The Journal of Systems and Software 109 (2015) 50–61

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software

Diane Kelly*

Royal Military College of Canada, Canada

ARTICLE INFO

Article history:
Received 29 July 2014
Revised 6 July 2015
Accepted 14 July 2015
Available online 19 July 2015

Keywords:
Scientific software
Software development
Knowledge model

ABSTRACT

This paper presents a model of software development based on knowledge acquisition. The model was formulated from 10 years of studies of scientific software and scientists who develop software as part of their science. The model is used to examine assumptions behind software development models commonly described in software engineering literature, and compare these with the observed way scientists develop software. This paper also explains why a particular type of scientist, one who works in a highly risk-averse application domain, does not conform to the common characterization of all scientists as "end-user programmers". We offer observations of how this type of scientist develops trustworthy software. We observe that these scientists work outside the ubiquitous method-based software development paradigms, using instead a knowledge acquisition-based approach to software development. We also observe that the scientist is an integral part of the software system and cannot be excluded from its consideration. We suggest that use of the knowledge acquisition software development model requires research into how to support acquisition of knowledge while developing software, how to satisfy oversight in regulated application domains, and how to successfully manage a scientific group using this model.

1. Introduction

Scientific software development has been characterized as end-user programming (Segal, 2004), considered a candidate for Agile iterative development (e.g., Ackroyd et al., 2008), and has been regulated with waterfall-style software quality development standards (Canadian Standards Association). Scientists themselves characterize their development approach as "a-methodical" (Truex et al., 2000). This confusion of views of scientific software development hampers the creation of useful and useable tools, quality standards, and software development paradigms for scientists. Our aim in this paper is to (1) describe the common characteristics of the scientific software developer that we encountered in our studies, (2) argue that these scientists do not fall under the definition of "end-user programmers", and based on our studies, (3) offer a different model of what drives software development by these scientific software developers, and (4) provide a new understanding of the software engineering research that would benefit this type of scientific software development and use.

For this paper, we define scientific software as application software that includes a large component of knowledge from the scientific application domain and is used to increase the knowledge of science for the purpose of solving real-world problems. We use the word "scientific" to include engineering applications.

Scientific software, by our definition, includes examples such as software to model loading on bridges, study safe operation of nuclear plants, track paths of hurricanes, locate satellites in telescope images, check mine shafts for rock faults, model medical procedures for cancer treatment, model dispersion patterns for toxic particulates, and study ocean currents for ecological impact.

The term "scientific software" has been used for a wide variety of software types that do not share the same quality requirements or the same management priorities. Software written to become a commodity product, for example, is managed to meet delivery dates and budget constraints. Software written to verify the safety of a radiation procedure, has to be correct, to the exclusion of all else.

We also exclude from our definition, software whose primary purpose is to control equipment. As explained in Kelly (2008), the quality goals of software that controls potentially dangerous equipment, such as avionics and nuclear reactor shut-down software, are different from the quality goals of scientific software that computes models of physical phenomena, such as tracking the path of a hurricane. If there is a failure in avionics software, the preference is that the

* Tel.: +1 613 541 6000×6171.
E-mail address: kelly-d@rmc.ca

http://dx.doi.org/10.1016/j.jss.2015.07.027
0164-1212/

software degrades as gracefully as possible. If there is a failure in software tracking severe weather, the preference is that it crashes and makes the problematic calculation as obvious as possible. One side effect is that any software quality standards targeted at control software are inappropriate to be applied to scientific software.

We also exclude generalized tools from our definition. Even if the tools are primarily intended for use by scientists, for example mathematical libraries, software layered to hide the complexity of high performance computing environments, and fourth generation languages intended for scientific computation. We include, instead, the applications built "on top" of these tools, which are aimed at solving a particular scientific problem.

To clarify, we further characterize scientific software with the following:

(a) a scientific domain specialist is necessarily involved in the process of developing the software;
(b) the user of this software has some minimum knowledge of the associated scientific domain, to allow correct interpretation of the output data;
(c) the user is the recipient of all output from the software, meaning the software's purpose is not to control equipment;
(d) the software's primary purpose is to provide data for understanding specific real world problems, meaning that the scientists we study do not develop generalized tools and libraries to support computational computing;
(e) the overriding software quality is correctness – or more accurately, trustworthiness – and if trustworthiness fails, then all other software qualities are irrelevant.

This paper is organized as follows:

The next section describes the set of studies of scientific software developers that we carried out from 2004 to 2014. This body of work provides the background for our understanding of the scientific software developer, and a basis for our discussions in the balance of this paper.

Next, we contrast our findings with the commonly held characterization of scientists as "end user programmers". Ko et al. (2011) provide a definition and detailed discussion of the characteristics of "end user programmer". We argue that end user programmer does not provide an accurate characterization of the type of scientist we are studying. Hence the body of research on end user programmers cannot be applied unilaterally to this type of scientist and their software.

In Section 4 of this paper, as an alternative to the view of scientific software developer as end user programmer, we offer a model of scientific software development based on the scientist acquiring knowledge from five knowledge domains. We are not proposing a new process theory, but an empirical example of an alternative to "methods". We use our knowledge domain model to explain how approaches based on methods assume a fragmentation of knowledge that is detrimental to the development of scientific software.

In Section 5, we discuss, from our studies, the activities scientists engage in to advance their knowledge and to maintain trust in their scientific software, while not engaging in methods.

Finally, Section 6 concludes with a summary of the contributions of this paper.

2. A set of studies of scientific software development

2.1. Overview of a synthesis of research

From 2004 to 2014, we carried out a variety of studies looking at different aspects of scientific software development. In this section, we give a brief description of each study, a list of references that provide further details, and explain the findings salient to the discussion in this paper. The discussions in this paper are a synthesis of this work.

The studies took different formats from open-ended interviews of a group of scientists to working with and observing one scientist. Our findings are consistent across all studies, that the scientists developed viable software practices, were well conversant with their software and hardware environments, had adapted testing strategies that integrated with their scientific goals, and did not follow any "systematic" software engineering paradigm.

2.2. Longitudinal study of an example of nuclear software

In the first study (Kelly, 2009, 2008, 2004), we examined the structural characteristics of an example of nuclear simulation software, over a 20 year period of its lifetime (1980–2000). Lehman's Second Law of Software Evolution (Lehman et al., 1998), states that, "As a [software] system evolves its complexity increases unless work is done to maintain or reduce it." Yet, we found no evidence of increasing complexity such that it detrimentally impacted the maintenance and use of the software, even though nearly every line of code had been changed, an observation made by others dealing with scientific software (Boisvert et al., 2000). Instead, we found the software exhibited stable parts that "provide a firm foundation for continued development and change" (Kelly, 2008). The stable parts of the software reflected ties to the application domain. Program design and use of data structures mapped readily to application domain concepts. This agrees with work done by Guindon (1990) who found that domain specialists tend to organize "knowledge structures by functional characteristics of the domain" in which they are specialists. Variable names established in the original version persisted through the 20 year evolution and reflected an established vocabulary used by the software developers. The software architecture used for the original 1980 application is typical of scientific software (Boisvert et al., 2000), is well understood, and is relatively simple. The software successfully evolved over 20 years, yet it was maintained by scientists with no formal software engineering education. The success seemed to be due to their using simple, well-understood software architectures, conceptually linking structures to the science, and maintaining consistent scientific and software vocabulary. There was uniform, unwritten agreement on how to make changes to and extend this software. The study suggests, amongst the scientists who worked on this software, an understanding of coding style, naming conventions, and software architecture, that endured for 20 years and across scores of people.

