Paper 117-2010

The Power of User Defined SAS® Code in SAS® Data Integration Studio

Anders Holte, EnterCard International

ABSTRACT

Extensive use of standard SAS® Data Integration Studio Transformations is often made when designing data warehouses, thereby reducing maintenance requirements and the need for user defined code. This paper instead advocates using the SAS Data Integration Studio API against user defined code. Thus using the full power of SAS to build and maintain warehouse processes may yield several advantages, however an effort is required to design modules. EnterCard uses a layer of user developed, object oriented code in a framework for development and run time batches. SAS macros are designed to maximise reusability and be called from Data Integration Studio, heavy clients or SAS® Enterprise Guide®. The SAS Data Integration Studio layer is "thin" with less metadata, allowing for maintenance of separate modules, with version control of code by third party tools. The framework services are discussed and SAS Data Integration Studio examples shown. The framework runs in multiple warehouses serving our Swedish, Norwegian and Danish offices. Our approach has afforded flexible and effective development (even by junior developers), secure execution and homogeneous processes.

INTRODUCTION

The warehouse development and run time batches at EnterCard are based on a heavy coding that has historically emerged from user developed code. The code a valuable source contains large amount of business logic that has to be kept.

The first warehouses were built before the new multi tier architecture of SAS came along. The use of effective coding in SAS was inherent, making it easy to continue that when SAS Data Integration Studio was introduced.

The value of a user interface like SAS® Data Integration Studio should not be underestimated, it provides a "live" documentation on the original processes that creates and deploys jobs, ensuring it is never out of date as opposed to other documents written after the processes are built. It is often said by SAS developers that one must make the most out of SAS Data Integration Studio by using standard SAS transforms when available, if possible build processes only by the graphical user interface. That is often regarded as the most effective in maintenance, and when operated by less experienced SAS programmers.

But too extensive use of standard SAS Data Integration Studio transforms may not be good for performance, and also sometimes yields overcomplicated processes. Taking control of the processes by user transforms therefore has the potential to make simpler processes that are easier to maintain and yield better performance, and we can leverage the full power of SAS in batches. Through the API offered in SAS Data Integration Studio we can do this without actually losing any important features.

This Paper presents key features of our framework and some examples of the use of such transforms. Well designed, structured libraries of code together with a "thin layer" of SAS Data Integration Studio processes might offer a range of advantages in organizations like ours. The paper shows our approach in this area. It also shows some services that are offered by our framework, and how it is implemented in Data Integration Studio. The core framework is designed and developed by the author.

OUR DWH ENVIRONMENT

EnterCard is a Scandinavian credit card company. We have warehouse environments supporting the organization in Sweden, Norway and Denmark. We have a batch oriented environment with a lot of user written code with business logic controlling warehouse processes and file exchange to partners. We have started implementing DI Server and BI server environments in SAS 9.2. Figure 1 shows an overview of our environment, in fact a template folder structure from which we develop the SAS Data Integration Studio interface.

We have at the moment three regional warehouses with a fairly straight forward 4 layer design. Normal warehouse architectural principles govern how we design the environment. The DWH levels are:

- A staging layer containing recently read data where logic regarding technical reading of data is applied.
- A Primary layer for historical data, where no business logic is applied.
- A Business layer where business logic is applied (joining tables, calculating business elements)
- A functional Data Mart Layer where business areas have their specially prepared detailed and aggregated data.

Importantly in the scope of this paper we have several external source code libraries. We have the **BATCH** folder for entries that drive the data flows and which are application specific according to business logic. Further we have the common **business layer** with macros that contain business rules, and the general **apptools** folder for near general source codes, and finally the **TOOLS** folder for the fully generic routines.

In the Data Integration Studio folder structure we have our Applications folder (Figure 1). It has a generic part belonging to the FRAMEWORK which is the main topic of this paper. It contains a TOOLS part documenting generic routines. Further it contains the user transforms. We have the most frequently used in the process editor, expanded in Figure 11. Some of them are shown throughout figures and examples. They are organized in sub

folders to make them easier to find. They are also found under the Transformations tab (see also Figure 2, the left part). In the supplementary user transforms folder more user transforms are found.

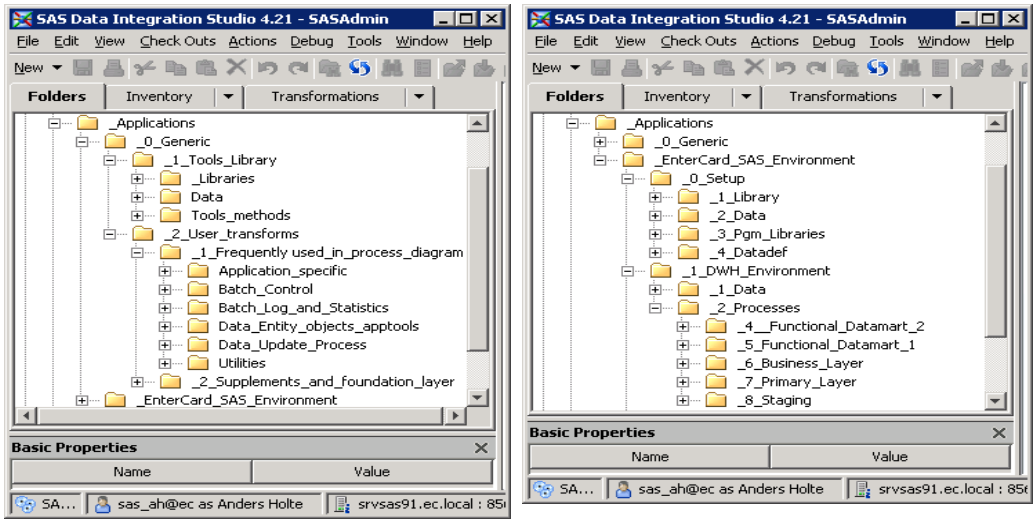

Figure 1. Structure of our environment with the generic part to the left and the setup and DWH part to the right.

The framework is applied in the areas of the EnterCard SAS Environment (Figure 1). Within one such area we have a SETUP part which serves the applications. It contains tables and views, definitions for session startup, global parameters, autocall libraries and data libraries. The latter typically are imported from physical data libraries, thus also covered in SAS Data Integration Studio. Further we have the DWH environment with warehouse data and processes for DWH layers and file exchange routines. We can have a common setup for the applications areas, or separate setups.

CONCEPTS

STANDARD TRANSFORMS

When building processes most often a mix of standard transforms and user written code is applied (Figure 2). Standard transforms are limited and may yield inefficient processing and sometimes overcomplicated processes. We often need to take control by our own code. Data steps performing joins enables good control and flexible data manipulations, and are often preferable to SQL joins because of that. Also, the full detail of all data flows do not need to surface in the process diagrams as long as the inputs and outputs of permanent tables are covered. Putting more data handling into user transforms would accomplish that if wanted.

