practice the new elements of the style guide, maybe several months, before starting reviews. When doing reviews, select code that was written after the practice period. Don't confuse code that was written prior to the rollout with what was written after.

Eventually, compliance with the new guidelines should be incorporated into annual objectives and have an effect on annual assessments.

CONCLUSION

Good programming practices (modular programming techniques, internal documentation and a consistent style) makes code readable and understandable, which makes maintenance and reuse easier, faster and reduces errors – thus reducing cost. These practices should be encouraged or required from programming staff members and can be documented in a style guide.

As programming style is a personal preference, getting buy-in from staff can be challenging. As long as the requirements don't overly restrict individuality and are introduced appropriately, buy-in can be achieved.

REFERENCES

Subramaniam, Venkat and Andy Hunt. 2006. Practices of an Agile Developer. Raleigh, NC : The Pragmatic Bookshelf.

Hunt, Andrew and David Thomas. 2000. The Pragmatic Programmer. Boston, MA : Addison-Wesley.

Bishop, Matt. "Robust Programming." Department of Computer Science, University of California at Davis. 1998-2002. Accessed March 11, 2021. Available at http://nob.cs.ucdavis.edu/~bishop/secprog/robust.html.

Howe, Denis. "The Free On-line Dictionary of Computing." 1985. Accessed March 11, 2021. Available at http://foldoc.org/Scalability.

Howe, Denis. "The Free On-line Dictionary of Computing." 1985. Accessed March 11, 2021. Available at http://foldoc.org/Extensible.

Welton, David N. "Downwardly Scalable Systems." Accessed March 11, 2021. Available at https://www.welton.it/articles/scalable_systems.html.

RECOMMENDED READING

- *The Pragmatic Programmer*
- *Practices of an Agile Developer*

CONTACT INFORMATION

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

Alan Meier
Cytel, Inc.
Alan.Meier@cytel.com