A New Paradigm for the Development of Analysis Software

Diane Kelly[1] and John Harauz[2]
[1] Royal Military College of Canada
(kelly-d@rmc.ca)
[2] Jonic Systems Engineering

Abstract

For the CANDU industry, analysis software is an important tool for scientists and engineers to examine issues related to safety, operation, and design. However, the software quality assurance approach currently used for these tools assumes the software is the delivered product. In this paper, we present a model that shifts the emphasis from software being the end-product to software being support for the end-product, the science. We describe a novel software development paradigm that supports this shift and provides the groundwork for re-examining the quality assurance practices used for analysis software.

1. Introduction

Software used in the Canadian nuclear industry for safety, operational, and design analysis of any aspect of a CANDU nuclear generating unit falls under the auspices of the Canadian Nuclear Regulator. For regulation purposes, the software quality standard CSA N286.7 is used to provide the quality requirements for this class of software. The focus of the standard is on activities seen by the software engineering community as those that improve the quality of the software.

Unfortunately, the software quality road that has been followed for so many years for this class of software only tangentially addresses the issue that is the main concern of both the Canadian Nuclear Regulator and those developing and using this software: the quality of the science.

In this paper, we discuss the difference between the viewpoints of ‘quality of science’ and ‘quality of software’ and the impact this makes on both the development of this class of software and appropriate quality assurance activities. The paper also includes a case study with an example of analysis software from the CANDU industry. The case study provides evidence that a new development viewpoint is appropriate. The conclusion opens the discussion on how to change the viewpoint of quality assurance to match the new development viewpoint.

2. Putting Analysis Software into Perspective

Analysis software falls into a larger class of software commonly called scientific software, which itself falls into the class of software known as end-user software.

Scientific software has the major characteristic of embodying a significant amount of scientific knowledge, be it knowledge about fluid flow, large body interaction, forces on bridges, or protein modeling. Most often, because of the necessary scientific understanding, this software is written by a scientist. The purpose of the software is to enable scientific (or engineering) studies. Using software to enable scientific studies is relatively recent compared to the traditional empirical and theoretical approaches to science. The power of computing environments has allowed studies that have been previously prohibitive because of danger, inaccessibility, or the immensity of the computations involved. The software itself is not the goal of the exercise, but the science. The software is a means to an end, not the end.

This puts scientific software into the category of end-user software. End-user software is generally written by the person or group intending to use the software. Their goal is not to market a software product, but to create a tool that enables whatever task is at hand. End-user computing covers a vast spectrum of computing capabilities and interests. Examples vary from the open source development of Linux to individuals using spreadsheets. Segal [20] points out that scientists as end-users are unique and has coined "professional end-user developers" to signify that difference.

Professional end-user developers are characterized [20] as domain professionals having expertise in a scientific discipline. They are comfortable doing software programming and can learn a programming language readily. Their interest, however, is in doing the science related to their expertise, not in writing a software product for the sake of a software product.

Software quality concern comes in at this point. Scientists are not software professionals and examples exist [6, 17] where mistakes in the software have caused problems. There are surprisingly few such examples and the ones that have surfaced have caused the science to go awry. If the science hadn't been affected, the level of quality of the software, measured in any way, would not have been an issue.

Analysis software provides data for reports that advise on the design, safety, and operation of the CANDU generating units. The data from the software is scrutinized and analyzed by professionals and incorporated with the professional's knowledge to advise on design and operational parameters. It is this final advice that is crucial to the safety of the CANDU units. The software itself does not advise in any way, does not directly affect how the generating units are operated, and does not produce a final answer to engineering problems. The software provides data for a particular scenario designed by the scientist. The scenario depends heavily on the scientist's analysis of the situation and the scientists' model of the system. The scientist does an interpretation of the data obtained from the software and does this interpretation in the wider context of the problem domain. The analysis software itself is not the product.

With the analysis software not itself being the product, we have to consider the user of this software in a very different light. The user, that is, the scientist, is a major part of the end product. In order for us to trust the advice of the scientist, the scientist must be allowed to do her science and be fully supported in this endeavour. One essential support is the software tool used to do the scientific study.

Almost all commercial software we use today is treated as a consumable – the user is provided with a finished product and learns how to use it. In other words, the user has no ability to change the software in order to address his or her problem or task. As pointed out in [19], this doesn't work for scientific research. Scientists work on "ill-defined or wicked problems [that] cannot be delegated from domain professionals to software professionals" [4]. In other words, the scientist must be heavily involved in developing the software models. In addition, the scientist must be able to readily make changes to her models and to the way the models are computed. Otherwise, "if systems cannot be modified to support new practices, users will be locked into existing patterns of use". [4] For a scientist investigating an unsolved problem, this doesn't work.

