Paper 58-28

Beyond Debugging: Program Validation

Neil Howard, Ingenix, Basking Ridge, NJ

Abstract

"Act in haste and repent at leisure; code too soon, and debug forever."

Raymond Kennington

In their paper on debugging, Lora Delwiche and Susan Slaughter say that good debuggers make good programmers. Let's take that one step further to say that good analysts and problem-solvers make good programmers. Just because a SAS® program is free of errors, warnings, notes, and bugs does not guarantee that the program is doing what it is supposed to do. This tutorial addresses the process that follows debugging: program validation. It covers techniques for ensuring that the logic and intent of the program is correct, that the requirements and design specifications are met, and that data errors are detected. It also discusses fundamental SAS program design issues that give purpose and structure to your programming approach.

This paper will address: the definitions of verification, validation, testing, and debugging, as well as the structure of the Software Development Life Cycle (SDLC). It will illustrate how the SAS system can be used to help satisfy the requirements of your SDLC and accomplish the tasks of verification and validation.

Since as much as 80% of a programmer's time is invested in testing and validation, it's important to focus on tools that facilitate correction of syntax, data, and logic errors in SAS programs. The presentation focuses on wide variety of SAS features, tips, techniques, tricks, and system tools that can become part of your routine testing methodology.

Introduction

[...Overheard at an interview for a SAS programming position: "But you don't ***have*** to test SAS programs!!!"....]

As the interviewers quickly escort the confused candidate out the door, they recall how often it is assumed that a fourth generation language "does so much for you" that you don't have to test the code. The SAS system is easy to use, and the learning curve to productivity is relatively short. But SAS is just as easy to ABUSE. Programmers and analysts must not lose sight of the indisputable facts: data is seldom clean, logic is too often faulty, and fingers walk clumsily over keyboards. Condition codes are not an accurate indicator of successful programs.

There are traditional methodologies for preventative pest control, but there is no PROC TEST-MY-CODE or PROC WHITE-OUT or PROC READ-MY-MIND. The SAS system offers many features for identifying syntax, logic, and data errors. The results will undoubtedly include reduced stress and bigger raises for SAS programmers, satisfied clients, accurate output, and quality programs that are reliable and maintainable. This supports the business need to deliver results to the FDA, minimize time to approval and time to market.

Definition of Terms

The responsibility for ensuring program quality remains with the programmer. Today's competitive environment demands that we discuss testing methods and useful SAS system tools that will help us meet the challenges of verification, validation, testing, and debugging. The following definitions were provided by the validation maven at Parke-Davis and serve as the benchmarks for this paper.

VERIFICATION: Checking (often visual) of a result based on predetermined criteria, e.g., check that a title is correct.

VALIDATION: The process of providing documented evidence that a computerized system performs its functions as intended and will continue to do so in the future.

TESTING: Expected results are predetermined, so actual results can be either accepted or rejected by one or more of the testing types: unit, integration, worst case, valid case, boundary value, alpha, beta, black box, white box, regression, functional, structural, performance, stability, etc.

DEBUGGING: The process of finding and correcting the root cause of an unexpected result.

Terms are Relative

The author conducted a brief informal survey: 1) within Parke-Davis, among the Clinical Reporting Systems management team, selected senior programmers and systems analysts, clinical team leaders, developers and biometricians, and 2) beyond the company, within a selected network of colleagues in the SAS community. The intent of the survey was

to see how well our baseline definitions help up against greater scrutiny, perception and application.

IEEE on Terms

One survey respondent follows the Teri Stokes school of validation, based on IEEE standards. Std 1012-1986, "IEEE Standard for Software Verification and Validation Plans", states:

VERIFICATION is "the process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase." Informally, making sure your program is doing what you think it does.

VALIDATION is "the process of evaluating software at the end of the software development process to ensure compliance with software requirements." Informally, making sure the client is getting what they wanted.

TESTING is "the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs), and to evaluate the features of the software item."

IEEE Standard 610.12-1990, "IEEE Standard Glossary of Software Engineering Terminology, offers DEBUG: "To detect, locate, and correct faults in a computer program. Techniques include use of breakpoints, desk checking, dumps, inspection, reversible execution, single-step operation, and traces."

