PhUSE 2009

Paper PO11

Concepts Learned From Our Programming Cousins

Dante diTommaso, F. Hoffmann-La Roche, Basel, Switzerland
Christof Binder, F. Hoffmann-La Roche, Basel, Switzerland

ABSTRACT

SAS® Programmers have an extensive catalog of references available to them through the SAS Bookstore, but very few explore the process of developing disciplined, structured programs and applications. Training opportunities for clinical programmers are often even more limited in this regard. We have opportunities to learn details about existing or new features or modules, but rarely about the fundamentals of our craft. For those, we have to venture into the exciting and enlightening libraries of our cousins developing C and Java applications.

INTRODUCTION

For clinical programmers, programming is only one measure of our expertise, with our varied responsibilities and commitments to a development team often infringing on our actual programming time. As a result, it is not uncommon that programming tasks are undertaken on an *ad hoc*, "let's just get it done" basis. Often our training opportunities and records reflect the low priority of programming craftsmanship. We have opportunities to learn the syntax of languages or new modules (such as SAS 9.2's Graph Template language, or the ODS Statistical Graphics procedures) but we rarely have opportunities to learn cutting edge programming techniques for planning, designing, writing and testing robust code. As we evolve in our careers from writing stand alone programs to developing and maintaining libraries and complex systems, it is only through determined personal effort that we are able to identify and benefit from basic training in software engineering, despite the fact that SAS programmers are often effectively engineering complex software systems. In the authors' experiences such training in crucial fundamentals has only been available outside our immediate industry, by coming from or venturing into other professional programming worlds like C and Java.

With this paper we hope to motivate readers to reach beyond the immediate resources, and we challenge trainers to do the same. Time pressures and varied responsibilities may explain our engineering shortcomings, but they are not a valid excuse. In fact, applying the latest techniques and best practices to any of our tasks can only help to ease pressures, improve efficiency and ultimately accelerate clinical development.

Below we highlight a few important concepts of software engineering which rarely filter through to our profession. These represent small windows into principles that have evolved in recent decades into today's best practices for software development. We base our discussion on popular programming books throughout this period (see **References**), and in doing so hope to motivate readers to more thoroughly explore these resources. By reducing the structure of this paper into the four most basic stages of software development, we intend to emphasize that at each stage we can learn from our programming cousins. We hope that this journey can become as engaging and transformational for you as it has been for us.

READ

By 1971, Weinberg was already bemoaning the demise of program *reading*. As access to machines increased with the proliferation of terminals, programmers no longer had to wait in line to turn in card decks, or hang around the coffee machine or vending machines while waiting for the computing center to return their outputs. Programmers no longer congregated in an informal manner to share and discuss their ideas and their programs. It is incredible to us that so early in the evolution of our profession, the simplest learning tool with the greatest potential was already considered old fashioned and all but dead.

I therefore consider myself very lucky to have started my career as a reader. Luckier still to have had two seasoned mentors who couldn't have been more different, except that they both insisted that I read their programs. Hard copies. Their styles were complete opposites. One was the most technical programmer I have encountered, always thinking on at least one level removed from the current task and writing code that wrote the code for the current task. His counterpart was the truest possible disciple of Einstein's apocryphal aphorism: "Everything should be made as simple as possible, but not simpler." He remains for me the master of the transparent program with complex results.

Without Weinberg's promotion of reading programs (1999, originally 1971), I might not have had that opportunity; program reading might have truly died. Instead, it has gained support in the broader software world. But in our corner, the decline of reading continues. For us, writing code often feels like an intensely personal endeavor, but it shouldn't. The code is not us, the code is not ours; it's a tool that we deliver to our employer, and we should regard it as such. Submitting to the review, analysis and constructive criticism of peers feels uncomfortable and threatening, as if the content of our programs might reveal some weakness. Software engineers have long known the risks associated with taking the product of our professional labors (our programs) too personally, but the phenomenon nonetheless remains as a barrier to learning and assimilating best practices. Weinberg recognized these risks when he coined the term "egoless programming", and the awareness that he forced to the surface of the programming world has inspired the paired programming and code review approaches that today are core principles of "extreme" and "agile" software engineering methodologies.

