

## SASUnit: Automated Testing for SAS

Greg Barnes Nelson, President and CEO  
ThotWave Technologies, LLC. – Cary, NC

### ABSTRACT

Data management is one of the cornerstones of SAS as a language and critical to pharmaceutical research and development. SAS programs that access, manage, analyze and report data are often taken from vast libraries of tools that are used and reused for consistency; their reuse in similar projects is another benefit. Over time, the number of potential uses of any one program or macro is challenged by the amount of time it takes to test, retest and validate these programs. As these programs become part of the production eco-system in a clinical research environment, their testability, robustness and manageability as "built-in" to the software development process become increasingly important.

This paper discusses a specific approach to “building in” a process for every program that monitors the conditions SAS programs encounter and proactively tests for and announces validation issues. We will explore the concept of automated tests through assertions, events and their attributes, event status management, and automatic notification of events to interested parties. These concepts are presented from the perspective of the SAS programmer and the systems analyst.

### BACKGROUND

This paper provides information on a system we created to help provide a layer of robustness and manageability for programs written in and around SAS. We built this system to support two major needs we perceived in the area of SAS applications: A way to create real time events coming out of SAS programs and a way to build into each program an automated way to assert or exercise test cases. This paper will focus on the latter as its primary theme. The topic of real time event management has been discussed elsewhere (Barnes Nelson, G. and Wright J. , 2004) and we encourage you to review that paper if you find yourself running applications that require real time notification of either business or technical events.

In general, the system we developed was a way for us to build in test cases that could be automatically run every time the program was 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 (XP; *see Test-First Design references at the end of this document*).

In the SAS programming world, we implemented this as an API (application programming interface). The API used a SAS macro interface, as these are well used throughout the industry and common as a framework for reusability. When the conditions that the “test” was evaluating were NOT true, then an exception was fired and any number of things could be communicated to the programmer (or end user). In our world, these exceptions

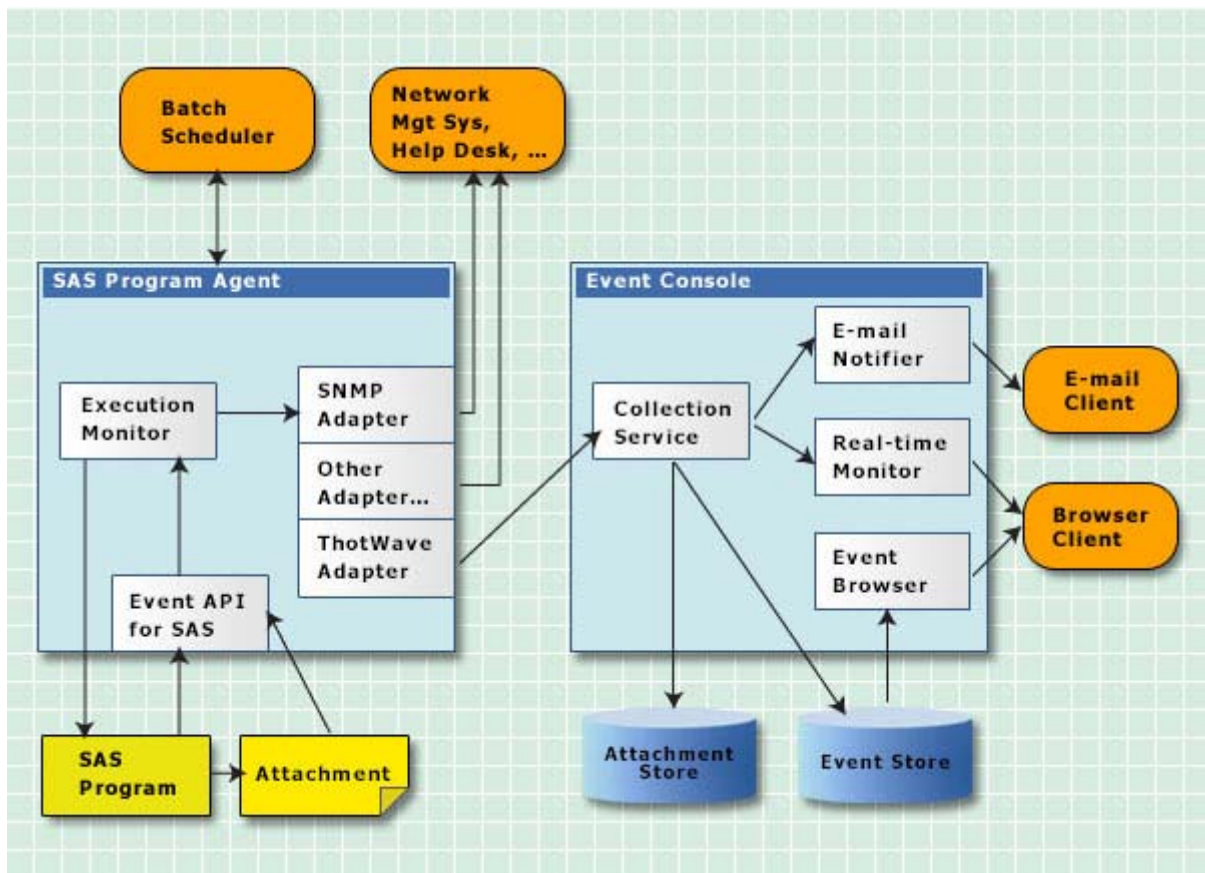
are referred to as “events”. An event could be as simple as knowing when a dataset was not readable or had less than a desired number of observations. Events could also be triggered when certain business rules might not have been true. For example, you might have a rule that prints out a report whenever 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.