2.3. Regression testing of an academic nuclear program

The next study involved a software engineer working with a team of nuclear scientists to set up automated regression testing of their application (Cote, 2005). The software engineer was working with an academic copy of an application that had been successfully commercialized. The academic copy was to further evolve as new scientific models became available. The software engineer was frustrated in her efforts to apply standard software engineering testing practices (e.g. Jorgensen, 1995) to this example of scientific software. Equivalence class testing was impossible to apply because of the difficulty of reliably defining input data boundaries. Some calculations were extremely sensitive to any changes in data or code, including changes in compiler options. Only the scientist, not the software engineer, could decide pass/fail criteria for regression tests, due to the sensitivity of some calculations, floating point round-off error in all calculations, and varying required accuracy for different output values. Run times for some executions of regression testing were much longer than that recommended in standard software engineering. This exercise illustrated the difficulty and extensive effort required in applying software engineering systematic testing to scientific software.

2.4. Interviews of academic scientists who develop software

In 2008, we published our results from a series of interviews of scientists working at two academic institutions. Our interviewees "seldom discussed design as a distinct step in software development" (Sanders and Kelly, 2008). We also found that "scientists didn't consider redesigning [their software] a valuable use of their time. Scientists generally want to do science, not write software, and certainly not introduce [error] by changing software that works." When questioning scientists about their software testing practices, we found that the scientists considered that "the code is tightly coupled to the theory", testing is used "to show that their theory is correct", that problems uncovered by testing "might be with the theory, the theory's implementation, the input data, or the [expected] result. All the scientists we interviewed doggedly pursued causes for their output not matching [expected results]". From our interviews, we suggested that, "We need research in test-case selection methods that deals realistically with computational singularities, long runtimes, large input data sets, extensive output, and software with complex domain content." The scientists we interviewed considered the software as inseparable from their science and testing was a key part of doing science. None of their testing could be clearly characterized as any of the systematic testing practices described in the software engineering literature.

2.5. Study of an astronomer and his testing approaches

Subsequently (Kelly et al., 2011), we paired a software engineer with an astronomer who had inherited software in order to add a major new model. The software engineer was interested in assessing the effectiveness of unit testing and code inspection in the context of scientific software and output accuracy. The astronomer, as a developer, wanted to assess his ability to alter the software without destroying the trust he needs while using the software. The results of this study provided us with "… a more complete picture [of] the differences in concepts and priorities between testing as it's described in the software engineering literature and as it's applied to a scientific application" (Kelly et al., 2011). The astronomer abandoned the unit testing procedures in favor of activities that allowed him to increase his intimate knowledge of the software. He invented a new activity that melded together code inspections with scenario-based testing, based on the scientific goals of the software. He created a library of exemplar test cases that could be used in the future and identified high-risk types of mistakes and how they could impact his work as a scientist. The scientist's goal for the testing exercise was to increase his own knowledge of the software, but then use that knowledge to assess his trust of the software. The increasing knowledge of the scientist was an integral part of the assessment process. With the scientist considered as part of the software system, the goals, the software assessment techniques, and the adequacy criteria for the exercise all work towards improvement of both the scientist's knowledge and the trustworthiness of the software. Two key findings emerged from this study: one, the scientist must be considered as part of the software system and, two, their testing is not product based, but knowledge based.

2.6. Examination of testing scientific software for accuracy

We picked up the thread of research on test case selection for scientific software. Hook, in his thesis research (Hook, 2009; Hook and Kelly, 2009) created a large number of "mutant" variations of computational modules each with different code mistakes. He then ran a large battery of tests to find the mistakes embedded in the mutant pieces of code. The surprising outcome of this study was that a very small number of "well-chosen" tests could identify the vast majority of the mistakes in the mutant codes. The issue was to identify a-priori "well-chosen" tests. Gray (Gray, 2010; Kelly et al., 2011) used a number of different standard software engineering testing techniques to discover what a "well-chosen" test was. He concluded that randomly created tests with values within valid and reasonable ranges worked as well, if not better than systematic testing. However, there seemed to be a need to augment the test cases with "well chosen" tests for "boundaries". The testing of boundaries of physical phenomena being modeled by the software played a key role in selecting effective test cases. This meant that the tester had to have some knowledge of the science embedded in the software. Hook concludes in his thesis that, "Computational scientists need domain specific testing techniques" (Hook, 2009). This suggests that scientists may routinely create small numbers of "well chosen" tests instead of a large battery of systematic tests, using their knowledge of the science and the algorithms in the software. This appears to be a viable alternative to software engineering coverage-driven systematic testing.

2.7. Interviews of industrial scientific software developers

In 2012, we documented (Kelly, 2013) a series of interviews of industrial scientists who work in high-risk application domains. The goal was to create a list of software development activities that these scientists deemed useable and successful in their software development efforts. The premise is that these scientists must achieve success with their software or risk catastrophic events in their application domains, which included radiation therapy, nuclear power safety, and mine safety. As part of this study, we sent out a survey to medical physicists questioning them about their software development activities (Kelly and Salomon, 2015). Both these studies revealed that the scientists spent extensive time testing and are worried about their testing. The software was not conceptualized as a separate entity from their application theory or from its use in the application domain. As a result, testing activities were integrated with understanding of the application domain. The tests cannot be categorized according to software engineering terms since each scientist designed unique approaches to testing to address the problem at hand. Testing often was needed to address the difficulty of reproducing published theoretical results. Theory, despite common beliefs, does not translate directly to code and "you have to inject a pile of engineering" to get it to work (Kelly, 2013). As well, we observed that the process of software development, investigation of scientific theory, and use of the software was a seamless activity with no clear breaks. The software engineering concept of dividing software development into discrete steps, tasks, phases, or iterations was not observed. The conceptualization, development, and use of the software were often carried out by one, or a highly integrated team of scientists, to produce one integrated product: the answer to the scientific question.

The seamless activity of moving from problem to solution via software was supported by "cooperation, communication, and negotiation" (Kelly, 2013). The industrial scientists talked about "a lot of continuity, hallway conversations, and collaborative work", that teamwork was essential with people eager to help each other, and that "silos don't work" (Kelly, 2013).

2.8. Knowledge dissemination in a medical computing lab

In spring of 2014, we interviewed people from a medical computing lab (Kelly and Skordaki, 2014). The purpose of this study was to examine how knowledge was disseminated amongst the group of scientists and students who developed or used software in the lab. There was no mention of any standard software engineering documents such as design or requirements. The overwhelming source of information was the human practitioner, with support documentation coming in a wide variety of formats and content, sometimes specifically designed for the needs of the user/developer. There was no indication of discrete steps or phases in their development

work – only one continuous activity from scientific problem to answer. The most important information was who had the necessary knowledge in order to move the work forward. Knowledge flow was supported by organization factors that contributed to social communication. Communication was stressed, but likely because of the supervisor–student authority structure, teams did not exist as in the industrial structure.

2.9. Validating five knowledge domains

All the scientists' work was focused on gaining knowledge in order to answer the scientific question. Early in our observations, it was apparent that knowledge came from five distinct domains (described in detail later in this paper). We presented these ideas in workshops at the IBM Center for Advanced Studies Conferences in 2003, 2009, and 2010, as well as in various presentations to scientists. The five knowledge domains were the basis of an analysis of long-term evolution of an example of scientific software (Kelly, 2004). The graphical representation (Fig. 1, below) of the five knowledge domains was refined with input from the scientists.

Our studies over the past 10 years presented us with a puzzle. Scientists' approach to software development is at odds with the method-based approaches used by software engineers (Segal, 2005). Yet, the scientists we interacted with, did extensive testing, and many were very conversant with coding, their compilers, and their operating systems. They were all well aware of the impact of mistakes on their science but usual software engineering concepts (e.g. requirements specification, unit testing, test coverage, design documentation, design reuse, etc.) were missing from their dialogues with us.

