# Paper 004-2010

# Creating Easily-Reusable and Extensible Processes: Code that Thinks for Itself

Faisal Dosani, RBC - Royal Bank of Canada, Toronto, ON, Canada
Lisa Eckler, Lisa Eckler Consulting Inc., Toronto, ON, Canada
Marje Fecht, Prowerk Consulting Ltd., Toronto, ON, Canada

## ABSTRACT

It's easy to write code that answers only one need. It's more challenging and time-consuming to develop a "hands-off" process that adapts to many needs. In the long run, time and effort is saved by building a modular process with wide applicability.

This paper investigates the implementation of a framework to help in building efficient and reusable code. We start by looking at design considerations prior to the start of coding, including identifying design patterns and utilizing Metadata driven logic. Then we consider effective ways to split logical sections of code into easily reusable components.

Examples will be presented including

- components of the planning process
- design framework
- key features of flexible code
- macros that enable creating hands off code with minimal intervention.

A basic understanding of SAS® and the SAS Macro Language is assumed throughout the paper, however the concepts may be beneficial to a wider audience.

## 1. INTRODUCTION

Every once in a while, instead of being asked to "just quickly build another one based on what we already have", we get asked to design and build something from scratch which is carefully planned for consistency, flexibility AND ease of operation and maintenance. Of course, the flexibility includes the ability to anticipate needs which haven't been defined or don't yet exist. This can be both an enviable opportunity and a daunting responsibility. The easy route would be to follow the precarious path of a quick and dirty project implementation, but maybe now is the time to make the extra effort and save the headaches later.

Have you considered the following?

- the similarities of this project to other work you have completed, including the commonalities in coding and processes
- the ongoing maintenance issues that spaghetti code and processes would create
- the organizational or reporting hierarchies this process needs to serve.

If you try to focus on the bigger picture when projects and requests come across your desk, you can create flexible and extensible solutions that

- avoid maintainability issues
- enable "speed to market" of results
- build reusable code and processes that will benefit you and your team.

This paper explores the planning and development efforts involved in a real-life information delivery project. The intended result of the project was the creation of a framework consisting of **easily reusable** and **easily extensible** code and processes. The goal of the framework was a multi-layered and multi-component approach that would adapt to the delivery of information (data and reports including a large number of metrics and dimensions) for 3 possible components:

- *enterprise* wide
- *division* specific
- *project* specific.

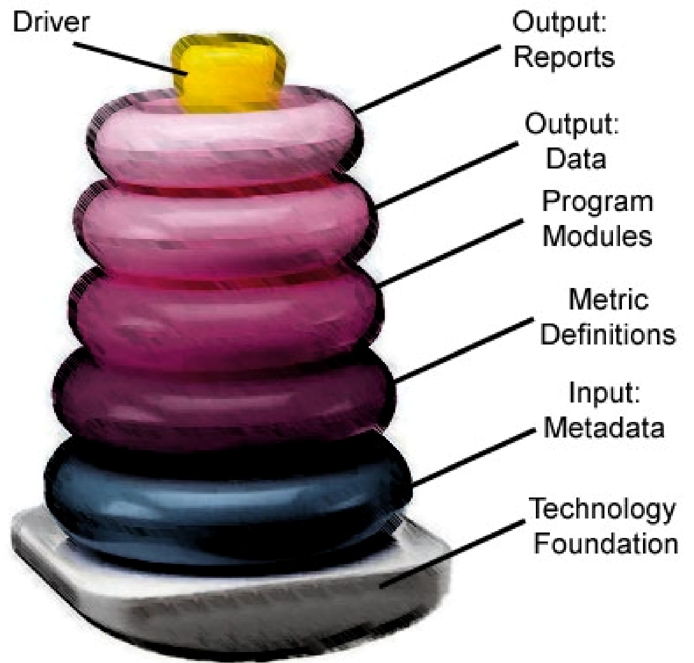
An important requirement was the ability to deliver information for any of the 3 components either in unison or alone (eg: if requirements for the *project*-specific component were not yet available, we would still deliver the *enterprise* and *division* information).

The development process was segmented into layers that adapt to the specific project or task. The layers include:

- technology foundation
- metadata (information about the specific task / project)
- data element definitions (eg: dimensions, metrics, etc)
- program modules
- output data (eg: reference tables and analytics data)
- output reports.

A driver program (see diagram) forms the core which holds together the relevant components and project-specific parameters.

Obviously, this approach only works with proper planning and discipline. That is the first area we will focus on.




## 2. PLANNING AND REQUIREMENTS GATHERING

Best practices dictate that before any code is written, we should investigate the needs, wants, wishes, goals and priorities of our business partners (stakeholders and key users). Ideally, both of these groups will be represented on the team. Gathering input from these people should be a first step. The perspectives of both the stakeholders and key users are essential to capture the big picture as well as the details. Documenting and feeding this information back to the team will help build consensus and maintain a record of the principles which will determine both the early decisions and those which may need to be revisited in subsequent phases.

The principles may require that every result or product contain some common or consistent elements, some divisional elements and some customized elements. We need to establish

- which elements must always be included at the enterprise level. This may be a high-level decision related to overall priorities of the *enterprise*. The *enterprise* component may support a consistent look and feel for the results or may provide uniform contextual or relationship data.
- which elements may vary by *division* or be optional
- which will always be customized for a particular *project*.

Defining a hierarchy of components at this stage will aid in identifying which are required and which are optional. The process must flow smoothly whether or not the optional components are present. This will support a modular design and will also help to break the project into manageable sub-tasks for detailed requirement gathering, technical design, development, testing and delivery. Once a hierarchy is agreed upon, tasks for various components can proceed independently.

Let's consider some general information delivery scenarios across different types of businesses:

- *Marketing Example*: In a Marketing organization, we may always include some profile data about the client and the overall relationship, some data about a divisional relationship to that client, and then the specifics of what has changed during a particular marketing campaign which we can attribute to that campaign.

