Paper 126

Adding Extensions to the SAS/Warehouse Administrator™

Peter R. Welbrock
Strategic Information Systems, Inc.
Philadelphia, PA

Abstract

It is highly unlikely that any administrative tool for a data-warehousing project can functionally encompass all requirements. Each data-warehousing project is, by necessity, very different from all others. The SAS/Warehouse Administrator™ (known as the Administrator through this paper), using the API available with Release 1.3, allows the designers of the data warehousing process to build their own extensions. This preserves one of the essential prerequisites of all data warehousing projects: that of flexibility. This paper covers the three major stages of adding an extension to the Administrator:

- Recognizing the need to build an extension.
- Building the extension through the use of Frames
- Importing data from the extension into the Warehouse Administrator using the API

To illustrate the techniques, the paper will use the example of incorporating seminal explicit business needs into the data-warehousing process and then moving some of this metadata into the Warehouse Administrator.

Introduction

Successful data-warehousing projects are usually forward, rather than backward engineered. A forward-engineered project is one where the business needs of the organization are initially documented, and then the data-warehousing process is built around those needs. A backward-engineered project is where the data is gathered and then the business somehow has to find a way to use this data to meet its own ends.

This latter approach, the backward-engineered process is usually techno-centric. This means that technological issues are usually considered before business issues. Commonly, the Information Systems departments, rather than the business units drive these data-warehousing processes. The problems arise when these data-warehousing processes do not meet the business goals (if indeed any have ever been defined). It is possible to build a process that avoids this pitfall[i], but one of the major problems of doing this is the lack of tools available to aid in this process.

The technological part of the data-warehousing process: the extraction, transformation and loading of the data warehouse have tools available for both administration and management. One of these tools is available from SAS Institute, the SAS/Warehouse Administrator™. This software tool, if correctly used, can help the data warehouse designers to map both the elements that make up the data warehouse, and the process and techniques by which those elements are either referenced or created. As already mentioned, it is an administrative and management tool, not a design tool. If the data warehouse design is faulty, the Administrator can, at best, help create an efficient and well-documented *flawed* data warehouse.

It is very important for the data warehouse designer to understand that a tool like the Administrator is not the answer to sound design, but is an implementation tool. To create a software tool that will design (as well as implement) the data-warehousing process could well be a fruitless task. The design process is in all probability so fundamentally different between data-warehousing projects that it would end up being 'nothing to all men'. The implementation of the data warehouse, however, has common threads:

- The data must come from somewhere, therefore *extraction* must take place.
- The data will not, in all likelihood, be in the required form, so it will have to be *transformed* in some way (including all the transformation steps[ii])
- The data must be *loaded* into the data warehouse.

There are more steps than these three (e.g. the summarization of data and the creation of data marts) but the point is that there are common steps. Because of this commonality, it is

possible to design a generic tool that will help in the implementation of these disparate steps within the data-warehousing process. This, however, raises the question as to how the remainder of the data-warehousing process can be both documented and implemented. There are two distinct parts to this question:

- What exactly are the pieces of the process that the Administrator cannot satisfy?
- How is it possible to coordinate these pieces with those documented in the Administrator?

Why Build An Extension To The Sas/Warehouse Administrator?

As discussed above, there are two major pertinent points:

- Successful data-warehousing projects are forward-engineered.
- The Administrator manages and implements the data-warehousing process, but it does not design it.

Based upon these two points, it makes sense that the addition of a tool that aids in the forward-engineering of the data warehousing process, that can link to the administrative abilities of the SAS/Warehouse Administrator will help in the formulation of a successful data warehousing project.

This paper covers one example of building an extension. This example might be typical of the requirements of many data warehousing processes, but should not be used as a generic model upon which to base such extensions. All data warehousing processes differ from each other, especially in design. This example covers one part of one methodology and is not intended to be all encompassing.

