PharmaSUG 2014 - Paper TT13

Internal Consistency and the Repeat-TFL Paradigm: When, Why and How to Generate Repeat Tables/Figures/Listings from Single Programs

Tracy Sherman, InVentiv Health Clinical, Cary, NC

Brian Fairfield-Carter, InVentiv Health Clinical, Cary, NC

ABSTRACT

The concept of 'repeat' tables/figures/listings (TFLs), those being groups of 'similar' output that are understood to require less time and effort to produce than 'unique' TFLs, is probably familiar to most programmers. But even with the interest programmers have in improving efficiency and reducing maintenance overhead, it's surprising how often the relationship between repeat TFLs isn't exploited in code-writing. Perhaps this stems from inadequate planning (i.e. allocating work to a rapidly-assembled programming team without first assessing relatedness among TFLs), possibly combined with a lack of understanding of either the importance of grouping related output, or of the programming techniques that can be applied in producing repeat TFLs, but in any event we often see projects that follow a '1 table/1 program' (1:1) paradigm, and which treat each TFL in isolation.

This paper illustrates the often-overlooked dangers inherent to the '1:1' paradigm, and proposes as an alternative that multiple repeat TFLs should be generated by single programs, not only as a means of improving efficiency, but also to safeguard quality (particularly with regard to consistency in style and in computational methods). Simple and practical tips for deciding when groups of TFLs should be treated as repeats are offered, along with an illustration of how to set up a single program to generate multiple repeats without adding significantly to program complexity.

INTRODUCTION

Any programmer who has been involved in discussions of project scope and project costing will undoubtedly have been asked to provide counts of 'unique' and 'repeat' TFLs. 'Unique' TFLs are understood to be those that do not share common attributes, and as such require more time and effort to produce than do 'repeat' TFLs.

Repeat TFLs instinctively fit with the iterative processes we usually think about when writing code, so it's surprising when we run across projects that display this manner of organization:

Table #	Title	Program	Programmer
1.1	Reasons for Discontinuation (All patients)	t_disc_all.sas	Moe
1.2	Reasons for Discontinuation (Safety population)	t_disc_saf.sas	Curly
1.3	Reasons for Discontinuation (ITT population)	t_disc_itt.sas	Biff
2.1	Adverse Events (Safety population)	t_ae.sas	Jimbo
2.2	Adverse Events leading to Study Discontinuation (Safety population)	t_ae_disc.sas	Mabs

In other words, related and similarly-structured tables, where in all likelihood the only difference is in the primary record-selection criteria, are each created by separate programs (and possibly by different programmers). This is what we might refer to as a '1 table/1 program' (1:1) paradigm.

Though we automatically see the inefficiency in this arrangement, we might ask whether it really matters; the loss of efficiency might seem trivial, but are there other, more significant dangers to this paradigm? Well, consider how 'senior review' of statistical output tends to play out: table production and QC might proceed at a healthy pace, but then in the final days before output is due, statistical review produces comments like these:

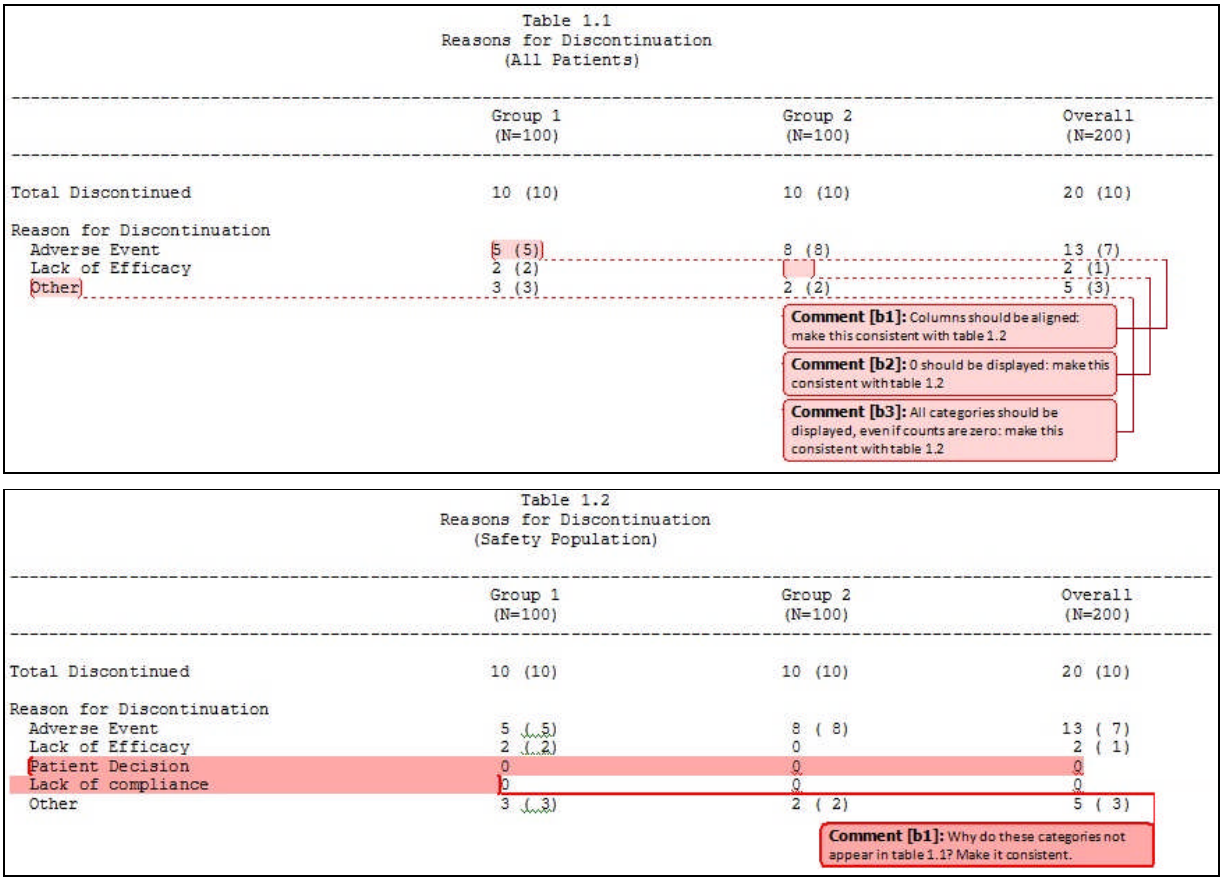

Table 1.1
Reasons for Discontinuation
(All Patients)

	Group 1 (N=100)	Group 2 (N=100)	Overall (N=200)
Total Discontinued	10 (10)	10 (10)	20 (10)
Reason for Discontinuation			
Adverse Event	5 (5)	8 (8)	13 (7)
Lack of Efficacy	2 (2)		2 (1)
Other	3 (3)	2 (2)	5 (3)

Comment [b1]: Columns should be aligned: make this consistent with table 1.2

Comment [b2]: 0 should be displayed: make this consistent with table 1.2

Comment [b3]: All categories should be displayed, even if counts are zero: make this consistent with table 1.2

Table 1.2
Reasons for Discontinuation
(Safety Population)

	Group 1 (N=100)	Group 2 (N=100)	Overall (N=200)
Total Discontinued	10 (10)	10 (10)	20 (10)
Reason for Discontinuation			
Adverse Event	5 (5)	8 (8)	13 (7)
Lack of Efficacy	2 (2)	0	2 (1)
Patient Decision	0	0	0
Lack of compliance	0	0	0
Other	3 (3)	2 (2)	5 (3)

Comment [b1]: Why do these categories not appear in table 1.1? Make it consistent.

Figure 1. Some common statistical review comments pointing to stylistic inconsistency.

In other words, there's nothing inherently 'wrong' with the tables, and no fault has been found with computed values, but there's a lack of 'internal consistency' in style of presentation ('internal' in the sense of being within the confines of a specific project). Statistical reviewers will often preface these kinds of review comments with a general observation that "things need to be consistent: the specifics of the convention don't matter, but you need to have a convention".