- *Sales Example*: In a Sales organization, we may always include some overall annual corporate sales targets, some data about one division's sales over time, and then the current year's sales performance by one particular team.

- *Manufacturing Example*: In a Manufacturing organization, we may always begin with some context on the overall relationship with a supplier and then look at what they are supplying to one particular division or physical location, and then the specifics of cost and quality of one item supplied.

These are very different environments but the information delivery requirements have some commonalities. Each follows a similar hierarchical structure where we go from contextual or relationship data which is always included to divisional or product-related data to customized, specific data. Part of the planning will involve distinguishing the consistent from the distinctive elements:

- What will always be included, either at the *enterprise* level or the *division* level?
- What aspects must always be customized for each *division* or *project*?
- Are some of the customizations simply filters which could be controlled by metadata or parameters or are they truly unique requirements?

Even as design and development proceed, additional opportunities to streamline the customization may emerge as patterns become evident.

### Key Users

Identifying the intended recipients or users of each type of result at this early stage is helpful because it may expose details on what information should or should not be included, depending on the audience and possible sensitivity of the data. For example, is it appropriate for every viewer of a departmental sales report to see what the overall results of the *enterprise* are? This will support decisions about how, when and where results are to be delivered.

### Minimum requirements

If the typical situation is to always include a standardized *enterprise* component and select one *division* component and one customized *project* component, we will want to determine whether we should reasonably provide partial results: if the component for a specific *project* doesn't apply or hasn't yet been defined, will just the *enterprise* component and a *division* provide meaningful information? If so, we'll want to be able to deliver the results of *enterprise* and *division* components and optionally insert a *project*-specific component later. The stakeholders should be able to anticipate whether we need to design for component combinations like *enterprise* and *project* in the absence of *division* or *enterprise*, *division* and a combination of two different *project* components. It may take many discussions and consideration of very detailed scenarios to establish the rules around the **minimally acceptable inputs and results** before we can understand what design constraints are appropriate.

### Controlling the process

Can we anticipate what changes to the overall requirements are likely to occur in the future? If so, this should influence the design to maximize flexibility and ease maintainability. If we're anticipating re-using the same processes in many ways, we must also consider the process timing and frequency. If the data are only available after some date or event, how can we detect when that has been reached? The ideal here would be to capture some metadata which dictates when the process is valid for a particular *project*. Again, the stakeholders and key users can help identify what metadata would be meaningful to control the process so that we can prepare to capture it. For example, we may have an external table which allows us to identify what all the *division*s of the *enterprise* are. That table can provide control information as to what divisions require reporting on a monthly basis. We may have processes that run conditionally, dependent on some event, so we need to consider how to recognize that event to allow for as much automation as possible.

### Results – What formats are required?

As noted in the **Key Users** section, it is imperative that the requirements' gathering includes a focus on the expected results that will be produced. Likely, the key users represent a cross-section perhaps including

- business users
- analysts
- project planners.

Each of these users will have different needs, which should be identified early so that the design and expected inputs can accommodate the results.

You should consider the wealth of options including

- analytical datasets (SAS format or something else)
- summary reports and their required format
  - Excel – static or dynamic
  - PowerPoint results
  - PDF documents
- statistical reports.

Once the necessary results are identified, you can then begin exploring the toolsets needed, such as ODS, Microsoft Office Add-Ins, OLAP tools, BI tools, etc.

### Validation

Part of the exploration of high-level requirements should include a discussion of how we can deliver and validate our results. How will we know when we get it right? **Delivering intermediate products for review and user acceptance during the development will help to structure the project and build respect and acceptance.** At this stage we can plan for who will review and provide sign-off on each of the deliverables.

The early discussions with the business partners should provide an understanding of their goals and priorities and what the essential and the flexible or optional components are. After these discussions have been completed, documented and a common understanding of the requirements is agreed on, process design can begin.

## 3. DESIGN

### 3a. Process and documentation

We are now at the stage where we need to look into the design and documentation of our framework. Our design should fulfill the needs of our clients today and in the future. This line of thinking falls into the realm of **design patterns**.

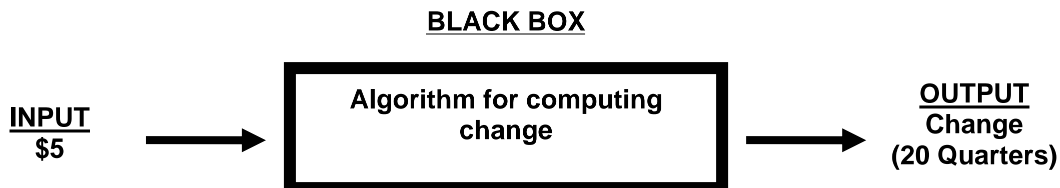
**What is a design pattern?**

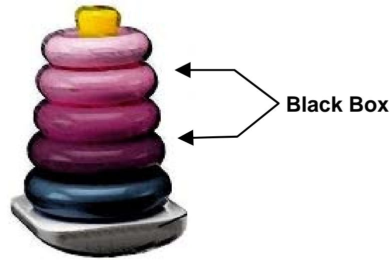
- a blueprint (methodology) for solving problems in a variety of situations
- describes the process flow for a framework and how different components interact with each other
- patterns should be reusable and not a one off implementation.

A simple example might consist of extracting data, transforming this data, and finally delivering this data in its intended format (Extract, Transform and Load, or ETL). This delivery can be a report, loading it to another system or some sort of user interface. What is important is the pattern describes the flow and high level components which the framework supports. This ETL example can be considered a pattern in that it helps define a structure which is reusable and common. Breaking a framework into such components helps modularize and simplify the processing.