Reading is therefore not only a valuable learning tool, but essential for building team cohesion. It can help bring down personal as well as technical barriers due to different programming techniques (if team members can get past the discomfort and assimilate an egoless psychology). I've long wondered how my first programming group would have evolved if my two mentors had also forced each other to read their programs. Uniformity was clearly not within reach or appropriate, but learning opportunities (even for seasoned programmers) and a better understanding of and appreciation for colleagues' thinking certainly was.

Reading and the subsequent critical review can also greatly improve the quality of software. This is not mere speculation, but the proven conclusion of emergent methods such as paired programming. Encouraging peers to challenge complex logical structures and general approaches could benefit clinical programmers just as it has software engineers. An expert macro programmer may miss an opportunity to use data step arrays more efficiently, and likewise a data step wizard may struggle unnecessarily due to an aversion to macro programming. Both circumstances represent not only learning opportunities but also opportunities to streamline production code at the critical, early stage: before in needs to be maintained.

Weinberg was probably not the first, and certainly not the last, to recognize that "the human eye has an almost infinite capacity for not seeing what it does not want to see" (p. 55). Invite someone to read your code; they'll see what you can't. And agree to return the favor.

THINK

Think before you write. It seems absurd to even mention. But in reality, time pressures often make the temptation to dive directly into programming irresistible. This approach guarantees that, with perhaps the exception of trivial tasks, we rarely follow a straight path from problem to solution.

Programming is problem solving, and without a thorough understanding of the root problem we are unlikely to solve it. Jon Bentley (1999) provides several enlightening case studies of problem abstraction and reduction. The first pages of Programming Pearls impart an essential lesson gleaned from the write-first approach. When you find yourself in a technical situation, wondering "How can I accomplish this next step?", it's a good time for a new perspective. Ask instead "Why do I want to do this?". It could be that you've started down the wrong path, inventing new problems along the way, and are now dedicating your efforts to solving problems of your own creation, rather than the intended root problem. Don't be afraid to start over -- but with a planning stage. The time you same may very well be your own.

You can protect yourself from starting down the wrong path by thinking about a problem and planning a solution from beginning to end before writing any code. This does not preclude writing, entirely, but don't start by writing candidate production code. Pseudocoding has become an accepted process in software engineering, as described by both Bentley (1999) and McConnell (2004; a reference manual which should be chained to every programmer's keyboard). It is not unlike planning a complex program by first writing out a comments framework. Plan your overall strategy and core algorithms. Look for patterns that allow reduction of complexity or centralization of reusable algorithms. Experiment with specific technical solutions in isolation, before reaching the point where that solution becomes essential to the algorithm rather than one alternative.

Think about the general version of the problem, rather than an immediate task. Given 10 lab samples in an initial data delivery, no one would consider writing code to handle exactly 10 lab samples. Nonetheless, programs often exhibit such hard-coded "magic number" logic simply because the developer allowed current circumstances to constrain the problem statement. We should be vigilant in eliminating such assumptions while both reading code as well as planning new code. Doing so even for one-off tasks reinforces this preferred approach, which then becomes habitual.

Bentley summarizes decades of software experience with a simple proposal: with a proper problem definition, algorithm design, and data structure in mind, "writing correct code is usually easy" (p. 33).

WRITE

What makes code good? Efficiency? Those incredibly efficient, hyper-nested SQL blocks that take hours or days to perfect, but moments to execute -- are they good? The answer is not obvious, however much I may have exposed my bias simply by asking.

"Simple, few parts, easy to maintain, very strong." - Gen. Chuck Yeager, test pilot

Gen. Yeager, as quoted by Jon Bentley (1999, p. 6), was assessing a plane engine, rather than a program. But the criteria are equally applicable to software. Note that Yeager didn't bother to mention that the engine actually ran, or that it was capable of lifting a craft off the ground and safely delivering it to its destination. Assessment of code "goodness" is irrelevant if it is not above all else accurate and correct. An algorithm is accurate if it does what the developer intended. The resulting program is correct, if what the developer intended also satisfies users' needs.

CONSTRUCTING CODE

Starting with pseudocode or a comments framework can help to deliver quality code. Pseudocode differs from comments in both structure and content. While comments explain program statements and blocks, pseudocode follows the structure of code without relying on any consideration of a particular programming language (whether features or limitations).

As with a comments framework, starting with pseudocode allows the programmer to *design* without the immediate burden of *implementation*. As McConnell highlights, pseudocode is easier to review since it describes strategy unobscured by syntax, and thus explicitly supports an iterative design process. This brings all the benefits of catching any problems at an early stage, and redesigning prior to substantial coding efforts. Once programmers begin replacing pseudocode with real code, the pseudocode often becomes the comments.

