Unit testing as a cornerstone of SAS application development

Dante Di Tommaso, F. Hoffmann-La Roche

Tools Management, Basel, Switzerland

Confidence in software is good, but certainty is better. The test strategy at the heart of a validation plan determines the level of certainty a software project achieves. *Ad hoc* testing may momentarily convince a developer that a program is complete, but is inappropriate for programs and results intended for others. Perfect testing, on the opposite end of the spectrum, is rarely achievable. Testing is an exercise in both discipline and balance. Learning, adopting, and adapting testing practices refined by software professionals can strengthen a programming team even more than pursuing subtle new programming techniques. The benefits to a development project can be dramatic. Unit testing has proven invaluable as a cornerstone of the validation plan for an ongoing SAS® development project, rewarding the team with efficiency and confidence just short of certainty. The following discussion summarizes the awareness and advantages we have gained from these recent experiences, and encourages the reader to explore and promote these practices in their own environments. While our examples and discussion are based on SAS, the concepts are generally relevant to software development.

Keywords: Software development life cycle, Application development, Code construction, Unit test, Test-driven development

Introduction

Di Tommaso and Binder previously summarized basic software development concepts gleaned from classic programming texts.[3] Testing, addressed briefly in that discussion, warrants emphasis and focus.

Too often testing of SAS programs is limited to an *ad hoc* method. A clear record of repeatable validation activities typically does not exist. The original reviewer may recall running a few 'procs' and producing matching results. Some code may remain from prior efforts, but may no longer complete successfully due to temporal dependencies. None of this helps colleagues repeat those steps following a change to the analysis environment (whether a change to requirements or the data).

To keep the scope of this discussion manageable, we limit ourselves to unit testing as defined by Howard and Gayari: testing actual against predetermined, expected results.[5] To ensure the content remains tangible, we base the discussion on lessons learned during an ongoing, complex SAS development project. The authors propose a standard approach that allows developers to focus on sensible tests rather than on the secondary tasks such as test execution, evaluation, and reporting.

Correspondence to: F. Hoffmann-La Roche, Biometrics Operations, Tools Management, Malzgasse 30, CH-4052 Basel, Switzerland. Email: dante.di_tommaso@roche.com

Unit Test Foundation

Functional or unit testing is a fundamental concept in software development, particularly for so-called test-driven development.[1] The intent is to structure tests around individual components with well-defined scope. It establishes and protects a contract between the user and developer for agreed functionality.

Although such formal testing concepts have remained largely outside the SAS programming world, there is no good reason for this. Unit testing brings transparency, efficiency, and robustness to software development and maintenance activities. In recent years, several authors have raised awareness of and encouraged the SAS community to embrace these concepts, most notably SAS validation pioneers within Parke-Davis[5,8] and ThotWave Technologies.[7,13]

The authors have recently implemented unit testing as a cornerstone of development activities for a new analysis and reporting macro library. This experience has been transformative, and has proven the value of investment upfront in a standardized test framework. As emphasized by Howard and Gayari, 'the cost of correcting errors increases exponentially with the stage of detection'.[5] The intent of unit testing is not simply to detect errors early, but rather to prevent them from entering the code base.

Unit test example

Consider a unit of code that returns a count of words in a string. Since we want to focus on test concepts, a

© 2011 Pharmaceutical Users Software Exchange (PhUSE)
Published by Maney
DOI 10.1179/175709311X13166801334316

```
/*+---------------------------------------+*/
/*| COUNT_WORDS requirements:             |*/
/*| Assign &RESULT the count of space-    |*/
/*| delimited words in user-specified string |*/
/*+---------------------------------------+*/
%macro count_words(  string
                   , result    = countw
                   , longword  = longw
                   , delimiter = ' '
                  );
%mend count_words;

/*+------------------------------+*/
/*| TEST STRATEGY to confirm reqs |*/
/*+------------------------------+*/
data _null_;
  array string [4] $
        _temporary_   ( 'I am.number 4' 'Alt-Alt Alt' '-One-' ' ' );
  array delim  [4] $
        _temporary_   ( ' .'            '- '          ' -'    ' ' );

  array exp_cnt [4]
        _temporary_   ( 4               3             1       0   );
  array exp_len [4]
        _temporary_   ( 6               3             3       0   );

  do idx = 1 to dim(string);

    %COUNT_WORDS( STRING[IDX], DELIMITER=DELIM[IDX] );

    if countw = exp_cnt[idx] and longw = exp_len[idx] then
         put 'PASS test ' idx;
    else put 'FAIL test ' idx;
  end;
run;
```