The system 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  $\geq 0$  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.

In addition, we wanted 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 wanted 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

While our goals were fairly ambitious, we knew if we could accomplish them, significant value would be added to the applications we build for clients. In addition, we knew that much of the time and energy developers put into creating automated test scripts and error handling routines could be reduced, since most of the work would be rolled into this framework. An overall system diagram of how events are captured (from SAS programs), monitored and displayed is shown here to give you an idea of what we will be discussing in this paper.



### LOG MANAGEMENT: GETTING TO ISSUES BEFORE USERS DO

Traditional techniques for getting at the log are useful if you have a single driver program or a simple set of things to look for in your logs. Some techniques use tools like *perl*, *awk* and *sed* to post-process the log and parse out the ERRORS and WARNINGS. These have two fundamental problems we sought to overcome:

1. Occasionally the SAS program would take several hours to complete – we wanted to know about it much sooner if a problem existed.
2. Although an indication of an individual log with errors is convenient, this data didn't provide enough information about (a) their context and (b) other types of events – such as informational or business rule violations.

Part of the job of the Event System was to give us a really good way to surface the SAS log in a summary format that culled important pieces from the log and formatted them in a way that was particularly useful for the programmer.

### THE NEED FOR TESTING IN SAS PROGRAMS

To ensure application and system availability, developers need to detect or avert problems in their applications proactively. SAS does not have a facility built into the Data Step or Procs to manage the automated handing of

errors. Much of this responsibility falls on the developer to write code, ensuring that if something is supposed to happen (a dataset should have more than 0 observations, for example), it does so without problems.

Proactive Testing (and robust development) is all about:

- Detecting potential risk areas and evaluating their impact
- Providing alternative conditions when they do occur
- Providing descriptive information about what happened and where

Typically, in large-team development environments with SAS, very little code is reused across teams and within large organizations as the developer tends to write unique “utility” functions to accomplish his goal. It is the developer’s responsibility to test for conditions and capture return codes that can result in errors. This method is subjective, confusing, usually results in extra coding; further, it may return non-standard results and messages may differ from programmer to programmer.

