Good Programming Practice – Robustness

Introduction

Section content first added by: Greg Nelson

This article is a snapshot in time of the ongoing development activities of the Good Programming Practices Advance Hub page found here. As such this is not intended to represent a final set of recommendations or state of the art in programming. This is necessarily an evolving article. We are presenting with the aim of whetting the appetite of a larger community of participants and graciously and openly accept all contributors.

Concept Defined

The concept of software robustness is as old as the concept of programming. In Code Complete, Steve McConnell refers to robustness as:

“the ability of a system to continue to run after it detects and error”

The word robust, when used in the context of statistical programming refers to a program (or application) that performs well not only under normal circumstances but also under conditions that stress its original purpose and assumptions. People have often referred to this concept as:

- Error-proofing (Six Sigma)
- Test First Design (Object Oriented Design)
- Defensive Programming

Robustness is something that should be designed into software from the ground up; it is not something that can be successfully tacked on at a later date. Many of us have had to take on code that was developed by others where this was not built into the overall design and, well, nobody likes to fix other people’s mistakes.

Robustness is just one of the many facets of software development that make reading, modifying, updating and managing code easier. In fact, many of the philosophies shared throughout this article have direct roots, which can be traced back to the UNIX Philosophy. They are listed here for those unfamiliar with the concepts. As you look at these rules, you’ll see how these have been translated into various best practices by authors over the years in the SAS and R programming worlds.

1. Rule of Modularity: Write simple parts connected by clean interfaces.
2. Rule of Clarity: Clarity is better than cleverness.
3. Rule of Composition: Design programs to be connected to other programs.
4. Rule of Separation: Separate policy from mechanism; separate interfaces from engines.
5. Rule of Simplicity: Design for simplicity; add complexity only where you must.
6. Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.
7. Rule of Transparency: Design for visibility to make inspection and debugging easier.
8. Rule of Robustness: Robustness is the child of transparency and simplicity.
9. Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.
10. Rule of Least Surprise: In interface design, always do the least surprising thing.
11. Rule of Silence: When a program has nothing surprising to say, it should say nothing.
12. Rule of Repair: When you must fail, fail noisily and as soon as possible.
13. Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.
14. Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.
15. Rule of Optimization: Prototype before polishing. Get it working before you optimize it.
16. Rule of Diversity: Distrust all claims for “one true way”.
17. Rule of Extensibility: Design for the future, because it will be here sooner than you think.

The Rule of Robustness states that robustness results from transparency and simplicity. Software is transparent when a skilled programmer can examine its source code (i.e., the original version written by a human in a programming language) and soon comprehend how it works. It is simple when its operation is sufficiently uncomplicated that a programmer can visualize with little effort all of the potential situations that it might encounter. The more that programs have both of these qualities, the more robust they will be.

Another important tactic for creating robust software is to write general code that can accommodate a wide range of situations and thereby avoid having to insert extra code into it just to handle special cases. This is because code added just to accommodate special cases is often buggier than other code, and stability problems can become particularly frequent and/or severe from the interactions among several such sections of code.

SAS, R, Mathematic and other statistical software, have a major advantage over compiled and unreadable software as far as robustness is concerned. It is that the source code can be reviewed in its entirety, and thus it is easier to find and correct errors than when the code is compiled, thereby obfuscating the actual code or sub-routines.

Applying the Philosophy's Rule of Composition (i.e., to design programs so that they can be connected to other programs) also helps result in robust software. This is because it leads to small, modular programs that are easier to comprehend and to correct than are larger ones that attempt to do many things. It is also because input received from other programs (as contrasted to that from humans) can be particularly effective for stress-testing software and thus for helping to provide tolerance for unusual and large inputs.

Importance of Rubust Software

While it seems inconceivable that the importance of writing robust programs needs to be clarified, we will do so in brief.

- **Quality-critical applications**. Applications where quality is critical, such as clinical trials, can benefit from a rigorous approach to testing and data completeness checking
- **Efforts to create highly reusable macros or include files.** A robust approach helps ensure coverage of different usage scenarios, and provides a lot of sample code for how the reusable modules are called.
- **Code that will have a long life**. Any code that is going to be maintained for a long time may benefit from a suite of automated tests, because it reduces the risk and testing time associated with enhancements.
- **Complex applications with many interdependent modules.** Robustness helps decrease overall debugging time, because individual modules are tested independently before integration.
- **Improved information flow.** Where there are a number of programmers working together, robust programming techniques such as test-first-design help document in the requirements into the code.
- **Improved productivity.** While designing programs with robustness in mind, you make your job easier as the programmer and more importantly, anyone who comes behind you and has to modify or test your programs.
- **Fewer defects.** Because we design in quality, there are studies that show we have fewer defects when we design in robustness.

