PharmaSUG 2013 - Paper MS07

If ‘standard code’ is so great…

Brian Fairfield-Carter, ICON Clinical Research, Redwood City, CA

ABSTRACT

Since CROs traditionally employ ‘lean’, ‘do more with less’ project teams, their use is increasingly attractive to sponsors as a cost-trimming solution. In the analysis and reporting world, the push for ever-greater efficiency typically focuses on the concepts of standardization and code re-use, and specifically on ‘standard macro libraries’. But efficiency is an elusive concept: anyone who has performed code maintenance knows that gains in processor time can easily be dwarfed by issues of ‘human readability’, and that modifying code can turn out to be more time-consuming and error-prone than writing code from scratch; standard macros can be restrictive and involve a considerable learning curve.

A review of conference papers dealing with ‘standard code’ illustrates a consistent thread: standard macro libraries, at least those involved in analysis and reporting, exist almost exclusively in the domain of pharmaceutical companies, and are virtually non-existent in CROs. This begs the question: if ‘standard code’ works in certain contexts, and seems intuitively to offer efficiency gains, why does it not seem to hold greater prominence in the CRO world?

This paper seeks to catalog the tradeoffs inherent to code libraries, and to challenge the conventional dogma that standardization inevitably produces efficiency/accuracy gains, in exploring the near absence of macro libraries in the CRO world. This discussion then provides a backdrop for proposing what may be a more workable approach to code re-use: one that focuses on planned code adaptation, and on code organization and architecture, where code writing deliberately anticipates future adaptation.

INTRODUCTION

While claims of ‘increased efficiency and accuracy through the use of standard macros’, and the suggestion that ‘fully validated’ macros can reduce long-term validation overhead, sound both impressive and plausible, it’s not clear that they’re necessarily true. If ‘standard code’ is purported to make things run more smoothly and improve our bottom line, why is its use not more universal? What are the logistical problems that limit implementation and use? What are the circumstances in which ‘standard code’ might actually be counter-productive?

In interviewing prospective employees, it’s interesting to (informally) poll their experiences with ‘standard macro libraries’ (the use of, contribution to, and opinions on the effectiveness of). While providing an anecdotal rather than ‘systematic’ survey, the overall conclusion is nonetheless that among candidates coming from the CRO world, experience with standard macros tends to be restricted to the use of sponsor-provided code, and among candidates having any experience with standard macro libraries, opinions do seem to be pretty consistently negative: “it’s a pain and doesn’t work very well”, “wastes a lot of time and is difficult to learn how to use”, “requested updates take a long time”, “documentation is out of date and hard to understand”, etc. This makes one wonder, is it because of mere pride and not liking being ‘told what to do’ that ‘power SAS users’ almost universally loath standard programs, or can such code actually have a damaging and inhibitory effect - one that actually contradicts the supposed benefits?

A search on Lex Jansen’s Homepage (http://www.lexjansen.com/) for conference papers dealing with various permutations of ‘standard macro library’ highlights a couple of (probably self-evident) things: SDTM transformations are represented where ADaM conventions are not, and virtually none of the companies represented are CROs. Perhaps neither of these observations is particularly impressive, but just illustrates what we know intuitively: first, SDTM rules are more rigid than ADaM rules, since ADaM rules need to accommodate diverse research objectives, and this in turn means that SDTM rules are much more amenable to implementation in standard code. Second, pharmaceutical companies have the luxury of defining their own standards while CROs have to respond to a diverse array of client requirements. There is a third and possibly less obvious observation that can also be made: discussion of standard macro libraries is usually prefaced with the assumption that they improve efficiency and lower costs, but empirical evidence that would support this assumption isn’t offered (I have yet to come across a paper that actually attempts to quantify these efficiency gains). The question remains whether efficiency metrics would necessarily lead to the same conclusion, and whether the factors that contribute to efficiency and accuracy are even quantifiable, especially where such factors are as diverse as they are, and involve such complex tradeoffs.

Although there is a notable absence of CRO industry representation among ‘standard macro library’ conference papers, that doesn’t mean the aspiration doesn’t exist in the CRO world. On the contrary, speaking from experience, the objective is perpetually ‘on the books’ as a long-term but ultimately unrealized goal. The reasoning that is typically proposed is that reporting requirements are to some extent consistent between studies (particularly studies belonging to the same sponsor), and that as such we should be able to write a set of ‘standard programs’ that can be applied across similar studies. We seem to instinctively accept the desirability of ‘standard code’, and hence most programming job descriptions make some reference to ‘standard macros’, as do most resumes, and yet when it comes down to it, implementation is notably lacking among CROs.

Why is there this contradiction then? If standard analysis & reporting code seems to be feasible, according to the pharmaceutical companies, and if CROs tend to cite the development of standard code libraries as an efficiency-producing objective, then why don’t CROs tend to have such code libraries? It would seem that the problem really lies in determining what form such a library of re-usable code should take in a CRO setting.

STANDARD CODE AND CODING STANDARDS

To start off, we need to consider the various connotations the term ‘standard code’ can take. In the most rigorous instance, macros exist as inaccessible ‘black boxes’ where updates only take place though a very stringent change-control process. To allow greater flexibility, standard macros may be ‘demoted’, where copies are modified to meet specific requirements at (for example) the level of individual studies. Standard code can also be viewed as that which meets a standard set of requirements, so we often see what may be more aptly referred to as ‘template’ programs, adopted as ‘standards’ but copied and modified within individual studies.

