Large Scale Standard Macros - A Methodical Approach to Development and Implementation

Jeremy Gratt, Modular Informatics, Thornwood, NY
John Adams, Boehringer Ingelheim Pharmaceutical, Ridgefield, CT

Abstract

Much of the analysis for clinical trials in the pharmaceutical industry is done with very large standard macros. The use of these macros facilitates the goal of standardizing and accelerating the production of clinical reports within a company. Each standard macro is typically specialized to analyze and report on a specific type of data such as efficacy, safety, drug interactions, or demographics.

The challenges encountered during the development of large scale SAS® macros and the development process adopted at Boehringer-Ingeheim will be described. This methodical approach improves development efficiency and maintainability. Essential elements of the approach that will be covered include the software development life cycle, macro storage and versioning, modular program structure, program documentation, and common macro tools.

Introduction

The development and implementation process for large scale SAS® macros in company-wide applications can quickly become unmanageable. Specifications, code complexity, code maintainability, life cycle management and testability are just some of the problems faced. Clearly, an organized approach is needed. This paper will describe the methodology and tools that are used at Boehringer-Ingeheim to address these issues. These methods and tools were derived from established software development practices.

For the purpose of this paper, we define a large scale macro as one that performs significant data collection, massaging, analysis and/or reporting tasks. We will often refer to these as ‘systems’ or ‘macro systems’ because they are complex and perform many functions.

For example, one of the large scale macros developed for Boehringer-Ingeheim is used for analysis and reporting of lab data and creates over 30 different types of standard reports and has thousands of lines of code. The methodology and associated tools have been successfully applied to the development and implementation process of many large and small macros at our company.

The challenges posed by designing large-scale macros

Large macros bring with them a range of associated potential problems. The following is an overview of the major issues involved:
Structure: When a macro contains hundreds of steps, it is clearly impractical to simply write the code in one big macro. It makes more sense to break the code down by function into smaller, more manageable sub-macros. To avoid chaos, when a program is divided into sub-macros, the relationship between these components must be clearly defined.

Flexibility: Large macros may be used for many years. During that time period, it is likely that changes will be required to alter existing features or add new features. The underlying structure of the macro should be built in such a way that it is easy to make these changes.

Versioning: As the macro undergoes change, it is necessary in our company-wide environment to be able to implement new versions while still keeping older versions of the macro available. A method is needed to handle the versioning process.

Documentation: Good documentation forms the basis for communication between all development team members including users, designers, developers and testers. It can be almost as challenging to maintain macro user requirements, functional specifications, design specifications and test plans as it is to write the actual macro. However, this documentation is crucial for large-scale macros with increased complexity, especially when a team of developers are working on the same macro. If possible, macros should be built so that documentation becomes an integral part of the programming process.

Naming Conventions: Large systems can contain hundreds of different objects that need to be named. It is probable that the most commonly used approach to naming datasets, variables and macros in SAS® is to simply use an ad-hoc abbreviated description. Establishing a set of rules for naming objects can help to make code more understandable between developers. For example, it may aid understanding and ease coding if naming conventions are used that number objects and/or use names that clarify relationships.

Debugging: Debugging a large macro can be a daunting task. The SAS® log is used for debugging purposes, but with large macros the log can get filled up very quickly and become unmanageable. Methods should be used so that the location of bugs within the macro can be easily pinpointed.

Testing: A large macro can be a challenge to test. When a macro contains many input parameters, it may not be possible to simply use ‘brute force’ testing methods to test all input parameter combinations. A well thought out test plan should be developed that addresses both ‘Black-box’ and ‘White-box’ testing. The macro structure and modules should be designed to ease the testing process.

Reusable code: When building a suite of large scale macros, there may be parts of the macro that can use the same code. These functions should be identified and shareable code should be developed.

Environment Specific Concerns: Every implementation will have to satisfy certain environment-specific requirements. This may include operating system, or remote processing issues. Managing SAS® resources including macro variables and the WORK folder can also be a concern. If possible, these types of environmental concerns should be addressed within the
macro structure to minimize the amount of coding needed to meet these common requirements. For example, this might include handling remote processing with a macro call and/or creating a separate work folder for the macro to use independently.

**Regulatory Requirements:** The industry for which the macro is being developed may contain additional regulatory requirements. As an example, for the pharmaceutical industry, software development it is necessary to demonstrate (during regular audits) that a software development life cycle has been established and is being followed. This includes traceability requirements to show that the code updates are reflected in the documentation.