We recommend experimenting with the reverse approach, as well. Once you've written a program, read through just the comments, which should follow program intent and structure. If reading the comments does not impart a reasonable overview of what the program does and why is does it in this particular way, then most likely the comments are of poor quality. As McConnell states, "[c]omments are easier to write poorly than well, and commenting can be more damaging than helpful" (p. 781). Luckily he continues with a discussion of effective comment techniques.

MAINTAINABILITY

Program writing is often program maintenance or *editing*. After correctness, program maintainability is a top mark of quality code. Given a choice between an efficient or a maintainable algorithm, and barring any critical efficiency demands, we would choose the later without deliberation. Bentley again provides several good examples of short-term efficiency tricks leading to unexpected, unwanted problems that were painfully difficult to resolve. Efficiency has its value, but it must be justified vis-à-vis sacrifices in maintainability.

Editing typically involves editing someone else's code. There are several questions to keep in mind when writing code to improve an editor's future experience. Most are variations on every child's favorite question, often the hardest to address with a satisfying response: "Why?" Readers and writers alike should maintain a child's enthusiasm for asking "Why is this code here?" There are several potential responses, and knowing that response can greatly ease the burden of code maintenance. Often the answer lies in the original requirements: conventions for having a zero study day or not; conservative approaches to deriving durations of events or attributing lab results to specific interventions.

There may be machine dependencies, such as differences between Windows and UNIX SAS. These should be highlighted in code, used only when unavoidable, and ideally handled in a way that is both transparent to users and easily maintained (such as isolating machine dependent instructions in centralized or otherwise clear locations).

There are also limitations of specific language technique, which can be dangerously temporal. Before SAS8, there were many reporting programs that relied on writing out, reading back in and parsing SAS PROC results. With SAS8, many of these techniques were rendered obsolete since SAS updated the layout of proc results to align with ODS. Such dependencies should be transparent to anyone responsible for code maintenance.

Asking these questions throughout the lifecycle of programs can contribute immensely to code quality, and therefore to its the long-term value. Developers should always be prepared to defend their approach and techniques, comfortable doing so, and willing and eager to revise code should a defense crumble.

TEST

An all-too-common method of testing clinical analysis programs is *ad hoc* testing. In case analyses need to be repeated, there may or may not be a clear record of what validation steps were taken following the original analyses. The original reviewer may report that s/he ran a few "freqs" and produced matching results. Some code may still be available from that first round of review. What code does remain may no longer run due to temporal dependencies. None of this helps a new team member repeat those steps following a change to the reporting environment (whether a change to analysis requirements or study data).

UNIT TESTING

Functional or "unit" testing is a fundamental concept in software development, particularly for so-called "test-driven" or "test-first" development. The intent is to structure and optionally automate testing, and is often described as a "contract" between the user and developer for agreed functionality.

Although it has remained largely outside the clinical programming world, there is no good reason for this. To the

situation introduced above, unit testing could bring thorough efficiency to the software verification task, and therefore confidence, as well.

Unit testing is first-level functional testing, and the only level that we'll demonstrate in detail. In our companion paper **TU07**, the authors provide additional discussion of different test scenarios (see **References**). To continue exploring test levels, start with the "Unit Testing" entry in wikipedia, and follow that discussion through to integration, system, regression testing, and wherever else your mouse takes you.

TEST-DRIVEN EXAMPLE

Consider a trivial, self-contained "unit" of code that simply returns the separate components of a two-level SAS name: the library and member names. We want to focus on the testing concept, so the macro definition itself is irrelevant. Taking a test-driven approach, we start by writing our tests for the planned macro %SPLITNAME, which begins as an empty shell. Only after our tests are finalized, would we implement a macro definition that passes all test.

```
/* MACRO SHELL */
%macro splitname(name=, libname=LIBNAME, memname=MEMNAME);
%mend splitname;

/* TEST STRATEGY */
data _null_;
  array twolevel [4] $ _temporary_ ('SASHELP.CLASS' 'SasHelp.Class'
                                    'Work.Temp' 'Temp');
  array explibs  [4] $ _temporary_ ('SASHELP' 'SASHELP' 'WORK' 'WORK');
  array expmems  [4] $ _temporary_ ('CLASS' 'CLASS' 'TEMP' 'TEMP');

  do idx = 1 to dim(twolevel);

    %SPLITNAME(NAME=TWOLEVEL[IDX]);

    if libname eq explibs[idx] and
       memname eq expmems[idx] then put 'PASS test ' idx;
    else put 'FAIL test ' idx;
  end;
run;
```

