

IT'S NOT A BUG, IT'S A FEATURE!!

Neil Howard, MCI, Colorado Springs, CO

Linda Williams Pickle, National Center for Health Statistics, Hyattsville, MD

James B. Pearson, Jr., Hughes Network Systems, Gaithersburg, MD

Abstract

[...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 TEST-MY-CONCEPT or READ-MY-MIND. The SAS system offers a wide variety of features for dealing with syntax, logic, and data errors. This tutorial will address tips and techniques for leveraging these features and validating the output of your programs. The results will undoubtedly include reduced stress and bigger raises for SAS programmers, satisfied clients, quality programs that are reliable and maintainable, and accurate output!

Disclaimer

The text of this paper is somewhat theoretical and a tad philosophical, perhaps dry. But that shouldn't detract from the importance of the topic! It should also be noted that there will be an additional *handout* at the presentation that will focus strictly on SAS code, tips, tricks, and solutions.

Introduction

The quality of a programmer's work is evaluated by results. When results are incorrect or the program fails to meet expectations, consequences range from annoyed users to frantic calls in the middle of the

night. Use a fourth generation language such as the SAS System should raise the quality providing an improved environment for the programmer. The SAS system handles the coding detail at the bits and bytes level and provides libraries of pre-tested functions and procedures, freeing the programmer to concentrate on the data and on concepts. The responsibility for ensuring program quality remains with the programmer. Today's competitive environment demands that we focus on these issues. We will discuss testing methods and useful SAS system tools that will help us meet this challenge.

Bugs and Testing

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

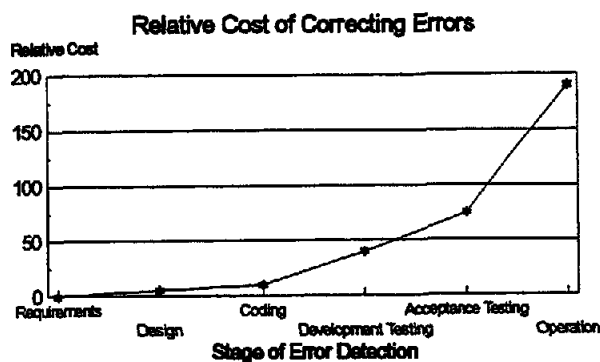
How are errors introduced into a program? Naturally, no one intends to produce flawed code. "Bugs" just turn up mysteriously to wreak havoc when we least expect it. However, we should recognize "bugs" for the defects that they are and attempt to produce code with zero defects. It is not enough to assume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

Beginning Tutorials

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, retesting, 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.



Source: Farley

Figure 1

We will describe techniques that can be useful for testing and debugging 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 and can introduce 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 general model of this life cycle is shown in Figure 2.

The life cycle begins with a need expressed by a future user of the system. This "requirements

specification", a statement of needs given from the user's point of view, is translated into a "functional specification", a detailed statement of what the system is supposed to do. As a simple example, a manager may ask for a list of employees showing salaries by individual, job title, and department. The functional specification may include the following tasks: read employee file to obtain employee name, salary, job code, department code; read Personnel Department job file and department file to obtain job title and department name; merge these data together by employee, using the Personnel files as a table lookup; sort file by department; sum salaries by department and print listing; sort file by job title; sum salaries by job title and print listing.

Obviously, the successful preparation of the functional specifications depends on clear communication between the requester and the programmer. For example, does the manager want a separate listing by job title and by department, or a single listing with totals by job title within each department? Much work can be saved in the early stages by frequent, detailed, interactive communication with the user to verify expectations. Remember that during these discussions the user and the programmer are both learning about the system, refining their understanding of the requirements by asking many questions.

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

After all parties have agreed upon 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 that problems can be isolated early in the cycle when corrections are less costly. As individual modules are accepted as error-free after initial testing, they can be tested in larger and larger units (e.g., routines, programs, systems) until the entire system is 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.