‘Black box’ code is often treated as a ‘validated system’, meaning that validation is assumed to have been rigorous enough that the intended results are attained regardless of context; re-validation is only required when code is updated to meet revised functional requirements. ‘Demoted’ code on the other hand, and any code not developed as ‘black box’ code is generally understood to require the same validation procedures as would single-use code. Validation needs to be repeated any time that context changes.

Because in each instance some manner of code alteration (maintenance or adaptation) is potentially required, probably the most important attribute of programs falling into any of these interpretations of ‘standard code’ is that they must adhere to some form of ‘coding standards’, alternately referred to as ‘good programming practice’. For example, a definition of ‘good programming practice’ might read as follows:

- Use the standard header at the top of each program
- Use of the tab character is prohibited – use spaces instead
- At minimum, comments are written in a program to identify/separate logical blocks of code, to add information that is not easily discernible by reading the code and/or to simplify understanding when code is complex
- Data steps & procedures should be separated by a blank line(s) and end with a run or quit statement
- All SAS procedures include a DATA= statement to identify the dataset which is being processed
- Regular and consistent indentation is used:
  - On all lines between the data/proc statement and the conclusion of the data step/procedure
  - Within do loop, select statements
  - For any other situation in which consistent indentation will improve readability
- An OUT= statement is used when sorting permanent datasets to avoid overwriting

Coding standards of this nature essentially relate to making code human-readable, but scope tends to be fairly limited: little or no attention is given to ‘architectural’ considerations like the grouping of operations & declarations, or the separation of data retrieval, summarization and reporting tasks, but it is these factors that will limit the ability of someone other than the author of the code to identify specific functionality. (Consider, for example, the kind of architectural standards that would probably apply to the source code behind an operating system). And it is, in turn, the ability of programmers other than the original author to maintain code that makes for truly workable ‘standard code’ regardless of which connotation of the term is being applied.

PREDICTABILITY OF REQUIREMENTS

If coding standards define the maintainability of code, then predictability of requirements really governs how successful an attempt at standardization is likely to be; this is why standard macros tend to accumulate around SDTM transformations rather than around ADaM conventions. Consider for example how much variability exists between projects even in something as mundane as the derivation of treatment emergence: rules for imputing partial dates are highly dependent on individual study design, as are rules for defining ‘treatment windows’. Bottom line is that derivations that are by their very nature dependent on study design should probably not be placed in the guise of ‘standard code’, since this can create the illusion that code is ‘re-usable’ when it really isn’t.

Requirements of a ‘reporting environment’ tend to be fairly predictable: pagesize and linesize settings, file re-direction, production of alternate files types (.rtf, .pdf, etc.), handling of titles and footnotes, ‘page x of y’ page numbering, etc. These peripheral aspects tend to show the greatest success in standardization, but in a CRO setting even the basics of a reporting environment have to be flexible to client requirements. As requirements become less predictable, and show greater variability across studies (as is the case with data derivation and summarization), the greater the investment that will be required in developing code that satisfies the various permutations, the greater the maintenance overhead will become, and the greater the risk of error.

‘STANDARD CODE’ TRADEOFFS

Decision-making is all about tradeoffs, and if we think about tradeoffs that come into play when deciding if or how to implement ‘standard code’, we’ll recognize that they don’t just apply to the mechanics of code development and maintenance, or the logistics of code use, but that they ultimately involve learning, attitudes and behavior, and proficiency in our programming work force. Remember that efficiency is an elusive quality, and gains in one area invariably produce losses in others; the trick is to achieve the right balance and compromise, and to mitigate the losses.

In our discussion of how or if CROs should implement standard code, it might be useful to catalog the applicable tradeoffs under some broad classifications:

- Resource Allocation
- Flexibility, Complexity, Accuracy, Transparency
- Innovation and Professional Development

It’s by appreciating the benefits and draw-backs within these categories that we begin to see the problem as more complicated than simply co-opting some programs that seem to meet some functional requirements and declaring them standards.

RESOURCE ALLOCATION

Complex systems are generally created through some form of ‘design-build’ process, consisting of ‘requirements analysis’ and scope assessment, design, implementation and user buy-in, on a scale that wouldn’t be seen in ‘normal’ analysis programming. As such, standard code doesn’t arise spontaneously out of normal project work, and project budgets don’t allow for code maintenance that goes beyond the scope of the given project. Instead, the development and maintenance of ‘standard macro libraries’ requires dedicated resources, something that is generally antithetical to the ‘trimmed down’ nature of CROs.

We could say then that there is a resource-allocation tradeoff, where non-billable time dedicated to the development of standard code would have to be weighed against the possible efficiency and accuracy gains. For code intended to have broad application, documentation (and training) must be included in development costs. A clear and unambiguous description, written in ‘natural language’, can actually be much more difficult to attain than a workable piece of code, and as a result documentation usually lags behind code development, and tends to be of inferior quality.

It’s also important to consider whether what we’re attempting to standardize actually represents a significant proportion of the programming budget or accounts for a significant source of error. For instance, should we attempt to develop and administer ‘standard’ code for certain summary tables if these tend to take minimal time anyway and aren’t a significant source of error? Is it actually upstream, in the development of analysis datasets, that the greatest time investment is made, and the greatest risks associated with misinterpretation lie?

