Industrial Scientific Software – a Set of Interviews on Software Development

Diane Kelly
Department of Mathematics and Computer Science
Royal Military College of Canada
Kingston, ON Canada
kelly-d@rmc.ca

Abstract

An ethnographic study is used to explore activities carried out by industrial scientists to successfully develop their software. The extremely rich data set that resulted helps paint a picture of their development context. Apparent are the mismatches between software development methods commonly described by the software engineering community and the practices successfully used by the industrial scientists. Instead of following any type of prescribed method, the scientists follow what has been described as an *amethodical* approach to software development. Acceptance of the validity of this approach could provide an important alternative to how we currently view software development.

1 Introduction

Scientists have been writing software for a very long time (eg., [8]). About eight years ago, software engineers started looking at what scientists were doing and publishing what they were doing wrong (eg. [24]). Generally, scientists seemed to be ignorant of good software engineering practices.

In earlier work [18], we interviewed academic scientists who wrote software for their research. Several were well aware of practices recommended by the software engineering community. Others were soliciting help for their problems and it was not immediately obvious what could be done for them. It also wasn't clear if there were differences between working practices of academic scientists and those of industrial scientists.

To explore these questions, a series of interviews of Canadian industrial scientists was carried out. Instead of purposely looking for "what they do wrong", the interviews sought to find out what practices worked for them. To eliminate pre-conceived biases, the exact interview questions were not determined ahead. The interviewer only had a list of broad areas of discussion. The scientists were interviewed in their own environments and were encouraged to tell stories and take control of the interview and its direction. They were only given gentle nudges if they were wandering too far in their narrative. The result was an extremely rich set of ethnographic data.

From this data, a general picture emerged of the software development environment of these scientists. Immediately apparent is the bad fit of standard software engineering development methods to how scientists work and how they should develop software.

Section II offers a set of factors intended to define and differentiate different groups of scientific software. Section III describes in more detail the interviews of the industrial scientists. Sections IV, V and VI discuss findings from the interviews in terms of existing software development and acquisition models. Section VII lists activities described by the scientists in the interviews that were key to successful development of their software. Section VIII concludes.

2 Defining and Differentiating Scientific Environments

In describing research related to scientific software and software engineering, difficulties arise in describing ideas and terminology across cultural divides. The most basic difficulty here is the definition of scientific software. The variation in size, use, applications, and personnel involved is enormous. We offer a definition of the specific type of scientific software involved in our research. We also offer a set of "factors" that define a continuum on which to place most scientific software, and we place software that we deal with in our research on that continuum. The aim is to precisely define the environment of the developers involved in the interviews described in this paper and to make more evident the reasons for the development choices they make.

The definition we use for the term scientific software is *application software that provides data to directly support scientific decisions*. Most frequently, this is modeling software that allows the scientist to examine a situation computationally, particularly in cases where experiments and real-world observation are inappropriate.

Our suggested factors to define a continuum to place types of scientific software are the following:

- the level of *risk tolerance* in the application domain
- *expected lifetime of the software*
- *expected duration of commitment* of the developer
- *distance of the developer from specific scientific questions*
- *breadth of expected knowledge*

Risk tolerance is an important differentiator for scientific software. Conversations with other researchers have described groups such as financial mathematicians, where the application domain implies risk-taking and the developers treat their software accordingly, doing very little testing to check for trustworthiness. Instead, the scientists involved in the interviews described in this paper, are highly risk averse. Their application domains include such as medical radiation treatments, operating nuclear power stations, and identifying structural faults in mine shafts. All the scientists in our interviews reiterated that the software must not lie to them. The data they glean from the software to support their scientific or engineering decisions must be correct. Therefore, there can be no compromise on quality of the software, where quality is trustworthiness. An additional observation can be made regarding the effects of risk aversion. In the academic environment, the social process of accepting the correctness of a result happens at the time of publication or presentation of a graduate thesis. In a highly risk averse scientific or engineering environment, the social process happens well before "publication" of a report. This leads to scientists having conversations and information exchange about their software and its output continually during development.

Expected lifetime of the software described in the interviews is decades. This is often a feature of scientific software [5]. The implications are that long-term maintainability is important. Given that most of the software described in the interviews was already decades old, and was accompanied by well established and successful development practices, new technologies and methodologies that imply sweeping changes and that would likely be superceded by other technologies within a short timeframe, are not considered useful.

