THE DESIGN AND USE OF METADATA: PART FINE ART, PART BLACK ART

Frank DiIorio, CodeCrafters, Inc., Chapel Hill NC
Jeff Abolafia, Rho, Inc., Durham NC

Introduction

The complexity of even small pharmaceutical projects can be daunting. Consider the deliverables: patient profiles, listings, domain and analysis data sets, Define files, tables, and figures. Even in a single study, these routinely total hundreds of files. For NDA submissions, these are but a single piece of a larger "puzzle."

Consider as well the documentation and human resources pushing the study through its life cycle. Project managers need to monitor the completion status of the files. Statisticians and analysts have to identify data requirements and lay out "dummy" displays. Programmers have to write the programs to create the data and reports using specifications that are often, to be kind, "fluid."

Creation of high-quality output requires coordination of effort and clear and immediate communication of results. Rho, Inc. has migrated much of the requisite project management and data and display specifications to carefully designed and utilized metadata. By moving items that describe data sets and displays from documents and low-level programs into data sets, we have realized significant gains in productivity and quality of output.

This paper describes the current use of metadata at Rho. It:

- Discusses the motivation for using metadata
- Describes the metadata architecture
- Identifies tools that access the tables
- Presents examples, comparing metadata and non metadata-driven programs

The paper is largely conceptual and nearly code-free. While we emphasize application development in the pharmaceutical industry, we feel the underlying concepts regarding metadata design and implementation are valid across industries.

The Need for Metadata

Let's begin with an overview of a typical project's programming requirements (**Figure 1**, next page). This is, essentially, a map of the "before metadata" landscape. Among the notable characteristics are:

Many Files, Many Formats. Specifications for dataset creation, statistical analysis, data displays, variable derivation, and other items are typically held in a variety of formats. Word files, Excel spreadsheets, and other formats are, indeed, convenient for the statistician or project manager who is used to and comfortable with these tools. None, however, have the structure and security of a database – audit trails, controlled views, and other features that are second nature to database designers are lacking or poorly implemented here. The variety of tools also makes it somewhat difficult for an analyst or programmer to easily move from one document format to another. This is, in other words, a "whole" that is not characterized by "parts" that work well together.

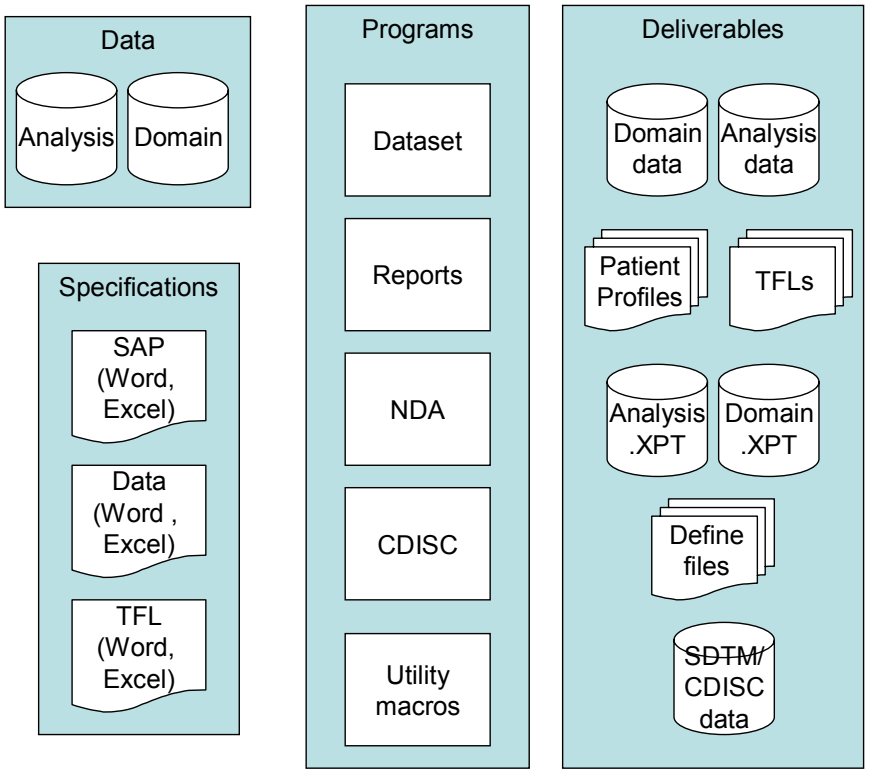
Specifications Are Not Data. Word and similar applications have the advantage of being familiar to most people, and they can produce attractive output. What they *cannot* be seen as, however, is a data source. That is, they cannot be programmatically manipulated. This makes extraction of table specifications, variable characteristics, and the like nearly impossible. This has a significant impact on work flow, as we will see in the next section.

Duplication of Effort. Resources such as data, format, and macro libraries must be allocated at the beginning of a program. One solution is coding the necessary macro variable definitions, LIBNAMEs, options, and FILENAMEs in every program. When a change is required (e.g., different data source,

modified option setting), it must be applied to every program. The potential for incomplete or inconsistent modification is significant.

The situation is somewhat improved if a standard `AUTOEXEC` file is used by all project programs. Changes can be made to a single location and they will automatically be picked up by any program using it. Still, the `AUTOEXEC` approach has shortcomings. It cannot, for example, ensure standardized library naming from

Figure 1: Organization, Pre-Metadata



study to study. The file can, if left to the design whims of a project programmer, include files that in turn include other files – an overarchitected approach that makes debugging and modification time-consuming.

Change. Any one involved for even the briefest time on an NDA submission or similar study has been exposed to an environment characterized by rapidly changing requirements. To describe but a few:

- The definition of analysis variables may require refinement or correction.
- Specification of use of this data in patient profiles, listings, tables, and figures may also change.
- Change to a display may even require a modification to the data being displayed (i.e., the change is pushed "upstream").
- Variables need to be renamed, relabeled, recoded, or dropped from datasets.

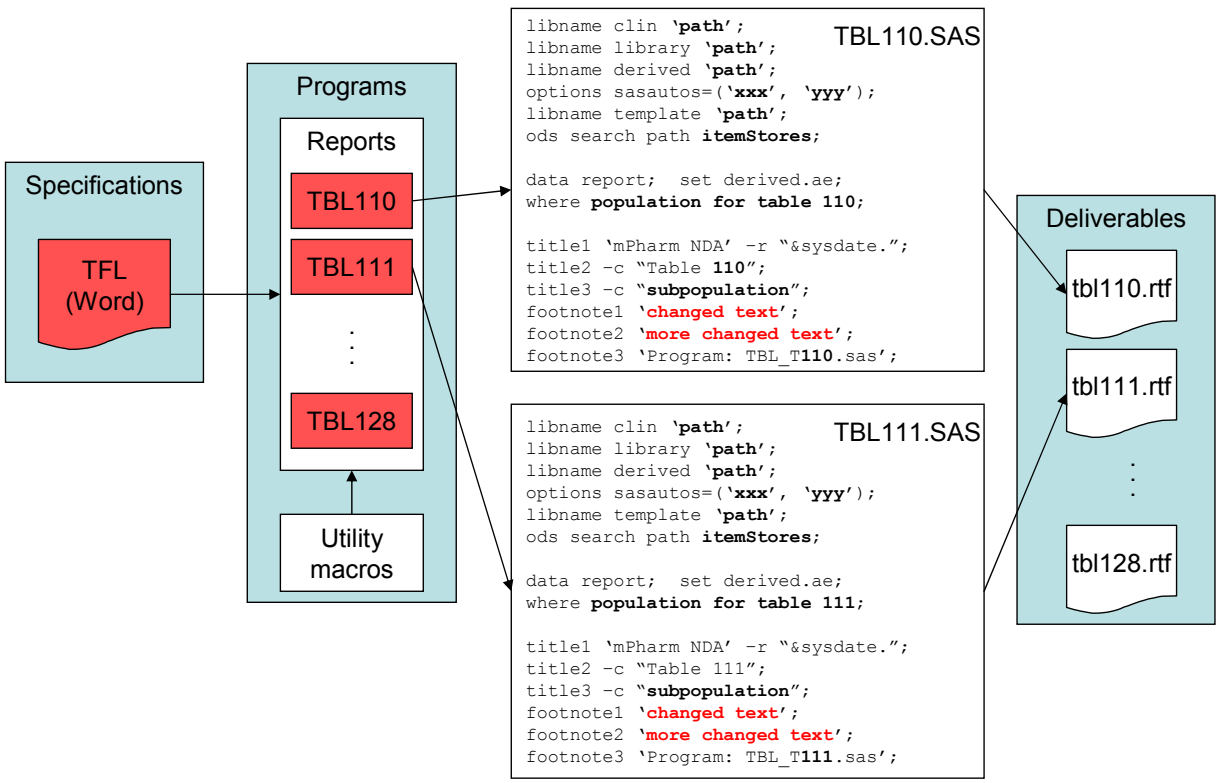
Change in and of itself is normal, and should be welcomed as an sign that people at various stages of the process – statisticians, programmers, medical writers, *et al.* – are examining project artifacts closely. What can be problematic is, first, ensuring that the change is communicated to the programmers and second, that the changes are correctly made in all of the affected programs.

