Models of scientific software development

Judith Segal
The Open University
Milton Keynes MK7 6AA UK
+44 1908 659793
j.a.segal@open.ac.uk

ABSTRACT
Over the past decade, I have performed several field studies with scientists developing software either on their own or together with software engineers. Based on these field study data, I identify a model of scientific software development as practiced in many scientific laboratories and communities. This model does not fit the standard software engineering models. For example, the tasks of requirement elicitation and software evaluation are not clearly delineated. Nevertheless, it appears to be successful within the context in which it is used. In the context in which scientists collaborate with software engineers, however, I describe problems which arose from the clash of this model with a traditional, phased software engineering model. Given these models, I discuss the issues which have to be addressed in order to determine the software techniques and tools which might best support scientific software development in different contexts.

Categories and Subject Descriptors
H 1.2 [**Human factors**]:

General Terms
Human Factors

Keywords
Scientific software development, software development practice, field studies

1. INTRODUCTION
The over-arching question underlying my research is: how might software engineers best support professional end user developers such as scientists? Professional end user developers [1] are people working in highly technical, knowledge rich professions, such as financial mathematicians, scientists and engineers, who develop their own software in order to advance their own professional goals. Unlike many end user developers, they are used to formal languages and abstraction and hence tend to have few problems with coding per se. Like all other end user developers, however, they do not describe themselves as software developers and have little formal education or training in software development.

In order to address my over-arching question, I feel it is imperative to try and uncover how professional end user developers actually go about their development tasks, in order to identify tools, techniques etcetera which might meet their needs and fit in with the context of their software development. I thus undertook several field studies, of financial mathematicians, of earth and planetary scientists, and most recently, of structural biologists [1], [2] and [3].

As I shall describe in section 2 below, my field studies reveal ways of software development by scientists which run counter to traditional models of software engineering. For example, requirement and software evaluation activities are not clearly delineated. Nevertheless, these models appear to be successful in a particular context, which I shall characterise.

Section 3 again draws on my field study data to describe the problems which occurred when the context of development changed and it became necessary to involve software engineers in the development process because the software involved was just too complex for professional end user development. I shall illustrate the clashes which occurred between the scientists, who believed they knew how to develop software (but according to the model described in section 2), and the software engineers, who believed they knew how to develop software (but according to a traditional phased model of software development).

In section 4, I reflect on the limitations of my field study data and probe the question of whether other contexts and other viable models of scientific software development exist. Section 5 returns to the data of an ongoing field study and is thus somewhat speculative. Whereas sections 2 and 3 concentrate on the differences between professional end user developers and the more traditional software engineers, section 5 indicates how both groups might learn from each other about scientific software development In section 6, I return to my over-arching research question in order to discuss the issues which need to be addressed in order to identify those established software engineering techniques and methods which best fit the various contexts of scientific software development.

I begin by considering scientists developing software.

Conference'04, Month 1–2, 2004, City, State, Country.

2. A Model Of Scientific Software Development By Scientists

I shall begin by discussing possibly the most common context in which scientists develop software, that is when the software is intended for use either by the developer herself/himself or by closely co-located colleagues, for example, people working in the same laboratory. I shall then describe another context of scientists developing software in which the software is developed within a closely co-located group but intended for the wider scientific community of which the developers form a part.

My field studies, [1], [2] and [3], reveal that scientists who develop software for themselves or for their close colleagues, do so in a very iterative and incremental manner. Requirements emerge, as the understanding of both the software and the science evolves. A piece of software is produced, some reflection takes place ('is this really what I/you want? Can I improve it?), and development continues. There is no discrete phase of requirements gathering or of software evaluation. Testing is of the cursory nature which would enrage a software engineer ('does the software do what I expect it to do with inputs of the type I would expect to use? If I don't have any real expectations of the output, does the software behave in a way I really would *not* expect given inputs of the type I would expect to use?').

The development of scientific software in the context in which the software is not intended for local use but rather for use within a close-knit but distributed scientific community does not seem to differ significantly from the description above. In one situation with which I am familiar, the core developers are firmly established as scientists in the scientific community, are working either in the same room or in adjacent rooms and have been working together for at least 4 years. The requirements for the first version of a piece of software arise from the developers' own experience of the scientific field and from discussions with local scientists. After testing is conducted in the same way as described above, the software is sent to another 'trusted' laboratory ('trusted' in the sense that the developers are confident that this other laboratory will engage with the software and communicate back their findings), before being released into the wider scientific community. I infer that there is a sense of community-wide ownership in the software from the fact that the community contributes many 'bug reports' (which either do, in fact, report bugs or make suggestions for improvements, that is, suggest new requirements). In addition, a few members of the wider community have contributed extensions to the original software.

