Developing software for a scientific community: some challenges and solutions

Book Section

Handbook of Research on Computational Science and Engineering:

Theory and Practice

Joanna Leng
Consultant, UK

Wes Sharrock
University of Manchester, UK

Volume I

ENGINEERING
SCIENCE REFERENCE

Managing Director: Lindsay Johnston
Senior Editorial Director: Heather Probst
Book Production Manager: Sean Woznicki
Development Manager: Joel Gamon
Development Editor: Chris Wozniak
Acquisitions Editor: Erika Carter
Typesetters: Brittany Metzel, Lisandro Gonzalez
Print Coordinator: Jamie Snavely
Cover Design: Nick Newcomer

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Handbook of research on computational science and engineering: theory and practice / Joanna Leng and Wes Sharrock, editors.
p. cm.
Summary: “This book offers a timely introduction to the possibilities in computational science and engineering to advance the ongoing research and applications leading to the discovery of new resources and cutting edge developments”-- Provided by publisher.
Includes bibliographical references and index.
ISBN 978-1-61350-116-0 (hardcover) -- ISBN 978-1-61350-117-7 (ebook) -- ISBN 978-1-61350-118-4 (print & perpetual access) 1. Science--Data processing. 2. Engineering mathematics--Data processing. 3. Numerical analysis--Data processing. 4. Mathematical models. 5. Computer simulation. I. Leng, Joanna, 1965- II. Sharrock, W. W. (Wes W.)
Q183.9.H36 2012
501’.13--dc23
2011032075

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 8
Developing Software for a Scientific Community:
Some Challenges and Solutions

Judith Segal
The Open University, UK

Chris Morris
STFC Daresbury Laboratory, UK

ABSTRACT

There are significant challenges in developing scientific software for a broad community. In this chapter, we discuss how these challenges are somewhat different both from those encountered when a scientist end-user developer develops software to address a very specific scientific problem of his/her own, and from those encountered in many commercial developments. However, many developers of scientific community software are steeped in the culture of either scientific end-user or commercial development. As we shall discuss herein, neither background provides sufficient experience so as to meet the challenges of developing software for a scientific community. We make various proposals as to which development approaches, methods, techniques and tools might be useful in this context, and just as importantly, which might not.

INTRODUCTION

Many scientific software projects intended for a broad scientific community succeed in that they make a significant contribution to the science. Many, however, fail. Some of these fail for scientific reasons (the underlying science was imperfectly understood), or because of coding problems (for example, an inappropriate choice of implementation language). Another less obvious cause of failure is the differences in the behaviour, knowledge, values, assumptions and goals between three different groups of people involved in such projects. These three groups are

DOI: 10.4018/978-1-61350-116-0.ch008

Table 1. Two snapshots from the first author's field studies:

Scientist: Anyone can develop software. Why should we listen to the advice of a professional software developer?	(Professional software developer is deeply offended)
Professional software developer: We need to start off with a clear document of your requirements, and then we'll draw up a requirements specification document which you can check.	*Scientist:* But that simply isn't how we work.

scientists; scientific end-user developers, that is to say, scientists who are developing software for their own use or for that of their close colleagues; and professional software developers, to whom the science is just another user domain.

In writing this chapter, we draw heavily on the field studies conducted by the first author, an academic, in a variety of scientific settings, and on the many years' experience developing scientific software of the second author, a professional software developer.

Our aims in writing this chapter are:

- To articulate some specific challenges facing scientific software developers. These challenges have their origins either in the culture of scientific end-user development or in the nature of science itself.
- To suggest ways in which these challenges might be addressed.

In what follows, we shall firstly articulate the behaviour, knowledge, values, assumptions and goals that characterize much scientific end-user development and then discuss the challenges which these characteristics pose when the context of the development is broadened. We then go on to discuss which development approaches, methods/techniques and tools might be useful in scientific software development, and, equally importantly, identify some which will not. Finally, we discuss how this identification of effective ways of supporting scientific software development can be progressed.

Throughout this paper, we stress the importance of context. A couple of examples give a flavour of this importance:

- A particular tool which is useful in a commercial development context might not be so useful in a scientific;
- Assumptions which are perfectly justified in a setting where a scientist is developing software for himself/herself to explore a particular scientific question might not be justified in other development settings.

This emphasis on the importance of context means that it is difficult to set any hard-and-fast rules along the lines of 'scientific software developers should apply *this* testing technique to their software'. We hope rather that this chapter might provide the means by which you might recognise the challenges in your particular development context, and suggest some ways by which you might address such challenges.

There is a caveat which we should stress here. One chapter cannot possibly say all there is to say about the challenges facing developers of software for a scientific community. We focus here on the challenges posed by the culture of scientific end-user development, as revealed by our field studies. These studies did not include FLOSS developments (free libre open source software), see the later section on future research directions. We also took little cognisance of CSCW (computer supported cooperative work) literature. We comment further on this literature in the additional reading section.

A Pervasive Culture of Scientific Software End-User Development

Scientists have been engaging in end-user development, that is, in writing software in order to address their own scientific problems, for sixty years or more. Over these decades, a pervasive culture of scientific software development has emerged. 'Culture' is an overloaded term meaning different things to different people. What we mean here by 'culture' is the habits and normal behaviours, the accepted (though perhaps not articulated) values, assumptions and goals of a group of people, in this case, scientific end-user developers working in a traditional setting. Later we shall discuss those scientific end-user developers who work on codes which evolve over years often in a high performance computing (HPC) setting, but in this section we focus on scientists who write software, typically on a PC, in order to address a particular scientific problem of their own and/or of their close colleagues sitting round them. Their focus is entirely on the scientific problem. They have little or no interest in the software once the problem has been solved. Typically, these scientists' formal education in software development has been limited to a few Fortran lectures at University. Other than this, their knowledge of software development has been garnered informally from popular books on the subject, from the Web, from their colleagues, and often from the working codes they have encountered.