**The Software Development Process**

A number of software development process standards are used to promote best-practices and/or to evaluate the competency of software-producing organizations. These standards include, but are not limited to, the Rational Unified Process, ISO 9001 Standards, and the ISE Capability Maturity Model. Elements of the ISE Capability Maturity Model will be described here to demonstrate the key features of a solid software development process.

**The ISE Capability Maturity Model**

Carnegie Mellon University developed the Capability Maturity Model℠ (CMM) for Software [6] for the U.S. government to help them assess companies bidding on large software projects. This model provides a method for rating the relative 'maturity' of an organization’s software development process. Using this model, an organization’s ability to create software is rated on a scale from 1 (least mature) to 5 (most mature). Unfortunately, software development practices in most companies would fall within Level 1. The model, however, can be instructive for any software-producing organization to see where they can improve their process.

Although no official CMM audit has been performed, the software development methodology, tools and processes that we discuss in this paper go a long way towards satisfying the best practices of a Level 3 organization – a defined software development process. The activities of such an organization at this level of maturity include (from [6]):

- A standard software process for the organization is developed and maintained.
- Requirements for a software product are well established and the resulting software is kept consistent with these requirements.
- Software estimates are documented for use in planning and tracking the software project.
- Appropriate software engineering methods and tools are integrated into the project’s defined software process.
- The software code is developed, maintained, documented, and verified according to the project’s defined software process.
- The documentation that will be used to operate and maintain the software is developed and maintained according to the project’s defined software process.
- System and acceptance testing of the software are planned and performed to demonstrate that the software satisfies the requirements.
- Consistency is maintained across software work products, including the software plans, product descriptions, allocated requirements, software requirements, software design, code, test plans, and test procedures.
• Peer review activities are planned
• Defects in the software work products are identified and removed.

The BI Software Development Lifecycle

The standard software development lifecycle for the development of the BI SAS® macros is described in the following diagram (please refer to the table of step descriptions following the diagram):

<table>
<thead>
<tr>
<th>STEP</th>
<th>STEP DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>User Requirements</strong>&lt;br&gt;User Requirements for the functionality of the macro are provided by the user-community. This includes information about the analysis and reporting inputs and outputs for the macro system. This specification may include business-specific processing rules and rough examples of reporting output.</td>
</tr>
<tr>
<td>2</td>
<td><strong>Functional Specifications / Traceability Matrix</strong>&lt;br&gt;The functional specifications describe the functionality and/or processes of the macro that will satisfy the user requirements. This document is created from a standard template. The structure of the input and output datasets, and macro parameters are also defined. When needed, more detail regarding business-specific processing rules is included. A traceability matrix is created that maps user requirements to macro functional specifications.</td>
</tr>
<tr>
<td>STEP</td>
<td>STEP DESCRIPTION</td>
</tr>
<tr>
<td>------</td>
<td>------------------</td>
</tr>
<tr>
<td>3</td>
<td><strong>Module Shells / Program Meta</strong>&lt;br&gt;The macro is structured with a number of component modules. In this step the module outlines and supporting data are created:&lt;br&gt;a. All planned module shells are created (without the code), each containing documentation describing what the module will do and a time-estimate for completion of the module. Another macro tool will automatically extract the documentation from these modules to create the preliminary (in Step 4) and final (in Step 9) design specification document.&lt;br&gt;b. Some modules are data driven. The meta-data that drives these modules is created in an Excel spreadsheet. These tables include the input parameter processing specifications and specifications for driving the text-based report processing modules of the program.</td>
</tr>
<tr>
<td>4</td>
<td><strong>Design Specs (first draft)</strong>&lt;br&gt;A draft design specification insures that a program structure and plan for development is in place. The first draft of the design specification is created automatically by extracting the documentation from the module shells (created in step 3a). The program meta-data is also transferred from the Excel spreadsheets into the design specification.</td>
</tr>
<tr>
<td>5</td>
<td><strong>Time Estimates</strong>&lt;br&gt;Time estimates for the project are automatically extracted from the module shells (created in step 3a) and converted into an official time estimate document.</td>
</tr>
<tr>
<td>6</td>
<td><strong>Macro Development</strong>&lt;br&gt;By the time the actual macro development begins, the structure of the macro has been clearly defined. Work is divided up between developers and the code is written into the empty module shells. By this stage, each shell already contains a description of what the module should do. This will help the developer code the logic for that module. As the code is written, additional documentation is added in the module, as needed. The meta-data defined in Step 3b may also be further refined.</td>
</tr>
<tr>
<td>7</td>
<td><strong>Peer Review</strong>&lt;br&gt;At planned intervals the macro code undergoes a peer review process. The programmer(s) performing the review will examine the underlying source code, the functional specification, and the design specification and will provide comments and/or suggestions on each module. A standard code review template in Microsoft Word is used.</td>
</tr>
<tr>
<td>8</td>
<td><strong>Developer Testing</strong>&lt;br&gt;A module-level and system-level developer test plan is written and executed by a programmer who is not working directly on development. A macro-based tool is used to test each module and to automatically create a test results document.</td>
</tr>
<tr>
<td>9</td>
<td><strong>Design Specs (final draft)</strong>&lt;br&gt;Before the macro is released for user-acceptance testing, the final design specification document is generated. This document will include all of the changes made during the development process to the original design.</td>
</tr>
<tr>
<td>10</td>
<td><strong>User Acceptance Testing</strong>&lt;br&gt;A user acceptance test plan was created during the development phase of the macro. This test plan is executed by a designated user and must pass prior to release. All tests are mapped into the traceability matrix to insure that each feature has been tested.</td>
</tr>
<tr>
<td>11</td>
<td><strong>User Training, Macro Release</strong>&lt;br&gt;Users are trained prior to the official release of the macro. The new macro (or version) is distributed as a compiled macro catalog to all company locations.</td>
</tr>
</tbody>
</table>