The model of software development described above runs counter to any traditional model of software engineering especially as regards testing, but appears to be widespread. For example, researchers studying scientists carrying out high performance computing developments have also noted a reliance on emergent requirements, see, for example, [4]. The presumption must be that, in the context in which these models are used, scientists feel that they work. I shall now discuss the characteristics of this context.

The gathering of requirements in this informal way is predicated on the developer(s) being firmly embedded in the community and so having strong intuitions as to the initial requirements; being able to develop quickly a piece of software to reify those requirements and then being able to evaluate this software by having scientists close at hand (or well known to the developer) who are willing to engage. (It's much more difficult to resist a request from a close colleague to 'have a look at this and tell me what you think' than a more formal request). As to testing, I will now propose an argument that the cursory nature of testing as described above is entirely consistent with the nature of experimental science. As many philosophers and historians of science attest, see, for example, [5], scientists take as a given that the apparatus by which they obtain their data works. Only if the data run counter to what the scientist broadly expects, does she/he draw back and start examining her/his underlying theory and assumptions, one of which is that the apparatus works. If this is true when the apparatus is a telescope, may it not be equally true for software regarded as a mechanism for obtaining or manipulating data? If the developer and any other users get broadly expected, or at least, not totally unexpected, results from the software, is it not then consistent with their experimental experience for them to assume the software works?

What I am saying here is that there are at least some situations in which software engineers should not try to impose the full machinery of traditional software engineering on scientific software development. It is not the case that scientists developing their own software in the contexts discussed above are able coders but totally undisciplined, as was once said to me by a software engineer. They are just as disciplined as the context demands. A group of professional end user developers following an unsuccessful collaboration with software engineers used the term 'factory methods' disparagingly to mean the traditional phased model of software development, in which each phase, such as requirements elicitation or testing, is discrete and undertaken by different people, or by the same person with different hats on. The scientists were quite clear that these methods were not appropriate in their context of development.

There are, of course, other contexts of scientific software development, and in the next section I shall describe some problems which arise in the context when software developers and scientists collaborate, and which may have their origins in the very different models of software development held by these two groups.

3. When Scientists Meet Software Engineers

In this section, I shall describe (aspects of) two field studies. In both, the scientists were the customers of the software development. In the first, the focus is on the scientists failing to meet the software engineers' expectations based on the latter's model of traditional, phased software development. In the second, the emphasis is on the software engineers failing to meet the scientists' expectations based on the latter's model of software development as characterised in section 2 above.

3.1 Scientists failing to meet software engineers' expectations

The first situation is described in [3] where I discuss the difficulties that arose when some space scientists and engineers collaborated with software engineers in order to develop a library of components for embedded instrument software, and attempted to follow a phased, traditional software development as recommended by the European Space Agency. Requirements posed the biggest problem: the software engineers expected to receive a formal requirements document; the scientists consistent

with their previous experience of software development expected the requirements to emerge. User testing was done in the cursory manner described in section 2 above. Specification documents, did not fulfil their communication role (the scientists were used to informal, face-to-face communications). Nevertheless I have to report that the instrument with its embedded software was delivered to the satellite in time, though it isn't yet known whether the software will perform as expected (the instrument does not reach its destination for at least another 5 years).

3.2 Software engineers failing to meet scientists' expectations

In the second as-yet unpublished field study, the development was steered by scientists who were, or had been, professional end user developers. Among other challenges facing the development were those of requirements and scheduling.

In the following extract from a tape transcript, one of these scientists describes his experience of providing requirements in his own laboratory. This scientist had developed his own software for decades but, having risen to the top of the scientific tree, is no longer doing so.

> 'So all I told one person [the developer] was: I want you to find a way of doing a … fast graph matching problem, in an interface that is easy to use and shows you everything you need to know on the interface, and that's all I said. And he went away for a year and came back and here is the system.'

The developer in question was a professional end user developer: he knew the science; he worked in the laboratory where the software was going to be used, and thus had the resources to continually firm up the requirements and evaluate the software in the manner described in section 2 above. It was thus absolutely reasonable to provide this particular developer with such a terse statement of requirements.

The scientist went on to say that this situation is typical:

> 'In most of the types of things we … think of a requirement to do, we don't know the requirements at a precise exact level, we don't know the answer in any way, we can define the problem basically in half a page of text and expect the [developer] to go away and do it.'