Repetition. Another characteristic of the status quo environment is, arguably, the most exasperating from the standpoint of the statistical programmer. Display specifications are often characterized by a high degree of repetition and a small degree of variation. Twenty displays may vary only by the study population subset or categories they represent: one display may be process the entire population, another may categorize by age group, or subset by those who left the study prematurely. In all cases, the underlying program is the same but the data that feeds into it, along with title text, varies.

Consider This Scenario

Consider the scenario presented in **Figure 2**, below. Ten tables that are identical in layout except for the underlying table population. The program that produces one of the tables is, essentially, identical to the other nine table programs. The only difference is the selection of the population and the title text. Having 10 clones of the same program is inefficient for initial program development, and exposes the programmer to the same set of pitfalls described in the "Change" section, above. If, for example, the text or order of

Figure 2: Footnote Text Change, Pre-Metadata



footnotes is altered, the change must be applied to all 10 programs.

This scenario has many of the negative elements described earlier: the specifications are in a Word document and thus not readable from a program; the change has to be made by the analyst, then communicated to the programmer; and the programmer must make the change in multiple locations.

At best, will make for late nights at the office. At worst, the changes will be ineffectively communicated and/or not made in every program, thus raising the possibility of incorrect output being shipped to the client. It's worth noting that while it may be possible to automatically validate the body of the table, the header and footer areas are usually manually inspected.

What would be preferable is a way to automatically propagate the changes to all 10 programs. That is, we want to abstract the programs, making them, in effect, a form or template that receives specifications such as data subsetting, title text, and the like. The program creating the display becomes highly data-driven. We'll see an improved, metadata-driven version of this program shortly (**Figure 7**, page 11, if you want to cut to the chase).

What To Do?

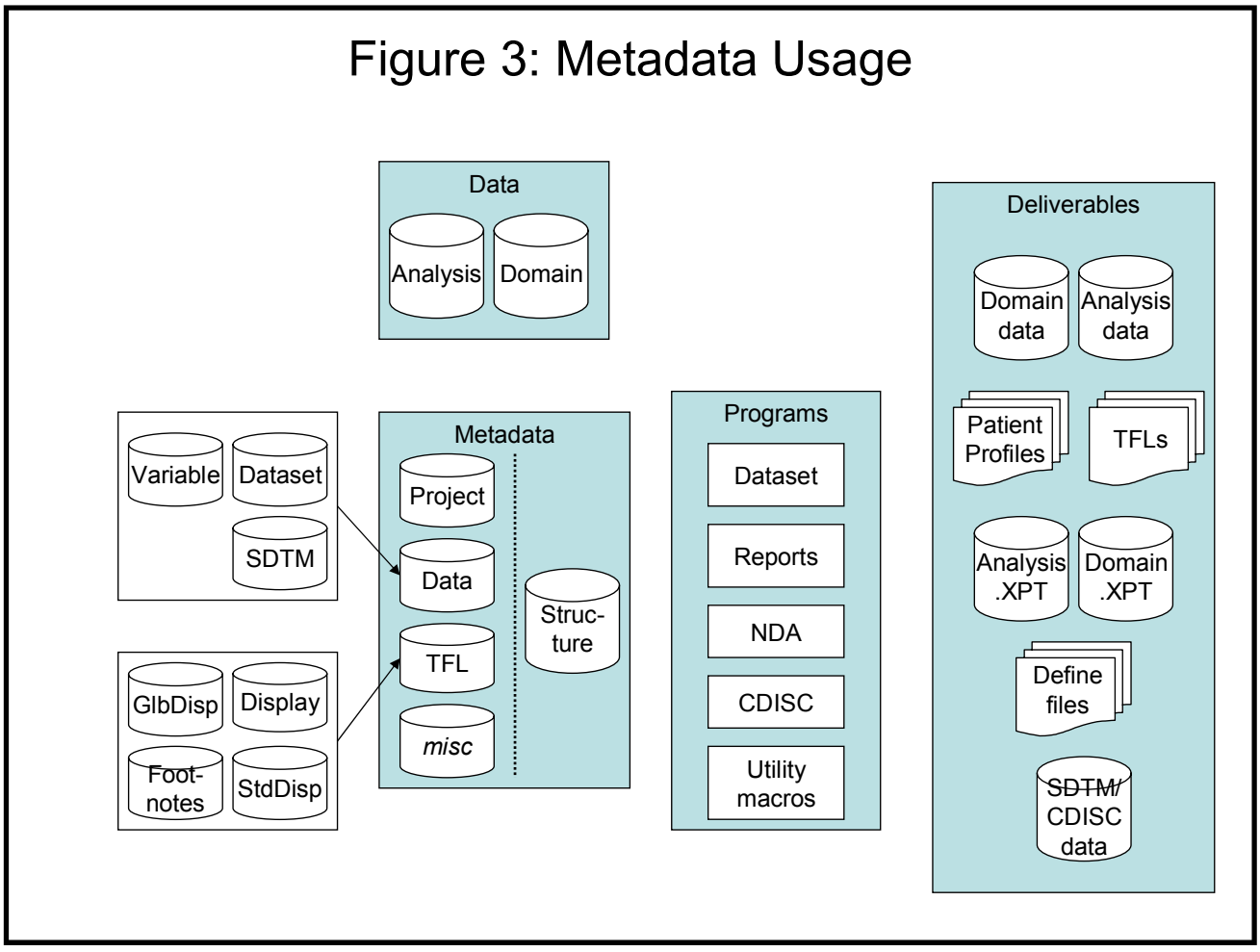
What would be desirable, given the preceding discussion, is non-redundant storage of project artifacts in file formats that are programmatically accessible. If data, TFL and other specifications are stored as data, they can be easily manipulated. Further, if they are stored in the same format across studies and projects, the uniformity can be exploited and a corporate-level library of access tools can be developed. Standardization and tools provides the application programmer the means to rapidly develop output, knowing that it is timely and accurate. What's needed, in short, is a well-designed collection of metadata tables.

Metadata Overview

Based on the problems identified above, we moved items such as directory structures, variable definitions, TFL components out of programs and Word documents into machine-readable metadata. In this section, we describe the metadata groupings and describe their contents. This is followed by some observations on the evolution of the data, and leads us into the wide-open realm of the metadata macro/server.

The strategic positioning of the metadata tables is shown in **Figure 3**, below (compare this with **Figure 1**, page 2, and see that traditional specifications have been replaced by metadata).

Figure 3: Metadata Usage



Groupings

The tables fall into a few general categories. Note that the classification is for discussion purposes only – there is no reason why, for example, an application could not draw on Structure, Project, and Display metadata tables. There are currently five groupings of metadata:

- **Structure** — Tables describe the organization and usage of directories for a project and its component studies.
- **Project —** Contains protocol name and description for the project and each of its component studies.
- **Data —** Describes individual datasets and variables.
- **Display —** For each TFL, contains fields for display names, titles, subpopulations, footnotes, links to display-creation program files.
- **Miscellaneous** — Stores items that are not project-specific, such as option settings, global macro variables, and administrative items.

Content

The content of the major metadata groups and tables follows below.

Structure

The table that is central to nearly all metadata usage is `STRUCTURE`. This table describes the location of each directory in a study. It captures the location, possibly wildcarding directories for multiple studies, usage (data library, macro autocall, format library), and other options.

This table serves as the basis for all LIBNAME statements, autocall and format search paths, and ODS template library allocations. The settings are described only once (in the table). If a change needs to be made – for example, new order of macro paths, or concatenate data libraries – the change is made here, and is automatically reflected in programs that use it. Just *how* the programs transform, say, a Microsoft Access table into SAS option statements and LIBNAMEs is discussed in "Metadata Tools," below.

Project

This level of metadata contains high-level, descriptive information about a project and its component studies. The table contains one observation per study. Fields include: the study title; protocol number; type of study; and location of key study documents (SAP, manuscripts), phase, number of treatments.

Data

We create two datasets for each study and data type (source / "raw" and analysis / derived). `DATASETS` holds dataset-level metadata, one record per dataset in the study. It contains fields that:

- Describe the contents and structure of the dataset
- Discuss how the dataset was created (essentially background material useful for programmers)
- List variables that uniquely identify an observation in the dataset
- Filter a dataset, identifying whether it should be included in particular types of output. A dataset may, for example, be part of an ISS but not an ISE.

There are also text fields to enter general comments about the dataset.

The `VARIABLES` table is the companion to the `DATASETS` metdata. For every dataset in the `DATASETS` metadata, the `VARIABLES` table contains one observation per variable, describing:

- Descriptors such as type, length, format, and label;
- For "raw" data, the source page number in the Case Report Form;
- For derived data, a narrative of how the variable is created;
- Controlled terms or codes;
- The desired variable order when writing datasets for FDA submission;

- A date-time stamp identifying when the most recent change to the observation was made;
- Filter variables that facilitate selection of variables for different types of output. The ability to filter at the dataset-variable level is a great asset, and is discussed in the "Examples" section, below.