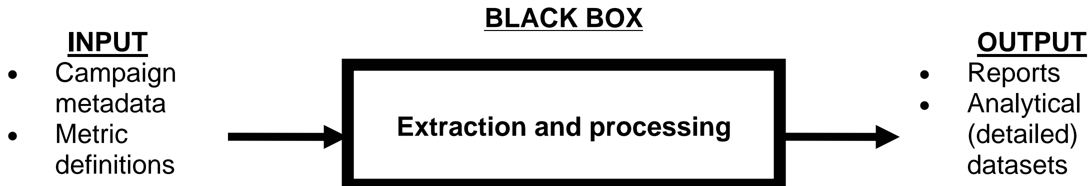
One of the keys to creating a re-useable framework will be to separate the logic into compartments. Think of these compartments as little black boxes. They take something in (metadata, metric definitions) and give you something back in return (data, reports). The user of the black box doesn't need to know anything about the inner workings, all they care about is what they input and what they get back in return. If you can break down the majority of programming into this kind of thinking you can create a plethora of reusable, robust and portable code.

Let's put the design pattern into some real world context such as a change machine. The change machine concept is pretty simple. We put currency into the slot and a few seconds later we get change from the dispenser. Most people will not know how the inner workings of the change machine work exactly, they just know if they put in a five dollar bill they will have twenty quarters returned. This is the beauty of modules. They shield the users from internal complexities so the users can concentrate on what is important, like getting change. You can even break the process down further into sub-modules within the bigger context. Think of the coin dispensing mechanism as its own module as all it requires for input is the number of coins to dispense. This is within the bigger machine but you see common theme between them. They all take some sort of input, do some processing, and have some end result.



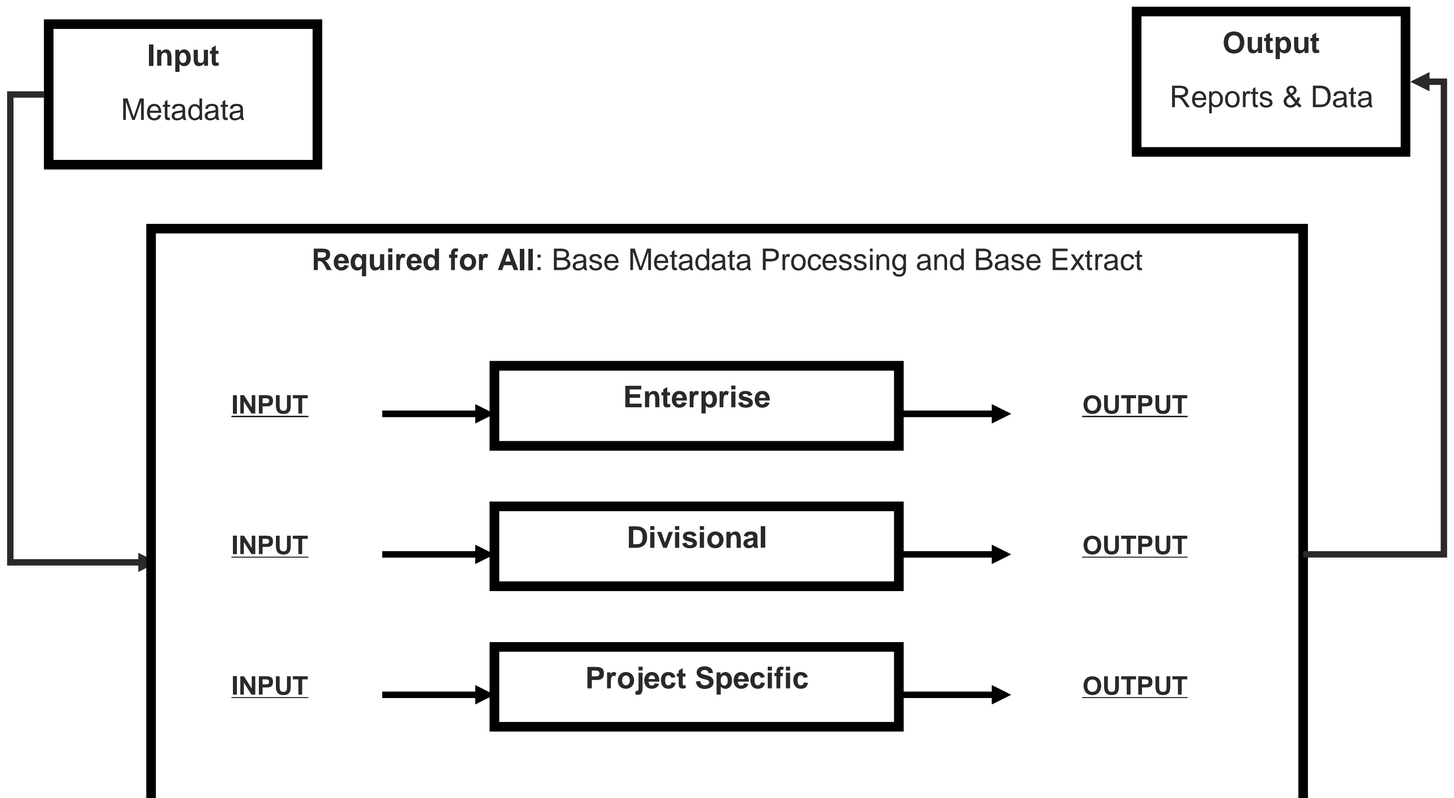



Within the context of a marketing campaign our top level design pattern might take on the following form:




Working with our business partners and through our planning phase we should gather the required input which feeds into the black box and returns the information we hope to deliver. The black box contains our processing and logic (example: defining the client profile and divisional information) and shields the business users from the inner complexities allowing them to concentrate on the things which are important to them such as the information they will be providing and receiving in return.

Taking this methodology a step further we can apply the black box not only to individual components but the entire process. This leads us to the notion of a super black box which takes in information and outputs information for the bigger framework. Within the super black box we have our base processing and mini black boxes which can represent our various components.



We've taken a big complicated problem and broken it down into smaller manageable problems, but more importantly have enabled the following key principles of our design:

- Separation of logic
- Reusability
- Shielding users from internal complexities.