Unlike the volatile employment environment currently found in the IT industry (eg., see [6]), scientific development groups interviewed expected new-hires to commit to at least five years. Developers interviewed talked of long-term commitments where they expect to be responsible for their software ten years down the road. This impacts how a developer thinks about their work and what sort of mechanisms they put in place to ensure comprehension of their software at some undetermined point in the future. This is in direct opposition to environments where developers are on short-term contracts or academic environments where graduate students end their commitment to the software they write after their theses are finished.

Distance of the developer from a specific scientific question distinguishes software, on one extreme, that is written with no interaction with a specific user, that addresses a general class of well-understood questions, and that is written for the purposes of commercial enterprise. On the opposite spectrum is software written to address a novel question and is used solely by the developer and/or colleagues. The software described in the interviews was close to this latter category, but with important differences. The software is written by the application domain scientist or groups of scientists, and user groups are composed of scientists in similar disciplines, varying in size from three or four individuals to hundreds. In all cases, however, the developers and the users are closely linked, often physically co-located, and the developers take an active role in answering specific novel scientific questions. In all cases, the software is seen as a means to an end, never the end in itself.

The expected breadth of knowledge for the developer groups interviewed covers five knowledge domains (see Table 1), ie., operational knowledge of the software (how to use the software), real world knowledge (the physical world that the software is a part of or simulates), theoretical knowledge (generally, the mathematical models embedded in the software), software knowledge (how to successfully express the computational model in a software language), and execution knowledge (the hardware environment the software is executed on). Many of the interviewees insisted that this breadth of knowledge is essential for successfully developing and making changes to the code.

3 Description of Interviews

In 2008, Sanders and Kelly carried out a series of interviews of academic scientists working at two educational institutions [18]. Their work supported the findings of others (eg., [20]) that there is a mismatch in the thinking of scientists who develop software for their scientific pursuits and the recommendations from the software engineering community. In 2012, Kelly extended this research to Canadian industrial scientists and engineers who develop software as a major part of their industry work.

Industry scientists were interviewed from four different disciplines: non-medical nuclear research (5), clinical medical physics (1), seismic analysis (2), and flow analysis (5). The interviews were open ended and encouraged story-telling. Aside from gathering identification type of information (such as size of software, size of user group, age of software, etc.) the interview used questions and prompts that encouraged the interviewees to elaborate on their experiences. General discussion areas covered at least the following topics:

- What typical user requests do you receive? What was most challenging? What helped you to resolve it?
- What was the most challenging coding issue you've faced? What helped resolve it?
- What do you like most/least about the design of the software you work on?
- What are the best resources for obtaining information about your software?
- How do you convince yourself or someone else that your answers are trustworthy? What do you do by habit to convince yourself the code is giving trustworthy answers?
- What parts of the software are volatile? stable?
- What one activity would you recommend to someone new to be convinced their code is right?
- What do you do about user training to address risk factors in the use of the software?
- Describe the management issues, eg. regulator, internal management, handling users, trade-off between quality and time, that has the most impact on your work?

Most of the interviewees were long-term employees (over ten years) and most held either Masters or PhD degrees in engineering or physical science disciplines. All were eager to tell their stories and brought to the table additional issues that were relevant and insightful.

Over fourteen hours of interviews were recorded digitally and the interviews were transcribed into nearly fifty type-written pages of major points. From this, a set of themes was identified and transcribed points were associated with the themes.

The following themes were identified:

- characteristics of users
- impact of outside regulators
-differences between academic settings and industrial settings
- software requirements
- focus on science when developing the software
- software tools
- code reviews and discussions
- mismatches between software engineering and scientists who write software
- code design issues
- software development management issues
- advice to students
- testing scientific software

This paper focuses on one aspect of "mismatches between software engineering and scientists who write software", that of software development methodologies and their application to the work of scientists writing scientific application software.

4 Ubiquitous Software Development Methods and Management Wisdom

Software development methodologies, as described in the software engineering literature, are roughly split into two camps: document driven and agile. Royce [17] first described a heavily document-driven software development method in 1970. By example, Royce said that a five million dollar project (in 1970) should have a fifteen hundred page design document. Royce's methodology is often called the waterfall method (eg. [23]) because of its cascade of process phases, one dependent on the previous being fully completed. It has been long known that the waterfall method, at its purest, is impractical because no one has the omniscient view to be able to decide a full set of requirements or complete design for a software system before any of it is ever built. Despite this, software development standards such as ISO 9000-3 essentially follow the waterfall method. It is easy to find consulting companies advertising on the web exhorting the values of the waterfall method, that by following it, projects will be completed on schedule and on budget.

