PharmaSUG 2017 - Paper AD17

Metadata integrated programming

Jesper Zeth, Jan Skowronski, Novo Nordisk A/S

ABSTRACT

With the growing complexity of pharmaceutical projects it is becoming increasingly relevant to address how we are programming, and how we ensure that our programming is as flexible as possible with regards to implementing changes and re-using programs across projects. It is essential to be able to implement requested changes as fast as possible, with certainty that the changes are implemented everywhere needed, and that we are able to scale changes across projects, trials and subjects.

One of the important tools that can be used in this context is integrating metadata in both data and output programming. When designed properly the use of metadata in programming provides the ability to implement a variety of changes across datasets and outputs, by only changing the actual metadata, making changes both very fast and uniform.

The purpose of this paper is to illustrate how metadata can be integrated in both data and output programming tasks. The examples in the paper are based on experience acquired as part of a Novo Nordisk submission team. Metadata was used in all data and output programming to a large extent, and a number of tools using these metadata were developed. Finally the paper will elaborate on learnings following the chosen metadata model and the challenges that were experienced in the process.

INTRODUCTION

In this paper we present some of our experiences and considerations in relation to moving from a copy-paste programming approach to an approach relying on metadata, and how metadata in this context can be used to reduce hard-coding, and increase transparency, in our programming. The content of the paper is primarily based on our experience with SAS® programming, and in particular what we have experienced during our recent projects.

DIFFERENT APPROACHES TO PROGRAMMING

At a very high level three different types of programming approaches can be identified differentiated by the level of re-usability, complexity in programming, and flexibility in relation to changes.

COPY-PASTE PROGRAMMING

The first approach is what can be described as *"copy-paste programming"* in which the programmers approach often is *"I have earlier created a output that looks very much like the requested, and I have the program right here which will work if I just change…."*. A first shot at the output is very quickly produced and the programmer can move on to the next requested output.

In this approach data/study specific issues are handled in each program by hard coding *"if studyid eq 123 then"*, code is replicated across programs, and following this, changes and maintenance have to be implemented in several programs. To implement changes and new requirements, when working in an environment like this, requires a very detailed knowledge regarding all programs and data. There is a high risk of inconsistency when a change is made in one program but not others, as well as it will be very time consuming to implement changes.

CODE REPOSITORY

In the *"code repository"* approach definitions, and program fragments are placed in a code repository and afterwards these programs are used in the data and output-programming.

This approach accommodates a lot of the issues related to the "copy-paste programming" approach as it is possible to place data/study specific issues directly in the code repository, i.e. the code *"if studyid eq*

123 then" will be placed in and used from the code repository. However still all the data/study specific issues are hard coded into the programs in the code repository, which depending on complexity can be a more or less difficult task to keep track of. In a small scale, e.g. a single study, the code repository can be a manageable solution, but as a given project or task grows in complexity the code repository will increase making it more error-prone as the code gets more and more complex.

METADATA INTEGRATED PROGRAMMING

In "*metadata integrated programming"* all data/study specific issues in output and data programming are read and maintained from metadata, and thereby special cases, that in the two previous sections required hardcode, are separated from the programming. Metadata specific issues are continuously identified, added and maintained as a project progresses. When a hard-coded and/or repeated item is identified, the item is analysed and, if deemed feasible, it is added to the metadata to be used in the programs as a metadata-read-value. Special cases are in other words moved from programs into metadata, leaving the programs more readable as metadata are more transparent. Programming in this way requires the programmers continuous focus on using metadata, avoiding hard-coding and accept that the identification of a new metadata issue might involve re-programming when new metadata has been added. In the short term the re-programming will take up additional time, but in long term this additional time will be saved when massive changes are requested across outputs.

WHAT IS METADATA?

The definition of metadata is subject to much discussion as to what is precisely meant by the concept, spanning from the common "data about data"[1] to a more specific and detailed description involving purpose of the metadata, metadata users, and context of the metadata usage. In this paper we adopt the definition from National Information Standards Organization (NISO) stating metadata as "structured information that describes, explains, locates, or otherwise makes it easier to retrieve, use, or manage an information resource"[2].

There are different sources of metadata that are relevant to address in the context of programming in the pharma industry, of which two major sources are; SAS® standard metadata, and User-defined metadata.

Several sources exist on SAS standard metadata of which the SAS dictionary tables are the most applicable in both data and output programming. SAS dictionary tables represent a very usable "standard metadata repository" in which the SAS System on-the-fly keeps track of all datasets, attributes, formats, indices etc. in any libname assigned in a given SAS session. The dictionary tables have many applications, and can assist programming in multiple ways[3]. Massive resources can be found on this in the SAS literature.