Displays

Standard output for most studies includes a series of tabulations, graphic displays, and listings of individual observations, collectively referred to as TFLs. A study can require hundreds of TFLs. Likewise, it is the norm for these displays to be based on a much smaller number of unique layouts. Twenty tables could be laid out similarly, for example, varying only by the population used in each table – treated patients, age greater than 65, and so on.

The `DISPLAY` metadata group describes key features of each TFL. The `DISPLAY` table contains:

- Display number
- Title lines
- Footnote codes (see next paragraph)
- Datasets used by the table
- Display type (Table, Figure, Listing)
- Filtering information expressed as both descriptive text to use in titles and syntactically valid SAS program statement.
- A list of variables needed to create the display.

The `FOOTNOTES` table complements `DISPLAY`. It contains a field with a short footnote code and a longer text field containing the actual footnote text. The linkage of the `DISPLAY` and `FOOTNOTES` tables emphatically demonstrates the power of metadata-driven processes. Recall the example at the end of the previous section: a footnote's text had to be manually changed in multiple programs. Using metadata, the task is vastly easier and the output more reliable: a single change is made to text in the `FOOTNOTE` table, then the affected programs are rerun *without requiring modification*. Just *how* the footnote table's content is passed to the table program is discussed in the "Tools" section, below.

Other tables in this metadata grouping describe general features such as titles and footnotes common to all TFL's, font name, and point size. **Figure 4** (next page) illustrates the contribution of the different display metadata tables to a summary table.

Comments

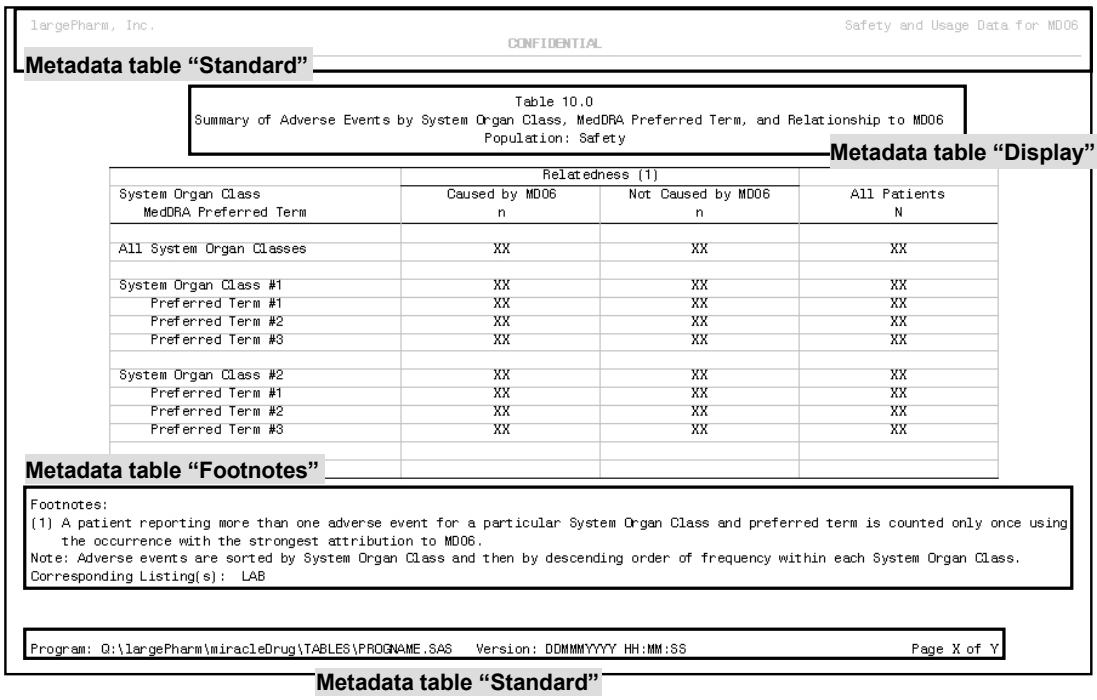
These Are "Living Documents" The metadata architecture outlined in this section is not static, but simply shows the system at a particular point in its evolution. As new needs arise (e.g., CDISC's ADaM and SDTM), new *tables* can be easily fit into the overall design. Also, new uses for existing metadata may require additional *fields*. It's important to realize that these and other scenarios are as desirable as they are inevitable. It is, in effect, an acknowledgement of the power of metadata-driven programming.

If You "Live By the Metadata," You Can Also "Die By the Metadata." Continuing with the footnote example – if the *wrong* footnote text is entered, or if the `DISPLAY` metadata incorrectly identifies footnote codes, all that has been accomplished is more elegant production of bad results. A set of quality control tools for the metadata was not part of the original design, but the need for them quickly became apparent.

The Choice of File Format Is Important. Early versions of the tables were held in SAS datasets and Excel spreadsheets. These had a number of drawbacks: there were performance issues with the SAS/Share server, the SAS and Excel "out of the box" user interface for editing data was awkward; multiple users entering data into Excel sheets was not possible; and the SAS Version 8 tools for reading Excel sheets (DDE, PROC Import, Access descriptors) were quirkier than we would have liked.

This suggested the need for a more robust database solution with a friendly and easily modified user interface. Microsoft Access 2003 is now used for entry and storage of all metadata. Now, rather than type a Word document, users enter dataset, TFL, and other specifications using Access forms. Our bridge to

Figure 4: DISPLAY Metadata Usage



this data is the SAS Version 9 ACCESS engine. It has proven to be extremely stable, and has eliminated performance, entry, and data loss issues.

It's *Mostly* a Manual Process. Some metadata tables can be at least partially populated from existing internal and external sources. Consider some of these:

- **Standards organizations** — CDISC, for example, publishes Excel files that can be viewed as the "Gold Standard" for SDTM domain files.
- **Existing Systems —** Internal data management systems are usually table-driven
- **Client Systems —** Clients usually supply documentation that describes data.
- **SAS Dictionary Tables —** SAS metadata contains significant content about "as built" data. Contrast this with the home-grown metadata, which describes data "as we'd like it to be built."

These and other sources can be used to seed the initial versions of metadata tables. The coding required to transform the source into "Version 1" metadata is well worth the effort: metadata development time is reduced, and analyst resources are freed up.

The Underlying Idea Is Familiar. Metadata is hardly new. It predates SAS software, and has been part of SAS for years, in the form of Dictionary Tables. Once you understand the Tables' scope and content, it's hard not to get excited about the range of possibilities for their use. The metadata that we have described here is conceptually similar to the Dictionary Tables in its breadth of application. Together, these two forms of descriptive data enable creation of robust, extensible applications that would be difficult, if not impossible, to write in their absence. Just *how* to make best use of the metadata is discussed in the next section.

Multiple Tables Are the Norm. Even if an application needs information from what appears to be a *single* data source, accessing the information often requires references to *multiple* tables. Suppose, for example, an application needs a list of all datasets and variables in a study that are to be exported to the client's database. The list would need to be filtered not only using the `VARIABLES` table, but also the

DATASETS table. That is, the list would consist of eligible variables from eligible datasets. These and other routine tasks are not particularly difficult to code, but they do suggest the need for an application layer between the metadata and programs that use it.

Providing Easy Access. One doesn't need to have an active imagination to wonder just *how* the metadata can be used. It is rich in content, fully describes processes in need of automation, and is complex. Effective use of the metadata requires tools that simplify table access. These are discussed next.