Figure 1 Initial macro interface and test strategy, demonstrating invalid assumptions

simple example suffices; the purpose of the macro and its implementation do not matter. Test-driven development involves writing the tests while devising the requirements for the planned macro %COUNT_WORDS. As emphasized by Nelson and Wright, such testing requires a code base that is separate from the application code base.[7,13] The macro definition, itself, can begin simply as an interface; an empty shell.

For our purposes, the user requirements are documented in comments in Fig. 1. SAS 9 users will notice that this ubiquitous macro is now obsolete, replaced by the native SAS 9 function COUNTW().

Test requirements

Do not develop the macro without first acquiring a basic understanding of the requirements, and initial concepts for both the interface and tests.

Thus, driven by requirements, we begin with a macro shell and a test strategy in Fig. 1. Note that the structure of the tests includes two fundamental components: typical inputs are listed in arrays 'string' and 'delim'; expected results for each test are asserted in the arrays 'exp_cnt' and 'exp_len'.

Strict control and specification of the input data plus the ability to specify in advance and with certainty the expected results are the key to unit testing.[10] In our project, test programs are stand-alone programs that generate their own pre-defined test data and expected results. The macro being tested can serve nearly any purpose, as long as the results are consistent and predictable.

As written, the test cases reflect explicit, implicit, and invented requirements: return a numeric result (explicit); count duplicate words and return 0 for null strings (both implicit); handle one or more user-specified delimiters and return the length of the longest word (both requirements apparently invented by the developer and parameterized as &DELIMITER and &LONGWORD, respectively).

The reader may have further noticed that the test strategy, given the ambiguous requirement for a 'result', assumes that the macro should run in a DATA STEP and assign values to DATA STEP variables, by default COUNTW and LONGW. This could easily be an inappropriate assumption. Moving forward with this example, the authors assume that the requestor later clarified in the requirements that the macro should create macro variables, rather than DATA STEP variables.

The need for customizable delimiters may be genuine, having emerged from discussions around use cases. The need may also come from development planning, knowing that similar functionality is already required for a separate circumstance. In either case, the requirements should have been updated.

Otherwise, there may be no business justification for investing the extra effort in developing, testing, documenting, and maintaining this extended functionality; or presenting the user with the additional complexity in the interface. Developer satisfaction could justify some additional effort, but restraint is often equally appropriate. Developers should at least assess such considerations separately and conscientiously.

Some implicit requirements are likely no concern to the users; others will directly affect users and require their input and confirmation. Correctly assessing a situation and then taking the appropriate actions are essential skills, and rarely as simple as this example suggests.

The requirements are the contract between the users and developers, and should be complete and robust. The tests and eventually the macro definition must be based on, rather than establish, the requirements.

Test transparency and standardization

At this point, the above code confirms that an empty macro definition can only fail:

```
FAIL test 1
FAIL test 2
FAIL test 3
FAIL test 4
```

Several aspects of these tests are inadequate. There is a general lack of transparency in (1) the total number of tests; (2) the rationale behind these particular tests; and (3) the overall interpretation of these results. Without such transparency, the tests add minimal value. Even 10 times as many tests implemented in this manner should leave a reviewer unconvinced of robustness and quality. Further, a future developer responsible for maintaining and extending functionality will have trouble simply understanding the tests. This inflates the risk associated with application and test maintenance by building false confidence that a testing framework is protecting existing functionality.

Defining the right set of tests is an important task. All subsequent activities can distract from that primary task, and should be reduced to a minimum: test implementation, execution, assessment, and reporting. There is no need to re-invent these secondary activities for each component tested. In fact, eliminating variation in these secondary activities leads to valuable gains in transparency and efficiency.

Test standardization

With these ideas in mind, we implemented a stand-alone `%PASSFAIL` macro to handle these routine tasks, allowing developers to focus on selecting optimal tests. We automated these secondary activities by standardizing how we define and implement tests. We established a standard data structure sufficiently flexible to meet our needs, allowing us to implement tests using a uniform, transparent approach. The `%PASSFAIL` macro can then process these standardized tests and report results in an equally consistent and transparent manner.