Extension Assumptions and Design

This extension is based upon the methodology outlined in detail the paper: *Is Your Data Warehouse Successful? Developing a Data Warehouse Process that responds to the needs of the Enterprise*[iii]. This paper purports a three-tier methodology to designing a data-warehousing process:

- The design of a *Conceptual Data Warehouse* that contains the business needs that have to be addressed.
- The development of an intermediate stage, or the *Transition Phase* of the process, where the defined business needs are translated into their technological equivalent.
- The building of the *Implemented Data Warehouse* which is the classic interpretation of the data-warehousing process. This phase starts with the extraction of data and ends up with the transformed data being made available to the business users.

As already mentioned the Administrator best addresses the final of these three stages. It does not explicitly handle the creation of business requirements and map these into the implemented data warehouse.

What if, however, it would be possible to extend the formalization of the data-warehousing process beyond just the implementation? If it were truly possible to put together a data-warehousing process that was driven by the business needs. This would lead to a data warehouse that would be both encompassing and efficient, with the minimum of redundancy.

One of the problems that a reverse engineered data-warehousing process leads to is the ‘build and they will come’ outcome. This assumes that if enough data is available to the business users, they will find a way to use it to meet their needs. This assumption is flawed and leads to a ‘limited use’ data warehouse that has a niche appeal to the enterprise. This situation would be avoided with a data-warehousing process based upon the business needs. To do this in a scalable fashion, however, there needs to be a tool that can both coordinate and communicate with the implementation manager (the Administrator). In the example within this paper, this is achieved through the use of SAS/AF® and the Administrator Metadata API.

The Metadata API communicates directly with the underlying metadata within the Administrator. It does this using the SAS Screen Control Language (or SAS Component Language in Version 7 and later releases). The communication essentially takes place by using *methods* that in some way will affect the underlying metadata contained within the

Administrator. This means that to use the API, it is necessary to have at least a basic understanding of SCL.

The API will allow for metadata within the Administrator to be both extracted and updated. This means that it is possible to use SCL, *outside the Administrator environment* to directly affect the internal metadata. This makes building extensions to the Administrator relatively simple. What it also means, on a cautionary note, is that there is not guidance from the Administrator user interface to prevent any problems from arising. Another advantage to the API is that over time, the structure of the metadata within the Administrator will evolve. Using the API will mean that any code written that accesses or uses the Administrator metadata will still work, even though the underlying metadata structure has changed. This will become increasingly important, as the entire SAS software suite will move toward an intended common metadata structure.

So, what would we want to achieve by adding an extension to the Administrator? In the following example, we will be moving toward documenting[iv]:

1. The *vision* (business scope) of the data-warehousing process.
2. The *enterprise needs* that have to be addressed by the data warehouse to meet the stated *vision*.
3. The *business elements* that are the components of each of the *enterprise needs*.
4. The *data elements* that are mapped from each of the *business elements*.
5. *Owners* of each of the above elements.

There are several rules involved here:

1. Each e*nterprise Need* is unique.
2. A single *business element* can be mapped to many *enterprise needs*.
3. A single *data element* can be mapped to many *business elements*.
4. An *owner* can be mapped to any number of elements.

The above four elements: *vision*, *enterprise needs*, *business elements*, and *data elements* will be documented using a SAS/AF® user interface. Once these are documented, then two of these *elements* are directly applicable to the Administrator: the *data elements* and the *owners*.

At this time, it is not possible in the Administrator to link an owner to a column of data, only to a table level. This would be especially useful for administrative purposes from a business perspective, so this extension would allow for this. In other words, it is more than likely that the business owners of columns within a single table could be owned by more than one business owner.