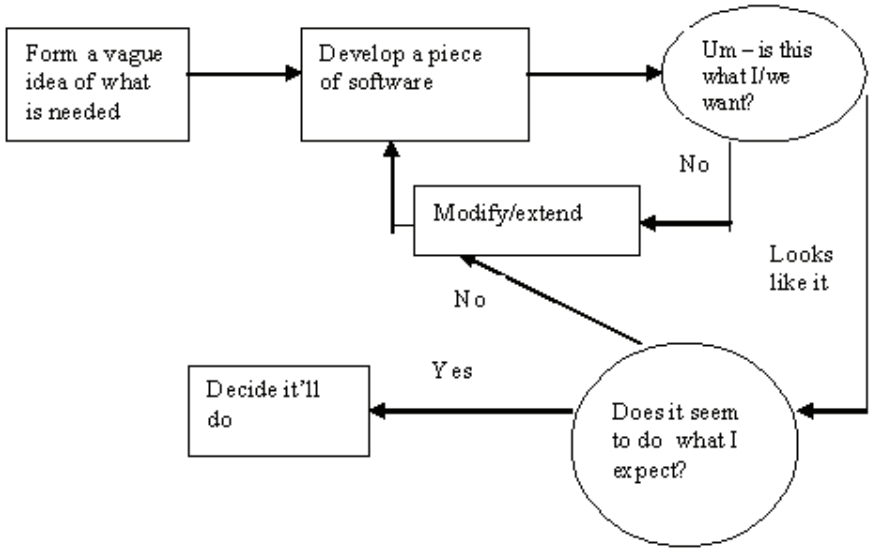
Field studies have been published of such scientific software end-user development activities among financial mathematicians, Segal, 2001, earth and planetary scientists, Segal, 2005, and structural biologists, Segal, 2009a. Despite the differences between the scientific domains and the fact that the financial mathematicians operated in a commercial environment and the other scientists in a variety of academic environments, the field studies reveal a common model of development practice and common values and assumptions in all the studies (Segal, 2007). Figure 1 is of this common model of development practice.

This model evokes instant recognition when shown to scientific end-user developers or anyone who has worked with them. Given that it has emerged over decades of scientific end-user development, it is not surprising that it is a very successful model in this context. But its success is entirely dependent on the characteristics of the context as we shall now discuss.

A professional software developer, knowing about the many models of software development in the software engineering literature (the waterfall model; the spiral model; the joint application development model; etcetera, etcetera, etcetera), would be taken aback by the model of Figure 1. Where are the activities for establishing requirements? Where does software design fit in? How can the question 'does it seem to do what I expect?' possibly act as a basis for testing? How can 'deciding that it'll do' be a viable acceptance criterion? How about issues of usability? The answers to all these questions lie in the context.

- *The establishment of requirements*. The establishment of requirements is an informal activity which pervades the whole of the development. The developer, as a potential user of the software, has a deep understanding of the scientific problem and certain ideas about how the software might address it, although these ideas are not necessarily articulated. He/she develops a piece of software to explore them. Then he/she reflects: does this piece of software address the scientific problem? Wouldn't it be nice if the software did *this*? The software should not do *that*. If there are other people around working on the same or similar issues, then it's easy for the developer to involve them in these reflections ('Come and have a look at what I've been working on').

Figure 1. A model of scientific end-user software development, adapted from Segal and Morris (2008)



- *Software design*. Design isn't an issue for a relatively small piece of software intended to address a particular scientific problem with the (perhaps tacit) expectation that it will be discarded when the problem has been successfully addressed.
- *Testing*. As with requirements activities and as described above, informal evaluation pervades the whole of the development. This evaluation is grounded in the developer's scientific intuition and judgement: the questions asked by the developer of the software are 'Does it seem to do what I expect?' and 'Does it seem to adequately address the scientific problem?' If the answer to either of these questions is 'no', then the software is modified and/or extended. The field studies referred to above provide no evidence of any formal testing activities.
- *Usability*. One reason frequently put forward for the lack of usability of many software products is that the developers have implemented the product as if they were the users. But in the case of scientific end-user software development, the developer *is* the user, or one of the potential users. Usability, like the establishment of requirements, and testing, is simply not a big issue

We now turn our attention to the values and assumptions commonly held by scientific end user developers working in the context above. We have seen that such developers have no reason to value the knowledge, skill and effort required to establish requirements or to design software or to test software or to ensure usability of the software. The implication is that they tend to see software development in terms merely of coding, a simple matter of translation of scientific ideas into a programming language. And given that scientists are, to a greater or lesser extent, used to the manipulation of abstract concepts and formal languages, coding does not pose them a major problem. Given all this, we should not have been as surprised as we were at the evidence from the cited field studies of the low value placed on software development knowledge and skill. The following quotes are from Segal, 2007:-

'I think the attitude towards computing.. [is] it's something you do in your spare time. I don't think people have any idea how long it actually takes to sit down and write a program. I think we quite happily imagine that you just ... spin it off in half an hour over your lunch time.' [planetary scientist]

'everybody in theory knows how to do [software development]…. It's assumed that everybody knows what to do' [financial mathematician]

This low value is reflected in appointment policies. Two examples from the first author's field studies are:

- A man was appointed to a post called 'project programmer' when his experience of developing software was limited to the Fortran course he had done at University (Segal, 2007);
- A leading scientist commented that it was common to appoint people to software development projects in situations where they had proved themselves as scientists and their current funding was running out, regardless of their software development skills and experience (Segal, 2009a).