Metadata Tools

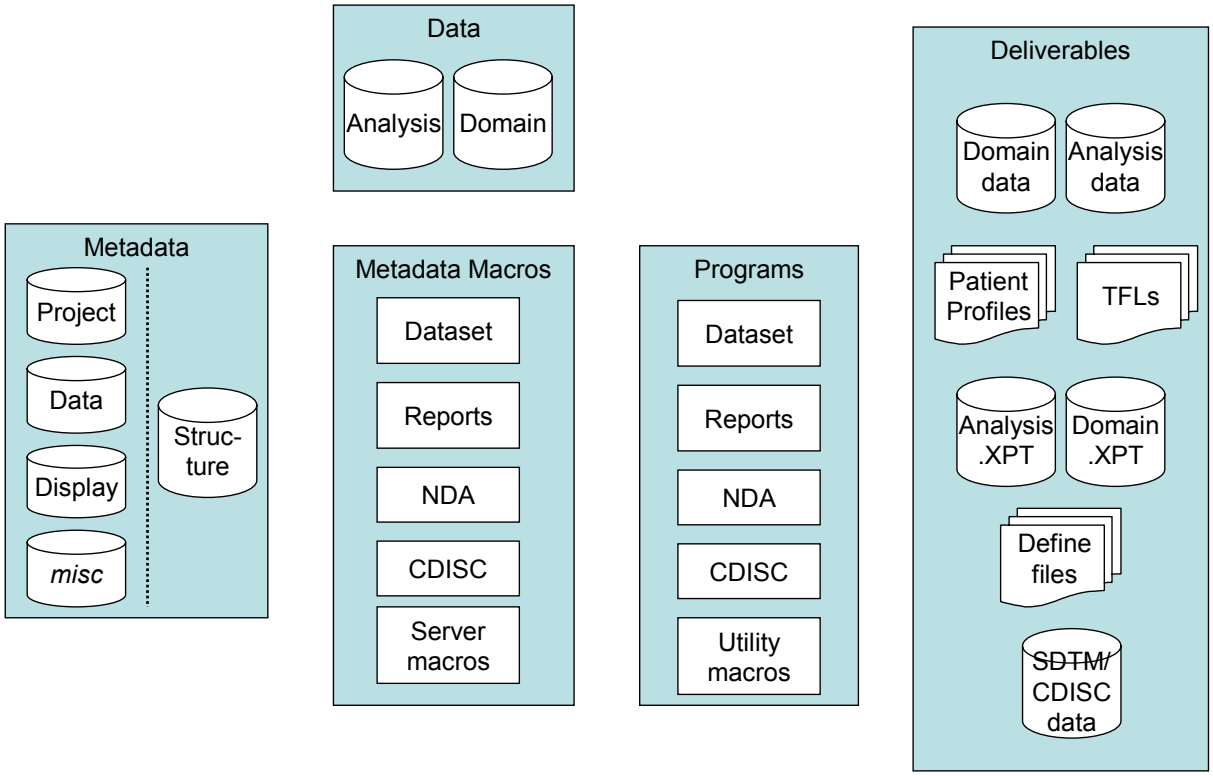
The metadata is the engine driving applications throughout the project life cycle. Consider for a moment how useful an engine is without, say, a car wrapped around it. It might emit a gratifying, throaty roar, but would ultimately be useless because it couldn't take you anywhere.

Our experience with metadata design quickly pointed out that you would not gain efficiency – indeed, you would become *less* productive – without applications making access more or less transparent. This section describes some of the tools Rho has developed toward this end. As with the design of the tables themselves, the tools were developed on a somewhat ad hoc, "heat of the moment" basis.

Consider the Need

To underscore the need for access tools, consider the programming requirements needed for accessing metadata that describes a tabular display. The display type and number must be located in the DISPLAY metadata. We must also gather and correctly sequence the footnotes from the FOOTNOTES table. Finally, we retrieve standard headers and footers from the GLOBAL table. Once all the pieces are identified, they

Figure 5: Add Metadata Macros



need to be presented to the table-writing program in an agreed-upon format (macro variables, datasets, etc.). To be thorough, we should add checks to ensure data quality: were all the footnote codes specified in `DISPLAY` actually present in `FOOTNOTES`? Do we have complete title text? And so on.

We could, of course, write code in each table program to perform these actions. More likely, we would want a tool that would do the work for us, reading the required tables, and creating a set of macro variables that would make the metadata readily accessible. The process to create macro variables for Table 10.1 should be as simple as:

```
%getSpecs(type=table, id=10.1)
```

The macro would perform all the activities described above, and would produce diagnostics that would quickly give the table programmer an indication of success or failure. The SAS Log would contain messages showing what was created by `%getSpecs`, along with what was cautionary or problematic.

Clearly, a library of tools for TFL generation, LIBNAME assignments, option setting, and the like comprise the application "body" which surrounds the metadata "engine" described at the beginning of this section. Clearly, too, we want to allow programmers who want to read the metadata directly to do so. The Big Picture that has emerged over time (**Figure 5**, page 8) is one that has metadata at its core and that is amenable to different levels of end-user programming effort.

Tool Descriptions

While there is nothing to prevent a programmer from directly utilizing the metadata, it's far more likely that the metadata will be effectively used if a "server" layer is interposed between the data and the application. **Table 1,** below, shows some of the tools currently being used by developers at Rho. Table entries correspond to SAS macros that access metadata. In some cases, they are small and tightly-focused,

Table 1: Metadata Access Tools, By Deliverable and Metadata Group

	Deliverable		
Metadata Type	**Data**	**TFL**	**NDA**
Structure	Standardized program startup	Standardized program startup	Standardized program startup
Project		Create compound description	Create compound description
Data	Create ATTRIB and RENAME statements; create dataset and variable lists	Verify datasets, variables needed for a display are available.	Create ATTRIB statements, variable lists
	Document dataset characteristics		Export data per FDA requirements.
	Generate data create spec document		Create "define" file per FDA requirements
	Assign variable labels		Verify "define" file links exist in referenced PDFs.
	Check data-metadata consistency (type, length, etc.)		Verify data, "define" contents match what is expected, given metadata
Display	Combine metadata sources for a given display	Create Title, Footnote statements	Generate Codes lists from external sources
		Create DATA step code fragments for filtering data	Display derived variable ancestry – show circular, invalid references
		HTML displaying Log, output links, other info for each TFL	
		Verify accuracy, completeness of metadata	

and are used by other applications (e.g., the ATTRIB statement generator). Other macros are longer, reference other metadata and utility macros, and are standalone applications (the "define" file generator).

Examples

Footnotes Revisited

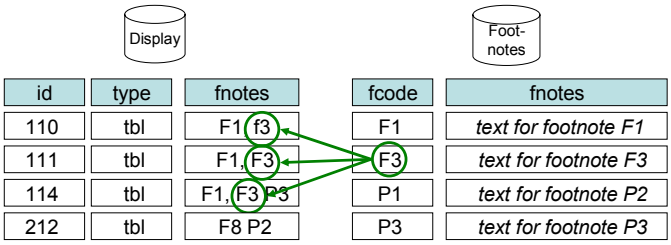
Recall the footnote scenario discussed earlier and illustrated in **Figure 2** (page 3). The Word document containing the footnote text changed, then each of the 10 programs using the footnote had to be updated to reflect the new footnote text. **Figures 6** and **7**, below and next page, demonstrate a metadata-driven approach to the problem.

Figure 6 presents a greatly-simplified version of the tables. The `DISPLAY` table field `FNOTES` holds a set of footnote codes. The first row uses footnote codes `F1` and `F3`. The metadata macro that processes Table 110 will create footnote statements containing the formatted text from the `FOOTNOTES` table.

In this scenario, using metadata and metadata macros, the change is automatically picked up by the display programs and *no modification to the programs is required*. Another important advantage of the metadata macro is that it brings problematic conditions to the programmer's attention (here, `ID` 212's reference to non-existent footnote code `P2)`.

The revised, metadata-aware programs are shown in **Figure 7** (next page). The hard-coded footnote text

Figure 6: DISPLAY-FOOTNOTES Interaction



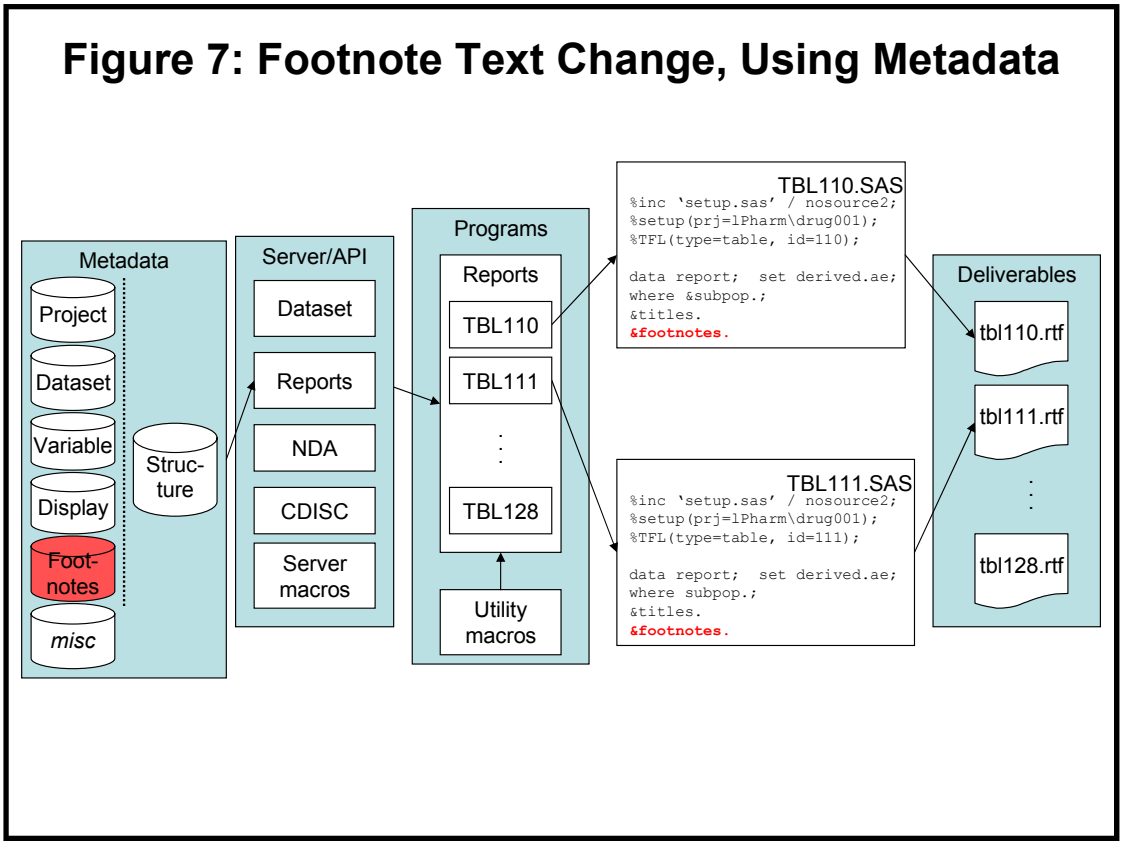
we saw in **Figure 2** has disappeared. In its place is a single reference to macro variable `FOOTNOTES`, created by macro `%TFL`.