Survey Results

For the most part, the survey respondents were consistent in their definitions, especially for validation and debugging. Verification and testing were murkier subjects, often assumed to be the same thing.

Comments on verification included: 1) quick and dirty check, 2) the systems testing portion of the SDLC, 3) it's definitely a Department of Defense word, not commonly used, 4) identifying that something is correct, 5) creation of test data for test plan, 6) making sure a program does what [it] says, the results are accurate and stand up to the specs.

Validation was consistently related to the SDLC – it "is" requirements, specifications, development, verification, and user acceptance. Respondents said validation was: 1) thorough documentation of the SDLC, 2) formal, accomplishes the specs for inclusion in the research reports, 3) inclusive of change control and retirement, 4) to ensure compliance with regulations, making it legal, 5) formal test plan, test data, system verification, creation of valid protocol, user acceptance. Recurring words were: reproduce-able, efficacious.

Testing yielded the most vague responses. Some felt it was synonymous with verification or that it is part of validation. Other comments: informal testing during the development phase, putting the program through the ringer, comes in many disguises (unit, integration, systems, etc.). "If someone asks me to test something, I ask them if they want it verified or validated."

Respondents said a program wasn't ready for validation if it still had bugs in it. Debugging was said to be "testing of logical systems to ensure they're not subject to errors of certain a priori identified types." Generally, debugging was felt to be the fixing of errors in the development phase of the SDLC.

Zero Defect Programs

[...Lubarsky's Law of Cybernetic Entomology: There's always one more bug....]

How are errors introduced into a program? Naturally, no one intends to produce flawed code. "Bugs" just turn up mysteriously to wreak havoc when we least expect it. However, we should recognize "bugs" for the errors that they are and attempt to produce code with zero defects. It is not enough to assume without verification that the work of good programmers will be correct. The slightest typo in the code or misunderstanding of the user requirements can cause serious errors during later execution.

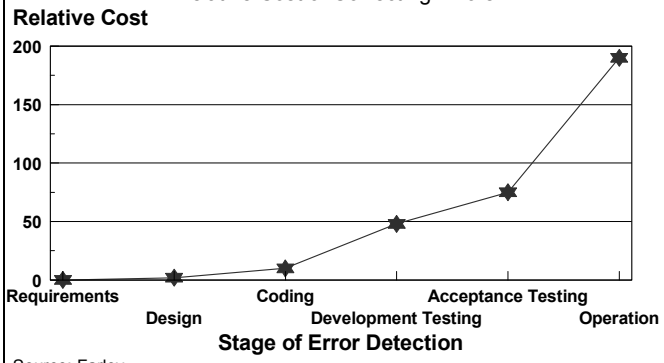
The purpose of testing is to identify any errors and inconsistencies that exist in a system. Debugging is the art of locating and removing the source of these errors. Together, the testing and debugging tasks are thought to cost 50% to 80% of the total cost of developing the first working version of a system. Clearly, any techniques that will facilitate these tasks will lead to improved productivity and reduced costs of system development.

We must broaden our philosophy of testing to go beyond the syntax check. As shown in Figure 1, the cost of correcting errors increases exponentially with the stage of detection. An error in a large system discovered after release of the product to the user can cost over two hundred times the cost of correcting the same error if it were discovered at the beginning of the system's development. Costs to fix these later errors include changes in documentation, re-testing,

and possible changes to other programs affected by the one in error. While costs of correcting errors in smaller systems or segments of code do not increase so dramatically over time, early testing can still reduce total costs substantially and improve system accuracy and reliability.

Figure 1.

Relative Cost of Correcting Errors



Source: Farley

Testing and debugging are recommended at all stages of the software development life cycle: determination of functional requirements; systems or program design; coding, unit testing, and implementation; and validation of results. Many features of the SAS System can facilitate testing and debugging, resulting in time saving, programmer-efficient, cost-effective program design into your daily routine. Incorporation of these techniques into the design, from the smallest module to the overall system, will result in a higher quality product delivered in a shorter time, not to mention reduced stress and increased self-esteem among the programming staff.