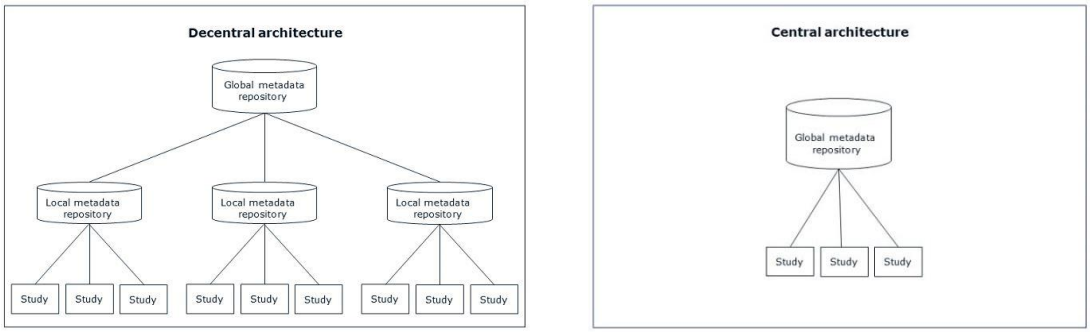
User-defined metadata is the main target of this paper, and is as the name says, data defined with a specific programming and/or configuration purpose in mind. The purpose of user defined metadata is to reduce or remove hard-coding from programming, avoid repeated code, and to centralize implementation of the same changes across outputs and/or datasets. A simple example could be labelling of a sub group, implemented in several outputs. The labelling could potentially be subject to change as programming progresses requiring changes in all programs. If all programs read the sub group labelling from the user defined metadata, a change in the labelling can be implemented simply by changing the metadata making the change very time efficient.

---

[1] John E. Bentley. 2001. "Metadata: Everyone Talks About It, But What Is It?", *Proceedings of the Data Warehousing and Solutions section SUGI*, 125-26. Long Beach California.
[2] National Information Standards Organization (NISO) (2004). "Understanding metadata", NISO Press, ISBN: 1-880124-62-9.
[3] Kirk Paul Lafler. 2005. "Exploring DICTIONARY Tables and Views", *Proceedings of the coders corner SUGI*, 070-30. Philadelphia, Pennsylvania.

ORGANISING AND MAINTAINING USER-DEFINED METADATA

All metadata are stored in a repository and defining and working within this repository requires some considerations in relation to how metadata is organised and anchored in an organisation. In a high level perspective you can differentiate between a centralized and a decentralized architecture[4]. In a centralized architecture only one metadata repository exist in which all metadata is stored and maintained, and all metadata usage will be based on this repository. In a decentralized architecture both a centralized and a decentralized repository coexist. The central repository contains all uniform metadata elements across projects, whereas the local repository contains the project specific metadata elements. The two architectures are illustrated in the figure below.



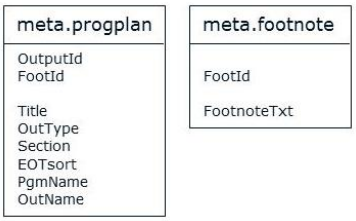
In the decentralized architecture the local repositories overrule the central repository when needed, and make up for gaps that the global metadata cannot cover. Metadata in the decentralized repository is considered as standalone instance with their own administration and content.

The decentralized architecture is what comes closest to the “architecture” we have been working in. In our latest project we received a large amount of metadata from a global repository but a substantial amount of user-defined metadata was added in the local, or project specific, metadata repository that we used in our programming. In our case the user-defined metadata was entered, and maintained, using Microsoft Excel. All files placed in one directory for easy and logical access. A dedicated program imported and transformed the Excel-files into SAS datasets that were loaded to a separate metadata libname constituting our metadata repository. All metadata was scheduled for update several times a day in addition to ad-hoc updates when we received urgent requests.

APPLYING USER-DEFINED METADATA IN PROGRAMMING

Metadata can be applied in programming by any combination of “data directly applied in programming”, and “utility programs developed with any given programming task in mind”.

In the example below two metadata datasets are defined; `meta.progplan` containing a complete list of all outputs and related metadata, and `meta.footnote` containing all footnotes across outputs.



When working on a given output, in this example identified by `OutputId eq 12345`, the metadata can be queried as described below.

[4] Presentation by David Marco, London 2002 ”Building and Managing the Meta Data Repository”.

In the simplest case we use SQL to extract all needed metadata by creating macro variables to be referenced directly in the programming.

```
proc sql ;
  select a.title,
         a.outName,
         a.pgmName,
         b.footnoteTxt
  into   : outputTitle,
         : outputName,
         : outputProgramName,
         : outputFootnote
  from  meta.progplan  as a
        left join
        meta.footnote as b
        on a.footId eq b.footId
  where a.outputId  eq 12345;
quit;
```

Here the title, output name, program name and footnote is placed directly in distinct macro variables.