In answer to the document driven method, agile methodologies have proliferated [2]. Examples of these methods replace communication via documentation with face-to-face communication facilitated by such practices as on-site customers, a simple shared story instead of requirements, stand-up meetings, pair programming, and constant refactoring where anyone can change any piece of code anywhere. Boehm and Turner [4] suggest that software projects can use a mix of document driven and agile methods.

Beck, in his book on Extreme Programming [3], applies a common extension to the Iron Triangle of Project Management. To the constraints of scope, cost, and time he adds quality as a fourth constraint. In project management, it is understood that a change in one of the constraints impacts the others and a fixed constraint is achieved by managing the other constraints. If a project is conceived where all the constraints are considered fixed, then the project is unmanageable.

Aside from the requirement of omniscience, the waterfall methodology fails the Iron Triangle test. If the project is expected to be on budget (cost) and on schedule (time), that leaves only scope and quality to manage. However, scope is set by the extensive requirements documentation that is expected to be fulfilled, and quality is determined by the proscribed activities of verification and validation. By management wisdom, all projects following the waterfall methodology are doomed to fail.

The Agile methodologies such as Scrum and Extreme Programming aim to address this failure by fixing the cost and schedule of the project and managing scope. The project's functionality is divided into small pieces and implemented in short time periods. At the end of each time period, a working piece of software is delivered. If there is sufficient time and funding, another piece of functionality is tackled. Quality is also a variable, since it is determined by the customer, assuming customer satisfaction will change as time and money runs out.

Since the scientists interviewed were close to the specific scientific question being answered, their use of the software is always to answer novel questions. The approach to a new question is investigative and not completely predetermined. Because of this, a waterfall approach is a poor fit. Paths will be followed, changes will be made based on current observations, and new paths chosen. One scientist commented that he could spend a month staring at specifications or equations on pieces of paper and they won't tell him what he needs to know – he has to write the code and observe.

Several case studies describe applying Agile methodologies to the development of scientific software (eg. [1, 15]). However, considering the Iron Triangle for managing a project to develop a piece of scientific software, methodologies such as Extreme Programming and Scrum are also a poor fit for the scientists interviewed. Both these Agile methods make two assumptions that are not true for the scientific software described in the interviews. First, that the software is the deliverable, and second, that cost and schedule are more important than quality and scope. If the scientific question cannot be answered to the required precision, then the software, no matter how quickly or cost effectively it is developed, is of no use.

5 Other Development Models and Software Acquisition Modes

5.1 Open Source Development Model

The open source development model was examined by Raymond [16] to explain why it seemed to be so successful. One of its advantages is to allow "lots of eyes" on the code, which results in mistakes being spotted. The other advantage is to allow users to scrutinize and understand the software and make changes applicable to their specific situation.

Our series of interviews of academic scientists in 2008 revealed a pastiche of opinions about the use of open source in their scientific endeavours. One scientist opined that you get what you pay for and never used open source application software in his particular field of engineering. Others belonged to a large user community using specific open source software for their scientific modeling. The open source software allowed them to make whatever changes were needed for the scientific question being investigated and the large community afforded discussion groups, online help, data, and sample output.

The industrial scientists interviewed had two main reasons for not using open source software or offering up their systems as open source.

One is the issue of intellectual property. Before 1980, the author's experience was that scientific software was shared widely in the associated industries [14]. This afforded the same advantages as open source since a large number of interested and capable people were sifting through the software and quickly flagged anything that looked wrong. As the 1990s arrived, the limited open source model in industry mostly went away. The term proprietary was replaced by the term intellectual property, where some commercial enterprises viewed the source code as a trade secret and a source of revenue. The software described by the industrial scientists interviewed in 2012 is considered part of intellectual property and open source is not considered.

The second reason is related to professional concerns, particularly in risk-averse industries. The software used to provide data for decisions has to be controlled in a number of ways. Both its contents and use have to be deemed trustworthy. This is a long and pain-staking process and the scientists are opposed to a situation where a "free-wheeling" user or developer can readily make changes outside that control.

