PharmaSUG 2026 – Paper AP-418

Code Hard and Put away Wet: Replacing Hardcoded SAS® Software Quality Checks with Data-Driven Design and Defensive Programming Techniques That Validate Code and Control Data

Troy Martin Hughes

ABSTRACT

You wouldn't ride that pony hard and put her away wet, so why subject your SAS® software to the same ill treatment?! *Defensive programming* describes a risk management strategy that aims to identify threats to software functionality and/or performance before they occur and, where possible, to identify pathways to programmatically mitigate those risks (or to communicate realized threats to stakeholders). This talk introduces defensive programming techniques that can be implemented when software executes, such as verifying SAS program file state (i.e., availability, accessibility) and program file metadata (e.g., filename, checksum, creation date, version). Data-driven software design further espouses the reality that "software" comprises not only code but also the underlying control data that drive that code's functionality and flexibility. Thus, defensive data-driven design requires a fuller risk management strategy that evaluates risks not only to code but also to control data, including lookup tables, control tables, configuration files, and other control files. Finally, although not considered to be "software," domain data sets and other data sources can be no less essential to ensuring software success, so the state and quality of these transactional data can be evaluated as well. Given this more expansive view, this text further demonstrates defensive programming techniques that evaluate the state and metadata of required input data, be they SAS data sets, spreadsheets, CSV files, XML, JSON, or other interoperable (i.e., *canonical*) formats. SAS user-defined subroutines, built using PROC FCMP, the SAS Function Compiler, call Python user-defined functions to extract file metadata and to calculate file checksums—all of which aim to ensure the availability of software prerequisites at runtime. Defensive programming methods should be implemented where robust software must execute reliably, and data-driven software should incorporate these best practices.

INTRODUCTION

Defensive programming has been extolled and documented ad nauseum, including by the author, insofar as only the briefest of introductions to its concepts and design is warranted here. The goal of defensive programming is to make software more robust and resilient to both known and unknown risks, which ultimately improves software reliability. In some cases, known vulnerabilities are identified (such as software's reliance on an external program file) and their risks mitigated (such as by verifying that the external file exists). In other cases, more generalized errors or failures can occur, and these must be identified and handled. *Exception handling* processes and concepts are introduced in the author's text: *Why Aren't Exception Handling Routines Routine? Toward Reliably Robust Code through Increased Quality Standards in Base SAS*. (Hughes, 2015)

Defensive programming (including exception handling) aims to deliver full business value—that is, full functionality and full performance—in the face of software risk. In many cases, however, failure thwarts software functionality, and the role of defensive programming aims instead to ensure *graceful termination*, and causes software to fail predictably to a known state. This is 2026 and everybody needs a safe space, eh? Graceful termination also requires apt documentation or communication of a failure to stakeholders, status dashboards, etc. For example, the handling of missing, locked, or otherwise invalid data sets is demonstrated in the author's text: *Yo Mama is Broke 'Cause Yo Daddy is Missing: Autonomously and Responsibly Responding to Missing or Invalid SAS® Data Sets Through Exception Handling Routines.* (Hughes, 2021)

In some cases, defensive programming can deliver *partial business value*, which might encompass full functionality despite reduced performance. For example, if a required data set is missing or locked, software might enter a busy-waiting cycle in which it repeatedly tests for file existence and availability until either the file can be accessed or a parameterized amount of time has passed; in the former case, full functionality is

delivered, albeit with a delay (i.e., reduced performance), and in the latter case, the process terminates without delivering business value. The author introduces busy-waiting defensive programming in a prior text: *From a One-Horse to a One-Stoplight Town: A Base SAS Solution to Preventing Data Access Collisions through the Detection and Deployment of Shared and Exclusive File Locks*. (Hughes, 2015) Busy-waiting concepts are further explored in a subsequent text: *Beyond a One-Stoplight Town: A Base SAS Solution to Preventing Data Access Collisions through the Detection, Deployment, Monitoring, and Optimization of Shared and Exclusive File Locks*. (Hughes, 2015)