We returned to statements we made in an earlier publication: "… scientific software includes the scientists as an integral part of the system." The approach to software development "must include the scientist's knowledge and goals" (Kelly et al., 2011). This means that everything the scientist does includes the state of his/her own knowledge relating to the scientific question being answered. There is a continuum from initially posing the scientific question to producing the final answer. The software, as well as the scientist, is a part of the continuum. It is this unbroken continuum we need to understand.

2.10. A characterization of risk-averse scientific software developers

From our studies, we extracted common characteristics of the scientists and how they work. These characteristics are important for understanding the parameters of our current research and of the conclusions that we draw.

To characterize the scientists we studied, we derived a set of dimensions (Kelly, 2013) based on our research. Using these dimensions, we describe these specific scientists:

(i) When the scientific application area is highly risk-averse, mistakes in the software can negatively impact the safety of the public or the health/safety of an individual. We observe that the scientist applies this risk-averse mindset to his/her software goals and only engages in software practices that limit risk-taking.

(ii) When there is deep and complex domain knowledge implemented in the software, output is dependent on successful interaction of different parts of the software such as input data, solution techniques, approximations and empirical models, hardware characteristics, and compiler choices. The output requires significant domain knowledge for correct interpretation. We observe that none of these scientists consider themselves software experts, yet they have a deep understanding of their own particular software environment, including hardware, data, operating systems, compliers, etc.

(iii) The scientific developer has formal education in the application area, and as such, it is their area of choice for their career. After significant investment of time and effort into associated software, the scientific developer is usually committed to the software for 10 or more years. We observed that this impacts the scientists' ways of addressing long-term maintainability and knowledge sharing. They build up resources for understanding their environment and ensure the software clearly communicates its contents.

We term these scientists as risk-averse scientific application developers.

An important observation is that these scientists use the knowledge that they gain from both developing and using their software in order to answer their scientific question. While developing their software, they may find that an established empirical model does not address a new situation, or published theoretical results cannot be reliably implemented in code as published.

As application developers, these scientists see themselves as an integral part of both the software system and the answer to the scientific question. Often, they see themselves "on the firing line", that is, responsible for the answer they give to the scientific question, and by inference, responsible for their own knowledge and understanding, for the data that they base their knowledge upon, and for the software that has produced the data.

The impact of the scientists' answers, knowledge, data, and software can be on a large segment of the population, or can seriously affect the health or safety of individuals. As a result, the software cannot be managed as if it is divorced from both the scientist and the public at large. The product of the software activities is the knowledge of the scientist and the users of the software are the recipients of this knowledge.

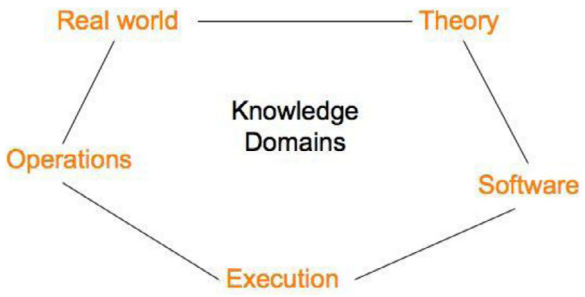

Fig. 1. Model of five domains that contribute knowledge to software development.

3. Scientists as professional end-user programmer

The most ubiquitous characterization of scientists who develop software is as end-user programmer. This allows the software engineering community to slot scientists into a body of research to help understand and recommend software engineering approaches to improve the scientists' software work. The most obvious reason to characterize scientists as end-user programmers is because they do not consider themselves to be in the software business. Segal (2004) refined the label to "professional end-user developer" where scientists are characterized as working in highly technical and knowledge-rich environments, are proficient with formal languages and abstractions, and have few problems with coding and learning software languages. Segal (2005) illustrated the differences – and the conflicts – in development approaches between software engineers and scientists, where it is concluded that the software itself is not valued by the scientists and "the process of creating software [is] highly iterative and unpredictable" (Ko et al., 2011).

In this section, we argue that the type of scientists we consider in this paper are not "end-user programmers", by the definition given

by Ko et al. (2011). This allows a comparison and further clarification of the differences between Ko et al.'s definition of end user programmer and the characteristics of the scientist developers included in our study.

3.1. Definition from literature of end-user programmer

Ko et al. (2011) defined an end-user programmer as "programming to achieve the result of a program primarily for personal, rather than public use". They emphasize that end-user programmer refers to the intent of the programming task, not the programmer's identity nor the tools they use, nor whether the task is related to work or personal endeavors.

Ko et al. (2011) continue to characterize an end user programmer as follows:

(i) "the tool and its output are intended to support the developers' particular task, but not a broader group of users or use cases";
(ii) "… the key difference between professional software engineering and end-user software engineering is the amount [sic] attention given to software quality concerns";
(iii) '… professional software developers spend more time [than end user programmers] testing and maintaining code than developing it and they often structure their teams, communication, and tools around performing these activities …";
(iv) "systematic and disciplined activities that address software quality issues are secondary to the goal that the program is helping to achieve", "… systematic [testing] is often in conflict with end-user programmers' goals, because it requires time on activities that they usually perceive as irrelevant to success";
(v) "end-user software engineering can be characterized by its unplanned, implicit, opportunistic nature, due primarily to the priorities and intents of the programmer …";
(vi) "they [end-user programmers] may be less likely to learn formal languages in which to express requirements …";
(vii) "requirements and design in end-user programming are rarely separate activities";
(viii) "Regression testing has been used in relation to spreadsheets; beyond this, these approaches have not been pursued in end-user development environments";
(ix) "Where EUSE [end-user programmers] and professional SE [software engineers] differ is that end-user programmers' priorities often lead to overconfidence in the correctness of their programs";
(x) "Many [end-user programmers] lack accurate knowledge about how their programs execute and, as a result, they often have difficulty conceiving of possible explanations for a program's failure … [using] 'quick and dirty' solutions, such as modifying their code until it appears to work."

This list distinguishes between end user programmers and professional software developers in the hopes that software engineers are able to provide guidance in the form of currently accepted software engineering practices.

3.2. Risk-averse scientific application developers and end user programmers

In this section, we compare each of the characteristics listed by Ko et al. (2011) with the characteristics we observed amongst the scientists we studied. We argue that unilaterally including all scientists as end user programmers ignores significant differences between our group of scientists and those who apparently conform to the descriptions of end user programmers. Our comparison serves to further characterize the scientific developers we are discussing in this paper.

First, the distinction of software developed by an end user programmer as "personal, rather than public use" is problematic. All programming done by a scientist as part of his/her professional work can be considered to be public use. The data generated by his/her program "is used to increase the knowledge of science for the purpose of solving real-world problems" (from our definition in Section 1). Whether the data increases the scientist's personal knowledge or whether the data is passed along to others for further processing, does not lessen the potential impact of that data and its accompanying gain of knowledge (or failure to gain knowledge).

(i) Number of users

Because of the domain knowledge necessary to understand its use, scientific software seldom has the number of direct (hands-on-the-keyboard) users as other types of software products, particularly commodity-based products. However, the concept of "use" of scientific software needs to be far more encompassing than that of other types of software. Data from scientific software has to be interpreted by the scientist in order to make actual use of it. The scientist becomes an extension of the data – and hence an integral part of the software system. The general public cannot look at data coming from medical physics software and decide if a radiation treatment plan will be effective. But the physicist can and will act upon that data, to the benefit of the cancer patient. The cancer patient becomes the user of that data/human-scientist system. Similarly, the general public cannot interpret output from software tracking a severe weather system, but a meteorologist can. The data/meteorologist software system can produce weather warnings for the benefit of the public. There are two shifts of thinking necessary here. One is that the scientist must be an integral part of the software system in order for the system to be useful. Second, users of this scientist/software system are the recipients of such products of the system as radiation treatments, weather warnings, safe bridge designs, and assurance that the walls of a mineshaft are intact. The use of products from scientific software directly impacts the public at large. Even though one scientist may develop and act as the hands-on user of a particular piece of scientific software, the software system cannot be considered, based on the number of users affected by the scientist/software extended system, the same as a baseball coach tracking his team's statistics on a spreadsheet.