The low value afforded to software development knowledge and skill is also indirectly reflected in reward structures. The evidence of the field studies is that rewards, recognition and promotions in science are based primarily on publications of scientific results and not on developing the software which enabled those results. The second author, working as a software project manager in a research establishment, was once told that further promotion would be conditional on publishing six papers. Yet publishing is rarely seen as part of the remit of a software project manager. This emphasis on publications promotes the development of software directed only at producing such publications with little heed given to the wider issues of software engineering (such as testing) discussed above.

We should stress that in the context in which it originated, this model of development with its attendant values and assumptions, works on the whole. The biggest risk is that such a development, with its lack of emphasis on testing, might produce software which does not correctly reflect the known science and thus produces erroneous results, see, for example, Miller, 2006 and Hatton, 1997. This risk is exacerbated by the fact that whereas the scientific results as published are subject to peer scrutiny, the software by which the results are obtained is often not. So, provided the results are consistent with scientists' intuitions, the errors in them arising from errors in the software are not easily identifiable. In the case reported by Miller, the results were credible to biologists who might want to use them but less so to expert crystallographers whose role involves producing such results, Jeffrey 2007.

Despite this risk, because of the immediacy of such development and the deep domain knowledge of the developer, we are convinced that this type of scientific end-user development has contributed greatly, and will continue to contribute greatly, to the advancement of science (Morris and Segal, 2009).

SCIENTIFIC SOFTWARE DEVELOPMENT OUTSIDE THIS CONTEXT

In this section, we consider contexts where scientific software is used to address a variety of scientific problems and/or a variety of users over a period of time. We have identified five such contexts, but allow that there might be more, and that there might be overlap between the five. These contexts are where software developed in the scientific end-user context as described above escapes (uncontrolled) or migrates (controlled) into a wider context; where scientific end-user

developers work on high performance computing systems (HPCS); where scientific end-user developers work in partnership with professional software developers; and where software developed in a research environment is re-engineered to provide tools for practitioners.

The Software Escapes

Here, the software is developed within a scientific end-user development context as described above. It is recognised as being useful, and appropriated (and perhaps modified in an ad-hoc manner) by other scientists in slightly different contexts, and hence escapes (as it were) from the local context for which it was developed into the wider context of the lab and thence into the community. But the software might not be sufficiently robust, reliable, efficient, maintainable or usable, to meet its change of goals.

The Software Migrates

Here, software may be developed originally in the scientific end-user context described above but then made available perhaps via an open source model for the scientific community to scrutinise, modify and extend. The problem here is that there is very unlikely to be the expertise within the scientific community to optimise the software with respect to its change of goals given the broader context of its use, and achieve the necessary robustness, reliability, efficiency, maintainability and usability.

High Performance Computing Systems

A further context of scientific software development is that of the development of high performance computing systems, HPCS, for the purposes of (say) complex simulations. Although the authors themselves have not conducted any field studies of this practice, others have. For example, Easterbrook and Johns, 2009, acting as participant observers, studied the practice of a group of climatologists. Here the context was one in which climatologists worked together over a period of decades maintaining and extending a set of climate models. An interesting insight from this work is that, like the scientific end-user developers described earlier, the climatologists have over the years evolved a software development model which, while appearing very strange to a conventional software engineer, nevertheless completely fits the context in which they work.

Another group of field studies has emerged from the recent DARPA initiative, http://www.highproductivity.org/, which is concerned with improving the productivity of HPC systems (Basili et al., 2008). These field studies are concerned with the development of simulation software in academic contexts and government agencies, and focus on how software engineers might best support such developments. We shall discuss these studies further in a later section of this chapter.

Scientific End-User Developers and Professional Software Developers Working Together

Here, it is recognised that the software is too complex for scientists to develop alone, and hence scientists and software engineers develop the software together in partnership. There are several examples of this in the literature, (De Roure and Goble, 2009, Macaulay et al., 2009, Segal, 2009a, Thew et al., 2009). We shall focus on this partnership in a later section in this chapter.

Re-Engineering Research Software Into Tools for Practitioners

This is a common situation about which, we believe, very little is known. An example is "translational medicine", see for example http://www.translational-medicine.com/, which aims to apply cutting-edge research in the life-sciences to

clinical practice. But the goals which should be met by software intended to support research in the life-sciences (chief among which, we think, is that the software should be flexible so as to enable the exploration of research questions) are different from the goals that should be met by software intended to support clinical practice (chief among which, we think, are correctness and robustness. For example, one shouldn't be told that one's blood pressure is 250/30, nor should the software embedded in a clinical instrument crash).

We believe that the most common development approach for transitioning software from research to tool is that of scientific end-user developers and professional software developers working together on software originally developed by the former. Given the risks associated with such transitioning, we also believe that far more research is needed in this area.

We shall now consider a context about which we, the authors, do know quite a lot: scientist end-user developers working in partnership with professional software developers.

SCIENTIFIC END-USER DEVELOPERS AND PROFESSIONAL SOFTWARE DEVELOPERS WORKING IN PARTNERSHIP

Figure 2 and Figure 3, based on the first author's field studies, illustrate the clashes that can occur when professional software developers work together with scientific end-user developers. Before we explore the nature of these clashes further, we shall discuss the role of scientific end-user developers in such a partnership.

The Essential Role of Scientific End-User Developers in the Partnership

Whereas professional software developers are likely to have strong intuitions as to what is required of (say) payroll or hotel reservation software, they are very unlikely to have any intuition as to what is required from software aimed at (for example) computational chemists or protein crystallographers. It therefore goes almost without saying that, because of the complexity of the scientific domain, it is essential that scientists be effectively involved in scientific software development. However, involving scientists in the development purely as end-users is problematic, as discussed in Segal, 2009a, and Segal and Morris (submitted). For example, scientists are very reluctant to interrupt their scientific endeavours in order to contribute to the development of software that they may never use given the shortness of many research contracts.