Related Concepts

The world of software engineering is a broad and complex eco-system. From the early days of low-level machine languages, there has been a desire to insert standards and methodologies into the development process. In the global healthcare world, we all have seen SOPs that outline the software development life-cycle, test and validation and are acutely aware of quality assurance processes present in the organizations we serve.

We don't discount these intentions in any way. We do, however, realize that having a documented process such as an SOP does not replace thinking of better ways to solve problems and make our programs out-survive our tenure or their intended purpose.

General Design Patterns

Test First Design

Test-Driven Development (TDD) reverses the normal process of programming. Instead of writing some code and testing it, in TDD you begin by writing a test, then write the code to make the test pass. This backward approach is the inspiration for a paper that we will refer you to a SUGI 31 paper entitled Drawkcab Gnimmargorp: Test-Driven Development with FUTS.

In general, the desire is to build in test cases that could be automatically run every time the program is executed. The concept of writing test programs within your code as you built your application comes from the Test-First Design principles discussed in software development circles – primarily in the Agile methodologies such as eXtreme Programming.

Interestingly, this test-driven methodology is most popular among the practitioners of Extreme Programming (XP), which is more widely known for informality than formality.

Test-First-Design encourages us to:

- First, write the simplest code that could possibly work with a small set of data.
- Then, once the tests are running green, take stock and consider whether any of our code could be improved by refactoring (fancy word for revising).
- If there are more features to implement, start again by creating another test.
- Finally, test the entire program with a full set of data for robustness.

In the SAS programming world, we have the Framework for Unit Testing SAS (FUTS). This concept is simple: when the conditions that the "test" was evaluating are NOT true, then an exception is fired and any number of things could be communicated to the programmer (or end user). In our world, these exceptions are referred to as "events". An event could be as simple as knowing when a dataset was not readable or had less than a desired number of observations. Events could also be triggered when certain business rules might not have been true. For example, you might have a rule that prints out a report whenever there are serious adverse events; otherwise, your program prints the standard report. Instead of coding complicated if-then statements in macros, we can use the automated testing environment to handle these conditions for us.

FUTS was developed so that any programmer could say what tests or preconditions should be true in any program. When they were true (e.g., my dataset has >=0 observations), the test was executed and if conditions were satisfied, continued. When the tests failed (dataset was not present or had < 1 observation), the test would fail and trigger an event.

For some real world examples of this approach in the Pharma/Biotech industry, we will refer you to the following papers:

- *Automated Testing and Real-time Event Management*: An Enterprise Notification System. by Nelson, Greg, Danny Grasse, and Jeff Wright
- *SASUnit: Automated Testing for SAS*. by Gregory S. Nelson
- *SCLUnit: SAS/AF implementation of the xUnit unit testing framework*

Defensive Programming

Defensive programming is another technique intended to ensure that your program will continue to operate despite how it is used or how the data changes. We have all seen our programs take on a life of their own after we released it into the real world and someone comes back to us and says, "it doesn't work for me". The idea can be viewed as reducing or eliminating the prospect of Murphy's Law having effect. Defensive programming techniques are used especially when we don't really have a good sense of how someone is likely to use it – or more directly in the world of statistical software – we don't know what the data might look like every time it is run.

Wikipedia defines Defensive Programming as an approach to improve software and source code, in terms of:

- General quality - Reducing the number of software bugs and problems.
- Making the source code comprehensible - the source code should be readable and understandable so it is approved in a code audit.
- Making the software behave in a predictable manner despite unexpected inputs or user actions.

A sampling of the publications that have been published in the SAS and R worlds that speak specifically to defensive programming include:

- Defensive programming techniques by John Woods & Jennie McGuirk
- Basic Defensive Programming Techniques by Baoxian Lan & Daniel Tsui
- SAS Programming in the Pharmaceutical Industry by Jack Shostak
- Robust Programming Techniques in the SAS® System by Alice M. Cheng

Error Proofing (Six Sigma)

Error proofing, or mistake proofing as it is often referred to in Six Sigma, refers to the implementation of fail-safe mechanisms to prevent a process from producing defects. This activity is also known by the Japanese term poka-yoke, from poka (inadvertent errors) and yokeru (to avoid). Although this common-sense concept has been around for a long time, it was more fully developed and popularized by Shigeo Shingo in Japan. The philosophy behind error proofing is that it is not acceptable to make even a very small number of defects, and the only way to achieve this goal is to prevent them from happening in the first place. In essence, error-proofing becomes a method 100% inspection at the source rather than down the line, after additional value has been added (wasted). Achieving extremely high levels of process capability requires this type of focus on prevention rather than detection.