We also want to take control because of performance. At one site in Norway they built a process with SAS standard transforms using six separate SQL joins to add columns to a large table. Six views were generated when running with real data, leading to a 150 GB utility file, and the job was not able to finish. By instead using a user written transform using SAS Formats and a data step view the execution time was reduced to 10 minutes(*). This is an extreme example, but shows the need to check the performance of standard transforms.

But if we use a “User Written Code” standard transform (Figure 2), we get code in principle applicable only in that job, not directly enabling reusing the job step. Therefore the preferable way would be to make a user transform by the TRANSFORMATION GENERATOR.

USER TRANSFORMS

SAS® Data Integration Studio has come with an interface to build user transforms (Figure 3). They are created easily from the TRANSFORMATION GENERATOR dialog that produces objects that can be dragged into a process diagram of a job. We have exploited this by designing a number of objects that are built as user transforms, and which CALL a **separate** macro. We call them “User defined API Objects” (see Figures 4,6,7). These thus form a clear cut interface. The calling layer is in Data Integration Studio, the underlying objects are called with parameters, built as a macro stored in an external autocall library. An autocall library is where we store SAS macros in a search path, the macros being compiled and run when first called, a mature SAS feature in batch systems. The autocall environment is started in either a workspace server (SAS Data Integration Studio or Enterprise Guide) or a server session from a heavy client. By separating out the sources we enable using these in server processes either operated from a workspace server in SAS Data Integration Studio or SAS Enterprise Guide or possibly SAS stored processes

(*) Example made available by independent consultant Per Sondrup Nielsen (psondrni@online.no).

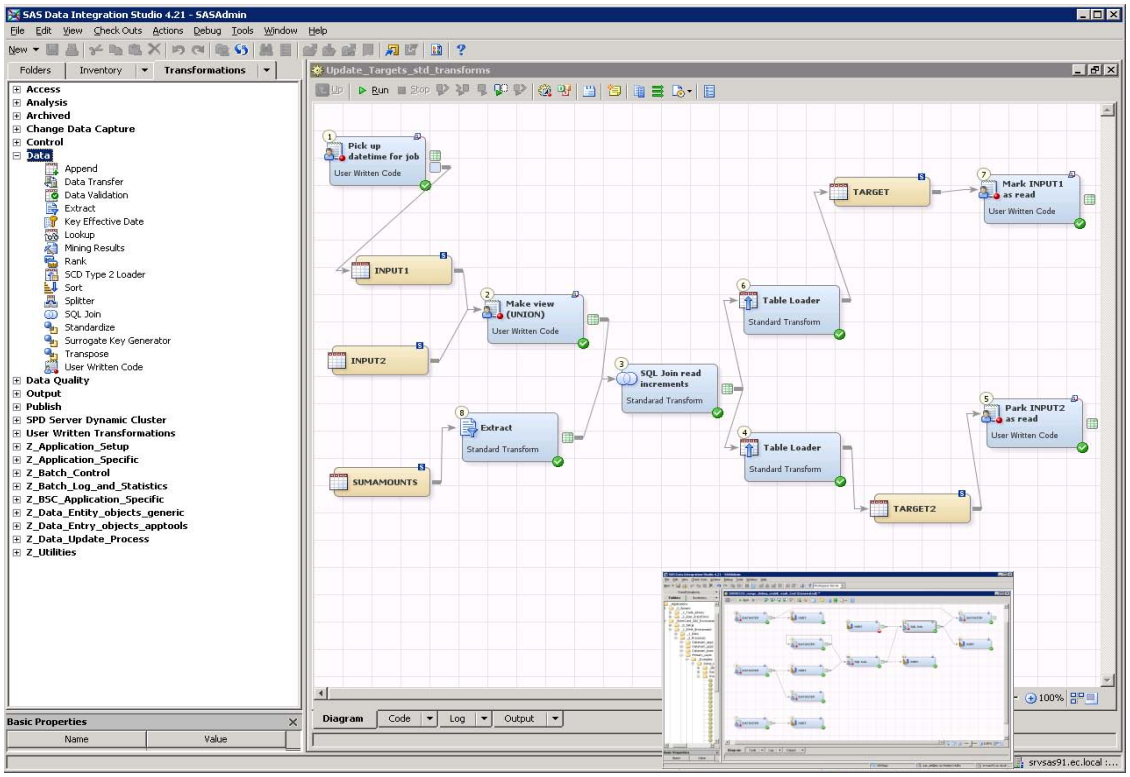

Figure 2. A rather simple job with three data sources and two output tables. At the start (node 1) we pick up the current datetime (user written). Typically we start out using standard SQL Joins, but have to put in our own code at some locations. We use standard table loaders. After updating the target tables successfully we have to mark records read in the two main sources (user written). Typically jobs might get more complicated (inset).

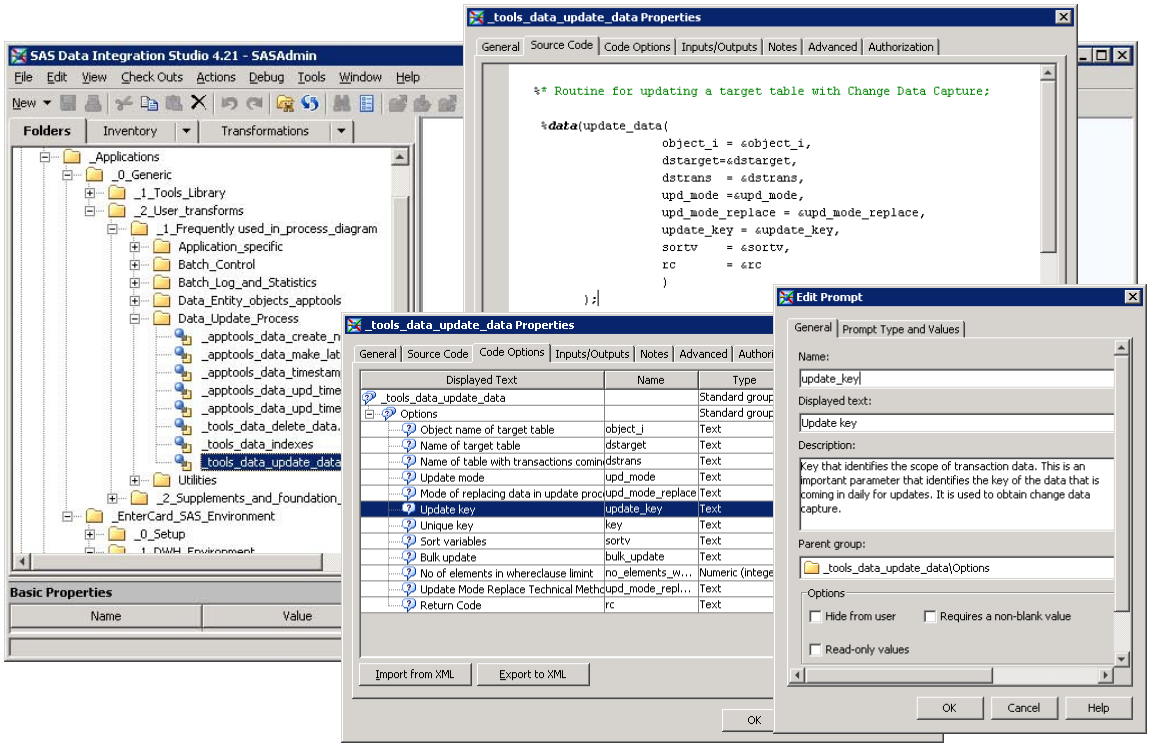

Figure 3. Example of a user transform "_tools_data_update_data" built with the transformation generator, with its SOURCE, and CODE OPTIONS with edit screens. Note: A SEPARATE macro is called. Data integration Studio offers a number of features to ease programming: Each code option has a name, and they appear as macro variables in the source code (see the source code pane), also &_input and &_output macro variables are available for each input and output table (see examples, point 4).