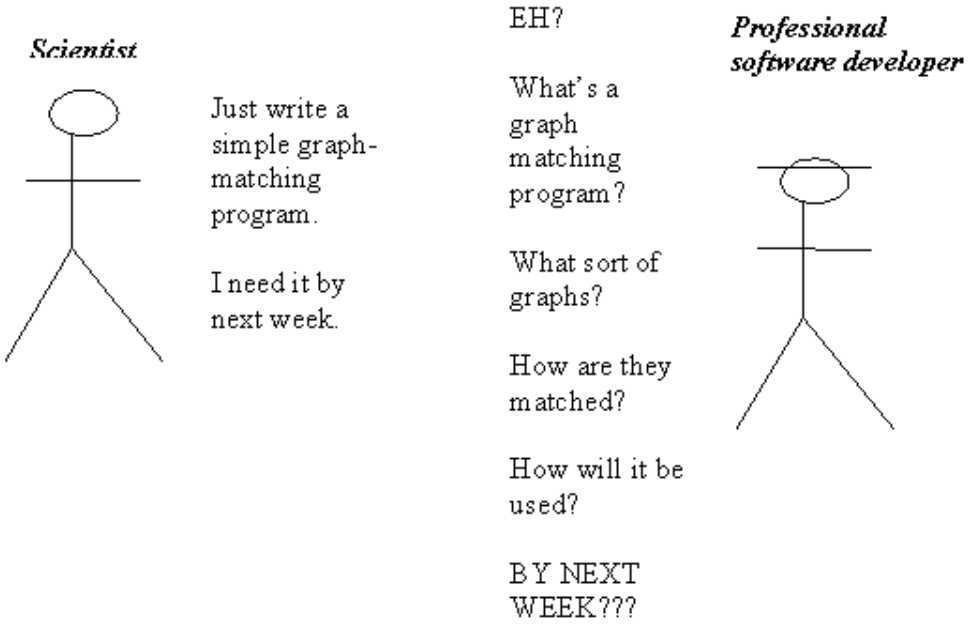
Formally involving scientific end-user developers as members of the development team can go some way towards alleviating this problem. As argued in Segal and Morris (ibid), the scientific end-user developer knows enough both about the scientific domain and about the particular software development to act as an effective bridge between the development team and the potential users, informing the development team of the users' requirements and the users of both the potential and limitations of the software.

Another less immediately obvious role for scientific end-user developers is that of growing the community of users. In these cash-strapped days, growing the user community beyond that for which the software was developed is essential for securing funding for continuing maintenance and development.

Everett Rogers, 2003, in his influential book synthesising current knowledge about technology diffusion, comments:-

'Most individuals evaluate an innovation not on the basis of scientific research by experts but through the subjective evaluations of near peers who have adopted the innovation' (Rogers, 2003, p.36)

Figure 2. An example of a clash between a scientist used to requesting software from scientific end-user developers and a professional software developer, from Segal, 2008, inspired by the field study described in Segal, 2009a.



In other words, scientists are most likely to be persuaded to adopt some software if other scientists in their community convey to them how the software has supported their scientific endeavours. And this is what scientific end-user developers, conversant with both the software and the potential user community, can do very effectively.

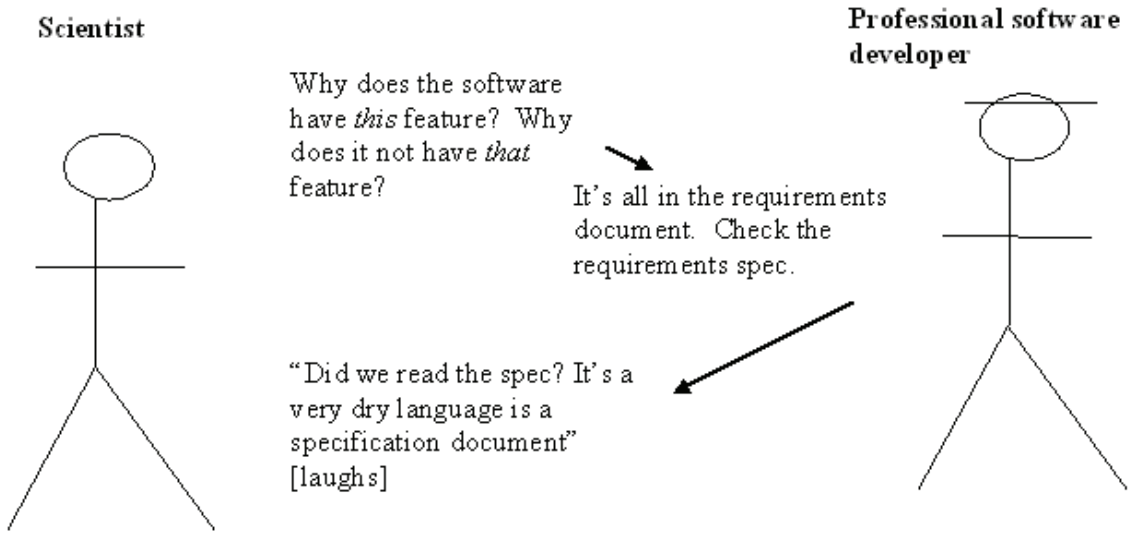
Challenges to the Partnership Posed by the Pervasive Culture of Scientific End-User Development

'When Chris disagreed with us, he wasn't always wrong' (a remark made by a senior scientist with scientific end-user development experience referring to the second author, a professional software developer, after some years of their working together).

To a greater or lesser extent, there are always problems of collaboration and communication when a software development team works together. In this section, drawing heavily on Segal, 2009a, and Segal, 2009b, we shall argue that these problems are greatly exacerbated by the influence of the pervasive culture of scientific end-user development. The challenges we shall discuss include those which impact on the composition of the development team, time estimates for achieving particular development tasks, challenges to the authority of the professional software developers in technical matters, and the length of time that users have to wait for the software.