(ii) Software quality concerns

From all the scientists that we interacted with, the common software quality concern was "trustworthiness". All the scientists equated their professionalism with trust in the software and trust in the data produced by the software. One scientist stated that the software must never lie to him. If they could not trust the software, it was not used. If they had access to the code, we observed that their pursuit of anomalies in the software was relentless. The lack of quality concerns attributed to end user programmers was not observed.

(iii) Activities other than coding

All scientists we interviewed or observed spent extensive time testing their code, reading their code, and discussing different aspects of theory and use of their code. They endeavored to keep their code as simple as possible in order to reduce the amount of coding time, and to reduce the clutter that could hide the science (e.g., Boisvert et al., 2000). Teams working on a code product were highly functional, each team member bringing something unique to the table. The teams were not organized around software engineering processes or tasks, but around knowledge useful for the project.

Team organization is different from that of software engineers, but time spent on activities other than programming

may be proportionally even greater than that credited to software engineers. One indicator is that the concept of simplicity seems to be more prevalent in discussions of scientific software than in discussions about software engineering (Floyd and Bosselmann, 2013). Programming simplicity supports spending more time elsewhere other than coding. Qualitative observation suggests this time is spent on testing and hypothesis building.

(iv) Systematic activities addressing quality

Systematic activities as prescribed by software engineering were generally not used by any of the scientists we observed or interviewed. Instead, the scientists used a wide variety of approaches, often unique, to specifically address the problem at hand. Their approaches were often integrated with the science, such as naming variables the same as scientific quantities, verifying code against theory equations, validating output against real world measurements, testing algorithms against known solutions, and using data structures that reflect the science. Software engineering style of systematic testing that covers the software procedurally from beginning to end, is not seen as conducive to increasing the scientist's knowledge, and so is not used. We found that scientists may have developed a "well chosen" and targeted approach to testing, centered around their knowledge rather than prescribed by software engineering practices.

(v) Unplanned development

Unless forced to follow a prescribed software engineering development paradigm, scientists appear to develop their software in a random, unplanned manner. In a following section, we present an alternate model of how scientists develop their software, showing that there are well considered factors driving their choices of activities, and these choices are not random.

(vi) Formal languages for requirements

Scientists are well versed in mathematics and the formal expressions of their scientific domains. These expressions, and their theory, serve as their requirements languages.

(vii) Requirements and design are not separate activities

We have observed that the scientists do not disentangle requirements from design, nor from coding or use of the software. There is a continuum from scientific question to answer, channeled through the software. For the scientists, the phases of requirements, design, code, and test are intimately tied together and any attempt to separate them sets up artificial barriers to what the scientists are trying to accomplish. In our observations, the scientists are working successfully outside the method-based software development paradigms that dictate separated phases of requirements, design and implementation.

(viii) Regression testing

Scientists use regression testing where it makes sense in their environment. Reproducibility is important to the scientists, but can be achieved by saving previous executables (configuration management) or by adding new input options to bypass competing code. From the study of Cote (2005), we found that setting up an automated regression suite to achieve high code coverage was extremely problematic and highlighted why scientists' find other ways to achieve the same results.

(ix) Over confidence in correctness

The scientists do not separate the software from their science. If the output from the software is in doubt, the scientist may blame the theory as much as the software. However, the scientist pursues the questionable output, making changes in a "feed-back loop" style (Sanders and Kelly, 2008) until he/she is satisfied that everything is correct and the output can be trusted. We did not observe over confidence. Conversely, there was consistent concern with ensuring trust in the software.

(x) Lack of knowledge to pursue program failures

Industrial scientists we interviewed insisted that new hires need to know fundamentals about operating systems and compilers, and must be willing to learn everything about their new environment (Kelly, 2013). We've talked to a computational psychologist who had a deep understanding of his multiprocessor parallel computing system and a medical physicist who understood all the different data formats output by his equipment. One scientist commented that they need to know the "big picture" or risk not recognizing a problem and how to fix it.

On the surface, scientists may conform to the definition and characteristics of end user programmer: single or small groups of developers, small groups of users, focused on end use instead of the software, and lack of visible and systematic software engineering practices. However, this viewpoint is far too software product centric and misses the important differences in how scientists work. Because the idea of scientific software must be extended to include the scientist as part of the software system, with his/her knowledge as the deliverable, scientific software potentially serves a very large user group. We are discovering that scientists have developed viable alternatives to method-based views of software development, focusing instead on knowledge-building activities.

4. A model of knowledge acquisition as a driver for the development of scientific software

In order to fully understand how scientists view and develop their software, we need to change from a method-based view of software development where the product is the software, to a non-method view of software development where the product is the scientist's knowledge.

At least since 1990 (e.g., Guindon, 1990), researchers have considered how knowledge is acquired and expressed in any type of software. Earlier, researchers (e.g., Curtis et al., 1979) were aware that human understanding played a key role in successful maintenance of software. More recently, (e.g., Ralph, 2012) there has been work on the underlying theories of how the human agent designs a software system without following systematic methods. Ralph (2012) states that the software development literature is dominated by "methods" which are, by definition, prescriptive and "…include practices, techniques, tools and phase models", and that, "… phases are also explicitly adopted in the official IEEE Guide to the Software Engineering Body of Knowledge". We found methods dominate software quality standards as well (e.g., Canadian Standards Association).

Ralph (2012) proposes an alternate process theory of software design, based on realizing perceptions, refining understandings, and recording that understanding in the technological artifact (the software), in other words a design theory that does not prescribe methods and is based on acquiring knowledge. The model we propose can be viewed as an extension of the alternative that Ralph is proposing.

In our model, the act of *developing* the software contributes to the scientist's understanding. Developing software is not solely a means of producing an executable. For the scientist, recording the understanding after refinement ultimately takes place not only in the software code, but in scientific reports and other products based on the scientist's acquired understanding. This is a much broader view of "software development" that includes as an end product, the increased knowledge of the scientist, the answer to the scientific question, the reports and products associated with the answer to the scientific question, as well as the software executable.

To address this broader view of software development, in Section 4.1, we introduce a model with five broad knowledge domains that contribute in some way to the development of a piece of scientific software. The amount of knowledge required from each domain depends on the environment of the software and its intent, but each domain is present to some degree.

In Section 4.2, we illustrate the flow of knowledge with a hypothetical, but typical example of scientific software development. In Section 4.3, we discuss and illustrate the fragmented knowledge assumed by software development method-based paradigms described in textbooks, taught in academia, and assumed by software quality standards. This fragmentation impedes the flow of knowledge and impairs the coherent understanding of the whole, the understanding that scientists need to answer their scientific question.

4.1. Knowledge domains

In 1990, Guindon wrote that "high level software design is characterized by the integration of multiple domains of knowledge at various levels of abstraction." He points out that "there is little hope of understanding [the design] process without identifying the domains of knowledge designers bring to bear …" (Guindon, 1990). In 2004 (Kelly, 2004), we concretized Guindon's idea of knowledge domains in a study of drivers for long-term evolution of a sample of scientific software. For this study, we used five knowledge domains that contribute to the development of scientific software. To some extent these knowledge domains are involved in all software systems. Table 1 gives a brief description of each. A longer description can be found in Kelly (2011). Fig. 1 shows a graphical representation of our knowledge domain model. The lines in Fig. 1 connect the knowledge domains that scientists commonly consider the most closely associated.