`%PASSFAIL allows us, for example, to reduce the test code above to a set of records like the following in a standard data set:`

```
test_type       = 'M'                 ;
test_id         = 'CW.01.02'          ;
test_dsc          =  'Test  multiple
                  delimiters'        ;
test_mac        = 'count_words'       ;
pparm_string    = 'Alt-Alt Delim'     ;
kparm_delimiter= '%str(-)'            ;
test_expect_sym = 'countw=3|longw=5'  ;
                 OUTPUT              ;
```

The above observation defines test 'CW.01.02' as a `%COUNT_WORDS` call with one positional (`string = 'Alt-Alt Delim'`) and one keyword parameter (`delimiter = '%str(-)'`), and the expectation that the macro will assign two macro variables: `&COUNTW = 3` and `&LONGW = 5`. `%PASSFAIL` then executes and assesses each test, and summarizes results. Note that this test specification matches the revised requirements to create macro symbols, rather than DATA STEP variables.

The above test for an incomplete version of `%COUNT_WORDS` produces the results in Fig. 2.

In addition to this test summary, we separately confirm that standard SAS logs only include messages associated with the specified tests. An enhancement of `%PASSFAIL` could integrate this step, as well.

The example above suggests the use and benefits of `%PASSFAIL`, which we otherwise do not describe in detail. Mainly, the `%PASSFAIL` package continues to evolve, including both the macro itself as well as its test package. The macro is currently capable of testing dataset and macro symbol results directly, and format catalogues indirectly; it could be extended to test additional items such as text outputs. There is currently no protection against a particular test interfering with a subsequent test executed in the same SAS session, another important area for enhancement.

A simplified implementation of `%PASSFAIL` could combine Test ID and Description into a single parameter for less flexible messaging and reporting; and could further combine the macro name and parameter settings into a single string with explicit, flexible executable code (rather than leaving `%PASSFAIL` to build specific macro calls). In particular, removing explicit parameter settings, and therefore

```
FAIL  - CW.01.03 , Non-missing string, default delim
  Expected:
  Returned:
  Expected Global Syms: countw=3|longw=9
  Returned Global Syms: countw=4|longw=9

### -----
###
### COUNT_WORDS Overall: ALL TESTS PASSED - 3  but  Failed - 1  test(s)!
###
### -----

   001   CW.01.01    Missing value parameter
   002   CW.01.02    Basic call, duplicate words
   003   CW.01.03    Non-missing string, default delim
   004   CW.01.04    Non-missing string, multiple delims
```

Figure 2 Summary of %PASSFAIL execution and evaluation including failure details

dependency of the test data structure on the interface of a particular macro, would permit pooling of test data across components and therefore centralized management of test specifications. Please contact the author for `%PASSFAIL` details, or to collaborate on its evolution.

A test-driven approach allows the developer to build up a set of transparent, repeatable tests while constructing the bespoke macro. With the protection of unit tests in place, any developer can subsequently modify code with confidence. The recipe for such confidence is simple: create tests for all behaviour worth protecting.[2]

Test Space and Coverage

Selecting suitable tests is critical since the tests protect existing functionality. Test design can involve more art than science. Keeping several considerations in mind can help unleash the test artist in all.

Most importantly, keep tests simple and transparent. Do not create clever tests that exercise several requirements at once. The beauty of super-tests that promise to consolidate testing to a few cases can quickly unravel when something changes, whether the requirements, the development team, or other. Keep tests simple and focused. Test one concept at a time to ensure transparency and maintainability.

Apply the same principle to the code that creates the tests. There are situations that benefit from clever, character-sparing code. Unit testing is not one. Using a standard test structure such as `%PASSFAIL` specifies, we write tests as explicitly and clearly as possible such as via simple DATALINES or plain variable assignments, as illustrated above.

Next, a clean test strategy can only emerge from a clear understanding of the test space. Think along two general dimensions: interface and implementation.

Interface tests

'Interface' comprises inputs, which can exist on several levels. Imagine a simple case: a macro that has two parameters, each with two valid values. With just $2 \times 2 = 4$ combinations of inputs, complete test coverage appears feasible.