The dichotomy between software consumers and software producers (eg. software engineers) has been recognized by end-user research over the past twenty years [4, 19]. End-user research defines *prosumers* who are 'techno-sophisticated … have little fear of modifying and evolving artifacts to their own needs … do not wait for someone else to anticipate their needs, … can decide what is important for them … [and] participate in learning and discovery and engage in experimenting, exploring, building, framing, solving, and reflecting." This clearly defines a scientist.

The software system appropriate for the scientist working on her typical "wicked problem" fits well with the *prosumer* model. The scientific systems are "focused on the 'unfinished' and take into account that design problems have no stopping rule, need to remain open and fluid to accommodate ongoing change, and for which 'continous beta' becomes a desirable rather than a to-be-avoided attribute." [4] In addition these scientific systems move from "guidelines, rules, and procedures to *exceptions, negotiations, and work-arounds* to complement and integrate accredited and expert knowledge with informal, practice-based, and situated knowledge." [4]

Unfortunately, software quality standards such as CSA N286.7 are meant for consumer software characterized as a delivered product with a definite static termination stage. Users are intended to consume the static product and not dynamically change the software in order to investigate the task they are solving. Current software quality standards do not fit well with software that is essentially in "continuous beta", which is the reality of scientific software.

Fischer et al [3] propose an alternative model suited to end-user developers. Here, we examine Fischer's Meta-Model for its suitability for analysis software. In the next section, we describe the Meta-Model. In Section 4 we present a study that illustrates the appropriateness of the Meta-Model, using a longitudinal study of the analysis software SOPHT (Simulation of Primary heavy water reactor Heat Transport). Section 5 discusses details of a novel approach to developing analysis software and its impact on quality assessment approaches. Section 6 concludes.

3. End-User Software Development for Analysis Software

Fischer et al [3, 4] propose a Meta-Model to support the development of software intended for end-user development. They describe a *seeding, evolutionary growth, reseeding* model (SER) for end-user developers. The first phase builds *seeds* "that can evolve over time through small contributions of a large number of people" [4]. Evolutionary growth is seen as an unplanned (and un-plan-able) phase of growth of the system. *Reseeding* is the phase where deliberate restructuring and enhancement takes place to produce a new set of improved seeds. Fischer [4] quotes von Hippel as saying "Users that innovate can develop exactly what they want".

Fischer intends his Meta Model for all end-user developers including those who have no interest in coding (eg. an example Fischer uses is a set of seed programs that can be used by a kitchen designer). As a result, this model is described in very high-level terms and details are missing how it is to be implemented. Scientists, as professional end-user developers, are on the far end of the prosumer scale where they are very close to being producers. If we consider analysis software as a specific application of Fischer's Meta-Model, then we'd like to learn how to fill in details on the implementation of the Meta-Model.

For analysis software there should be some core part of the software that is the foundation of the computations being carried out. For example, the set of difference equations that underlie the computations and the matrix solution to the simultaneous sets of equations could be parts of the analysis software that are relatively static over the years of evolution. Long term evolution of software is largely unpredictable [18] leaving the software designer only with the ability to predict classes of changes, at best. Included in the seed can be parts of the software that are left open for the analysis engineer to change as needed. Eventually, at appropriate points, parts of the software considered useful to the user base at large can be included in renewed seeds. The design of the software "differentiates between structurally important parts for which extensive professional experience is required [such as experience in computational techniques] … and components which users should be able to modify to their needs because their professional knowledge is most relevant." [4]

One important aspect of Fischer's model is "collaborative work practices rather than focusing on individuals" [4]. The ideas of "communities of practice" have been articulated since about 1999, but they have existed since people have worked together for a common goal [16]. Research has shown that promoting communities of practice has benefits including greater knowledge sharing, better maintained long-term organizational memory, improved handling of unstructured problems, new ideas more readily spawned, decreased learning curve for new employees, and reduced rework and reduced "reinventing the wheel" [16]. Fischer's model of software development provides the focus for a community of practice that supports the use and evolution of the software.

Fischer admits that "there are many open issues about quality and trust in cultures of participation" [4]. Two important points are made in its favour. One is that the community has a greater sense of ownership: open code is continually being peer reviewed, with bug fixes shared and vetted. Second, Fischer claims that the users have increased realization that errors will always exist, resulting in users always being critical of what the computer outputs rather than blindly accepting it.

From the viewpoint of quality assurance, a new approach is needed. A proposed approach is discussed in Section 5.