In building and using a software system, knowledge from each domain is acquired, as needed. The knowledge from a particular domain that is needed for one software system may be trivial, yet for another may be extensive. Almost always, the knowledge needed for a software system proceeds from general knowledge (usually acquired through formal education) to specific knowledge. For example, a programmer may know how to code in C, but does not, as yet, know the conventions and particular implementations used, or will be used, in the system he/she is working on. Knowledge from one domain may suggest knowledge that must be acquired in another domain. For example, knowing that the hardware has parallel architecture and that the theoretical solution technique is computationally intensive may prompt the developer to acquire software knowledge on how to take advantage of parallel processing. The scientist may also need to re-examine the theory to understand how to segment the algorithm into discrete data packets and independent processes.

Table 1
Knowledge domains that contribute to the development of scientific software systems.

Knowledge domain	Description
Real World knowledge	Knowledge of real 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 software system.
Execution knowledge	Knowledge of the software and hardware tools used to create, maintain, control, and run the software system related to the problem being solved.
Operational knowledge	Knowledge related to the use of the computer software system in order to solve the Real World problem.

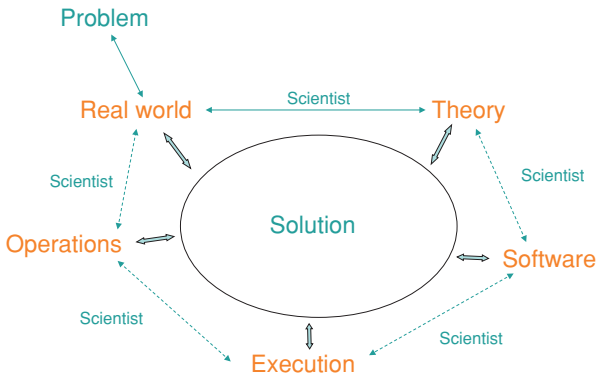

Fig. 2. Knowledge acquisition model used by scientists to develop software to solve real world problems. The arrows represent the use of one domain to increase knowledge in another. The solid line represents general (formal) knowledge existing for the person acquiring specific knowledge, and the dashed line represents specific knowledge being acquired where general (formal) knowledge may be missing in one or both the accompanying knowledge domains.

In the next section, we augment the knowledge domain model to illustrate the scientist's progress from scientific question to solution where software is part of the solution.

4.2. Knowledge acquisition view of the development of scientific software

Using observations from our sets of studies with scientists, we provide a view of the underlying processes that drive scientists' approach to software development, or more correctly, their approach to gaining knowledge to answer their scientific question.

We explain using a hypothetical, but typical, example of scientific software development as part of answering a scientific question. Fig. 2 provides a general view of scientific software development.

In our example, the problem to be solved involves an industrial plant where fluids are transported to different points around the plant via a complex piping system with valves, pumps, bleed points, injection points, and other pieces of equipment that contribute to changes in temperature, pressure, flow rate, and quantity of fluid in the system. The problem to be solved concerns the integrity of the piping system when equipment fails, such as pumps stopping, valves jamming, pipes leaking, and feed systems failing.

Scientists can move smoothly from the Real World Knowledge Domain to the Theoretical Knowledge Domain because of their training. They move from theory to software in order to both confirm the theory and construct a computational exploration of the problem. Knowledge and trust are acquired in the process. Often, there is an interaction with the other Knowledge Domains. For example, the scientists can choose a set of governing equations that describe the flow of fluid in a piping system. They decide on a discretization approach to move from the continuous equations to the discrete, algorithmic world of the computer. Next, a solution technique is chosen. Once this is coded in the software language, a set of data is used to generate values for the equations, and the software is run to produce data. Several problems could emerge at this point. The algorithm may not converge or the input data used may not be sufficiently accurate. Across the entire piping system, scientists may realize that the algorithm does not conserve mass or energy. The source of the problem may be the wrong set of governing equations for the particular problem being tackled, inappropriate assumptions or simplifications, inaccurate measurement data from the real world plant, inappropriate solution techniques for the equations being solved, truncation error from the simplifications, round-off error from the hardware, catastrophic cancellation because of particular values being combined in the code, and incompatible discretization techniques for the input

data. As discovered during the case study described by Cote (2005), even the optimization option chosen for the compiler can affect the output by several orders of magnitude. The interaction of input data, theory, discretization choices, solution technique, machine precision, and order of coded calculations require significant exploration and acquisition of knowledge from all five knowledge domains.

In this exploration, the scientist may decide that the pump model needs more detail to provide a more accurate pressure profile. This may require getting field data from the plant or manufacturer (Real world), developing an empirical model (Theory world), setting up new input data parameters (Operational world), and adding a new module in the software code (Software Knowledge). After the new module is running, the scientist may code certain pump parameters to be calculated and plotted in order to compare to plant data. Each activity increases knowledge in one or more knowledge domains. Nowhere is the software considered the end product. The software is a part of the integrated acquisition of knowledge from all knowledge domains. The end product is the scientist's increased knowledge moving him/her towards an answer to the scientific question.

Fig. 2 illustrates how the five knowledge domains (including software) are integrated and feed into the solution to the Real World Problem. There is no division between Problem domain and Solution domain as described by the method-based approaches. The software is not considered a separate product, but is a part of the Software Knowledge domain, constructed to help in the exploration. The scientists work by spending time in areas where their knowledge is lacking until sufficient understanding is achieved to move elsewhere. The scientist is an integral part of the system and makes the decisions on what is to be done next. What may appear to the outside observer as a random walk, the scientist's activities are all focused on increasing knowledge and building the scaffolding (the software code, the theory, the input data, conversations with others, etc.) to support this increase of knowledge.

Typical with scientific software, there are dependencies amongst all the elements that are a part of the development and use of the software. These dependencies cannot be predicted ahead and have to explored to understand their effects in specific sets of circumstances. It is the dependencies that require the greatest exploration and knowledge acquisition, all within specific contexts. Fig. 2 shows the scientists working in a manner that is essentially represented by a fully connected flow graph, motivated by the "solution", and where the scientist is free to move from any knowledge domain to any other at any time.

4.3. Using the knowledge domains to explore method-based software development

We acknowledge that software development outside the scientific domains ranges from something that looks similar to what we have observed for scientists to something that approaches a strict method approach of, say, the waterfall model (Royce, 1970).

The discussion in this section explains how a strict methods approach to software development assumes that the software system is handled in disconnected fragments. Our discussion then presents the impact that this assumption makes on the flow of knowledge. This is important in understanding why the imposition on scientists of, say, quality standards based on the waterfall model, is detrimental to the development of scientific software. In other words, the coherent understanding that the scientist is trying to achieve is contrary to the assumption that the system can be handled in fragmented pieces.

Traditionally, the interaction between a user and a professional software developer begins with requirements engineering. Whether the outcome of requirements engineering is recorded as a document, such as in document-driven development methodologies (Boehm and Turner, 2005), or requirements engineering is carried out as a continuing dialogue, as in agile methodologies (Boehm and Turner, 2005), the assumption is that the user is in possession of the Real World knowledge of his/her environment, and the developer is not. The reason for the large amount of research in the area of requirements engineering is succinctly stated by Jackson (2001), "… it's widely recognized that many systems have failed – often very expensively or even disastrously – because their requirements were not properly determined. The hardware and software functioned correctly, but the function they performed was not what was needed. The developers' failure was in capturing and understanding the problem, not in devising or implementing the solution".