Further, with this 1:1 paradigm for table production we often see review comments like these:

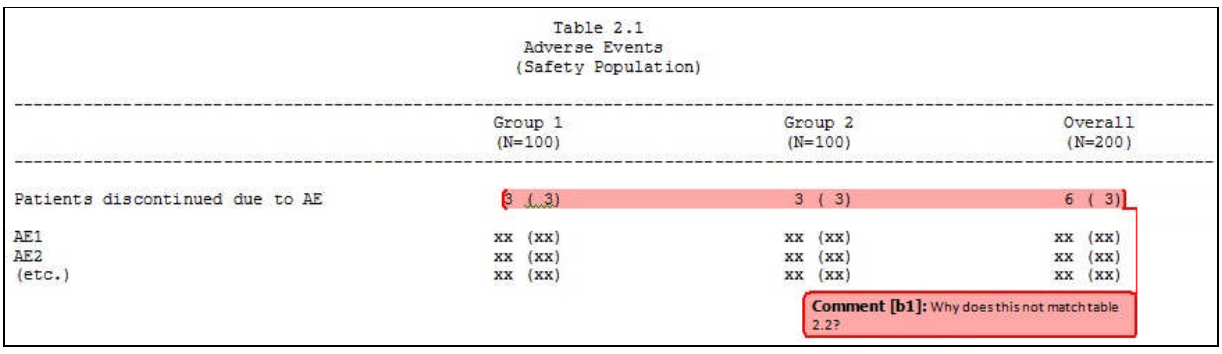

Table 2.1
Adverse Events
(Safety Population)

	Group 1 (N=100)	Group 2 (N=100)	Overall (N=200)
Patients discontinued due to AE	3 (3)	3 (3)	6 (3)
AE1	xx (xx)	xx (xx)	xx (xx)
AE2	xx (xx)	xx (xx)	xx (xx)
(etc.)	xx (xx)	xx (xx)	xx (xx)

Comment [b1]: Why does this not match table 2.2?

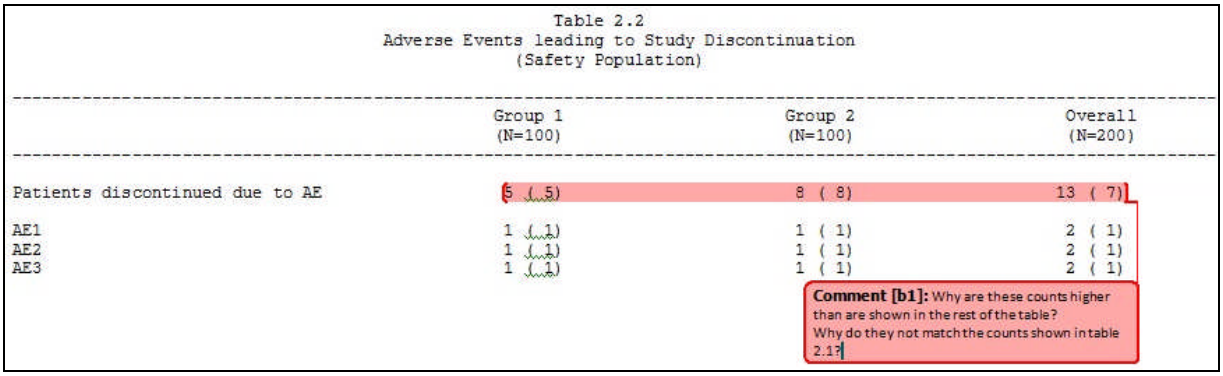

Table 2.2
Adverse Events leading to Study Discontinuation
(Safety Population)

	Group 1 (N=100)	Group 2 (N=100)	Overall (N=200)
Patients discontinued due to AE	5 (5)	8 (8)	13 (7)
AE1	1 (1)	1 (1)	2 (1)
AE2	1 (1)	1 (1)	2 (1)
AE3	1 (1)	1 (1)	2 (1)

Comment [b1]: Why are these counts higher than are shown in the rest of the table? Why do they not match the counts shown in table 2.1?

Figure 2. Common statistical review comments pointing to an absence of expected parity.

Again, there's nothing necessarily 'wrong' with the calculated values, or at least they can in each case be logically 'justified' as alternate interpretations of (arguably) ambiguous requirements: counts of patients discontinued due to AEs in table 2.1 derive from AE records where action taken in response to the AE is 'Study Discontinuation', while the same row in table 2.2 shows the count of patients indicating 'AE' as their reason for discontinuation (i.e. on a Study Discontinuation CRF page), as indicated by the fact that the values agree with what is shown in table 1.2 (the 'AE' row on the 'Reasons for Discontinuation' table). We might expect these two data sources to agree in a clean/locked database, but we might not be able to make such an assumption on interim & incomplete data. A reviewer should however be able to assume that *the same data source will be used when the same summary value appears in more than one place*.