But of course problems arise with this approach when the developer is not a professional end user developer but a software engineer. In this case, the software engineer doesn't understand the science; doesn't have any intuition as to the nature of the requirements; is not embedded in the particular scientific community of practice. The informal way of gathering requirements as described in section 2, does not work. One can't just, as in the quote above, 'expect the developer to go away and do it'.

Another obvious point of departure between the professional end user developers' model of software development and that of the software engineer, is to do with the time that development takes. The professional end user developer does not usually have to worry about explicit requirements gathering or testing (as discussed above); or about portability, or about forced maintainability when for example, operating systems or third party software is upgraded, or about security of data or working on a shared code-base. (I am not talking here about professional end user developers who work on high performance computing systems, which presents a slightly different case, see [4]). These issues add greatly to development time, and the evidence from my field studies is that professional end user developers, used as they are to a model of quick iterative development, do not appreciate this. A software engineer told me that for any particular software task, his estimate of the time it would take was usually about three times greater than the estimate of the professional end user developers with whom he was collaborating.

In this section, I've discussed the problems that arose in collaborations between scientists with one model of scientific software development and software developers with another (though of course, the expectations of the professional end user developers described in 3.2 would probably not be met by any software engineer regardless of the model of development he/she espoused). This discussion is based entirely on my field study data and might thus be thought to be somewhat limited. In particular, my field studies only revealed two scientific software development models, one for the software engineers and one for the professional end user developers, and were conducted in a limited number of contexts. In the next section, I shall discuss other models and other contexts.

4. Limitations of my field study data

As described above, from my field study data I have identified one model of scientific software development for scientists and one for software engineers. The former is a model of rapid, incremental, iterative development with integrated phases of requirements gathering and software evaluation, and somewhat cursory testing of the software at the end of (what may be termed) a release. The latter is based on the traditional, phased, broadly waterfall model. The question arises as to whether other models exist (and just haven't arisen in my studies).

The answer in the professional end user developer case is probably 'yes, to a certain extent' – there are, I think, deviations to the model in the context of high performance computer systems, where effort has to be expended on code optimisation and parallelisation, [6]. Nevertheless, in the absence of any disconfirming evidence, I argue that the model described herein is the standard model by which professional end user developers produce software in a context which does not involve high performance computing and in which the software is being developed for use by a close-knit scientific community to which the developer belongs.

The answer to the question of whether there are any models of software development not identified in my field studies, is very definitely 'yes' as regards the software engineering world. This does have models of development other than the traditional, phased water-fall model. It just so happens that this latter was the one espoused by the software engineers in my field studies, but there are other non-traditional models around. The ones which are attracting most attention at the moment, I think, are the ones which come under the umbrella of 'Agile Methods', subscribing to the values of the agile manifesto (http://www.agilemanifesto.org). These include XP (eXtreme Programming), [7], DSDM (Dynamic Systems Development Methods), see http://www.dsdm.org/, and the Crystal family of methodologies, [8]. Each of these methodologies presents a coherent account of tried and tested practices in such software

development approaches as rapid application development, prototyping and incremental development. And as I have demonstrated above, rapid application and incremental development are at the heart of professional end user development practice. So my next question is: could the 'clashing models' problems described in section 3 be alleviated if the software engineers were to espouse an agile model of software development? There is some evidence in the literature to suggest that the answer to this question is 'yes'. There have been some case studies, for example, [9] and [10], in which software developers have described successful experiences of using agile methods with scientists. On the other hand, the experience of my co-editing a special issue of IEEE Software devoted to developing scientific software has revealed that many scientists equate 'agile' with the model of professional end user development described in section 2 above, and take no cognisance of the disciplined practices of each of these methods.

I explored the question of agile methods in the case of the field study with the space scientists, [3], where, following the approach of [11], I suggested that there were some parts of the software – the 'back end' parts essentially – where it would be profitable to use a traditional phased software engineering approach to development, and parts, notably those to do with user requirements, where agile methods might be preferred.