The dialogue between the user and the developer requires the user to abstract away the specific problems in his/her Real World, in other words, abstract away the specific problems being addressed, and describe his/her needs in terms of an idealized solution. The user moves away from the Real World to a Theoretical world, describing what he/she thinks will afford a solution. Often the user is not well versed in doing this type of abstraction.

Through the emphasis on requirements engineering, the software developer is assumed not to be knowledgeable about either the Real World or the Theoretical world of the user. Through dialogues with the user, the developer acquires some knowledge of the user's Theoretical world, but the developer interprets that knowledge by what is relevant to the software product.

The developer creates the software product; the user does not participate in the coding or understanding of the software or execution domains of the product. However, the user may participate in acceptance testing of the product. In absence of the user, the developer may test the product against the requirements document.

Delivery of the software product most often does not include the source code. Little or no knowledge from the Software Domain is passed to the user. Flow of knowledge from the developer to the user is typically confined to the knowledge domain we label as "Operational Knowledge". The developer can provide a "user's guide" on how to operate the software. The knowledge the developer provides for operational knowledge is "general" knowledge. This general knowledge is based on the Theoretical Knowledge from requirements engineering and Software and Execution Knowledge possessed by the developer. The user must acquire specific Operational Knowledge on how to use the software to solve his/her Real World problems, based on the "user's guide" and without knowledge of the Software or Execution domains. The developer has no mandate to solve the user's Real World problems. Indeed, the developer does not have the ability to do this. The developer is not expected to acquire knowledge of the user's Real World beyond what is garnered during requirements engineering.

Fig. 3 illustrates the sharing of domain knowledge by user and software developer. The lines in the figure represent knowledge being acquired. The arrows indicate where knowledge from one domain is used to increase knowledge in another. The labels on the lines are the agents (either developer or user) acquiring knowledge. The dashed lines represent specific knowledge that is being acquired, but the agent lacks general knowledge in one or other of the connecting knowledge domains.

The illustration shows the clear division between user knowledge and developer knowledge. Software development methods expect this divide to exist and label them as the "problem domain" and the "solution domain". The link between the two domains is the "software product", considered to be the Solution to the user's Problem. In fact, with the software product in hand, the user still has to acquire the specific knowledge in order to formulate the Solution to his/her Real World problem.

Our model showing flow of knowledge amongst five knowledge domains illustrates the fragmentation of knowledge, not only amongst the human participants in the exercise, but also in the flow paths. This is the view assumed by the dominant methods literature.

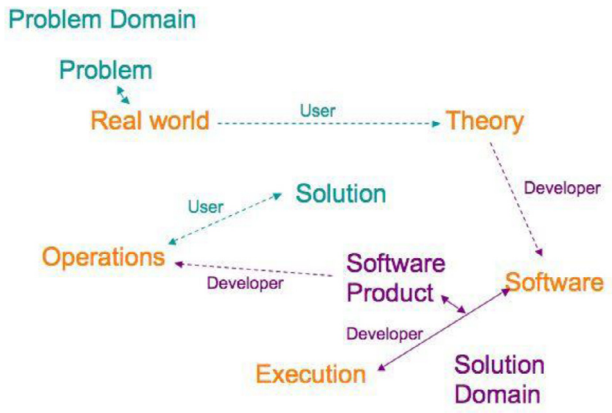

Fig. 3. Model of knowledge flow assumed by method-based software development paradigms. The arrows indicate acquiring knowledge in one domain based on knowledge in another. The labels indicate who is acquiring the knowledge. The solid lines indicate existing general (formal) knowledge in both connecting domains. The dashed lines indicate missing general (formal) knowledge in one or both of the connected knowledge domains.

This is a view that does not work for scientists developing scientific software, for the following reasons:

(i) The number of flow paths through this model are substantially reduced from the fully connected "flow graph" we have observed the scientists following. This restricts freedom of exploration necessary to understand dependencies in their work.
(ii) The software product is designed to be static and is assumed to be the solution to the Real World problem when in reality it is not, even with non-scientific software.
(iii) If we assume that the scientist's role is strictly user, then the scientist is denied knowledge about a large part of the "solution" to his/her scientific problem.

5. How scientists develop software outside the "methods" approach

Our observations (e.g., Kelly, 2013; Sanders, 2008; Sanders and Kelly, 2008), and that of others (e.g., Sletholt et al., 2012), are that scientists engage in software development outside the methods paradigm. But, because methods are so dominant in the software engineering literature (Ralph, 2012) and assumed by many to be the only valid approach to software development, scientists have been criticized for not following methods (e.g., Merali, 2010) and have been offered a plethora of advice on how to apply methods (e.g., Baxter et al., 2006; Crabtree et al., 2009).

As noted by other research, scientists do not see themselves as software developers (Sanders, 2008) but as a professional who uses software as part of their exploration for a solution to a scientific problem. The model they use is a continuum of activity where the roles of software developer, tester, requirements engineer, are not separated. Sletholt et al. (2012) commented, "normally the scientists don't assume any specific roles" and that "none of [their] interviewees were able to identify transitions between the activities, as most of these activities (such as coding, analysis, design, and testing) are carried out more or less simultaneously." Sletholt et al. (2012) found that the two non-commercial projects they studied " didn't use any of the agile practices related to requirements … people involved in these projects didn't perceive any particular problems with requirements, even though they didn't use the agile practices." The authors suggest that, "This might be explained by the fact that the development is based on personal motivation …". In our work, we see personal motivation being driven by acquiring knowledge for the problem at hand. From our observations, scientists engage in a continuous flow of discovery and knowledge acquisition, leveraging that knowledge in order to further explore. As part of this continuous flow, the software is developed and moved towards contributing in a meaningful way to the solution of the scientist's real world problem.

Instead of studying how scientists should conform to the methods paradigm, we chose to study scientists whose application domains demand extreme care in their work, and to observe how they develop software. The scientists' approach to supporting knowledge acquisition and software development can be described within three areas: software implementation, testing, and social communication.

5.1. Software implementation

Every scientist we interviewed emphasized two qualities for their software code: simplicity and readability. Readability is defined by the scientists as 'making the science in the code obvious'. These two qualities underlie all the implementation efforts of the scientists, and come from their desire to communicate. Software code written in a high-level language has two purposes: sending instructions to the hardware and communicating to the human reader. Obviously, the first has to work properly, but in the interviews, the scientists emphasized the second.

Simplicity is part of readability. According to research (Floyd and Bosselmann, 2013), simplicity is mentioned more in literature related to scientific software than in literature related to software engineering. One interviewee called the approach "simple rational programming".

The scientists' practices in making the code readable reflected many software engineering tenets: well named variables, in-code comments, and cohesive modules. However, the scientists put their own stamp on each of these.

Variable names preferably are the same as standard usage in the application theory, for example *P* for pressure, *Rho* for density, and *H* for enthalpy. The goal is to develop a vocabulary that is consistent throughout the theory documentation, the software code, and the user manual. Naming conventions are agreed on "by everyone". "You don't want to obscure the algorithms." This reflects the scientist wanting to support knowledge acquisition throughout the development process, as shown in Fig. 2.

In-code comments are to explain not only *what* the code is doing, but *why* the code is doing it like that. One comment was that the "documentation in the code has to be very clean".

Similar emphasis on strong links between the scientific theory and the code are reflected in comments about modularization. One scientist commented, "modularization should reflect the physical entities you're modeling". Naming the modules should, again, reflect the theory. The code within the modules should be consistent in style and readable. One scientist commented on self-documentation: "The best way to describe what the program does is in the programming language. Definitions in natural language are not as clear as definitions written in a programming language." The preference for the scientists is to use a programming language that looks like the mathematics of their theory, allowing as simple transition between the two as possible.