Having experienced how other languages provide a method to introduce automated test cases within the code base itself, we perceived an opportunity that required good, solid design (for example, in Java we had JUnit). The imperative was to create a built-in facility (or API) that allowed the developer to “assert” events when something was or wasn’t true. For example, when ensuring that a dataset has more than 0 observations, we want to be able to test the conditions that:

- the dataset exists;
- the dataset is “readable” by the current program
- and the dataset has 1 or more observations

Instead of letting each and every programmer figure out how to test these conditions—and spend costly coding time—we developed a standard interface that allows us to manage the automated testing, error handling and event notification. In addition, this interface gives the user an opportunity to report a fatal error, produce a warning or generate an informational note about the condition. Furthermore, auxiliary information such as the dataset or output (from ODS or other methods) can be attached to the event so that it can be used later to review the information more fully without spending time in search of the root cause. Of course, the power of these interfaces lies in their support of any number of tests. Common types of event-generating assertions include:

**Is equal and not equal** – tests the condition of value equality. The values that can be tested often include testing if two datasets have the same number of observations, two variables, macro values, etc.

**Is zero and not zero** – tests the condition for a value of zero (or not zero). This is useful in evaluating if a dataset has zero observations or the expected value of something is or is not zero (such as return codes, observations).

**Is null and not null** – tests whether a variable contains a value.

**Is empty and not empty** – tests whether or not a dataset is empty.

**Datasets are identical and not identical** – tests whether or not a dataset matches another (with and without the metadata matching).

**Whether a file exists or not** – tests whether or not a file exists.

## **TEST FIRST DESIGN**

The concept of test-first design comes from the Java world (especially eXtreme Programming) where developers work with the consumer of the application/ program and write the test cases before any code is actually written. In the eXtreme Programming (XP) or Agile methods of software development, the process of developing code is less valued than the results of the programs themselves, thus, utilizing the programmer to do just enough coding to deliver the expected results. This process essentially supports the idea that if you know what constitutes a failed test case, then you know what the program should do. This exemplifies the notion that we can design software by short and quick iterations that consist of writing a failing test case, making the test pass, and then refactoring (and/or continuing to build in additional functionality). Advocates of eXtreme Programming champion the use of tests as a development tool. Here, we devise tests for all of the major components of a program before they are coded. This means that you are forced to define precisely what your program does; you know where to begin writing your code and you know when you are done writing your program. Our goal is to be able to reproduce our results and, of course, recognize dependencies among our programs early, working to minimize them. You will have segmented your work, so you know at the beginning of the day what you have to do, and you know when you are done. This style of programming gives you a continuity of development that is inaccessible otherwise. Mistakes are detected early enough that so that long debugging sessions and rework are avoided.

As you develop new functionality, you determine what you have to do, write a test case for the desired capability and get the program to work against that test case. When it succeeds, you're done. A key aspect of this process is that developers write small incremental steps, get it right, and then continue. Meanwhile, because the test cases are incorporated into the programs, you are always testing as you add new functionality. When you get this right, development is a continuous process of testing, seeing a simple thing to fix, fixing it, testing, while getting positive feedback continuously.

## **Error Checking versus Automated Unit Tests**

In this paper, we have not distinguished between automated unit tests and error checking. It is important to mention a distinction between these two similar but different tasks:

1. Instrumenting programs with event detection (of which, error checking is one example)
2. Writing automated unit tests and test-first design as a development method

The first item is about putting tests into the body of the codebase that apply to the general case. These tests are largely concerned with checking pre-conditions. Pre-conditions define the conditions under which the module is valid to run at all. Some of these tests might be considered post-conditions, such as checking the exit status of some module that we depend on, but "event detection" in general is not very specific at checking results, especially compared to unit tests.

In contrast, automated unit tests consist of an additional codebase that is only created to exercise and validate the main codebase. Each unit test sets up a known input condition, runs the module to be tested, and then validates that the expected result was achieved. So the main focus of automated unit test is post-conditions (expected results), and these post-conditions are completely specific to one set of defined inputs.

In the examples below, we have largely demonstrated the former – that is, we have introduced examples of error checking that evaluates whether or not our pre-conditions and business rules are true. In order to effectively introduce unit tests, we would need a baseline against which to compare our tests. For example, assume that our report generated an ASCII file. We would take the output from our test case and compare it to a known output which we trust as being the truth. It is often more difficult and time consuming to perform these kinds of tests as they require a very robust testing framework and a great deal of discipline in order to be effective.