Let's define a "**User defined API OBJECT"** (Figure 4) as being a SAS Data Integration Studio user transform with corresponding underlying autocall macro, an "**API MACRO**".

User defined API Object examples would be:

- API OBJECT (a SAS Data Integration Studio user transform): _tools_setup_runPgm1 with source code that calls a corresponding API macro which in turn in this case calls another program named in an option (Figures 6,7, 8,9)
  - The API MACRO: An autocall macro named setup_runPgm1 that will run that program
- API OBJECT (a SAS Data Integration Studio user transform): _tools_data_update_data with source code that call a corresponding API macro, and with options which will be parameters in a macro call (Figures 3,6,7,8)
  - The API MACRO: An autocall macro named data_update_data with parameters controlling the update process.

The API Objects introduces somewhat more complexity at first sight, but offers a range of advantages:

- Clear cut interface between the SAS Data Integration Studio layer (graphical, having all that is needed) and the object below, making the SAS Data Integration Studio layer more static and transferring more maintenance to code in underlying objects/macros. We still keep the important features of SAS Data Integration Studio with column tracing and impact analysis.
- We can use any of the underlying objects/macros in batch code, making it possible to use **identical** objects in SAS Enterprise Guide process flows or batches developed and run directly on server sessions from heavy clients. This makes it very flexible and transparent. The setups of the environments in either are identical. When developing we work both from heavy clients and SAS Data Integration Studio (or possibly from SAS Enterprise Guide).
- The underlying source code may be treated under professional version control from for instance Subversion.

Disadvantages would be:

- It requires discipline in design in development.
- Although it is easy to get started, it requires preferably a modular approach that needs to be elaborated on to get it right, requiring persons that have at least some development experience.

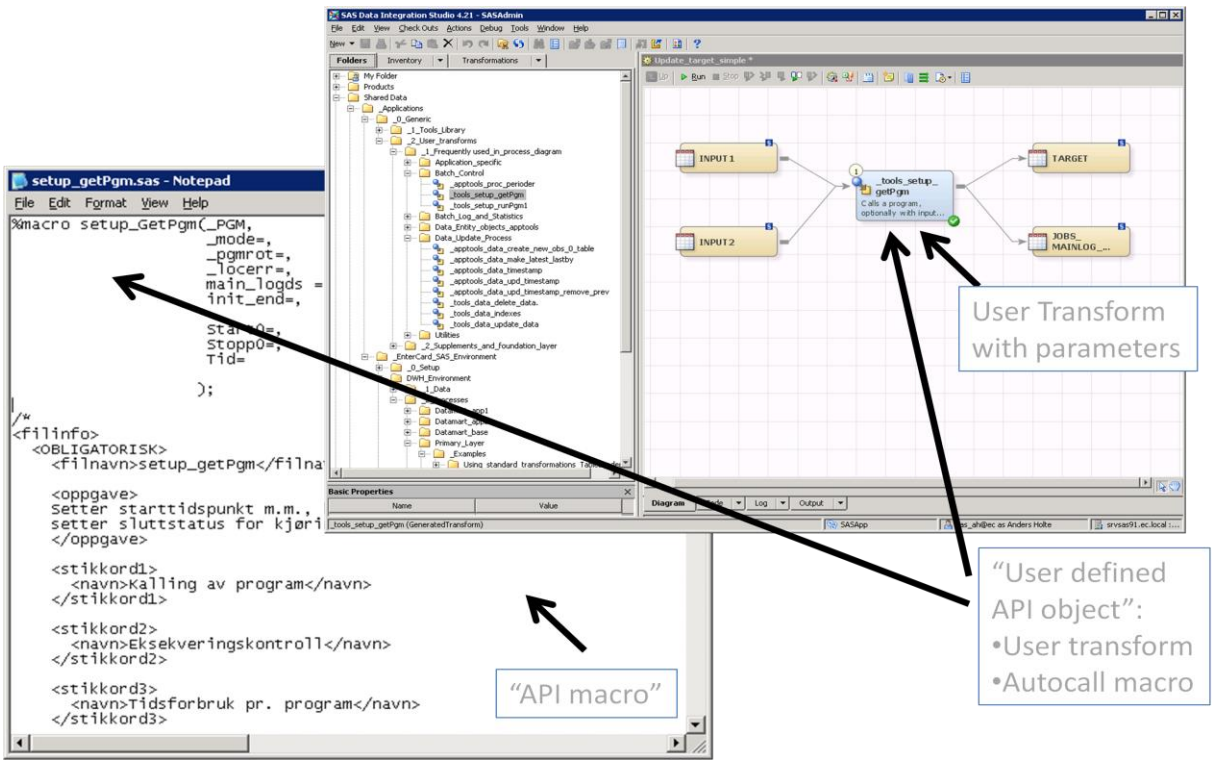

Figure 4. What we call a user defined "API Object". It consists of a user transform created as a transformation generator object with parameters, which calls an external source program (API macro) which is an autocall macro.

MODULARIZATION – REUSABILITY – OBJECT ORIENTATION

We encourage modularization and reuse of objects whenever possible. We have developed a rather extensive library of objects/modules offering services for batch jobs .They are organized in a hierarchy from business specific rules down to generic modules (Figure 5), with high level objects and low level objects, the latter most flexible. We work according to a **framework** with maximum reusability in mind. One specific operation with its piece of code should never be stored at separate locations, nor developed twice.

An example of a business logic module is the calculation of turnover according to specific rules different for Norway and Sweden (Figure 5). The module is implemented as macros containing elements needed to define and calculate turnover.

The actual framework consists of server side application setup with the SETUP API Objects, data handling services with the DATA API Objects, and various utility routines for operations performed repeatedly with the UTILITY API Objects. It is accessible in the generic part in the folder tree (Figure 1). The objects in Figure 2 do not belong to the framework. The objects in Figures 3,4,6,7,8,9,10 do.

The services offered by the framework enable the developer to concentrate more on the application specific business logic. Such modules are not part of the framework, they use the framework. They also are designed with a modular approach, and most often stored as server side autocall macros in “batch” folders accessible both for batches and Workspace servers.