A more reusable solution applied on the metadata would be to develop a utility macro that reads it and creates the required macro variables as shown in the example below.

```
%macro readOutputMetadata(mLib=meta,  outputId=);
  %global outputTitle outputName outputProgramName;
  data _null_;
    set &mLib..progplan;
    where outputId  eq "&outputId.";
    call symput('outputTitle',title);
    call symput('outputName',outName);
    call symput('outputProgramName',pgmName);
  run;
%mend  readOutputMetadata;

%macro readOutputFootnote(mLib=meta, outputId=);
  %global outputFootnote;
  proc sql noprint ;
    select footnoteTxt into : outputFootnote
    from &mLib..footnote
    where footId in
      (select footId
       from &mLib..progplan
       where outputId eq "&outputId.") ;
  quit;
%mend readOutputFootnote;

%macro outputMetadata(mLib=meta,  outputId=);
  %readOutputMetadata(outputId=&outputId.);
  %readOutputFootnote(outputId=&outputid.);
%mend outputMetadata;
```

The “macro-fication” of the metadata extraction routine results in the following simple one liner ready to be applied in the output programming.

```
%outputMetadata(outputId=12345);
```

Following the execution of the macro we have a number of macro variables ready for use in the programming.

Both direct querying and macro enabled metadata has the effect of removing, or at least reducing, hard-coding from the programming, and both can be applied in output as well as data programming. The important point in this context is the use of metadata in the programming and not so much how the metadata is accessed.

IDENTIFYING METADATA

When working with metadata integrated programming the elements that should be added to the repository needs to be identified and decision on how to use these elements needs to be made.

In our latest task metadata integrated programming was applied when performing a pooling of studies used for an integrated summary of safety and efficacy. In the process of designing the metadata the following important elements were considered.

The first task was to identify which **studies** were needed in the pooled database, and how to make this element scalable with regards to adding more studies if/when required. Each study resides in a separate standardized folder structure. This information was stored in metadata along with various study level characteristics such as trial phase and various study specific groupings that made sense from a reporting view.

studyid	include	instance	pool1	pool2	pool3	pool4	pool5	pool_ph1	pool_ph2	pool_ph3	all
nn-study-1	1	instance_a_20161107	1	1	1	0	1	0	0	1	1
nn-study-2	1	instance_c_20161107	1	1	0	1	1	0	0	1	1
nn-study-3	1	instance_g_20161107	1	0	0	1	1	0	0	1	1

Using this metadata we were able to assign all required libnames to access the required data for the data pooling. If any of the study source data was updated all that was required to access the new data, was to update metadata.

Study design plays an important part in deriving treatment periods, and is required in both data and output programming. A metadata dataset with flags by study on cross-over design, drug-drug design, single-dose, multiple-dose, and if the study was a phase 1, 2 or 3 was defined.

phase	cross-over	single-dose	multiple-dose	drug-drug
3	0	1	0	0
3	0	1	0	0
1	1	1	0	0

In the output programming the major part of the tables and plots were based on metadata to some extent. All outputs were registered in the **programming plan** metadata, where each output can be uniquely identified. In the programming plan several output related information was added such as title, orientation, output name, footnotes etc.

Output Type	Output Title	Program Name	Output Folder	Output Name	Pgm ID	EOT Section	Programmer (Init)
Figure	Figure output title 1 - Treatment a	f_output_by_treatment	Outputs	f_output_by_treatment_a	51012	6.2	PROG_A
Figure	Figure output title 2 - Treatment b	f_output_by_treatment	Outputs	f_output			
Figure	Figure output title 3 - Treatment c	f_output_by_treatment	Outputs	f_output			
Figure	Figure output title 4 - Treatment d	f_output_by_treatment	Outputs	f_output			
Table	Table output title 1 - Treatment a	t_output_by_treatment	Outputs	t_output			
Table	Table output title 1 - Treatment b	t_output_by_treatment	Outputs	t_output			

Programmer (Init)	EOT Sorting ID	Orientation	Footnote ID's
PROG_A	620000000101	landscape	DP01#DP02#DP03#SN01#SN02#SN07
PROG_A	620000000102	landscape	DP01#DP02#DP03#SN01#SN02#SN07
PROG_A	620000000111	landscape	DP01#DP02#DP03#SN01#SN02#SN07
PROG_A	620000000111	landscape	DP01#DP02#DP03#SN01#SN02#SN07
PROG_B	620000000120	landscape	DP01#DP02#DP03#SN01#SN02#SN07
PROG_B	620000000122	landscape	DP01#DP02#DP03#SN01#SN02#SN07

The “Footnote ID’s” variable in the programming plan above contains a string of footnote ids, separated by #, as each output can contain several footnotes. Each id in the string refers to a **footnote** that is registered in a separate metadata dataset making it possible to re-use footnotes across outputs. The selected footnotes are afterwards made available for output programming by a macro loading the footnotes in to macro variables.

name	content
DP01	BMI: Body mass index (kg/m^2)
DP02	SBP: Systolic blood pressure
DP03	DBP: Diastolic blood pressure

Labelling, ordering and set of values were also an element that was added as metadata. Using this data enabled the control of labelling and ordering of variable values in any outputs, but the data also enabled the programs to address the complete set of outcomes even if there were missing categories[5] . For example a summary of gender will have the outcome set of {‘M’, ‘F’}. If the category ‘M’ was missing in the clinical database, the metadata ensured that this information was not lost when displayed. The actual metadata dataset has the structure, and for this example the content, illustrated below.

varname	value	ord	descrpt
sex	m	1	Male
sex	f	2	Female

[5] See the SAS PRELOADFMT for same functionality.