5.2 Commercial Software

Commercial suppliers of application scientific software have a very different goal from the scientists using the software. For the commercial suppliers, the software is the deliverable and their goal is financial. The functionality of the software has been chosen based on known and common tasks. This is to maximize their potential user base. This puts the commercial suppliers at the opposite end of the *distance from specific scientific question* spectrum from the scientists in both our 2008 and 2012 interviews. The academic scientists interviewed in 2008 pointed out serious mismatches between their needs and the management of commercial software.

First, scientists pursue new lines of thought and soon find the commercial software lacks the functionality and models needed. Turn-around time for major changes to commercial software was as much as two years, leaving graduate students without a project.

Second, commercial software did not guarantee backwards compatibility with older versions as new releases were issued. This left scientists with older versions, data, and results that could not be replicated with the new versions. This is out of step with the long lifetime expected for scientific software described in the interviews. Commercial software suppliers work in a much shorter time frame.

Third, the reputation of academic scientists is based on the trustworthiness of their research, and hence the trustworthiness of the software they use. Transparency is needed to track intermediate calculated values, solution techniques and models used in the software, as well as details of the code itself. The commercial supplier worried about intellectual property may not reveal details beyond what's published in research papers. One of the industrial scientists commented that in some cases, implementing something as it is described in publications will "blow up on you": you have to "inject a pile of engineering" to get the code to work. In other words, this "pile of engineering" does not appear in any publications.

The industrial scientists interviewed described their concerns with commercial software. In some cases, their managers had decided to replace home-grown software with commercially available software based on appealing selling factors. The commercial software was "developed by professionals", it was "developed according to software development standards", and it was "approved by a regulator".

The reality is that the industrial scientists found the software to be very buggy and they didn't have the source code to fix it. Since the Canadian market is very small compared to that of the international software suppliers, response to bug fixes and requests for changes are slow or non-existent. More disturbingly, the commercial suppliers take no responsibility for problems that occur with the software in use. The installed environment and the needs for the software are highly variable. In one case, when the scientists could not get the attention of the supplier to provide fixes, the commercial software was shelved and replaced by the old in-house software. One problem is the commercial suppliers do not see themselves as active participants in the specific scientific question that needs to be answered.

5.3 End-User Development Model

Segal has characterized scientists as "professional" end-user developers [21]. These are people who work in highly technical, knowledge-rich environments, They are proficient with formal languages and abstraction, and as such, they have few problems with coding and learning software languages.

Segal documented a case study where laboratory space scientists and software engineers were teamed to develop a new piece of software [20]. The mismatch in their approaches to managing the development stemmed largely from the scientists' exploratory approach to determining the details of the software and the insistence of the software engineers for up-front requirements so they could fulfill their contract of software development. It was a conflict between software engineers applying the waterfall methodology and scientists applying something that looked like Agile because of the iterations involved, but wasn't. The scientists were close to the scientific question that needed to be answered, and the software engineers did not see themselves in that same space.

Fischer has described an end-user development model [10] but his examples illustrating its use do not include the development of scientific software. Fischer classifies end-users as everything from retail employees designing a kitchcn cabinet layout to people who are "techno-sophisticated" and comfortable with computer technologies. He further describes his "prosumers" (as opposed to consumers) as people who "have little fear of hacking, modifying, and evolving artifacts to their own requirements. They do not wait for someone else to anticipate their needs, and they can decide what is important for them. They participate in learning and discovery and engage in experimenting, exploring, building, tinkering, framing, solving, and reflecting." This sounds very close to the environment of the scientific software developer in our interviews, except lacking the risk-aversion factor.

Fischer offers a model of software development for his end-user communities. He describes a Seed-Evolve-Reseed (SER) model of software design that "creates open systems at design time that can be modified by their users acting as co-designers, requiring and supporting more complex interactions at use time. SER is grounded in the basic assumption that future uses and problems cannot be completely anticipated at design time, when a system is developed. At use time, users will invariably discover mismatches between their needs and the support that an existing system can provide for them."

Fischer does not give details on how his SER approach will be implemented, but asks, "How we can support skilled domain workers who are neither novices nor naive users, but who are interested in their work and who see the computer as a means rather than as an end? How we can create co-adaptive environments, in which users change because they learn, and in which systems change because users become co-developers and active contributors?" Again, this appears to be philosophically in line with what scientists need.