Remove User-Written Formats

Let's use metadata for another "program makeover.". Some US regulatory agencies such as the FDA accept SAS datasets as deliverables, but prohibit references to user-written formats.

Figure 8 (next page) shows an non-metadata approach. A specification document identifies the datasets and variables to create, along with characteristics such as data type, length, label, and format. The programmer could manually create a list of variables with user formats, then remove the format reference using PROC DATASETS. This coding strategy is included at the end of each dataset-creation program.

Figure 9 (page 12) shows a compact, single-program solution utilizing metadata that comes from SAS (Dictionary tables) and is home-grown. We read Dictionary table `FORMATS` to create a list of native SAS



formats, then use metadata tables `DATASET` and `VARIABLES` to identify variables with formats *not* in the native format list. Once these variables are known for each dataset, it's a simple matter to create the requisite PROC DATASETS statements.

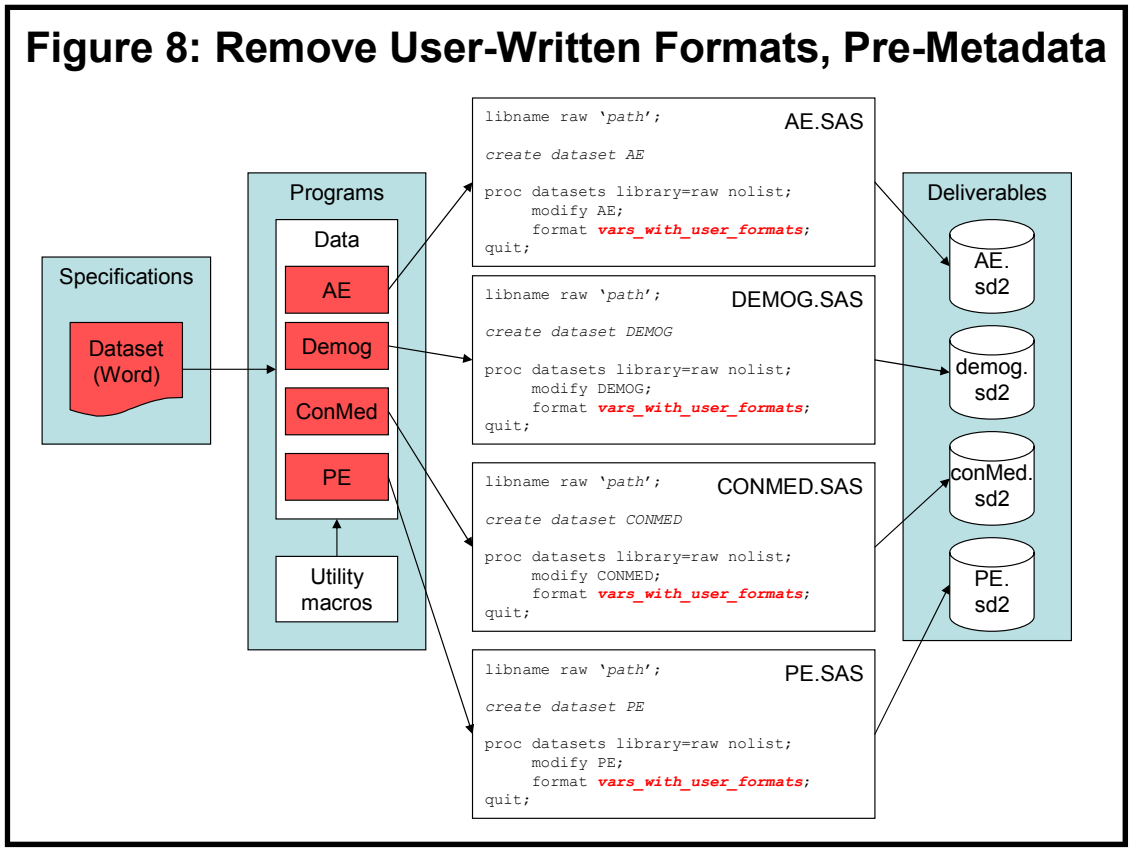
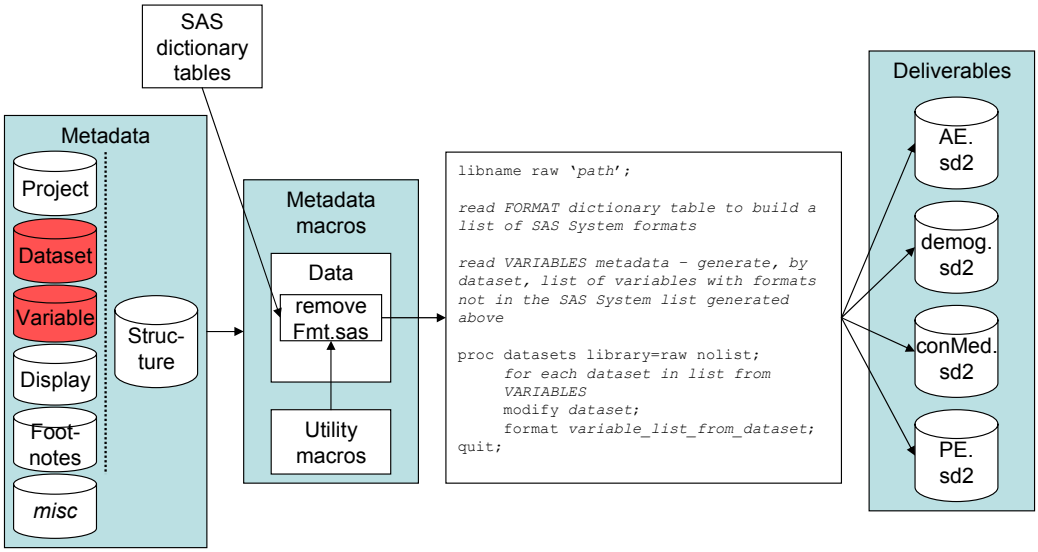


Figure 9: Remove User-Written Formats, Using Metadata



Readers familiar with Dictionary Tables could rightly say the entire task could be done without any home-grown metadata (the `COLUMNS` Dictionary table contains format information). The main reason we did not take this approach was that the `DATASET` and `VARIABLES` tables provide an added level of quality control – we can identify variables with user formats *and* can verify, via one or more filter variables, whether the variable should be written to the dataset. The filtering information is home-grown value-added that is not in the SAS metadata. The program in Figure 9 demonstrates how the two forms of metadata are complementary, rather than independent.

Reorder Variables in a Dataset

Rather than using Viewtable or a similar tool to arrange variables while browsing, clients often request that variables be stored in a specific order. In the absence of metadata, the programmer would need to review a specification document, then, for each dataset, manually enter the variables in a RETAIN or SQL SELECT statement. This harkens back to the problems identified at the beginning of this paper: the manual process is tedious, error-prone, not easily validated, and time-consuming, particularly if the specification changes more than once. A representation of the pre-metadata solution is shown in **Figure 10** (next page).

A much cleaner and more compact approach is shown nearly in its entirety in **Figure 11** (next page) Recall our earlier comment about metadata continually changing. We added a sequence field (`SEQ`) to the variable-level metadata to meet the client's requirements. Had it not been a request that other clients were likely to request, we would have opted not to alter the metadata.

With the sequencing information available in the metadata, program `REORDER.SAS` becomes straightforward. It is notable for its use of a utility macro (`%DISTINCT`) to produce a list used by the macro. `%DISTINCT` reads dataset `MDATA` and produces macro variables `LEVELS` and `NLEVELS`, containing the names and count of unique values of variable `DSET`. The macro specifics are less important than the metadata macro's effective use of general-purpose utilities. The macro becomes smaller (a single macro call instead of a dozen or so lines of code) and more reliable (by using validated tools such as `DISTINCT`).