A **statistics format** was added to the metadata repository in order to control the display of numbers in outputs. The statistics formats were added using a format code (`formatcd`) and statistics (`value`) as a combined key, as in this way the same statistic can be formatted with different precision simply by adding a new format code.

formatcd	value	format
default	n	6.
default	mean	8.2
default	median	6.2
default	min	8.2
default	max	8.2
lab1	n	6.
lab1	mean	8.3
lab1	median	6.2

In the example above `formatcd eq 'default'` provides the necessary display formats needed for that particular case.

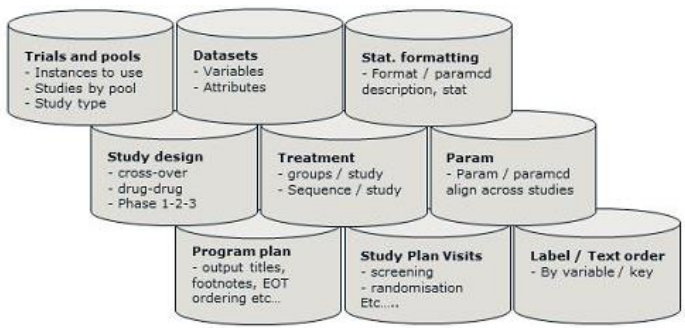
As illustrated below all elements in the table are to some extent derived from metadata.

	Title obtained from metadata			
Labeling Descriptors obtained from metadata	**Grouping labels obtained from metadata**			
	Derivations with display formats obtained from metadata			
	xxx	xxx	xxx	xxx
	xxx	xxx	xxx	xxx
	xxx	xxx	xxx	xxx
	xxx	xxx	xxx	xxx
	xxx	xxx	xxx	xxx
	Footnotes obtained from metadata			

A particular central metadata element in the model was the "**dataset attributes**" dataset which contained all variables and datasets in the pooled database. All datasets was created based on the registered attributes in this metadata ensuring that all attributes were aligned across variables and datasets. Furthermore these metadata were used as the foundation of the define.xml on the pooled database.

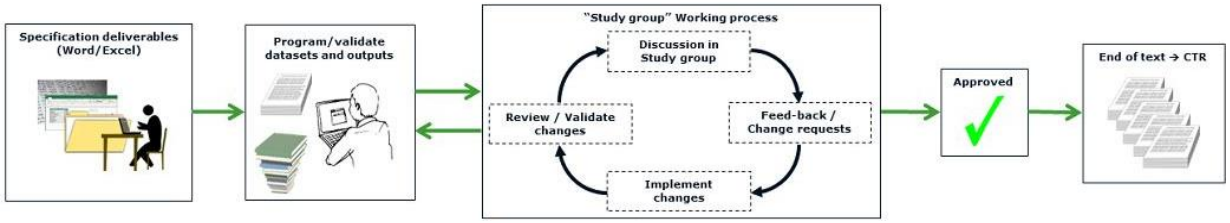
memname	name	type	length	format	ctrlterm	term	label	var order
ADSL	STUDYID	char	17				Study Identifier	**1**
ADSL	USUBJID	char	27				Unique Subject Identifier	2
ADSL	SUBJID	char	9				Subject Identifier for the Study	3
ADSL	SITEID	char	15				Study Site Identifier	4
ADSL	AGE	num	8	3.			Age	5

The figure below lists the user defined metadata we have been working with in our latest task.



THE "REAL-LIFE" SETTING

In a real life setting a typical working process is illustrated in the figure below, in which the joint work of the study group, statisticians, and programmers result in a Clinical Trial Report (CTR) as the final output.



The different outputs are initially specified in a number of table shells often delivered as a number of Microsoft Word/Excel documents. Following this the programming starts, delivering outputs to a study group, which gives feed-back on the outputs, resulting in change requests on the outputs which in turn has to be implemented and reviewed, and once again discussed in the study group. The study group approves the outputs resulting in the end of text, and finally the CTR.

In the following the three different programming approaches will be described in relation to the working process and the "real-life-setting", illustrating the pros and cons in the different approaches.

THE COPY-PASTE PROGRAMMING APPROACH IN THE "REAL-LIFE" SETTING

Using the copy-paste programming approach, the actual output programming will start once the deliverables specification is finished. Each programmer involved in the task will have a look in their respective archive of programs, and probably find a program with some resemblance. In this program the necessary adjustments will be implemented, and quickly the first draft of an output will be in place for the study group discussion. The archived programs will be based, not only on the current specification, but will be influenced by all other specifications that the program has been used for in earlier cases, and therefore the structure of the program will not necessarily reflect the current specification. Across outputs, and data definitions, any number of definitions might be common, but this is not identified ahead of programming, and therefore the programming will be characterized by repetitive programming, and redundant code. As a consequence of this approach the programming will most likely suffer from a lack of transparency as each program will have its own structure and logic.