**Software development tools**

Software development tools help support this development process in the following ways:

- The modular structure of the macro is defined and the modules are created before any coding begins. This helps to clarify the program design and accelerate the implementation phase.
• Design documentation is stored in standard macro modules and a design specification document is extracted from the modules.
• Macro ‘sub-systems’ eliminate the need for manual coding of specific program functions.
• Automated testing tools aid the organization, execution and documentation of developer testing.

Macro System Structure

Modular Program Structure

The concept of modular, or component-based program design was developed in the late 60s / early 70s. In the seminal paper, “The Modular Structure of Complex Systems” [2], the authors state:

The primary goal of the decomposition into modules is reduction of overall software cost by allowing modules to be designed and revised independently.

Modular programming consists of segmenting a larger program into multiple small macros which each perform a specific task. In order for the modules to act independently, the inputs and outputs for each module must be clearly defined. David Parnas [1] was one of the first authors to write about the process of modular decomposition. In this paper, he describes the following two possible decomposition methods:

1. Flowchart Method (similar to the process flowchart for a program).
2. Information Hiding Method (called the ‘abstraction’ or ‘encapsulation’ method)

An example of the flowchart method would include breaking down an ordered data-building process. For example, an analysis of adverse event data might require that the following steps be performed in order:

1) Merge AE and treatment data
2) Collapse AE data to one record per episode
3) Merge AE dictionary data

In this case, it might make sense to simply divide up the task into three separate modules.

An example of the information hiding method would include checking the validity of input macro parameters. In this case, a macro is created that checks the validity of input parameters against a dataset containing valid values and returns a message whether or not valid values were entered. In this way, the parameter-checking feature is contained in this module and the details of this process are ‘hidden’ from the rest of the program.

Our approach has been to use a combination of these two methods. In general, we design the program using the flowchart method, but when we feel it is valuable, we build general-purpose modules (also called sub-systems) that perform more general tasks such as parameter checking or report output generation.
**Program Hierarchy**

Regardless of the method used, modular decomposition is usually an iterative process. A program is first divided into higher-level modules (e.g. check inputs, perform analysis, generate output), further sub-divided into smaller sub-modules, and so on. The number of decomposition levels used should depend on the scope and purpose of the software. For example, the modular structure for the A-7E flight software (described in [2]) has become a model that has been emulated throughout the computer industry. This software was designed using a three-level modular decomposition.

For our system, it was determined that a two-level decomposition was adequate. The decomposition and the notation that we use will be described here.

First, our macro system is divided into high-level components called blocks. An example is presented in the table below. Each block performs a certain general task and is assigned a 2-digit number that specifies its processing order. An example of a typical block structure might look like the following table:

<table>
<thead>
<tr>
<th>Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Process inputs</td>
</tr>
<tr>
<td>20</td>
<td>Generate analysis dataset</td>
</tr>
<tr>
<td>30</td>
<td>Create report-specific analysis</td>
</tr>
<tr>
<td>40</td>
<td>Generate output</td>
</tr>
</tbody>
</table>

Each of the blocks is further broken down into separate macros that we call modules. Each module is assigned the 2-digit block number + an additional 2-digit identifier. Finally, each macro system is assigned a 3-digit system prefix. For this example, our XAE macro system uses the prefix ‘XAE’. This prefix is used at the beginning of each module name. A sample block and modular structure for an ‘XAE’ macro can be seen in the following table.