And in other cases, *partial business value* might entail software that delivers only partial functionality—for example, software that builds only three of four expected reports, where the creation of three reports (partial business value) is preferred to no reports (no business value). These various exception management pathways, which span from full functionality to partial functionality to graceful termination, are described fully in the author's follow-up text: *Ushering SAS Emergency Medicine into the 21st Century: Toward Exception Handling Objectives, Actions, Outcomes, and Comms*. (Hughes, 2016)

This text focuses narrowly on defensive programming techniques that aim to ensure prerequisite files are available and accurate to support software execution. For example, file *checksums*—unique cryptographic identifiers—and file creation dates are demonstrated as two metadata components that can aid in file evaluation, and these control data can be maintained in separate control files that drive defensive programming processes. Similarly, the author demonstrates how control tables maintained as data dictionaries can be used to validate programmatically whether SAS data sets (or other data sources, such as Excel spreadsheets) contain the correct variables, variable order, and variable data types: *The Doctor Ordered a Prescription…Not a Description: Driving Dynamic Data Governance Through Prescriptive Data Dictionaries That Automate Quality Control and Exception Reporting.* (Hughes, 2018)

DATA-DRIVEN SOFTWARE DESIGN

Data-driven software design is defined as "software design in which the control logic, program flow, data rules, business rules, data models, data mappings, and other dynamic and configurable elements are abstracted to control data, and are interpreted by (rather than contained within) code." (Hughes, 2022) In this paradigm, code becomes the *interpreter* of business rules rather than the *container* of business rules—a *business rules engine (BRE)*, as it were, that can evaluate and operationalize external rules. Additionally, *software* itself must be understood to comprise not only code but also control data. And these paradigm shifts dictate that data-driven design must be reflected in defensive programming in at least two ways.

First, as the definition of *software* is expanded to include control data—including parameters, configuration files, control tables, and other control files—these control data must be afforded the same defense as their corresponding code. For example, if code is versioned, or if code metadata (such as program file create date, update date, or checksum) are evaluated at software execution, these techniques likely should be expanded to apply to the software's control files.

Second, data-driven techniques themselves can be used in defensive programming, such as by generating templates against which transactional data sets or other data or control data must be compared to validate data structure, file format, file accuracy, file completeness, etc. Moreover, where a program file or control file is expected to remain stable over time, its checksum can be calculated, saved for posterity, and re-evaluated at software execution to ensure the correct file is being used. In both examples, the structure or metadata are maintained external to the code, espousing true data-driven design.

SETUP FOR SCENARIO

Consider the print_something_twice.sas program file, which contains the PRINT_SOMETHING_TWICE macro:

```
* print something twice;
%macro print_something_twice(something);
%put &something;
%put &something;
%mend;
```

Next, consider an unspecified SAS program file that might require the PRINT_SOMETHING_TWICE macro for execution. That is, the print_something_twice.sas program file is a prerequisite that must be referenced via the %INCLUDE macro statement. However, prior to this file inclusion, defensive programming objectives might seek to validate that the correct macro is being included. To facilitate this file validation, the file metadata can be captured for the validated file and subsequently compared.

Whereas PRINT_SOMETHING_TWICE hardcodes its iteration by *always* printing something two times, a more configurable (and data-driven macro) might instead parameterize the number of times something should be printed. This parameter could be declared and passed directly (i.e., by value); or the location, filename, and parameter name could be passed (i.e., by reference); or myriad other methods could be used. One of the most straightforward methods simply passes a parameter containing the number of times something should be printed, and this macro can be saved as print_something.sas:

```
* print something a parameterized number of times;
%macro print_something(something, num);
%local cnt;
%do cnt = 1 %to &num;
   %put &something;
   %end;
%mend;
```

This data-driven design effectively requires the control data (&NUM) to be separated from the code interpreting it. Thus, print_something_control_data.csv operates as a simplified control table that contains a single parameter:

```
how_many_times,5
```

That is, the control table first must be accessed, after which 5 can be extracted and passed to the PRINT_SOMETHING macro. Alternatively, PRINT_SOMETHING itself could have opened the control table and extracted the value, although this design is not demonstrated. For example, the following DATA step reads the CSV file and prints the value (5) to the log:

```
data _null_;
   infile "&loc.print_something_control_data.csv" dsd;
   length varname $50 value 8;
   input varname $ value;
   put value=;
run;
```