Building the Extension

The data required to build this extension is outlined below. Note that this data is simplified for to illustrate the concepts of building an extension. There are five tables as follows:

Column Name	Type	Label
vision	C	Vision item desc.
visn_num	N	Unique Identifier
visn_own	C	Item Owner

Table Name: dw_visn
Label: Warehouse Vision
Table 1

Column Name	Type	Label
nd_num	N	Unique Need #
visn_num	N	Unique Vision #
nd_desc	C	Need Description
nd_own	C	Need Owner
nd_rule	C	Need Rules

Table: dw_needs
Label: Enterprise Needs
Table 2

Column Name	Type	Label
ele_num	N	Unique Element #
ele_desc	C	Element Description
ele_own	C	Element Owner
ele_rule	C	Element Rules

Table: dw_busel
Label: Business Elements
Table 3

Column Name	Type	Label
Dt_num	N	Unique data #
Dt_desc	C	Data Description
Dt_own	C	Data Owner
Dt_rule	C	Data Rules
Dt_table	C	Data Table
Dt_type	C	Column Type
Dt_len	N	Column Length
Dt_col	C	Column Name

Table: dw_dt_el
Label: Data Elements
Table 4

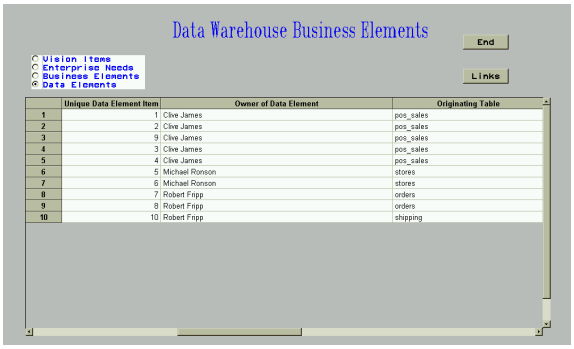
Column Name	Type	Label
Ele_num	N	Unique element #
nd_num	N	Unique Need #

Table: dw_el_nd
Label: Need to Element Relationship
Table 5

Column Name	Type	Label
Ele_num	N	Unique Element #
dt_num	N	Unique Data #

Table: dw_bs_dt
Label: Business Element to Data Element Relationship
Table 6

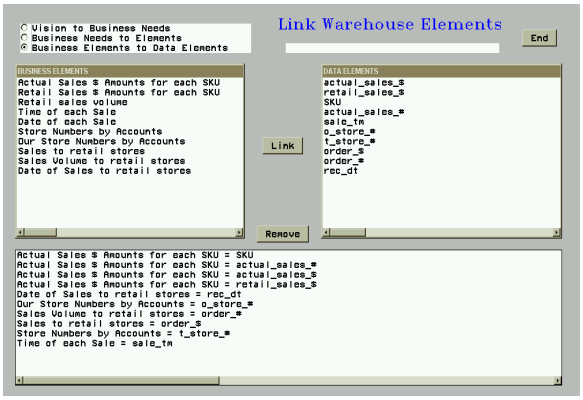
These tables are maintained through the use of two SAS/AF® frames. Although it is possible to become very fancy with these frames, again for the purposes of illustrating the major concepts, the frames in this example are fairly simple. The first frame (Table 7) simply gives the user the option of selecting one of the four element tables (Tables 1-4 above) and then editing that table using a Data Table object. Although there is no inherent referential integrity on the tables, in a production environment this would be advisable. From Version 7 onwards it would be possible to use integrity constraints that do not exist with Version 6.



Frame to update extension warehouse elements
Table 7

The second frame (Table 8) is a little more complex. It sets up the links that are required between the different elements that are documented in Tables 1-4. For example, every single Business Need (Table 2) must be associated with a Vision Item (Table 1). This is because the vision of the data warehouse will be the direct driver for the needs. Each business element will be made up of a single or multiple data elements. It will arise that in some situations, the data does not exist to actually meet a business element, in which case it cannot be linked. In those situations where an business element can be mapped to data elements, the second frame (Table 8) will do the job.

Table 8 is a frame that is made up of a single radio box (to select the type of link) and three list boxes. Two of the list boxes contain all potential elements for the link and the third list box contains the already established links. There are two push button objects that either create or remove links. From a SAS/AF perspective this frame is fairly elementary and it could be improved upon. It does however, successfully allow for a controlled way to build the relationships between the extension warehouse elements.