This is similar to defensive programming and test first design.

Error Checking Versus Error Proofing

In writing SAS programs, it is often advantageous to protect yourself from unexpected inputs such as out of range data, format change, macro variables or changes in the environment in which the SAS program executes. Error checking is done posthumously – that is, after the fact; whereas, error proofing is done a priori, before the fact.

This is an important distinction when considering how you want to design your program. It is our contention that you should error-proof your code, not simply report the errors.

Data Driven Programming

Data-driven programming is a programming methodology in which the program statements describe the data to be matched and the processing required rather than defining a sequence of steps to be taken. In SAS, this is used quite often and can take on the form of having parameters supplied through metadata, formats or table lookups. As you may have seen in your environments, programs which hard-code a list of values can easily be rewritten with data-driven programming. This method replaces the hard-coding of values and instead abstracts the lookup to some external source.

Data driven programming is perhaps one of the most widely documented approach to writing robust programs and some authors have included:

- *Using Metadata and Project Data for Data-Driven Programming* by Brian Varney
- *Parameter Gathering Techniques for Data Driven Programming* by Renato G. Villacorte
- *Let SAS® Write and Execute Your Data-Driven SAS Code* by Kathy Hardis Fraeman
- *Accessing SAS® metadata and using it to help us develop data-driven SAS programs* by Iain Humphreys
- *SAS Programming and Data Analysis: A Theory and Program-Driven Approach* by Leonard C. Onyiah
- *The little SAS book: a primer* by Lora D. Delwiche, Susan J. Slaughter
- *Your Place or Mine: Data-Driven Summary Statistic Precision* by Nancy Brucken

Proactive Alerting and Notification

Extrapolating on the **Rule of Repair** (When you must fail, fail noisily and as soon as possible), the concept of proactive alerting and notification has also been a popular topic in the SAS community – from entire systems that proactively alert users to problems to log parses which simply grep the log to cull out the warnings and errors that might be present.

The goal is to easily communicate to the programmer and/or end user when an error (i.e., failed test) occurred. Even when SAS ERRORS or WARNINGS occurred, we want to be able to signal an event and categorize them in such a way so that anyone could figure out where they occurred and in what context. Finally, by giving the programmer a way to categorize tests, she was allowed to centralize the testing and error codes. In summary, we wanted to accomplish the following:

- Provide a consistent logging facility for errors and exceptions (to support both the standard SAS log and a summary version)
- Ensure “auditability” of SAS programs that were run (who, what, when)
- Surfaced SAS “events” real time (even before the job was completed)
- An error or warning in the log doesn’t always give you the context in which the error occurred so we wanted a way to determine the “call stack” (to see which jobstream, job, program, and macro where the event occurred) of how programs were called – which provided context for errors and warnings
- Consistent way of checking “pre-conditions” without having to turn each program into a macro to perform conditional logic
- Provided a testing framework for developers so that they could “assert” test cases in their code

As of SAS 9.2, we now have the ability instrument our own programs using the SAS logging framework (Log4SAS.) this is documented in at support.sas.com and presented at SAS Global Forum 2008: SAS®9.2 Enhanced Logging Facilities.

Other references that speak to the proactive notification of errors include:

- *Automated Unit Testing for SAS® Applications* by David A. Scocca
- *Testing Your SAS Code Against Known Data* by Sharon Schiro
- *Making Music in SAS: Using Sound to Alert Users of Errors and Data Discrepancies* by David Fielding
- *Automated Testing and Real-time Event Management: An Enterprise Notification System* by Greg Nelson, Danny Grasse & Jeff Wright
- *You’ve Got E-Mail: Automatic Log Checking Via E-mail Notification* by Aaron Augustine, & Prasenjit Dutta

Platform Independence

Another method commonly used in SAS programming is the ability to run programs on a variety of platforms such as UNIX and Windows.

Here the task is to abstract all of the operating system specific items such as filenames, libname paths and system commands and utilize SAS built-in system macro variables to automatically detect which operating system and programmatically determine which path names are to be used.