It is unlikely that either macro, PRINT_SOMETHING_TWICE or PRINT_SOMETHING, would need to be modified frequently, so a cryptographic checksum can be calculated and used to validate that the correct version of the macro has been accessed (via the %INCLUDE macro). A control table, on the other hand, facilitates configurability, and thus might be expected to change each time software is executed. For this reason, although a checksum still can be calculated on the control file, other validation methods such as evaluating the data structure, its contents, or its metadata (e.g., creation date or update date) are more likely to be useful. These methods are demonstrated in the following sections.

LEVERAGING PROC FCMP TO LEVERAGE PYTHON TO EXTRACT METADATA

The FCMP procedure (aka the SAS Function Compiler) has numerous superpowers, among which is the ability call Python user-defined functions (and subroutines) natively using a Python interpreter of choice. This text only glancingly introduces the FCMP procedure, although the author's textbook fully demonstrates FCMP syntax, functionality, and its clear contributions to improving SAS software quality: *PROC FCMP User-Defined Functions: An Introduction to the SAS® Function Compiler, Second Edition*. (Hughes, 2026)

Consider the GET_FILE_INFO user-defined subroutine, which retrieves the create date, modification date, file size, and MD5 checksum for any file:

```
%let loc=d:\sas\;                    * MUST BE CHANGED BY USER;
proc fcmp outlib=work.funcs.py;
   subroutine get_file_info(folder $, filename $, filesize, md5hash $,
         create_datetime, update_datetime);
      outargs filesize, md5hash, create_datetime, update_datetime;
      declare object py(python);
      rc = py.infile("&loc.get_file_info_python_function.py");
      rc = py.publish();
      rc = py.call("get_file_info_function", folder, filename);
      filesize = py.results["size_bytes"];
      md5hash = py.results["file_md5"];
      create_datetime = py.results["dt_created"];
      update_datetime = py.results["dt_modified"];
      endsub;
quit;
```

The SAS subroutine is merely a shell that represents a higher level of software abstraction, and GET_FILE_INFO_FUNCTION (a Python function defined inside the get_file_info_python_function.py program file) pulls the file metadata and calculates the file checksum, all of which are returned to the SAS subroutine using the Python Component Object RESULTS method. The subroutine in turn modifies the FILESIZE, MD5HASH, CREATE_DATETIME, and UPDATE_DATETIME parameters, and makes these arguments available to the calling program (via call by reference, which is optionally specified using the OUTARGS statement). That is, although a SAS subroutine (whether built-in or user-defined) is incapable of truly “returning” a value to the calling program (because the RETURN statement is supported in function definition but not in subroutine definition), a subroutine can nevertheless modify one or more values in the calling program via the OUTARGS statement.

The GET_FILE_INFO SAS subroutine in turn calls the GET_FILE_INFO_FUNCTION Python function:

```
import os
from datetime import datetime
import hashlib

def get_file_info_function(folder, filename):
    "Output: size_bytes, file_md5, dt_created, dt_modified"
    path_file = os.path.join(folder, filename)
    try:
        _stats = os.stat(path_file)
        dt_created = datetime.fromtimestamp(_stats.st_ctime)
        dt_modified = datetime.fromtimestamp(_stats.st_mtime)
        size_bytes = _stats.st_size
        file_md5 = hashlib.md5(open(path_file, 'rb').read()).hexdigest()
    except:
        size_bytes = None
        file_md5 = None
        dt_created = None
        dt_modified = None
    return size_bytes, file_md5, dt_created, dt_modified
```

The function extracts the file creation date, file modify date, and file size, and subsequently calculates the MD5 cryptographic checksum using the MD5 method in the HASHLIB package. These values are subsequently returned to the SAS subroutine via the Python RETURN statement.