4. Retrospective Study of SOPHT

In this section, we investigate whether Fischer's Meta Model is appropriate to the development of analysis software. In particular, we look at the appropriateness of the seed-evolution-reseed (SER) model and whether there is evidence that this model fits well with successful development of analysis software.

A thermal hydraulics simulation code, SOPHT (Simulation of Primary heavy water reactor Heat Transport), was examined for changes evident over an eighteen-year period. SOPHT models the thermal hydraulics properties of the primary and secondary heat transport systems of a CANDU generating unit. The authors of the software, J.Skears, T.Toong, and C. Chang received the J.S. Hewitt Award in 1998 for their work. SOPHT is a recognized success story in analysis software.

We examine characteristics of changes that happened to SOPHT to observe if the *seed-evolution-reseed* (SER) type of activity is potentially present, especially the beginning *seed* phase. If so, the SER model could readily be adapted to analysis software.

Several different studies on SOPHT have been previously carried out [11, 13], comparing versions of the code from 1980 and 1998. The studies used the common block data structure as a surrogate for changes in the rest of the code. As explained in [11, 13], this is a viable alternative to characterizing the entire code and provides an accurate picture of the evolution of the entire software.

The structure of SOPHT is typical of scientific software as described by Arnold and Dongarra [1]. Processing for SOPHT starts with transforming the large input data set into an internal format more usable for the software. In SOPHT, the internal data is held in a FORTRAN structure called a common block. The common blocks are made available to the various parts of the software depending on the computations taking place. The significant part of SOPHT is the solution engines whose functioning is driven by the contents of the input data. This combination of extensive input data and engines driven by the contents of the input data make SOPHT flexible for both the engineer doing a study with existing models in the code and for the engineer who has to add new models to the code. The success of this design was evident in previous studies [11, 13] where software "decay" typically described in software engineering literature [eg. 18] was not observed.

Three versions of SOPHT were examined to identify whether a "seed-evolution-reseed" pattern was evident. The common blocks of the 1980 base version of SOPHT were compared to the common blocks of two 1998 versions. One 1998 version (denoted 1998-O) was evolved by code developers who were part of the original SOPHT development team and continued as the de-facto experts for the code. One of their goals was to maintain a generic version of the software for wide use. The second 1998 version (denoted 1998-A) was evolved by a separate team who had one specific use for the code. This second team had direct contact with the first team and freely shared information about the code.

Relevant data is given in Table 1.

In the 18-year study period, Version 1998-O more than doubled its number of common blocks. Nearly half of the new common blocks provide additional functionality that represents new science models. Over half of the existing common blocks in the original 1980 version were extended to also provide new science. This illustrates the dynamics typical and necessary in software used to support scientific and engineering endeavours.

Only 3 of the original common blocks in the 1980 version were deleted. Examination shows that this is not a failure to prune obsolete code, but indicates instead a well-designed original core code.

Six new common blocks were added to the 1998-O version to address concerns about backwards compatibility – there is a regulatory requirement to be able to reproduce studies carried out with the code. Five new common blocks provide version and date identification, again addressing regulatory requirements. Six new common blocks were added to support the breaking up of large subroutines into smaller pieces. [13] discusses whether this was an inappropriate activity, given that there is no research evidence that supports the claim that code size and code errors are related [eg., 2].

In the 1980 version, 28 common blocks are identified as unchanged or changed for housekeeping purposes. Housekeeping includes parameterization of numeric values that potentially can change, reordering of the variables in the common block due to restrictions imposed by a new FORTRAN compiler, clean-up due to change in memory devices, changes due to character type variables becoming available, removal of unnecessary restrictions on arrays, and variable names changed to increase understandability. In the 1998-O version, 18 common blocks were added for housekeeping purposes, largely maintainability and documentation purposes.

Of interest are the 23 common blocks from the 1980 version that were relatively unchanged. This is nearly half of the original set of common blocks. Seven of these are structures used to report input data and intermediate data during calculations. The rest of the relatively unchanged common blocks are all related to the fundamental solution techniques used for SOPHT. There are four data structures for setting up the network model, five data structures for holding state variables, and seven data structures directly supporting the solution techniques, such as time constants, flow matrix, power series calculations, property values, and limiting parameters such as transport times and time intervals.

There is clearly a core portion of the software that is largely unchanged over eighteen years – a seed portion. There are also changes that clearly lend themselves to "reseeding", such as the house keeping activities. The software absorbed extensive new engineering models without "breaking" [11, 13] and was recognized for its contribution to the CANDU industry.

When we examine and compare the changes evident in the 1998-A version, we find that changes are not as extensive and follow a slightly different pattern due to the different goals of the group.