My field studies, of necessity, only covered a limited number of contexts of scientific software development. These were contexts in which developers and users were embedded in the same close-knit scientific community as in section 2, and where scientists were working in partnership with software engineers, as described by section 3, either because the software was rather too complex to be developed by scientists alone or because it was designed to serve the needs of a rather disparate scientific community. It is easy to imagine contexts of scientific professional end user development other than those described by my field studies where the model described in section 2 does not suffice and the development of scientific software could be improved by better use of software engineering techniques and tools. For example, there is the situation of 'software creep', where a scientist develops a piece of software as a one-off to solve a particular problem; the software is then adopted by his colleagues who modify it in an ad-hoc manner to solve each of their own particular problems; it then becomes part of a suite of software available to the whole community - and lurking within it is the ticking time-bomb of untested ad-hoc cobbled together software. There is also the example of research software developed within a research environment according to the model described above, being redeveloped to become production software for use outside research laboratories in, for example, medical environments. Other examples include those where the software is very complex or safety critical. Just as I argue that it is not the case that scientists developing software should always be encouraged to adopt blanket software engineering methods, so I argue that it is not always the case that software engineering methods are entirely irrelevant to scientific software development. Consider for example the 'factory methods' (phased development) described in section 2 as being unfit for the development of scientific software to support the research of a close-knit scientific community. It is clear that these methods are sometimes perfectly fitting for scientific software development, for example, in the redevelopment of research software to production software when all that the former might contribute to the latter is the requirements as specified by the behaviour of the software. I shall return to the issue of fitting software engineering methods to the appropriate scientific development context in section 6 below.

In the sections of the paper so far based on the field study data, that is sections 2 and 3, I have been at pains to contrast the models of scientific software development held by scientists and software engineers and to describe the problems that may be caused by the clashing of these models. In the next section, based on an ongoing field study and thus very preliminary, I demonstrate that it is possible for scientists and software engineers to learn from each other about scientific software development.

5. Learning from each other

Despite its title, this section focuses on professional end user developers – or rather, one professional end user developer – and the lessons he learnt from working with software engineers. This reflects the fact that my field studies are ongoing and I hope to pursue the topic further. At the end of the section, I speculate on what software engineers might learn from working with professional end user developers, but I don't actually know whether there are firm grounds for this speculation: this is still a matter for investigation.

The developer who is the focus of this section is a typical professional end user developer He describes himself emphatically as a "research scientist" despite the fact that his working life is spent on developing or modifying software for the use of the laboratory in which he works. He drifted into this work as a result of the software development he had to do for the purposes of his PhD thesis (in science) which led to his gaining a reputation in his laboratory of being a capable software person. Both his scientific colleagues in the lab in which he works and the software engineers with whom he collaborates hold him in great respect both for his domain knowledge and for his coding ability.

In an interview with the writer, he said he had learnt 'a vast amount' from working with software engineers. Part of it was the result of working on a shared code-base, which he had never done before.

> 'Pretty much previously, if I'd needed to write something, I wrote it, all of it. It was rare even in Fortran that I would import a library that wasn't a maths library…to some extent I had done a bit before of maintaining other people's code, but not as varied as in [the collaboration with the software engineers] of going into a bit of code and seeing that it's written in a very different way to the way that I would have written it. And that's interesting both in that you learn new ways of writing code and discover that the way you've been doing it is extremely long-winded and in many cases actually substantially less robust than the way someone else has done it – and that's quite interesting because before I never really had reason to review anyone else's code.'

His awareness of certain issues became heightened:

> '… it does come down to maintainability and portability and that is something I had been only dimly aware of previously'

And especially his appreciation of the importance of testing, as the following dialogue between him and the interviewer (me) demonstrates:

> Professional end user developer: 'That has also been something new for me. Testing just didn't happen. .. One assumes that if the numbers came out of the other end, they were right. And that is in hindsight, an embarrassingly stupid assumption."
>
> Interviewer: "Yes, but not an assumption you can always test as a scientist because you don't always know what the right output should be."
>
> Professional end user developer: "No but you can try and write code that is testable … and so the concept of writing code that is testable has been a very useful one that I have tried to carry over particularly from Java where I've learnt it to the other languages where it's not as easy or the tools don't exist to make it as easy."

As to what the software engineers might have learnt from the professional end user developers, I speculate it is the lesson that is one of the main arguments of this paper: that there is no such thing as 'one model fits all', and that the way in which professional end user developers construct software makes perfect sense in the context of their being embedded in a close-knit scientific user community..

6. Software engineering and scientific software development: a discussion

Thus far in this paper, I have identified various contexts in which scientific software is developed. These include: scientists developing software for their own laboratory or close-knit scientific community (where the model of software development described in section 2 might well suffice); software engineers developing software in partnership with scientists in section 3, and scientists developing software in contexts where software engineering tools and techniques might well have a place, such as where the user group is more diverse than that containing the developers, as discussed in section 4. I now turn my attention to the complex question of which established software engineering tools and techniques might best support scientific software developers in these various development contexts.

There are, I think, several issues to be explored here. The first three echo Glass [13], where he argues for a mapping between application domains and software engineering methodologies.