Thus, at the beginning we have a program, and a testing strategy based on our expectations of that program. Note the structure of the tests: typical inputs are pre-loaded in the TwoLevel array; expected results are split into the component arrays ExpLibs and ExpMems. As written above, the test strategy assumes that the program should ignore case on the input strings, and return results uniformly in upper case. The tests also indicate that the same program should handle both two- and one-level names. All such requirements must be documented in detail prior to writing any code, including the test code. This requirements document is in fact the contract between the users and developers. The tests and eventually the macro definition must be based on the requirements, rather than establish them. At this point, the above code produces the following results:

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

CHOOSING THE RIGHT TEST STRATEGY

It's critical to note that, much like the medical interventions we help develop, there is no panacea for software testing. Unit testing can help, but cannot solve all problems. Note that the above test also assumes that the macro works in a data step and returns data step variables LIBNAME and MEMNAME. These are, of course, critical details which should be based on explicit requirements. To be valid, the testing strategy must also match the intended use of the unit. A quick check of the requirements could reveal that in fact the program should behave as an in-line macro, returning either the libname or the memname, as the calling program requests. In this case, the unit "interface" (parameterized inputs) needs to be modified, and the test strategy completely revised:

```
/* MACRO SHELL */
%macro splitname(itemname, request);
%mend splitname;

/* TEST STRATEGY -- Return LIBNAME */
%LET LIBNAME = %SPLITNAME(SASHELP.CLASS, LIBNAME);
%if &libname = SASHELP %then %put PASS;
%else %put FAIL;
... repeat LIBNAME requests for additional input types ...
```

```
/* TEST STRATEGY -- Return MEMNAME */
... repeat tests for MEMNAME request ...
```

After writing just the first test for the proper requirements, it should be clear that there is again a test pattern that lends itself to looping. In the first example, we used data step arrays to take advantage of this pattern. Now we could use macro language to alignment the test strategy with the intended use:

```
/* MACRO SHELL -- In-line result for LIBNAME or MEMNAME requests */
%macro splitname(itemname, request);
%mend splitname;

/* TEST STRATEGY -- Repeat all tests for LIBNAME and MEMNAME requests */
%macro test_splitname;
  %local requests twolevel libnames memnames
         rdx nextreq ndx nextmem expected;

  %let requests = LIBNAME libname MEMNAME MemName;
  %let twolevel = SASHELP.CLASS SasHelp.Class Work.Temp Temp;
  %let libnames = SASHELP SASHELP WORK WORK;
  %let memnames = CLASS CLASS TEMP TEMP;

  %let rdx = 1;

  %do %while (%scan(&requests, &rdx, %str( )) ne );
    %let nextreq = %scan(&requests, &rdx, %str( ));

    %let ndx = 1;

    %do %while (%scan(&twolevel, &ndx, %str( )) ne );
      %let nextmem = %qscan(&twolevel, &ndx, %str( ));
      %let expected= %qscan(&&&nextreq.s, &ndx, %str( ));

      %let result = %SPLITNAME(&NEXTMEM, &NEXTREQ);

      %if &result eq &expected %then %let result = PASS;
      %else %let result = FAIL;

      %put &RESULT : request &nextreq from &nextmem;

      %let ndx = %eval(&ndx + 1);
    %end;

    %let rdx = %eval(&rdx + 1);
  %end;
%mend test_splitname;

%test_splitname;
```

Now we have an automated set of tests that uniformly FAIL. The above code produces the following results:

```
FAIL : request LIBNAME from SASHELP.CLASS
FAIL : request LIBNAME from SasHelp.Class
FAIL : request LIBNAME from Work.Temp
FAIL : request LIBNAME from Temp
FAIL : request libname from SASHELP.CLASS
FAIL : request libname from SasHelp.Class
FAIL : request libname from Work.Temp
FAIL : request libname from Temp
FAIL : request MEMNAME from SASHELP.CLASS
FAIL : request MEMNAME from SasHelp.Class
FAIL : request MEMNAME from Work.Temp
FAIL : request MEMNAME from Temp
FAIL : request MemName from SASHELP.CLASS
FAIL : request MemName from SasHelp.Class
FAIL : request MemName from Work.Temp
FAIL : request MemName from Temp
```

PASSING THE TESTS

Are we now ready to develop %SPLITNAME? No. It is critical at this point to evaluate the test strategy with a critical

eye. How complete are these tests? Several questions can help form a robust strategy.

