PharmaSUG 2015 - Paper TT01

Team-work and Forensic Programming: Essential Foundations of Indestructible Projects

Brian Fairfield-Carter, inVentiv Health, Cary, NC

ABSTRACT

Everyone probably agrees that project success hinges on teamwork and communication. The concept of teamwork often emphasizes leadership and organizational structure over individual responsibility, but the scale and complexity of analysis programming projects mean that effective teamwork demands individual autonomy and initiative and should ideally involve a cooperative effort among peers. And despite generally recognizing the importance of communication, we rarely see pragmatic suggestions on communication techniques.

With autonomy comes responsibility: while 'exercising initiative' means that we do things as we recognize a need, we also have to try and anticipate the impact of our actions (choice of programming constructs & style, methods of program organization, etc.) on other team-members. We have to avoid 'cleverness' for its own sake, and instead focus on producing code that will be intelligible to anyone who has to maintain or adapt it.

Effective teamwork requires effective communication. To communicate in a technical setting, we need to gather and present evidence, evaluate premises, and draw conclusions via logical deduction; in other words, we need to treat communication as a *forensic* exercise. Illustrating ideas with 'forensic evidence' is a powerful way of identifying ambiguity and differences in interpretation, and should therefore be the backbone of any technical discussion.

This paper addresses the practical question of how to program as part of a team, discussing programming traits that contribute to rather than detract from a team effort, and proposes 'forensic programming' as a vital technique for communicating between team members.

INTRODUCTION

Teamwork, Communication and Responsibility

Programming as part of a team is a tricky business. Programming is a solitary exercise, demanding great concentration and immersion, and one that invites us to obsess over technical minutia. At the same time, programming as part of a team effort is unavoidable: projects are generally too large for one person to handle alone, and require a distributed effort. In our drive to become 'technically skilled', we tend to accumulate obscure/esoteric ('advanced') programming tricks and constructs, and cultivate personal idiosyncrasies, and give little thought to what it really means to program as part of a team, what responsibilities team-membership carries and what compromises need to be made for the sake of team functioning. In short, personal motivations often run contrary to effective team functioning.

The idea that 'communication is vital to teamwork' is fairly universal, but the practical application of this concept is not. Have you ever had a QC programmer offer, by way of 'validation findings', the statement "I followed the spec and I don't match your numbers"? Or simply been presented as 'validation findings' a dump of unqualified PROC COMPARE output? Similarly, if a client questioned an unexpected or counterintuitive result in a summary table, would it be acceptable to simply say "I followed the SAP (statistical analysis plan), and that's what I got"? In each of these cases some expectation was not met, but nowhere near enough information was provided to define the expectation or to address root causes for the discrepancy. In order to be meaningful, any statement about discrepancies or unexpected results needs to be qualified by at least a certain amount of investigation.

The more rote aspects of team organization and leadership have to be supplemented with an acceptance of personal responsibility on the part of all team members. This personal responsibility includes two key things: recognizing and adopting traits that serve team functioning (taking into consideration what effect programming choices will have on other team members, and on the long-term viability of the project), and taking on the task of investigating discrepancies and unexpected results, to provide context to any discussion that takes place.

Even extensive project planning doesn't remove the need for interpretation and discussion. Not all contingencies can be anticipated (especially when it comes to data collection), and aside from the inevitable 'scope creep' that afflicts most projects, consider that study objectives and requirements exist as layers of abstraction: table shells (mockups) derive from a Statistical Analysis Plan (SAP) which in turn derives from a study Protocol, and within and between each of these layers there is the potential for ambiguity and the need for interpretation.

Programming activities tend to expose ambiguity, and are required to adapt and respond as interpretations are revised. This need for adaptability means that a primary motivation behind code development needs to be *ease of maintenance and revision*, reflected in intelligent program organization and design, or how we conceptualize, design, plan and execute programming tasks:

- Keeping things simple ('just because you *can* doesn't mean you *should*')
- Making program organization logical and predictable
- Recognizing and resolving ambiguity

These factors are important not just when we maintain and adapt our own code, but are vital to accommodating turnover in project teams. Too often we're confronted with the choice between trying to work with unintelligible 'inherited' code, or throwing this code out and reprogramming from scratch, and this usually results from individual programmers having not considered the possibility of someone else having to work with their code. Responsible team members need to take a long-term view of the uses to which code they develop might be put.

Forensic Programming

To support a particular interpretation of requirements, or to demonstrate the implications of competing interpretations, or to show how a particular result derives logically from source data (what we might refer to as 'results defense'), we usually need to present concrete evidence. The programming we use to assemble this evidence is what we refer to here as 'forensic programming'. We're co-opting the term 'forensic', which would normally refer to the methods of gathering and presenting evidence before a court in making a legal argument, but the term is appropriate in our context, since when we're engaged in 'forensic programming' activities we're attempting to present evidence that can lead to one and only one conclusion (where the aim is not, of course, to selectively present evidence in order to promote a given conclusion, but rather to present information that will eliminate ambiguity and rule out alternatives.) Forensic programming should be thought of as a communication technique, targeted at cultivating a common understanding of project data and objectives.

Forensic programming needs to do two things: first, it needs to gather and present the *minimum* information necessary to make a case, omitting anything that is extraneous or distracting, and second, it needs to eliminate potential confounding factors. When attempting to independently corroborate a specific result in a summary table, the biggest potential confounding factor is of course the analysis dataset that the table is based on: an outlier shown in a statistical summary could result from an outlier in the raw data, or it could be the unintended result of data manipulation in the derivation program, and for obvious reasons its important to be able to distinguish between the two.