## **REAL WORLD EXAMPLES IN PHARMACEUTICAL / CLINICAL PROGRAMS**

Up to this point, we have described any number of tests that could be done to improve the use and reuse of SAS programs. Let us now turn our attention to seeing some of these things in practice. Specifically, we want to demonstrate the use of Test-First Design and the ThotWave Event System.

The best way to elucidate this idea is to take a simple example that might be common to us as SAS programmers. Traditionally, most SAS programmers write programs, submit them, review the output, and revise as needed. The challenge of course is when you have a complex program (or even lots of simple programs), the time it takes to add new functionality is often dwarfed by the time it takes to fully test the impact of the changes on all associated components. Using the test-first design approach, we maintain an exhaustive suite of unit tests (tests for individual components) within the code itself. To that end, we adhere to the following guidelines:

- No code goes into production without tests
- Write the tests first
- Tests determine what code you write

## **Creating Events in SAS**

Here is our task at hand: Suppose we have a demographic dataset that we would like to read to produce a simple listing of patients. To ensure that the program does not fail, we need to perform the following tests:

1. Test whether the demog dataset exists and has >0 observations

2. Make sure that every patient has one and only one record
3. Ensure that the age range for our patient population is in line with our expected age ranges for the study
4. Ensure that the descriptor portion of the dataset matches what we expect as input to our reporting procedure

Let us take you through this example, step-by-step, of how we would design this program using a test-first design.

- Step 1. Write a test case that fails.
- Step 2. Write the program so that the test passes
- Step 3. Continue to add test cases to evaluate whether the requirements of the program are being met.

One of the basic building blocks of test-first-design (TFD) is that we should be able to make sure that we see our program fail first, and then write the program to make it pass. For test driven development, proving error cases are very important. You often want to write unit tests to exercise the error case and make sure the code handles it appropriately. Typically in java you write

```
try
{
    doSomeErrorCase();
    fail("no exception");
}
catch (MyAppException expected) {}
```

This code succeeds if `doSomeErrorCase` throws a `MyAppException`, it reports an error if it throws a different kind of exception, and fails if no exception is thrown. In the Event System, we recommend having a set of test cases that prove that the error conditions actually fire events when the event fails.

Since the first step in this program is to read a dataset and evaluate whether or not the dataset exists and is not empty (has >0 observations), let's test that condition first. You may want to split this up into two different assertions: (1) that the table exists and (2) the dataset has >0 observations.

**Test 1.** So we start building our program by creating a simple test case proving that our table does not exist (pointing the program to read from a dummy library, for example, would work here).

```
%assert_exist(CLINLIB.DEMOG, level=ERROR, type=TABLE_MISSING, ABORT=YES)
```

Next, we fix that by making sure that the library points to the correct location and it has a dataset called `DEMOG` inside. If we create a zero record dataset, we can test the second condition (has >0 observations).

```
%assert_not_empty (CLINLIB.DEMOG, TYPE=TABLE_EMPTY, LEVEL=ERROR, ABORT=YES)
```

Once we validate that the dataset exists but fails when there are no observations, we can replace the dummy dataset with the correct dataset. As we continue to build our program and add additional functionality, we automatically get the benefit of the test cases being run each and every time we run it.

In the previous tests, we were able to use the assert API directly since these tests are common and we had asserts to do exactly what we wanted.

**Test 2.** In the next requirement, we have been asked to test whether or not the demog dataset had one and only one unique patient record in this dataset. This also demonstrates the idea of using ODS to produce an attachment that contains the out of range observations when more than one record is found.

```
Proc SQL;
    create table WORK.DUPLICATE_EXCEPTIONS as
        select patientid, count(patientid) as count from &CLINLIB..DEMOG
group by patientid having count > 1;
quit;
%assert_empty (WORK.DUPLICATE_EXCEPTIONS, TYPE=EXCEPTION_TABLE, LEVEL=INFO,
                ABORT=NO, ATTACHDATA= DUPLICATE_EXCEPTIONS)
```

Here, we created a table that held all of the exceptions, and in our test case, asserted that we assumed it would be empty. If it wasn't, we triggered an event. Note, that the event was not an error (or at least we did not treat it as an error), but rather just an informational event. When triggered, we attach the table to the event so that we can review the exceptions.