<table>
<thead>
<tr>
<th>Full Module Name</th>
<th>Module Description</th>
<th>Block # and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAE1010</td>
<td>Check input macro parameters</td>
<td>10 -- Process input parameters</td>
</tr>
<tr>
<td>XAE1020</td>
<td>Read and Check input datasets</td>
<td></td>
</tr>
<tr>
<td>XAE2010</td>
<td>Merge AE and Treatment Data</td>
<td></td>
</tr>
<tr>
<td>XAE2020</td>
<td>Collapse AE Records</td>
<td>20 -- Generate analysis dataset</td>
</tr>
<tr>
<td>XAE2030</td>
<td>Apply MedDRA coding to AE data</td>
<td></td>
</tr>
<tr>
<td>XAE3010</td>
<td>Transpose data for AE listing</td>
<td>30 -- Create report-specific analysis</td>
</tr>
<tr>
<td>XAE3020</td>
<td>Calculate N (%) for AE tables</td>
<td></td>
</tr>
<tr>
<td>XAE4010</td>
<td>Create AE listing output</td>
<td>40 -- Generate output</td>
</tr>
<tr>
<td>XAE4020</td>
<td>Create AE table output</td>
<td></td>
</tr>
</tbody>
</table>

Note that the real adverse event macro developed at BI uses many more modules and that this is a simplified example.
Block execution

To execute a block of modules, we use a general purpose `%XRUN` macro. For example, to execute block 10 of the AE macro, the following macro call is used:

```
%XRUN(XPREFIX = XAE10);
```

The `%XRUN` macro will find all macros stored in the WORK.SASMACR catalog that begin with the prefix `XAE10` and will execute them in order. Using our previous example, the `%XRUN` macro will execute all modules in block 10 in the following (alpha-numeric) order:

```
%XAE1010;
%XAE1020;
```

Using the `%XRUN` macro adds flexibility to the macro system. For example, if an additional module is needed to cross-check input parameters, a new module `XAE1030` can be created. When `%XRUN` executes it will find this module and execute it. This open system structure also makes it possible to create custom on-the-fly system modifications.

It is also possible to use the `%XRUN` macro to select a list of modules to run. For example, in our sample AE macro, it is not necessary to run every module in block 30 every time the macro runs. E.g., if only a table is created, then only `XAE3020` should execute.

In this case, the following `%XRUN` call is used:

```
%XRUN(XPREFIX = XAE30, XLIST = &XANAL);
```

This macro will run all macros beginning with `XAE30` and contained in the list of modules specified by the `XLIST` macro parameter.

Finally, `%XRUN` also provides housekeeping functions to handle both local and remote processing.

The Shell Macro and Kernel Macro

The macro’s structure at the highest level consists of two macros: the shell macro (this is called by the user) and the kernel macro (that controls the execution of the macro’s blocks). The structure of these macros are intimately related to our version control implementation.

```
%MACKRO XAE( VERSION = , ... )

%XSYSTART( VERSION = &VERSION, ... );

%XAEKERNEL;

%XSYSTOP;

%MEND;
```
This shell macro, %XAE, is called by the end-users and contains three macro calls and nothing else. In our version control implementation, the same shell macro is used for all versions of the macro system. Therefore, the shell macro cannot contain any code that may need to change in a future version. The %XSYSTART macro performs system set-up tasks. This includes loading the version-specific modules (from catalogs) and the kernel macro (%XAEKERNEL). The %XSYSTOP macro performs system clean-up tasks at the end of program execution.

The %XAEKERNEL macro is the master control macro that controls the execution of the system blocks, as seen below:

```
%MACRO XAEKERNEL;
%XID(...);
%XRUN(BLOCK = XAE10);
%XRUN(BLOCK = XAE20);
%XRUN(BLOCK = XAE30);
%XRUN(BLOCK = XAE40);
%XRP(...);
%MEND;
```

This macro contains %XRUN calls to execute blocks of modules. The macro also contains calls to %XID and %XRP. These are macro sub-systems that process input parameters and create text output respectively.

**System diagram**

The following is a sample diagram that describes the system-level structure defined in previous sections. A macro in the diagram can call any connected macro that is directly beneath it.
This structure is planned during the design phase of macro development. The structure, therefore, is
decided before any code is written.