Frame to link extension warehouse elements
Table 8

The SCL required to build these frames is very standard. It is not the technical part of building these frames that is of interest however. It is the fact that they allow for the developer/manager of the data warehouse to begin to draw direct relationships between what the business requires (as seen by the vision and enterprise needs) and what will be needed within the data warehouse to meet these requirements (the data elements in Table 4).

Note, however, that a particular business element might require more than one data element. Nowhere in the building of these links

is the relationship between the data elements actually defined. For example suppose a business element is defined as total discounted sales. To actually calculate this, two data elements will be needed: the sales actual sales amount and the discount amount. It is obvious that to calculate the total discounted sales would require taking the discounted amount away from the actual sales. This relationship is not defined at this stage however. The reason for this is that the Administrator has the capabilities to store that information. It would be counterproductive to store that piece of metadata in two different places. Our intent here is to augment the capabilities of the Administrator, not to duplicate them.

At this stage we have a series of SAS data sets that contain seminal business that cannot be incorporated in the Administrator with two exceptions. The first exception is the metadata that pertains to data elements. These data elements actually exist and could be used as the basis for ODDs (Operational Data Definitions). In Table 4, there exists information about columns of data that exist in source systems that will be used to populate the data warehouse if the vision were to be met. The second exception is the owners of the elements.

Using the Metadata API to help populate the Administrator

Now there exists data that can be used by the Administrator, the problem of incorporating it becomes apparent. Before the API, to move this data into the Administrator would have been a case of complex programming (that would be to no avail if the fundamental model of the Administrator were to change), or retyping the data. The API allows for a consistent interchange of data between the Administrator and outside sources.

How complex is the API to use?

As with most software, there is a learning curve associated with using the API. To successfully use it, the developer will need to have at least intermediate level of skill with SCL and a thorough understanding of SCL lists[v]. The API is not be open to applications outside of SAS unless they are able to call SCL methods.

From a conceptual perspective, the API is quite simple. It sees the data within the Administrator as belonging to either a primary or secondary repository. A primary repository is data that is metadata that is stored within a data warehouse environment. A secondary repository is metadata stored at a data warehouse level. Metadata within a primary repository cannot reference data within a secondary repository (since by definition the environment is independent of the warehouse). Only one secondary repository can be active at a time, therefore the API cannot directly move metadata between them. This is a very important concept to the API since before using data within the Administrator, it is first necessary to attach to a repository (make it active).

The API does not support complete write access to the data within the Administrator. The documentation for the API[vi] covers thoroughly what can and cannot be updated. The API does not allow write access to processes defined within the Administrator, but most elements can be updated.

The way that the API works is for a link to be established to the repository (either primary or secondary) that is to be updated. Once this link is made, then there are a series of *methods* that can be utilized to perform the extraction of or the reading of metadata. Depending on the element that is being accessed, the parameters passed to the methods will vary. For example, a commonly used method will be *_add_metadata_* which, as its name suggests will allow for the addition or update metadata. Depending upon what is being updated, the parameters passed to this method will vary. For example if a *detail table* were being updated (by definition to a secondary repository), then different information will be needed by the method than if a libref were being added (which has to be added to the primary repository).

The parameters that are passed to the available methods are usually done so in the form of SCL lists. Results from methods (e.g. the *_get-_metadata* method) will also be passed through SCL lists. For this reason, it is essential that the developer be expert in the understanding and manipulation of these SCL lists. What is common to most methods however, is that there will be ID, NAME and DESC properties associated with every method call. This is because in the Administrator, these three are the key identifiers for each of the elements.

To use the API, it is therefore necessary to have an understanding of the methods that are available. These methods are quite simple in their construct and there are only fifteen of them. However, more complexity comes in the form of understanding *Metadata Types*. These represent all of the 'objects' within the Administrator that can be accessed through the API. For example, a metadata type could be *WHHOST*, which represents the host definitions (that must be in the primary repository). Another example is *WHGRPODD*, which is the metadata type for ODD Groups. These metadata types are important because to use the metadata within the Administrator, it is necessary to first know exactly what is being accessed (is it a detail table, or is it a summary table).