To retrieve metadata for the two SAS macro files and the single control file, the following DATA step creates the Files data set in which each filename appears in the Filename variable:

```
data files;
   infile datalines dsd;
   length filename $50;
   input filename $;
   datalines;
print_something_twice.sas
print_something.sas
print_something_control_data.csv
;
```

The OPTIONS statement is required to specify where the FCMP procedure compiled and saved the GET_FILE_INFO subroutine (which had been specified previously using the FCMP OUTLIB option):

```
options cmplib=work.funcs;
```

Thereafter, the following DATA step calls GET_FILE_INFO, passes the folder and filename, and retrieves the create date, modified date, file size, and checksum:

```
data file_metadata;
   set files;
   length filesize_bytes 8 filename_hash $32 create_datetime update_datetime 8;
   format create_datetime update_datetime datetime19.;
   call get_file_info("&loc", filename, filesize_bytes, filename_hash,
create_datetime, update_datetime);
run;
```

The DATA step creates the File_metadata data set, which contains the retrieved and calculated metadata for all three files.

	filename	filesize_bytes	filename_hash	create_datetime	update_datetime
1	print_something_twice.sas	112	e958d51836d7c78a80e38b626308cd49	13SEP2025:16:37:49	14SEP2025:21:07:59
2	print_something.sas	168	613ad16ed8c91e55533f4f1f50546052	13SEP2025:17:18:46	14SEP2025:21:20:10
3	print_something_control_data.csv	18	2e4e8afa23ef6d7fdd6089b670d6aa82	13SEP2025:17:07:28	14SEP2025:21:48:09

File_metadata Data Set

These metadata can be subsequently used as the control baseline against which software components (i.e., prerequisite files) will be validated when running, as demonstrated in the next section.

VALIDATING SOFTWARE PREREQUISITES AGAINST EXPECTED METADATA

The previous section extracted and calculated file-level metadata, which can facilitate defensive programming objectives when trusted baselines are consulted and compared to real-time metadata. Consider the DO_SOMETHING macro, which requires the PRINT_SOMETHING macro, which in turn requires the print_something_control_data.csv control table (to specify the number of iterations to print something):

```
%macro do_something;
* validate that the program file checksum matches;
%local filename bytes hash create_dtg update_dtg;
```

```
%let filename=print_something.sas;
%let bytes=0;
%let hash=01234567890123456789012345678901;
%let create_dtg=0;
%let update_dtg=0;
%syscall get_file_info(loc, filename, bytes, hash, create_dtg, update_dtg);
%put &=bytes &=hash &=create_dtg &=update_dtg;

* evaluate real-time calculated checksum against expected value;
proc sql noprint;
   select filename_hash into: hash_true from file_metadata where
      filename = 'print_something.sas';
quit;

* only import and call PRINT_SOMETHING if hash values match;
%if &hash=&hash_true %then %do;
   %include "&loc.print_something.sas";

   * evaluate if the required control table exists;
   %if %sysfunc(fileexist(&loc.print_something_control_data.csv))=1
         %then %do;

      * perform other validation (NOT SHOWN) on the control table;
      * extract the iteration parameter (HOW_MANY_TIMES) from the control table;
      data _null_;
         infile "&loc.print_something_control_data.csv" dsd;
         length varname $50 value 8;
         input varname $ value;
         call symputx(varname, value);
      run;

      * dynamically call PRINT_SOMETHING using parameterized value;
      %print_something(Hey there!, &how_many_times);
      %end;
   %end;
%else %do;
   %put Something Failed!;
   %end;
%mend;
```

When DO_SOMETHING is executed, the message is printed five times, as specified in the control table:

```
%do_something;

Hey there!
Hey there!
Hey there!
```

```
Hey there!
Hey there!
```

The DO_SOMETHING macro showcases (at least behind the scenes) some of the complexity inherent in defensive programming, as several steps must be conditionally executed:

1. The current checksum of print_something.sas is calculated by calling the GET_FILE_INFO user-defined subroutine via the %SYSCALL macro statement; this current checksum is saved into the &HASH macro variable.
2. Because the PRINT_SOMETHING macro is not expected to change over time, the baseline checksum for the validated version of print_something.sas had already been saved in the File_metadata data set, and the SQL procedure extracts this checksum and saves it to the &HASH_TRUE macro variable.
3. Only when the checksums match (between the current version of the macro program file and the previously validated version of the same program file) does the DO_SOMETHING macro conditionally %INCLUDE the DO_SOMETHING macro.
4. The FILEEXIST built-in function evaluates whether the control table exists, and is used as a proxy for validating control files. In a more realistic example, the data structure and contents of the control table would need to be validated as well—for example, the parameter name, data type, and format. The modify date of the control file could also be used to ensure the table had (or had not) been changed before (or after) some established threshold.
5. The DATA step extracts the HOW_MANY_TIMES parameter from the CSV control table and initializes the &HOW_MANY_TIMES macro variable to its value (5).
6. Finally, the PRINT_SOMETHING macro is called, which relies on both the parameterized message (“Hey there!”) and the iteration parameter, retrieved from the control table.

Throughout this process, the metadata against which files are validated are maintained in the File_metadata data set; thus, even more complex (and extensive) metadata could be incorporated into this comparison and validation while supporting this data-driven defensive design.

CONCLUSION

Don’t ride your pony hard and put her away wet! Rather, defensive programming aims to make software more reliable by improving robustness to failure, and should address both known and unknown risks. Within data-driven software design, the objectives of defensive programming must be expanded to encompass risks to not only code but also the control data driving code. Although this text demonstrated a single, straightforward example in which the hash checksum of a SAS macro program file must be evaluated against a validated checksum baseline, and in which an associated control table is (only) validated to exist, it outlines processes through which a control table can be used to instill defensive programming methods.

REFERENCES

Hughes, T. M. (2015). Beyond a One-Stoplight Town: A Base SAS Solution to Preventing Data Access Collisions through the Detection, Deployment, Monitoring, and Optimization of Shared and Exclusive File Locks. *Midwest SAS Users Group (MWSUG).* Omaha, NE. Retrieved from https://www.mwsug.org/proceedings/2015/BB/MWSUG-2015-BB-18.pdf

Hughes, T. M. (2015). From a One-Horse to a One-Stoplight Town: A Base SAS Solution to Preventing Data Access Collisions through the Detection and Deployment of Shared and Exclusive File Locks. *SAS Global Forum (SGF).* Dallas, TX. Retrieved from https://support.sas.com/resources/papers/proceedings15/3380-2015.pdf

Hughes, T. M. (2015). Why Aren’t Exception Handling Routines Routine? Toward Reliably Robust Code through Increased Quality Standards in Base SAS. *Western Users of SAS Software (WUSS).* San Diego, CA. Retrieved from https://www.lexjansen.com/wuss/2015/80_Final_Paper_PDF.pdf

Hughes, T. M. (2016). *Data Analytic Development: Dimensions of Software Quality.* Hoboken, NJ: John Wiley and Sons.

Hughes, T. M. (2016). Ushering SAS Emergency Medicine into the 21st Century: Toward Exception Handling Objectives, Actions, Outcomes, and Comms. *Western Users of SAS Software (WUSS).* San Francisco, CA. Retrieved from https://www.lexjansen.com/wuss/2016/94_Final_Paper_PDF.pdf

Hughes, T. M. (2018). The Doctor Ordered a Prescription…Not a Description: Driving Dynamic Data Governance Through Prescriptive Data Dictionaries That Automate Quality Control and Exception Reporting. *South Central SAS Users Group (SCSUG).* Austin, TX. Retrieved from https://www.lexjansen.com/scsug/2018/rs27.pdf

Hughes, T. M. (2021). Yo Mama is Broke 'Cause Yo Daddy is Missing: Autonomously and Responsibly Responding to Missing or Invalid SAS® Data Sets Through Exception Handling Routines. *Southeast SAS Users Group (SESUG).* Retrieved from https://www.lexjansen.com/sesug/2021/SESUG2021_Paper_93_Final_PDF.pdf

Hughes, T. M. (2022). *SAS® Data-Driven Development: From Abstract Design to Dynamic Functionality, Second Edition.* San Diego: Kindle Direct Publishing.

Hughes, T. M. (2026). *PROC FCMP User-Defined Functions: An Introduction to the SAS® Function Compiler, Second Edition.* San Diego, CA.

CONTACT INFORMATION

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

Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com