Macro Storage and Version Control

There are two possible reasons for making a change to a macro system: (1) to fix a system bug, or (2)
to alter or enhance macro system functionality. It is a requirement at BI that it be possible to go back
and recreate reports exactly as they were submitted to the regulatory authorities, even if the report was
created many years ago. Because of this, a simple SAS®-based version control system was
developed so that all macro versions are available to the users.

Boehringer-Ingelheim uses NetInstall for software distribution. When users click on the custom SAS®
icon, NetInstall is launched first to update the user’s environment.

Within this environment, each user will have the same STDMAC library that contains SAS® catalogs of
compiled macros:

Identical copies of this library are placed on the local and remote machines. The STDMAC library
contains a SASMACR catalog that contains compiled macros. This catalog is linked to the SAS®
autocall library so the macros may be called by the end-users. The end-user shell macros are only
stored in this SASMACR library. As seen above, the XSYSTART macro is also stored in this catalog.

Each of the other catalogs contain a separate version of compiled sub-macro (for example, see the
sample modules to the XAE macro described in the previous section). The naming system for the
macro catalogs are {System Prefix} + 5 digit version number. For example, XLB00102 is the catalog
name for XLAB version 1.02. See the above diagram for the contents of the sample XAE00100
catalog.

All catalogs contain only compiled macros. Because it is necessary to store additional version
information (such as update notes, and the date the version became effective), each version catalog
also contains a macro called %XLABEL that contains version specific meta-data. In other words,
%XLABEL acts like a label for each catalog. The %XSYSTART macro 'reads' the labels from each
catalog and builds a dataset of version meta-data. By default, the most-current version will be selected, but user parameters may be used to select a specific version number or to select the version that was effective at a defined date. The selected version is displayed in the log, so that the user knows which version was used.

Sub-systems that perform a specific task are also stored in macro catalogs. E.g. a sub-system called XXX stores common sub-macros like %XRUN that are used to support the system structure. The sample %XSYSSTART call below specifies which sub-systems to load. %XSYSSTART will load all components for the XAE macro, as well as the XID, XRP and XXX sub-systems:

```
%XSYSSTART( XSUBSYS = XAE XID XRP XXX , ...);
```

**Program Documentation**

The primary requirement of system documentation is to help “software engineers / programmers understand previously written applications, so that subsequent maintenance and evolution is made easier and more predictable.” [3] However, documentation of a large macro system that contains many modules can be a challenge. Often, there is not enough time to properly document a macro system, and when documentation is written it can quickly fall out of synchronization with the underlying source code.

Our solution to this problem was to integrate the documentation into the structure of the macro module and to create a set of tools that automatically extract this documentation and transform it into an RTF design specification document. The tools that have been developed serve a similar purpose as the Java-doc tools used in the Java programming language [4].

Specifically, most of the program documentation is stored within the input macro parameters of each module. These parameters are part of a standard module template. Another macro called XMR (for X-Macro Report) automatically generates the sections of a macro design specification. This macro can extract the documentation from each module and produce a report.
The following displays are samples of program module documentation that was created automatically by the XMR macro:

A simple overview of macro system blocks and modules

<table>
<thead>
<tr>
<th>Block / Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 00</td>
<td><strong>Macro Parameter Meta-data</strong></td>
</tr>
<tr>
<td>XAE0020</td>
<td>Set up Default values of macro parameters</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block 01</th>
<th><strong>Preparation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>XAE0130</td>
<td>Set up Default values of macro parameters</td>
</tr>
<tr>
<td>XAE0135</td>
<td>Create the dataset with deterioration criteria</td>
</tr>
<tr>
<td>XAE0145</td>
<td>Create Table Template 1 summary rows dataset</td>
</tr>
<tr>
<td>XAE0150</td>
<td>Create Age Category dataset</td>
</tr>
<tr>
<td>XAE0170</td>
<td>Create Collapsing/Condensing criteria dataset</td>
</tr>
<tr>
<td>XAE0185</td>
<td>Create Table Where dataset</td>
</tr>
</tbody>
</table>

Summary of information about each module

**XAE2040 - Add Dictionary Levels to AE Episodes**
Description:
This macro will add all needed dictionary information to the dataset containing AE episodes; depending on the input parameters, special search categories or project-specific special search categories will be added.