There is a four stage process for using the API:

- Know what you want to do … then select the method.
- Know what you want to do it on … then select the metadata type.
- Know what is required by the method for that particular metadata type … and then construct the SCL list containing that information.
- Then run the method.

Examples

The examples below relate directly to the data that is contained within the warehouse extension illustrated in Tables 1-6. Much of the code for these examples is in line with those in the documentation[vii] so that it can be used by anyone that has access to the Administrator.

Attaching to a repository

Whenever the API is used, it is always necessary to attach to at least a primary repository. The SCL code below illustrates attaching to the sample data warehouse that is shipped with the Administrator and will therefore work with at any site that licenses the product.

```
/*************************************
Instantiate the Metadata API Class. This will make the methods required to read or write to the Administrator metadata object available. Note that there are three common properties to all metadata objects: ID, NAME and DESC. These will be contained in all metadata property lists. Note the importance of understanding the structure and use of SCL lists
*************************************/
i_api=instance(loadclass
                ('sashelp.metaapi.metaapi.class'));
/*****************************************
Access the Administrator Sample primary metadata repository. This is in essence the Demo Warehouse shipped with the Administrator.
*****************************************/
path="!sasroot\whouse\dwdemo\_master";

/*****************************************
Put in the physical location of the metadata object we will be referencing. repos_type refers to the type of object, in this case it is the Warehouse Environment.
*************************************/
repos_type="WHDWENV";
/*************************************
Create a metadata list and insert the location information above into that list. Put it in the list as a named item called LIBRARY. The metadata list l_inmeta will contain a series of sublists.
*****************************************/
  l_inmeta=makelist();
  l_lib=makelist();
  l_inmeta=insertl(l_inmeta,l_lib,-1,'LIBRARY');
  /*********************************
Create a list called l_path and insert it into the metadata master list called l_inmeta.
  *************************************/
  l_lib=insertc(l_lib,' ',-1,'ENGINE');
  l_path=makelist();
  l_lib=insertl(l_lib,l_path,-1,'PATH');

  l_opts=makelist();
  l_lib=insertl(l_lib,l_opts,-1,'OPTIONS');
  l_path=insertc(l_path,path,-1);
  /*********************************
At this point we have a list that contains a sublist that references the LIBRARY. This sublist contains two others, called ENGINE, PATH and OPTIONS. The only values in the list at this stage is the one for PATH that was set earlier. This list will be used as a parameter to the _set_primary_repository_ method. It will then have performed its task and can be deleted.

The next stage is to set the primary repository. Remember that it is always necessary in using the API to attach to a repository.
  *************************************/
call send(i_api,'_set_primary_repository_',
```

```
        l_rc,l_inmeta,repos_type,
        primary_repos_id,l_meta);

/*************************************
The primary repository will now be
set(ACTIVE). The primary repository (an
environment) will be linked to secondary
repositories (Warehouses). It is now possible to
find these secondary repositories.
*************************************/
l_reps=makelist();
l_meta=setniteml(l_meta,l_reps,
                 'REPOSITORIES');
call send(i_api,'_get_metadata_',l_rc,l_meta);
/*************************************
Primary repositories are connected to secondary
repositories. Primary repositories cannot contain
any references to secondary repositories since
they are independent of them. Information about
the secondary repositor will be contained in the
list l_reps (which is a sublist of l_meta). Get the
first of the secondary repositories and set it
ACTIVE.
*************************************/
 l_sec_rep=getiteml(l_reps,1);
 call send(i_api,'_set_secondary_repository_',
          l_rc,l_sec_rep,sec_repos_id);
```

This code above illustrates how to make active both a primary or secondary repository. Code, similar to the above will be needed every time the Administrator API is used. Although it might seem complex to begin with, looking at each of the lists created whilst running the code helps make sense of the code.