HOW TO PROGRAM AS PART OF A TEAM

The concept of teamwork shows up in a lot of different contexts, and contains some common facets that can be dispensed with pretty quickly: a cooperative effort dictates that team members follow some basic rules, carry out tasks according to some general division of responsibility, and interact with other team members. With programming, each member of the team is expected to follow programming Standard Operative Procedures (SOPs), refer to a programming plan for general allocation of work, work on assigned tasks, and communicate progress and issues as necessary.

An equally vital aspect of teamwork is that of anticipating fellow team-members, and this is often where teamwork falters. In order to participate in the discussion and interpretation of (i.e.) protocol, SAP and table shells, all team members share responsibility for the 'critical reading' of study documentation. Just as project size and complexity dictate that programming be done as a distributed effort, the comprehensive and detailed understanding of every aspect of a project also needs to be distributed. This is why the 'project lead', while coordinating and providing consistency, should not *necessarily* have responsibility for specifications for all analysis datasets: the programmer who actually writes the derivation code for a given dataset will become the de facto expert on that aspect of the project. By accepting responsibility for reading and challenging project documentation, team members anticipate both the demands of their own roles, and the needs of other team members.

All team members also need to recognize the tradeoffs and potential impact of programming style and program organization. Programmers are not interchangeable 'resources', but show great variability in experience and knowledge, and even greater variability in how problems are understood and conceptualized. So *when programming as part of a team*, we need to strive to write 'universally understandable' code. This is served in part by providing explanatory comments in programs, but comments alone are not enough to ensure adaptable and maintainable code (explanatory comments will not necessarily compensate for awkward or confusing code, and all too often the scope of comments does not extend beyond the self-evident: "read in and sort data"). In striving for universally understandable

code, we need to try and make code 'self-explaining': the code itself should be clean, simple, explicit, and organized in a way that provides a clear narrative for the sequence of events taking place.

Keeping things simple: just because you *can* doesn't mean you *should*

There are alternate ways of solving virtually any problem, and while SAS® documentation tends to show us *how* to apply specific functionality, it doesn't often give us subjective insights on *when* to apply this functionality, or what the 'best' solution is in a given instance. Writing code for your own amusement, or to try out functionality, is much different from writing code as part of a team: the former serves as a pretext for trying out new ideas and exploring the limits of what's possible, while the latter very much involves avoiding unnecessary complexity and *getting the most out of the least amount of functionality*. When deciding on methodology, it's very important to challenge every aspect and ask 'is this really necessary?', 'is this the simplest way of achieving the desired end?', and 'would everyone else on the team have a reasonably easy time picking this up and working with it?'. Unless an acceptable balance between efficiency and 'human readability' is maintained, the world's most clever piece of code will be of less benefit to the team as a whole than one which achieves the same end while using simple constructs and common language elements.

Example 1: dynamically-generated lists versus static lists

Suppose you had to create a dataset by concatenating several raw datasets, where the raw datasets reflected multiple CRF pages belonging to the same domain. For instance, imagine a study with three Adverse Event CRF pages, for events with onset prior to treatment, during treatment, and during washout, where the raw datasets were named 'AEPRIOR', 'AETRT', and 'AEWASH'. By using a SAS metadata table ('sashelp.vtable'), you might imagine generating a list of raw datasets dynamically, and looping through this list in a 'set' statement to concatenate the datasets:

```
proc sql;
  select memname into :dsnlst separated by " " from sashelp.vtable
    where upcase(libname)="RAW" & upcase(memname) like "AE%"
  ;
quit;

data ae;
  set
  %let i=1;
  %do %until(%scan(&dsnlst,&i)=);
    raw.%scan(&dsnlst,&i)
    %let i=%eval(&i+1);
  %end;;
run;
```

Much simpler and more explicit code could of course achieve the same result:

```
data ae;
  set raw.aeprior raw.aetrt raw.aewash;
run;
```

, and this begs the question: under what circumstances should one or the other of these approaches be adopted? Just because you *can* use interesting language features (metadata table and macro loop), and even though this presents a more 'generalized' solution, it doesn't mean that it's the preferred method in every setting; the real question is whether or not there's any possibility that the list of raw 'AE' datasets might vary or change. If the list is predictable and static (and contains a relatively small number of elements), then there's really no justification for the more complicated approach. However, if the list has the potential to vary (for instance, if the 'treatment' and 'washout' datasets are only added as the study progresses, but won't exist in early data cuts, or if the code is to be adapted for a number of similar projects where there's some variability in CRF pages), then the more complex solution is probably also the more robust.

Highly 'generalized' code is not always desirable, partly because it's less readable and can obscure the sequence of events, and partly because it can contain pitfalls for blind use and adaptation. Consider for example taking the metadata approach shown above, and using it in a study that includes datasets 'AEPRIOR', 'AETRT', 'AEWASH' as well as 'AENDPT', where 'AENDPT' contains some sort of 'Alternative Endpoint' data completely unrelated to Adverse Events.

Example 2: multiple leading ampersands

Most people have little difficulty in visualizing how macro variables are resolved, since macro variables are in essence just a device for text substitution. If we have 3 macro variables &n1, &n2, and &n3 initialized with values 20, 20 and 40, respectively, then its fairly obvious that the statement

```
%put &n1 &n2 &n3;
```

will write the string "20 20 40" to the log. 'Secondary' substitution in macro variable resolution is a common and useful device, which can greatly reduce the number of statements required to complete a task. The construct essentially provides more generalized 'lexical meaning' to a set of tokens (placeholders), where specific meaning is captured iteratively depending on context. SAS implements this via multiple leading ampersands, and taking our three macro variables &n1, &n2 and &n3, it's fairly easy to envision how resolution takes place in the following loop:

```
%do trt=1 %to 3;
  %put &&n&trt;
%end;
```

In each iteration, the first 'pass' replaces &&n&trt with &n1, &n2 or &n3, and the second 'pass' replaces these macro variable with the values that they were initialized to, meaning that what is written to the log is as follows:

```
20
20
40
```

So the two leading ampersands provide for an additional layer of abstraction or generalization (&&n&trt is a generalized representation of &n1, &n2, &n3).