Figure 10: Reorder Variables, pre-Metadata

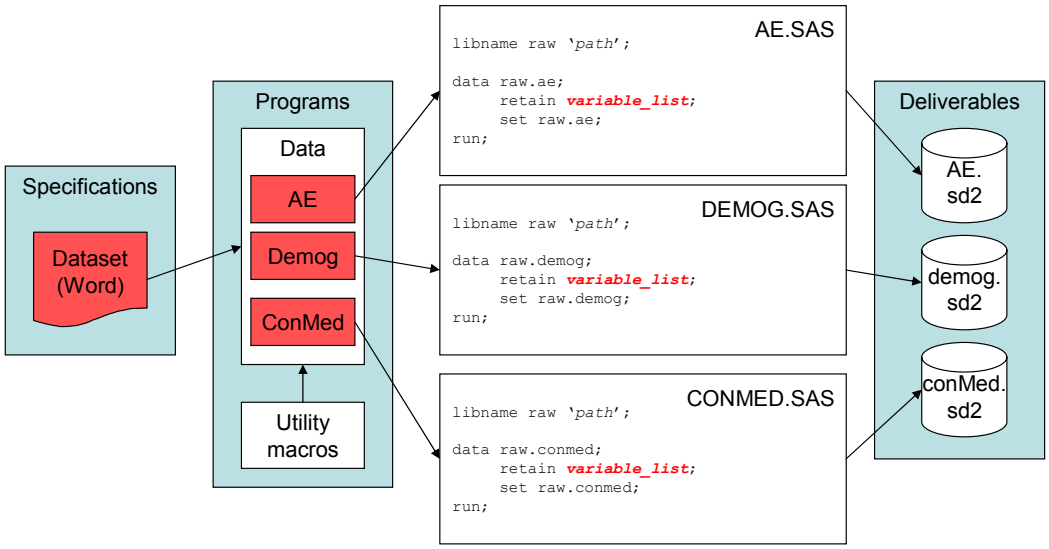
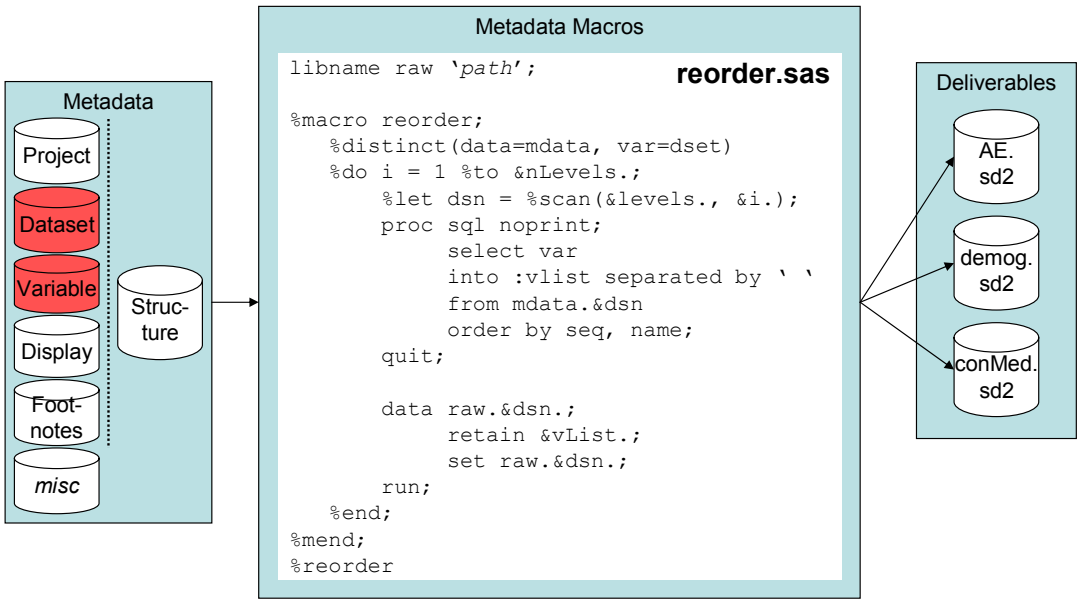


Figure 11: Reorder Variables, Using Metadata



Comments on Tools and Usage

Two general points about metadata usage and tool building have been implied throughout the preceding discussion. We explicitly mention them here to highlight their importance.

Build with Existing Tools

Utility Library. Metadata macros, like any other tools, use a library of general-purpose macros. The importance of reliable, comprehensive macros cannot be overemphasized. Likewise, the range of tools is noteworthy.

Some of the macros are purely diagnostic, used only by the tool builder. These include: writing a brief listing of one or more datasets' contents to the SAS Log; writing macro variable values to the SAS Log in a clearly understandable order (contrast with `%put _global_;`) displaying settings for a user-specified list of system options; and identifying datasets and macro variables that were created during execution of a macro (an aid in preventing unwanted artifacts being produced by a macro).

Other utility macros are intended to be used as part of a larger application: uppercase a list of macro variables; verify one or more variables exist in a dataset; return the number of observations in a dataset; count the number of tokens in a macro variable; quote the tokens in a macro variable; and so on.

There is, of course, a modest cost to using the utility macros: the flow of program execution gets a bit more complex, since the macro is called from a metadata or similar high-level tool; the globally available macros perform parameter checks and other activities that may be redundant or unnecessary, given the needs of the calling program, thus unnecessarily consuming CPU or other resources.

These drawbacks are almost always negligible and are greatly outweighed by the huge advantage they offer when constructing applications. Without an observation-counting utility, one would have to code the following to create macro variable `NOBS`:

```
proc sql noprint;
     select nobs into :nobs
     from dictionary.tables
     where libname="MASTER" and memname = "DEMOG"
           and memtype = "DATA";
quit;
```

Admittedly, this is not rocket science, and can be easily coded. It is far easier to refer to an autocall macro, especially if dataset counts have to be taken multiple times in a program:

```
%obsCount(data=master.demog, count=nobs)
```

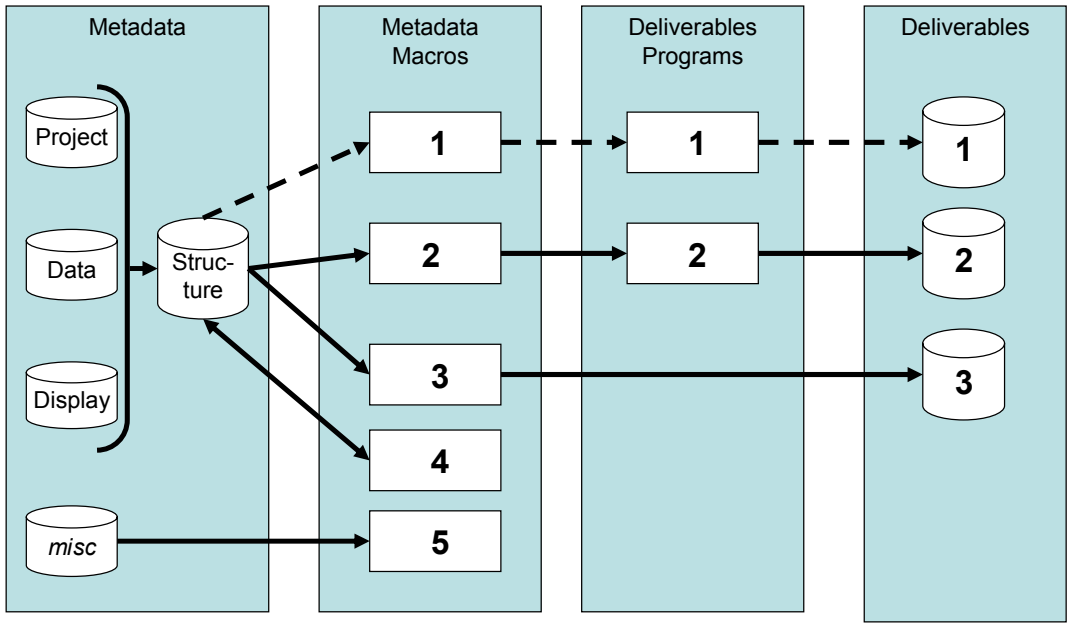
`obsCount` not only returns the item of interest – macro variable `NOBS` – but also performs checks that would not normally be peformed while writing in-line code. It also writes messages to the SAS Log that apprise the user of the macro's success or failure. It is, in short, a means to easily develop robust, concise programs.

SAS Dictionary Tables. Home-grown metadata is conceptually similar to putting the SAS Dictionary Tables to work. It bears mentioning here that some of the most powerful, elegant, and extensible macros in the entire Rho system are those that use *both* the SAS metadata (the Dictionary Tables) *and* the site-specific, home-grown metadata tables.

Usage Levels Vary

It's unproductive and even counterproductive to force all users to use the full-blown system as described above. The system as currently architected allows different levels of access and usage. A few of the possibilities are illustrated in **Figure 12** (above). The Figure contains a simplified and abstracted view of the system. The numbered items represent some possible scenarios of metadata usage.