The ultimate purse-holders of a particular software development for a scientific research community are scientists at the top of their field: they commission the software; they appoint the development team; they keep some sort of check

Figure 3. Another example of a clash between professional software developers and scientists, from Segal, 2008, inspired by the field study described in Segal, 2005.



on the development to ensure that it is delivering what they need. Although such scientists are not likely to be developing their own software currently, it is very likely that they have done so in the early stages of their career and very probable that they are steeped in the pervasive culture of scientific end-user development. The implications of this, as described in an earlier section of this chapter, are that such scientists tend not to value sufficiently the skill and knowledge required to develop software for a community. They tend not to appreciate the importance of the establishment of requirements, a sustainable design, testing, and usability in contexts outside that of scientific end-user development. It follows that they do not appreciate the need to make resources available for these activities.

One consequence of this is on the composition of the development team. We have already described how scientific end-user developers may be appointed to the team not on the basis of their software development expertise but because they need funding. As to the professional software developers on the team, the purse-holder scientists might not recognise that such developers need expertise in aspects of software engineering (such as requirements management and testing) which are not relevant in the pervasive scientific end-user development context and thus might not recognise the need to look for such expertise in potential appointees. Indeed, having no practical experience of such aspects of software engineering themselves, the scientists might not be in a position to judge such expertise. In addition, lack of appreciation of the necessity of this expertise might lead to insufficient funding being made available to lure suitably talented developers away from business.

Another consequence is the tendency of purse-holder scientists to be wildly optimistic about the time that software development tasks take. The experience of the second author is that such scientists habitually estimate the time taken to achieve a particular task as being about a third of his estimation.

Within the software development team, the lack of value ascribed to software development knowledge and skill might lead to the scientific end-user developers being loath to accept the tech-

nical suggestions and leadership of a professional software developer. In the field study described in 2009a, this led to some potential collaborations between scientists and professional software developers becoming completely unviable.

A further problem is caused by the immediate gratification afforded by the pervasive scientific end-user development context where a perceived software need is almost immediately met. In more complex software developments, this is not the case. Before such software is released to the users, requirements have to be established and negotiated; sustainable designs have to be established; and testing has to be done. Any gratification afforded to the users by the delivery of the software is thus deferred. This might lead to frustration both on the part of the users with their experience of almost instant fixes, and on the part of the scientific end-user developers in the development team with their experience of providing almost instant fixes and of being rewarded by 'getting smiles on users' faces' (as said by a scientific end-user developer in one of the first author's field studies).

Addressing These Challenges

Challenges posed by ingrained behaviours, values and assumptions such as those described above can be very difficult to recognise. Such recognition depends on articulating one's own (often deeply hidden) values, assumptions and habits, inferring those of one's collaborator, and seeing where mismatches occur. Segal, 2009a, describes situations where cultural mismatches led to either the collaboration failing completely or to one or both of the parties in the collaboration shifting their values, assumptions or behaviours so as to reach a compromise. It has to be noted, however, that such shifts are very difficult and not achieved without considerable open-mindedness on the part of the collaborators and a considerable amount of pain. However, as the quote at the beginning of this section illustrates, they do happen.

In the next section, we consider how scientific software developers might best be supported by the various current development approaches, methods and techniques, and tools. We should make two important points here. The first is that the choice of 'best' development approach or method or technique or tool depends very heavily on a deep understanding of the context in which the development takes place. As Basili et al., 2008 say:

'To understand why certain software engineering technologies are a poor fit for computational scientists, it is important to first understand their world and the constraints it places on them' [Basili et al., 2008, p.30]

Presumably, lack of such understanding is the reason for the following:-

'...the history of HPC is littered with new technologies that promised to increase scientific productivity but which are no longer available' [ibid., p.32]

The second point is that finding (or constructing) candidates for the 'best' approaches, methods, techniques and tools is currently an active topic of research, see, for example, the DARPA project, http://www.highproductivity.org/, and later discussion in this paper.

DEVELOPMENT APPROACHES, METHODS AND TECHNIQUES, AND TOOLS

One Size Does Not Fit All

The first point we want to make very strongly is that tools, techniques and methods which have been found generally useful within professional software development practice and thus form part of the Software Engineering Body of Knowledge (SWEBOK) are not necessarily useful in

a scientific software development context. This is hardly surprising since such tools, techniques and methods have largely arisen from commercial developments and it is well documented that scientific software development has many aspects which distinguish it from commercial (Carver et al. 2007). For example:

- As befits the essential nature of research, requirements in scientific software development are largely emergent, whereas in commercial developments, most requirements are generally specified a priori.
- In scientific software development, as opposed to in commercial developments, there is often no test oracle, that is, no physical data against which to test the output of the software. For example, consider software which enables complex simulations of a nuclear explosion: physical data from an actual nuclear explosion might be hard to come by.
- Even where experimental data exists, it may be an unrealistic goal to simulate it exactly. Computational scientists are sometimes satisfied with models that match trends in values without matching the exact values. In these cases, what is meant by a "correct" program is unclear.
- The aim of scientific software development is to enable its users to advance science rather than to make a profit, as in commercial developments.

The inappropriate application of approaches, methods, techniques and tools to scientific software developments can lead to great frustration. Those frustrations illustrated in Figure 3 above and Figure 4 below are inspired by the field study described in Segal, 2005. Here, professional software developers worked in partnership with space scientists in order to develop embedded software for an instrument that was going to be sent up into space. The development model was that suggested by the European Space Agency for small software developments. This model was a waterfall-like linear model with discrete phases – specification of requirements; design; implementation; testing – and with a heavy reliance on the role of documents for both communication between the partners and for managing the development. The space scientists, however, were steeped in the traditional culture of scientific end-user development and used to requirements emerging (rather than being specified upfront) and to communication being informal and face-to-face (rather than being dependent on documents). (Despite the difficulties that ensued due to the inappropriateness of this development model, it must be said that the development appears to have been ultimately successful.)