Reading Metadata using the API

```
/*************************************
Now we are attached to a primary repository
(an environment in Administrator parlance) we
can now extract information from it. The
example below will create a list of ODDs called
l_odd_tables. Note the use of the metatype
WHODDTBL.
*************************************/
 of_type=primary_repos_id || . || 'WHODDTBL';
 l_tables=makelist();
 call send(i_api,'_get_metadata_objects_',
          l_rc,of_type,l_odd_tables);
```

This is a very simple example. What was needed was the primary repository id and knowledge of the metadata type required to read ODD information (in this case WHODDTBL). With these two pieces of information, the *_get_metadata_objects_* method could be run. In this case, the SCL list l_odd_tables looks like this:

```
A0000001.WHODDTBL.A000001M='Expense
    Register'
A0000001.WHODDTBL.A000002G='Installations
A0000001.WHODDTBL.A0000037='Products'
A0000001.WHODDTBL.A000003J='Sample Remote
    Data'
A0000001.WHODDTBL.A000003Z='Services'
A0000001.WHODDTBL.A000004F='Software Sales'
A0000001.WHODDTBL.A000005Z='Remote
    COBOL program'
A0000001.WHODDTBL.A000008K='Sales People by
    Region'
A0000001.WHODDTBL.A000008T='ODD'
```

As expected, this complies exactly to the ODDs listed if one were to use the Administrator interface.

The next example of reading data from the Administrator uses the secondary repository already made active in the code above. Again, the code is fairly simple, given that the secondary repository id and the metadata type are known. The same method is used as in the previous example, *_get_metadata_objects_*.

```
/*************************************
Now there is an active secondary repository,
using the _get_metadata_objects_ method, the
detail tables within that repository ascertained.
These will be in the list l_det_tables.
*************************************/
of_type=sec_repos_id || 'WHDETAIL',
        l_tables=makelist();
call send(i_api,'_get_metadata_objects_',l_rc,
         of_type,l_det_tables);
```

In the example above, the l_det_tables will contain a list of detail tables defined within the active secondary repository.

Writing Metadata using the API

The next two examples illustrate how the API can be used to update the Administrator's metadata. The first example is the easiest to follow, but the latter example is more interesting since it uses the data created in the extension to the Administrator earlier.

The first example adds a libref to the metadata. The libref must be added to the primary

repository, since this is where the environmental settings are made.

```
/*************************************
Assign a new library to the metadata: This will
be at the environmental level (primary
repository).
*************************************/

rc=clearlist(l_meta,'Y');
lib_id=scan(primary_repos_id,1,'.');
new_type=lib_id || '.WHLIBRY';
l_meta=insertc(l_meta,new_type,-1,'ID');

l_meta=insertc(l_meta,'SUGI',-1,'LIBREF');

l_meta=insertc(l_meta,'LIBREF FOR SUGI24',
                -1,'NAME');

l_path=makelist();
l_path=insertc(l_path,'d:\sugi24',-1);
l_meta=insertl(l_meta,l_path,-1,'PATH');
call send(i_api,'_add_metadata_',l_rc,l_meta);
```

The example begins with the l_meta SCL list being cleared. This list is the one that will contain all the parameters that must be passed to the method to add the metadata, given the correct metadata type.

The SCL variable *new_type* will contain the identifier for the primary repository and the reference WHLIBRY which is the base metadata type for SAS libraries. This metadata type requires certain parameters to be passed to the *_add_metadata_* method. These include a sublist within l_meta named ID that has the value of *new_type*, a name for the library (in this case SUGI), and a description (LIBREF FOR SUGI24). Optionally a path has been specified. Note that the path is contained not as an item in *l_meta* but as a sublist. This allows for concatenated libraries to be assigned within the Administrator. The WHLIBRY metadata type will allow for any parameter to be passed to the method that will reflect the setting up of a libref within the Administrator itself.

Once the SCL list *l_meta* is as complete as we need it to be, it can then be passed to the *_add_metadata_* method and a new libref will be added. This example will work for anyone who has the Administrator licensed.