The Software Development Life Cycle

The software development life cycle is a formal segmentation of the evolution of a software product, from the glimmer of an idea through development to final acceptance by the users. A traditional model of this life cycle is shown in Figure 2.

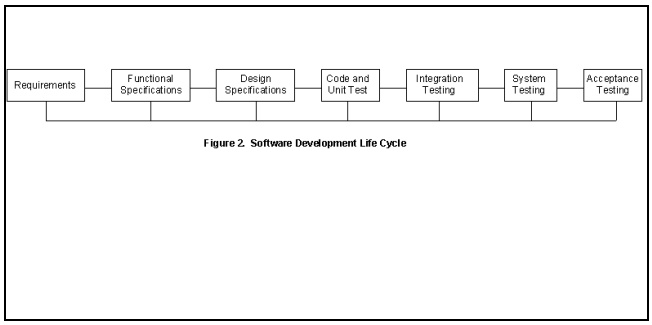


Figure 2. Software Development Life Cycle

Figure 3 shows the model utilized by Parke-Davis.

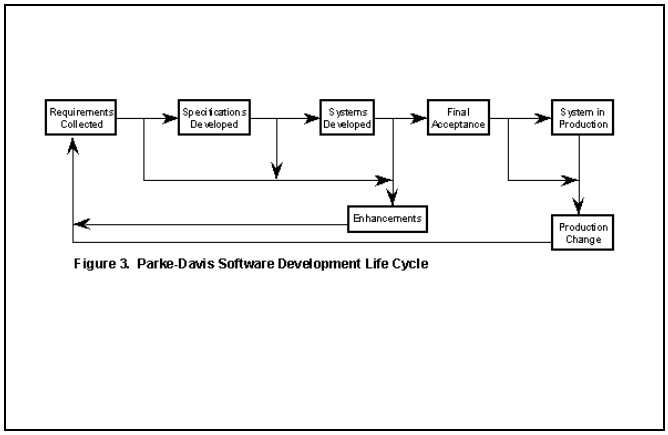


Figure 3. Parke-Davis Software Development Life Cycle

The life cycle begins with a need expressed by a future user of the system. These requirements specifications become a statement of needs given from the user's point of view, in the form of a detailed statement of what the system is supposed to do.

Obviously, the successful preparation of the requirements document depends on clear communication between the requester and the programmer. For example, does the manager want a separate listing by patient ID and by treatment group, or a single listing with totals by patient ID within each treatment group? Include frequent, detailed, interactive communication with the user to verify expectations. Remember that during these discussions the user and the programmer are both learning about and determining the product, refining their understanding of the requirements by asking many questions, and documenting changes.

Once all parties agree on WHAT is to be done, the programmer prepares the *specifications*, a detailed plan of HOW the computer system will achieve these goals. It is at this stage that plans for testing and implementation should be developed.

After all parties have agreed to the specifications, and reviewed and accepted the various plans, *implementation* of the system can begin. Each stage of development should include a testing phase, so problems can be isolated early in the cycle where corrections are less costly. As individual modules are accepted as error-free after initial testing, they can be tested in increasingly larger units (routines, programs, systems) until the entire system is integrated and ready to be released to the user.

System release generally includes turning over documentation, files, and programs to the user who will test the system further. It is only rarely that the cycle ends here; some level of system maintenance

will be required over the remainder of the life cycle of the software.

Requirements

Manubay's Laws for Programmers:
1) If a programmer's modification of an existing program works, it's probably not what the user wants;
2) users don't know what they really want, but they know for certain what they don't want.

Requirements will include not only a statement of the overall goals of the system and descriptions of input and output records and reports, but details that may have been overlooked in initial discussions. If possible, the user should supply coding schemes, including missing value or other special codes, valid ranges for variable values, key algorithms to be used, and any logical tests that should be included (e.g., check to see that someone who claims to never have smoked has no information on amount smoked). The user often can supply test data, actual or textbook data, which is invaluable in checking your understanding of the requirements.

Users who are unfamiliar with data processing procedures or programming usually will not necessarily volunteer this information. It is therefore in your own best interest to ask questions, lest you discover too late that your design was based on faulty assumptions. Often at this stage a prototype can be developed to bring necessary changes in requirements or functional specifications to light early in the cycle. Working with the user to develop and modify the prototype can be an efficient way for everyone to refine the needs and expectations of the system. *Walk-throughs* of the requirements will also help clarify the details. Try to anticipate future needs of the system to avoid design changes later.