In the example above, the substitutions required to resolve the macro variable are few in number, meaning that it is fairly easy to visualize what happens simply by reading the fragment of code. However, it's probably fair to say that most people would *not* have an easy time visualizing what a macro variable like &&&&n&trt&x might stand for, and in fact, that programming problems that actually *require* more than 2 leading ampersands to solve are vanishingly rare to nonexistent. (There's an analogy to be made with architecture, where building design requires the application of only three dimensions, and while further dimensions exist in theory, their use in the blueprint for a building would be nonsensical.) Macro variables with more than 2 leading ampersands tend to show up in two contexts: first, where a programmer is deliberately trying to write complicated code (either for amusement or for the sake of competition, or from the misguided perception that the 'best' programs have the fewest possible statements), or where programs are poorly organized, often from a faulty understanding of objectives; and neither of these lend themselves to a team setting. A good rule of thumb is as follows:

> ***If you find yourself using more than 2 leading ampersands on a macro variable, try re-thinking your approach to the problem!***

Example 3: 'common' procedures for descriptive statistics and frequency counts

This is likely to be a contentious question, but what are the most commonly-used (and hence well-understood) procedures for calculating descriptive statistics (n, mean, median, etc.) and frequencies? It seems likely that they are PROC UNIVARIATE and PROC FREQ, though many programmers may champion alternatives, such as PROC MEANS and PROC SUMMARY. The question is, if there is a single procedure that meets a basic requirement (like calculating n, mean, median, min and max) and that the majority of programmers use habitually, what is the point to using a less-common procedure in the same context, and what is the impact on the rest of the programming team?

In the interest of anticipating fellow team members, it seems reasonable that we would try and use the procedures that the majority of other programmers would expect to see in the given context. The justification for *not* doing this would have to be if an alternative procedure offered a *substantial* improvement or efficiency gain (as opposed to just offering some feature that was 'kind of cool').

Example 4: tradeoffs associated with an expanded 'vocabulary'

One way of looking at the wealth of clever tips presented in conference papers, and new features promoted in each SAS release, is that they broaden the available programming 'vocabulary', and in much the same way that new language constructs and words broaden the vocabulary of spoken language. But like with spoken language, just because you *can* make use of a wide vocabulary doesn't necessarily mean that you *should*. In fact, doing so can pose certain tradeoffs and can ultimately be detrimental: the statements "perambulate to the market of merchantable commodities" and "walk to the store" mean basically the same thing, but while the latter will be universally understandable, the former won't, and will likely also cause great annoyance.

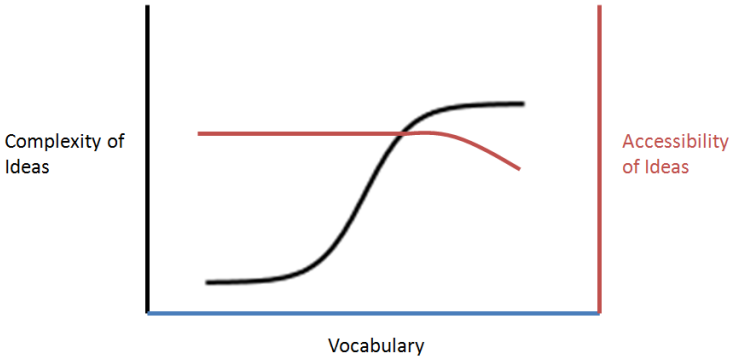

Figure 1. Hypothetical relationship between 'vocabulary' and the complexity of ideas that can be expressed, and the accessibility of these ideas.

As suggested in Figure 1, we might imagine that as vocabulary increases, our ability to express complex ideas also increases, *up to a point*. In the ascending portion of the vocabulary/complexity curve, we might say that the broadening of vocabulary is *necessary* in order to facilitate expression, but that where the curve levels off is where we are starting to say the same things in different ways, and the additional vocabulary may become pointless or even counter-productive. If we continue to apply additional vocabulary beyond this point, the 'accessibility' of ideas (the 'universality', or the number of people who will easily comprehend the idea) may actually diminish, and this among ideas that would otherwise be perfectly understandable if stated in simpler terms.

Additions to programming vocabulary can offer more, and sometimes more refined, ways of doing the same thing, and they can also provide new and enhanced functionality, and solutions to previously unsolvable problems (or at very least a dramatic improvement in efficiency). It's probably safe to say that that PROC REPORT, especially in conjunction with ODS, falls in the latter category. Few people would say that PROC REPORT/ODS offers nothing that a data-_null_-based reporting system wouldn't also offer, or at very least that PROC REPORT doesn't offer a significant gain in efficiency.

There are other areas though that fall in the first category (that simply offer alternative ways of doing the same thing), and it's probably a good idea to challenge these with a healthy dose of skepticism and a "devil's advocate" approach: do they offer a *significant* improvement in efficiency within a given context (a significant reduction in the number of statements required to perform a ask, and *without* compromising 'human readability')? If the answer is 'no', then you might say that there's no justification for 'modernizing', especially if there's the risk that other members of your team don't share your awareness of the broader vocabulary.

Take simple string concatenation as an example:

```
c=trim(left(a))||trim(left(b));
```

This might seem a bit clumsy, with an improvement being offered by

```
c=strip(a)||strip(b);
```

or

```
c=CATS(a,b);
```

A devil's advocate would acknowledge that the last solution is the most attractive and concise, but would have to ask whether the improvement qualified as 'significant': was there a substantial reduction in the number of statements, and are the solutions equally self-explanatory? The 'efficiency gain' actually looks fairly negligible, and awkward though it may be, the first solution is also very obvious, which is an advantage given the range of knowledge and experience that may exist among programmers who end up having to adapt or maintain the code.