These review comments might still seem trivial, and with obvious remedy, but consider if the traits they represent were multiplied across a couple hundred tables: the review comments would achieve epidemic proportion, and often at a stage in the project when there's little time available for updates (since this sort of 'senior statistical review', with the types of checks for internal consistency that may be overlooked in more compartmentalized QC/validation, often takes place very late in the project). Consider also that when there are extensive and systemic inconsistencies across a set of output, even when the reported values are themselves justifiable, it tends to damage the credibility of all of the summaries presented, even those that may in fact be perfectly correct.

Finally, what this type of project organization does is to demand maintenance of multiple copies of essentially the same code: at best the program to produce table 1.2 will be a modified copy of the program to produce table 1.1. But with different programmers assigned to tables belonging to the same series, the maintenance requirements are greatly amplified by differences in programming style; not only does this practice tend to create large numbers of the kinds of review comments illustrated above, but efforts to coordinate revisions are hindered by variability in individual programming style, and often take multiple iterations. Even with a fairly stable project team this is difficult to manage, but consider what happens when the initial bolus of work is performed by a project team cobbled together in a hurry, where programmers are then re-allocated to other projects and you're left with the task of wading through all these programs yourself, and enduring widely varying program style and structure while trying to impose consistency after the fact.

ALTERNATIVES TO THE 1:1 PARADIGM

So given the problems with internal consistency posed by this 1:1 paradigm, what are the alternatives? One obvious one is to use 'standard' and/or project-specific macros, for instance where the calculation of basic descriptive statistics and frequency counts is handled not by code contained in each table program, but by macros stored in a macro library. This might be taken further, such that project-specific macros would be developed to meet the requirements of each repeating table structure (i.e. frequency counts by treatment group and other user-specified 'by' groups, where the output is a 'report-ready' dataset to be passed to PROC REPORT).

There are a number of draw-backs to this approach though:

- External macros present a learning curve and an impediment to quickly-assembled project teams
- Macro-library architecture invariably becomes complicated: either the number of macros becomes excessive, in order to allow for variations within the set of TFLs, or individual macros become complicated and hard to maintain (they take large numbers of arguments, and show extensive table-specific branching).

- Division of responsibility can become clouded: individual TFL programmers may expect the macro developer to provide table-specific branching in order to accommodate variability among tables belonging to the same series, or alternately you may end up with multiple programmers trying to modify the same macro to specific needs.

THE 'MANY TABLES:1 PROGRAM' (MANY:1) PARADIGM

Unless project teams are stable, it is far preferable to make each TFL program as self-contained as possible, and limit reliance on external macros; this is far more likely to allow a programmer recently assigned to a project to be immediately productive, and will make it much easier to work with 'inherited' code when the project team changes. Repeat tables should be assigned to the same programmer, not only because that will avoid conflicting interpretations within a given series, but because it will make it possible for the same TFL program (via in-line macro code) to produce all tables within the series (which will in turn enforce stylistic consistency and, probably more importantly, allow revisions to be made in one place rather than in parallel across multiple programs).

Adopting this 'many tables/1 program' paradigm will involve compromise: on the one hand it won't necessarily address inconsistencies between tables which share some common attributes but which aren't really part of the same series (i.e. presentation of frequency counts in a Demography table might still differ from that in an AE table), but it will without question streamline revisions that might need to be made to address these inconsistencies. It will really provide your best defense against internal inconsistency, while minimizing overall complexity in your programming environment and allowing for changes in your project team.

DECIDING WHETHER OR NOT TABLES ARE ‘REPEATS’

Classifying tables as repeats is very much an art: there are no hard and fast rules, and much has to do with aesthetics, specifically how cluttered and inelegant you're willing to let you code become for the sake of maximizing output from a single program.

Ultimately though it boils down to categorizing TFLs based on 'structural' and 'thematic' characteristics:

Structural characteristics

- Number and arrangement of rows and columns
- Presence of frequency counts, percents, descriptive and/or inferential statistics

Thematic characteristics

- Data source (i.e. AE versus Lab versus Demographic data)
- Record-selection criteria (simple analysis-population filter, or more complex filters)
- Data-handling rules/methods of calculation (i.e. single record per patient, or single record per patient per event; denominator as the treatment group size, or as the number of patients providing a response)

Tables that are structurally identical but use different primary record-selection criteria and/or summarize different but parallel suites of parameters are obviously repeats (i.e. AE tables repeated by analysis population; lab tables where all that differs is the suite of lab parameters, such as hematology versus chemistry); less obvious are the cases where tables are 'nested' rather than being structurally identical (i.e. an AE table that gives frequencies by treatment group can be seen as being 'nested' within an AE table that additionally stratifies by maximum AE severity (this will be illustrated in greater detail later)).

