Software Development Processes and Analysis Software: A Mismatch and a Novel Framework

Diane Kelly
Royal Military College of Canada
kelly-d@rmc.ca

John Harauz
Jonic Systems Engineering
j.harauz@sympatico.ca

Abstract

This paper discusses the salient characteristics of analysis software and the impact of those characteristics on its development. From this discussion, it can be seen that mainstream software development processes, usually characterized as Plan Driven or Agile, are built upon assumptions that are mismatched to the development and maintenance of analysis software. We propose a novel software development framework that would match the process normally observed in the development of analysis software. In the discussion of this framework, we suggest areas of research and directions for future work.

Keywords

analysis software, software quality standards, software development process, scientific software

1. Introduction

Scientific software is used to simulate situations that would be unsafe, too expensive, or otherwise prohibitive to study. In the nuclear industry, scientific software, also known as analysis software, is used to simulate scenarios related to studies in safety, operation, and design. In any of these areas of study, the primary need is that the software returns a trustworthy answer. This is the quality of prime importance and is non-negotiable.

Recent research (eg., [1], [2], [18], [19], [20]) has identified that the class of software that includes analysis software is unique in its characteristics related to development and maintenance. The software engineering community has only come to this realization recently. Unfortunately, the myriad of software development processes, quality practices, software quality standards, and such have been developed based on the characteristics and understanding of classes of software other than analysis software. This paper explains why quality practices and standards in current literature are seriously mismatched to the development of analysis software. Based on our observations, we suggest a novel framework for characterizing the development of analysis software.

The next section provides some basic characteristics of analysis software that impact its development. Section 3 discusses these characteristics in the context of mainstream software development processes. Section 4 presents and discusses our novel framework and Section 5 concludes.

2. Analysis Software

2.1 Definition

The development of analysis software has the following three characteristics. First, the software is written to answer a scientific question. The question can be general, such as, "Is it safe to operate this nuclear generating station?" or specific, such as, "What is the response of this valve under these conditions?" Second, the writing of the software necessitates the close involvement of someone with deep domain knowledge in the application area related to answering the question. The person with the deep domain knowledge, the scientist or engineer, usually writes the software. Third, the software provides output data to support the scientific initiative. A human is in the system loop to examine the data and make observations and ultimately, to answer the scientific question. The expectation is that the data provided by the computer solution is correct and will not misguide the scientist seeking an answer to the question.

Our definition of analysis software excludes the following: control software whose main functioning involves the interaction with other software and hardware; user interface software that may provide the input for and report of scientific calculations; and any generalized tool that scientists may use in support of developing and executing their software, but does not of itself answer a scientific question.

Analysis software may have an extensive graphical user interface or interact with other software or hardware to obtain data. It may run on complex multi-processors and require middleware support to make use of hardware capabilities. But in the following discussion, we are talking exclusively about the code that implements the science application whether it is designed as a separate module or inserted into a product that includes these other parts.

2.2 Characteristics of Analysis Software

The most noteworthy of the characteristics of analysis software is the focus on the need for correctness, or trust, in the software. Some scientists stated baldly, "the software must never lie to me" [18]. As explained by Kelly [14] "This [trustworthiness] takes priority over efficiency, reliability, user-friendliness, cost-effectiveness, and portability … If the software gives the wrong answer, all other qualities become irrelevant." Underlying this need is determinism: under the same conditions, the software must give the same answer. In other words, computations using the software are repeatable.

Technically, it is impossible to prove correctness, or even decide the meaning of correctness, for any complex piece of software. However, we can provide evidence that increases our trust. What evidence we need and how to provide that evidence requires an understanding of analysis software and how it is created.

Output from analysis software is the culmination of a series of interactions amongst scientific models, numerical techniques, data, source code, and hardware [5]. Change of hardware platform or compiler option can alter the values of output. Similarly, different expressions in source code that mathematically are equivalent, can give different values of output, due to the fact that computer computations are neither associative nor distributive. Assumptions in scientific models can interact with constraints in solution techniques, causing unexpected results. Because of this, scientists must spend extensive

time exploring and understanding the implementation of this complex interaction. Development of analysis software is a discovery process, requiring frequent iterations between coding and testing to discover the correct combination of the interacting parts. In addition, correct functioning of the parts of the whole, although certainly desirable, in no way guarantees the correct functioning of the whole. The whole must be thoroughly explored.