**Input Datasets:**

<table>
<thead>
<tr>
<th>Internal Name</th>
<th>External Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XINDATA</td>
<td>AEWORK XAE2030G</td>
<td>Collapsed AE dataset</td>
</tr>
<tr>
<td>XINDATB</td>
<td>AEWORK XAE1520A</td>
<td>Work dictionary dataset</td>
</tr>
<tr>
<td>XINDATC</td>
<td>AEWORK XAE1520B</td>
<td>Input Special Search Category dataset</td>
</tr>
<tr>
<td>XINDATD</td>
<td>AEWORK XAE1520C</td>
<td>Input User-defined Category dataset</td>
</tr>
</tbody>
</table>

**Output Datasets:**

<table>
<thead>
<tr>
<th>Internal Name</th>
<th>External Name</th>
<th>Description</th>
<th>Delete?</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOTDATA</td>
<td>AEWORK XAE2040A</td>
<td>Working dataset</td>
<td>YES</td>
</tr>
<tr>
<td>XOTDATB</td>
<td>AEWORK XAE2040B</td>
<td>Working dataset</td>
<td>YES</td>
</tr>
<tr>
<td>XOTDATC</td>
<td>AEWORK XAE2040C</td>
<td>Working dataset</td>
<td>YES</td>
</tr>
<tr>
<td>XOTDATD</td>
<td>AEWORK XAE2040D</td>
<td>Working dataset</td>
<td>YES</td>
</tr>
<tr>
<td>XOTDATE</td>
<td>AEWORK XAE2040E</td>
<td>AE dataset with Dictionary information</td>
<td>NO</td>
</tr>
</tbody>
</table>

**Input Macro Variables:**

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Variable Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDICTTYP</td>
<td>Dictionary Type</td>
</tr>
<tr>
<td>XSSC</td>
<td>SSC Needed</td>
</tr>
<tr>
<td>XUDAEC</td>
<td>UDAEC Needed</td>
</tr>
</tbody>
</table>

Module Outline:

- REF010 DATA STEP: Skip MedDRA merge if WHO data is used
  - INPUT: XINDATA
  - OUTPUT: XOTDATE

- REF020 DATA STEP: Remove variables from input dataset
  - INPUT: XINDATA
  - OUTPUT: XOTDATA
  - [More steps deleted]
The next table shows where each dataset in the macro system is used:

<table>
<thead>
<tr>
<th>Dataset (Input dataset)</th>
<th>Action</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;INAE (Input AE dataset)</td>
<td>Used by XAE0020</td>
<td>(Set up Default values of macro parameters)</td>
</tr>
<tr>
<td></td>
<td>Used by XAE1205</td>
<td>(Create Subset of MedDRA dictionary)</td>
</tr>
<tr>
<td>&amp;INAEANAL (Input AE Analysis dataset)</td>
<td>Used by XAE0020</td>
<td>(Set up Default values of macro parameters)</td>
</tr>
<tr>
<td>AEWORK.XAE0020A (Input AE Analysis dataset)</td>
<td>Used by XAE0130</td>
<td>(Set up Default values of macro parameters)</td>
</tr>
<tr>
<td></td>
<td>Used by XAE0195</td>
<td>(Plausibility of macro parameters values)</td>
</tr>
</tbody>
</table>