There are 39 common blocks in the 1998-A version that are either unchanged or received minor housekeeping changes when compared to the original 1980 version. Sixteen of these (a significant overlap) are the same common blocks largely left unchanged in the 1998-O version. This supports the identification of "seed" portions in the original design of the code.

Table 1: SOPHT Common Block Changes 1980 to 1998 – Two Versions. For comparison, SOPHT 1980 version had 54 Common Blocks.

Count of Common Blocks	**1998 Version-O**	**1998 Version-A**
Total Number	122	96
Additional Common Blocks after 1980	68	56
Deleted Common Blocks	3	14
Unchanged or housekeeping changes	23	39
New science/engineering models added to existing 1980 common blocks	28	2
New science/engineering models added with new common blocks	33	32
Created for backwards compatibility	6	-
Created for version identification	5	-
Created to support smaller subroutines	6	8
Created for housekeeping purposes	18	16

Since the goal of the 1998-A version was the design of a new station, unused common blocks were deleted, and there was no requirement for backward compatibility or version identification. Over half of the new common blocks are for new science models, again illustrating the need for flexibility in the original software design to meet engineering and scientific needs.

In addition to the software itself possessing characteristics of a meta-model type of design, the community of users and developers during the eighteen-year period had all the characteristics of a "community of practice". Developers and users were in close contact, knowledge and code were shared, code was closely scrutinized with great interest and there was a feeling of ownership by everyone involved [11].

5. A Novel Approach to Quality Assessment

In [12] arguments are made for "innovative standards for innovative software". Two characteristics make analysis software substantively different from consumer software. First, it must be tractable enough to allow the scientist to do her science. Second, *the science not the software* is the product.

Current approaches to quality assurance are a mismatch for analysis software. First, they assume the software is the product; second, they assume the software product is finished and consumable; third, they assume that development process, software metrics, and documentation will significantly contribute to quality. There is no proof that any of this has measurable results [eg., 2, 15, 22].

The approach to quality assurance for analysis software needs to shift in two ways. First the focus needs to be more on the science. The science is the deliverable and the software is a support tool for the scientific decisions. The data from the software should be regarded with the same scientific questioning as data from any other source. We need to credit the scientist for being in-the-loop. She provides the judgment and the expertise for the final scientific decision.

Second, the amount of effort placed on software quality assurance should be directly proportional to the benefits gained. The goal for any quality effort should address the issue of trust. Shapiro [21] comments on software in general that “pragmatism [moves] the question from one of *correctness* to one of *confidence*.”

Kelly and Harauz proposed in [8] a framework for the development of analysis software. The framework has four phases: - **P**lanning Phase - **E**xploration Phase - **T**ransition Phase - **E**volution Phase (PETE). This framework meshes well with the SER Meta Model.

If developing a new piece of analysis software, the first Planning - Exploration - Transition cycle can prepare the *seed* portion of the software. In the Kelly-Harauz framework, the Evolution Phase is a continual repeat of the Planning – Exploration – Transition cycle and corresponds to Fischer’s Evolution in the SER Model. The decision to *reseed* can be carried out within another Planning-Exploration-Transition cycle at the appropriate time. In both the PETE framework and the SER Meta Model, the evolution phase continues with the new *seed*.

The cycle of the Kelly-Harauz framework that produces either the *seed* or *reseed* portion of the software could lend itself to more traditional verification techniques: appropriate types of testing and peer review have been investigated in [5, 7, 9, 10, 14]. During evolution, portions of the software may change with every scientific study. At this point, it is more appropriate for quality assurance to focus on the science, asking questions about assumptions made, data used, models developed, sensitivity of parameters, and comparisons with other appropriate information.

The result is that two different approaches to quality assurance are needed. One approach, for the production of the *seed* portion of the software, could follow more traditional software quality assurance practices, with the recognition that even the *seed* requires an exploratory phase. The second approach, applied during the production of scientific and engineering studies (the evolution phase), needs considerable reframing from what is currently used.

Research is needed to flesh out the details of both approaches. Particularly for the evolution phase, the approach needs to be holistic, focusing not just on the software, but on the whole solution. It needs to put the role of the software into perspective: during *evolution*, software is not the product, science is the product.

6. Conclusions

An End-User Developer model for creating and evolving software appears to be an appropriate fit for analysis type software. A development model proposed by Fischer creates a seed portion of the software that enables its evolution as required by scientists who are dealing with “wicked problems”. Kelly and Harauz also proposed a development model with an evolution phase. The two models mesh well and can provide a basis for a new development paradigm for analysis software.