FLEXIBILITY, COMPLEXITY, ACCURACY, TRANSPARENCY

In some cases, if we’re lucky, we can make code more ‘flexible’ (as in, able to handle a wide range of conditions and/or able to meet a wide range of requirements) without increasing complexity. We might, for instance, transpose data to an orientation that is more appropriate to the tasks at hand, change the order in which tasks are performed, or make better use of SAS functions. More often than not though, we achieve greater flexibility at the expense of greater complexity, which tends to make code less accessible (or less ‘transparent’) to other programmers.

Consider, for instance, this real-life scenario: programs were written for a suite of ‘standard’ analysis datasets, originally designed for a small number of protocols but subsequently expanded to include several new protocols. Variability among the new protocols was handled by means of extensive protocol-specific branching:

```
%if &protocol=aaa %then %do;
%end;
%else %if &protocol=bbb %then %do;
%end;
```

To someone unfamiliar with the code attempting to make revisions, the programs devolved into what is generally referred to as ‘spaghetti-ware’, and the task of collectively maintaining ‘standard programs’ for these multiple protocols clearly became greater than the sum of each protocol handled individually. The problem was that the decision had been made early on to handle multiple protocols within a single set of programs, meaning that the situation had to be redressed by means of ‘de-macrotization’. So when embarking on such a strategy, we need to consider the potential long-term costs and risks (an analogy can be made with environmental remediation, where the question should be posed for any new development: will the eventual cost of decommissioning and clean-up exceed the economic benefits of the development?).

Another example is provided by ‘standard’ analysis datasets that try to accommodate variability between projects by adding variables (derivations and flags). This means that for any given study, the ‘standard’ dataset may include extraneous variables that have no relevance or meaning in the specific context. At best these extraneous variables are distractions or ‘red herrings’ but at worst they get used accidentally, particularly where dataset specifications are vague, ambiguous or altogether lacking.

A dataset that includes extraneous variables increases the risk of error, since it opens up the possibility that the wrong variable will be selected for analysis. Blindly calling a ‘standard’ program, without first acquiring familiarity with assumptions behind the program, can have equally disastrous results. For instance, consider this figure:

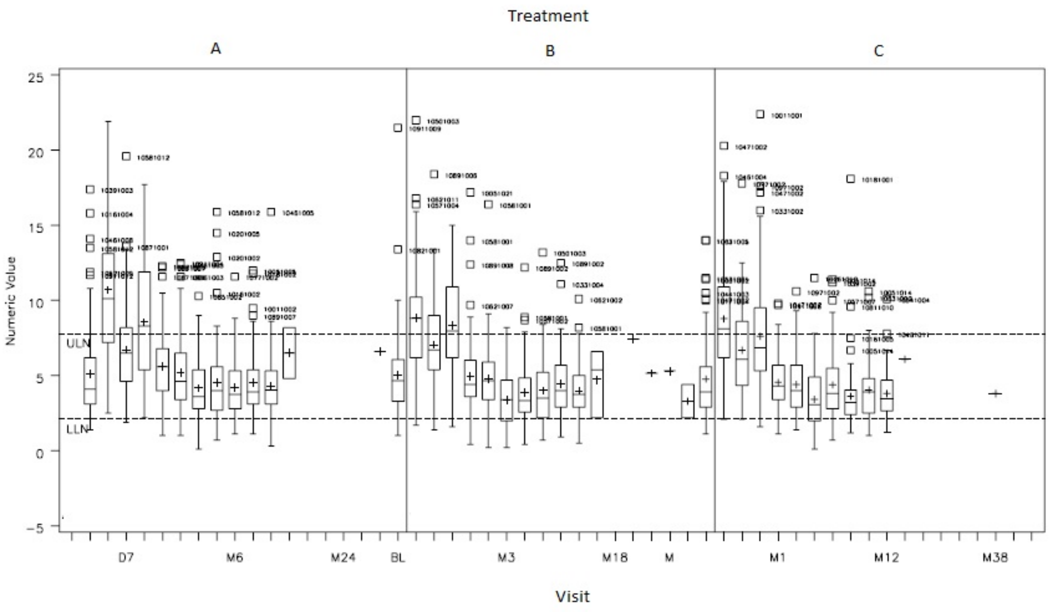

Figure 1. Unintended consequences of blind code re-use.

It’s not entirely clear what went wrong here, but this figure was originally intended to show visits baseline (BL) to month 38 (M38) for each treatment group. Instead, note how the x-axis labels are scrambled; it’s possible that the dataset passed to the figure-generating program didn’t anticipate the expectations of the program, or that the program was never intended to handle multiple treatment groups in the first place, but in any case, blindly calling the figure-generating program clearly didn’t produce the desired result. Further, as the output was generated by an ostensibly ‘validated/standard’ program, careful review of output was apparently circumvented.

The preceding examples all come from standard macros deployed within pharmaceutical companies, and all illustrate a key point: while these programs may have functioned per expectation when used by programmers familiar with their correct usage, exposing them to a wider user group, including contractors and other service providers, can cause problems, particularly when documentation is inadequate. Becoming invested in standard code may produce internal gains in efficiency and accuracy, but there may be negative consequences when the manner in which output is produced changes, such as when the user group broadens to include external service providers and contractors.

Macro as Meta-Language

One might argue that the kind of ‘information hiding’ that standard code seeks to achieve is already provided, and at an appropriate level, by SAS functions and procedures themselves, and that attempting to go beyond that level of black-box encapsulation is unnecessary and even counter-productive. Imagine a PROC REPORT block where every element has to be passed in via macro parameters, and the user has to know what each macro parameter means (as opposed to simply writing a PROC REPORT step from scratch).