When it comes to actually writing code, the degree of relatedness between tables roughly translates into the number of parameters, or points of variability, we have to build into a macro. Where tables are not structurally identical the decision of whether or not to treat them as repeats is a matter of weighing relative costs: is the additional complexity adequately offset by the increased ease of maintenance and revision? There is often no obvious answer, and it will depend on your ability to anticipate complexity, based on your knowledge of the source data, and your interpretation of the table shells.

ORCHESTRATING ‘REPEAT’ TABLES

Once we identify a series of repeat tables, how do we orchestrate generating this series from a single program? Basically, generating repeat tables is just a matter of separating the 'boiler-plate' which is common to all tables from those elements that vary between tables, and using macro substitution for the latter. For example, say we had the following series of (hypothetical) repeat tables:

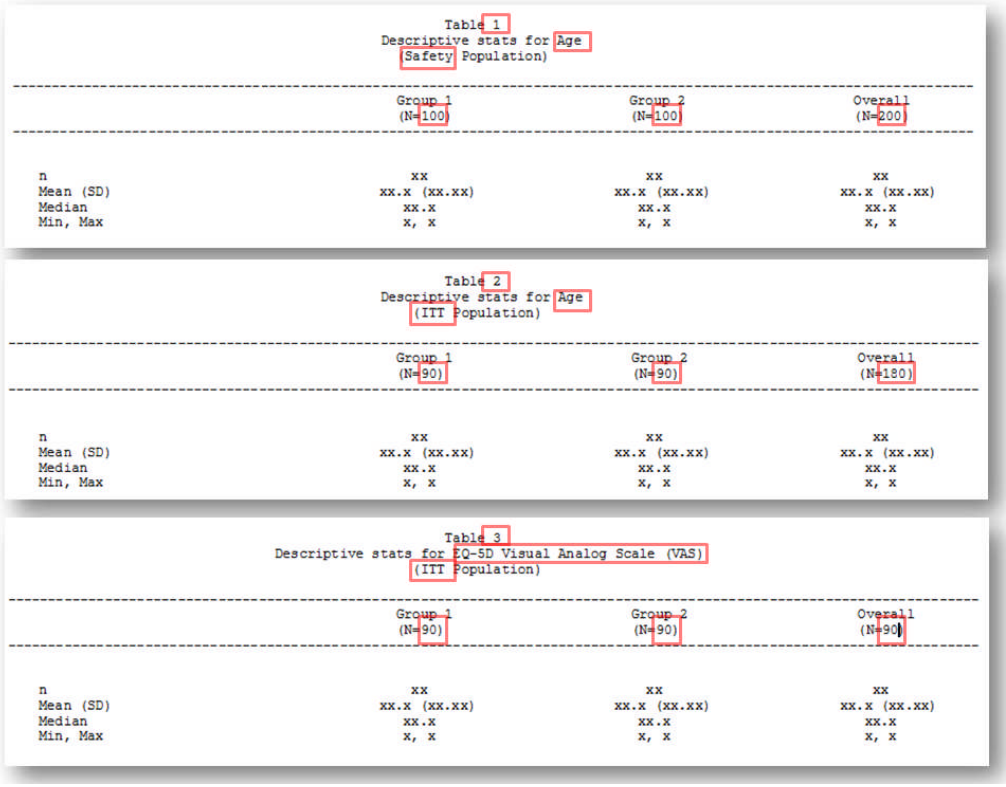

Table 1
Descriptive stats for Age
(Safety Population)

	Group 1 (N=100)	Group 2 (N=100)	Overall (N=200)
n	xx	xx	xx
Mean (SD)	xx.x (xx.xx)	xx.x (xx.xx)	xx.x (xx.xx)
Median	xx.x	xx.x	xx.x
Min, Max	x, x	x, x	x, x

Table 2
Descriptive stats for Age
(ITT Population)

	Group 1 (N=90)	Group 2 (N=90)	Overall (N=180)
n	xx	xx	xx
Mean (SD)	xx.x (xx.xx)	xx.x (xx.xx)	xx.x (xx.xx)
Median	xx.x	xx.x	xx.x
Min, Max	x, x	x, x	x, x

Table 3
Descriptive stats for EQ-5D Visual Analog Scale (VAS)
(ITT Population)

	Group 1 (N=90)	Group 2 (N=90)	Overall (N=90)
n	xx	xx	xx
Mean (SD)	xx.x (xx.xx)	xx.x (xx.xx)	xx.x (xx.xx)
Median	xx.x	xx.x	xx.x
Min, Max	x, x	x, x	x, x

Figure 3. A hypothetical series of repeat tables, with points of table-specific variation highlighted.