Are all the tests necessary? Are important tests missing? Just because we get a lot of results, doesn't mean our strategy is thorough or complete -- i.e., well structured. Redundant tests are not helpful. Individual tests should exercise different program logic. All tests should be reviewed for relevance and the tests overall should be reviewed for coverage of program logic. If you're having trouble writing down all reasonable tests, then maybe the program does too much. Consider a redesign, and likely decomposition into components of reasonable scope. This is also a critical step for maintenance. A single, over-ambitious program is likely a maintenance nightmare as well.

Who will use this program? Handling of unexpected inputs (and therefore testing of that handling) should be more thorough if the program is available to end-users rather than buried within a larger system.

How are the results of the program likely to be used? What language considerations could contribute to program instability? For example, are users of the %SPLITNAME macro likely to use the results in a comparison like "`%if %splitname(&mydata, LIBNAME) eq WORK %then ...`"? If so, then we know from our SAS experience that some results could be mistaken for a macro keyword like OR or AND. The test-writer should therefore include some "unexpected" two-level names like "`AND.OR`", and the developer should consider quoting the result to help users avoid unexpected downstream failures.

What systems will the program run on? Are there language/system interactions that could cause problems. Certainly for SAS programmers the answer is often "Yes!"

After careful consideration and revision of testing strategy, we could finally begin developing the in-line macro definition, and eventually feel satisfied and confident once each of the failures planned above becomes a '"PASS".

THE PAYOFF

The payoff for this up-front effort should be apparent. It does not only help the original developer meet the original requirements while keeping the original scope focused (although these benefit are already immense). When the requirements for this program eventually change, a near certainty, the existing unit tests protect the agreed behavior up to that point. The same care used in writing the original tests is also required in revising and extending those tests to cover new expectations. But much of the effort may already be covered in a good testing structure.

CONCLUSION

We have not done justice to the authors that we rely on and return to regularly for refresher courses on programming's best practices. It would be impossible to do so in a review of their bountiful anecdotes, wisdom and advice. Our goal is instead to whet the appetite and curiosity of our peers, to inspire them to explore the world of dedicated software engineers.

Yes, these technical skills are clearly just one of many skills we must juggle in our careers, where contextual expertise is often mandatory and more relevant. This does not, however, preclude us from benefiting from advances in software engineering, where context can vary greatly and therefore be overshadowed by technical expertise. In fact, we can profit all the more by learning their principles, adapting them to our responsibilities, improving our technical abilities and thereby allowing more time and energy for competing priorities.

More importantly, it should be easy for us to learn these lessons. This means we shouldn't have to explore these topics individually, in our spare time. Our employers should recognize these technical professional needs and the benefits of addressing them. Doing so would generate a pressure on trainers to develop courses that not only deliver technical details, but crucial fundamental concepts as well.

REFERENCES & RECOMMENDED READING

Bentley, Jon (1999), Programming Pearls (2nd Edition), Addison-Wesley, New York, New York (USA), 256 pages.

McConnell, Steve (2004) Code Complete: A Practical Handbook of Software Construction (2nd Edition), Microsoft Press, Redmond, Washington (USA), 960 pages.

Weinberg, Gerald M. (1998) The Psychology of Computer Programming (Silver Anniversary Edition), Dorset House Publishing, New York, New York, 292 pages.

Binder, Christof, and diTommaso, Dante (PhUSE 2009 - TU07), *Take me for a test drive*, http://www.lexjansen.com/cgi-bin/xsl_transform.php?x=phuse09&s=phuse&c=phuse#po.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Dante DiTommaso
Statistical Applications Manager, Team Lead
F. Hoffmann-La Roche Ltd
Methodology and Innovation
Malzgasse 30
CH-4052, Basel, Switzerland
Tel: +41 61 687-4314
dante.di_tommaso@roche.com

Christof Binder
Statistical Applications Manager
F. Hoffmann-La Roche Ltd
Methodology and Innovation
Malzgasse 30
CH-4052, Basel, Switzerland
Tel: +41 61 687-4286
christof.binder@roche.com