As an example of a problem that could have been avoided by more interactive discussion, a small system designed at the National Cancer Institute to analyze food frequency questionnaire data changed any missing value code to '.'. It was only after the initial data analysis that the scientist decided to distinguish between the missing value responses of "ate the food, but don't know how often" and "don't know if he ate the food". Every program had to be changed to accommodate these new codes. This problem could have been avoided by using multiple missing value codes (e.g., '.A', '.B') which could easily be combined whenever necessary.

Documentation of the functional specifications should be approved by both the user and systems designer, and should provide a link back to the original requirements specifications received from the user. This trace to the user's requests will not only serve as a reminder of what was required and why, but may point to possible modifications of the requirements that can reduce programming time and complexity and possibly improve the product. The documentation also allows verification that the functional requirements will meet the objectives of the original request and that the system end products satisfy the functional requirements.

Specifications

After the functional and requirements specifications have been reviewed and accepted, the actual design of the programmer system can begin. Again, the *specifications* detail how the requirements are to be accomplished. Additional testing strategies should be incorporated into this design to further save time, money, and effort at later stages of development or system use.

Often one of the first tasks in system design is to do a flow diagram of the required processes, including the input/output flow of the data. This design diagram can identify tasks that can be done in a single DATA or PROC step, repetitive tasks that could be coded as macros, and independent modules that could be coded by different programmers. It also will identify subsets of the data that could be used instead of the master file to reduce processing time. Flow diagrams will also identify critical module interdependencies (e.g., can program B read the file written out by program A?).

The specs document should identify each "process" that must be performed within the program or system; one process will correspond to one DATA or PROC step. The document should include for each process: 1) a description of what is being accomplished in that step, 2) pseudo-code stepping through the required logic, using SAS data set and variable name references, 3) identification all input and output files, and 4) lists of data elements in each file. At this point, the design specification begins to resemble a SAS program: comments, code, data flows, and data set contents. The SAS system provides us with many tools to implement these ideas through the use of macros, procedures, and DATA steps. Its "English-like" syntax, when coupled with the use of meaningful variable names or formats, can facilitate a line-by-line translation directly from the design specifications.

Once the basic design of the system has been sketched out, specifically what should be included to facilitate testing in later stages? First, whenever any data (variables, records, or files) are modified, provide documentation of the change and a means to recover

the original data if necessary. The use of PROC CONTENTS and PROC PRINT (at least for a portion of the file) for newly-created or modified files will also provide a link back to the original specifications.

Secondly, plan to show intermediate results frequently, again by the use of PROC PRINT or PUT statements written to the SAS log during DATA step execution. This will simplify debugging later by allowing verification of small sections of code and can point to sources of error before the program aborts. Checking of intermediate results by the program can also lead to efficient "error trapping", that is retaining control of processing by the program when invalid data are encountered.

For example, checking to see that a denominator is not zero prior to division allows the program to take corrective action, such as substituting a small value for the zero or dropping the observation altogether. This approach is preferable to a program abort in the middle of a long, expensive run. Printing intermediate results also facilitates the checking for numerical accuracy of complex calculations.

Any repetitive task identified by the system design should be implemented as a macro or subroutine. The repeated use of code rather than coding repeatedly reduces the possibility of coding errors to a single routine and facilitates debugging and later maintenance of the module. Often a generalized macro can be executed conditionally, i.e., depending on the value of an input variable or pre-specified parameter. If these macros are useful in several applications, a macro library can be created to easily share the code. Similarly, format libraries may be created and shared by all programs in the system to ensure the consistency of the categorization of variables.

Finally, certain coding conventions can be adopted to ease the debugging and program maintenance tasks. The freeform syntax of the SAS System allows the use of indentation to set off DO loops and code which is executed conditionally. Spaces and comments may be used freely to make the program easy to follow. See coding conventions section.

Testing

[...Gilb's Law of Unreliability: Undetectable errors are infinite in variety, in contrast to detectable errors, which by definition are limited....]