The framework is designed with object orientation in mind to minimize the need for parameter transfer. We have applied to a certain extent principles of objects implemented as macros with “attributes” and “methods”. The API macros encapsulate an operation, i.e. business logic or common technical element. The macro in itself has features that encapsulate attributes (macro variables) that are local to that macro. That is important to maintain control. Further the “methods” are what operations the macro can do, typically controlled by parameters.

We have pure technical oriented framework API Objects, and more business oriented API Objects (currently under development).

DATA API OBJECTS (Figure 11): Some of the macros are automatically created by SAS code, for example the DATA ENTITY OBJECTS. These are a special kind of API objects defined one for each table, typically TARGET TABLES. They are generated by the framework if we decide to use them for a particular table. By using an extensive set of API objects we maintain data entity object information in setup tables and notably in data entity macros (Figure 10). The latter are important API objects that are used in data update processes in our own “table loader”, DATA_UPDATE_DATA (examples points 6) and 7)), which itself is an API object. It may use the data entity object to get the necessary update parameters. The data entity objects contain an attributes “instance variables” part, and a “methods” part. The attributes contain the table name, library, sortedby columns and others. They also contain update key and update mode (Figure 10) which control the update process, like the standard SAS Data Integration Studio Table Loader. The methods part contain such things as the length definitions of all columns in the table, formats and informats for columns, variables keep list of the table, and variable labels list (examples points 3) and 5)).

We thus have a generic API object that performs table updates and another that contain update and dictionary info. We have really reusable objects to use in any process whenever we need info on a table (examples points 3) and 5)). One main advantage comes by the fact that the data entity objects are stored OUTSIDE the SAS Data Integration Studio transforms, they do NOT reside inside SAS metadata, and we do not need to click our way through all jobs if needed to modify parameters. But many API objects are designed so that we can OVERRIDE default behavior. Notably, running an update using the data update API object, we can by default use the info from a data entity object to control it all, or we can override specific parameters. Also, if we do not choose to use data entity objects, we can still run the data update API object by giving all the parameters.

UTILITY API OBJECTS (Figure 11): These are where we may have the broadest benefit by defining all the bits and pieces into utility API objects that does tasks that we do over and over again, making it easier for us, and doing it more securely. These are examples of API objects that enables flexibility and promotes reusability. General objects that do just one thing are often the most valuable and may be used in more situations. Figure 11 and the following text lists examples.

SETUP API OPBJECTS: These are almost never used by developers. They are running during startup of a session, either in a SAS Data Integration Studio session or an SAS Enterprise Guide session when the workspace server is started, or in a server session. The setup would be identical in all those sessions. The setup API Objects are rather extensive, they control the autocall search path, global macro variables, and librefs for data libraries (also defined in SAS Management Console or Data Integration Studio). They also control several other things like redirecting the SAS log and maintaining run time statistics in the control data library.

Modularization benefits would be:

- Increase efficiency in maintenance. Since code doing the same do not reside in different places, in the long run it is easier to maintain control.
- Developing may be easier and more secure. Lately SAS Data Integration Studio has got features for having processes feed into other processes, thus potentially enabling creation of common processes. By adding the concept of API macros underneath those we may add extra power and flexibility.
- Increase process performance meaning taking control by using our own code
- Make it possible to choose execution environment by designing objects and modules that can run identically

in a server or workspace server session (Figure 12). This might be especially important for business logic modules.

- For instance with the data entity and data update API Object it introduces the possibility of having senior warehouse staff implement update process handling according to data and process models. This is a crucial job. Others may then concentrate more on implementing the application specific business rules, getting the increments right, and then send it for update. It may enable increased flexibility. Users with different skills and experience levels can do a good job. What remains for developers to do is
    - Implement application specific programs, typically batch folder macros that are called from either other macros or from API Objects.
    - Support in deciding update keys .
    - The update process can then in principle be rendered, it will work.
- The job of the administrator and warehouse responsible would be
    - Designing data models and process models
    - Define update key according to the structure of data and what identifies the rows of daily incoming data.

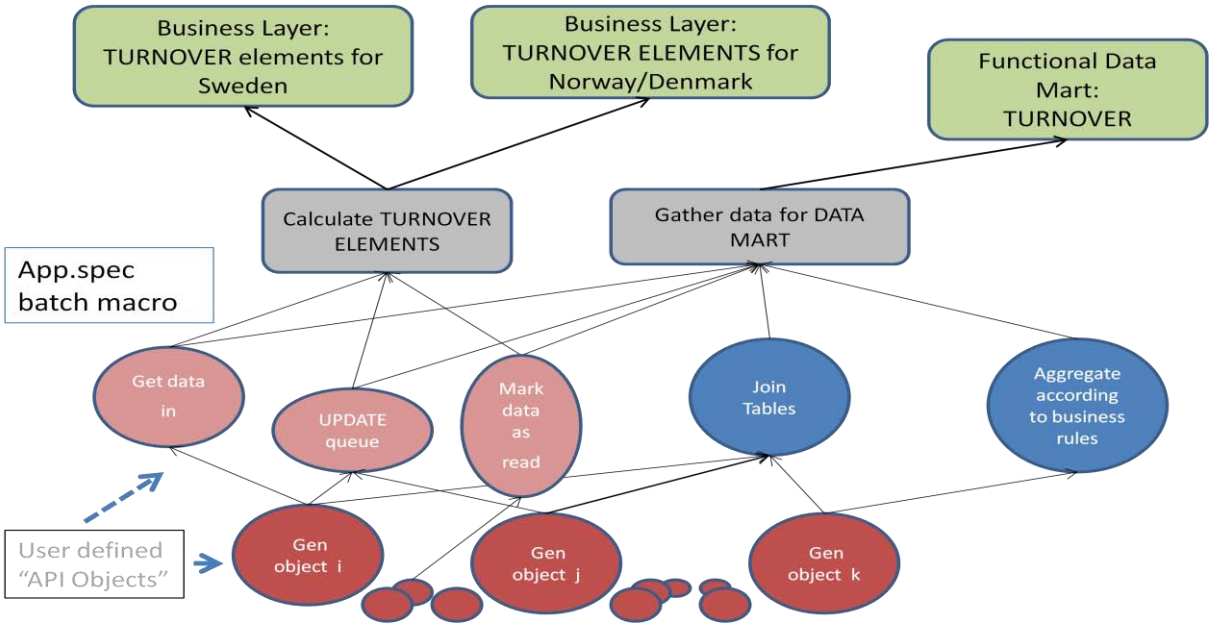

Figure 5. Schematically how modularization is implemented using turnover business handling as an example. We put together business elements and store it in the Business layer by reading data into a queue, and mark the read records after successful operation. Then we gather data for the Functional data mart by joining and aggregating data according to business rules. The modularization comes into action by using reusable "user defined API Objects" (pink) used by several processes. The reusable modules in turn use generic objects (red). The table joining and aggregation are application specific for this situation (blue), but they use generic objects to accomplish the task. The more generic, the more reusable and useful the modules are.