Note that these tables are structurally identical, but differ along thematic lines (tables 1 and 2 use different analysis populations, and table 3 uses a different data source and presents descriptive statistics for a different variable from that used in tables 1 and 2).

Under the '1:1' paradigm, table 1 would probably be programmed first, and the program would then be copied and modified in order to produce tables 2 and 3. In getting used to working with the 'many:1' paradigm, the simplest approach is to *start* by writing a program as though it was intended to generate just a single table:

```
*** DATA RETRIEVAL & FILTERING;
proc sort data=derived.adsl out=adsl;
  by usubjid;
  where saf="Y";
run;

*** GROUP COUNTS;
proc sql noprint;
  select count(*) into :n1 - :n2 from adsl group by trt;
quit;

*** SUMMARIZATION;
proc univariate data=adsl;
  var age;
  …
run;

*** REPORTING;
title1 "Table 1 Summary Stats for Age"
title2 "(Safety Population)";

ods rtf file="mytable.rtf";

  proc report data=final…;
```

```
    …
  run;

ods rtf close;
```

Once the program functions correctly in producing a single table, we'd review the points of variation between tables in the series and convert the program to an in-line macro, using parameters to handle the variations between tables (note that setting titles and footnotes from an external file (i.e. Excel data imported into SAS is a fairly common approach, or even just a simple text file) rather than by hard-coding them in the table program itself will greatly reduce clutter, but isn't strictly necessary):

```
%macro rpeat(series=,dsn=,pop=,var=);

  *** DATA RETRIEVAL & FILTERING;
  proc sort data=derived.&dsn out=derived;
    by usubjid;
    where &pop="Y";
  run;

  *** GROUP COUNTS;
  proc sql noprint;
    select count(*) into :n1 - :n2 from adsl group by trt;
  quit;

  *** SUMMARIZATION;
  proc univariate data=derived;
    var &var;
    …
  run;

  *** REPORTING;
  %mtitle(id=myprogram&series); *** (SET TITLES AND FOOTNOTES
                                     (BASED ON PROGRAM NAME AND SERIES));
  ods rtf file="mytable&series..rtf";

    proc report data=final…;
      …
    run;

  ods rtf close;

%mend rpeat;

%rpeat(series=1,dsn=adsl,pop=saf,var=age); *** (STATS ON AGE, SAFETY POP);
%rpeat(series=2,dsn=adsl,pop=itt,var=age); *** (STATS ON AGE, ITT POP);
%rpeat(series=3,dsn=adqol,pop=itt,var=VAS);*** (STATS ON VAS, ITT POP);
```

The resulting program remains virtually unchanged, except that macro tokens have been substituted at the points of table-specific variation; three tables (and potentially many more) are now generated from the same set of statements (meaning there will be no formatting differences between them, and any revisions are automatically applied to the entire series of tables).

The number of parameters in this example is fairly small: a 'series' number (&series) to distinguish between tables and to direct the retrieval of the appropriate titles and footnotes, the name of the input dataset (&dsn), the analysis population flag (&pop) used in record-selection, and the variable to be used in analysis (&var). Attention must be paid to the number of parameters as the program is being developed: large numbers of parameters may signal that the tables in question are not actually all part of the same series (criteria for identifying the series of repeats may need to be re-visited), or it may mean that the structure of the program itself should be revised. For instance, in some cases it may be better to use a small number of parameters, but handle table-specific requirements by means of '%if/%then' branching; this should be approached with caution though, since excessive table-specific branching will also detract from program maintainability.

HANDLING GREATER COMPLEXITY & BETWEEN-TABLE VARIABILITY

Recall that in the series shown above, differences were thematic rather than structural. More involved thematic differences (i.e. complex record-selection rules) can also be accommodated, usually by adding table-specific code branching via '%if/%then' blocks:

```
%if &dsn=adqol %then %do;

  proc sql;
    create table derived_ as select * from derived
      where usubjid in
       (select distinct usubjid from <dataset> where <criteria>);
  quit;

%end;
```

Too much of this sort of branching (and/or having code within such a branch that is too involved or extensive) will make the program unmanageable, and the cost to program maintenance should be weighed against the benefits of keeping the number of programs to a minimum.