Consider the intended users. The need to handle (and test) unexpected inputs depends on the target audience. An internal library used as core, hidden components of a larger application does not need to handle unexpected inputs as thoroughly as end-user programs. For end-user programs, the code should handle unexpected inputs, and tests should exercise this handling.

Consider the results of the program, as well, and how a user might subsequently use the results. If a program returns a value rather than executable code, consider macro-quoting issues. SAS autocall macros such as `%LOWCASE` and `%QLOWCASE` are typically available in pairs, with and without a macro-quoted result. Adopting this SAS convention is one convenient option, and familiar to SAS users.

The test space is less obvious for complex interfaces and for macros that process unpredictable data. To what extent should a macro check the validity of user-supplied parameters or handle various combinations of missing records, variables and values in input data? Is internal SAS messaging sufficient, or should the custom application intercept issues and provide detailed guidance? Farrugia effectively addresses data dependencies and considerations.[4]

The requirements combined with considerations of use cases should guide the tests, and highlight important and feasible interface tests.

Implementation tests

'Implementation' covers the logical flow of the program. Readers may protest that the implementation is irrelevant, as long as the requirements are confirmed and protected with interface tests. But a particular implementation could require testing that seems unnecessary *vis-à-vis* the interface. Imagine that a `%COUNT_WORDS` implementation that has two separate logical blocks: one for when the default delimiter is active, and a separate block for when the user specifies a non-default delimiter. Given such an implementation, testing is incomplete without seemingly redundant tests for default and non-default delimiters. Assessment of the implementation dimension, in this case, would expose a poor design.

If apparently redundant tests do not disturb the developer, an independent reviewer may catch the inefficiency. In effect, traditional code review of both the application code base as well as the test code base can lead to improvements in both.

Continuing from above with `%LOWCASE`, learn from SAS's counterexample and avoid implementations that increase test burden and maintenance in general. One might reasonably expect that `%LOWCASE` and `%QLOWCASE` implementations differ by a single character. Or that `%QLOWCASE` is a secondary program that simply quotes the results of the primary implementation, `%LOWCASE` (or *vice versa*). In fact, these two autocall macros are surprisingly different. This approach only introduces unnecessary test and maintenance burden.

When feasible, invest upfront in complete test coverage. With practice, implementing complete test coverage for a focused, well-defined test space costs little compared with the long-term benefits. The certainty, alone, that unexpected consequences are unlikely when making future modifications or extensions is invaluable.

Design Feedback

An understanding of the test space also serves as design feedback. Focused program components have focused test plans. If the test plan becomes complex, requiring tests of seemingly unrelated functionality, take a step back and review the scope of the component. Overloading a single component with 'related' functionality is likely an inefficient approach. Clean, dedicated components are not only easier to understand, test and maintain, but are simpler to use and allow greater flexibility.

Detecting that unit tests are losing focus can in fact create opportunities to better modularize the components themselves, and ultimately improve the longevity of the application.

Test Maintenance

Unit tests must be maintained along with the corresponding components. Any change to a component necessitates review of existing tests. From a minor code change, to general refactoring for performance and maintainability, to extending the component, the tests will likely require revision as well. The component and its test programs are inseparable.

Automated Test Framework

Having established the test approach and commenced the testing code base, developers' thoughts should turn to test automation during development cycles. While test suite automation is beyond the scope of this paper, the authors do wish to highlight that several options are already available to SAS developers: ThotWave's Framework for Unit Testing SAS programs (FUTS),[13] Scocca's SCLUnit,[11] and HMS Analytical Software's SASUnit.[6,12] FUTS and SCLUnit are both available from SASCommunity.org. SASUnit is available from sourceforge.net.

Conclusion

Unit testing is just one of several cornerstones upon which to construct an efficient, robust, and maintainable application.[1,9] Initial investment in processes, skills, and discipline can be intimidating. Ever tightening timelines can obscure the value of such upfront investment. Do not be discouraged. In our experience, we profited immediately from our efforts. We are able to respond rapidly to frequent changes in a dynamic environment with the certainty that we are protecting established behaviour. We can move forward quickly with application enhancements and extensions, while a parallel, robust test code base defends against unintended changes.