**Test 3.** Next, we want to make sure that the age range of all the patients in our data is correct. That is, we have an expected age range (lo and hi) and we need to evaluate whether or not this is true. By using the event API, there are a number of ways to do this:

- We can compare the low and high ranges with the expected value by simply comparing the values coming out of the dataset
- Produce an exception report when the value is OUT OF RANGE
- We could store the expected values of the ages in a dataset and compare the data coming out of our real data with a comparison dataset.

Like most things in SAS, there is probably any number of additional ways to do this. In practice, we would suggest using the exception report approach. For simplicity in this example, we will simply produce a macro variable for the high and the low values found in the dataset and compare that to our expected values.

```
Proc SQL noprint;
    select min(age) into: _MINAGE from CLINLIB..DEMOG;
```

```

        select max(age) into: _MAXAGE from CLINLIB..DEMOG;
quit;

%assert_sym_compare(&_MINAGE, 18,
    level=INFO, type=COMPARISON, abort=NO, operator=NE)
%assert_sym_compare(&_MAXAGE, 35,
    level=INFO, type=COMPARISON, abort=NO, operator=GT)

```

**Test 4.** Next, we want to make sure that before we try to print the demog dataset, we have the right variables and lengths that we expect. This is much more important if we were to do a series of complex manipulations in preparing an analysis dataset, but this simple example should demonstrate the concept.

```

%assert_compare_equal( BASE=master.demog, COMPARE=clinlib.demog,
    TYPE=TEST,
    LEVEL=INFO,
    MESSAGE=expected event master.demog not equal to clinlib.demog );

```

**Test 5.** As an additional step to ensure that our program is as robust as we would like it to be, we want to add a step to evaluate whether or not our PROC PRINT worked. That is, the procedure ran without error. To do this, we compare the global macro symbol that tells us the value of the return code of the last run procedure. If the return code is not zero, then we know something went wrong.

```

Proc print data=CLINLIB.DEMOG;
Run;

%assert_zero(&SYSERR, level=ERROR, message=Procedure exited with errors,
    type=PROC_ERROR);

```

The final program with all of the test cases described above are included at the end of this paper for the reader's benefit to see how we built the program incrementally and continually added new test cases as we introduced new functionality.

### Assert API Discussion

In the previous examples, we used the event system API to assert our test cases. We believe that this provides the programmer a strong foundation for writing code so she can spend more time being able to produce results rather than spending time on writing test cases. Of course all of these things could be accomplished without the Event System API. Because one of the fundamental assumptions is that we provide conditional logic for handling our exceptions, all of the previous examples would have to be coded inside a macro and not simply executed as a program. For example, in our one line test case where we evaluated whether or not a dataset had >0 observations,

we might imagine a specialty macro being developed to handle these cases. For example, the shell of the code to do just these tests might look something like this:

```
%if %table_not_exist(CLINLIB.DEMOG) %then %do;
  %put ERROR: Table CLINLIB.DEMOG does not exist;
  %goto errexit;
%end;

%if %table_empty(CLINLIB.DEMOG) %then %do;
%put ERROR: Table CLINLIB.DEMOG is empty;
  %goto errexit;
%end;

  /* code goes here... */

%errexit:
/* error handler */
%mend;
```

We believe that this method has following drawbacks

- Verbose – simply more code to manage and maintain
- Hard-coded message – notice the put statement provides a message to log the results
- Same exit status as any SAS error (since we used the trick of coding ERROR: in our put statement, SAS recognizes this as an error and sends that back to the environment)
- Awkward control structure (goto error exit)
- Doesn't work in open code; requires macro
- 3 macros to maintain (table\_not\_exist and table\_empty in addition to the macro that controls the conditional logic)

By implementing the assert API call (%assert\_not\_empty), we accomplish the following benefits:

- Standard error message (with optional override)
- Traceback – provides the user not only the information that the condition occurred, but where it occurred
- More description of error (TYPE/LEVEL/CATEGORY)
- ON\_EVENT allows user exit – which is useful for exception handling
- Able to customize exit status

- Abort forces termination of the program at that point (versus going into syntax check mode and continuing to run)
- Forces standardization (e.g., types and messages )
- Events are triggered which can be infinitely more flexible

By merely applying the ThotWave Event System to a SAS application, a complete event trail for all WARNINGS and ERRORS is generated by SAS. Of course, the real power is being able to trigger your own messages and events. The assertion-based API for creating events not only produces an audit trail, but also provides for a robust testing framework for SAS programs.