As part of the interviews in 2012, an SER approach to software development was discussed with the scientists and engineers. The SER approach was described as creating a central core of highly flexible software for users to readily modify as required (Seed), observe what the users do with the software (Evolve), and issue a new version to better support user development efforts (Reseed).

For most cases, it was not readily apparent how the SER approach would work with current software, its uses, environment, and community. Fischer explains how his approach "reduces the gap in the world of computing between a population of elite high-tech scribes who can act as designers and a much larger population of intellectually disenfranchised knowledge workers who are *forced* into consumer roles." The most apparent problem was that of scale. When the user community is not much larger than the development community, and when the two are overlapping significantly, as is often the case with scientific software, then the SER model is unwieldy.

Only one software system described during the 2012 interviews fit the SER model. Due to regulatory requirements in the application domain of this software, the space of time between new software releases is very long (as much as two years). The regulator has imposed a waterfall style software development standard that requires extensive documentation, several phases of verification and validation, and a plethora of plans.

To mitigate the impact of long release times on users, the developers of this software have pushed as much of the choices made in the modeling software out to the user in the form of data input. The software is about a half million lines of FORTRAN code and allows the user immense flexibility in what models they can build and how. The scientists interviewed described the unexpected creativity of their users and the broad range of questions they are addressing. There is a large well-connected user community, extensive user documentation, and courses given by the development group. The draw-back is complexity. Building a model and successfully running it requires a great deal of domain knowledge. One of the developers said it took years for a new user to become independently proficient with the software. There is a trade-off. Using an SER-like model for the design of this piece of software increases the complexity of its use substantially. Card and Glass [7] noted that there is an inherent complexity in any problem and the solution can push that complexity around, or make it worse, but not get rid of it.

6 Amethodical Software development

One of the scientists interviewed worked in a regulated industry under the auspices of a software quality standard that was based on the waterfall model. The scientist had been tasked with "backing out" requirements and design documents for a software system that had been in production for fifteen years. If the documentation was not completed, the system had to be taken out of service.

The scientist researched the literature on software design to determine the computing industry's recommendations. After her research, she concluded, "In the software engineering literature, software design is in a state of upheaval." This is a telling observation from a capable and informed outsider of how software engineering presents itself. Instead, this scientist pointed out a paper by Truex et al [22] that deconstructed the word "method" and offered a different view of software development methodologies. Truex et al offer the following quote: "We have conflicting evidence as to whether systems development methods are ever used; or that they work successfully if indeed they are ever used." Instead, Truex et al explain that software development mostly is *amethodical*, and they continue in the paper to explain what they mean.

They offer a definition of amethodical software development: "Amethodical systems building implies management and orchestration of systems development without a predefined sequence, control, rationality, or claims to universality." They also clarify that "… amethodical may reject structure but does not imply anarchy or chaos."

The characteristics of method and *amethod* that these authors describe were found to fit well with the stories scientists were telling in the 2012 interviews. The balance of this section explores these different characteristics and their relation to the development of scientific software by scientists.

6.1 Methods Require a Nearly Perfect Fore-Knowledge

Truex et al state [22], "In order for methods to actually control information systems development, developers must hold a nearly perfect knowledge of a throng of interrelated factors."

Kelly [12] has described five knowledge domains that capture the knowledge areas that contribute to developing a software system. These are summarized in Table 1.

Knowledge Domain	**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: Knowledge common to all software development.

For any waterfall or its related methods (eg., V-model, Spiral) the assumption is that all knowledge is acquired and recorded before the system is built. It has been generally agreed that this approach does not work for the majority of software systems.

If we examine one of the more popular Agile methodologies, Extreme Programming, it fails in this regard as well. Extreme Programming allows any developer to make changes anywhere in the system as they see fit, which means that they are fully capable of doing this at any time. Developers do testing, but the onsite customer does the acceptance testing and makes the decision that the software is fit for use. Again the customer is seen as fully capable to do this.

Looking at any job site that advertises for programmers, systems analysts, or the like, one finds that nearly all the postings ask for experience in at least ten different technical skills in areas such as programming languages, hardware environments, and tools. In other words, employers are looking for people who are fully capable in Execution Knowledge and Software Knowledge. There is no accommodation for learning-on-the-job. This leaves three of the five Knowledge Domains to be covered off by the knowledge of the customer. In other words, the on-site customer, to make her decision on the acceptability of the software, has to know the Physical World in which the software will be embedded, the Theory that should be included in the software that underlies both its models and its use, plus current and future use (Operational Knowledge) of the software for its intended lifetime. These are ambitious expectations for the on-site customer and can only work in systems that are very simple or that are exact replicas of something already developed.