Imagine yourself as the eight-year-old consultant hired by Microsoft to "break" their software. Or, you've just been hired as a tester of new SAS products. The Pentium chip incident still haunts. You are aiming for the User Feedback Award of the century. You are Inspector 12.

Begin with the assumption that products are not thoroughly tested. You are going to have a plan of attack. You ***will*** find those bugs. Use the same philosophy in your day-to-day programming.

What passes for "testing" is all too often just the processing of a single sample data set. Worse yet, the correct results for this data set are often unknown. What does this prove? Certainly NOT that the system is error-free. Unless test data sets have known conditions and predictable outcomes, we can't be sure what (or if) they are testing.

So, what should we be testing? First, as many conditions expected from potential input data should be run through the system as possible. These data should include extreme values (largest and smallest expected values for variables), bad data, and missing data. A large volume of data should be processed to examine time and storage requirements. Output should be verified by independent methods (e.g., by hand calculations).

Less used, but equally necessary, is structural testing of the code itself. For example, each pathway through the code is tested, checking that each subroutine can be reached and that the code is as efficient as possible. This type of testing can identify logical dead-ends, infinite loops, and code that would never be executed.

Finally, any algorithms or mathematical calculations need to be checked for correctness, accuracy, and potential over- or under-flow. For example, has the correct version of the chi-square statistic been specified? Is the algorithm used the one least susceptible to round-off errors?

Tests such as these should be conducted on every part of the system, from the smallest unit of code to the links between DATA and PROC steps and other programs. Structural testing of each small unit as it is completed will limit the potential sources of error in the more comprehensive tests to links between the smaller units. Although there is disagreement among practitioners about the order of testing (e.g., top-down vs. bottom up), everyone agrees that all parts of the system must be tested individually and as an integrated system.

Test data can easily be generated within the SAS system. An OUTPUT statement within nested DO loops (code will be included in handout) will create a

data set where certain variables are systematically varied. Random number generator functions are available for the more common statistical distributions to create "typical" data for variables with known statistical properties. Use of the OBS= and FIRSTOBS= options with the SET statement allows testing of small subsets of an actual input file.

Although it is impossible to prove that a system is totally error-free, inclusion of these general types of testing along with careful documentation of the input and output data for each test should result in a high degree of confidence in the integrity of the system.

Good testing requires commitment and acceptance, knowledge and practice, as well as forethought and preparation.

Myths of Testing

One web source identified at least two myths of TESTING. The first is that size matters and the second that testing is intended to show that software works.

All elements of the testing plan are applicable to all programs regardless of size. The depth and breadth may vary, the testing may not take as long or as many resources for smaller projects, but the same testing scenarios should exist for all efforts.

Software testing is really a destructive act aimed at confusing, maiming, breaking, crashing, and gagging the program. Validation then becomes the process of producing the final "runs" and documentation that satisfy the user.

Types of Testing, Real and Imagined

AGGRESSION TESTING: If this doesn't work, I'm gonna kill somebody.
COMPRESSION TESTING: []
CONFESSION TESTING: Okay, Okay, I did program that bug.
CONGRESSIONAL TESTING: Are you now, or have you ever, been a bug?
DEPRESSION TESTING: If this doesn't work, I'm gonna kill myself.
EGRESSION TESTING: Uh-oh, a bug... I'm outta here.
DIGRESSION TESTING: Well, it works fine, but can I tell you about my truck...
EXPRESSION TESTING: #@%^&*!!!, a bug.
OBSESSION TESTING: I'll find this bug if it's the last thing I do, even if it kills me..
OPPRESSION TESTING: Test this now!
POISSION TESTING: Alors! Regardez le poission!
REPRESSION TESTING: It's not a bug, it's a feature.
SECCESSION TESTING: The bug is dead! Long live the bug!
SUGGESTION TESTING: Well, it works but wouldn't it be better if...

Debugging

The Last Law: If several things that could have gone wrong have not gone wrong, it would have been ultimately beneficial for them to have gone wrong.