It might be possible to keep track of structure and definitions in one isolated program but looking across the complete programming task it will quite likely become unmanageable. Implementing bulk changes, following a request from the study group, will at best be very time consuming and costly, and in worst case scenarios the implementation will be close to impossible. Finally troubleshooting errors in the programs will become inherently harder as the amount of code grows and the "special cases" can very easily get "lost in translation". The code is simply hard to maintain, and more error prone.

The example below shows a very simple piece of programming – a `PROC TEMPLATE` with colours and symbols hard coded in the programs. At first sight this is very straight forward and simple but as the number of programs increase it is hard to keep track of which colours and symbols is used in which programs on different studies.

Program from trial a containing code

```
proc template;
  define statgraph scatterplot;
    begingraph;
      discreteattrmap name="symbols" / ignorecase=true ;
        value "m" / markerattrs=(color=blue symbol=diamondfilled);
        value "f" / markerattrs=(color=red symbol=circlefilled );
      enddiscreteattrmap ;
      discreteattrvar attrvar=groupmarkers var=sex attrmap="symbols" ;
      layout overlay;
        scatterplot x=height y=weight / name="scatter" group=groupmarkers ;
      endlayout;
    endgraph;
  end;
run;
proc sgrender data=sashelp.class template=scatterplot;run;
```

Program from trial b containing code

```
proc template;
  define statgraph scatterplot;
    begingraph;
      discreteattrmap name="symbols" / ignorecase=true ;
        value "m" / markerattrs=(color=cx16a629 symbol=x);
        value "f" / markerattrs=(color=cx13478c symbol=y);
      enddiscreteattrmap ;
      discreteattrvar attrvar=groupmarkers var=sex attrmap="symbols" ;
      layout overlay;
        scatterplot x=height y=weight / name="scatter" group=groupmarkers ;
      endlayout;
    endgraph;
  end;
run;
proc sgrender data=sashelp.class template=scatterplot;run;
```

THE CODE REPOSITORY APPROACH IN THE "REAL-LIFE" SETTING

Using a code repository programming approach accommodate a lot of the problems characterizing the copy-paste programming approach. Common definitions and values are identified, and placed in a central code repository. This approach requires an initial analysis of programming requirements following the specification of deliverables, and a substantial part of the required repository code will be identified in this initial analysis. The actual programming will not be started as fast as was the case in the "copy-paste programming approach" as the analysis will have to be performed up front. As the task progresses and the study group discuss and request changes in outputs, new pieces of code will be added to the code repository, and other pieces of code will be changed.

Following that all definitions is placed in the central code repository all maintenance will be handled only in the repository making changes more manageable, and in this way bulk changes can also be implemented in a more flexible manor as changes in the central repository automatically will be implemented in the different output programs. On the other side adding new definitions to the code repository might require some of the earlier output programming to be revisited in order to implement the new definition using the code repository. However in relation to the working process this program revisit should be more than made up for by the centralization of definitions causing changes to be implemented in the same way in all relevant programs.

The example below shows how the code repository could be used to add colours and symbols in a `PROC TEMPLATE`. Note that although the name of the macro conveys the meaning, the eyes still need to trace the macro-code in order to get an overview of what is actually happening. Also, as can be seen, this simple task requires a lot of if-then processing.

Code repository

```
%macro plotSymbCol(projectId=,trialId=,param=,value=,colorVar=,symbolVar=);
  %global &colorVar. &symbolVar.;
  %if "&projectId." eq "12345" %then %do;
    %if %sysfunc(lowcase("&trialId.")) eq "a" %then %do;
      %if "&param." eq "sex" %then %do;
        %if "&value" eq "m" %then %do;
          %let col =blue; %let sym =diamondfilled;
        %end;
        %else %if "&value" eq "f" %then %do;
          %let col =red; %let sym =circlefilled;
        %end;
      %end;
      %goto setVars;
    %end;
    %else %if %sysfunc(lowcase("&trialId.")) eq "b" %then %do;
      %if "&param." eq "sex" %then %do;
        %if "&value" eq "m" %then %do;
          %let col =cx16a629; %let sym =x;
        %end;
        %else %if "&value" eq "f" %then %do;
          %let col =cx13478c; %let sym =y;
        %end;
      %end;
      %goto setVars;
    %end;
  %end;
  %setVars:
    %let &colorVar.  = &col.;
    %let &symbolVar. = &sym.;
%mend plotSymbCol;
```