The scientists who were interviewed, instead talked about learning as they developed the software. Their learning is spread over all five Knowledge Domains and they insist a valuable team member is open to this broad learning. They also talked about expectations for new-hires. The key expectation is problem solving ability, not technology and not development paradigms. Every scientist interviewed talked about mentoring new-hires and expected to do a lot of this.

The environment of scientific software development as described by the interviewees is a learning environment where no one is expected to "know it all" at the outset, but expected to develop a broad knowledge in all five knowledge domains (as in Table 1).

6.2 Cooperation and Communication More Important than Sequence of Activities

Truex et al state [22], "Any causal chain of dependent intermediate products in an information systems development project is largely imaginary. Any one of the chain of products can be created first, and all others constructed for compatibility… In fact, the assumption that exactly one activity must be first is methodical." They point out what are the drivers of *amethodical* development: "Compatibility of activity outcomes is not really dependent on their sequence, but rather on cooperation, communication and negotiation."

The industrial scientists interviewed talked about "agreement in the group how to code", "a lot of continuity, hallway conversations, and collaborative work", and activities where "any change that isn't obvious, [the scientist] sits with one other person, prints off the code, and chats over it." In all the interviews, there were conversations about collaboration and communication. This is the de facto way of working for these scientists, which fits with the risk averse environment of these scientists.

6.3 Human Choice Instead of Predefined Order

Truex et al state, "Amethodical information systems development is the outcome of the exercise of unprogrammed, independent human choice during every development project activity."

The industrial scientists interviewed were highly skilled people. Their skills form the basis of their problem solving ability and their programming ability. Given the multitude of factors involved in the solutions they are seeking (e.g., solution techniques, numerical grids, coordinate systems, finite approximations, etc.), the scientists explore and make decisions based on their explorations. Their exploration and decision-making are driven by the science, the difficulty of different parts of the problem, and availability of information, amongst other considerations. All the scientists talked about evolving their software and in every case the evolution started in a different place. At the same time, the software they were discussing were key systems currently in production.

6.4 Creative Use of Ideas Instead of Predefined Activities

Truex et al explain that *amethodical* means being non-conformant in how tools are used and activities carried out during software development. “The amethodical assumptions suggest that information systems development is a pastiche of activities, events and products. … Each element of the development pastiche is very likely to violate, in some way or another, the structures of any particular information systems development method.”

A case study carried out in 2010 [11] described how a scientist designed his own approach to testing a new piece of software despite being directed to use a standard software engineering testing method. The new approach worked far better for the scientist’s goals. The 2012 interviews also revealed a tendency for the scientists to use tools and activities in unique ways.

In one case, the scientists were mandated to use code walkthroughs as part of their quality assurance activities. They shared the target code well before the scheduled walkthrough meeting and an informal back-and-forth interaction amongst the scientists ensued. Questions were asked, an open door policy reigned, and problems with the code were sorted out. The walkthrough meeting was reduced to a formalization of the paperwork.

Even at the coding level, the scientists were aware of the non-conformist direction they had chosen. One commented that their code “might be ugly with respect to computer science but the science is readable.” Given that the science is of utmost importance, having the “science readable” is the preferable route for the sake of trustworthiness over the long lifetime of the software.

7 What the Scientists Interviewed Do Instead of Method

7.1 General Approaches

From the 2012 interviews, it was apparent that there was a common set of activities the scientists used to assure the trustworthiness of their software. Along with this was a common set of values that underwrote the activities. The most apparent characteristic driving their activities was risk-aversion.

First on their list was testing. Testing was focused on the science. It was iterative and exploratory. The approaches used were as varied as the scientific questions being modeled. Scientist wrote test harnesses for some code, examined the output of small units of code against known solutions, offered up completed code for users to try on real-world problems, examined sharp physical transitions to see if they were reproduced correctly, ran the codes against whatever real world data they had, reran the new changes on important data and compared to earlier runs, slowly built up sections of code and tested as they went carefully considering the output data at each step. Testing was integrated with code development and learning.