The goal of analysis software is to answer a technical question that generally cannot be answered otherwise. As a result, there is no half-measure in the development in the software. Either the software answers the question within acceptable error tolerances – and the software is trusted to do this – or it doesn't. As well, the scope is indivisible in another way. For example, the scientific model cannot be implemented in one release and the solution technique saved for the next release.

3. Three Key Observations on the Development Analysis Software

There are substantive assumptions underlying mainstream software development processes that are not fully discussed, and possibly not realized by those applying the processes to their software development. This section makes three key observations related to these assumptions and discusses why both Plan Driven and Agile development processes are mismatched to the development of analysis software.

3.1 Iron Triangle Model for Software Project Management

The "iron triangle" of project management, first described by M. Barnes in 1969, provides a simple model to understand the parameters that affect the success or failure of a project. In software engineering, "quality" is commonly added (eg., [3]) as a fourth corner, or variable, to accompany the usual variables of scope (functionality), schedule, and cost.

Our observations on analysis software can be mapped onto this enhanced project management model.

Quality for analysis software is primarily repeatable "correctness", or alternatively, trustworthiness. This immutable goal must be achieved. This variable is fixed in the project management plan for analysis software.

Scope is the scientific question that must be answered. This question is often difficult to divide, and so the scope is fixed.

We can argue that cost and schedule are trumped by scope and quality. The output data must be trusted to answer the scientific question. If the scientific question cannot be answered by the software, then the project cannot be completed. A fixed-schedule-fixed-cost environment is very high-risk for the scientist since the only variables left to manage are scope, which may be difficult to sub-divide, and quality, which cannot be compromised.

3.2 Problem Domain Versus Solution Domain

Jackson [11], [12] emphasizes the difference between problem domain and solution domain in developing software. He points out the need for the software developer to

become " ... expert in those aspects of the application [problem] domain that affect the design and construction of the software." ([11], p.251) Jackson emphasizes the importance of extensive and detailed requirements engineering in order for the software developer to fully develop his/her understanding of the problem domain. This has spawned considerable work in trying to find effective means of modeling and expression in requirements documents (see for example, [15], [21]). The emphasis in Plan Driven development models is on the need to explore the problem domain with the aim of reducing the amount of exploration when dealing with the solution domain. The approach is based on the statement that, “Many of the problems encountered in software development are attributable to shortcomings in the processes and practices used to gather, document, agree on, and alter the product’s requirements.” ([21], p. 4)

This runs counter to two realities in the development of analysis software: one is the need for the scientist developing analysis software to explore the solution domain because of the complex interactions amongst the different contributors to the computational solution. Second, the scientist who develops analysis software is already Jackson's "expert" in the problem domain. An extensive understanding of the problem domain is already present in the scientist/developer. As far as expressing their understanding of the problem domain, scientists have a long history of precise vocabulary and notations for documenting their needs and ideas. In the development of scientific software, the problem domain can be precisely captured by the scientist-expert in the domain, allowing attention to be turned relatively quickly to the solution domain, which is where extensive time and attention are required.

3.3 Mainstream Software Development Processes

Boehm and Turner [4] divide software development processes roughly into two categories, Plan Driven approaches and Agile approaches. However, as noted by Segal [20], software professionals generally tailor a methodology to their specific context, but need a base methodology that is a close fit to their situation. We provide a brief description of the two categories of development process, assuming these provide the base methodologies used by software professionals.

The original Plan Driven approach, commonly known as the waterfall model, was first described by Royce [17]. Royce’s premise was to provide a mechanism of discussion, documentation, review, reflection, and testing that guaranteed that the code, when written, matched its requirements. The discovery process happens mostly when the requirements are elicited and specified. The subsequent steps in the process become more and more mechanical, such that the writing of the code should involve little or no discovery. This would otherwise involve an expensive (in both time and cost) backtracking, rewriting documents, and redoing reviews. The assumption is that requirements and design can be specified in detail and that knowledge of the system and its interactions is relatively complete before implementation. Quality assessment is focused on consistency of products against specifications. This approach has no provision in the implementation phase for the extensive exploration needed for analysis software. It also becomes both a time and resource sink if the exploration necessitates changes to fundamental structures in the software. Updating previously documented plans becomes a drain on the resources needed for further exploration of the solution.