## **AUTOMATED TESTING**

Hopefully by now we have demonstrated the idea of having a testing platform that is easy to implement in both legacy code and new code. The benefits include:

- Providing a framework for program execution (and testing)
- Knowing what's happening with the programs (technical metadata)
- Standardizing the interfaces for errors (not necessarily just SAS errors)
- Reducing the time to market for new programs to be deployed or changes to be made
- Ensuring that requirements are met as the test cases are developed first
- Increasing the maturity of SAS applications

## **Other Benefits of an Automated Testing Framework**

To provide automatic unit tests each time a program is executed was an important criterion, but we also wanted to be able to ascertain the context of the program. In other words, we needed to be able to run individual programs *and* test the various conditions, as well as “trace” the flow of the program from step to step. By fulfilling these objectives, we knew we could provide much more descriptive information about which program called subsequent programs or macros during the entire job sequence. We now find this information extremely helpful in debugging problems that might have been caused by an upstream data issue or dependency upon something else (results from previous steps, intermediate datasets, macro values, etc.)

The event context includes the category for the current program and the SAS call stack, reflecting the chain of programs that were called leading up to and including the current program. To establish a valid event context, we developed a methodology that allows the event system to know where the event occurred and in what context. This is particularly useful for debugging programs and macros that may be called by any number of other

programs. Most of this methodology is hidden from the programmer, given one of our goals was ease of use for developers to add to existing legacy applications.

Furthermore, as assertions are executed, messages can be sent to SAS indicating whether or not the error is severe enough to warrant complete stoppage of the program and the program stream. This information can be used as a return code in shell scripts, run management systems and schedulers (such as the UNIX cron facility or LSF Scheduler).

The ways in which test cases can be exercised in programs are too numerous to mention here. However, we have seen the following types of test cases implemented:

- Compare two datasets to see if they are equal (both in content and/or structure)
- Evaluate whether or not a dataset is empty
- Confirm that a string (or macro variable) is not empty or is equal to another string
- Test for out of range values
- Data dictionary validation
- Test P-Values and other statistics
- Test that the data does not have duplicates
- Ensure that you have access to an Oracle Table and that it has >0 observations

## **CONCLUSION**

The *thinking data*<sup>TM</sup> Toolkit for Event Management (referred throughout this paper as the Event System) is a powerful tool designed to help organizations monitor mission critical systems. Designed on a framework for full life-cycle application and business rules monitoring, the Event System enables developers and business users alike to monitor, manage and subscribe to enterprise application events from a central location.

The Event System allows all parts of the enterprise to report events that can be recorded and monitored from a common interface. As events are generated from various programming environments (SAS, Java) and operating system shell environments, they are pushed to collectors that serve as conduits to the Event System. Here, the events are collected, added to the system and stored in the database. After the events are stored, a separate process scans the event database and sends out email notifications according to the rules defined in the subscription table.

## **CONTACT INFORMATION**

Your comments and questions are valued and encouraged. Please feel free to contact the author at:

Greg Barnes Nelson  
[greg@thotwave.com](mailto:greg@thotwave.com)  
2054 Kildaire Farm Rd, #322  
Cary, NC 27511  
800.584-2819 - Phone/ Fax

## REFERENCES

- Validation, SAS, and the Systems Development Life Cycle: An Oxymoron?* Neil Howard, Parker-Davis/Warner-Lambert, Ann Arbor, MI and Michelle Gayari, Parker-Davis/Warner-Lambert, Ann Arbor, MI. Paper presented at the annual conference of North East SAS Users Group. PharmaSUG 2000. Seattle, WA.
- Iterative Code Development: Two Testing Techniques* Craig Dickstein, ASG Inc. Paper presented at the annual conference of North East SAS Users Group. NESUG 97. Baltimore, MD.
- Validation and SAS Programming: Benefits of Using The System Life Cycle Method.* Russell Newhouse, Parke-Davis Pharmaceutical Research. Paper presented at the annual conference of SAS Users Group International. SUGI 22. San Diego, CA.
- Automated Testing and Real-time Event Management: An Enterprise Notification System* Barnes Nelson, Greg and Wright, Jeff. Paper presented at the annual conference of SAS Users Group International. SUGI 29. Montreal, Canada.

