

## ***THINK Before You Type... Best Practices Learned the Hard Way***

Marje Fecht

Prowerk Consulting Ltd, Mississauga, Ontario, Canada  
and Prowerk Consulting LLC, Cape Coral FL, USA

### **ABSTRACT**

We have all done it...

You leave a meeting with a new project on your plate – you sit down at your computer and start programming – a day later and lots of code on the screen, you try to remember the real purpose of your latest task. Or, worse, you finish your *task* only to find that it is not what was requested. Discipline and planning aren't always easy but they pay off in the long run!

OR

You deliver the quick adhoc and move onto the next task on the list. A month later, someone wants the "adhoc" again with just a wee variation. WHAT??? You thought you would never look at that code again? Generalization was the last thing on your mind and now you have to dig through all that code... Discipline and planning aren't always easy but they pay off in the long run!

This presentation focuses on best practices to help you **minimize effort** and **maximize results**. Although the concepts are more project-focused than SAS®-specific, example SAS code will be shared so that you can apply the concepts more quickly.

### **INTRODUCTION**

After 30 years of reporting, analytics, and SAS usage, it seems an appropriate time to reflect on the lessons learned. There are so many examples I encountered (and created) including the good, the bad, and the ugly. Each example becomes a lesson learned **if** you reflect and consider how you can improve on what you have just done or seen or inherited. This paper presents a series of examples to demonstrate some best practices – most of them were learned the hard way!

For example, we all enjoy the thrill of quickly delivering results, so there is one less task on the list. But, the **quickly** part can come back to haunt us if we don't think beyond the task to the bigger picture.

#### Example 1:



Have you ever QUICKLY written code assuming it will never be used again?? Is it now 5 years later and the quickly developed SPAGHETTI CODE is still in production? Worse still – does the code require YOUR time to tweak every time you run it? And, even worse, have others COPIED the code and put it into their production processes (and even given you credit for the code)?

Maintaining code is a big time-consumer and time-waster. This paper provides tips for avoiding this trap.

#### Example 2:



Have you **inherited** a time-consuming process or code that takes forever to "get ready" to use?? Does that process not only need to continue but it also has to be replicated?

This paper explores a real-life campaign reporting process that consumed about 5 days of time to produce reporting on a single campaign. Multiply that by over 5 campaigns a month and that doesn't leave much time for other work. The revised process now takes about ½ hour per campaign.

### Example 3:



Do you work on large projects that take so long to complete, that no one can remember why they were requested in the first place? Worse still, do the results provide no value when they are finally delivered?

This paper proposes an iterative approach that gets results in front of the requesters quickly, so that your efforts add value early in the project lifecycle.

## **BENEFITS**

What are some of the benefits you might enjoy after reading this paper?

- Reusable / repeatable processes
- Modularity and generalization of code
- Speed to market
- Consistent results (via code sharing)
- Satisfaction that you delivered value to the original team (before they all change jobs) !

## **THE QUICK ADHOC ? !**



*Lesson Learned:* There is no such thing as **the quick adhoc!**

You may think that the latest request will never be expanded upon or requested again but trust me, **it will.**

Before you process the request and crank out the results,

- Write up a brief explanation of the request. In fact, write it up as a COMMENT block for your code! Include the specifics like:
  - What is the population of interest (ie: what subset of data is being requested? )
  - Is there a particular time frame?
  - If this is a report, what might it look like in terms of
    - Rows
    - Columns
    - Format (dynamic vs static)
    - Metrics / measurements
    - Quantities / statistics desired.
- Send the above information back to the requester to confirm that you captured their needs.
- Then, before you start writing a program, think about **what could change about this request in the future?** If the request is *time-specific* or if it focuses on just a *subset of the available population*, chances are good that you can generalize the code so that it can be easily adapted to future requests. (see examples of code generalization from the **suggested reading** section including (1) and (2))

The end result? While the above steps may add 15-30 minutes to the task, you have accomplished a few key benefits:

- You confirmed that you understood the request, rather than delivering something that is off target.
- You built a program that can be adapted to future requests.