Types of Errors: Once we have designed the system with these general tests in mind, what specific types of errors should we anticipate and how can the SAS system help to avoid them? Following a classification scheme presented by Beizer, potential errors can be categorized as: 1) misunderstandings about the requirements specs; 2) interaction between subroutines or programs and other components of its operating environment; 3) logic or numeric errors; 4) data errors; and, 5) simpler coding errors.

The first three function-related errors are misinterpretations or miscommunications of the detailed tasks required by the user. Frequent, detailed communications and walk-throughs with the staff responsible for design and implementation are necessary to avoid these problems. Later testing can verify that we are building the product right, but testing cannot determine whether we are building the right observation's old and new values. If only a few (or none) of the observations should have been changed, PROC COMPARE could be used to compare data sets before and after a routine; this would be more efficient than PROC FREQ for variables with ht product.

For each test planned, we need to decide upon a measure of the correctness of the results. For example, results of a "variable re-coding" routine can be verified by a PROC FREQ cross-tabulation of the old and new codes. Comparison of the means of the old and new distributions, e.g., by a PROC UNIVARIATE, is not the same as comparison of each of many possible values.

System-related errors arising from a misunderstanding of the hardware or operating system interfaces rarely occur when using the SAS system, since these interfaces are transparent to the user. One source of this type of error might be in writing a device driver for use with SAS/GRAPH.

It is possible, however, that the programmer may misunderstand certain assumptions or defaults of the software system. For example, not knowing that missing values on the right side of the assignment statement propagate to the left will lead to missing totals if a single component of the sum is missing. It is critical to understand how different procedures and DATA step (built-in) functions handle missing values.

Mistakenly thinking that ORs and ANDs are processed left to right in an expression could lead to a logical error, since AND takes precedence.

These errors can easily be avoided by not taking anything for granted -- check for missing and invalid values, include every possible value in IF-THEN-ELSE clauses, and include parentheses for clarity in numeric or logical expressions. Finally, a common error occurs when using the FIRST. and LAST. features in conjunction with a conditional DELETE statement (e.g., IF SEX=MALE THEN DELETE) -- if the first (or last) occurrence of the BY variables is deleted prior to the FIRST. statement, the FIRST. condition will never be recognized.

Input and output data may be printed using PROC PRINT or writing PUT statements to the log in the DATA step to determine whether the data is of the format expected by other programs or DATA steps. Several PROCs, such as SUMMARY and MEANS, can create output data sets available in later steps. However, variables in the original data set that are not included in the CLASS, VAR, or BY statements will be dropped from the newly-created data set; these may be retained by including them as ID variables or by merging them from the old to the new data set.

Occasionally, we request tasks that are beyond the allowable limits of memory or CPU time at our installation. Solutions include processing subsets of the data, reducing data for manageability, deleting unnecessary variables, or using a less taxing computation method. Errors arising from requesting a cross-tabulation of variables with too many values can be avoided by using PROC SUMMARY instead. Even when our task runs within system limits, we should strive for time- and storage-efficient code to minimize cost. Whenever possible, a single DATA step should be run to create new variables to be saved for later PROCs rather than recreating the working file each time a PROC is run.

PROC SORT is one of the procedures most likely to cause CPU time and storage errors, because of the need to write and read sort work files. One solution is to use the KEEP statement to access only the variables necessary for the task. Similarly, the number of variables in the BY statement (i.e., the sort key) should be kept to a minimum to conserve resources. Sorting by all possible variables is not only wasteful but also unnecessary; since the combination of fewer than five variables is usually enough to uniquely identify a record.

Comments

Since we were students and junior programmers, we have been hearing the virtues of commenting programs, and yet comments are still the rarest of conditions in legacy or ad hoc code. The most effective way to think about commenting code is to realize the impact on testing and validation. Every comment saves additional time in debugging logic and/or walking through code.

Comments can be written in three ways:

COMMENT statement	**(keyword statement)**
***** text..... ;**	
/* text... */	**(can be imbedded in statements)**

A potential approach is to include the *specifications* throughout the program in the form of comments. This is not necessarily a substitute for a standalone spec document, but this method has been shown to be a powerful tool in testing and validation.