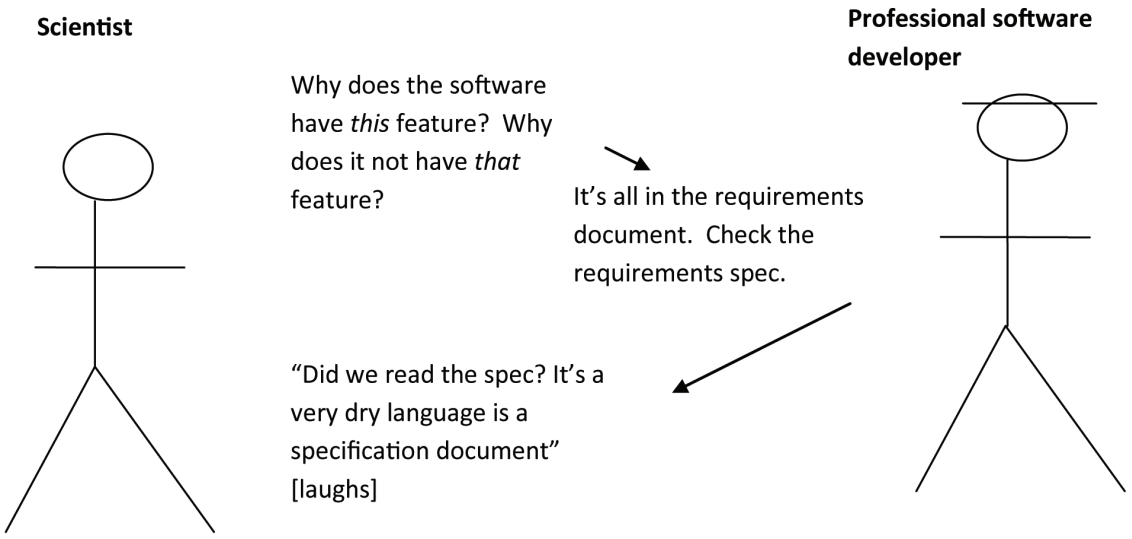
Based on the evidence of her field studies, the first author of this chapter has argued strongly that software engineers should not regard themselves as the sole repository of software development good practice, Singer et al, 2009. Similarly, given the evidence of their field studies of the development of HPC systems, Basili et al., 2008, comment:-

'Several software engineering practices generally considered good ideas in other development environments are quite mismatched to the needs of the HPC community. We found that keys to successful interactions [between software engineers and computational scientists] include a healthy sense of humility on the part of software engineering researchers and the avoidance of assumptions that software engineering expertise applies equally in all contexts' (Basili et al. p.29).

Development Approaches

'I say that I'm adhering to 'agile' methods when all I'm really doing is fighting against formal planning/reporting requirements imposed externally' (an experienced scientific developer, speaking to the second author).

Figure 4. Different attitudes towards documents (Segal, 2005). The quote is taken directly from the field study. The evidence is that neither the requirements nor the requirements specification documents were read by the scientists.



We have discussed above several issues that are relevant to the choice of an overall development approach in the context of scientific software development:

- In typical scientific software developments, requirements emerge rather than being fully specified up-front;
- Discrete phased development models, such as waterfall-type models, are not appropriate in this context;
- The model of software development evolved over years of practice in the scientific end-user developer context is an iterative feedback model with small iterations;
- The effective engagement of the intended users in the development is vital.

All these issues point to some sort of agile and user-centred development approach.

Proponents of agile approaches ascribe to the agile manifesto (http://www.agilemanifesto.org), in which:

- Individuals and interactions are valued over processes and tools.
- Working software is valued over comprehensive documentation.
- Customer collaboration is valued over contract negotiation.
- Responding to change is valued over following a plan.

There are various approaches by which these values can be embedded in software development. The most well-known of these are arguably eXtreme Programming (XP) (Beck, 2000), Scrum which focusses on project management, http://www.scrumalliance.org/, and DSDM (http://www.dsdm.org/).

There has been interest in the application of agile approaches to scientific software development since the early years of the millennium, see, for example, Wood and Kleb, 2002, Bache, 2003, and Segal, 2005. However, in the authors' experience, it is frequently the case that scientific software developers claim erroneously that their

development follows an agile model. Often, all such developers mean is that the development mirrors that of the usual scientific end-user developer model in being an iterative feedback model. The quote at the top of the section illustrates the fact that what people do, and what they say they do, is not always the same thing.

There are some exceptions to this: Ackroyd et al., 2008, describe the writers' experience of applying XP practices to the development of experimental control and data acquisition software, and Pitt-Francis et al., 2008, do the same in the context of computational biology. In both cases, the practices as articulated by Beck, op.cit., had to be tailored to the particular context of use. This is not surprising: development methodologies in general can rarely be used 'out of the box' but have to be tailored to practice, see, for example, Glass, 2002.

As to user-centred design, Macaulay et al., 2009, describe their experience of applying user-centred methods in the context of extending imaging software in the life sciences so that it can be used in a wider context than that for which it was originally developed. However, as we have pointed out earlier, achieving the effective engagement of scientists in a scientific software development can be problematic.

Melding together agile approaches with user centred design (UCD) is not straightforward (McInerney and Maurer, 2005). Agile philosophy is to develop production code as soon as possible in order to obtain quick feedback from the customer; UCD, on the other hand, requires a deep understanding of the users, the activities which the software is intended to support, and the context in which the activities take place. Often, this understanding is obtained by the use of prototypes before any production implementation takes place. How UCD and Agile activities may be integrated effectively is currently the subject of active research.