Regular Expression (REGEX) functions can fall into both categories: in problems that *require* that 'abstract strings' be defined, REGEX functions provide the only truly robust solution, but if they are used in contexts where simple text matching would suffice, then they serve as nothing but an alternate (and much more confusing) way of doing the same thing.

Take for example a program designed to read SAS programs and identify dataset references. This program would have to be able to identify a wide range of strings, such as

```
data adsl;
set derived.adsl;
proc sort
       data=derived.adsl out=adsl;
DATA derived.ADAE;
proc univariate data=adsl;
create table TEMP as select var1, var2
    from adsl
(etc.)
```

These are all instances of what you might think of as an 'abstract string', one that is case-insensitive and (ignoring the SQL sentence), contains the words 'set', 'proc' or 'data', possibly an '=' sign after the word 'data', with any number of blank spaces, possibly a procedure name, any number of carriage returns, and a dataset name that may or may not reference a data library. The point is that it would be difficult if not impossible to capture all imaginable combinations via simple text matching.

Instead, an abstract string (or a collection of abstract strings) would have to be defined that would capture all possible permutations. Just to illustrate, and without getting into a discussion of regular expression syntax, consider that the string "`set derived.data`" might be represented as an abstract string by the regular expression

```
(^set|\sset)\s\S+\.\S+
```

This translates as "a string starting with the word ‘set’ (or the word ‘set’ preceded by any number of blank spaces), followed by any number of blank spaces, followed by two words separated by a ‘.’ character, followed either by the termination of the string or by any number of blank spaces". In order to flag 'set' statements while parsing a SAS program (using case-insensitive matching, and targeting permanent datasets as opposed to work datasets), you might do the following:

```
id=PRXPARSE("/(^set|\sset)\s\S+\.\S+/i");
flag=PRXMATCH(id,response);
```

This approach is going to be totally unintelligible to anyone who has not previously worked with regular expressions, but is unavoidable in programming tasks that require text matching against abstract strings.

In contrast, imagine a problem that can be solved with simple text matching, like where a variable contains a small and finite number of possible values:

```
if compress(upcase(response))="YES" then code=1;
else if compress(upcase(response))="NO" then code=2;
else if compress(upcase(response))="UNK" then code=3;
```

While this can also be solved using regular expressions:

```
return=PRXPARSE("/\s(YES)\s|\s(NO)\s|\s(UNK)\s/i");
match=PRXMATCH(return,response);
code=PRXPAREN(return);
```

, this approach offers no particular efficiency gain in the given context, and in all likelihood will confuse the heck out of anyone who has not previously worked with regular expressions. Smart programmers, who are unfamiliar with regular expressions but end up inheriting this code, should probably throw it out and re-program using methods that they're comfortable with (i.e. simple text matching), especially where time is limited; continuing to use and/or adapt this code without first acquiring a reasonable understanding of regular expression syntax would be a risky proposition. Responsibility, though, really lies with the original author of the code, who should be making judgments about the necessity of the approach in the first place, and recognizing the possible implications of choosing methodology that is unnecessarily complicated for the given task.

Ultimately, programming as part of a team requires that we be critical of our own work: there's no excuse for having complicated solutions to simple problems, and quite often excessive complexity is nothing more than a symptom of unclear thinking and poor program organization.

Making program organization logical and predictable

Organizing code in a logical fashion serves two purposes: first, it minimizes the number of statements required to meet functional requirements, and second, it places functionality in predictable locations (i.e. so that you know where to look when modifying your own code, or when adapting someone else's). In general, programs that are organized logically tend to be 'economical' (they do much with very little) and yet easy to read, and programs that are 'dense' and monolithic are difficult to read and often also display poor program organization.

Not everyone shares the same concept of 'logical organization', but there are a few basic and obvious principles that can be applied. First, derivation hierarchy should be respected. This means that 'lower-level' derivations should not be repeated as part of 'higher-level' derivations: code for imputing partial AE onset date should not be repeated as part of the derivation of AE duration. Similarly, 'higher-level' derivations should not be implemented and then 'qualified' later on in the program. Consider where a per-protocol population flag is initially derived based on a protocol deviation dataset, and then, further on in the program, a flag is created for 'patients having a screening lab draw', and this flag is then used to further refine the per-protocol population flag. The order of derivation implementation should follow the dependency hierarchy; otherwise, maintenance of these higher-level derivations will be a frustrating exercise, and derived datasets will tend to suffer from a lack of internal consistency.

Second, data sets should be oriented (i.e. as 'long/narrow' or 'short/wide') according to context; 'by-group' processing on a coding variable (i.e. a long/narrow dataset which we would say follows 'basic data structure') is usually preferable to repetitive processing on a horizontal set of variables (too often we see a snarl of macro code that could have been avoided by a simple transposition step).

Third, wherever possible, major tasks should not be interleaved, but should be implemented as discrete, sequential steps, placed where most people would expect to find them. For instance, a program that creates a summary table should follow the general pattern of *data retrieval and filtering*, followed by *data summarization*, followed by the assembly of a *report-ready dataset*, followed by *reporting*.

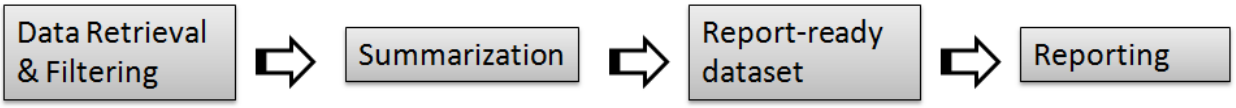

Figure 2. Discrete, sequential steps in creating a summary table.