Structural differences can also be accommodated, up to a point, and one common one is in allowing for variability in the number of decimal places presented for descriptive statistics, which can again be handled via table-specific statements (in this case taking advantage of existing parameters):

```
%if &series=1 or &series=2 %then %let precision=0;
%else %let precision=1;

data stats;
  set stats;
  mean_=put(mean,8.%eval(&precision+1));
  sd_=put(sd,8.%eval(&precision+2));
  --etc.--
```

, or alternately by increasing the number of parameters:

```
%rpeat(series=3,dsn=adqol,pop=itt,var=VAS,precision=1);
```

There are other, less obvious instances where what might appear at first to be significant structural differences between tables can actually be easily accommodated, two of which are described in the following sections.

TABLES WITH AND WITHOUT BY-VISIT STRATIFICATION

Consider the following tables:

Table 20
Summary of Baseline Score
(Safety Population)

	Group 1 (N=100)	Group 2 (N=100)	Overall (N=200)
n	xx	xx	xx
Mean (SD)	xx.x (xx.xx)	xx.x (xx.xx)	xx.x (xx.xx)
Median	xx.x	xx.x	xx.x
Min, Max	x, x	x, x	x, x

Table 20.1
Summary of On-Treatment Score, by Visit
(Safety Population)

	Group 1 (N=100)	Group 2 (N=100)	Overall (N=200)
Visit 1			
n	xx	xx	xx
Mean (SD)	xx.x (xx.xx)	xx.x (xx.xx)	xx.x (xx.xx)
Median	xx.x	xx.x	xx.x
Min, Max	x, x	x, x	x, x
p-value*	.xxx		
Visit 2			
n	xx	xx	xx
Mean (SD)	xx.x (xx.xx)	xx.x (xx.xx)	xx.x (xx.xx)
Median	xx.x	xx.x	xx.x
Min, Max	x, x	x, x	x, x
p-value*	.xxx		

Figure 4. Structurally dissimilar but 'nested' tables (horizontal stratification).

At first glance these tables seem to have significant structural differences: not only does table 20.1 include horizontal by-visit stratification, but it includes p-values for treatment comparisons that do not appear in the baseline table. Tables like these are not uncommon, where the baseline table is grouped in the analysis plan with demographic and baseline characteristics summaries, and the post-baseline summary is grouped with efficacy summaries, and the programming plan reflects this by assuming that there will be separate programs creating baseline and post-baseline summaries.

If we think about it though, we'll recognize that these tables are 'nested': a hypothetical table that provided summaries at baseline and at all post-baseline time points, and that provided p-values at all time points, would contain all of the elements required for both tables. As such, we could write a program as though we were creating this hypothetical table, and then simply exclude the extraneous rows when producing the respective baseline and post-baseline tables.

In short, this program would calculate descriptive statistics and p-values for all visits, regardless of whether table 20 or table 20.1 was being produced, and record-exclusion would take place in the final PROC REPORT step:

```
proc report data=final(where=(
                           %if &series=0 %then %do;
                             visitn=0 & statn<=4
                           %end;
                           %else %if &series=1 %then %do;
                             visitn>0
                           %end;
                           )) ... ;
```

Here we assume that rows are numbered statn=1 to 4 for the descriptive stats, and that baseline is denoted by visitn=0 and post-baseline by visitn>0. Inclusion or omission of the visit labels would be provided by a table-specific 'compute' block:

```
proc report data=final ...;
  column ...;
  define ...;

  %if &series=1 %then %do;
    compute before visitn;
      line @1 visitn visfmt.;
    endcomp;
  %end;
run;
```

So ultimately, the program only demands the complexity that would have been required to produce table 20.1 on its

own anyway, with the addition of two fairly trivial conditional steps in the PROC REPORT block, and thereby produces tables showing a certain degree of structural dissimilarity.