Maintenance or Development?

'He was spending a lot of time on maintenance so I sent him on a time management course', the line manager of a scientific software developer talking to the second author.

Maintenance does not form part of most scientific end-user development since the (perhaps tacit) assumption here is that the software will be discarded once the particular scientific problem for which it was developed has been addressed. And in general, maintenance is not considered a development activity: the software is developed; the development is finished according to some criteria; the software is then handed over to the users; it then enters a maintenance phase. Given the fact that each delivery of a piece of scientific software might raise some new scientific questions which can only be addressed by an extension or modification to that software, we argue that maintenance and development of scientific software is inextricably linked. De Roure and Goble, 2009, describe this situation in the context of the development of Taverna and MyExperiment as:

'...[leading to] a perpetual beta software development methodology' (De Roure and Goble, p.93)

And Carver et al., 2006, commenting on their field studies of HPC developments, say:

'.. rather than being released and maintained like long-standing IT projects, these projects are under constant development' [Carver et al., 2006, p. 37]

This necessary interlinking of development and maintenance is often not recognised, as illustrated by the quote at the top of this section. Up until recently, it appeared that the UK research councils representing the scientific communities took the same view as most scientific end-user developers, that is, that software could be discarded once

it had addressed a specific scientific question. This view is reflected in the fact that software funding from these councils was made only for a limited period of development, see, for example, Macaulay et al., 2009. Recently, the situation has changed, with scientists becoming very aware of the importance of developing sustainable software, that is, software that has a useful life beyond the original users and the original science it was intended to support. We shall discuss this further in a later section.

Testing

'If my program provided some output, I assumed that it was correct. In hindsight, that is incredibly naïve.' (scientific end-user developer talking to the authors).

Hook and Kelly, 2009, present a nice model (in our opinion) of computational science software development showing that at any point, errors can creep in. They start right at the beginning:

- Measurements of the real world lead to the formation of a theory (but are there errors in the measurements?)
- The formation of a theory/model is based on approximations of the real world measurements (but are the approximations valid?)
- The theory is represented by algorithms (but do the algorithms correctly represent the theory? And do they converge to a solution for all possible inputs?)
- The algorithms are translated into source code (but are there faults in the code?)
- The source code is compiled and the compiled code optimised (but does this lead to inappropriate rounding or concurrency errors?)

This articulation of the myriad opportunities for errors demonstrates the importance of testing. It is not clear that this importance is always recognised by scientific software developers, as illustrated by the quote at the top of this section. We have seen how, in many scientific end-user development contexts, testing is treated relatively lightly. Sanders and Kelly, 2008, on the basis of their interviews of computational scientists, make the interesting observation that when the output of scientific software is not what the scientist expects, then he/she looks for faults in the theory or the algorithms rather than faults in the code. This is consistent with the comments made in Segal, 2008, that the scientists' trust in their software is akin to their trust in their scientific instruments. Software, in common with, for example, telescopes, is presumed to be correct/working properly unless it is absolutely palpably obvious that it is not.

How to test effectively in a scientific context remains an active topic of research. It is clear that testing methods cannot be adopted wholesale from the software industry. For example, in discussing scientific end-user development at the beginning of this chapter, we noted that testing in this context depends heavily, if not entirely, on the judgement of the scientist-developers that the output of the software is reasonable in the scientific context. We also noted that testing is not a separate activity but is entwined with the establishments of requirements. These facts imply that the situation where testing is the province of a testing department separate from the developers, as is common in the software industry, is not appropriate for scientific software development. It is also clear that the effectiveness of testing methods depends on the development context. For example, we have noted that one of the problems of testing scientific software is that there may not be an oracle. However, in HPC settings, algorithms might be prototyped in Matlab or some other high level language and then implemented in Fortran so as to optimise the performance (Sanders and Kelly, 2008). The earlier versions of the code offer a partial solution to the Oracle problem in that the final program should produce the same results.

However, when further modifications are needed, there is the extra cost involved in modifying the oracles as well as the final code.

As to other recent work on scientific software testing, Hook and Kelly, 2009, discuss a procedure for choosing appropriate tests using mutation sensitivity testing. In addition, the second author of this chapter has had positive experiences of swapping codes with other development teams for review, and is also convinced of the importance of unit testing in scientific software development. He is aware of the caveat, however, that older dialects of Fortran encourage programmers to keep a lot of data in a COMMON block, which makes the code less modular and hence less amenable to unit testing.

Tools

'My fixes are not in the release. I will have to make a special build for my users.' (scientific developer working on a distributed development project for a distributed community of users).

In our experience, the sum total of the tools used by many scientific end-user developers are Emacs (a command based text editor), a Fortran compiler, and Make. In contrast, many, if not most, professional software developers make use of integrated development environments (IDEs), which comprise a set of integrated software development tools, including, normally, a source code editor, a compiler/interpreter, automated build tools and a debugger. Other tools, such as a version control system and support for object-orientation software development, may also be included.

With respect to the choice of programming languages, Basili et al., 2008, note that C and Fortran dominate the HPC community. This is consistent with our own experiences of computational scientists implementing complex algorithms though not of scientists whose focus is on managing large data sets, such as structural biologists. One reason for this choice of language is the longevity of much computational code. Many computational scientists work on codes which were originally developed many years ago in Fortran. Although the codes might be improved by being re-implemented in other languages, Sanders and Kelly, 2008, point out the risk inherent in doing so and the feeling of the computational scientists that "if it aint broke, don't fix it". Or as Sanders and Kelly put it:-

'Scientists generally want to do science, not write software, and certainly not introduce risk by changing software that worked.' (Sanders and Kelly, 2008, p.24)

Another reason for choosing Fortran is that users might have to modify code to fit their own contexts (Carver et al. 2007). The code thus has to be written in a language that it can be assumed that users will know, and Fortran fits the bill, having arguably become the lingua franca of computational scientists.