If a program intersperses these tasks (especially if additional data retrieval and filtering takes place after data summarization) then it is much more likely that maintenance and adaptation of the program will miss essential sections of code. For example, consider a table program that does the following:

```
proc sort data=derived.dsn1 out=dsn1;
  by usubjid;
  where saffl=1;
run;
proc freq data=dsn1 noprint;
  table var / out=freq1;
run;
data final1;
  set freq1;
  labl="Frequency count 1";
run;

---(etc.)---

proc sort data=derived.dsn2 out=dsn2;
  by usubjid;
  where saffl=1;
run;
proc freq data=dsn2 noprint;
  table var / out=freq2;
run;
data final2;
  set freq2;
  labl="Frequency count 2";
run;
```

Notice that the major tasks of data retrieval and filtering, calculation of summary values, and creation of 'report-ready' data are repeated in non-adjacent sections of the program (what we refer to as 'interleaving' of tasks: data retrieval/filtering followed by data summarization, followed by further data retrieval/filtering, followed by further summarization, etc.). This may seem trivial in this simplified example, but applying this practice in a 'real world' setting could mean that the primary filter '`where saffl=1`' is repeated in blocks of code that are separated by dozens or even hundreds of lines, greatly increasing the probability that any required revisions to this filter will not be consistently implemented.

Lastly, fragmentation should be minimized. Always be aware that someone else may end up adapting or maintaining your code, and this is often made much more difficult if tasks have been unnecessarily separated into macros (in-line macros, if they interrupt and obscure program flow, or external macros that have to be tracked down in a macro library). Sure, external macros potentially reduce the size of the programs that call them, and can reduce code 'redundancy', but they are *not* universally advantageous and should be used sparingly: the tradeoff is that they obscure the sequence of events in program execution, often require an understanding of semantically-meaningless parameter names (&var1, &var2, etc.), and often involve nested (and hence confusing) macro calls. In a lot of cases it is preferable to accept a certain amount of code redundancy in exchange for meeting functional requirements in the simplest and most obvious way possible.

Unfortunately, macros like the three examples below show up with surprising frequency:

```
%msymput(var=myvar);
```

It's hard to see what advantage this might offer over the single statement required in base SAS:

```
call symput("myvar",myvar);
```

Any additional defensive code that '%msymput' might offer would probably be better placed in open code, in the datastep initializing the macro variable, where it would be easy to interpret and modify.

```
%msort(dsn=%str(derived.adsl),vars=%str(usubjid),keepvars=%str(usubjid age));
```

Again, compared to the open-code equivalent

```
proc sort data=derived.adsl out=adsl(keep=usubjid age);
  by usubjid;
run;
```

, there's no obvious efficiency gain, and the macro does little other than interrupt what would otherwise be perfectly 'human-readable' code. The same is true of the following:

```
%macro mexistd(dsname);
  %global exist;
  %if %sysfunc(exist(&dsname)) %then %let exist=YES;
  %else %let exist=NO;
%mend mexistd;
```

, where the time spent learning how to use the external macro would be much better directed toward Googling 'how to tell if a SAS dataset exists' and making use of '%sysfunc(exist())' directly.

Aside from these basic principles of code organization there is one obvious but often-overlooked concept that deserves to be stated, which is that *aesthetics matter*. Virtually every programming department in every company has some sort of 'good programming practice' guidelines, which stipulate among other things the consistent use of lower-case in code writing, the consistent use of indentation, and the 'one statement per line' rule. In spite of this, code like this

```
DATA DM3; set dm2;
if a=b then
        DO;
 code="Y";coden=1;
c=a;
```

shows up with surprising frequency. Violating basic aesthetic principles can make otherwise well-organized code excruciating to read, and following these same aesthetic principles can go a long way toward redeeming code that is otherwise not terribly well-organized.

Recognizing and resolving ambiguity

Ambiguity is an interesting phenomenon: it is problematic, but also derives from the 'efficiency' of natural language (where the same words and phrases can be used to express multiple meanings), and from our tendency to express things figuratively. We all tend to 'get' the concept of ambiguity, but have tunnel vision when it comes to interpreting ambiguous instructions (we latch on to our own interpretations and fail to recognize alternatives), and it's probably fair to say that the vast majority of validation and review findings come not from programming errors (syntactical errors that SAS automatically defends against reasonably well) but from conflicting interpretations of ambiguous instructions.

It's possible for ambiguity to exist in 'formal' language such as SAS, but language interpreters are usually designed to identify and warn against at least some types. For instance, when SAS encounters a 'multiple-merge' step, where multiple instances of 'by' variables exist on two or more datasets, it is ambiguous which records are to be matched and SAS issues a note to the log. SAS does not, however, always warn against syntactic ambiguity, like when there are ambiguous conditionals having 'and' and 'or' clauses combined without parentheses:

```
if a and b or c;
```

This could be interpreted as

```
if (a and b) or c;
```

or

```
if a and (b or c);
```

, yielding quite different results. While it is possible to construct conditionals like this in a way that anticipates how SAS will resolve the ambiguity, it's far more human-readable, and hence safer, to use parentheses to make the intended meaning explicit (few people, author included, will know automatically how SAS interprets a condition like 'if a and b or c').

Ambiguity is also permitted via constructs like 'proc sort nodupkey':

```
proc sort nodupkey data=dsn;
  by usubjid date;
run;
```

, since record selection within each combination of 'by' variables is for all intents and purposes arbitrary. Often this doesn't matter (like if the requirement is literally to capture unique instances of usubjid and date), but if the intent is to capture some associated value (like maximum blood pressure on each date) then this will cause problems: either the wrong value will be selected outright, or the correct value will be selected but only because of some implied condition (like that the data set was sorted by usubjid, date, and blood pressure in a previous step).