Another way of looking at it is to consider the care with which the SAS language has been constructed, where (for instance) keywords and other aspects of syntax are chosen to promote semantic consistency, and ask if the structure and conventions employed in a standard macro exercise the same rigor. Learning to use a pre-written macro is essentially an exercise in learning a ‘meta-language’, or a language built on a language, and this exercise is laborious if the meta-language is not rigorously structured. Think of an extreme example, where parameters in a macro are simply named ‘var1’ and ‘var2’, or where the macro itself is simply named ‘macro1’. In other words, ‘syntax’ and ‘rules’ of the macro-as-meta-language may be ambiguous or inconsistent, especially when implemented by multiple programmers and/or where programmers don’t take the time to document, and to enforce consistency.

Further examples range from taking something as simple as packaging ‘call symput’ into a standard macro intended as a ‘tool’ or ‘utility’, to standard reporting macros that are so complicated, require so many calls and take so many parameters that the coding requirements easily outstrip the Base SAS code that would otherwise be employed. In both cases the ‘meta-language’ is not nearly as intuitive as the Base SAS language on which it is constructed, and as a result the learning curve for new programmers attempting to get up to speed in producing output can be greatly and unnecessarily increased. So as a rule of thumb we might say that simple tasks, like writing a value to a macro variable, should always be accomplished directly via constructs provided by Base SAS, and that excessive numbers of parameters probably signal that a macro is not workable as a standard, as do macro calls that require more lines of code that would be used to achieve the same result with Base SAS.

INNOVATION AND PROFESSIONAL DEVELOPMENT

Of all the attributes of a statistical programmer, probably the most prized is what we would refer to as ‘creative problem-solving ability’: the ability to respond to the unexpected, fill in gaps in incomplete information, and find solutions where none were immediately obvious. These are the traits we should be considering when we talk about ‘growing’ project teams and ‘building capacity’, since these are what enable us to deliver projects on time while using ‘lean’ project teams. Creative problem-solving ability is not something that is innate to a lucky few, but is learned, just like the rudiments of programming, through coming to grips with progressively more complex problems.

If we take rudimentary tasks and package them into standard macros, it might seem like a good way of keeping programmers from having to waste time on trivialities, but at the same time it can hinder the development of knowledge and skills among less experienced programmers, which then makes it more difficult to make the jump to more complex problems. Much of the efficiency that defines CROs actually stems from the kind of flexibility and adaptability that is demanded of employees, which presents a key trade-off: if we establish standard code, we may inhibit the development of this adaptability and problem-solving ability.

If we want to build proficiency in our programming work-force, perhaps we should start by taking into consideration how people learn, and ask if our conception of standard code libraries contributes to or hinders the learning process. For instance, Driscoll (2002) proposes the following principles for how people learn:

> *Learning occurs in context: Learning must happen within certain context. Without an appropriate setting, learning is unlikely to succeed.*
>
> *Learning is active: "Tell me, I forget. Show me, I remember. Involve me, I understand." This Chinese proverb suggests that learners have to be mentally active during learning activities, make connections between the new knowledge and existing knowledge, and construct meaning from their own experiences.*
>
> *Learning is social. Learners benefit from working collaboratively in groups so that they can hear different perspectives and accomplish the learning tasks with the help of their peers and experts.*
>
> *Learning is reflective. Learning is facilitated when learners are given chances to express and evaluate their own thinking.*

Gains in proficiency require that these tenets of learning be exercised, especially those of ‘active’ and ‘reflective’ learning, and these necessarily involve confronting the very same basic tasks that we may be intent on hiding inside standard code. There is a tradeoff then, indeed a paradox, in that if one of the intents behind standard code libraries is that they enable less experienced programmers to produce results, and rapidly perform more basic tasks, that this may actually stifle key aspects of the learning process. Assumptions about efficiency gains may be short-sighted, if the flip side is that we’re failing to cultivate the kind of creative problem-solving ability that can provide far greater productivity down the road.

Analysis programming is really the collision point between raw clinical data and desired inferences, and is where we discover if everything upstream (CRF design, data monitoring, data cleaning, etc.) is really sufficient to address the research questions being asked; it’s where we discover if we need special data-handling rules to overcome unexpected conditions in the raw data. This means that analysis programming necessarily involves a critical evaluation of source data, and a consideration of any implicit assumptions behind a given data summary in the context of specific raw data being applied. For example, a count of patients still on-study often assumes that all patients who are off-study provide either a completion or a reason-for-discontinuation record. At interim points during trial conduct, there may be internal inconsistencies due to incomplete data; for instance, a patient having no discontinuation record may nonetheless have an Adverse Event record showing action taken as study discontinuation. Identification and handling of these sorts of issues, and other forms of error-trapping and defensive coding, are usually done most effectively as part and parcel of code development, and often need to be tailored to the structure of individual studies. Writing code from scratch, as opposed to running pre-written code ‘on faith’, is often vital to these kinds of insights, meaning that what may appear superficially to be inefficient may actually ultimately produce better quality.