EXAMPLES

Having available such objects as mentioned in this paper enables us to do a number of operations within a job step. We might want to do "small" job steps, or job steps that do more in one operation.

In Figure 6 we accomplish the same as in the job in Figure 2, but using our API Objects. The same is also accomplished in Figure 7. The difference between the two is that we do all in one step in Figure 7, while we expand the graphical flow more in Figure 6. The two are included here to illustrate that we might choose to simplify the SAS Data Integration Studio process and do more of the job underneath. The API objects may be dragged into the process editor individually (Figure 6), or their API macros may be called from another source (Figure 7). Also, Figure 8 illustrates an important point that it often might be convenient to mix both standard SAS transforms and user defined API objects within a job.

The following example job steps illustrate more in detail what is going on. Note that all example steps (points 1) through 9)) would run from the **Figure 7** job. On the other hand, in **Figure 6**, point 1) will run under the first API Object, then point 2 the whole step) will run under one API Object (marked "2"), then points 6) and 7) will run to update target tables, and finally points 8) and 9) will run to update timestamps.

The two input data sources contain amounts for main regions, and sub regions, and also have columns for date, a read timestamp, and a timestamp called IN_PL for when the record was loaded to the target table. The SUMAMOUNT table contains among other columns regional amounts and group amount category. It does not have distinct records per region, the region amounts are all repeated for each sub region.

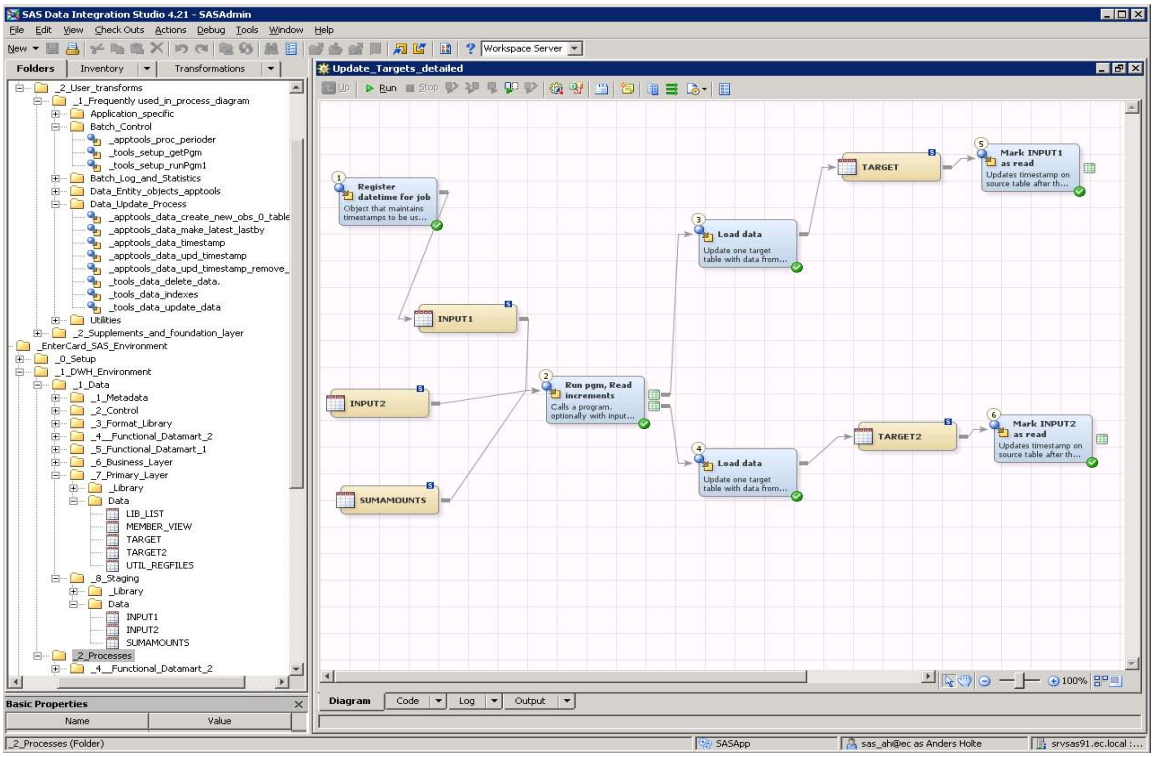

Figure 6. The same job as in Figure 2. A rather simple job with three data sources and two output tables. At the start we pick up the current datetime using an API Object (see the last two steps). We read the records not read before from INPUT1 and INPUT2 by the SETUP_RUNPGM1 API Object. We also have SUMAMOUNTS coming in with aggregated values to be joined onto the union (set in data step) of the two other input tables. We then update TARGET and TARGET2 by the same API Object which is our own table loader, (the DATA_UPDATE_DATA API Object). The updates are incremental. After updating the target tables successfully we have to mark records that are successfully read in INPUT1 and INPUT2. We use the timestamp value from the first step, marking those not read previously with the timestamp. This is done for both tables with an API object. More details are given in the examples.

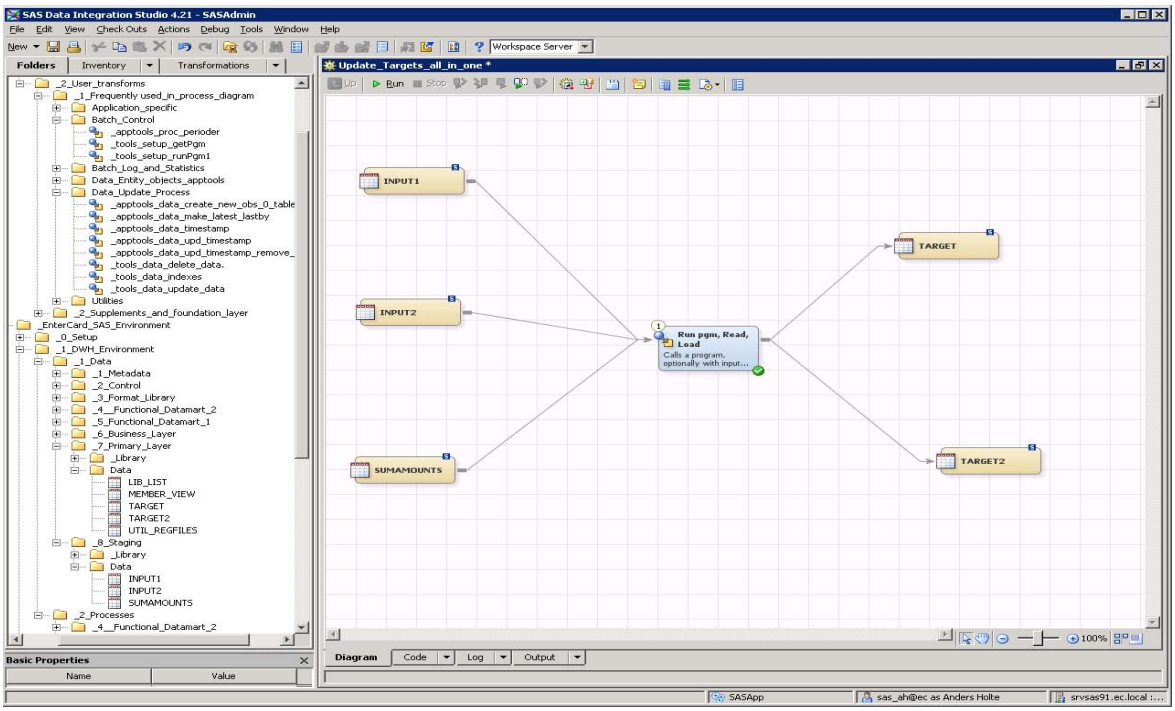

Figure 7. A job that does the same as in Figure 6, but all in one step. The details of the job are all accomplished within one program that are called from the SETUP_RUNPGM1 API Object. Since the API macros are all transparent and available, we can pick up the time, update target tables, and mark records successfully read from one API Object. More details are given in the examples.