As we develop as programmers, we tend to learn to avoid ambiguous constructs and implied conditions in program code. In most cases the ambiguity we encounter, and that which poses the greatest challenge, is in natural language. Consider for example a statistical summary that gives a frequency count of patients who *acquire* a given condition. Use of the term 'acquire' carries a couple of implicit assumptions: it suggests a temporal element probably consisting of the duration of study conduct (i.e. 'acquire during the study'), and it implies that the condition did not exist at some point in the past. To cite a real-world project as a 'case study', both the validation programmer and the client assumed the count was simply 'the number of patients who display the given condition', partly on the strength of exclusion criteria which were supposed to prevent patients with the condition from being enrolled. The production programmer took a very literal interpretation, that being 'the number of patients who did not have the condition at or prior to baseline, but developed the condition post-baseline'. As it turned out, a couple of patients had actually been enrolled into the study in violation of exclusion criteria, meaning that the number of patients *having* the condition was higher than the number of patients who *acquired* the condition during study conduct.

Ambiguity forces us to make judgments and decisions in the face of incomplete information, and often there is no 'right' answer. There is usually a definable set of alternatives though, and cultivating the ability to recognize these alternatives is vital to resolving discrepancies. As with other aspects of teamwork, the ability to recognize and resolve ambiguity only develops with time and with an investment of effort, but key to developing this ability is the cultivation

of language skills, first and foremost by reading and writing (this is why everyone should make the effort to write conference papers!).

The most direct insight into ambiguity is often yielded by taking a perfectly literal interpretation of each instruction: consider the literal connotation of each word, and then test the effect of alternate and/or less literal meanings. For example, the literal meaning of the word 'acquire' is "to come into possession or ownership of", but the 'spirit' behind the phrase 'patients who acquire' (in other words, the figurative intent) might simply be 'patients who have'. The presence of these alternative interpretations defines an ambiguous context, which we could resolve by nailing down the intended connotation of the word 'acquire'.

FORENSIC PROGRAMMING

Given random probability, it's easy to envision how a 'correct' answer might arise purely by chance, or how an answer might be correct, but for the wrong reasons. It's also easy to envision how two equally defensible treatments of the same data might produce different results. This is why, as we recall from high school math classes, it's never enough to simply state an answer: we also need to show 'how we got there'. This principle really needs to be applied to QC and validation, but quite often what a validation programmer presents as 'QC findings' is a statement along the lines of "I followed the specification and I got different results from you", sometimes with a dump of PROC COMPARE output offered as corroboration. Any time a result is challenged, it must be done so against expectations, and the expectations must derive from premises and deduction. The 'evidence' cannot simply be an illustration that a discrepancy exists, but should also show what the *expectation* is, based on source data and analytical rules.

QC activities should not place the onus on the production programmer to defend results. Consider validation in a 'legal' context: the production output is 'the accused', challenged by validation which serves the role of 'prosecution'. We would never say that the accused is required to prove innocence, but rather that the prosecution is required to prove guilt. The same holds with QC: when discrepancies arise, the burden of proof lies with the QC programmer, who needs to present a case proving the correctness of QC results over production results (a practice which can quite often expose errors on the QC side).

This paradigm, placing validation in a 'legal' context, should play a role in project planning, specifically in making sure that project budgets don't under-represent the cost of QC. QC is often naively placed in a subordinate role to 'production' programming, with cost templates that show things like "6 hours to program a table, and 2 hours for QC". This might seem plausible, given that production code needs to produce formatted/'word-processor-ready' output and under the idealized notion that QC is a mere formality conducted against air-tight specifications, but it doesn't really allow for the role QC should play in addressing ambiguity and differences in interpretation, and in the investigation of discrepancies.

As a QC programmer, once you have identified a discrepancy it is your responsibility to state precisely and succinctly why it is that you think the value you arrived at is correct, using evidence assembled via forensic programming. The forensic code doesn't have to be fancy (in fact, its better if it isn't), but it does have to be 'complete' and self-contained. This means that it has to corroborate a specific value, and has to do so without introducing confounding factors: forensic code that runs a 'proc print' on dataset 'AE15' (some arbitrary work dataset created mid-way through a table program) will not be complete or self-contained since it accepts as confounding factors all the prior steps that led up to the creation of dataset 'AE15'. In contrast, forensic code that runs a 'proc print' on the 'ADAE' analysis dataset, with a 'where' clause that selects a target set of records that go into a specific frequency count, would qualify as complete and self-contained.

Starting with a very simple example, imagine a disposition table including a count of randomized patients: the production output shows a total of 100 randomized patients, and the QC program gets a count of 99 (lets say the QC program uses the criterion that randomized patients have a non-missing date of randomization). Evidence can be presented via a miniscule fragment of forensic code:

```
proc sql;
  select count(distinct usubjid) from derived.adsl where randdt>.;
quit;
```

Independently of anything else that goes on in the QC program, the count of 99 patients can be derived directly and explicitly from the source data; the criteria can then be easily checked against possible alternatives (its not uncommon with interim/dirty data cuts that a patient might be randomized and assigned to a treatment group but show a missing randomization date).

In an example as simple as this, you might not literally send the production programmer the fragment of code along with the output, but might instead just say "I get a count of 99 as the number of patients with non-missing randomization date". But overkill (providing all the 'forensic evidence' even in a simple matter) is still preferable to saying "my count doesn't match - check your program".

As a more involved example, consider an adverse event (AE) table giving patient counts by body system and preferred term, where the production and QC output show matching counts, but show differences in percents. It's unlikely that the discrepancies would arise from syntactical errors, or that they would arise from confusion over how a percent is calculated (and since the counts match we know that fault doesn't lie with the numerators); it is more likely that the discrepancies result from differences in record-selection criteria when calculating denominators.