## Documentation

Another equally important part of preparing our framework is properly documenting it so users fully understand the design which our framework will evolve into. Documentation can include such things as:

- data definition tables
- database table definitions
- flow diagrams
- reporting templates.

The documentation should also outline how our various components communicate with each other to ensure a symbiotic relationship throughout. This relationship should be thoroughly documented as we need to know how each of these components will interact with each other, like pieces of a jigsaw puzzle fitting together. Detailed information on expected input and expected output will be a key in facilitating the smooth interaction between such modules. This can be a specific structure of a dataset which will be passed along and accepted by each module or may even be a list of key expected fields which will be used for merging or matching data.

## Metadata

One key component which is sometimes forgotten is a control or metadata module. Such a module can help immensely in making our process totally hands off and further improve the possibility of making components more generic. Designing our framework to incorporate this from the beginning will not only save time now but in the future as metadata will drive which sections of the framework are called upon and when. It's like an air traffic controller, controlling scheduling and traffic flow and making decisions about de-icing, managing flights on the ground and in the air.

Below are some possible metadata examples by industry (not a complete list):

**Marketing example**

- Project identifier
- Project title / description
- Project start date
- Project end date
- Line of business indicator
- Test / control grouping indicators
- Success definitions
- Project costs
- Expected financial and relationship results

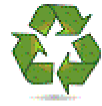
**Sales example**

- Project identifier
- Reporting start period
- Reporting end period
- Region identifier
- Target product identifier
- Base sales of product
- Advertising costs

**Manufacturing example**

- Project identifier
- Inventory start date
- Inventory end date
- Supplier
- Region indicator
- Facility Indicator.

Metadata will allow us immense flexibility as we develop and grow the framework. It will not only help **centralize conditional logic** but help with processing, scheduling, delivery of data, and automating such. The **project Identifier will be the key driving force** when relating metadata back to the framework as it will tie everything together. Each project will have a set of definitions within metadata describing certain cues or functionality which the framework will handle.



**What does this mean for the production team?** Instead of manually checking what gets run when or where the results need to be delivered, we can place some of those conditions in metadata, allowing us to **control the process via metadata** rather than manually altering the actual SAS code. This will cut down on quality assurance processing and development work in the future and offers much more flexibility.

Metadata helps to enable customizable modules without creating specific customized code. This is what makes it essential to any reusable and extensible project.

Much of what is described will be an iterative approach as scenarios change and new circumstances emerge. Building a multi-purpose framework to handle your business requirements may need several cycles of development before you can truly generalize the components to handle all the business needs anticipated by your business partners. Process improvements will become evident as components and layers are designed and built.

### Coding Conventions

It can't be stressed enough that in addition to documenting our design and framework it is just as important to document the coding modules we anticipate building. They should follow similar:

- naming conventions
  - Example:
    - divisional variable name - div_easternUs_numberOfClients
    - divisional dataset name - div_easternUs_Sales_Dataset

- storage locations
  - Example:
    - standard production code storage location name - /myprojectname/code/prod
    - standard production dataset location name - /myprojectname/data/prod
    - standard production log location name - /myprojectname/log/prod

- input / output styles such as consistent formatting.

## 3b. Code

The way we architect our framework will have a huge impact on the coding techniques and tools which will be utilized. Making sure a unified approach is used throughout the code will ensure consistency and take a lot of discipline. From our design documentation and planning we should have an idea of the different modules we will want to build, how they differ and how they will be similar if at all.

We want to separate logic and instructions where they differ and consolidate and generalize where they are similar. This will promote reusability, help us naturally break our framework into logical entities, and help with the manageability. For example, common *enterprise* elements and *division* elements can be different enough to separate into individual components. We can reuse top level elements as much as possible across our framework rather than grouping them with lower level elements which might constantly be changing as the *division* changes.

There are many possible levels of code and algorithm modularization. When determining which to employ, you should consider

- code complexity
- applicability for generalization
- ease of use.

We will explore two modularization techniques used in this project

- driver / source programs
- macro modules.

## Driver Programs / Source Programs

Consider one very common approach to program creation where you

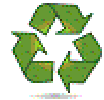
- locate an existing program similar to the functionality you require
- copy the code and store it as a new and unrelated program
- start making changes to the code to tailor it to the current task
- run and test the code as if it had never been used before.

This results in a lot of lengthy programs that ALL may require changes as the business rules and data change.

Now, consider the approach where you recognize that sections of your programs can easily be re-used by just supplying the information that changes via macro variables or control datasets. You proceed to break the code into those sections and store them as separate, callable modules. This approach is what we call **driver / source**.

- identify extensible code segments and store them as a single dated copy of each ***module*** (segment of source code)
- document the input that each ***module*** requires
- create a ***driver*** program that provides macro variable input for the current scenario and then calls the appropriate modules.

A repeatable process like this requires additional and extensive testing to confirm that changes to the source code modules accommodate all ***drivers*** that call the ***modules***. But the payoff is that when business rules change, there is only ONE source program to revise rather than a multitude. And, once the flexible process is set up, re-use of code is simple.



## Driver/Source example

Assuming that you have generalized your required extraction, summarization, and reporting processes into **source modules**, then when you are ready to start reporting for a new project, just create a **driver** with necessary parameter values as input to your source modules.

The below example **driver** program assumes that you have two generalized **source** programs that accept input and provide the extract, summarization, and reporting that is required. Assume the source programs are named

- `ProjectReportingExtract_mktg_2009_01_10.sas`
- `ProjectReportingOutput_mktg_2009_01_13.sas`

Contents of **driver program**

```
*** Driver Program - specify appropriate values for parameters;
%let type = mktg; *** Project Type (mktg, sales, usage, etc.);
%let prestart = 01OCT2009; *** Pre project period for comparison;
%let prestop = 31DEC2009; *** End of Pre project period;
%let poststart = 01JAN2010; *** start project tracking;
%let poststop = 30JUN2010; *** expire date - stop tracking;
%let title = January 2010 Introduction of Incentives;
%let products = ('VRS', 'GQL', 'BGO', 'DLA');
%let codes = ('015', '119', '214');
*** run standard extract and reporting programs;
%include 'ProjectReportingExtract_&type._2009_01_10.sas';
%include 'ProjectReportingOutput_&type._2009_01_13.sas';
```

## Improved Driver/Source example

The above approach works great, until your ***source*** code changes. Then, you need to find all ***drivers*** that are calling the source, and you need to change the date stamp in the `%include`. Not fun! To avoid changing the ***version*** in 100's of drivers, use a ***placeholder reference*** in your drivers so that the most current source program is called.