Figure 12: Metadata/Macro Usage Varies By Task



1. Programs creating deliverables only require `STRUCTURE` metadata, presumably for allocating libraries and autocall and format search paths. The metadata macro layer is needed only for program initialization. The `STRUCTURE` metadata and `%SETUP` metadata macro effectively replace autoexec or hard-coded program startup tasks.
2. The locations of `PROJECT`, `DATA`, and `DISPLAY` metadata are identified by `STRUCTURE` metadata. The programs creating deliverables make use of the program initialization macros and other macros that manipulate and simplify access to the metadata. This is usually the most effective use of the system because it uses a set of tools (the macros) to provide programmers and end-users with access to the metadata. It isn't necessary to see the metadata. One only has to understand the inputs to and outputs from the "black boxes."
3. A metadata macro processes one or more metadata tables and creates output based solely on the display or reformatting of the metadata. The client deliverable could be, for example, dataset documentation based on merged and formatted `DATASETS` and `VARIABLES` tables.
4. No client deliverables are produced in this scenario. Macro inputs and outputs are contained within the metadata system. A metadata macro could, for example, read CRF or "raw" data and make a first pass at Domain-level `DATASETS` and `VARIABLES` metadata. Another application in this scenario could be metadata validation, where the macro identifies inconsistencies and potential problems that were not identified during the manual creation of the metadata.
5. This path through the system uses a global, non project-related table and so does not require access to the Structure data. Since it does not use the project-specific data around which the majority of our activity revolves, it is quite literally the "path less traveled." Macros here display system-wide settings and perform other, administrative tasks.

Conclusion

By now, the reader should begin to understand the paper's title. The design of the metadata and tools is part "fine art." One has to possess technical expertise to develop the tables and write the programs that present the metadata to the programmers. The process is also part "black art." Given that we were often dealing with entirely new programming and workflow issues, we frequently relied on the intuition that comes from being "seasoned" professionals.

Key Reasons to Develop Metadata

Uniformity. The fully realized use of metadata, particularly the `STRUCTURE` table, fosters uniformity *within* project-related studies and *across* projects. When corporate-wide directory structures, LIBNAMEs, and the like are similar, tools developed for one project can be applied to other projects with little or no modification.

Cost Reduction. Many of the examples in this paper highlighted the benefits of a single metadata entry point. If, for example, an analyst modifies a footnote value in metadata, the cost of the change is borne only once. Compare this to a metadata-free environment where the analyst enters the value in a Word document, followed by programmers who have to replicate the new text in multiple programs.

Workflow Change. A driving force behind metadata is the abstraction of programs, moving hard-coded items out of programs and into data stores accessible by SAS. If the metadata interface is friendly enough, and if tools are in place to trap inconsistencies in the metadata, print it, and the like then some of the entry can be offloaded to other, non-technical staff. This frees up analyst time and reduces time to delivery without sacrificing quality.

Quality Improvements. Specifications stored as data can be checked programmatically. Likewise, output generated using metadata can be validated with metadata tools. Effective use of metadata enables faster production of validated deliverables and is an ideal match for the times when the content and nature of the deliverable changes, must be reprogrammed, and then revalidated.

Metadata Is Inherently Multi-Use. A metadata source table can easily be used for multiple types of tasks throughout the project life cycle. The variable-level table can, for example, be used for:

- Creating attribute statements when a dataset is being built (e.g., use format, length, type fields in a metadata macro)
- Building keep and drop lists when exporting a dataset (use filter fields)
- Performing quality checks on data (compare as-built dataset characteristics to those expected by the metadata)
- Documenting data (create publication-quality documents listing variable attributes and derivation)

Happier Programmers. Repetitive hard-coding of title and footnote text is an unchallenging, error-prone task, especially when a deadline is looming. The metadata and the tools to access it allow programmers to focus on program functionality rather than what are, in essence, clerical issues.

Metadata Is the Way of the Future. Even if you don't buy into the idea of metadata, other people do. For example, the FDA currently accepts Define files, product labels, and safety reports as XML files. This format will eventually be mandatory. This pharmaceutical example is hardly unique. Given the inevitability of creating these data-rich, programmatically accessible XML files, it will simply not make sense to use Word-based specifications and *then* create XML.

Key Lessons Learned

Design Metadata *and* Tools. Metadata is, indeed, at the heart of the process changes we have implemented. Without tools that make its use transparent or, at minimum, "pretty simple," the tables' impact would have been blunted. Tools should be in place for end-user access and quality control.

Pay Attention to the User Interface. Even if the *idea* of metadata is appealing and its potential impact

is huge, it will gain only grudging acceptance if it is difficult to enter. Any metadata that must be manually entered should have a friendly, intuitive interface that guides entry, and preempts or, at the very least, catches errors and inconsistencies.

Document Everything. Documentation should describe the contents of the metadata tables and the macros that provide access to the tables. This improves ease of use, which in turn creates an atmosphere that gives more people "the metadata religion."

Be Flexible – Expect Change. Remember that just as organizations evolve, so does the content of the metadata. The change can be driven internally, by your regulatory environment, or by client needs. Whatever the source of change, welcome it as an opportunity.

Extensions

Create a Metadata Hierarchy. Some items (display font, point size, margins, system option settings, and the like) are identical from project to project. A metadata hierarchy for these items could be defined, storing them at corporate, project, and study levels. The lower levels in the hierarchy could accept or reset values set at a higher level.

Change Notification. While the approach described here percolates a changed field from metadata to applications (e.g., the footnote example), it begs the question "just what *is* the mechanism to trigger the actual re-creation of the affected project pieces?" Using our footnote example, how would we know to rerun the 10 table programs once the footnote text was changed? The metadata could be examined at set intervals or on demand, detecting changes and notifying analysts and programmers of processes that need to be rerun.

Metadata Catalog. Much of the dataset and display-level metadata is duplicated within project studies and between projects. The user interface for manually-entered metadata could be enhanced, allowing population of variable or display data to be done via a pick list, drag and drop, or similar access to a corporate-wide repository. Seen from this perspective, a collection of standardized metadata from multiple projects becomes a valuable corporate-wide knowledge base.

Contact

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

Frank@CodeCraftersInc.com JAbolafi@RhoWorld.com

Acknowledgements

Robert Anderson, Frank Porter Graham Child Development Center, University of North Carolina, Chapel Hill, made suggestions that led to the addition of Appendix A. Both the paper and the topic are more accessible for his gentle prodding to supplement the abstract text with concrete, real-world examples.

Russ Helms and **Ron Helms** of Rho, Inc. provided the impetus for metadata use at Rho. They have also created an environment at Rho that encourages experimentation, rewards success, and is sympathetic when ideas don't pan out.

April Sansom provided her typically thorough copyediting services, and is probably wondering why, with all those years of Catholic school education, the authors punctuate in a way that can only be described as whimsical.

Jack Shostak, Duke Clinical Research Institute, offered several insightful comments on an early version of this paper.

References

Dictionary table and macro design papers can be found at www.CodeCraftersInc.com.

Appendix A: Case Study

The processes described in this paper are, for the most part, deliberately left abstract. The underlying coding is interesting, challenging, and worthy of discussion, but the paper's focus is selling the idea of metadata, not the DATA steps and macros that are required for implementation.

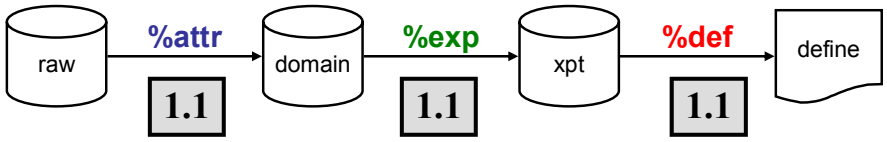
That said, an early reviewer of the paper said that he thought people would appreciate its content but might be at a loss putting the ideas into practice. Hence Appendix A, where we hedge our bets.

The Appendix study illustrates one of metadata's greatest strengths, its ability to be used by multiple applications. We focus on variable-level metadata, and show how it is used throughout the project life cycle. Bear in mind that

Figure A1: Multiple Uses of VARIABLES Metadata

dataset	var	len	type	fmt	lbl	CRFpg	seq	use
AE	SITE	3	C	$3.	Site	1	1	*
AE	ID	5	C	$5.	Subj ID	1	2	*
AE	AESEV	3	N	1.	Severity	6	3	*
AE	AESTDY	3	N	2.	Onset/day	6	6	*
AE	AESTMO	3	N	2.	Onset/month	6	5	*
AE	AESTYR	3	N	4.	Onset/year	6	4	*
AE	AEDUR	4	N	time5.	Duration	6	3	*
AE	AESER	1	C	aeser.	Serious AE?	4	999	S
1.2	attr exp def	attr exp def	attr	attr def	attr	attr def	def	exp def	attr exp def

Variable usage, by macro



the examples, while "real world," are for the sake of illustration only, and thus are both a bit contrived at times and incomplete (there is none of the parameter checking, testing for presence of data sets, etc. that a "real" macro would perform).

Figure A.1 presents a few rows and columns of the metadata. It also shows three macros **[1.1]** `attr`, `exp`, and `def`, that use the metadata. `attr` is used to convert raw, Case Report Form data into "domain" data sets that can be used by both the client and for the creation of data sets that have new, derived variables that were not captured in the raw data. The translation of the domain data to SAS transport file format is handled by `%exp`. Finally, the documentation required by the FDA (aka the "Define" file) is created by `%def`. Notice that not all variables are used by each macro **[1.2]**. This is one of the great strengths of *any* metadata dataset: it is a centralized, programmatically accessible location for storage of items relevant to *many* processes.