With AE tables, denominators are usually simply the count of patients in the given analysis population in the given treatment group. For illustrative purposes, we'll chose something slightly more complicated, and say that for each preferred term, the denominator is the number of patients in the given analysis population and treatment group having at least one AE within the given body system. This example is admittedly a bit contrived, but is intended to illustrate a point, namely that it's not always possible to 'internally corroborate' values within a table: in this case the denominator used in the percent calculation is not actually displayed anywhere in the table, and it's this hidden value that needs to be investigated. The same problem arises in certain types of shift tables, where the denominator in a percent calculation at each time point is usually the number of patients providing both a baseline and post-baseline value, but this denominator is not necessarily reported in the table.

With forensic programming, we want to reduce a problem to its more manageable constituent parts, so when building a case, the first thing to do is to isolate a specific discrepancy: select a specific 'cell' (combination of treatment group, body system, and preferred term) where a discrepancy occurs:

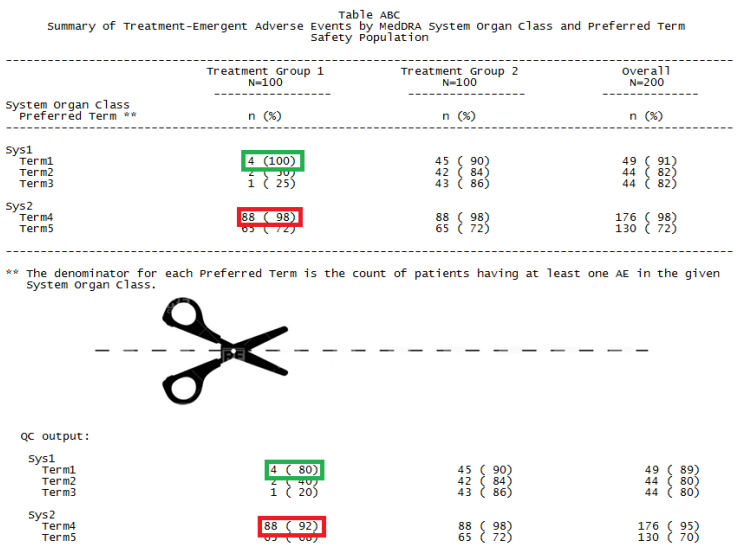

Table ABC
Summary of Treatment-Emergent Adverse Events by MedDRA System Organ Class and Preferred Term
Safety Population

System Organ Class Preferred Term **	Treatment Group 1 N=100 n (%)	Treatment Group 2 N=100 n (%)	Overall N=200 n (%)
Sys1			
Term1	4 (100)	45 (90)	49 (91)
Term2	2 (50)	42 (84)	44 (82)
Term3	1 (25)	43 (86)	44 (82)
Sys2			
Term4	88 (98)	88 (98)	176 (98)
Term5	65 (72)	65 (72)	130 (72)

** The denominator for each Preferred Term is the count of patients having at least one AE in the given System Organ Class.

QC output:

Sys1			
Term1	4 (80)	45 (90)	49 (89)
Term2	2 (40)	42 (84)	44 (80)
Term3	1 (20)	43 (86)	44 (80)
Sys2			
Term4	88 (92)	88 (98)	176 (95)
Term5	65 (68)	65 (72)	130 (70)

Figure 3. An example Adverse Event table and corresponding QC output, showing discrepancies in calculated percents. When the same discrepancy pattern appears to exist across several table cells, the ideal place to start is with a cell that contains a small count.

Preferably, select a cell that will yield a small count (the discrepancy in 'Term1' and 'Treatment Group 1', rather than that in Term4, in Figure 3 above) since this will keep ad hoc data listings small and manageable. (Of course, this ideal doesn't always exist, meaning that the investigation of specific discrepancies may need to use electronic file comparisons (PROC COMPARE or 'diff' utilities), but it's convenient when the number of records involved is small and can simply be listed via PROC PRINT.)

With table QC, where tables are presumably based on validated derived datasets, its less important to step back to the raw (or SDTM) data than it would be with 'results defense' following client review, but where practical its usually a good idea to use forensic programming as a secondary check on derived data, and eliminate derived data as a potential confounding factor. Having selected a specific discrepancy to dissect, query the source data in order to capture the records that went into your summary value:

```
proc sql;
  create table temp as select distinct usubjid, aebodsys, aepterm
    from sdtm.ae
    where usubjid in(select usubjid from derived.adsl where saf=1 & trtn=1)
    & aebodsys="sys1"
    /* & compress(aepterm)^="" */
  ;
```

```
  select count(distinct usubjid) from temp
  ;
quit;
proc print data=temp;
run;
```

Note that this code fragment is minimalist, performing no unnecessary tasks and capturing no extraneous information, and self-contained, in that it explicitly and concisely applies all record-selection criteria related to the selected summary value (the relationship between source data and output is explicit, and not obscured by intervening steps or work datasets). Translated into plain English, the query captures all unique combinations of patient ID, AE body system and preferred term, among patients in the safety population and in treatment group 1, for body system 'sys1' (and then from within that record set, counts the number of patients represented, which serves as the denominator for preferred terms falling under the given body system).

The output would look something like this:

```
5
-----

usubjid          aebodsys              aepterm
-----------------------------------------------
111              sys1                  term1
111              sys1                  term2
222              sys1                  term1
333              sys1                  term1
333              sys1                  term2
333              sys1                  term3
444              sys1                  term1
555              sys1
```

The 'case' that the QC programmer might present would consist of the 'forensic evidence' shown above: the plain English statement of record-selection criteria (possibly supported by the code fragment itself), followed by the count of patients that went into the denominator and (provided it is not too extensive) the complete list of records from which the denominator was drawn. The record-selection criteria (query) would represent the *premises* to the argument ('if a and b …'), and the listing output would represent the *conclusions* ('… then c'); the conclusions can then be challenged by challenging specific premises (for instance, whether or not records should be excluded where preferred term is missing).