## Test First Design Resources

- [http://www.junit.org/news/article/test\\_first/index.htm](http://www.junit.org/news/article/test_first/index.htm)
- <http://www.rubycentral.com/articles/pink/>
- <http://today.java.net/pub/a/today/2004/01/22/DozenWays.html>
- <http://www.extremeprogramming.org/rules/testfirst.html>
- <http://www.icas.edu/series/MPP/Hill.pdf>
- <http://www.computer.org/software/homepage/2001/05Design/?SMSESSION=NO>
- <http://www.artima.com/intv/testdriven.html>
- <http://c2.com/cgi/wiki/TestFirstDesign>
- <http://c2.com/cgi/wiki/TestDrivenProgramming>

## APPENDIX: EXAMPLE SAS PROGRAM WITH TEST CASES

```
/* -----  
* $Id: demogReport.sas,v 1.0 2004/03/06 $  
* $Author: Greg Nelson $
```

```

*
* <doc>
* @input SAS Table: &CLINLIB.DEMOG
* @purpose Access the DEMOG data and produces a simple listing report.
* </doc>
*
* Prepared by ThotWave Technologies, LLC.
*
* -----
* Revision 1.0 2004/03/06 Greg Nelson
* Initial Development
*
* ----- */
%header(module=%str($Id: demogReport.sas,v 1.0 2004/03/06 13:38:36 gnelson Exp
$), category=Test.Unit );

```

```

title1 "Simple Reporting Application - Test validation routine";
%generate_event( TYPE=MESSAGE, LEVEL=INFO,
                MESSAGE= demogReport.sas - Executing Version Revision: 1.0)

```

1

```

/* Test the dataset that it exists and has >0 observations */
%assert_exist(CLINLIB.DEMOG, level=ERROR, type=TABLE_MISSING, ABORT=YES)
%assert_not_empty (CLINLIB.DEMOG, TYPE=TABLE_EMPTY, LEVEL=ERROR, ABORT=YES)

```

2

```

/* Make sure that every patient has one and only one record */
Proc SQL;
    create table WORK.DUPLICATE_EXCEPTIONS as
    select patientid, count(patientid) as count from &CLINLIB..DEMOG
    group by patientid having count > 1;
quit;

%assert_empty (WORK.DUPLICATE_EXCEPTIONS, TYPE=EXCEPTION_TABLE, LEVEL=INFO,
               ABORT=NO, ATTACHDATA= DUPLICATE_EXCEPTIONS)

```

3

```

/* Test the expected age range */
Proc SQL noprint;
    select min(age)into: _MINAGE from &CLINLIB..DEMOG;
    select max(age)into: _MAXAGE from &CLINLIB..DEMOG;
quit;

%assert_sym_compare(&_MINAGE, 18,
                   level=INFO, type=COMPARISON, abort=NO, operator=NE)
%assert_sym_compare(&_MAXAGE, 35,
                   level=INFO, type=COMPARISON, abort=NO, operator=GT)

```

4

```
/* Test the metadata portion of the input dataset is correct */
%assert_compare_equal( BASE=master.demog, COMPARE= clinlib.demog,
                      TYPE=TEST,
                      LEVEL=INFO,
                      MESSAGE=expected event master.demog not equal to clinlib.demog )
```

5

```
/* Test Major procs to ensure that we didn't have any failures */
Proc print data=CLINLIB.DEMOG;
Run;

%assert_zero(&SYSERR, level=ERROR, message=Procedure exited with errors,
            type=PROC_ERROR)

%footer(module=%str($Id: demogReport.sas,v 1.0 2004/03/06 13:38:36 gnelson Exp
$));
```