TABLES WITH ADDITIONAL VERTICAL STRATIFICATION

Consider the following tables:

Table 15
Summary of Adverse Events
(Safety Population)

	Group 1 (N=100)	Group 2 (N=100)	Overall (N=200)
AE1	xx (xx.x)	xx (xx.x)	xx (xx.x)
AE2	xx (xx.x)	xx (xx.x)	xx (xx.x)
(etc.)	xx (xx.x)	xx (xx.x)	xx (xx.x)

Table 15.1
Summary of Adverse Events by Maximum Severity
(Safety Population)

	Group 1 (N=100)				Group 2 (N=100)				Overall (N=200)			
	Mild	Moderate	Severe	Total	Mild	Moderate	Severe	Total	Mild	Moderate	Severe	Total
AE1	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)
AE2	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)	xx(xx)
(etc.)												

Figure 5. Structurally dissimilar but 'nested' tables (vertical stratification).

Again, these tables appear to have significant structural differences, specifically the additional level of vertical stratification in table 15.1, and this is usually reflected by separate programs being written, probably named something like 't_ae.sas' and 't_ae_severity.sas'. However, as with the earlier example, these tables are actually 'nested': the columns in table 15 are simply the 'Total' columns from table 15.1 (although there may be imputation rules that apply to the determination of maximum severity, the values in table 15 and the values in the 'Total' columns of table 15.1 are simply counts irrespective of severity).

What these tables illustrate is the importance of 'looking ahead' when deciding how to organize your programming efforts: the temptation is always to tackle the 'basic' AE table first, and then work out the additional complexity of the 'by severity' table later, and in a separate program. But if we program the by-severity table first, we end up with everything we need for the basic table: we can simply exclude all but the 'Total' columns, giving us table 15. So our PROC REPORT code would look something like this:

```
proc report data=final split="|"...;
  %if &series=0 %then %do;
    column aeterm ("Group 1|N=(&n1)" tot1)
                  ("Group 2|N=(&n2)" tot2)
                   --etc.--
  %end;
  %else %if &series=1 %then %do;
    column aeterm ("Group 1|N=(&n1)|---" mild1 mod1 sev1 tot1)
                  ("Group 2|N=(&n2)|---" mild2 mod2 sev2 tot2)
                   --etc.--
  %end;

  %if &series=0 %then %do;
    define tot1 / " " width= ...;
    --etc.--
  %end;
  %else %if &series=1 %then %do;
```

```
  define mild1 / "Mild"     width= ...;
  define mod1  / "Moderate" width= ...;
  define sev1  / "Severe"   width= ...;
  define tot1  / "Total"    width= ...;
  --etc.--
%end;
```

Once again, aside from these modest additions to PROC REPORT, the program consists of nothing more than the complexity that would have been demanded by table 15.1 on its own, with table 15 being produced simply by excluding extraneous columns.

CONCLUSION

Typical validation practices (those involving independent double-programming) may be effective in trapping genuine calculation errors, but they often don't address inconsistency in style of presentation. They also may not catch inconsistencies in the assumptions made about data sources and record-selection, since often QC programmers will simply make inferences based on the values produced by the production code, in striving to eliminate discrepancies rather than to test underlying assumptions and the interpretation of requirements. These issues of 'internal consistency' therefore often fall to senior reviewers to identify, and often at a late stage in the project. Without adequate project planning or training, and particularly with the adoption of the '1:1' paradigm in allocating and programming repeat TFLs, issues of internal consistency can become the proverbial Achilles' Heel.

By planning out your programming effort to assign related and repeat TFLs to the same programmer, and to produce multiple repeat TFLs from single programs, not only can threats to internal consistency be vastly reduced, but when revisions are required they can be done in an efficient and streamlined manner. Like with CDISC conventions that recommend developing the 'right' number of analysis datasets, TFL programming plans should apply similar art in categorizing TFLs along structural and thematic lines, evaluating the tradeoff between minimizing the total number of programs and the complexity of individual programs, in ultimately determining the 'right' number and organization of TFL programs.

ACKNOWLEDGMENTS

The authors would like to thank our employer inVentiv Health Clinical for supporting our participation in PharmaSUG, along with all our friends at inVentiv, especially Angela Ringelberg for encouraging creativity and sharing of ideas.

CONTACT INFORMATION

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

Name: Tracy Sherman
Enterprise: inVentiv Health Clinical
E-mail: Tracy.Sherman@InVentivHealth.com
Web: www.inVentivHealthclinical.com

Name: Brian Fairfield-Carter
Enterprise: inVentiv Health Clinical
E-mail: Brian.Fairfield-Carter@InVentivHealth.com
Web: www.inVentivHealthclinical.com