A useful side-effect of this exercise is to highlight deficiencies in the raw or source data: in this case the discrepancy arose from incomplete AE coding. The listing of records makes it very clear where differences in assumptions lay, and presents a matter of interpretation that needs to be raised.

Well-constructed forensic code should allow you to show what effect revised assumptions and alternative interpretations have, and this often highlights the root cause of a given discrepancy. In an ideal world, we like to pose questions only where we can also offer some plausible explanation. In non-forensic QC, we might simply say "our denominators don't match", where with forensic QC we can offer a possible solution at the point where we identify the discrepancy: "our denominators don't match, but if I exclude records where preferred term is missing, I think I can match your numbers".

As a last example, consider again the problem of identifying patients who 'acquire' a given condition as opposed to those simply having the condition. Identifying patients who have a condition doesn't require any temporal evaluation:

```
proc sql;
  create table having as
    select distinct usubjid from derived.dsn where cond='Y';
quit;
```

The set of patients acquiring a condition, on the other hand, consists of patients not having the condition prior to a given reference date, but having the condition after the reference date:

```
proc sql;
  create table acquiring as
    select distinct usubjid from derived.dsn
    where
```

```
      usubjid ^in(select distinct usubjid from derived.dsn
                  where cond='Y' & date<refdate)
     & usubjid in(select distinct usubjid from derived.dsn
                  where cond='Y' & date>=refdate)
  ;
quit;
```

Patients who acquire the condition are a subset of those who have the condition; if absence of the condition is a study inclusion criterion, there will be considerable interest in identifying the specific patients who have the condition but did not acquire it during the study (the sponsor will probably want to know which site(s) enrolled the patients, and may want additional details on those patients). Forensic code can aid considerably in these sorts of investigations; for instance, having already identified patients with the condition as well as the subset of patients having acquired the condition, it takes little work to identify the patients of interest, being those that were apparently enrolled in violation of exclusion criteria, or the 'having' *relative complement* of the 'acquiring' set:

```
proc sql;
  select distinct usubjid from having
    where usubjid ^in(select distinct usubjid from acquiring)
  ;
quit;
```

The amount of supporting information to present via forensic coding is discretionary, and depends on the complexity of the problem. Simple criteria such as 'patients with non-missing randomization date' don't generally require supporting evidence in order to 'make a case', but more complex criteria, and problems involving multiple data sources and secondary inferences often do. Consider for example the task of calculating the cumulative duration of hospitalization attributed (indirectly) to a specific class of adverse event (such as Anemia). The first step would probably be to identify all patients experiencing the given adverse event, and then for each patient, for each instance, to identify hospitalizations having hospitalization date on or after adverse event onset date; for each patient, the duration of these hospitalizations would then be summed.

Since there are several steps involved in the calculation, there are several possible origins for any discrepancies identified in summary results. These steps also serve as logical break-points for reducing the problem to discrete, manageable chunks, and for presenting evidence: list patients having the given event type, then list hospitalizations for those patients, indicating the hospitalizations starting on or after AE onset date, then calculate hospitalization duration per hospitalization, and finally calculate cumulative hospitalization duration per patient. This might seem like a fair bit of work, but it's necessary in order to isolate the specific step in the calculation where a discrepancy arose; habitually taking this systematic approach will on average save time over alternatives like 'brute force' code review.

More importantly, this systematic, forensic approach provides a vehicle for communication between QC and production programmers. Simply describing in general terms the sequence of events in a complex calculation, or quoting program code out of context, provides very weak evidence in support of a summary value and makes it hard to imagine where approaches to the calculation might have diverged; what is actually required is concrete and irrefutable evidence. Forensic programming can be simple or complex, but its usefulness is measured by the clarity with which premises can be stated, and the ease with which supporting evidence can be interpreted.

CONCLUSION

There is a truism behind every problem and every question, which is that "everything is obvious once you know the answer". This is what often leads us to overestimate the ease with which other programmers will make sense of code we author (what seems obvious to one programmer can be totally unintelligible to another), and to underestimate the requirements of effective communication.

In code writing, this truism often invites us to violate another, which is that in a team setting, and with code that is central to analysis and reporting (and which may have to be maintained or adapted by other programmers), *solutions should not be provided for non-existent problems*. For instance, highly generalized code, which anticipates a wide range of conditions, should only be applied where the potential for a wide range of conditions actually exists; and simple problems should have simple solutions, and should use common language elements rather than 'advanced' functionality (REGEX functions shouldn't be used where simple text matching is sufficient). Programmers should by all means learn about and experiment with advanced functionality, and share code that illustrates practical application, but in a project setting care should be taken to balance the tradeoff between complexity/ingenuity and ease of maintenance, with particular consideration given to the wide range of knowledge and experience that may exist among other team members.

Programming is essentially the act of translation between natural and formal language, with QC discrepancies arising out of differences in how this translation is implemented. As with any language, things tend to get 'lost in translation' when we try and address root causes of a discrepancy: concepts that are perfectly clear in our own minds get muddled when we try and express them, and this can present in our teams as a 'lack of effective communication'. 'Forensic programming' offers a powerful device for bridging this gap, since it emphasizes breaking problems down into discrete, manageable steps, and gathering evidence to support logical deduction. Programming tasks that are difficult to describe verbally, where we tend to resort to a lot of hand-waving and presenting code fragments and data out of context, can be discussed in concrete terms if we take a rigorous forensic approach, making a quasi-legal case by isolating confounding factors, and describing expectations via complete, self-contained, minimalist code blocks.

ACKNOWLEDGMENTS

I would like to thank inVentiv Health, and Colleen Benjamin in particular, for providing encouragement and supporting my PharmaSUG participation, as well as my family and my many friends and colleagues at inVentiv and elsewhere.

CONTACT INFORMATION

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

Name: Brian Fairfield-Carter
Enterprise: inVentiv Health
E-mail: fairfieldcarterbrian@gmail.com