Additionally, create a program that references the latest source version:

Contents of **`ProjectReportingExtract_mktg_CurrentPgm.sas`**

```
*** call LATEST VERSION of standard extract program;
%include 'ProjectReportingExtract_&type._2009_01_10.sas';
```

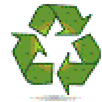
Revised contents of **driver program**

```
*** Driver Program - specify appropriate values for parameters;
%let type = mktg; *** Project Type (mktg, sales, usage, etc.);
%let prestart = 01OCT2009; *** Pre project period for comparison;
%let prestop = 31DEC2009; *** End of Pre project period;
%let poststart = 01JAN2010; *** start project tracking;
```

```
%let poststop = 30JUN2010; *** expire date - stop tracking;
%let title = January 2010 Introduction of Incentives;
%let products = ('VRS', 'GQL', 'BGO', 'DLA');
%let codes = ('015', '119', '214');
*** run standard extract and reporting programs;
%include 'ProjectReportingExtract_&type._CurrentPgm.sas';
%include 'ProjectReportingOutput_&type._CurrentPgm.sas';
```

Now that source modules have been created, and drivers are in place to run the process, we need to consider further task-specific modularization.

It should not come as a surprise that the SAS Macro language will likely be the choice for creating reusable modules within a framework. Macros have always been a great way to generalize by building code and passing parameters through a process to influence the results. We can write generic code that can handle several situations rather than repeating code over and over again. Macro tools will allow us to build the robust and versatile framework we desire.



Macro variables serve a similar purpose and are a component of the macro language. They can be used to feed information into code that only differs by certain parameters, and can be used for passing down instructions and conditions from metadata at a global level within our framework. This will allow our code to be very generic and mutate when we instruct it to do so. Using metadata coupled with macros will truly enable hands off processing, and through several iterations of code it will become obvious as to where these techniques can be implemented.

### Macro variable examples

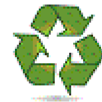
A neat trick to storing or not storing interim datasets is to use a macro variable as a flag. All datasets which you want to be able to store permanently or discard after your SAS session is over can be controlled by the INTERIMLIBRARY flag. If the variable is set to the value `interim` all datasets prefixed with `&INTERIMLIBRARY` will use the libname `interim` as defined. If the macro variable is set to `work` they will use the default `work` library and be discarded after your session ends. Practical use for this would be to store datasets during testing and once in production and stable you can use `work` to save space and resources.

```
%LET INTERIMLIBRARY=interim; /*work or interim */
libname interim "/mydirectory/sasdata";

data &INTERIMLIBRARY..myDataset;
/*..Code..*/
run;
```

Setting up storage locations is another excellent use for macro variables. Instead of sifting through all the code you have and changing locations in several SAS programs you can globally change a macro variable which will trickle down into all your code where you've referenced it.

In this example we are setting up locations to store data and logs. The base location will always be the same, but maybe for now we are just testing and not ready for production. For this we have the "stage" macro variable which we can alter when we are ready to make the move to production. This kind of technique helps minimize human error and maximizes the flexibility of the code.



```
%let dir      = /base_directory;
%let stage    = test; /*test or prod*/
libname data "&dir./data/&stage./subfolder1";
libname logs "&dir./logs/&stage./subfolder2";
```

### Macro examples

Earlier we discussed the use of design patterns to help streamline and create highly reuseable code. Let's take a look at such an example:

The macro `betweenList` uses a start date and end date as <u>input</u>, calculates all the *month-end dates* in between the two dates, and <u>outputs</u> the dates as a comma delimited list of quoted month-end dates. This can be helpful with trying to gain efficiencies in a SQL query where you are pulling time series data. Providing a list of dates instead of a *between-and* clause may improve the performance of the query. Further, this list could assist in parsing monthly sections of output or control extracts which need to be done one month at a time.

```
*** Macro betweenList;

*** Purpose: This macro will accept 2 dates formatted as "yyyy-mm-dd";
*** the first being a start date and second being an end date;
*** from these dates it will calculate all the month end dates;
*** in between and including these dates.;

*** Input: startMonth and endMonth dates in yymmdd10. format;
*** Output: Quoted string list of Month end dates to use in SQL query ;

%macro betweenList (startMonth , endMonth);
*** expects start and stop in yymmdd10. format;

%global me_dateList; *Month end date listing;

data _null_;
length me_dateList $9000;
start = input(&startMonth , yymmdd10.);
stop  = input(&endMonth , yymmdd10.);
current = start;

*Start of Do Loop;
do until (current ge stop);
   *place month and year into placeholder variables;
   *Calculate the current month end date;
   month=month(current); year=year(current);
   me_dt =  intnx('month' , current , 0, 'E');
   *if current month is less than or equal to the stop month then append;
   if current le stop then do;
      ** append to previous string, using a comma as delimiter;
      **if me_dateList is new then write for the first time else append ;
      if me_dateList=' ' then me_dateList = "'" || put(me_dt, yymmddd10.) || "'";
      else me_dateList = trim(me_dateList) || ", '" || put(me_dt, yymmddd10.) ||"'";
   end;
   *add one to the current month and loop through;
   current = intnx('month', current, 1);
end;
*End of Do Loop;

*Place the date list into our global macro variable;
call symput ('me_dateList' , me_dateList);
run;

%mend;

%betweenList ('2009-10-31','2010-03-31')
%put &me_dateList;
```