‘Pseudo-proprietary’ code (i.e. anything that creates the impression of having an ‘owner’, such as tightly controlled ‘black box’ code or even relatively complex template code) can have detrimental effects when it comes to professional development. First and most obviously, knowledge of both requirements and implementation can end up concentrated in a small number of programmers, which can lead to maintenance bottlenecks and documentation/training issues. More importantly though, it can create a sense of distance/detachment, a lack of engagement with the minute details of a problem, and a sense that responsibility for accuracy of output lies with the ‘owner’ of the code. We can sometimes see people taking the attitude that ‘I’m not sure if this is applicable to my project, but it’s not really my program and I hesitate to make changes…’. Again, writing code from scratch may seem superficially inefficient, but may also be vital to maintaining a sense of interest and engagement, which are in turn essential pre-requisites to high quality work.

ACHIEVING THE MIDDLE GROUND: CODE DESIGNED FOR ADAPTION & RE-USE

Taking all these trade-offs into consideration, and at the same time recognizing that some manner of code re-use does without question provide advantages, we have to ask what form this re-use might take that would provide benefits while minimizing risks and pitfalls.

Perhaps the best model of unconstrained code re-use and adaptation is found in the Open Source community. Source files will, for instance, usually include a comment header containing this statement:

```
// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
```

The implication is that when you include these source files, either verbatim or in some modified form, you ‘take ownership’, and accept responsibility for their performance in your own application. We could adopt the same thinking in SAS code libraries, where code is written to be helpful, but where anyone importing code into their own projects must treat the code as though they had themselves written it from scratch (understanding every facet in the context of the given project, and at very least corroborating the output through independent validation exactly as if the code had been written from scratch).

Rather than building code libraries that other programmers are *compelled* to use, we could take the attitude that our contributions to such code libraries would be driven by the desire to be helpful; that we write and submit code that we could imagine other people *wanting* to use. The question then becomes: what attributes would make code amenable to re-use, and would make us want to engage in code re-use? It’s really a question of how easy or difficult it is to ascertain:

- Does it do what I need?
- Can I modify it to my own purposes (more easily that simply writing a program from scratch)?
- Am I adequately warned of potential problems/pitfalls?

The answers to these questions are partly a matter of documentation, but also largely dependent on code organization/architecture. A description of input/output placed in a program header will not tell you anything about problems you may eventually face in adapting a program to project-specific requirements, nor does it in of itself inspire faith in the ‘robustness’ of a program; rather, the program code itself supports these judgments, and will give you a sense of whether you’re better off programming from scratch (and remember, in a CRO setting in particular these judgments need to be made very rapidly).

GUIDELINES FOR DEVELOPING ADAPTABLE CODE

While ‘good programming practice’ may be expected as a matter of course for all production code, if we’re writing programs to include in a code library, we probably need to work to a higher standard: we need to anticipate the critical evaluation a potential user will be making when deciding whether to adapt or write from scratch. We need to make programs instructive, so that a potential user will be able to see that the program does indeed do what the description in the header says it does.

Recall that ‘learning’ is said to involve making connections between new knowledge and existing knowledge: you might say that as we read through a program, the preceding steps have become our ‘existing knowledge’ and the current steps are our ‘new knowledge’, and that if we’re plagued with extraneous information (i.e. a whole series of WORK datasets named ‘data1’, ‘data2’, etc.) it impedes our ability to make these connections. Do you ever find yourself thinking “the step I’m currently looking at does something to dataset ‘final_1b’, but this dataset was created 200 lines earlier, has a fairly meaningless name, and by now I’ve completely lost track of the operations that created it in the first place”?

In music the term ‘augmentation’ applies where an interval (the distance between two notes) is widened by a semi-tone, creating a modified harmony but one where the original characteristics of the chord are still identifiable. We might apply the same principle to datasets, where we start with a well-defined dataset (say ADSL), and simply ‘augment’ it through various steps, but without re-naming it or removing any of the original ADSL content. It is then much easier to attach our understanding of the additions being made to our pre-existing knowledge of ADSL, since the ‘core’ of the dataset persists (meaning we can continue to take advantage of the knowledge we already have of the dataset), and derivations are added in manageable increments.

Based on these considerations, we might develop a general set of guidelines to apply when writing code specifically intended for re-use and adaptation, loosely categorized here under ‘basic aesthetics’, ‘architecture’ and ‘documentation’.

Basic aesthetics

Code should be structured to promote human-readability; as with most things, how much faith a potential user places in what you’re offering is largely dictated by initial appearances. In spoken and written language we follow grammatical rules so that we can be understood, and similar principles should be applied when writing code intended to be interpretable by other programmers.

- Consistent and informative use of indents (indents should designed to make the relationship between individual code statements more obvious)
- Limit the length of individual code statements, and avoid placing multiple SAS statements on the same line (similar to avoiding run-on sentences)
- Limit code block size: keep individual data steps small enough that they can be read at a glance (similar to using paragraphs to delineate specific ideas).
- ‘Proc’ steps should explicitly reference datasets (similar to how a ‘subject’ should be identifiable in a spoken sentence)
- Clean up and remove irrelevant code and comments

Architecture

This lends an additional degree of organization and coherence, just as chapters do in the organization of a book, and when modifications are required helps guide the search for specific functionality.