**Don't Stop Here...** You DID deliver the results, but consider a nice next step that will only take 5 minutes, and save time later. Build a history of this request (and those that will come) that enables you to easily locate and rerun the results or similar requests. For example

- Utilize a **numbering system** for all requests; it will help in communications with the requester(s) and will help you identify all components of the request including
  - Specifications
  - Programs
  - Data files
  - Logs
  - Results.

- Keep a **Request Log** (*inventory*) for easy referral.
  - This might be something as simple as a document or Excel spreadsheet. Or, it could be a web-based application that tracks status and progress on all requests.
  - When results are delivered, log the request as **Closed** and note the date delivered.
  - **The Key:** Make sure the log is searchable by number, and by keyword. I like to search for “similar requests” to gain valuable background and also to re-use existing code modules.
- File Naming
  - Utilize a file naming convention that includes at least the following components
    - Request Number
    - Version Date in `yyyymmdd` format for easy sorting
    - Request information such as requester or focus of request
    - **Example:** `VINT07_AB_IAV_20081102.sas` (which follows the format of `RequestID_Requester_Product_yyyymmdd.sas`)
  - Extend the same naming convention to
    - SAS programs
    - Logs
    - Results
    - Validation files.

## REPEATABLE PROCESSES



*Lesson Learned:* A repeatable process is only **truly** repeatable if it is designed to be repeatable!

I have inherited A LOT of not-so-repeatable processes. The instructions have “copy and paste” throughout. And, the instructions also rely on a lot of manual searches to find the right files, code, phrases, etc. .

Suppose you produce reports that use the same logic, extracts, and reporting formats on a regular basis for each new product / campaign / region. The programs and results are similar except for changes to:

- Dates
- Inclusion / exclusion criteria
- Title and footnote text
- Search logic
- Output locations
- Report contents
- etc.

Your current approach is to hunt (everywhere!) for the most recent version(s) you can find of the query / reporting program and then after you FINALLY find the file, you copy the code and change the name. Next you have to:

- Step through the code to locate all of the required changes
- Start typing to input the new information
- Run the code and OOPS – fix the semicolons and quotes you inadvertently overtyped
- Try again.....

A better approach is to design the process as **truly repeatable**. For example, similar to the **adhoc** example, identify components of your current code that are reusable. If you isolate and generalize those components and identify them as **includable modules**, then you can design your processes to reduce the pain.

## APPROACH

- 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
- Unlike the **adhoc** example, a repeatable process requires additional and extensive testing to confirm that changes accommodate all **drivers** that call the **modules**.

## EXAMPLE

Assuming that you have generalized the extraction, summarization, and reporting processes, then when you are ready to start reporting for a new initiative, just provide the parameter values as input to your standard programs.

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

- CampaignReportingExtract\_cheque\_2009\_01\_10.sas
- CampaignReportingOutput\_cheque\_2009\_01\_13.sas

### Contents of driver program

```
*** Driver Program - specify appropriate values for parameters;
%let type = cheque; *** Campaign Type (cheque, prodchg, usage, etc.);
%let prestart = 01OCT2008; *** Pre campaign period for comparison;
%let prestop = 31DEC2008; *** End of Pre campaign period;
%let poststart = 01JAN2009; *** start campaign tracking;
%let poststop = 30JUN2009; *** expire date - stop tracking;

%let title = "January 2009 Introduction of Lower Interest Rates";

%let products = ('VRS', 'GQL', 'BGO', 'DLA');
%let codes = ('015', '119', '214');

*** run standard extract and reporting programs;
%include 'CampaignReportingExtract_&type._2009_01_10.sas';
%include 'CampaignReportingOutput_&type._2009_01_13.sas';
```

## IMPROVED 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.

Create a program that references the latest source version:

### Contents of CampaignReportingExtract\_cheque\_CurrentPgm.sas

```
*** call latest version of source program;
%include 'CampaignReportingExtract_cheque_2009_01_10.sas';
```