Program from trial a using code repository

```
%plotSymbCol(projectId=12345,trialId=a,param=sex,value=m,
             colorVar=mcolor,symbolVar=msymbol);
%plotSymbCol(projectId=12345,trialId=a,param=sex,value=f,
             colorVar=fcolor,symbolVar=fsymbol);

proc template;
  define statgraph scatterplot;
    begingraph;
      discreteattrmap name="symbols" / ignorecase=true ;
        value "m" / markerattrs=(color=&mcolor. symbol=&msymbol.);
        value "f" / markerattrs=(color=&fcolor. symbol=&fsymbol. );
      enddiscreteattrmap ;
      discreteattrvar attrvar=groupmarkers var=sex attrmap="symbols" ;
      layout overlay;
        scatterplot x=height y=weight / name="scatter" group=groupmarkers ;
      endlayout;
    endgraph;
  end;
run;
proc sgrender data=sashelp.class template=scatterplot;run;
```

Program from trial b using code repository

```
%plotSymbCol(projectId=12345,trialId=b,param=sex,value=m,
             colorVar=mcolor,symbolVar=msymbol);
%plotSymbCol(projectId=12345,trialId=b,param=sex,value=f,
             colorVar=fcolor,symbolVar=fsymbol);

proc template;
  define statgraph scatterplot;
    begingraph;
      discreteattrmap name="symbols" / ignorecase=true ;
        value "m" / markerattrs=(color=&mcolor. symbol=&msymbol.);
        value "f" / markerattrs=(color=&fcolor. symbol=&fsymbol. );
      enddiscreteattrmap ;
      discreteattrvar attrvar=groupmarkers var=sex attrmap="symbols" ;
      layout overlay;
        scatterplot x=height y=weight / name="scatter" group=groupmarkers ;
      endlayout;
    endgraph;
  end;
run;
proc sgrender data=sashelp.class template=scatterplot;run;
```

METADATA INTEGRATED PROGRAMMING APPROACH IN THE "REAL-LIFE" SETTING

Using a metadata integrated programming approach shares a lot of the characteristics from the code repository approach as metadata integrated programming also rely on a code repository containing a number of macros used to enable the metadata usage in data and output programming. The major difference between the two approaches is how definitions is stored and implemented. In the metadata integrated programming it is attempted to put as much definitions as possible in datasets, and creating a code repository without hard-coding values. This approach also requires an initial analysis of programming requirements following, or in parallel with, the specification of deliverables, and therefore the actual data and output programming will not be started as fast as was the case in the "copy-paste programming approach". In this initial analysis a large part of metadata will be identified, and the metadata model will be determined. In this initial analysis phase it is also decided how to work with the metadata, and following this, which tools/utilities that has to be developed, and how these tools/utilities should be used in the programming.

In the process of programming, and the study group discussion, new metadata requirements will be identified as well as adjustments to both metadata and utilities will be required, and this interacting process will influence programming all the way until the final end-of-text has been generated, and all outputs, and data, have been approved in the study group. However, given the characteristics of the model, many of the across data / outputs changes can be handled by only modifying the metadata, and

thereby parsing changes to all programs using the metadata. Compared to the code repository the major difference is that adjusting a definition is handled in metadata instead of having to identify how and where to add or adjust the programming in a piece of code.

Using once again the `PROC TEMPLATE` example, the solution using metadata integrated approach is illustrated below. In this example all colouring and symbols are placed in the metadata dataset `meta.plotsym` with `projectid`, `studyid`, `param`, and `value` being the key. The metadata usage have been enabled through the macro `metaPlotSymbCol` creating specified macro variables, containing the required colours and symbols, to use in the output programming as illustrated in the two program examples.

meta.plotsym

ProjectId
StudyId

Param
Value
Color
Symbol

projectid	studyid	param	value	color	symbol
12345	a	sex	m	blue	diamondfilled
12345	a	sex	f	red	circlefilled
12345	b	sex	m	green	x
12345	b	sex	f	orange	y
12345	c	sex	m	yellow	circle
12345	c	sex	f	black	square

```
%macro metaPlotSymbCol(mlib=meta,projectId=,studyId=,
                       param=,value=,colorVar=,symbolVar=);
  %global &colorVar. &symbolVar.;
  proc sql noprint;
    select color, symbol
    into : &colorVar., : &symbolVar.
    from &mlib..plotsym
    where projectId eq "&projectId." and studyId eq "&studyId." and
      param eq "&param." and value eq "&value.";
  quit;
%mend metaPlotSymbCol;
```

Program from trial a using metadata

```
%metaPlotSymbCol(projectId=12345,studyId=a,param=sex,value=m,
                 colorVar=mcolor,symbolVar=msymbol);
%metaPlotSymbCol(projectId=12345,studyId=a,param=sex,value=f,
                 colorVar=fcolor,symbolVar=fsymbol);
proc template;
  define statgraph scatterplot;
    begingraph;
      discreteattrmap name="symbols" / ignorecase=true ;
        value "m" / markerattrs=(color=&mcolor. symbol=&msymbol.);
        value "f" / markerattrs=(color=&fcolor. symbol=&fsymbol. );
      enddiscreteattrmap ;
      discreteattrvar attrvar=groupmarkers var=sex attrmap="symbols" ;
      layout overlay;
        scatterplot x=height y=weight / name="scatter" group=groupmarkers ;
      endlayout;
    endgraph;
  end;
run;
proc sgrender data=sashelp.class template=scatterplot;run;
```