**Standard Module Template**

The system design documentation that was displayed in the previous section is extracted from text stored in the module input macro parameters. A sample of the module input parameter section is presented below.

```%MACRO XLB2027 (```

<table>
<thead>
<tr>
<th>OBJECT NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOBJ = XLB2027, LOBJ = Object Name,</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OBJECT TITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>XTITLE = Create Baseline and Last Value Marker, LTITLE = Object Title,</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BLOCK DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBLOCK = Block Description, XBLOCK = 20 - Lab Data Calculations,</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDESCRIP = DESCRIPTION, XDESCRIP = %STR(&lt;X&gt; Baseline Shall be the pre-drug value closest to day 0, &lt;X&gt; Last Value Shall be the last on-drug value, &lt;X&gt;),</td>
</tr>
</tbody>
</table>

Line breaks for documentation
/*------------------------------------------  
* | INPUT MACRO VARS                       |
*------------------------------------------*/

XBASELIN = &BASELINE,
LBASELIN = Baseline determination method,
XSORT1 = STUDY XTRT LABNM PTNOS,
LSORT1 = Main Sort Order,

/*------------------------------------------  
* | INPUT DATASETS                         |
*------------------------------------------*/

XINDATA = LABWORK.&LABDATA,
LINDATA = Lab Data,
XINDATB = LABWORK.&LABDATA,
LINDATB = Lab Data,

/*------------------------------------------  
* | OUTPUT DATASETS                        |
*------------------------------------------*/

XOTDATA = LABWORK.XLB2027A,
LOTDATA = Lab Data,
XOTDATB = LABWORK.XLB2027B,
LOTDATB = Patient-Lab Baseline Flag,

/*------------------------------------------  
* | DELETE OUTPUT DATASETS ?                |
*------------------------------------------*/

XOTDELA = NO,
LOTDELA = Delete Output Dataset A,
XOTDELB = YES,
LOTDELB = Delete Output Dataset B,

/*------------------------------------------  
* | OUTLINE           
*------------------------------------------*/

LOUTLINE = OUTLINE,
XOUTLINE = %STR(
<X> - REF001 PROC SORT: Lab Data for Baseline
<X> - INPUTS: XINDATA var XSORT1
<X> - OUTPUTS: XINDATA
<X> -
<X> - REF002 DATA STEP: Generate Baseline Lab Marker
<X> - INPUTS: XINDATA var XSORT1
<X> - OUTPUTS: XOTDATA XOTDATB
<X> -
<X> - REF003 PROC SORT: Sort patient labnm baseline data
<X> - INPUTS: XOTDATB var XSORT1
<X> - OUTPUTS: XOTDATB
<X> -
<X> - More .... More.......  
),

/*------------------------------------------  
* | Macro Switches                          |
*------------------------------------------*/

XSWITCH1 = &DEBUG,
LSWITCH1 = Debug Mode,
XSWITCH2 = &DELETESD,
LSWITCH2 = Allow Deletion of Output Datasets,
XSWITCH3 = &ABORTERR,
LSWITCH3 = Abort If Error,
XSWITCH4 = &MACRODOC,
LSWITCH4 = Output Macro Documentation,
XSWITCH5 = &RUNMACRO,
LSWITCH5 = Execute Macro,
A few key points about the module header:

- Macro variables come in pairs – one beginning with X and another beginning with L. The variable that begins with an X contains the macro parameter value. The variable that begins with an L contains a label for the macro parameter. The label helps the programmer who is reviewing the module and is also extracted into the design specification.

- All input datasets, output datasets and input macro variables are specified in the macro header. This insures that the program documentation describes all module inputs and outputs. Because all of the inputs and outputs for each module are clearly specified, this makes it possible to develop the modules independently.

- Output dataset names have the following form: [Module Name] + [letter A-Z]. For example, we define XOTDATB = AEWORK.XAE2040B. This identifies where the dataset was created. This naming system also makes it easy to scan the work folder for a particular dataset.

- The module description (XDESCRIP) is filled out during the design stage before any code is written. After the design is complete, the other macro parameters are filled out and the code is written.

Each module begins with a call to the XSTART sub-macro and ends with a call to a XSTOP sub-macro. These modules are part of the XXX sub-system so they are available for all macros to use. These macros perform housekeeping functions such as deleting temporary datasets, extracting macro parameters for documentation, error handling and launching a testing sub-macro.

Coding Standards

To maintain consistency, developers are directed to adhere to a number of coding standards. A few of these standards are described here:

- Developers will include in-line commented documentation of their macro code. In addition to this, before each PROC or DATA STEP, a one-line description is embedded in the %XPUT macro:

  %XPUT(REFO20 DATA STEP: Remove variables from input dataset);

  The REF number specifies the position of the PROC or DATA STEP in the module. The text in XPUT may optionally be displayed in the log with the module name and timestamp. This makes it possible to identify the exact place in the code where an error occurred. This also facilitates the detection of performance bottlenecks since every step contains a timestamp. Sample log:

  ---- 00:09:30 AE2040 REFO20 DATA STEP: Remove variables from input dataset

- If an error occurs (e.g. because the input dataset was missing specified variables), then the %XERROR macro is called. This macro sends an error message to the log and creates an error dataset that acts as a flag. When the error dataset is present, the macro performs an orderly shutdown.
• When datasets are created within the module, the following form is used:

```
DATA &XOTDATB (LABEL = "&SYSDATE &LOTDATB);
```

The X-variable is used as the dataset name. The L-variable is used as the dataset label (with &sysdate date-stamp). This insures that the dataset label in SAS® matches the dataset description in the design specifications.

### Data-driven application design

The concept of data-driven programming is described in [5]:

> Application design is perceived as data and nothing but data. This means that the application design is defined in a set of specially structured tables and is stored, updated, and managed in the same way as ordinary data. This is also, at times, called meta-data.

With data-driven programming, a dataset containing processing instructions is used to drive a program’s behavior. A macro is written that can interpret these instructions and perform the required processing.

For our projects, data-driven programming techniques have been applied to the processing of input parameters and the generation of text output. We constructed two sub-systems that perform these tasks. This approach has drastically reduced programming and maintenance time for projects, as this has eliminated the need to write custom code to handle these tasks.

#### The XID and XRP sub-systems

The XID sub-system is used for initial processing of a macro’s input parameters. XID can automatically check the validity of input parameters and can set parameter defaults. The validation and processing instructions for each user parameter are stored in a dataset called the input description file.

Due to the highly customized nature of the report output created at BI, traditional SAS® tools such as PROC REPORT were not sufficient to meet these requirements. The XRP (X-report) sub-system was developed to create text output. Report layout definitions are stored in a dataset called the report description file. This report description file describes the entire layout and content for each report (table or listing). In essence, a report’s definition is broken down into specifications for several different report regions (like title, footnote, column header, etc.) and stored in a dataset. There is an added benefit with this approach: by modifying the report description file, users are able to further customize their reports.

#### Structure of the data-driven meta-data

Program meta-data for the input description file and report description file is stored in a vertical data structure. Each row of data defines a single property. For example, if the parameter RSUBMIT has valid values of YES or NO, then this information would be stored as follows:
<table>
<thead>
<tr>
<th>Column</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>RSUBMIT</td>
<td>Name of macro parameter</td>
</tr>
<tr>
<td>Type</td>
<td>PTYPE3</td>
<td>Macro parameter type 3 means that RSUBMIT is a single-valued parameter.</td>
</tr>
<tr>
<td>Property</td>
<td>VVALS</td>
<td>Property type – in this case VVALS contains means that this row contains a list of valid values.</td>
</tr>
<tr>
<td>Row</td>
<td></td>
<td>Not used with the VVALS property</td>
</tr>
<tr>
<td>Column</td>
<td></td>
<td>Not used with the VVALS property</td>
</tr>
<tr>
<td>Value</td>
<td>YES NO</td>
<td>The list of valid values. The RSUBMIT parameter may be equal to YES or NO.</td>
</tr>
</tbody>
</table>

The XID sub-system would read this meta-data and use it to check that the RSUBMIT parameter contains a valid value. As another example, the XID system can also check that required variables are contained in an input dataset. The following meta-data defines AEINT (AE intensity) as a required variable in the INAE (input AE dataset) macro parameter:

<table>
<thead>
<tr>
<th>Column</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>INAE</td>
<td>Name of macro parameter</td>
</tr>
<tr>
<td>Type</td>
<td>PTYPE4</td>
<td>Macro parameter type 4 means that INAE is a macro parameter that points to an input dataset.</td>
</tr>
<tr>
<td>Property</td>
<td>VARNAME</td>
<td>Identifies a variable name required in the INAE dataset.</td>
</tr>
<tr>
<td>Row</td>
<td></td>
<td>Not used with the VARNAME property</td>
</tr>
<tr>
<td>Column</td>
<td>3</td>
<td>ID number of the required variable. Each required variable is given a different ID number.</td>
</tr>
<tr>
<td>Value</td>
<td>AEINT</td>
<td>The name of the required variable</td>
</tr>
</tbody>
</table>

The structure of the report description file is similar. The following example demonstrates how a title would be defined:

<table>
<thead>
<tr>
<th>Column</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report</td>
<td>LISTING1</td>
<td>Meta-data for standard listing 1</td>
</tr>
<tr>
<td>Region</td>
<td>TITLE</td>
<td>Title region displays titles at the top of the page.</td>
</tr>
<tr>
<td>Property</td>
<td>TEXT</td>
<td>This property defines the text to be displayed in the title</td>
</tr>
<tr>
<td>Row</td>
<td>1</td>
<td>Row number of the title</td>
</tr>
<tr>
<td>Column</td>
<td></td>
<td>Columns are not used with the TITLE region since titles spread across the page</td>
</tr>
<tr>
<td>Value</td>
<td>AE occurrences..</td>
<td>The text for the title. May contain macro parameters.</td>
</tr>
</tbody>
</table>

**Test Environment**

A macro-based test environment is used to perform developer testing on individual modules. This tool allows a developer (usually another programmer) to programmatically test each module with a set off rules. For all practical purposes, this technique allows us to check the internal logic of the module’s code and qualifies as a ‘white box’ testing tool. Detail and summary reports are automatically generated to show what and how modules were tested and the pass/fail status. These reports also become part of our overall macro documentation.
References


Acknowledgements

SAS is a Registered Trademark of the SAS Institute, Inc. of Cary, North Carolina.

Contact Information

John Adams jadams@rdg.boehringer–ingelheim.com

Jeremy Gratt jeremy@modularinformatics.com