### Revised contents of driver program

```
*** Driver Program - specify appropriate values for parameters;
%let type = cheque; *** Campaign Type (cheque, prodchg, usage, etc.);
%let prestart = 01OCT2008; *** Pre campaign period for comparison;
%let prestop = 31DEC2008; *** End of Pre campaign period;
%let poststart = 01JAN2009; *** start campaign tracking;
%let poststop = 30JUN2009; *** expire date - stop tracking;

%let title = "January 2009 Introduction of Lower Interest Rates";

%let products = ('VRS', 'GQL', 'BGO', 'DLA');
%let codes = ('015', '119', '214');

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

The above approach really works! A current **hands-off** production process publishes over 75 daily and 200 monthly reports. When reporting upgrades occur (new metrics / revised formats / new SAS features), they immediately roll out to **all** reports that are still running. Obviously, this requires end-to-end testing ☺ .

## NEW PROJECTS



*Lesson Learned:* You are NOT just a programmer... You can add value and decrease *time-to-market* by providing structure and content to the process.

We all dread the *project-launch* meeting request that has over 20 invitees and no background information. Or worse – without any involvement in the process - the email that says “here is what the project team wants you to produce”. Many folks in analyst or reporting roles forget that they **have a say!**

Do not sit back and play “order taker”. Take an active role in new projects by providing structure and encouraging:

- Well designed specifications
- Phasing – so that key results are delivered quickly to get **buy-in** and maintain project focus.

Consider some of the processes detailed in the next section, to help with your involvement in projects.

## KEY LEARNINGS



The examples in this paper have many common themes, including following a process to maintain focus! In general, the process to aim for is:

Specifications (develop / signoff)

- Research/planning
- Code plan
- Code Development for a Phase
- Debugging
- Testing / Validation
- Deliver Phase of results
- Back to code plan for next phase ...

- **Specifications**
  - Gathering Information by meeting with the intended business users of your results
    - What decisions will be made (questions answered) with what you deliver?
    - Requirements – what do they *think* they want?
    - Prioritization of needs (to help with phasing) – What do they perceive to be the most important pieces? This will help you develop phases (for releasing results quickly) to get early confirmation and buy-in. This is also a key for showing early successes!
    - Format and delivery channel for results: How do they want to receive the information?

- Develop documentation of what you perceive was requested (formality depends on scope of project). Include:
  - Phasing (so that you can deliver results quickly, get buy-in, and keep project sponsors happy)
    - Content of each phase
    - Expected results from each phase
    - Approximate timing for each phase ( based on specifications as stated)
    - Cost estimates of each phase, if appropriate.
  - Exclusions / inclusions / Population of interest
  - Metrics / Dimensions / hierarchies – including perceived definitions
  - Usability considerations.
  
- Deliver the Specifications and request Sign-Off / Approval.
  - Indicate that development does not begin until the specs are agreed to.
  
- **Research**
  - Understand the data
    - For the metrics, determine ranges / expected answers – before you get caught up in cranking out results
    - Identify data experts for help with validation, to make sure the numbers really make sense
    - What are the definitions and calculations required?
      - For example, if **# Active Accounts** is a metric, how is an *active account* defined? What is timeframe (last 30 days? Last quarter?) ? What activity (transactions, payments, etc) are included?
    - Are there data anomalies to be documented / considered?
    - Are there other reports / projects that are similar? If yes, what is different about the current project? What can be borrowed? **AVOID** producing differing versions of the same information! If your results are different, explain in advance **why** and make sure you have buy-in.
  
- **Code Plan**
  - Build program construct by starting with comments that detail your approach, including the steps needed to reach your end results.
    - This will serve as the **outline** for your program (much like an outline you use when preparing a paper or presentation)
    - Be granular, since with a phased approach many of the steps will come later in the process, and thus the comments will help you remember what is next ☺ .