Agile methodologies [7] were developed as an alternative to complete up-front planning. In Agile methodologies, iterations are formulated into steps that reduce planning and documentation to the minimum needed to deliver partial products in short time intervals. Agile methods are accompanied by sets of practices designed to fill the communication gap created by the reduction of documentation and up-front planning. Agile requires heavy involvement, on-site and throughout the development cycle, of the end-user or customer to reduce the need for extensive documentation. This may not be fully appropriate for the development of analysis software given the amount of time needed for exploration of the solution phase and where the customer's expertise in the problem domain is of little use.

The two most popular Agile development methodologies are Extreme Programming (XP) [3] and Scrum [16]. The main focus of both Scrum and XP is a short time box, usually less than a month, in which a task is completed. The goal is to deliver a product on time. The accompanying assumption is that the product is readily divisible, the parts can be meaningfully tested, and partial product deliveries are useful to the customer.

Analysis software may not be readily or meaningfully divisible into "deliverables". Extensive testing of parts, although recommended to avoid code errors (eg., [8]), cannot guarantee the integrity of the whole. Three layers of extensive testing is needed: to look for code errors, to verify the solution technique, and to validate the scientific model against the problem being solved. Most of this testing requires a completed product.

Agile methods unanimously focus on delivering a product by fixing cost and schedule and varying scope and quality. This approach puts any development of analysis software in jeopardy.

3.4 Summary of the Impact of Key Observations

The success of a software development process is impacted by its underlying assumptions being properly matched to the key needs of the software being developed. We summarize our observations so far for analysis software and the two mainstream software development approaches.

For analysis software, the overriding project management concern is quality in the form of "correctness" or "trustworthiness". Any software development process whose main focus is schedule or cost can jeopardize analysis software product. This points to the main mismatch between both mainstream development approaches and the development of analysis software.

As opposed to other classes of software, scientists and engineers developing analysis software already have knowledge of the problem domain, well-defined notations for recording this knowledge, and the tradition of mathematics for analysing this knowledge. Any software development process that requires additional documentation of problem domain knowledge runs the risk of requiring redundant documentation that is probably inferior to the traditional notations of the scientific or engineering problem domain.

The largest time sink for developing analysis software is not writing the code, but the "think time" that must be given to choosing the models and solution techniques to be included in the code. Any development process whose main goal is reducing time for coding will not provide any benefits when developing analysis software.

Exploration of the solution space is necessitated by the complex interactions amongst the different elements that make up analysis software. Explorations in the solution space could necessitate changes that may even impact the software architecture for the sake of obtaining a correct solution. Plan Driven approaches assume most exploration is done in the problem space and the implementation step can be mechanized. Plan Driven approaches also assume major changes to the software architecture are all identified well before implementation. Major changes are awkward in Plan Driven approaches and consume substantial resources best spent elsewhere. This points to a mismatch between Plan Driven approaches and the development of analysis software.

Because of these mismatches, it may be detrimental to enforce the associated practices of either Plan Driven or Agile approaches on the development cycle of analysis software without careful consideration of each practice and its underlying assumptions.

4. A Novel Framework for Considering the Development of Analysis Software

The salient characteristics of analysis software inform a different approach to its development and maintenance lifecycle. We propose that there are four overlapping phases to its development lifecycle that make up a characteristic framework:

- Planning Phase
- Exploration Phase
- Transition Phase
- Evolution Phase

4.1 Planning Phase

The Planning Phase involves identifying factors that are “given”, that can only be altered with difficulty or not at all.

An obvious “given” is the statement of the problem to be solved and its scope. Part of this problem statement is defining what success looks like. Always underlying this success is the understanding that the intended software system will be trustworthy.

Constraints on the development are also “given”. If the system must be written in a particular programming language to accommodate use of specific hardware, then this constraint is included in the planning phase. If it’s already known that a particular computational approach is to be used, then this constraint is also included here. However, the questions to be asked here are *not* “what programming language is to be used?” but “what constraints must be dealt with?”

4.2 Exploration Phase

The Exploration Phase involves cycles of different time lengths. There may be long cycles with extensive research involving both problem domain and solution domain. Or there may be short cycles to examine the response of particular choices made in the solution domain.

The iterations in the Exploration Phase are misidentified by software engineers to be “Agile-like” activities. The purpose of these iterations is not to deliver a partial product to the customer on time and within budget. The iterations are to address within the given constraints the problem statement given in the Planning Phase, in order to build a trustworthy system.

The Exploration Phase builds on the general knowledge available at the Planning Phase, Cycles of exploratory iterations allows more specific knowledge related to the system to be accumulated.

Exploration builds specific knowledge in five broad "knowledge domains" introduced by Kelly [13]. The five knowledge domains are summarized in Table 1[13].