The result from running the code above is the list

```
'2009-10-31', '2009-11-30', '2009-12-31', '2010-01-31', '2010-02-28', '2010-03-31'
```

If you have a series of drivers that you run sequentially in production, you want to be sure to clean-up the environment when a new driver starts. Thus, we recommend a few final additions to the driver.

```
%macro cleanUp;
*****************************************************************;
***  clean up work datasets and all pre-existing macro variables;
*****************************************************************;

*** clean up the work directory;
proc datasets lib=work nolist nowarn nodetails kill;
quit;

*** build list of all macro variables;
```

```
*** which we want to clean up;
data _null_;
  length cmd $200;
  set sashelp.vmacro; /*Use the SAS help macro dictionary table*/
  where
    scope = 'GLOBAL' and  /*Select all GLOBAL Macro variables with an offset of zero*/
    offset = 0 and
    /*These are macro variables which we do not want to include in our cleanup*/
    /*Some of these might include system variables*/
    (name ne: 'SYSDB' and
     name ne: 'SYSODS' and
     /*User defined macros which we want to protect*/
     name not in ('TABLE_DATE', 'MIN_MONTH_END', 'LOAD_FLAG')
    )
  ;
  /*Cycle through the list of macro variables and delete the ones which have been
included*/
  cmd = '%nrstr(%symdel ' || trim(name) || ' / nowarn );';
  call execute(cmd);
run;

%mend cleanUp;
```

The *cleanup* macro can be included in our macro library code and called before each production run or in-between to ensure we are

- managing our work space efficiently
- cleaning up old or unnecessary macro variables within our session.

An example of usage might look like the following

```
*** include macro library and call cleanup;
%include 'ProjectReportingMacroLib_&type._CurrentPgm.sas';
%cleanUp;

*** Driver Program - specify appropriate values for parameters;
%let type = mktg; *** Project Type (mktg, sales, usage, etc.);
%let prestart = 01OCT2009; *** Pre project period for comparison;
%let prestop = 31DEC2009; *** End of Pre project period;
%let poststart = 01JAN2010; *** start project tracking;
%let poststop = 30JUN2010; *** expire date - stop tracking;
%let title = January 2010 Introduction of Incentives;
%let products = ('VRS', 'GQL', 'BGO', 'DLA');
%let codes = ('015', '119', '214');

*** run standard extract and reporting programs;
%include 'ProjectReportingExtract_&type._CurrentPgm.sas';
%include 'ProjectReportingOutput_&type._CurrentPgm.sas';
```

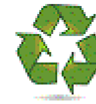
Macros can also be used to **control which process and logic gets executed**. Metadata coupled with macro logic can help execute code based on certain conditions. The code below will conditionally run a marketing initiative's success module only when the marketing initiative has come to the end of the response window.

```
/*
Code run during metadata processing
ProjectEndDate - from metadata and represents the End of the Marketing
                  Initiative Window
CurrentMonthEnd - generated with the current month end we are reporting for.
*/
data _null_;
set myLibname.metadata;
  if ProjectEndDate <= CurrentMonthEnd then do;
        call symput("ProjectWindowEnd",'Y');
  end;
  else  call symput("ProjectWindowEnd",'N');
```

```
run;

/*..........*/

/*Macro which drives logic in our driver program*/
%macro processSuccess;
%if &ProjectWindowEnd. = Y %then %do;
    %include "/myCode/abcCorp_marketing_success_module.sas";
%end;
%mend processSuccess;
%processSuccess;
```



These macros **enable flexibility by adapting the logic** to the current requirements.

## 3c. Managing Results

An important and often overlooked aspect of the planning process involves results management. *Results* may include:

- logs
- data sets
- reports
- excel spreadsheets
- validation files.

Effective **naming conventions** and **storage locations** help to insure easy retrieval of results and appropriate cross-referencing back to the code that generated the results. For example, consider the log resulting from running the program

```
projectID_extract1_20100215.sas
where projectID is a unique identifier for the project.
```

The associated log filename should include the SAS program name, for easy reference, along with the date time stamp corresponding to when the program ran.

```
projectID_extract1_20100215_yyyymmdd_hhmm.log
```

To automatically create the log file with each run of a program, use `PROC PRINTTO` to begin log writing at the top of the SAS program. We recommend further generalization of the program by using macro variables for the filename and file pathing to enable easy changes and usage for other purposes within the program (such as naming other files).

```
/* At top of program, initiate LOG writing so that all input is captured */
%let dir = /sas/projectID/;       /* output locations */
%let stage = prod;                     /* devl, test, prod */
%let filename = projectID_extract1_20100215;
%let DateTime   =
      %sysfunc(compress(%sysfunc(today(), yymmddN8.)_%sysfunc(time(),hhmm6.), ': '));

/* Route Log to a saved location for later perusal  */
proc printto
   log = "&dir./logs/&stage./&filename._&datetime..log";
run;
```

In the example above, notice that the storage location varies by the phase of the project. When development is underway, the results are located in the DEVL subfolder. As the project moves into testing, results are located in TEST, and finally in PROD when we move to production. This way, it is easy to locate and review earlier versions.

Similar storage locations and naming conventions are used for the reports and other non-SAS file results.

**What about SAS Datasets?**

When creating permanent SAS datasets, including a date-time stamp in the name may be difficult due to the 32 character limit. Further, the data set name may need to be more static for downstream programs. Instead, consider generation data sets.

Each data set in a generation data group has the same member name (data set name) but has a different version number. Every time the data set is updated, a new generation data set is created and the version numbers of the older versions are incremented. The DEFAULT version is called the base version, and is the most recent version of the data. What are the advantages of using generation data sets?

- The most current version of the generation data group is referenced using the base data set name. Thus, downstream programs do not need to worry about date-time stamps in the data set name.
- Older versions of the data are available for `PROC COMPARE` testing when validating the data.
- You don't have to ***remember to save*** the data before creating a new version ☺.

For further information on creating and using generation data sets, see the **`genmax`** and **`gennum`** data set options in SAS online documentation.