Next on the list was code reading. All interviewees mentioned reading the code as a key activity. One scientist commented, “When we got this code, we worked through it line-by-line to understand it.” Another commented, “If you don’t understand what’s in the code, you haven’t a hope of finding a bug.” This comment also relates to the expected broad knowledge base.

To support code reading, all the scientists interviewed were adamant that the code had to be self-explanatory. They wanted *useful* comments and plenty of them. Functions and variables were to be well named and tied to the science they represent. One scientist commented, “No Hungarian, please!” Another scientist described how they established a scientific vocabulary that was carried over to the variable names in the code. Another scientist said an important tenet for writing code was “don’t obscure the algorithm”. The view is that the science a developer embeds in the code must be apparent to another scientist, even ten years later.

All the scientists viewed their current jobs as long-term commitments. They spoke of two months to two years required to become independently proficient with their software and that the codes are repositories of “hundreds of man-years of research”. They spoke about “pride of ownership” and “personal responsibility” for the code they wrote, expecting to still be involved with their codes ten or more years down the road. This is a very different mindset from the revolving door careers faced by many computer science graduates.

Scientists, according to the interviewees, need to understand the “big picture”. One stated, “Scientists need the spark, inquisitiveness, desire to know everything.” Every part of the software product is integrated with the solution to the scientific question being examined.

Finally, teamwork plays an important role in the development of scientific software. All interviewees mentioned working in a dedicated team. One commented, “Silos don’t work.” Another said about students looking for a career job, “Find a company where you enjoy working with the people.” All the interviewees talked about agreement in the team, continuity, collaborative work, people eager to help each other, discussion of problems, and communication. This is an important factor in the success of their software, especially over the long term.

7.2 Onion Model for Phased Releases

The “onion” refers to the layered user community that exists for some examples of scientific software. Easterbrook and Johns [9] describe layered releases in a case study of climate modeling software. The user community in these examples of scientific software consists of one segment that may be co-located with the developers and works closely with them. A second segment is physically and knowledge-wise more removed from the developers. The core of the onion is the group of developers.

Two pieces of software described in the 2012 interviews followed a layered release scheme. New versions of the software are released from the core to the first layer, the user segment closer both physically and in domain knowledge to the developers. Problems with the new release are handled informally and quickly using face-to-face discussions and an open-door policy with this segment of users. One software group continued with the software released to the first user segment for a year, the other for two years, before the software was released to the second segment of users.

The inner core of users have a good understanding of the software and use it in “real world” work, giving the software a thorough test in the working environment. Their close relationship with the group of developers allows them to work efficiently and effectively through problems that crop up. Considered part of their beta-testing, one of the developers listed this as one of their most effective and important strategies.

8 Conclusions

The development of scientific software as described by the interviewees is a poor fit with the standard development methods prescribed by software engineering literature. One interviewee was worried because he was not following “software engineering best practices”, that he was doing things “ad hoc”. Yet, when he described his development activities, they were thoughtful, focused, meticulous, careful, and effective. He was not doing things “ad hoc”, but *amethodically*.

Another contributor to problems between software engineering and scientists is terminology. If specifically asked if they do “code reading”, or “unit testing”, some of the interviewees said, no. Yet, when they describe their development activities, they do something very close that would merit a “yes”. The difference is that they have invented an activity that isn’t exactly “by the book”, an activity that works better for their situation. The scientists interviewed were far more aware of good software practices (for their environment) and the pitfalls that software entails, than scientists in general have been given credit for.

Educating young scientists and engineers in good software practices is important. But so far, there has been a failure to identify suitable content for that education. This failure was discussed in two IBM CASCON workshops [13]. The scientists interviewed offered some suggestions. They said a plethora of tools and languages is not useful. This comment is based on the long lifetimes of the software as opposed to the rapid rate at which many tools and languages come and go. Also not useful are process, procedures, and paradigms. Instead the scientists want to see their new-hires arrive with strong abilities in problem solving, communication, and knowing the fundamentals of computing environments and of the appropriate science or engineering. They also want to see a love of learning, commitment, responsibility, and an open mind.