Program from trial b using metadata

```
%metaPlotSymbCol(projectId=12345,studyId=b,param=sex,value=m,
                 colorVar=mcolor,symbolVar=msymbol);
%metaPlotSymbCol(projectId=12345,studyId=b,param=sex,value=f,
                 colorVar=fcolor,symbolVar=fsymbol);
proc template;
  define statgraph scatterplot;
    begingraph;
      discreteattrmap name="symbols" / ignorecase=true ;
        value "m" / markerattrs=(color=&mcolor. symbol=&msymbol.);
        value "f" / markerattrs=(color=&fcolor. symbol=&fsymbol. );
      enddiscreteattrmap ;
      discreteattrvar attrvar=groupmarkers var=sex attrmap="symbols" ;
      layout overlay;
        scatterplot x=height y=weight / name="scatter" group=groupmarkers ;
      endlayout;
    endgraph;
  end;
run;
proc sgrender data=sashelp.class template=scatterplot;run;
```

METADATA INTEGRATED PROGRAMMING VERSUS METADATA GENERATED PROGRAMMING

How far one wants to take metadata driven programming is a matter of personal flavour, bearing in mind that there seems to be a trade-off between uniformity and complexity with increased metadata usage. In the extreme case metadata can be used as the actual program generator, and following this an output or data program would primarily contain different macro calls generating and executing the code.

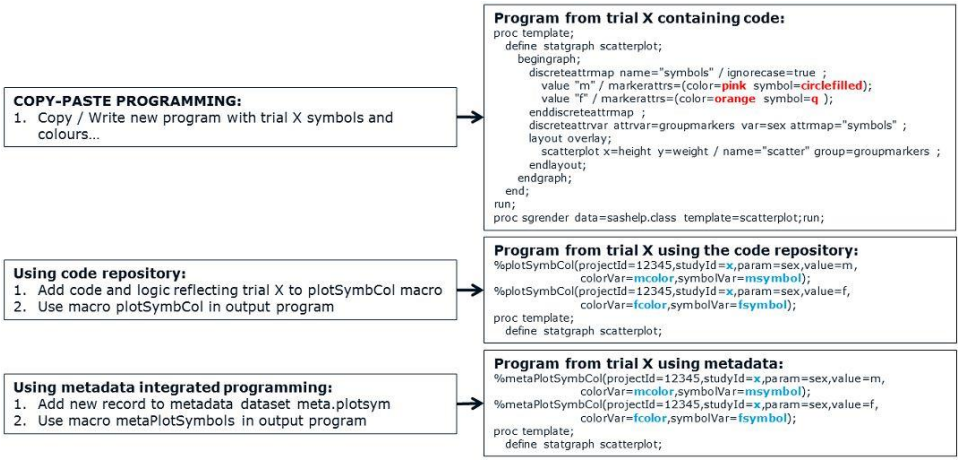
```
%macro metaCreateValueStatement(mlib=meta,projectId=,studyId=,param=,
                                outValVar=,outCountVar=,sepchar=@);
  %global &outValVar. &outCountVar.;
  proc sql noprint;
    select 'value' || ' "' || value || '" ' || '/ markerattrs=(color=' || color ||' symbol=' ||symbol || ');'
    into : &outValVar. separated by "&sepchar."
    from &mlib..plotsym
    where projectId eq "&projectId." and studyId eq "&studyId." and param eq "&param.";
  quit;
  %let &outCountVar. = &sqlobs;
%mend metaCreateValueStatement;

%macro plot(plotGroups=,plotNgrps=,sepchar=@);
  proc template;
    define statgraph scatterplot;
      begingraph;
        discreteattrmap name="symbols" / ignorecase=true ;
          %do i = 1 %to &plotNgrps;
            %scan(&plotGroups, &i, &sepchar.);
          %end;
        enddiscreteattrmap ;
        discreteattrvar attrvar=groupmarkers var
        layout overlay;
          scatterplot x=height y=weight / name=
        endlayout;
      endgraph;
    end;
  run;
%mend plot;
```

```
%metaCreateValueStatement(projectId=12345,studyId=a,param=sex,outValVar=groups,outCountVar=ngrps,sepchar=@);

%plot(plotGroups=&groups,plotNgrps=&ngrps,sepchar=@);

proc sgrender data=sashelp.class template=scatterplot;run;
```

Looking at the earlier `PROC TEMPLATE` example in the setup above, two macros generate the required code; one macro extracting the metadata and generating the required value statements with the defined attributes, and one macro using the value statements in the `PROC TEMPLATE`. The output program will only consist of the two macros being executed, and the `PROC SGRENDER` statements, and the program will in this way be very simple but at the same time it is not very transparent in the code itself. Any analysis of the code will require the programmer to have very detailed knowledge of the available metadata and macros generating the programming.