Functional Specifications

Functional specifications 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 be 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, either actual data or a textbook example, which is invaluable in checking your understanding of the requirements.

Users who are unfamiliar with data processing procedures or programming usually will not 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 specifications 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.

The moral of this story is that because requirements can change, functional specifications (and the subsequent design) should remain as flexible as possible. 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.

Design Specifications

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

Often one of the first steps in system design is to do a flow diagram of the required tasks, including the input/output flow of the data. This design diagram can identify tasks that can be done in a single DATA step, repetitive tasks that could be coded as Macros, and independent modules that could be coded by different programmers. It also will point to 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 design 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 the 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 allow nearly 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

Beginning Tutorials

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.

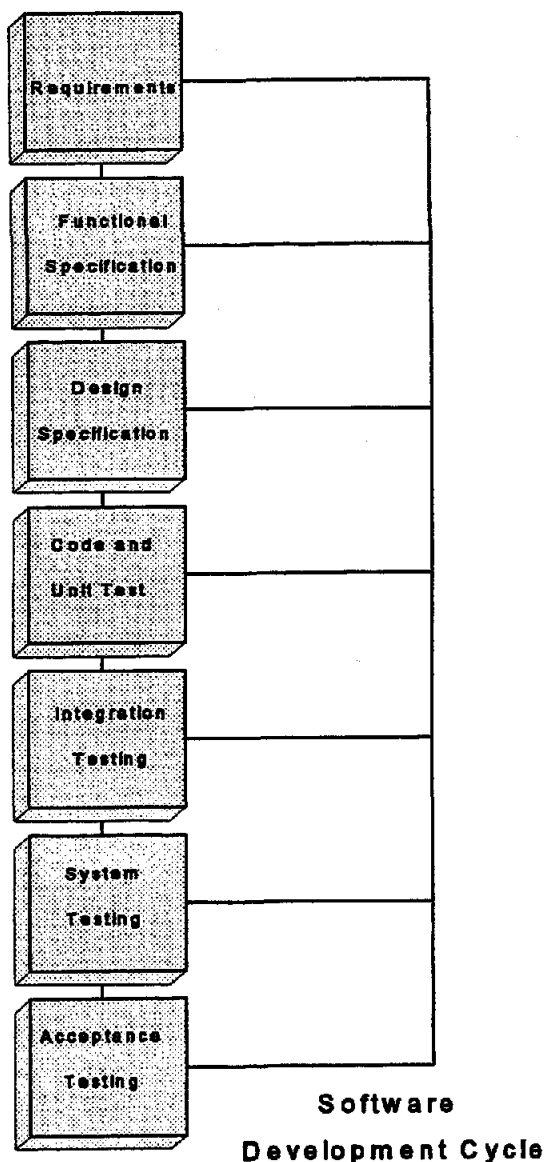


Figure 2

Secondly, plan to show intermediate results frequently, again by the use of PROC PRINT or the PUT statement. 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 to be preferred to allowing the program to abort, perhaps 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. For example, 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.

General Testing Techniques

Imagine yourself as the 8 year old consultant who tries to "break" Microsoft software or as a tester of new SAS products. Remember the Pentium chip incident. You are aiming for the User Feedback Award of the millenium. You are Inspector 12.

You will 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 own work on a daily basis.

What usually 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.

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. Don't make assumptions about how SAS functions or procedures treat missing values; it varies, and you need to check the manual and test for the results. 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 -- testing 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 will 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 steps and 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 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.

Potential 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, Figure 3 categorizes potential errors into five general types -- results of misunderstandings about the requirements specifications and about how a subroutine or program interacts with the other components of its operating environment, logic or numeric errors, data errors, and simpler coding errors.

The first three function-related errors are misunderstandings 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 product.