1.1

1.2

We see the entire process in action in **Figure A.2** (next page). The figure identifies global utilities that can be used by any application **[2.1]** (`%count`, `%distinct`, *et al.*) as well as macros that are designed specifically to support metadata usage **[2.2]** (`%removeFmt`, `%attr`, `%keep`, `%setup`). Notice, too, that the global utilities are used by the metadata utilities and the domain building program, `build_AE.sas`. Robust, well-designed macros can, and should, be used by *any* other program. This relieves the programmer of the need to continually write

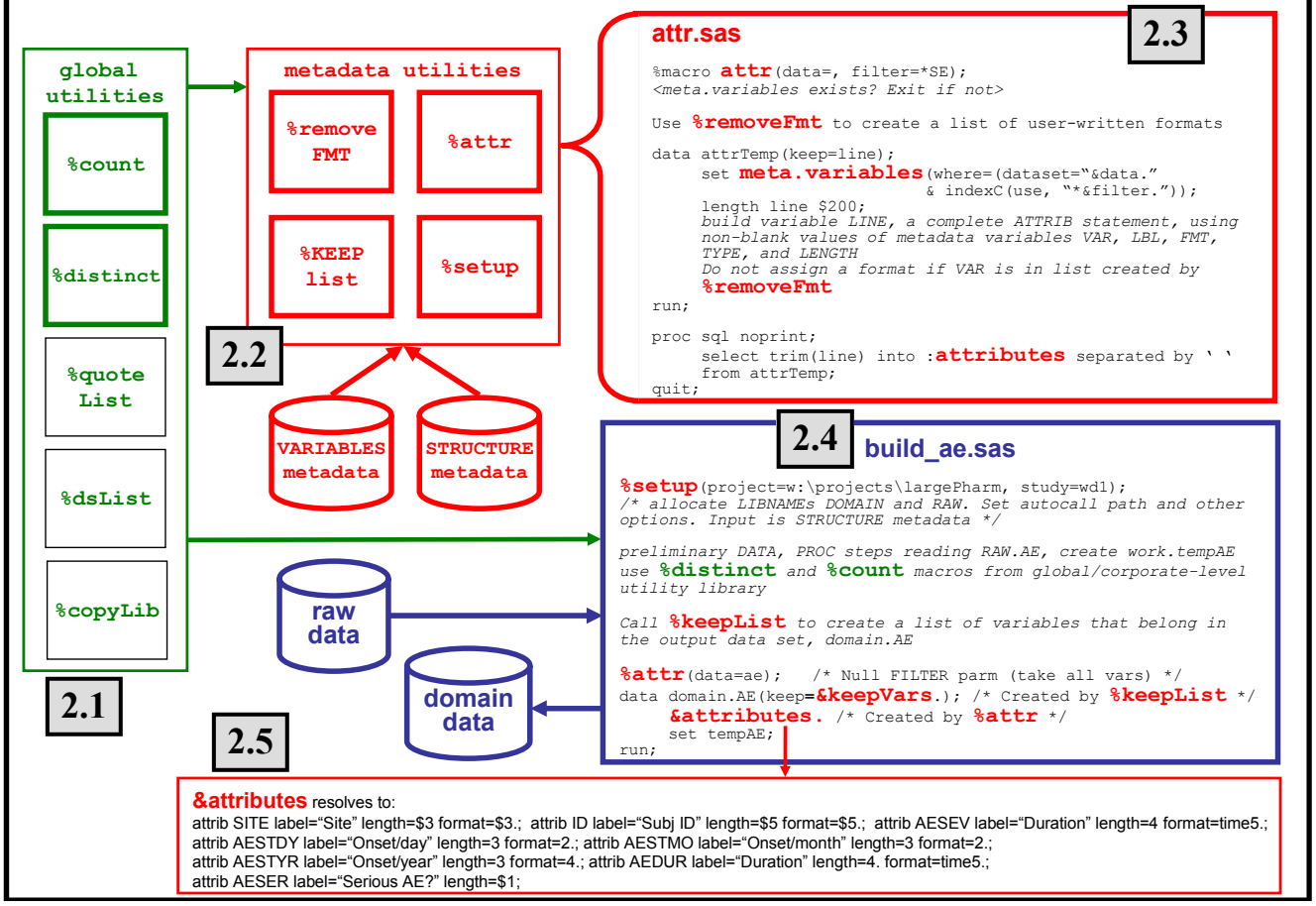
2.1

2.2

code that quotes a list, counts occurrences of a variable, and the like.

2.3 A stripped-down version of `%attr` is shown **[2.3]**. It reads the `VARIABLES` metadata and builds variable `line`,

Figure A2: VARIABLES Usage: Domain Creation



2.4 which is a complete `ATTRIB` statement for a variable. The `LINE` values are then concatenated and stored in global macro variable `ATTRIBUTES`. Program `BUILD_AE.SAS` provides the context for the macro. It uses `%setup` to allocate libraries, then creates dataset `work.tempAE`. Macro variable `ATTRIBUTES` **[2.5]** does the necessary format, length, and data typing,. It's important to realize that if the metadata changes, there are *no* changes required in the program. The changes will be picked up the next time `BUILD_AE.SAS` is run. The program in many ways becomes simply the template for displaying the metadata.

2.5

The next chapter in the data saga is creating SAS transport files from the domain data (this is the required format for submitting data to the FDA). **Figure A.3** (next page) illustrates the process. It is conceptually similar to `%attr`, with a slight twist. The domain creation program `BUILD_AE.SAS` in Figure A.2 called macros that, in turn, used the variable-level metadata. The export program uses the metadata *directly* **[3.1]**, reading it for each domain data set and creating a macro variable **[3.2]** (`VARLIST`) that holds the variables to use in the order specified by the metadata's creator. The metadata provides the flexibility demanded by the application: `SEQ` allows the variables to be arranged in any order. Like the assignment of data attributes in Figure A.2, the modification of the data is accomplished without touching the program that actually builds the data.

3.1

3.2

4.1 The last step in the process is described in **Figure A.4** (next page). The FDA requires that data set documentation in one of several specific formats, none of which can be created from a PROC CONTENTS (or similar) output data sets. `%def` is similar to `%exp`'s look and feel. It directly uses the `VARIABLES` metadata **[4.1]**, filtering and arranging fields as needed. This macro, however, exploits more metadata tables, retrieving the a dataset's structure / granularity from DATASETS **[4.2]** and page layout and other presentation guidelines from `GLBDISP` **[4.3]**. Once again, we see the power of centralized, programmatically-accessible metadata and, hopefully, become convinced that we never, ever want to return to the days of hard-coding these values in programs.

4.2

4.3

Figure A3: VARIABLES Usage: Data Export

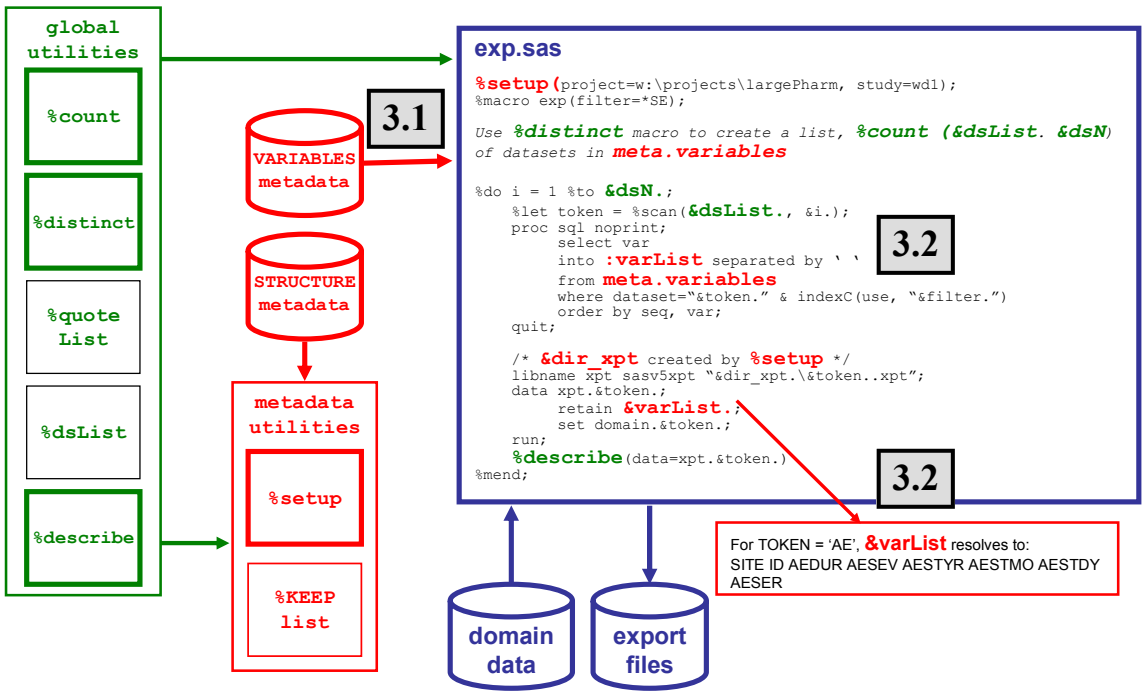


Figure A4: VARIABLES Usage: Define File