Some of the authors that speak to this topic are:

- *Make Your SAS® Code Environmentally Aware* by Clarke Thacher
- *SAS®, Linux/UNIX and X-WINDOWS systems* by Gady Kotler
- *Mainframe Files On Your Windows Desktop with SAS/CONNECT®* by Michael G. Sadof
- *You Want This New Application to Run on Our VAX, PC and SUN Computers???!!!* by David Franklin
- *Cross Platform Application Development in SAS: Exploitation of Multi-Vendor Architecture (MVA)* by Gregory S. Nelson

SAS Programming Techniques

Section content first added by: Benjamin Szilagyi

In this section, we will apply some of the general techniques outlined above and related them to how you can implement these with SAS. While we recognize that many of the ideas presented here are in no way novel, we do cite original work when appropriate.

Generalised SAS Code

In later sections, we highlight robust techniques that can be used when code is running within the macro compiler in SAS. There are times, when you want to prevent or highlight issues in open code – that is code, which is running by itself and not part of a macro or larger application.

Tip # 1. In the final code, there should be no dead code that does not work or that is not used. This must be removed from the program.

For example, the /*and visitnum>1*/ is no longer in use and it needs to be removed for better readability:

```
        proc sort data=adam.advs out=vs;
                where saffl='Y' /*and visitnum >1*/ and trt1a in (1,2);
        run;
```

Tip # 2. Error Checking Code should be included in the program where possible

Code to allow checking of the program or of the data (on all data or on a subset of patients such as clean patients, discontinued patients, patients with SAE or patients with odd data) is encouraged and should be built throughout the program. This code can be easily activated during the development phase or commented out during a production run using the piece of code

For example, the following code will report the partial date data into the LOG window:

```
        data ae;
                set raw.ae;
                if length(compress(aestdt))=9 then aestdtn=input(aestdt,date9.);
                else if aestdt>' ' then put 'ERROR:  DATA PROBLEM: Partial Date issue for IF-THEN in AE
step. ' usubjid= aestdt=;
        run;
```

Tip # 3. Try to produce code that will operate correctly with unusual data in unexpected situations (e.g. missing data).

For example, it is important to exclude the missing data for lab normal range indicator flags in order to the common mistake of showing 'LOW' or 'HIGH' in the LBNRIND column for missing values of LBSTRESN:

```
        data lb;
                set raw.lb;
                if .<lbstresn < lbstnrlo then lbnrind='LOW';
                else if lbstresn>. and lbstnrhi>. and lbstnrlo>.
                          and lbstnrlo<=lbstresn<=lbstnrhi
                        then lbnrind='NORMAL';
                else if lbstresn>lbstnrhi>. then lbnrind='HIGH';
        run;
```

Errors, Warnings and Notes

As a number of contributors have noted (no pun intended!), paying attention to the log is critical. As SAS moves from a batch-execution environment, to “service” based, understanding where the various logs are critical.

Techniques described by Susan J. Slaughter & Lora D. Delwiche[xv] include understanding errors in the made because of:

- Syntax
  - missing semicolonuninitialized variable and variable not found
- Data
  - missing values were generated
  - numeric to character conversion
  - invalid data
  - character field is truncated
- Logic
  - DATA step produces wrong results but no error message.

Tip # 4. Whenever you look at the log, you should generally be on the lookout for these terms:

- error
- not referenced
- never been referenced
- not resolved
- is not in the report def
- has more than one data set with repeats of by values
- current word or quoted string has become more than 200
- observation(s) outside the axis range
- is unknown
- warning
- uninitialized
- cannot be determined
- extraneous information
- missing values were generated
- observation(s) contained a MISSING value
- Invalid arguments
- truncated to 32 characters
- overwritten by
- can't modify it at this time

Tip # 5. Use the MSGLEVEL=I option in order to have all informational, note, warning, and error messages sent to the LOG.

For example[xvi]:

```
OPTIONS MSGLEVEL=I;
        PROC SQL;
          SELECT *
            FROM AE
                    ,LAB
            WHERE AE.SUBJID = LAB.SUBJID AND TRT= 'A';
         QUIT;
```

The log displays the following notation:

```
INFO: Index Rating selected for WHERE clause optimization.
```

Tip # 6. It is not acceptable to have avoidable notes or warnings in the log (mandatory).

Reason: They can often lead to ambiguities, confusion, or actual error (e.g. erroneous merging, uninitialized variables, automatic numeric/character conversions, automatic formatting, operation on missing data...). Note: If such a warning message is unavoidable, an explanation has to be given in the program (mandatory).