- Group related operations, and in particular clearly separate data retrieval, data summarization, and reporting-preparation tasks (interspersing these tasks can make for a maintenance nightmare, and will immediately signal to a potential user that adapting the program is a risky proposition)
- Separate data-driven or project-specific rules from generic calculations (other programmers will want to be able to tell what is ‘boilerplate’ (i.e. a step that retrieves standard descriptive stats from PROC UNIVARIATE) and what will likely need to be modified (i.e. things like project-specific date imputations))
- Maintain a clear connection between source and summary data (minimize ‘set-up’ and pre-processing). Summary tables are not necessarily designed to fall conveniently from source data, and the process of getting from source data to ‘raw’ summary results should not be cluttered with pre-processing intended to meet organizational requirements of the final output – this should always come as a post-processing step after the tasks of data retrieval and data summarization have been accomplished. (Keep in mind the concept of ‘one proc away’ - the further away you get from this, the less adaptable your code will be).
- Try to avoid ‘implicit assumptions’. For instance, sets of conditional statements should usually use ‘else if’ and be terminated by defensive code that traps any instances not handled by the preceding conditions (rather than simply assuming that no such instances will arise). More subtly, code that makes use of ‘if/else if’ constructs where the ‘else’ clauses refer to different variables from those referenced by the ‘if’ clause may also present unanticipated implicit assumptions (while these may make perfect sense to the original author, they can be terribly confusing to anyone else trying to make use of the code).

- Try to limit ‘artifacts’ such as temporary WORK datasets. For example, sub-setting and re-merging datasets in order to achieve cross-record comparisons if the same result can be achieved by augmenting a single dataset using ‘lag’ or ‘retain’.

Documentation:

Program documentation is notorious for stating the self-evident while deemphasizing guiding principles, purpose, or overall design. (How often have you seen comments that state ‘merge datasets’ rather than saying ‘retrieve last dose date from EX, and retrieve end of study date from DS, and censor at the later of these two dates’?) Comments should adequately narrate the main tasks carried out by a program, but should also alert a potential user of pitfalls, assumptions, and areas possibly requiring modification.

- Should tell the user what is being done (where this is not inherently obvious), but should also say why (for instance, if the idea of a ‘best response’ seems to be contradicted by a ‘conservative principle’).
- Should state assumptions (again, where these are not inherently obvious – what we want is to specifically bring attention to things that we think a potential user might not have thought of, but which the author did while in the agonized process of writing the program in the first place).
- Should state potential pitfalls
- Should advise a potential user of where and in what way project-specific requirements might be met

RE-USE AND ADAPTATION OF CODE

It’s probably fair to say that no code should ever be used ‘on faith’, and that this even applies to supposedly validated ‘black box’ macros: output should always be subjected to critical evaluation, regardless of the manner in which it was produced. Where ‘template’ code is adopted and modified, this is even more true; creating a version or copy of a program, even where no modification takes place, entails ‘taking ownership’ of the code in its new setting. This means that it’s the responsibility of the user to study and understand code before implementing it, and this should play a central role in the decision of whether to adopt code in the first place, or write it from scratch. Remember that your confidence in being able to maintain and modify code provides your safety net against late-breaking and project-specific requests for revisions, and that efficiency gains from code re-use have to be weighed against the risks associated with using code without a clear understanding of its functionality.

ILLUSTRATION: AN EXAMPLE PROGRAM DESIGNED TO PROMOTE RE-USE AND ADAPTATION

The following excerpt is from a program that produces an Adverse Event frequency count by highest reported severity (toxicity grade), and attempts to illustrate some of the ingredients proposed for programs intended for re-use and adaptation. The output table takes the following form:

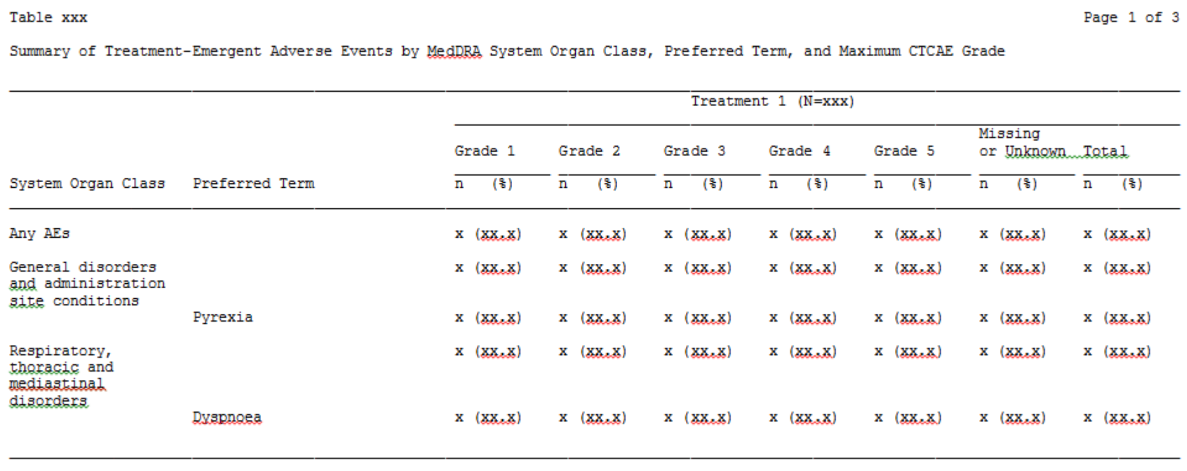

Table xxx Page 1 of 3

Summary of Treatment-Emergent Adverse Events by MedDRA System Organ Class, Preferred Term, and Maximum CTCAE Grade

		Treatment 1 (N=xxx)						
		Grade 1	Grade 2	Grade 3	Grade 4	Grade 5	Missing or Unknown	Total
System Organ Class	Preferred Term	n (%)	n (%)	n (%)	n (%)	n (%)	n (%)	n (%)
Any AEs		x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)
General disorders and administration site conditions		x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)
	Pyrexia	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)
Respiratory, thoracic and mediastinal disorders		x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)
	Dyspnoea	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)	x (xx.x)

Figure 2. A typical Adverse Event table stratified by treatment group and maximum severity.