Documenting All SAS Data Sets

Document your SAS data sets using the self-documenting features of SAS. PROC CONTENTS or the CONTENTS statement invoked with PROC DATASETS completely describes the observations and variables, attributes, labels, formats associated with the data. This implies that the use of the OUT= option within procedure calls creates intermediate data sets enhances the documentation provided by the procedure output. The SAS logs from test runs are outstanding documentation for test plans.

Coding Conventions

There have been many SUGI papers and posters on the elements of programming style and coding conventions for SAS programs. Standardization of certain elements of all programs within your organization so improves the readability and maintainability of programs that, again, the testing and validation processes are greatly optimized.

Past SUGI proceedings should be searched for definitive presentations on programming style (Frank Dilorio is a standout author on this topic). Consider grouping declarative statements for easy reference: LENGTH, RETAIN, array definitions. Isolate non-executables for easy testing: drop, keep, label, attrib, format, label. Use meaningful variable names and data set names for clarity and continuity. Adopt a macro naming (%macro __debug) convention. Use variables labels with procedure output for validation documentation and to enhance the meaning of output.

Develop a department style for comments and program header information. Using spaces, blank lines and indentation for readability and maintainability.

These suggestions only scratch the surface as far as developing a style, but the importance of the concepts can be seen for testing and validation.

Use of the SAS Log

Process, data, and coding errors are mistakes or omissions in our DATA step coding. Syntax or other fatal errors, such as array-subscript-out-of-range, cause an error message to be printed and processing to stop. However, much useful information can be gleaned from reading the NOTEs in the SAS Log -- these are warning or informational messages that do not stop processing although they may point to conditions that affect the accuracy of program results. For example, numeric over- or under-flow, taking logarithms of zero or negative values, uninitialized variables, invalid data, insufficient format widths, hanging DOs or ENDs, missing values, and character-to-numeric or numeric-to-character conversions are all conditions that generate NOTEs in the log. In addition, checking the number of observations and variables reported after each DATA step can point to errors in DROP, KEEP, DELETE, BY, OUTPUT, or MERGE statements or subsetting IFs.

The SAS log can be enhanced by writing PUT statements from the DATA step to create customized messages trapping data errors or special conditions encountered or to trace the logical flow within the DATA step. The _INFILE_ and _ALL_ options of the PUT statement can be used to print the raw data record or SAS variables (Program Data Vector), respectively. Examples include:

```
PUT 'ERROR: INVALID RANGE FOR SORT KEY'  _INFILE_ ;
PUT 'NOTE: SMOKER=YES BUT NO SMOKING HISTORY'  _ALL_ ;
PUT 'STARTED ROUTINE X';
```

The ERROR statement can be used not only to write messages to the log, but also to cause the _ERROR_ flag to be set to 0, thereby forcing a dump of the input buffer (if reading raw data) and/or the program data vector. An example of this technique in the DATA step is:

```
IF GROUP='TEEN' AND AGE > 19 THEN
    ERROR 'GROUP AND AGE DON"T MATCH';
```

PUT and FILE statements can be used when the raw data messages or data will be processed by another program or application. Multiple file names can be specified on the DATA statement allowing you to write different observations to different SAS data sets. Whenever a data error is discovered, for example, the observation can be written to an error file along with a character variable containing the error message. At the end-of-job, all of the errors can be printed together, rather than scattered throughout the SAS log. This technique also allows the error records to be sorted and printed by error type or identifier number to facilitate checking of the original documents. Alternatively, a file for each type of error may be created so that no sort is required to group records with like errors for printing. This technique (using multiple output files) can also be used to group the input data into a few categories (e.g., by sex and race), thus saving PROC SORT time.

Tracing Methods

While the PUT statements can document logical flow within a DATA step, other methods are required for tracing data flow within a program. PROC PRINTs can be used before and after a DATA or PROC step to verify processing of the input and to print intermediate results. The OBS= option is useful in limiting the amount of printout; the FIRSTOBS= option can be used to skip to a known trouble spot in the data set.

Similarly, PROC CONTENTS can be used between DATA or PROC steps to trace the data flow. This is a simple way to document the number of observations, the variables retained, and the attributes of the variables at each step.