Unit testing is a starting point, a gate to a path towards certainty of application accuracy and consistency. It is a compelling, even addictive, path that reduces risk and stress, and consequently increases efficiency of development.

References

1 http://en.wikipedia.org/wiki/Software_testing [cited 2011 Jul 24].
2 Bentley JE. Software testing fundamentals — concepts, roles, and terminology. In: Proceedings of the 30th Annual SAS® Users Group International Conference; 2005 Apr 10–13; Philadelphia, PA, USA: SAS Institute Inc.
3 Di Tommaso D, Binder C. Concepts learned from our programming cousins. In: 5th Annual PhUSE: Proceedings of the Pharmaceutical Users Software Exchange; 2009 Oct 19–21; Basel, Switzerland: PhUSE.
4 Farrugia R. A case study in using unit testing as a method of primary validation. In: PharmaSUG 2010: Proceedings of the Pharmaceutical Industry SAS® Users Group; 2010 May 23–26; Orlando, FL, USA: SAS Institute Inc. Available from: http://www.phuse.eu/download.aspx?type=cms&docID=2558. [Accessed August 10, 2011].
5 Howard N, Gayari M. Validation, SAS, and the systems development life cycle: an oxymoron? In: PharmaSUG 2004: Proceedings of the Pharmaceutical Industry SAS® Users Group; 2004 May 23–26; San Diego, CA, USA: SAS Institute Inc. Available from: http://www.lexjansen.com/pharmasug/2000/dmandvis/dm09.pdf [Accessed August 10, 2011].
6 Mangold A, Warnat P. Automatic unit testing of SAS programs with SASUnit. In: PharmaSUG 2008: Proceedings of the Pharmaceutical Industry SAS® Users Group; 2008 Jun 1–4; Atlanta, GA, USA: SAS Institute Inc. Available from: http://www.phuse.eu/download.aspx?type=cms&docID=573 [Accessed August 10, 2011].
7 Nelson GB. SASUnit: automated testing for SAS. In: PharmaSUG 2004: Proceedings of the Pharmaceutical Industry SAS® Users Group; 2004 May 23–26; San Diego, CA, USA: SAS Institute Inc. Available from: http://www.lexjansen.com/pharmasug/2004/datamanagement/dm10.pdf [Accessed August 10, 2011].

8 Newhouse R. Validation and SAS programming: benefits of using the system life cycle method. In: Proceedings of the 23rd Annual Conference of the SAS Users Group International; 1997 Mar 16–19; San Diego, CA, USA: SAS Institute Inc. Available from: http://www2.sas.com/proceedings/sugi22/APPDEVEL/PAPER20.PDF [Accessed August 10, 2011].

9 O'Donoghue S, Ratcliffe A. Configuration management for SAS software projects. In: Proceedings of the SAS Global Forum; 2009 Mar 22–25; Washington, DC, USA: SAS Institute Inc. Available from: http://support.sas.com/resources/papers/proceedings09/271-2009.pdf [Accessed August 10, 2011].

10 Schiro S. Testing your SAS code against known data. In: Proceedings of the SAS Global Forum; 2009 Mar 22–25; Washington, DC: SAS Institute Inc. Available from: http://support.sas.com/resources/papers/proceedings09/208-2009.pdf [Accessed August 10, 2011].

11 Scocca DA. Automated unit testing for SAS® applications. In: Proceedings of the SAS Global Forum; 2008 Mar 16–19; San Antonio, TX: SAS Institute Inc. Available from: http://www2.sas.com/proceedings/forum2008/002-2008.pdf [Accessed August 10, 2011].

12 Warnat PR. Unit testing and code coverage assessment with SASUnit. In: PharmaSUG 2004: Proceedings of the Pharmaceutical Industry SAS® Users Group; 2010 May 23–26; Orlando, FL: SAS Institute Inc. Available from: http://www.phuse.eu/download.aspx?type=cms&docID=2586 [Accessed August 10, 2011].

13 Wright J. Drawkcab Gnimmargorp: test-driven development with FUTS. In: Proceedings of the 31st Annual SAS Users Group International Conference; 2006 Mar 26–29; San Francisco, CA: SAS Institute Inc. Available from: http://www2.sas.com/proceedings/sugi31/004-31.pdf [Accessed August 10, 2011].