SCALABILITY AND FLEXIBILITY IN THE DIFFERENT PROGRAMMING APPROACHES

The three programming approaches are compared in the illustration below in relation to flexibility and scalability in the case of adding a new study output, i.e. a new study is added in a pooled database, and an existing output has to be produced for this study.



In the copy-paste programming approach the existing program will be copied, and reproduced using the new study data, and the new colouring and symbols used for this trial. In the code repository approach the new trial colouring and symbols logic are added to the macro `plotSymbCol`, and this macro can afterwards be used in the output program. In the metadata integrated programming approach the new colouring and symbols related to the new trial is added in metadata, and the macro `metaPlotSymbCol` can afterwards be used in the output program.

In smaller programming tasks the copy-paste programming approach might be an ok approach, but it will not be manageable in any larger projects, and the programming will only to a small extent be re-usable. Both code repository and metadata integrated programming approaches are relevant to consider when any given task is more than an ad hoc request.

As can be seen in the last two examples the metadata integrated programming approach resembles the code repository approach very much, as in both cases the new trial data is enabled in the output programming using a macro execution. The important difference in the two examples is how the programming information is stored and maintained, and to what extent the information is transparent afterwards. As the usage of the macros in the examples increase, more and more values will have to be added for the macro to handle, and in turn this will introduce an increasing complexity in the macro if all values are to be added directly in the code following the extreme case of using a code repository. It will become increasingly difficult to add new values, but also to read the code, and keep track of which values resolving from which conditions. Letting the values and keys reside in metadata, separated from the actual code, can simplify the required code, and also make the outcome more transparent. Any programmer can at any time look at the metadata, and given the key, find the values used in the programming.

CONCLUSION

Selecting the right programming approach is NOT black and white as, in any realistic scenario, none of the described approaches will be able to handle all cases. All projects will most likely contain some elements from all approaches, but it is important to do an initial analysis of the task size, complexity, and determine how volatile the programming is expected to be. And on this basis make a conscious decision on which of the three approaches that is to be the most dominant.

We realise that examples/cases in the paper are biased to illustrate the advantages of metadata integrated programming and that in other cases a well-documented and well programmed code repository can be quite effective as well. Having said this it is our experience that metadata integrated programming is the most feasible programming approach providing the most advantages.

Summing up, metadata integrated programming can

- increase re-usability of programming
- increase flexibility in relation to implementing changes across outputs
- increase transparency of programming
- increase scalability of programming
- reduce complexity in programming

Following these points, metadata integrated programming will be the less error-prone programming approach.

During our recent projects we acquired some experience in relation to both using metadata in programming, as well as how metadata-usage has been perceived by programmers not familiar with programming in this way. Part of the experience can be summarised in the following points:

Metadata Quality is very important and needs constant attention. Therefore a designated owner is required. This role is typically taken up by the project statistical programmer. It is important that this person at all time tracks the metadata and ensures that they are correct and to the point.

Changing metadata requires validation of programs. Each time metadata is updated it is necessary to validate that the changes do not have any side effects, and it is therefore necessary to create some check programs that can be executed following a metadata update.

Be ready to fight the "copy-paste programming approach" / "we are used to…." argument continuously. All programmers not familiar with programming using metadata will have a private "box of programs" that has been used earlier, and therefore considered the "easy-road".

Make sure that all programs use metadata. It is vital to perform regular code reviews ensuring the metadata is used, that it is used correct, and that all programmers have understood the model and work accordingly.

REFERENCES

John E. Bentley. 2001. “Metadata: Everyone Talks About It, But What Is It?”, *Proceedings of the Data Warehousing and Solutions section SUGI*, 125-26. Long Beach California.

National Information Standards Organization (NISO) (2004). “Understanding metadata”, NISO Press, ISBN: 1-880124-62-9.

Kirk Paul Lafler. 2005. “Exploring DICTIONARY Tables and Views”, *Proceedings of the coders corner SUGI*, 070-30. Philadelphia, Pennsylvania.

Presentation by David Marco, London 2002 ”Building and Managing the Meta Data Repository”.

ACKNOWLEDGMENTS

The authors thank Evelyn Guo and Gary Moore, Applications Development Section Chairs for accepting our abstract and paper; Jan Trangeled Knudsen, Jesper Hoegh Bager, Magnus Mengelbier, and Signe Olrik, for reviewing and giving feed-back on the paper; Steffen Vangsgaard for being part of the team working on the model used, and giving feedback on the paper; and the PharmaSUG Executive Committee for organizing a great conference!

TRADEMARK CITATIONS

CONTACT INFORMATION

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

Jesper Zeth
Novo Nordisk A/S
+45 30752626
jzt@novonordisk.com

Jan Skowronski
Novo Nordisk A/S
+45 30758131
jskw@novonordisk.com