## 4. TESTING AND QUALITY ASSURANCE

Our highly-modularized design lends itself to highly-modularized testing and quality assurance (QA). Unit testing will ensure that our modules receive metadata (via parameters) and produce a uniform result regardless of the values of the metadata. Our framework was designed and built with uniform interfaces between and across modules, creating SAS datasets with common keys. Results from each module can be tested independently as long as we have the common keys to bring the results together.

Essentials of our QA are:

- analysis of the contents of interim (or "QA") datasets which can easily be related back to our data requirements
- review of SAS logs from running each of our component modules
- walkthroughs of program logic to ensure what has been implemented matches what was in our detailed requirements
- analysis of our results in the form of analytical datasets and/or reports.

Initial QA of the first delivery of *enterprise* data will need to be exhaustive. It should be done by testers with in-depth knowledge of more than one division or group to ensure that the data extraction is done correctly and that the same metrics are understood to have the same meaning throughout the organization regardless of the *division* or *project*. The role of *enterprise* metrics are to provide consistent profile or relationship data across all deliverables so once our results have passed final QA the module will be frozen, copied to the production environment, and will be invoked by all of our job streams without further testing. Recognizing that there will likely be multiple iterations of data QA on each of our modules – at least for the pilot project – makes it worthwhile to develop a simple toolkit for validating data. There is no substitute for thorough and intelligent analysis of the initial data delivery but subsequent QA should focus on only those elements which should have changed.

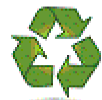
**For example**

If our *enterprise* module is to deliver 20 metrics and the first, detailed QA determines that 18 of those are correct and 2 need to be refined, isolating the 18 which shouldn't change and using `PROC COMPARE` to ensure that there were no unexpected changes to the good 18 metrics in subsequent deliveries will take only seconds to prepare and submit. A quick review of the output to ensure the `'No unequal values were found. All values compared are exactly equal'` message appears will then allow the QA time and energy to be focussed on the 2 metrics that were expected to change in our second iteration. Here is some utility code to compare results from different deliveries using the same sort of macro variables described in Section 3b. Note that only the first three macro variable definitions will need to change when reusing this code in the same environment.

```
%let old_dataset = enterprise_2009_09_01;
%let new_dataset = enterprise_2009_09_08;
%let not_same = var19 var20;

** same as Section 3b                                    **;
%let dir      = /base_directory;
%let stage    = test; /*test or prod*/
libname data "&dir./data/&stage./subfolder1";

** compare columns EXCEPT those we expect to be different **;
proc compare
      data = data.&new_dataset.(drop = &not_same.)
     compare = data.&old_dataset. (drop = &not_same.);
run;
```



This technique can be applied to each module at each iteration to help ensure consistency and data quality with minimal effort.

Much of the QA effort is devoted to ensuring that the results **data** in SAS data format are correct. After the data are correct we must also look at any **reports** or other files we are delivering. There is no substitute for a careful visual inspection of reports or formatted files to make sure there is no unexpected truncation or field overflow and that formats and rounding are as expected. If we convert from SAS to some other data format, such as Excel for reports or some other DBMS, there can be formatting quirks which are difficult to test for. Some things to look for include

- Is the smallest data value in each field displayed properly?
- Is the largest data value in each field displayed properly?
- Do rounded values match what would appear if we used `PROC PRINT`?
- Do calculated and formatted fields reflect the expected values?

Each new *division* module will also need thorough testing and QA. The same method as described above for focussing attention where it's needed for testing of *enterprise* will apply to *division*. Once each *division* has been validated, the module can be copied to production and invoked for each *project* without further testing. Our initial *project* module required extensive testing and QA review. For each additional *project* using the same framework, we assess the extent of differences from any existing *project* to isolate which elements require user QA testing. In many cases, we can reuse an entire *project* by tailoring the parameters passed to it without needing to touch the code in the module. This takes advantage of our design and the effort invested in early QA to minimize what's needed to deliver additional projects. Using standardized module names and storage locations, as described in the Code section above, and maintaining discipline around element names and formats throughout our hierarchy means the components fit together without concern over compatibility of interim results. Using a driver program of SAS code to select which modules to assemble into one stream allows us to pull together the various required and optional components and know that they will work together. This also reduces test time and effort for new *projects*.

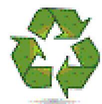
## 5. ROLLOUT

Once we have created the framework that will encompass our modules, we can start to focus on how to bring everything together in delivering the final product. We have focused on creating an extensible and reusable set of processes which if done correctly should simplify the rollout process.

We need to think about questions such as:

- how will new data requests be handled
- how and when will jobs be scheduled
- how will results get sent to users
- how to automate all of the above.

Metadata will be the driving force in addressing ways to handle our roll out process. Metadata is used to help describe the processes we are running and will be the gatekeeper to managing the execution, and delivery of such.



Let's examine some examples as to how we can manage the roll out dynamically.

**Job scheduling**

Imagine trying to determine which processes need to run monthly by working through a list of jobs, checking start and end dates. This can be very tedious, not to mention prone to human errors. Maybe step one of the development iteration involves creating a driver for each project identifier. Each of these drivers will describe a project and its attributes. Code can be conditionally run depending on what is passed down from our metadata.

Let's use an example where we want to run reports each month during a defined start and end period. This can easily be executed if the metadata defines when the processing start date and end dates are. By including conditional code in our driver program we can establish rules which the framework runs against thus automating which modules run. We can decide if the driver is supposed to run or not depending on the current month end in comparison to the processing start and end dates.