For example, merging data with the length (20) of USUBJID from dataset DOSE1 and the length (30) of USUBJID from dataset DOSE2:

```
        data dose3;
                merge dose1 dose2;
                by usubjid;
        run;
        WARNING: Multiple lengths were specified for the BY variable pain by input data sets. This may cause
unexpected results.
        INFO: The variable trt1A on data set WORK.DOSE1 will be overwritten by data set WORK.DOSE2.
        NOTE: There were 200 observations read from the data set WORK.DOSE1.
        NOTE: There were 200 observations read from the data set WORK.DOSE2.
        NOTE: The data set WORK.DOSE3 has 200 observations and 20 variables.
```

Data Step

Tip # 7. Be careful when merging datasets. Erroneous merging may occur when:

No BY statement is specified (set system option MERGENOBY=WARN or ERROR). Some variables, other than BY variables, exist in the two datasets (set system option MSGLEVEL=I), S writes a warning to the SAS log whenever a MERGE statement would cause variables to be overwritten at which the values of the last dataset on the MERGE statement are kept). More than one dataset contain repeats of BY values. A WARNING though not an ERROR is produced in the LOG. If you really need, PROC SQL is the only way to perform such many-to-many merges. Reason: One has to routinely carefully check the SASLOG as the above leads to WARNING messages rather ERROR messages yet the resulting dataset is rarely correct.

For example, merging two lab datasets with USUBJID, VISITNUM, and LBTESTCD, a note 'statement has more than one data set with repeats of BY values' is presented in the LOG. This will result in mis-merge. You may want to check the data to see if it is caused by duplicate records or more variables need to be added to the BY statement.

```
        data lb3
                merge lb1 lb2;
                by usubjid visitnum lbtestcd;
        run;
        NOTE: MERGE statement has more than one data set with repeats of BY values.
        NOTE: There were 1000 observations read from the data set WORK.LB1.
        NOTE: There were 1000 observations read from the data set WORK.LB2.
        NOTE: The data set WORK.LB3 has 1000 observations and 28 variables.
```

Tip # 8. Route any error messages to the log window using the ERROR statement with a data step to detect unexpected data points or records.

For example, you want to ensure there is no record with missing AGE value. To give you alert, you can simply employ the ERROR statement:

```
        data dm;
                set raw.dm;
                if age=. then error 'ERROR: Missing AGE for ' usubjid=;
        run;

        ERROR: Missing AGE for USUBJID=999-099-001001
        STUDYID=DRUG A USUBJID=999-099-001001 SITEID=001 COUNTRY=USA AGE=. SEX=F RACE=5 ETHNIC=NOT HISPANIC
OR LATINO TRT1PN=0 TRT1P=A RANDDTN=. HEIGHT1=. HEIGHT2=. WEIGHT1=. ERROR_=1 _N_=1
        NOTE: There were 20 observations read from the data set WORK.DM.
        NOTE: The data set WORK.DM has 20 observations and 14 variables.
```

Tip # 9. When coding IF-THEN-ELSE constructs use a final ELSE statement to trap any observations that do not meet the conditions in the IF-THEN clauses.

e.g. ELSE PUT variable= to the log window

Reason: You can only be sure that all possible combinations of data are covered if there is a final ELSE statement.

For example, to ensure all records in the VS dataset are within the study window, you can use the PUT statement to trap those records which do not meet the IF-THEN clause and display them in the LOG window for data issue reporting.:

```
data vs1;
           set vs(keep=usubjid adt trt1day1);
           ady=adt-trt1day1;
           if .<ady<=7 then avisitn=1;
                else if 8<=ady<=14 then avisitn=2;
                else if 15<=ady<=21 then avisitn=3;
                else if 22<=ady<=28 then avisitn=4;
                else put 'WARNING: Out of Window Records: ' usubjid= ady= adt= trt1day1=;
        run;
        WARNING: Out of Window Records: USUBJID=999-099-001001 ady=36 ADT=2010-12-09 TRT1DAY1=2010-11-03
        WARNING: Out of Window Records: USUBJID=999-099-001002 ady=56 ADT=2010-02-02 TRT1DAY1=2009-12-08
        NOTE: There were 172 observations read from the data set WORK.VS.
        NOTE: The data set WORK.VS1 has 172 observations and 5 variables.
```

Procedures

Tip # 10. Always use DATA= in a PROC statement (mandatory).

Reason: It ensures correct dataset referencing, makes program easy to follow, and provides internal documentation.