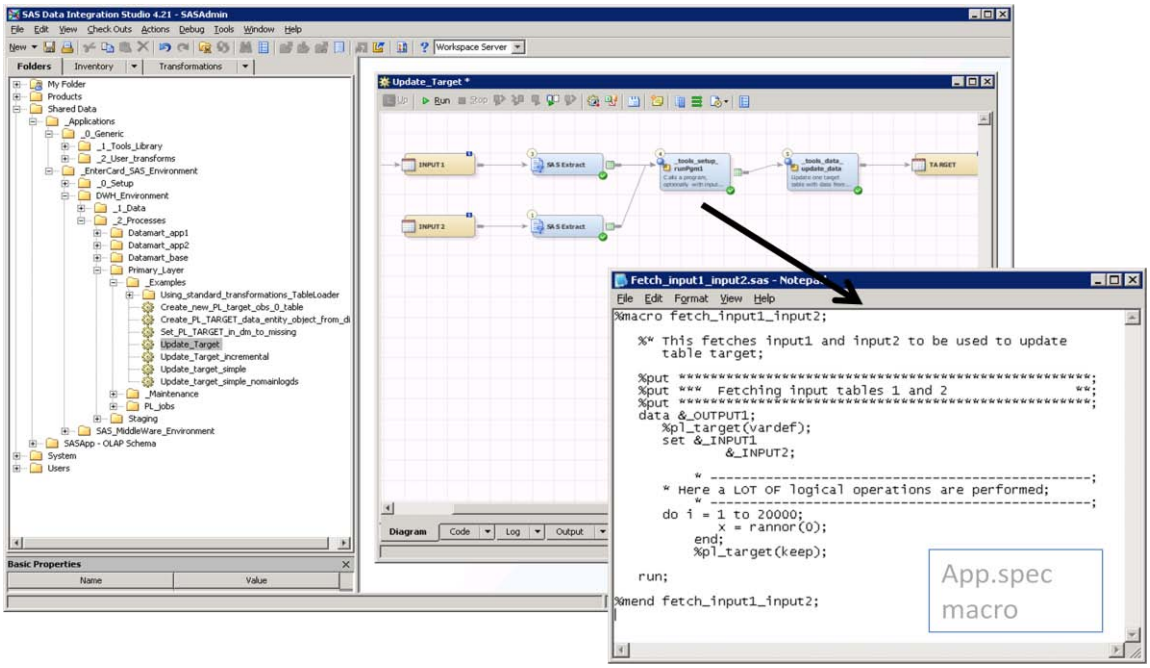

Figure 8. Another example job. We extract data from the two input tables INPUT1 and INPUT2 with the standard SAS transform EXTRACT. Then we put the data from the tow together by the API object SETUP_RUNPGM1 that calls a BATCH application specific program called fetch_input1_input2, which does a data step set and also some other operations. Then data are loaded into the TARGET table by the API Object DATA_UPDATE_DATA. This illustrates that we perfectly well can mix standard SAS transforms with API Objects in a job. More details are given in the examples.

In point 0) in Figure 9 we see the start point, a source code of a user transform that may call code given in the examples, see also Figure 3. Program elements are shown in the examples below. In point 1) the timestamp is picked up into timestampRead. We obtain the timestamp by running a very simple data step underneath.

```
1  %* Picks up current timestamp;
   %data(timestamp(get,var=read_timestamp));
   %put The timestampvalue is assigned to timestampRead;

   %if &locerr = 0 %then %do;

2    %* READS DATA, CREATES TEMPORARY OUTPUT TABLES;
       DATA _OUTPUT1(keep=%target(varkeeplist))
            _OUTPUT2(keep=%target2(varkeeplist));

         %target(vardef); %target2(vardef);   3

         %* JOINS IN region_amnt group_amount_category
            from SUMAOUNTS for each region;
         %* hash table hh definition comes here ...
            it is used because of
            it's prowerful features …..;

4        SET &_INPUT1(in=ina where=(in_pl = .))
             &_INPUT2(in=inb where=(in_pl = .));

5      FORMAT %target(formatlist); LABEL %target(labellist);
       FORMAT %target2(formatlist); LABEL %target2(labellist);
       * ------------------------------------------------;
       * Here a LOT OF logical operations might come;
       * ------------------------------------------------;
       if ina and hh.find() = 0 then OUTPUT _OUTPUT1;
       else OUTPUT _OUTPUT2;
     RUN;
     %let locerr = &syserr;

   %end;
```

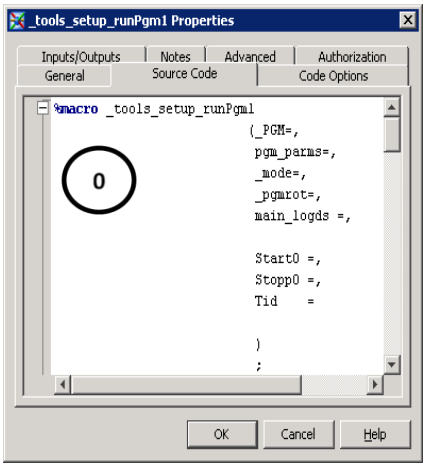

Figure 9. The source code tab of the API Object _tools_setup_runPgm1.

The whole of step 2) is what is called by SETUP_RUNPGM1 in Figure 6). It reads two output work tables from

two input tables, and also uses a hash table to join in two columns. It is used because of powerful features.

Point 3) illustrates a feature of DATA ENTITY OBJECTS: output table 1 and 2 keeps only the columns that are defined for the respective TARGET output tables by the “varkeeplist” feature. Also the “vardef” feature defines columns for output tables TARGET and TARGET2 by generating length statements.

Point 4) shows a set statement gathering data from the two input tables. Very importantly, the &_INPUT1 and &_INPUT2 tables are provided by Data Integration Studio, they contain the table names. &_OUTPUTn tables are also available, n being the output table number (Figure 8).