Knowledge Domain (KD)	**Description**
Physical world knowledge	Knowledge of physical world phenomena pertinent to the problem being solved.
Theory-based knowledge	Theoretical models that provide (usually) mathematical understanding and advancement of the problem towards a solution.
Software knowledge	Representations, conventions, and practices used to construct the solution.
Execution knowledge	Knowledge of the software and hardware tools used to create, maintain, control, and run the computer solution.
Operational knowledge	Knowledge related to the use of the computer solution.

Table 1. Five knowledge domains that contribute to the development and use of a software product.

During the Exploration Phase, developers with appropriate knowledge need to be available at the right time to expedite the exploration. We can likely assume scientists and engineers are available with appropriate Physical World and Theory-based knowledge.

There has been considerable discussion around appropriate Software and Execution domain knowledge for scientists. Segal [20] has observed situations where software engineers have been included in a development team to fill in missing Software Knowledge. This mix of expertise has not worked well due to impediments in transferring knowledge between people with disjoint understandings. Impediments included cultural, informational, and cognitive differences. Unless the software engineer has a deep understanding of the scientific domain and of developing analysis style software, the mix doesn't work. Other solutions are to train scientists to be software knowledgeable. However, the type and extent of appropriate software knowledge for scientists has not been adequately explored and is an area of research that needs attention.

Despite these possible difficulties, the product of the Exploration Phase is a working system whose goal is to address the factors set out in the Planning Phase. The byproduct of this Phase is the extensive specific knowledge accumulated by the system's developers.

4.3 Transition Phase

As pointed out by Sanders and Kelly [18], analysis software can be associated with three areas of risk: the science and theory embedded in the system, the implementation, and the use of the system. The strength of knowledge of the developers and in-depth exploration of the interaction of components of the system during the Exploration Phase contribute to reducing all three risks. The goal of the Transition Phase is to capture and transfer existing knowledge and technical details of the system to those using the system and evolving it.

Risk is high when the system leaves the hands of the original developers. Tacit knowledge gained by the developers throughout the Exploration Phase is extremely difficult to capture (eg. [6]) and pass on to subsequent users and developers. Unfortunately, there has been no research into the most effective means of transferring useful knowledge for systems such as analysis software. Frequently, we see documents typical of Plan Driven approaches used to transfer knowledge about analysis systems. There are three problems with this.

First, research has not established which pieces of information are most effective in transferring useful knowledge for use or successful evolution of any software system. In particular, there has been no such research related to analysis software.

Second, Plan Driven processes provide documentation for a software system during its *dynamic development*, specifically development following the Plan Driven process. If the Plan Driven process is inappropriate, then so is its documentation system. This can be seen from the descriptions of two typical documents included in Plan Driven development, the Requirements Specifications and the Design Specifications.

According to the IEEE Standard on the “Recommended Practice for Software Requirements Specifications” [9], the benefits cited are “… to consider rigorously all of the requirements before design begins…” This presumes no implementation phase exploration is required and clearly is a product belonging to a development process very different from that required for analysis software.

Similarly, the IEEE Standard on the “Recommended Practice for Software Design Descriptions” [10] states that the software design description “… is a translation of the requirements into a description of the software structure … [that] becomes a detailed blueprint for the implementation activity.” Again, there is the presumption of no significant exploration in the implementation phase.

Third, the production of the Plan Driven set of documents is mismatched time-wise with what is needed for analysis software. Plan Driven documentation is produced before implementation. For analysis software, it is the software product and the knowledge available after the Exploration Phase that is vital to both use of the system and continued evolution of the system.

The accumulated knowledge available at the end of the Exploration Phase needs to be captured, particularly with the goal of addressing the three areas of risk for analysis

software: theory, implementation, and use. We need to explore a combination of traditional ways of knowledge transfer (documents, scientific and mathematical notation) and novel ways to achieve these goals.

4.4 Evolution Phase

Each planned change for any example of analysis software returns the developer to the previous Phases in this process. The Evolution Phase is really a “do while …” loop that continues for the lifetime of the software and includes the previous three Phases inside the loop. As evolution of the system continues, knowledge that is continuously accumulating needs to be captured and transferred to new developers and users of the system. This transfer of knowledge needs to be effected in an efficient and useful manner. This is an area of research that has not been explored. On-going transfer of knowledge both for continued evolution of the software and for successful continued use of the software may need a blend of new ideas with mathematical notations and traditional media.