The final example will be to actually use some of the data we have created in the extension to the Administrator. This example will create a new ODD. An ODD is defined at the primary repository level and will be contained within an ODD group. This means that we must know the ODD group under which the ODD will be created.

```
/*************************************
Add a new ODD table An ODD is added to an
environment, not a warehouse. Therefore it will
be added to the primary repository, not the
secondary repository.

First, obtain information about the ODD groups.
We will want to add the new ODD within a
specific group. We need to know the group to
add the ODD
*************************************/

of_type=primary_repos_id || . || 'WHGRPODD';
        l_odd_groups=makelist();
call send(i_api,'_get_metadata_objects_',l_rc,
        of_type,l_odd_groups);

/*************************************
Select the group. In this case, we know that we
want the ODD in the existing Sales and
Marketing Sources ODD Group.
*************************************/
g_loc=searchc(l_odd_groups,
            'Sales and Marketing Sources');
group_id=nameitem(l_odd_groups,g_loc);
rc=clearlist(l_meta,'Y');
l_groups=makelist();
l_group=makelist();

/*************************************
The ID for the new type must be found. To
know this, we must use the prefix of the group
within which we want the ODD to reside. In this
case, we have found the group id (see above)
which is contained in the group_id SCL variable.
We use the scan function to find the prefix and
then attach the metadata type of WHODDTBL
(Warehouse ODD table) to it.
*************************************/

l_groups=insertl(l_groups,l_group,-1);
l_group=insertc(l_group,group_id,-1,'ID');
l_meta=insertl(l_meta,l_groups,-1,'GROUP');
repos_id=scan(group_id,1,'.');
new_type=repos_id || '.WHODDTBL';
l_meta=insertc(l_meta,new_type,-1,'ID');
```

```
/*************************************
The description of the table will be placed in the
l_meta list with the name NAME.  This is the
name by which the table is referenced within the
Administrator, not the name of a specific data
file.
*************************************/

l_meta=insertc(l_meta,'POS SALES',
                -1,'NAME');

/*************************************
Other information is then put in the l_meta list,
including the description and the name of an icon
 *************************************/

 l_meta=insertc(l_meta,
                'POS SALES source data (API)',
                -1,'DESC');

l_meta=insertc(l_meta,
         'SASHELP.I0808.ADEVENTS.IMAGE',
         -1,'ICON');
/*************************************
 A sublist of l_meta is then created called l_cols
and is named COLUMNS.  This sublist will
contain other sublists that will have information
about each of the columns to be added to the
ODD table.  To get the information from the
columns, we use the sugi.dw_dt_el (Table 4)
table we created in the extension.
*************************************/

l_cols=makelist();
l_meta=insertl(l_meta,l_cols,-1,'COLUMNS');
col_id=repos_id || '.' || 'WHCOLUMN';


wherec='dt_table="pos_sales"';
meta=open("sugi.dw_dt_el(where=(" || wherec
            ||"))");
call set(meta);

  do while(fetch(meta)=0);
   l_col=makelist();
   l_col=insertc(l_col,col_id,-1,'ID');
   l_col=insertc(l_col,dt_col,-1,'NAME');
   l_col=insertc(l_col,dt_desc,-1,'DESC');
   l_col=insertc(l_col,dt_type,-1,'TYPE');
   l_col=insertn(l_col,dt_len,-1,'LENGTH');
   l_cols=insertl(l_cols,l_col,-1);
  end;
  rc=close(meta);

  call send(i_api,'_add_metadata_',l_rc,l_meta);
```