For example:

```
        proc sort data=vs1;
                by usubjid;
        run;
        NOTE: There were 172 observations read from the data set WORK.VS1.
        NOTE: SAS sort was used.
        NOTE: The data set WORK.VS1 has 172 observations and 5 variables.
```

PROC Datasets

Tip # 11. At the end of the program or at strategic points, it is a good practice to use PROC DATASETS to delete unneeded data sets from the work library. This not only will improve performance, but more importantly will show the intention to the reader as well.

For example, the following code will only delete the datasets in the WORK library and format and macro catalogs will remain in the library:

```
        proc datasets lib=work kill memtype=data;
        run;
```

PROC SQL

- As with tip 3 code for missing values in case statements. Following from tip 9, Case actually forces a final else condition but missing values need to explicitly excluded from earlier conditions if they are not required.
- Use Coalesce (a specialized form of a case statement) when creating a table using a full join of two existing tables to ensure that no identifier variables have missing values.

```
proc sql;
 create table ae1 as
 select coalesce(ae.usubjid,dm.usubjid) as usubjid
        , brthdtc
        , sex
        , rfstdtc
        , aespid
        , aeterm
        , aestdtc
 from ae full join dm
 on dm.usubjid=ae.usubjid;
quit;
```

You would hope that in complete data, there would be no subjects in AE that were not in DM, but a report of partial or interim data may have these sorts of issues. The above code will still return missing if both variables are missing so use with caution or use a customized case statement if not using with primary keys. Try to avoid

```
select *
```

as well. Not specifically a robustness issue, but it makes it harder to work out which columns are present in the data, particularly when there are overwrite warnings.

Data Checks

<Looking for contributions from the community!>

Formats and Lookups

Tip # 12. When coding a user-defined FORMAT, include the keyword ‘other’ on the left side of the equals sign so that all possible values have an entry in the format.

Reason: A missing entry in a user-defined FORMAT can be difficult to detect. The simplest way to identify this potential problem is to ensure that all values are assigned a format. Note: This does not apply to INFORMATs. It could be more helpful to get a WARNING message when trying to INPUT data of unexpected format.

For example:

```
proc format;
           value avisit
           1 = 'Week 1'
           2 = 'Week 2'
           3 = 'Week 3'
           4 = 'Week 4'
       other='Out of Window'
           ;
run;
```

SAS Macros

<Looking for contributions from the community!>

Macro programs

<Looking for contributions from the community!>

Macro Variables

<Looking for contributions from the community!>

SAS Applications

<Looking for contributions from the community!>

Other Software Languages

Section content first added by: Benjamin Szilagyi

Tools and Vendors

Section content first added by: Benjamin Szilagyi

References

Adapted from http://en.wikipedia.org/wiki/Unix_philosophy

R. Kaufmann and D. Janzen, “Implications of Test-Driven Development: A Pilot Study,” Companion of the 18th Ann. ACM Sigplan Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, 2003, pp. 298-299.

H. Erdogmus, M. Morisio, and M. Torchiano, “On the Effectiveness of Test-First Approach to Programming,” IEEE Trans. Software Eng., Mar. 2005, pp.226-237.

S.H. Edwards, “Using Test-Driven Development in the Classroom: Providing Students with Automatic, Concrete Feedback on Performance.” Proc. Int’l Conf. Education and Information Systems: Technologies and Applications (EISTA 03), Aug. 2003;

Wright, J (2006), Drawkcab Gnimmargorp: Test-Driven Development with FUTS. Paper presented at the Annual Convention of SAS Users Group International, San Francisco, CA.
http://www.sascommunity.org/wiki/FUTS_-_Framework_for_Unit_Testing_SAS

Nelson, Greg, Danny Grasse, and Jeff Wright. 2004a. "Automated Testing and Real-time Event Management: An Enterprise Notification System." Proceedings of the Twenty-ninth Annual SAS Users Group International Conference, Montreal, Canada, 228-29.

Nelson, Greg. 2004b. "SASUnit: Automated Testing for SAS." PharmaSUG 2004 Proceedings, San Diego, CA, DM10.

http://en.wikipedia.org/wiki/Defensive_programming

http://support.sas.com/documentation/cdl/en/logug/61514/HTML/default/viewer.htm#a003250967.htm

Errors, Warnings, and Notes (Oh My) A Practical Guide to Debugging SAS Programs.
Sample 33602: Displaying Additional SAS Log Messages with MSGLEVEL=. http://support.sas.com/kb/33/602.html