Conditional Execution

Including the aforementioned PUTs, PROC PRINTs, and PROC CONTENTS in a SAS program will necessarily add to the length of the source program and to the output. Removing the extra code when testing is complete can be time-consuming and can introduce new errors into the code. Also, removing the code will guarantee that you will receive a change request for the program, requiring you to add back the extra statements for further testing. A better practice is to permit conditional execution of the testing/debugging aids. The simplest form of conditional execution is to use an IF-THEN structure for the PUT statements. For example, a variable DEBUG could be set to 1 or 0 at the beginning of the program to indicate whether or not the PUTs should be executed (i.e., IF DEBUG THEN PUT ...).

Conditional execution can be accomplished using the macro language or using external parameter files and %INCLUDE. For example:

```
%IF &DEBUG=DEBUG %THEN
    %DO;
        PROC PRINT DATA=TEST (OBS=500);
          VAR A B C;
        RUN;
    %END;
```

where DEBUG has been defined as a macro variable. This method can also be used to conditionally calculate intermediate statistics during testing using PROC MEANS or SUMMARY. These tools have been found to be so useful in testing and debugging that some companies have compiled several of them as generalized diagnostic macros which are stored in a macro library for company-wide use.

Modularity and Macros

Shared macros of course are useful beyond testing and debugging. Any tasks that are used repeatedly or in several programs are candidates for a macro. We have discussed earlier the advantages of this technique: guaranteeing that all programs perform the task in the same way, simplifying maintenance of the code (it only needs to be changed in one place), and reducing programmer effort. Format libraries achieve the same goal by standardizing variable categorization. Macros also provide a means of writing generalized programs for use by non-programming users who need only "fill in the blanks" in a macro call with their own variable names and parameters in order to run complex SAS programs. The Proceedings from SUGI conferences document many such macros.

Finally, the use of macros forces a modular structure onto the SAS code that has been shown (for any programming language) to reduce the probability of coding error. "Structured programming" techniques include a straightforward logical flow, without GO TOs, and the use of subroutines or macros for repetitive tasks. These programming techniques extend the modular structure of the SAS system, where the DATA and PROC steps define individual processes and many pre-coded subroutines and functions are provided.

A popular theme today stresses "re-use" of code. So, as you are writing, consider who might be "calling" your module or using your program as a model or "shell" for their own start-up work. There are also a variety of opportunities for "downloading" routines from bulletin boards or other code libraries. Some code will never die, and there are ramifications everywhere.

Discussion

Meskiman's law: There's never time to do it right, but there's always time to do it over.

Testing in a haphazard way, or not at all, can only lead to disaster and confusion. Planning for testing from the beginning of the systems design will result in program design that costs little in added programmer effort but pays off in higher quality work and less effort spent debugging and rewriting code. Being aware of potential program errors should point to ways we can improve our coding practices and produce a higher-quality product.

Afterword

"Dogs get mange. Horses get the staggers. Soldiers of fortune get malaria, and computer programs get errors. Without doubt, error is a disease endemic to software – and a congenital one at that. The infection sets in before the first page of code is written. No computer system has ever been born without it; its automation's answer to Original Sin."

Phil Bertoni, 1983

Acknowledgements

The author would like to thank Michelle Gayari, Jamie Colgin, Karen Malley, Jim Sundberg, Nancy Mathias, Steve Pohl, Bonnie Hartsuff, Rich Synowiec, Scott Kapke, Noel Mohberg, Russ Newhouse, Don Sizemore, Debbie Buck, Ala Sweidan, Lee Herman, Cindy Zender, Nancy Patton, Josh Sharlin, Michael Parkes, Nelson Pardee and Carol Brodbeck for sharing their experiences with validation and testing.

References

1. Boris Beizer. Software System Testing and Quality Assurance. New York: Van Nostrand Reinhold Company, 1984.

2. Richard Farley. Software Engineering Concepts. New York: McGraw-Hill Book Company, 1985.

3. SAS Institute Inc. SAS User's Guide: Basics, Version 5 Edition. Cary, NC: SAS Institute Inc., 1985.

4. SAS Institute Inc. SAS User's Guide: Statistics, Version 5 Edition. Cary, NC: SAS Institute Inc., 1985.

5. William Briggs. Quality assurance review and recommendations. Unpublished report. Capital Systems Group, Inc., Rockville, MD, 1986.