To aid the potential user in making that ‘adapt versus write from scratch’ decision, the program header contains the following notes:

```
    NOTES:
    Produces tables by body system/preferred term, or by preferred term only, and
    assumes that body system is provided by the variable AEBODSYS and preferred
    term by the variable AEDECOD, and toxicity by the variable AETOXGRN. Removing
    the toxicity-grade columns for each treatment group and reporting only the
    'total' columns will reduce the table to a 'typical' AE table (frequency counts
    by body system/preferred term or preferred term only, by treatment group and
    irrespective of maximum severity).

    Frequency counts at the level of preferred term are the same regardless of
    whether or not body system is displayed.

    Note that overall counts within each toxicity grade within body system may not
    equal the sum of counts within each preferred term, since the highest toxicity
    a patient shows within a preferred term may be lower than that shown by the
    same patient within the corresponding body system (watch out for this as an
    inevitable review comment).

    WARNING: This program does not impute missing toxicity grade, but is simply
    designed to report these in a separate column. (Since 'missing' is reported as
    column 6, a dummy toxicity variable is required when selecting maximum
    toxicity, to prevent missing toxicity from being selected as highest).

    WARNING: When producing a table that only reports grades 3/4/5, the filter
    needs to be applied as a 'table-specific filter' (i.e. don't be fooled into
    thinking you can simply remove the columns for grades 1/2), and in addition,
    you need to check whether or not your study requires that missing toxicity be
    imputed (i.e. as maximum toxicity) or can simply be represented in a separate
    'missing' column.
```

These notes are intended to help answer the questions “does it do what I want” and “am I adequately warned of potential pit-falls”. The question “can I modify it to my own purposes” is best answered by studying the architecture and organization of the program itself, in particular how well the tasks of data retrieval and set-up, summarization, and reporting are separated and organized, and by how well commented the program is (especially in highlighting steps that may require project-specific modifications). What follows are two excerpts from the program, the first providing data retrieval and set-up, and the second carrying out the core data summarization (frequency count and percent calculation) tasks.

```
%macro rpeat(series=,…);

  %*** -----START DATA RETRIEVAL AND SET-UP-----;

  proc sort data=sasdata.adsl out=adsl;
    by usubjid;
    where saffl="Y"; %*** (SAFETY POPULATION);
  run;
  proc sort data=sasdata.adae out=adae;
    by usubjid;
    where trtemfl="Y" & compress(aeterm)^=""; %*** (TREATMENT-EMERGENT, NON-MISSING
                                                     VERBATIM TERM);
  run;
  data adae;
    set adae;
```

```
      &filter; %*** TABLE-SPECIFIC FILTER (FOR EXAMPLE, AESER="Y" FOR SAEs);
    run;

    data adae;
      merge adae(in=_1) adsl(in=_2);
      by usubjid;
      if _1 & _2; %*** (SUBSET BY SAFETY POPULATION AND SECONDARY FILTERS);
    run;
    %*** WARNING: IN THE FOLLOWING STEP, HANDLING OF NON-CODED AEs MAY BE PROJECT-
         SPECIFIC. IN THIS STUDY, FOR TABLES SUMMARIZING BY BOTH BODY SYSTEM AND
         PREFERRED TERM, VERBATIM TERM IS SUBSTITUTED FOR MISSING PREFERRED TERM;
    %*** WARNING: IN THE FOLLOWING STEP, TREATMENT OF MISSING TOXICITY GRADE MAY BE
         PROJECT-SPECIFIC. IN THIS STUDY, MISSING TOXICITY IS >NOT< IMPUTED TO MAXIMUM
         TOXICITY.;
    data adae;
      set adae;
        --ETC.—
    run;

    %*** -----END DATA RETRIEVAL AND SET-UP-----;
```

Note here that set-up code is clearly identified and is fairly minimal, and that the result is simply an ‘augmented’ form of ADAE, making it fairly easy to visualize the data being referenced in the data summarization section (filtering has been applied, along with some limited coding to handle missing toxicity grades and preferred terms, but the resultant dataset is still named ‘ADAE’, and contains no ‘dummy’ records (i.e. to pool treatment groups), nor are there multiple subsets generated to capture maximum toxicity at each MedDRA coding level, since this is handled in the data-summarization section). Note also the comments prefaced with the word ‘WARNING’ (such that these will show up in a log summary) that highlight project-specific assumptions and that alert the user to potential project-specific revisions that may be required.