The overriding approach for the scientists is "write it [the code] on the assumption twenty years later you'll have to change it". This reflects their long-term commitment to their work and their desire to communicate clearly.

5.2. Testing

Testing for the scientists includes both static and dynamic analysis of the code.

Static analysis includes various approaches to reading the code. Code reading is an integral part of their software development rather than an after-the-fact assessment. The scientists also do not use any

of the software engineering terminology as defined by the software engineering literature. One scientist was specifically asked if he did "code reading". He answered, no, yet went on to describe reviewing his own code, and printing it off to sit with another person and "chat" over the code.

Another scientist described a process of implementing the code, having someone else read it over, and a "fair amount of discussion on how to get things done". A medical physicist described, on obtaining code from another medical center, sitting down with a colleague and reading every line of the new code. He emphasized they always did this.

The standard use of regression testing is to ensure new changes to the code base have not altered previous values. According to software engineering literature, regression test suites are fully automated, running a large number of short tests. Scientists typically have the problem that each test could run five to six hours and data sets are "a couple of terabytes", and that "IT balks at the amount of storage required". If regression testing is done, it is carried out on a select set of key cases, and only strategically important output is checked, usually by hand due to the differences made by round-off error.

Dynamic analysis, as with static analysis, is integrated into the implementation of the code. Often, the two are combined, as evident in our work (Kelly et al., 2011) where an astronomer used, for selected exemplar cases, a debugger to trace the execution and allow him to read the associated code. His main goal of testing was to understand the entire code. One scientist emphasized that writing and running the code is part of understanding the problem being solved. He stated, "I can't just stare at notes for a month … the code reveals the problems to me, [that you] layer the code on and test as you increase your understanding". Another scientist tries "small things for testing, things easily verified as correct, then add one thing at a time". The scientists stressed needing "intimate knowledge of the hardware" since different hardware platforms can give different answers, and needing "operating system knowledge to parse error messages and know how to handle them". Validation testing for the scientists is checking the test results against Real World phenomena, not against requirements specifications. For any type of testing, if there is an anomaly in the test results, all the scientists agreed that it is mandatory to understand why. To move the code towards solving the Real World problem, knowledge from the different domains has to be gained and integrated together.

5.3. Social communication

The strongest ethos of both teamwork and communication amongst the scientists we interviewed, existed with the industrial scientists whom we term risk-averse scientific application developers. The industrial scientists insisted that it was more important to "get it right" than "to protect turf", that there were to be "no heroes and no renegades". A number of observations were made on what these scientists do to support transfer of knowledge amongst colleagues when working on a project.

With the scientist, there is no boundary between the roles of theorist and developer. The boundary between the roles of user and developer is often blurred. Instead, the scientists we interviewed are part of multidisciplinary teams, with the disciplines often overlapping and based on knowledge domains rather than method roles.

All the interviewees emphasized communication. One interviewee explained, "There's a lot of hallway conversations and a lot of collaborative work. We know what each other are doing. Sharing in the community is pretty good."

Particularly interesting were expectations for new hires. They expected the new staff to interact with experienced staff, know their limits, ask questions, discuss, and collaborate. New hires were expected to broaden their knowledge: staying within the boundaries of his/her discipline does not work. All scientists talked of mentoring, hands-on learning, and long-term commitment – commitment of at least five years.

Communication worked best when management assigned a project and the scientists self-organized into a team with fluid boundaries and no fixed roles. The fluid boundaries allowed other knowledge sources (people) to be drawn into the team as needed, and roles were filled as necessary for the length of time needed. One person typically took on several roles in achieving a specific goal. The management paradigm we observed was not the task-based, segmented approach that is typical of much software development, but was a human-centric paradigm typical of knowledge-based organizational models in business management (e.g., Kochan et al., 2002).

6. Summary and conclusions

None of the current software engineering views on how scientists do – or should – develop software are universally applicable to the wide and varied range of what is termed, scientific software.

The most ubiquitous view is that scientists are end user programmers. This is based on observations that scientists do not self-identify as professional programmers, that they do not produce software as end products, that their user base is small, and that scientists are not using "systematic and disciplined activities that address software quality issues" (Ko et al., 2011).

We argue that this characterization does not fit the scientists we studied. The scientist must be considered as part of the software system, since the knowledge the scientist acquires is a key output of the system. With the scientist's acquired knowledge as an output of the system, that output (his/her knowledge) is used to make decisions about products that could affect the public at large. In effect, the user base of that output is enormous. The software development approaches used by the scientists we studied do not conform to method approaches, yet they are successful in their risk-averse application domains, and should be considered as systematic, but systematic based on knowledge acquisition, not on a list of tasks.

We present an alternative model of software development based on the idea of knowledge acquisition. Instead of following a sequence of tasks related to a software engineering method paradigm, the scientist interacts with software artifacts in order to fill gaps in his/her knowledge. The order and nature of these interactions are determined by the needs of acquiring knowledge by the individual or closely-knit team of scientists. To the outside observer, the activities may appear random and uncoordinated. In reality, they are driven by the need to interact with trustworthy sources of knowledge – none of these sources, including the software, must ever lie to the scientists. The scientist carries out a series of activities involving the software in order to acquire knowledge and to ensure the software – and other sources of knowledge – can be trusted.

From our studies, we make the following observations:

(i) Scientific teams who develop software do not identify members of the team according to software tasks, such as designer, tester, or requirements engineer. Instead, members of scientific teams are identified by their science or engineering specialty such as heat transfer specialist or two phase fluid specialist. Each are expected to explore their specific scientific problem from "cradle to grave", including development, testing, assessing trustworthiness, and use of software modules specific to their scientific goals.

(ii) Clearly delineated chunks in developing software such as iterations, phases, tasks, sprints, etc. are not evident as scientists develop software. Our model of knowledge acquisition suggests an uninterrupted flow of development with the scientist at the center, and ending when the scientific question is answered. Development can encompass the theory, the data, the

code, the hardware in the environment, and the scientist's understanding of the whole.

(iii) Testing is integrated into development in such a way that the trustworthiness of the developed software is ensured along the way. Scientists use a variety of assessment techniques, often designed specifically for the problem at-hand, including code reading, verification against theory, validation against real-world data, checks for self-consistency, and algorithm checking. Our research shows that mistakes in computational type code can reveal themselves with a small number of well-designed tests. We do not observe scientists using end-to-end "systematic" testing as described by software engineering literature. Instead, scientists appear to be successful in their testing using approaches that are highly integrated with their objective, and may constitute the "small number of well-designed tests". This alternative approach to "systematic" testing deserves more in-depth study.

(iv) One of the scientists we interviewed saw the work of her group as fitting "amethodical" development as described by Truex et al. (2000). Our model of knowledge acquisition shows how knowledge is acquired from five knowledge domains as the scientist develops software and moves towards answering their scientific question. The benefit from this view is to rethink how to support scientists in their endeavors, using knowledge acquisition as the driver rather than imposing methods.

(v) We observed a number of activities that our risk-averse scientists use to assure trust in their software. These activities are carried out without invoking methods. These activities potentially form a basis for future research into successful software development outside method-based paradigms.

There are three main impacts from our observations.

One, the knowledge acquisition view of software development offers an alternative to the ubiquitous method-based view. It opens a body of research on how to integrate testing with development and with knowledge acquisition, and how to integrate knowledge acquisition into other software development activities without prescribing and controlling how and when the developer does his/her work. The hope is that there are creative scaffoldings that will support acquisition of knowledge as an integral part of software development.