This snippet of code is more complex than the previous examples, but is easy to follow, especially if the *l_meta* list is printed out:

```
( GROUP=( (
ID='A0000001.WHGRPODD.A000008I'
         )[933]
        )[913]
  ID='A0000001.WHODDTBL'
  NAME='POS_SALES'
  DESC='POS SALES source data (API)'

ICON='SASHELP.I0808.ADEVENTS.IMAGE'
  COLUMNS=( ( ID='A0000001.WHCOLUMN'
            NAME='sls_amt'
            DESC='actual_sales_$'
            TYPE='N'
            LENGTH=8
           )[929]
          ( ID='A0000001.WHCOLUMN'
            NAME='rsls_amt'
            DESC='retail_sales_$'
            TYPE='N'
            LENGTH=8
           )[927]
          ( ID='A0000001.WHCOLUMN'
            NAME='sku'
            DESC='SKU'
            TYPE='C'
            LENGTH=5
           )[925]
          ( ID='A0000001.WHCOLUMN'
            NAME='sls_vol'
            DESC='actual_sales_#'
            TYPE='N'
            LENGTH=8
           )[923]
          ( ID='A0000001.WHCOLUMN'
            NAME='sale_tm'
            DESC='sale_tm'
            TYPE='N'
            LENGTH=8
           )[921]
         )[931]
 )[887]
```

This list is then passed to the *_add_metadata_* method which will then create the ODD table.

Summary

Very often it is easy to 'lose the plot' when it comes to solving business problems using technology. One of the reasons for this is because we become so involved in the

technology itself that the reasons we are using it become distorted or lost. It often helps to take a step back and question the purpose of using new technology. The Metadata API is one of those technologies that really requires a 'big picture' view. We know it exists, but when and why would we want to use it.

This paper has covered the reasons why it can be beneficial to use the Metadata API and has given some examples as to the techniques required to use it successfully. There are, however, some business benefits to building the extension to the Administrator that have not be outlined above:

- If every source field referenced within the data warehouse were to be referenced within the extension, then it would follow that there would be little, or no, redundancy within the data warehouse. This is because all of the data fields would be there for a specific and explicit reason. It would be possible to cross-reference every input (source) field into the data warehouse with a specific enterprise need. It follows, therefore, that if we can then map every input field to every detail table and summary table (as the Administrator can do), then we complete the full circle.
- Using the extension, it will be possible to know what the data warehouse is **not** doing. Only by knowing what the data warehouse is missing, based upon the vision (scope) is it possible to determine how successful it is[viii].
- We know that our code will work even if the underlying structure of the Administrator metadata changes.
- We are able to track the owner of every business need, business element and data element at an atomic level. In other words, there can be an owner of a specific business need, or an owner of a particular input column. This is especially useful from a business perspective because ownership at a table level might not make sense.

The extension example given in this paper could be extended a great deal. More information about the input fields could be generated and certainly far more could be integrated with the Administrator.

The key to using this technology is to fully understand what it can actually achieve from a business perspective before committing to its use. There has to be a good reason to be using the data from the Administrator outside of its interface before the resources are used to implement the Metadata API. This reason might well be the dissemination of metadata, through the Internet, SAS applications, or other applications, but as we have illustrated above, it could also be to improve the way the data warehouse responds to the needs of the enterprise.

---

[i] See Chapter 2, Peter R. Welbrock, *Strategic Data Warehousing Principles Using SAS® Software*, Cary, NC: SAS Institute 1998. 384pp.
[ii] See Chapter 6, *Strategic Data Warehousing Principles Using SAS® Software*.
[iii] See Peter Welbrock:*Is Your Data Warehouse Successful? Proceedings of the Twenty_Third Annual SAS® Users Group International Conference*
[iv] see (iii)
[v] See Lisa Horwitz *Harnessing the Power of SCL Lists, Proceedings of the Twenty_Third Annual SAS® Users Group International Conference*
[vi] See SAS/Warehouse Administrator™ Metadata API Reference, Release 1.3, First Edition, Cary NC: SAS Institute Inc., 1998.
[vii] See (vi).
[viii] See (iii)

Address:

Any questions regarding this paper will be willingly answered. Please contact:

Peter Welbrock
Strategic Information Systems, Inc.
223 Hampden Avenue
Narberth, PA 19072

Tel: 610 668 6417
e-mail: welbrock@erols.com