Let's check out a code snippet of what this might look like:

```
/* Create some example metadata */
data metadata;
  input projectID $ startDate:yymmdd10. endDate:yymmdd10.;
format startDate endDate yymmddd10.;
datalines;
project1 2010-01-01 2010-03-31
project2 2010-01-01 2010-03-31
project3 2010-02-01 2010-03-31
project4 2010-03-01 2010-04-30
;
run;

/*Set up previous month end date as current month end for the example*/
/*Typcially this would be dynamically generated*/
%let currentME = %sysfunc(intnx(month,%sysfunc(today()),-1,E));

/*Count the total number of Projects so we can create our macro variables*/
proc sql ;
     Select
        count(projectID)
     into :numberOfProjects
     From metadata
        ;
quit;
%let numberOfProjects = &numberOfProjects;

/*Place our metadata variable values into macro variables*/
/*so we can do logical processing*/
proc sql feedback;
     Select
        projectID,
        startDate,
        endDate
     into   :projectID_1 THROUGH :projectID_&numberOfProjects. ,
            :startDate_1 THROUGH :startDate_&numberOfProjects. ,
            :endDate_1 THROUGH :endDate_&numberOfProjects.
     From metadata
      ;
quit;

/*Cycle through metadata and compare with current month end date*/
/*If the current month end is between the start and end dates*/
/*we should call the corresponding driver module for the project*/
%macro masterMetadata;
%do i=1 %to &numberOfProjects; /*cycle through all the proejctIDs*/

    /*test the date condition - Check start and end date*/
    %if ( (%sysevalf(&&startDate_&i - 1) < &currentME) AND
          (%sysevalf(&&endDate_&i + 1) > &currentME)
         ) %then %do;

           /*Include code for the driver we want to call*/
           %let prog     =driver_&&projectID_&i...sas;
               %include code(&prog. ) ;


    %end;
%end; /*end of the do loop*/
%mend masterMetadata;
```

```
/*call the macro*/
%masterMetadata;
```

### New Requests

New data requests would involve the creation of new driver programs. These driver programs may only differ slightly in the logic and instructions they contain. This can be as simple as altering the project identifier so when the driver is executed it can refer back to the metadata which describes the execution rules for that project. This type of processing really simplifies the creation of new data or report delivery since metadata drives the processing.

### Data Delivery

Delivering the data to end users can also be driven from metadata cues. The metadata could describe the *enterprise* and *division* data which is being produced. Based on this information we can set up delivery locations or methods conditionally based on this metadata. Maybe a certain division would like an email pointing them to the location of their data, while another wants data placed in a certain location where they can locate it. This can all be achieved with conditional processing logic fed down from metadata.

```
/*We want to distinguish beetween US reports and Canadian Reports*/
/*They will be delivered to appropriate parties in respective locations*/
%macro dataDelivery(divisonID);

%if &divisonID = USA %then %do;
    /*US Reports should just be placed into the location with no notification*/
    libname data "&dir./data/&stage./subfolder1";
%end;
%else %if &divisonID = CAN %then %do;

    /*CAN Reports should be placed into the location with notification*/
    libname data "&dir./data/&stage./subfolder2";
    filename outbox email;
    data _null_;
          file outbox
          from="john.smith@mycompany.com"
          to="canadian.reporting@mycompany.com"
          subject="Canadian Reports Available";
           put "Canadian Reports will be available in the following location";
          put "&dir./data/&stage./subfolder2";
    run;
%end;

%mend dataDelivery;

%dataDelivery(USA);
%dataDelivery(CAN);
```

### A Step Further

The examples which we have discussed can be taken to a level higher where our driver programs would be automatically generated and executed based on metadata. This would mean our metadata would need to be fairly extensive in describing all the rules and relationships needed to work properly. A master program could take metadata and generate the conditional calls and rules to run our processes, in addition to **when** to run them. We have the project Start and End dates, so we know when the framework needs to be invoked for each project. Divisional indicator can describe the people who receive, and where they receive them. Project definitions would individualize the data to the requestors' needs. These metadata elements will help drive the use of the modular items we created and designed in earlier sections. A logical extension from generating a driver program for each unique *project* would be to have one **super-driver** program which invoked the relevant processes for the active projects for each reporting period.

## CONCLUSION

Building reusable and extensible code requires planning and discipline but the **benefits** outweigh the efforts. As demonstrated in this paper, some of the key benefits are

- Once the framework is in place and **metadata** is available for a project, results can be delivered rapidly with little effort. By defining and utilizing ***minimally acceptable inputs and results***, even if a component (such as *divisional* input) is not available, the other components (*enterprise* and *project*) are delivered to provide

immediate value.

- Code **modularization** not only helps **reusability**, but it also chunks the logic into digestible pieces. Documenting and sharing knowledge about short, focused modules is far easier than doing it for 1000's of lines of code in one chunk. The **Driver** / **Source** Programs along with **Macro Modules** are a key to the success of the framework. But, remember . . .

  a **repeatable** process like this requires additional and **extensive testing** to confirm that changes to the source code modules accommodate all ***drivers*** that call the ***modules***!

For large projects such as this, we also recommend **delivering intermediate products for review and user acceptance during the development which helps to structure the project and build respect and acceptance amongst the key stakeholders.** No need to wait for the BIG BANG! Let your key stakeholders enjoy results as early as they are available.

In the long run, time and effort is saved by building a robust modular process with wide applicability.

## SUGGESTED READING


1) Fecht, Marje. "Think Before You Type… Best Practices Learned the Hard Way?", *Proceedings of SAS Global Forum 2009.* http://support.sas.com/resources/papers/proceedings09/133-2009.pdf

2) Eckler, Lisa. "When Good Looks Aren't Enough", *Proceedings of NESUG, 2009.* http://www.nesug.org/Proceedings/nesug09/ff/ff13.pdf

3) Droogendyk, Harry and Fecht, Marje. "Demystifying the SAS Macro Facility" , *Proceedings of SUGI 31.* http://www2.sas.com/proceedings/sugi31/251-31.pdf


## ACKNOWLEDGMENT


The authors would like to thank James Moore for his review of the paper and his support of our approach during the project.


## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors via email at:


Faisal Dosani faisal.dosani@rbc.com
Lisa Eckler lisa.eckler@sympatico.ca
Marje Fecht marje.fecht@prowerk.com



SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.