After observing scientists working in their environment, Segal [19] concluded, “There is no such thing as ‘one model fits all’, and that the way in which professional end-user developers [aka, scientists] construct software makes perfect sense in the context of being embedded in a close-knit scientific user community.” The context of the scientific development environment is key to their mode of work. It is important that managers, the scientists themselves, and computing professionals look closely at the context of any software development activities they are considering. It is and has been detrimental to enforce paradigms and processes from one context to another, particularly from contexts outside the scientific domains. The five factors we described at the outset of this paper may help to place the types of software development activities best suited to an environment. The industrial scientists' *amethodical* approach to software development seems to be well suited to their particular environment. It is an interesting option that should be explored further by the software engineering community at large.

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 interviews.

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] Agile Alliance website "What is Agile Software Development?", http://www.agilealliance.org/the-alliance/what-is-agile/; accessed January 20, 2012

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

[4] Boehm, B. & Turner, R. Balancing Agility and Discipline. Addison-Wesley, Pearson Educational. 2004

[5] Ronald F. Boisvert, Ping Tak Peter Tang, editors, The Architecture of Scientific Software, Klewer Academic Publishers, 2001

[6] Business Computing World, UK, online, February 17, 2012; accessed on 10 January 2013 at http://www.businesscomputingworld.co.uk/it-job-churn-is-putting-business-continuity-and-recovery-at-risk/

[7] David N. Card, Robert L. Glass; Measuring Software design Quality; Prentice Hall, 1990

[8] George Dyson, Turing's Cathedral – The Origins of the Digital Universe; Pantheon Books, New York, 2012

[9] S. M. Easterbrook and T. Johns, "Engineering the Software for Understanding Climate Change", IEEE Computing in Science and Engineering, Vol 11 (6), pp. 65-74. November 2009

[10] Fischer, G., "End-User Development and Meta-Design: Foundations for Cultures of Participation", Journal of Organizational and End User Computing, 22(1), pp. 52-82., 2010, available at http://l3d.cs.colorado.edu/~gerhard/papers/2010-JOEUC.pdf

[11] Diane Kelly, Stefan Thorsteinson, and Daniel Hook, "Scientific Software Testing: Analysis in Four Dimensions", IEEE Software, May/June 2011, pp. 84-90; preprint available online in IEEE Computer Society Digital Library, Jan, 2010.

[12] Diane Kelly, "Innovative Approaches for Developing Scientific Software", Journal of Organizational and End-User Computing, special issue on Scientific End-User Computing; October-December 2011, Vol. 23. No. 4, pp. 63-78

[13] Diane Kelly, Spencer Smith and Nicholas Meng, "Software Engineering for Science", IEEE CiSE, Vol. 13, Issue 5, Sept/Oct 2011, pp 7-11

[14] Diane Kelly, Rebecca Sanders, "Mismatch of Strategies: Scientific Researchers and Commercial Software Suppliers", The Software Practitioner, ed. Robert Glass, July 2007

[15] Pitt-Francis, J., Bernabeu, M.O., Cooper, J., Garny, A., Momtahan, L., Osborne, J., Pathmanathan, P., Rodriguez, B., Whiteley, J.P., Gavaghan, D.J.; "Chaste: using agile programming techniques to develop computational biology software", Philosophical Transactions of the Royal Society, 366, 2008, pp. 3111-3136

[16] Eric S. Raymond, "The Cathedral and the Bazaar", available at http://archive.org/details/CathedralAndTheBazaar; accessed Jan. 20 2012

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

[18] Rebecca Sanders, Diane Kelly, "Scientific Software: Where's the Risk and how do Scientists Deal with it?", IEEE Software, special issue on software engineering for computational software, July/August 2008, pp. 21-28

[19] Segal, Judith. "Models of scientific software development"; SECSE 08, First International Workshop on Software Engineering in Computational Science and Engineering, 13 May 2008, Leipzig, Germany.

[20] Segal J., 'When software engineers met research scientists: a case study', Empirical Software Engineering, 10, 2005, pp. 517-536.

[21] Segal, J., "Professional end user developers and software development knowledge". Open University Technical Report No.: 2004/25, 2004

[22] Truex, D.P., Baskerville, R. and Travis, J. "Amethodical Systems Development: The Deferred Meaning of Systems Development Methods," Accounting Management and Information Technologies (10), pp. 53-79, 2000.

[23] Van Vliet, H., Software Engineering Principles and Practices, Wiley, Chicester, England, 2000

[24] Wilson, Gregory V.,"Where's the real bottleneck in scientific computing?", American Scientist, 94(1), 2006, pp. 5-6