Following the ‘set-up’ section is a clearly delineated ‘data summarization’ section. While this section is fairly dense, the effort has been made to reference only those (work) datasets created in the set-up section (so that there is no distraction from additional data retrieval or filtering), and to leave any ‘reporting-preparation’ tasks to a later section. SQL sub-queries are employed so that maximum toxicity can be selected ‘on the fly’ rather than requiring record sub-setting as an earlier step, and macro loops are designed so that the last iteration pools all records (for instance, the last iteration in the treatment loop pools all patients regardless of treatment group, which is why dummy-record generation was not required as a set-up step).

```
  %*** -----START FREQUENCY CALCULATION-----;

  %macro freq_;

    %do trt=1 %to &ntrt+1; %***TREATMENTS: NTRT GIVES THE NUMBER OF TREATMENT GROUPS,
                               NTRT+1 COMBINES ALL TREATMENT GROUPS;

      /*** DENOMINATOR: TOTAL COUNT WITHIN TREATMENT GROUP (SAFETY POPULATION) ***/
      %global n&trt;
      proc sql noprint;
        select count(distinct usubjid) into :n&trt from adsl
          where &trtvr=&trt or &trt=(&ntrt+1)
        ;
      quit;

      %do tox=1 %to 7; %*** (5 TOXICITY GRADES, PLUS MISSING (TOX=6),
                           PLUS OVERALL (TOX=7));
        proc sql noprint;

        create table _&trt&tox as select * from (

         /*** ANY AE (AT THE HIGHEST TOXICITY REPRESENTED) ***/
         (select distinct 0 as section, 0 as ord, "Any AEs" as aebodsys,
                                                 "Any AEs" as aedecod,
           put(count(distinct usubjid),3.0)||" ("||
           trim(left(put(count(distinct usubjid)/&&n&trt*100,8.1)))||")" as _&trt&tox

            from
              /*** WITHIN PATIENT, TAKE THE HIGHEST TOXICITY REPRESENTED ***/
              (select * from adae group by usubjid having aetoxgrn_=max(aetoxgrn_))
               where (aetoxgrn=&tox OR &TOX=7) & (&trtvr=&trt or &trt=(&ntrt+1))
          )

        union all

         /*** ANY AE WITHIN BODY SYSTEM (AT THE HIGHEST TOXICITY REPRESENTED) ***/
         (select distinct 1 as section, 0 as ord, aebodsys, "Any AEs" as aedecod,
           put(count(distinct usubjid),3.0)||" ("||
           trim(left(put(count(distinct usubjid)/&&n&trt*100,8.1)))||")" as _&trt&tox

            from
              /*** WITHIN PATIENT|BODSYS, TAKE THE HIGHEST TOXICITY REPRESENTED ***/
              (select * from adae group by usubjid, aebodsys
                having aetoxgrn_=max(aetoxgrn_))
            where (aetoxgrn=&tox OR &TOX=7) & (&trtvr=&trt or &trt=(&ntrt+1))
          group by aebodsys)

        union all

          /*** ANY AE WITHIN BODY SYSTEM/PREFERRED TERM (AT THE
               HIGHEST TOXICITY REPRESENTED) ***/
                       --etc.--
              /*** WITHIN PATIENT|BODSYS|PREF TERM, TAKE THE HIGHEST
                   TOXICITY REPRESENTED ***/
                       --etc.--
         )
         where section>.
         order by section, ord, aebodsys, aedecod
         ;

        quit;

      %end;

    %end;

    %*** FINAL DATASET CONTAINS 1 COLUMN PER TREATMENT GROUP PER TOXICITY GRADE
        (INCLUDING OVERALL TOXICITY, AND COMBINED TREATMENTS);
    data final;
      merge %do trt=1 %to &ntrt+1;
              %do tox=1 %to 7;
                _&trt&tox
              %end;
            %end;;
      by section ord aebodsys aedecod;
    run;
  %mend freq_;
  %freq_;

  %*** -----END FREQUENCY CALCULATION-----;
```

The nested ‘treatment’ and ‘toxicity’ loops fairly intuitively match the arrangement of columns in the table, and in fact the work dataset ‘FINAL’ is nearly report-ready. A final section (omitted here for brevity) performs any required formatting tasks: filling in zero-counts, combining body system and preferred term into a single column, if required, and establishing sort order if it is something other than alphabetical by body system and preferred term (for instance, in some cases output needs to be ordered in descending frequency), and ends with the PROC REPORT statements to produce the summary table.

CONCLUSION

Hopefully this paper will have touched on some of the practical considerations involved in deploying standard code in a CRO setting, be that anything from ‘black box’ macros to ‘template’ programs, and will promote some debate on the subject.

If we are to enjoy efficiency gains from code re-use, but without suffering maintenance bottlenecks and quality implications from potential misuse, and while respecting the objectives of growing project teams and building proficiency among less experienced programmers, we might do well to look to the ‘open source’ initiative, where we can find projects of great sophistication, but which promote a sense of ‘ownership’ among code adopters while supporting extension and adaptation. When developing code libraries, we should consider what code attributes will be the most amenable to re-use and adaptation; we should consider ways of drawing contributions from the entire ‘programming community’ and we should avoid having code development become 1-directional, in a way that fosters a sense of detachment, but should rather use it as a means of furthering professional development.

ACKNOWLEDGMENTS

I would like to thank ICON Clinical Research for consistently encouraging and supporting conference participation, and all my friends at ICON for their great ideas, enthusiasm and support, in particular Syamala Schoemperlen, Stephen Hunt and Jasmin Fredette.

REFERENCES

Driscoll, M. P. (2002). *How people learn (and what technology might have to do with It)*.

CONTACT INFORMATION

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

Name:	Brian Fairfield-Carter
Enterprise:	ICON Clinical Research, Inc.
E-mail:	fairfieldcarterbrian@gmail.com
Web:	http://www.iconplc.com/