    - Include the Phase numbers, so that you avoid building code before you should.
    - Include your plans for:
      - Locations for logs, output, programs – in both test and production environments

- Data sources and validation notes (from Research step)
  - Details of calculations
  - Plan for EXPANSION . . . Very few requests remain static, so acknowledge that and build your approach as generally as possible. Refer to **Suggested Reading** (1) and (2) for tips on generalization of code.
- **Code development for a Phase**
  - Focus on GENERALIZING whenever you can
  - Focus on MODULARIZATION. If you are *borrowing code*, do NOT copy / paste. Instead, consider “including the step” via a macro call or %include . Refer to **Suggested Reading** (1) and (2) for tips on modularization of code.
  - Include documentation of validation efforts, so you don’t forget what you learned
  - Be sure to **save** and **review** Logs. Utilize `proc printto` and include date-time stamps in the file names. Refer to **Suggested Reading** (2) for tips on file naming.
  - Use Version control for programs and code segments. Also incorporate *program version* information in the report footnotes. You need to be able to link results to the appropriate code and also be able to **roll-back** to something that worked ☺
- **Testing / Validation**
  - Assuming that you identified valid values, ranges, and expected results in the **Research** phase, then that is the first thing to review!
  - Test your programs to confirm that the logic is properly handling all possibilities. Refer to **Suggested Reading** (3) for some testing tips.
  - View samples of the intermediate datasets, to confirm that values, sequences, etc make sense.
  - Utilize `proc univariate`, or similar tools to validate metric ranges and values.
  - When changing code, utilize `proc compare` to confirm that your changes have not negatively impacted results. Confirm appropriateness of
    - Variable formats, lengths, and labels
    - Dataset attributes.
- **Deliver Phase of Results**
  - As each phase of results becomes available
    - Freeze the code version(s) and do not touch!!
    - Publish the results, and announce to the requesters. Request feedback on
      - Any data concerns
      - Usability of report format and contents.
  - While you await feedback, you can begin back at **Code Plan** for the next phase.

- **Moving to Production**

For a request that will **run regularly** (daily, monthly, quarterly), be sure to carefully **document** the production process including

- When and how does it run? (automatically vs manual process)
- What data / information (e.g., dependencies) needs to be available before it can run?
- How do users know when information is ready?
  - Consider automated email alerts when information is ready
  - Consider Web publishing of information.

Always maintain version control! Make sure that all versions of code (and modules) are properly named with date of last update, so that you can locate and troubleshoot as needed. Also, maintain the **modifications** section of your comments, so that you always know what changes were introduced (and by whom on what date).

## MY TOP 5 SUGGESTIONS

- Always write your code as generically as possible
- Use a phased approach, which includes delivering important results quickly (for buy-in)
- Participate in meetings – do not become an *order taker* - work directly with the information consumer
- Use code comments liberally, including starting with a code plan (in the form of comments)
- **SMILE** – it really is fun to deliver knowledge!

## SUGGESTED READING

- 1) Fecht, Marje and Stewart, Larry. "Are Your SAS Programs Running You?", *Proceedings of SAS Global Forum 2008*. <http://www2.sas.com/proceedings/forum2008/164-2008.pdf>
- 2) Droogendyk, Harry and Fecht, Marje. "Demystifying the SAS Macro Facility", *Proceedings of SUGI 31*. <http://www2.sas.com/proceedings/sugi31/251-31.pdf>
- 3) House, Laura. Presentation at TASS (September 21, 2007). "Testing Traps: Creating Good Test Data 101", [http://www.torsas.ca/downloads/TASS\\_20070921\\_Testing101.pdf](http://www.torsas.ca/downloads/TASS_20070921_Testing101.pdf)

## ACKNOWLEDGMENTS

The author would like to thank the following people

- Lisa Eckler – for including this presentation at SAS Global Forum 2009
- Laura House and Thomas Rothschild – for their review and valued feedback
- Robert Fecht – for his wisdom and insight, especially about the macro language components
- Harry Droogendyk – *for being "Harry"*

## CONTACT INFORMATION

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

[marje.fecht@prowerk.com](mailto: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.