The largest challenge is developing a new quality assurance approach that fits two very different phases of work in these models. Currently, quality assurance activities are assuming the software is a consumer product and so they focus primarily on the quality of the software. The science is the primary deliverable from the scientists using the analysis software and the science should be the primary focus of the quality approach.

We feel that the development paradigm we have identified provides a better understanding of the work involved both during the development of the software and during its use. Research and discussion is needed to further refine this development paradigm. Importantly, we need to identify activities within the paradigm that provide demonstrable assurance, suitable for inclusion in software quality standards.

7. References

[1] Arnold, D.C. and Dongarra, J.J., “Developing an Architecture to Support the Implementation and Development of Scientific Computing Applications”, in The Architecture of Scientific Software, editors Biosvert, R.F. and Tang, P.T.P.; Kluwer Academic Publishers, 2001

[2] El Emam, K., Benlarbi, S., Goel, N., Melo, W., Lounis, H., Rai, S.N., “The Optimal Class Size for Object-Oriented Software”, IEEE Transactions on Software Engineering, Vol. 28, No. 5, May 2002, pp. 494-509

[3] Fischer, G., Nakakoji, K., Ye, Y, “Metadesign: Guidelines for Supporting Domain Experts in Software Development”, IEEE Software, September/October 2009, pp. 37-44

[4] Fischer, G., “End-User Development and Meta-Design: Foundations for Cultures of Participation; Journal of End-User Computing, 22(1), 2010, pp. 52-82

[5] Hamlet, D., "When Only Random Testing Will Do", Proceedings First International Workshop on Random Testing, Portland, Maine, 2006, 9 pages

[6] Hatton, L. and Roberts, A., “How Accurate is Scientific Software?” IEEE Transactions on Software Engineering, Vol. 20, N0. 10, October, 1994, pp. 785-797

[7] Hook, D. and Kelly, D., “Mutation Sensitivity Testing”, IEEE Computing in Science and Engineering, November/December 2009, pp. 40-47

[8] Kelly, D. and Harauz, J., “Software Development Processes and Analysis Software: A Mismatch and a Novel Framework”, Canadian Nuclear Society Conference in Niagara Falls, Ontario, June 5, 2011; Proceedings of 2011 CNS Conference; 11 pages

[9] Kelly, D., Gray, R., Shao, Y., “Examining Random and Designed Tests to Detect Code Mistakes in Scientific Software”, Journal of Computational Science, Elsevier, Volume 2, Issue 1, pp. 47-56; March 2011

[10] Kelly, D., Thorsteinson, S., Hook, D., “Scientific Software Testing: Analysis in Four Dimensions”, IEEE Software, May/June 2011, pp. 84-90

[11] Kelly, D.,“Determining Factors that Affect Long-Term Evolution in Scientific Application Software”, Journal of Systems and Software, available online Dec 3, 2008; 82 (2009) pp. 851-861

[12] Kelly, D., “Innovative Standards for Innovative Software”, IEEE Computer, July 2008, pp. 88-89

[13] Kelly, D., “A Study of Design Characteristics in Evolving Software Using Stability as a Criterion”, IEEE Transactions on Software Engineering, Volume 32, Issue 5, May 2006, pp. 315-329

[14] Kelly, D. and Shepard, T., “Task-Directed Inspection”, Journal of Systems and Software (JSS), Vol. 73/2, October 2004, pp.361-368

[15] Kitchenham, B. and Pfleeger, S.L., “Software Quality: the Elusive Target”, IEEE Software, January 1996, pp. 12-21

[16] Lesser, E.L. and Storck, J., “Communities of practice and organizational performance”, IBM Systems Journal, Vol. 40, No. 4, 2001, pp. 831-841

[17] Miller, G. (2006). “Scientific publishing: A scientist’s nightmare: Software problem leads to five retractions”. *Science*, *314*(12), 2006, pp. 1856–1857.

[18] Parnas, D.L., “Software Aging”, Invited Plenary Talk, Proceedings 16th International Conference on Software Engineering, 1994, pp. 279-287

[19] Segal, J., "When Software Engineers Met Research Scientists: A Case Study", Empirical Software Engineering, 10 (4), 2005, pp.517-536

[20] Segal, J., “Professional end user developers and software development knowledge”, (Tech. Rep. No. 2004/25). Milton Keynes, UK: Open University, 2004.

[21] Shapiro, S., “Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering”, IEEE Annals of the History of Computing, Vol. 19, No. 1, 1997, pp. 20-54

[22] Shepperd, M and Ince, D.C., “A Critique of Three Metrics”, Journal of Systems and Software, Vol. 26, 1994, pp. 197-210