For each test planned, we need to decide upon a measure of the correctness of the results. For example, results of a variable recoding 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 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 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

Beginning Tutorials

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. Mistakenly thinking that ANDs and ORs 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 all numeric or logical expressions. Finally, a common error occurs when using the FIRST. and LAST. features along 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 PUT statements 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 data sets to be used for further processing. 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. For example, 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 unnecessary, since the combination

of fewer than five variables is usually enough to uniquely identify a record.

Use of the SAS Log

Process, data, and coding errors (Figure 3) are errors 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 *inspecting* 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 PUT statements to include customized messages noting 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 or SAS variables, respectively. Several examples are:

```
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 to cause the _ERROR_ flag to be set to 0, thereby forcing a dump of the input buffer (if reading raw data) and the program data vector (PDV). An example of this technique in the DATA step the statement is:

```
IF GROUP='TEEN' AND AGE > 19 THEN  
  ERROR "GROUP and AGE don't match";
```

The PUT statement with FILE option can be used as above when the messages or data will be processed

by another program or PROC. A better method of creating an error file is by the OUTPUT statement with multiple file names specified in the DATA statement. 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 # 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 of 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 (or Subroutines)

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. Many Macros of this type are presented at the SAS Users' Group meetings each year.

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.

Beginning Tutorials

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. Ramifications are everywhere.

Discussion

We have discussed methods of improving program quality by using simple tools available in the SAS System. Testing in a haphazard way, or not at all, can only lead to disaster and confusion. Planning for testing from the beginning of the system's 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 disease endemic to software — and a congenial one at that. The infection sets in before the first page of code is written. No computer system has ever been born without it; it's the automation's answer to Original Sin."

Phil Bertoni, 1983

SAS and SAS/GRAPH are registered trademarks of SAS Institute, Inc., Cary, NC.

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.
6. Federal Computer Week, June 19, 1995.

Figure 3.
A Classification of Potential Errors

specification error	ambiguous, incomplete, or contradictory specifications	"Calculate totals by department and job title" (separately or combined?)
function error	mis-specified or missing function	forgot to include calculation or intermediate statistics
testing error	mis-specification of intent or method of a test	generated test-data incorrectly
hardware/operating system interfaces	misunderstanding of how the hardware and operating system handles I/O with program	error in device driver specification
software system interface	misunderstanding of software system assumptions or defaults	a) assume variables initialized to "0", not "." (missing). b) misunderstood order of precedence for arithmetic or logical operators c) DELETE statement invalidates later LAST. or FIRST. statements
external interface	input/output to other programs or systems is in error	a) failure to check for bad input data b) writes wrong output record format
internal interfaces	error in input/output to other PROCs or DATA statements within same program	use of PROC SUMMARY to calculate totals dropped other necessary variables
resource management	task requires more than available CPU time or storage	a) too many variables to sort b) too many values for PROC FREQ c) not enough memory for PROC
numeric errors	incorrect algorithm or numeric over/under flow	a) failure to check for negative value before taking logs b) failure to trap large values prior to exponentiation
initialization	failure to set initial values or misplacement of initialization	a) failure to zero RETAINED totals b) DO I= 1 to 100; X = I/EXP(A*Y+B); END; (calculate EXP term outside loop)
control sequence	improper nesting of DOs, or error in path through code or in the DO condition	a) missing END statement b) GO TO non-existent label c) DO WHILE EXP(A*Y+B) < 0;
logic errors	error in logical code	IF 0 <= X <= 3 THEN Y = 'A'; ELSE IF 3 <= X <= 9 THEN Y='B'; (if X=3 should Y=A or B?)
data conversion errors	mis-specification of data attributes or length	character variables used repeatedly for numeric calculations
static data errors	constants mis-specified in code	self-explanatory
parameter error	error in data passed to subroutine or used for conditional execution	main routine codes sex as 1 or 2, subroutine looks for 1 or 0
coding errors	syntax errors, dangling loops, undeclared variables, typos in variable names, same variable name used for different sections of code	self-explanatory

adapted in part from Beizer