We use the “formatlist” feature of the two target tables to define formats for the columns, and labels for columns by the “labellist” feature (point 5).

```
%if &locerr = 0 %then %do;
   %data(update_data(
6            object_i = TARGET,
             dstrans=_INPUT1,
             rc=locerr)
          );
%end;

%if &locerr = 0 %then %do;
   %data(update_data(
7            object_i = TARGET2,
             dstrans=_INPUT2,
             rc=locerr)
          );
%end;
```

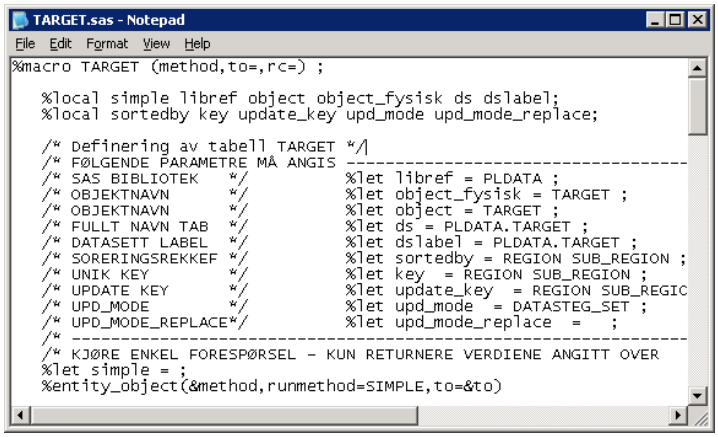

Figure10. The DATA ENTITY macro for TARGET.

The loading of the two target tables goes on in 6) and 7). We call the API macro DATA_UPDATE_DATA. Note that we only use two parameters beside the return code parameter. That is made possible by using the DATA ENTITY macro in Figure 10, which is automatically generated, and which we see has info on the key of the table and so on. Most importantly the update key and update mode are given. Together with the incoming transaction tables WORK._INPUT1 and WORK._INPUT2 from the previous step, these parameters are sufficient to perform the two updates. The updates are performed with change data capture since we use the API Object mentioned, which as that feature built into it.

At last we update the INPUT1 and INPUT2 (permanent) source tables by assigning the current timestamp from step 1) to the IN_PL timestamp variable in the two tables respectively (8), 9)). We use the DATA_UPD_TIMESTAMP API macro. We specify the source table, timestamp variable, the where clause to use when assigning timestamps, timestamp value, and return code.

```
%if &locerr = 0 %then %do;
   %data(upd_timestamp(source=INPUT1,
8                      Timestampvar = in_pl,
                       where_cl     = in_pl = .,
                       innlest      = &timestampRead,
                       rc           = locerr)  );
%end;
%if &locerr = 0 %then %do;
   %data(upd_timestamp(source=INPUT2,
9                      Timestampvar = in_pl,
                       where_cl     = in_pl = .,
                       innlest      = &timestampRead,
                       rc           = locerr)  );
%end;
```

THE MOST FREQUENTLY USED USER TRANSFORMS

Figure 11 shows the most frequently used user transforms that has been developed. There are more than twice as many in total so far. Not all these are usually needed in process diagrams. The others, and those under the _Supplements_and_Foundation_layer (Figure 1) are documented in the user transforms, but are mainly called from OTHER API Objects or other macros.

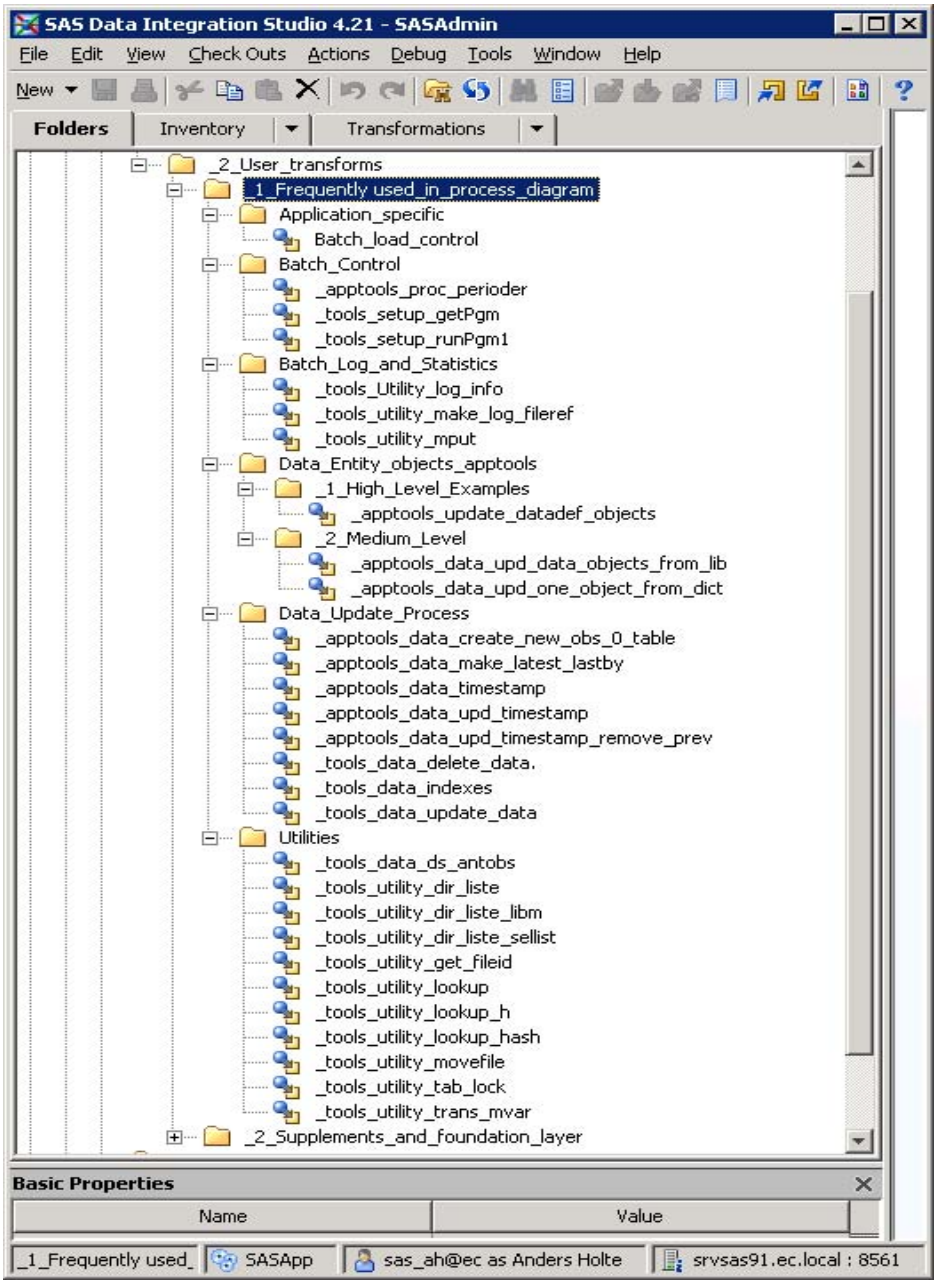

Figure 11. The most frequently used user transforms (API Objects) organized in folders in the folder tree.