The second impact is related to safety-related scientific development in industries that come under the auspices of software quality oversight. Software development in these industries is currently being assessed using methods-based software quality standards (e.g., Canadian Standards Association). Our model suggests that imposition of such standards may be detrimental to software quality, rather than improve it. Works needs to be done to explore how oversight can be satisfied within a knowledge acquisition development paradigm.

Third, the knowledge acquisition view of software development requires a different management paradigm. In this different paradigm, methods, prescribed tasks, and designated software roles are replaced with collaborating, knowledge-sharing teams, goal-based instead of task-based organization, and recognition that the scientists and their knowledge are a more important asset than the "software product".

Scientific software development within the risk-averse scientific application domains presents a very different "ecosystem" of software, developer, user, theory, and real-world problem. The outcomes of the interdependencies amongst all the elements of this ecosystem cannot be predicted before the software is constructed, and the software itself becomes an interacting part of the ecosystem along with hardware, theory, users, data, and knowledge. No one piece of this ecosystem can be isolated and considered effectively on its own. Our knowledge acquisition model is offered as a view of this type of system and provides a basis for further research and understanding on how best to provide software-based answers to scientific questions.

Acknowledgment

This work is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC). Many thanks go to the scientists and engineers who offered their time and enthusiasm for these studies. Several talks given by the author based on these interviews were funded by IEEE.

References

Ackroyd, K.S., Kinder, S.H., Mant, G.R., Miller, M.C., Ramsdale, C.A., Stephenson, P.C., 2008. Scientific software development at a research facility. IEEE Softw. 44–51.

Baxter, S.M., Day, S.W., Fetrow, J.S., Reisinger, S.J., 2006. Scientific software development is not an oxymoron. PLoS Comput. Biol. 2 (9), e87. doi:10.1371/journal.pcbi.0020087.

Boehm, B., Turner, R., 2005. Balancing Agility and Discipline. Addison Wesley.

Boisvert, R.F., Tang, P.T.P. (Eds.), 2000. The Architecture of Scientific Software. Kluwer Academic Publishers, Boston, USA.

Canadian Standards Association (1999) CSA N286.7-99 (R2012) – Quality Assurance of Analytical, Scientific and Design Computer Programs for Nuclear Power Plants.

Cote, N., 2005. An Exploration of a Testing Strategy to Support Refactoring Master's thesis. Royal Military College of Canada, Kingston, Canada.

Crabtree, C.A., Koru, A.G., Seaman, C., Erdogmus, H., 2009. An empirical characterization of scientific software development projects according to the Boehm and Turner model: a progress report. In: Proceedings of ICSE Workshop on Software Engineering for Computational Science and Engineering. SECSE'09, pp. 22–27.

Curtis, W., Sheppard, S.B., Milliman, P., Borst, M.A., Love, T., 1979. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. IEEE Trans. Softw. Eng. SE-5 (2), 6–104.

Floyd, B.D., Bosselmann, S., 2013. Simplicity research in information and communication technology. IEEE Comput. 26–32.

Gray, R.C., 2010. Investigating Test Selection Techniques for Scientific Software Master's thesis. Royal Military College of Canada, Kingston, ON, Canada.

Guindon, R., 1990. Knowledge exploited by experts during software system design. Int. J. Man-Mach. Stud. 33, 279–304.

Hook, D., 2009. Using Code Mutation to Study Code Faults in Scientific Software Master's thesis. Queen's University, Kingston, ON, Canada.

Hook, D., Kelly, D., 2009. Mutation sensitivity testing. IEEE Comput. Sci. Eng. 40–47.

Jackson, M., 2001. Problem Frames. Addison Wesley.

Jorgensen, P.C., 1995. Software Testing A Craftsman's Approach. CRC Press, Boca Ratan, Fl. USA.

Kelly, D., Salomon, G., 2015. Survey of medical physicists and their software skills. J. Appl. Clin. Med. Phys. 16 (1) in press.

Kelly, D., Skordaki, E.M., 2014. A medical computing lab and a model of learning. In: Proceedings of IEEE Canadian Conference in Electrical and Computer Engineering.

Kelly, D., 2013. Industrial scientific software – a set of interviews on software development. In: Proceedings IBM Centre for Advanced Studies Conference, CASCON. Toronto, Canada.

Kelly, D., 2011. Innovative approaches for developing scientific software. J. Organ. End-User Comput. 23 (4), 63–78 Special issue on Scientific End-User Computing.

Kelly, D., Thorsteinson, S., Hook, D., 2011. scientific software testing: analysis in four dimensions. IEEE Softw. 84–90.

Kelly, D., Gray, R., Shao, Y., 2011. Examining random and designed tests to detect code mistakes in scientific software. J. Comput. Sci. 2 (1), 47–56.

Kelly, D., 2009. Determining factors that affect long-term evolution in scientific application software. J. Syst. Softw. 82, 851–861.

Kelly, D., 2008. Innovative standards for innovative software. IEEE Comput. 88–89.

Kelly, D., 2008. A study of design characteristics in evolving software using stability as a criterion. IEEE Trans. Softw. Eng. 32 (5), 315–329.

Kelly, D., 2004. An Exploration of Evolutionary Change in an Example of Scientific Software Ph.D. thesis. Royal Military College of Canada.

Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrence, J., Lieberman, H., Myers, B., Rosson, M.B., Rothermel, G., Shaw, M., Wiedenbeck, S., 2011. The state of the art in end-user software engineering. ACM Comput. Surv. 43 (3) Article No. 21.

Kochan, T., Orlikowski, W., Cutcher-Gershenfeld, J.; "Beyond McGregor's Theory Y: Human Capital and Knowledge-Based Work in the 21st Century", prepared for the Sloan School 50th Anniversary Session, 11 October, 2002; available at: http://sloanweb.mit.edu/50th/pdf/beyondtheorypaper.pdf.

Lehman, M.M., Perry, D.E., Ramil, J.F., 1998. Implications of evolution metrics on software maintenance. In: Proceedings of IEEE International Conference on Software Maintenance, pp. 208–217.

Merali, Z., 2010. Scientific software: error … why scientific programming does not compute. Nature 467, 775–777.

Ralph, P., 2012. Sensemaking-coevolution-implementation theory: a model of the software engineering process in practice. In: Proceedings of Semat Workshop on a General Theory of Software Engineering Stockholm, Sweden p. 52.

Royce, W., 1970. Managing the development of large software systems. In: Proceedings of IEEE WESCON, pp. 328–338.

Sanders, R., 2008. The Development and Use of Scientific Software Master's thesis. Queen's University, Kingston, Canada.

Sanders, R., Kelly, D., 2008. Dealing with risk in scientific software development. IEEE Softw. 21–28.

Segal, J., 2005. When software engineers met research scientists: a case study. Empir. Softw. Eng. 10, 517–536.
Segal, J., 2004. Professional End User Developers and Software Development Knowledge. Open University Technical Report No.: 2004/25.
Sletholt, M.T., Hannay, J.E., Pfahl, D., Langtangen, H.P., 2012. what do we know about scientific software development's agile practices? IEEE Comput. Sci. Eng. 2–14.
Truex, D.P., Baskerville, R., Travis, J., 2000. Amethodical systems development: the deferred meaning of systems development methods. Account. Manag. Inf. Technol. 10, 53–79.

Diane Kelly is an Associate Professor in the Department of Mathematics and Computer Science at the Royal Military College (RMC) of Canada. She is cross-appointed to RMC's Department of Electrical and Computer Engineering and to the School of Computing at Queen's University. Diane has a Ph.D. and MEng in Software Engineering both from RMC. Her B.Sc. in Pure Mathematics and B.Ed. in Mathematics and Computer Science are both from the University of Toronto. Diane worked in industry for over 20 years as a scientific software developer, technical trainer, and QA advisor. She is a senior member of IEEE.