5. Conclusions

Existing mainstream development processes, Agile approaches and Plan Driven approaches, are based on assumptions that make them ill suited for the development of analysis software. Project priorities, timing, general knowledge requirements, exploration needs and accumulation of specific knowledge are all vastly different for analysis software as compared to other classes of software. Similarly, these differences underlie the mismatches of analysis software development with existing development processes.

We propose a new framework that acknowledges the characteristics of analysis software and the existing strengths of its development. Both the extensive exploration of the problem space and existing traditional methods of precise documentation are to be encouraged. The domain knowledge of the scientists and engineers involved in the software development are essential. The framework recognizes the management priorities that must exist for a successful project, the flexibility required for defining the initial problem statement, and the point at which knowledge capture is key.

A development framework that meshes well with the salient characteristics of building analysis software can provide the basis for an efficient and effective software quality standard and for recommendations for suitable development practices. The framework clearly points to where further research is needed in meeting these goals. Questions about effective and efficient means of knowledge transfer during the Transition Phase, appropriate software knowledge to support scientists during the Exploration Phase, and practices that contribute meaningfully to trustworthy software throughout all Phases, all need to be explored. We have a long history of successfully developing analysis and other types of scientific software. We need to consolidate the knowledge we have already accumulated. This framework provides a basis for moving forward.

6. References

[1] Ackroyd, K.S., Kinder, S.H., Mant, G.R., Miller, M.C., Ramsdale, C.A., Stephenson, P.C.. "Scientific Software Development at a Research Facility", *IEEE Software*, 25 (4), 2008, pp. 44-51

[2] Basili, V., Cruzes, D., Carver, J., Hochstein, L., Hollingsworth, J., Zelkowitz, M., & Shull, F., "Understanding the High-Performance-Computing Community: A Software Engineer's Perspective", *IEEE Software*, 25 (4), 2008, pp. 29-36

[3] Beck, K., *Extreme Programming Explained*, Addison-Wesley, NJ, 2000

[4] Boehm, B. & Turner, R., *Balancing Agility and Discipline*, Addison-Wesley, MA, 2005

[5] Boisvert, R.F. & Tang, P.T.P. eds., *The Architecture of Scientific Software*, Kluwer Academic, 2001

[6] Collins, H., Tacit Knowledge, "Trust, and the Q of Sapphire", *Working Paper Series, Paper 1, Cardiff University*, 2000

[7] Highsmith, J. & Cockburn, A., "Agile Software Development: The Business of Innovation", *IEEE Computer*, 34 (9), 2001, pp. 120-122

[8] Hook, D. & Kelly, D., "Testing for Trustworthiness in Scientific Software", *2nd International Workshop on Software Engineering for Computational Science and Engineering*, Vancouver, BC, *Proceedings International Conference on Software Engineering*, 2009

[9] IEEE Standard 830-1998 "IEEE Recommended Practice for Software Requirements Specifications", IEEE, 1998

[10] IEEE Standard 1016-1998 "IEEE Recommended Practice for Software Design Descriptions", IEEE, 1998

[11] Jackson, M., *Problem Frames*, Addison-Wesley, England, 2001

[12] Jackson, M., "Problems, methods, and specialization", IEE Software Engineering Journal, 9 (6), 1994, pp. 249-255

[13] Kelly, D., "Determining Factors that affect long-term evolution in scientific application software", *The Journal of Systems and Software*, 82 (2009), 2009, pp. 851-861

[14] Kelly, D., "Innovative Standards for Innovative Software", *IEEE Computer*, 41 (7), 2008, pp. 88-89

[15] Pfleeger, S. L., *Software Engineering Theory and Practice*, Prentice Hall, NJ, 1998

[16] Rawsthorne, D. & Shimp, D. "Scrum in a Nutshell", Scrum Alliance, 2009, retrieved 19 February 2010 from http://www.scrumalliance.org/articles/151-scrum-in-a-nutshell

[17] Royce, W., "Managing the Development of Large Software Systems", Proceedings IEEE WESCON, August 1970, 328-338

[18] Sanders, R. & Kelly, D., "Scientific Software: Where's the Risk and how do Scientists Deal with it?", *IEEE Software*, 25 (4), 2008, pp. 21-28

[19] Segal, J., “Professional end user developers and software development knowledge”. *Open University Technical Report No.: 2004/25,* 2004

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

[21] Wiegers, K.E., *Software Requirements*, Microsoft Press, Redmond, Washington, 1999