1. The identification of those established techniques in software engineering which might assist scientific software developers.
2. The characterisation of those contexts in which scientific software is developed, along dimensions such as: the degree of acceptable risk; the extent to which the developers are integrated in the same domain as the users; the extent of software engineering knowledge of the developers, etcetera.
3. The establishment of a mapping between software techniques identified in 1 and contexts as characterised in 2.

I introduce an extra issue to those of Glass:

4. How might scientists be made aware of those software engineering techniques and tools which might be relevant to their development?

With respect to the first three issues: in section 4, I touched lightly on the debate as to whether the problems encountered when software engineers and scientists developed software together, might be alleviated if the software engineers involved were to adopt agile methods. As far as scientists developing their own software is concerned, Wilson, [12], expresses concern at the lack of quality of much of this software, and found that scientific software developers of his acquaintance did not know of the most basic software engineering tools, such as version control systems. His response was to make available on the web a 'software carpentry course', in which he teaches techniques that he has identified as being most useful to the professional end user developer (see http://www.swc.scipy.org).

As to the fourth issue, there has been much concern expressed recently as to the size of the chasm between academic software engineering and practitioners (see, for example, [14] and [15]). One can only speculate on how much greater this chasm must be when the practitioners think of themselves primarily as scientists rather than as software developers. However good a job software engineers do at exploring the first three issues, their efforts will count for nought if scientific software developers don't at the very least become aware of their results.

I know of two attempts to introduce a course on software engineering techniques into science/engineering undergraduate courses (Wilson in personal correspondence with the author, and Kelly [16]). The course discussed by Kelly was deemed by her to be unpopular, which she ascribes in part to its not making enough links with the application domain. There is no evidence as to how successful these two courses were in subsequently influencing the scientist/engineers' development practices.

In section 5 above, I describe a professional end user developer who revised his philosophy of software testing after working on a project with a software engineer. I suggest that much software development knowledge among scientists is garnered by such happy accidents or by sharing knowledge with other scientists developing software. The sharing and creating of knowledge through communities (or networks) of practice is well-known; the problems this poses in the context of scientists developing software are discussed in some detail in [1]. One such problem is that there might not be a community of practice; there may be only one scientist developing software in a lab.

Clearly, much work needs to be done on the issues raised here. The contribution made by this paper is that it has articulated the relevant issues and made a start at addressing them. I look forward to a fruitful discussion.

Acknowledgements

I should like to acknowledge my heartfelt gratitude to all those scientists who were so generous in spending time with me and letting me into the secrets of their trade. I should also like to thank Chris Morris (especially), Diane Kelly, Marian Petre, Hugh Robinson, Helen Sharp and Greg Wilson, for many fruitful discussions.

7. REFERENCES

[1] Segal, J. 2007, 'Some problems of professional end user developers', VLHCC, IEEE Symposium on Visual Languages and Human-Centric Computing, 2007, 111-118

[2] Segal J., 2001, 'Organisational Learning and Software Process Improvement: A Case Study', in *Advances in Learning Software Organizations*, K-D Althoff, R.L. Feldmann, W. Muller (Eds.), Lecture Notes in Computer Science, Vol. 2176, Springer, 68-82.

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

[4] Carver J.C., Kendall R.P., Squires S.E., Post D.E., 2007, Software Development Environments for scientific and engineering software: a series of case studies. *Proc. ICSE 2007* .

[5] Chalmers, A.F., 1982. *What is this thing called science?* Open Universtiy Press, Milton Keynes, UK

[6] Squires S., Van de Vanter M.L., Votta L.G., 2006, 'Software productivity research in high performance computing', CT Watch Quarterly, November, 2006, 52-61.

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

[8] Cockburn, A., 2002, *Agile Software Development*. Addison-Wesley, Pearson Educational.

[9] Bache E. 2003, 'Building software for scientists: a report about incremental adoption of XP', at XP2003, Genoa, Italy.

[10] Kane, D. 2003. 'Introducing agile development into bioinformatics: an experience report', *Agile Development Conference, 2003.*

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

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

[13] Glass, R.L., 2004, 'Matching methodology to problem domain', Comm ACM, 47(5), 19-21

[14] Perry, D., Porter, A. and Votta, L. 2000. Empirical studies of software engineering: a roadmap. *Proceedings of the International Conference on The Future of Software Engineering*, Finkelstein, A. (ed.), ACM Press, pp 345-355

[15] Zelkowitz, M., Wallace, D. and Binkley, D. 2003. Experimental validation of new software technology. *Lecture notes on empirical software engineering*, Juristo N. Moreno AM (eds.), World Scientific Publishing Co., pp 229-263.

[16] Kelly, D.F., 2007, 'A software chasm: software engineering and scientific computing', *IEEE Software*, 24(6), 120-199.