Some **key** User Defined API Objects listed in Figure 11 are mentioned below:

Application_specific

- **Batch_Load_Control**: Objects to control loading, for instance reading or writing external files. It works with the input from a parameter table with the same name – Batch_load_control.

Batch_Control

- **_tools_setup_getPgm**: Call an arbitrary program and store run time statistics in the CONTROL library which is defined in the SETUP routines during startup of the session (Figure 4).
- **_tools_setup_runPgm1:** Call an arbitrary program without storing run time statistics (Figure 6,7,8,9).

Batch_Log_and_Statistics

- **_tools_utility_log_info:** Define underlying main and utility log infrastructure, perform write operations to logs and statistics tables. It also contains an object that defines the location of the SAS main log

The data entity part:. It contains routines at different levels, i.e. high level routines that call other lower level routines, and the lower level routines that contain more detailed handling.

- **_tools_data_upd_one_object_from_dict:** Define all components of data entity objects from dictionary

information for ONE table.

The Data_Update_process:

- **_tools_data_timestamp:** Maintains timestamps. Assigns the current timestamp to a macro variable for later use. Defines and optionally assigns timestamps for incremental loading (i.e. IN_PL, and others). (Figure 6)
- **_tools_data_upd_timestamp:** Sets a read timestamp column in records that are not previously read in a given table (Figure 6)
- **_tools_data_upd_timestamp_remove_prev:** Sets a read timestamp column in records that are not previously read in a given table, and at the same time remove records that was read last time.
- **_tools_data_delete_data:** Dele records in a table or delete table.
- **_tools_data_indexes:** Handle indexes in a table, either query, create or delete.
- **_tools_data_update_data:** Performs the update of a target table from an incoming transactions table by using change data capture algorithms, optimizing the process, inserting the sortedby data set option on target table for optimization, and so on- (Figure 3,6, 8).

Utilities are objects that perform various services (Continuously supplemented):

- **_tools_data_ds_antobs:** Find the number of observations in a table
- **_tools_utility_dir_liste:** Create a table or a view with a list of external files in a sub folder, optionally as a selection list
- **_tools_utility_dir_liste_libm:** Create a table with a list of members in a SAS library
- **_tools_utility_get_fileid:** Register external files in an index table and maintain this by returning the file id for already registered files or inserting new file registrations for new ones. This is used to store the file id in tables instead of the whole file name.
- **_tools_utility_movefile:** Move external files into another folder, creates the target folder if it does not exist.
- **_tools_utility_trans_mvar:** Transform a macro variable containing a number of elements by quoting elements, comma separate elements, or both.

CONCLUSION

The paper has tried to present the main part of our framework. It has been developed over several years. The reason for it was, and is, to help develop, maintain and robustly run jobs. The main architectural feature is the modularization and the division between the SAS Data Integration Studio logical layer and the working objects in the form of macros underneath (Figure 12). We have experienced that we very often need to use our own code because of complexity of tasks, to try to simplify the processes to a “suitable” abstraction level, or to control the performance of the jobs. We also want to easily reuse some of the business logic that we have had in user defined code.

What we have got is an environment with potentially flexible and powerful features. We very importantly are scalable in the sense that we can design processes with mixed SAS standard user transforms together with our own API Objects or API macros underneath. We have an environment which has identical setups on the server side for batches during the night, for development from heavy clients on the server, for Workspace servers working from SAS Data Integration Studio or SAS Enterprise Guide or possibly from stored processes (Figure 12).

The main advantage is that we have means to implement SAS Data Integration Studio with it’s main benefits, and at the same time design and keep flexible and effective routines. We may by this utilize the very powerful SAS data manipulation tools that otherwise may be unavailable by relying solely on standard transforms.

To successfully implement such an environment rather extensive work is required.

But SAS has made available a port to the underlying power of SAS through the User Transform Generator. It is easy to get started, and the features may be built over time.

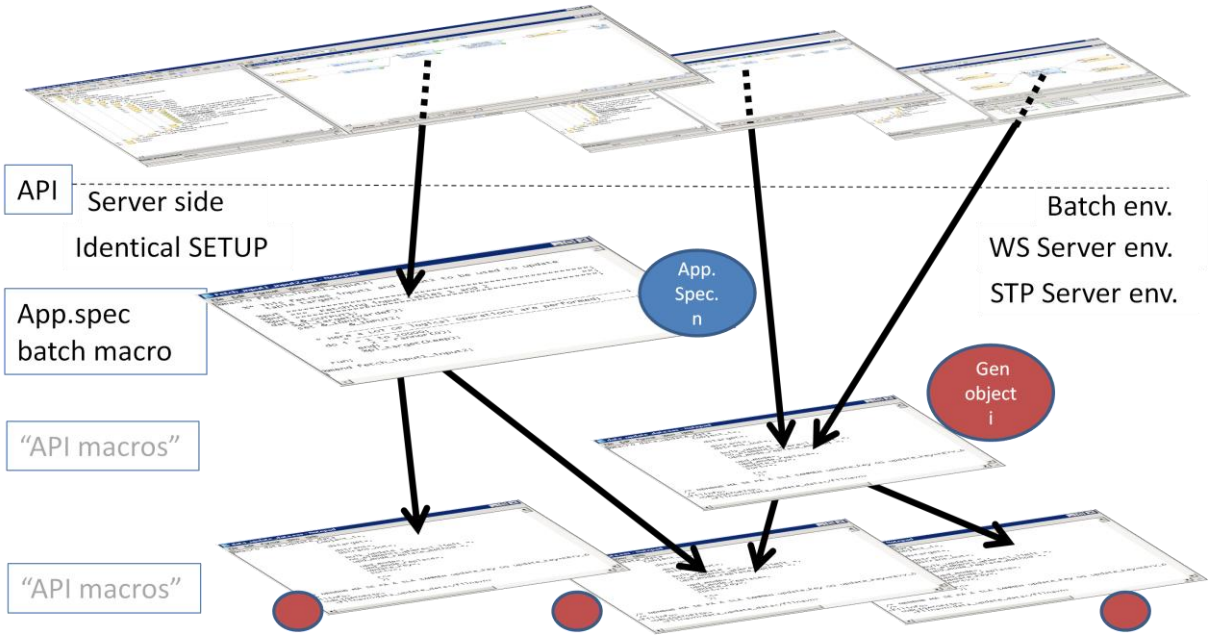

Figure12. To sum it up: Illustration of using the user defined API Objects discussed in this paper. Specific JOB STEPS might call application specific batch macros, which in turn may call general API macros. Or it might directly call API macros. They in turn may call one or several other API macros. By this we get an API from The logical SAS Data Integration Studio layer to the server side. The setup environments for both batch processing, Workspace server processing, or Stored processes are identical. We can maintain the server side environment with SAS Data Integration Studio or from a heavy servers side client, or possibly from Enterprise Guide. When it processes it all runs from deployed jobs which call autocall macros.

TRADEMARKS