There are several reasons why IDEs in general fail to support scientific software development. First and foremost among these is that few IDEs claim to support Fortran, and where they do, the support is basic compared to their support of, for example, Java. Fortran has a long history, and the changes have been backwards compatible, so it is now a very large language, the design of which began before the modern understanding of parsers. So sophisticated support for editing Fortran is hard to do. For example, alias recognition is hard. Older program suites make heavy use of COMMON blocks, and over time may come to map them inconsistently. In addition, older programs made up for the lack of storage management in the language by allocating a large array, and mapping it in a way that is not type safe.

A further reason is that the tools in IDEs generally do not have special features to support floating point calculations. For example, test cases for floating point applications rarely test for equality but rather use an error bar, and this is not generally supported in IDEs. In addition, Basili

et al., 2008, point out that many HPC systems are shared, and HPC developers have to submit batch jobs. For these developers, an IDE which does not support the submission of jobs to batch queues or debugging on parallel machines, is going to be useless. Finally, the field studies of Carver et al., 2007, reveal that HPC developers prefer what they perceive to be the flexibility of the UNIX command line to the rigidity afforded by an IDE.

There is somewhat of a chicken-and-egg situation here: scientific software developers are reluctant to use commercial IDEs and other commercial software development tools because they tend not to support scientific software development, and commercial tool makers are reluctant to put effort into making tools more supportive of scientific software development since the potential market is small. It seems clear that any improvement in development and maintenance tools aimed specifically at scientific software development is going to have to emerge from the community of scientific software developers themselves.

Notwithstanding the comments made above, there are certain generic software development tools which are very useful in supporting the maintenance/iterative development of scientific software developments. These include issue trackers, used to keep track of new or changing requirements, and version control software/code repositories, such as CVS or Subversion. The omission of certain fixes in the release referred to in the quote at the beginning of this section would not have happened if the version control system had been used properly. Wilson, 2006, was shocked to find that scientific end-user developers shared their source code files with their collaborators by means of email rather than by means of repositories. He ascribed this behaviour to the scientists' lack of knowledge of such repositories. But there is another plausible reason: it may be that the collaboration entails evolving a common code base in more than one direction to solve more than one problem, rather than evolving it in a more linear fashion as supported by most version control systems. However, some modern systems including Git (http://git-scm.com/) are specifically designed for this sort of collaboration, and so are a better match to the practice of scientific programming collaborations.

The essential problem according to Wilson (ibid) is that scientific end-user developers do not know about the tools – or indeed the software engineering techniques – which might be useful to them. Wilson has attempted to address this problem by means of a web-site, http://software-carpentry.org/.

FUTURE RESEARCH DIRECTIONS

In our opinion, there are several important directions for future research:-

- There should be a better understanding of the different contexts and cultures of scientific software development. The first author of this paper has conducted several in-depth field studies of the context of scientific end-user development as described in an earlier section of this chapter, and Basili, Carver and colleagues have considered the HPC context of running simulations. We are also impressed by Easterbrook and Johns' field study of climatologists. However, there are gaps in our collective understanding. For example, we are not aware of any similar field studies of scientific software developments which follow an Open Source model or of those in which research software is re-engineered into tools for practitioners, as in the translational medicine context for example. If you know of any, we would be very grateful if you could pass the information on to us.
- This paper focuses on some challenges in scientific software developments. But it would also be useful to investigate successful scientific software developments in

an attempt to articulate the factors which make them successful. There are a few such in the literature (for example, De Roure et al., 2009, and Macaulay et al., 2009) but it would be helpful to have more.

- When working in partnership to produce software for a community of scientists, neither scientific end-user developers nor professional software developers have strong intuitions as to which approaches, methods, techniques or tools will be useful. In the case of the scientific end-user developers, this is because they tend to be steeped in a development culture which has evolved within a very restricted context. In the case of the professional software developers, it is because their experience of commercial development, garnered either directly by practice or indirectly by reading standard texts on software engineering, might well not transfer to scientific software development contexts. The intuitions as to which approaches, methods, techniques or tools are best (or might be best adapted) can only come from practical experience. For example, professional software developers who habitually work within a particular scientific domain build up an intuitive understanding of what methods etcetera are useful in that domain. But such intuition is mostly tacit and rarely articulated and shared.

We believe that it is important to establish a community of practice whereby scientific software developers from a variety of backgrounds and working in a variety of scientific domains, can come together so as to share their experiences. If you believe this too, and would like to be part of building up such a community, then please contact the authors.

The knowledge shared by such a community could also inform the content of efforts to inform scientists of software engineering such as, for example, by the software carpentry site referred to above. It is widely agreed that many standard university software engineering courses are too free of context to be useful to scientists (Kelly, 2007).

- We have alluded in a previous section to a change in the attitude of the scientific community (at least in the UK) to the sustainability of scientific software. Before this change, scientific software development in research communities was funded only to the extent that it addressed a specific scientific problem, with the implication that the software would be discarded once it had produced the requisite scientific results. There is now some recognition of the importance of the sustainability of scientific software so that it is useful in contexts beyond that in which it was originally developed. The UK research council, the EPSRC, in 2010 made a grant of £4.3 millions to establish a software sustainability institute, see http://software.ac.uk/. The aim of this institute is to establish partnerships between domain-specific software engineers and scientists in which either both partners work directly together or the former provide a consultancy role. Field studies of these partnerships should provide essential information as to how they can best be made to work.

CONCLUSION

In our previous discussion, we have stressed the following point:-

One size does not fit all. The ‘best’ development approach to take, method or technique to use, tool to adopt, depends on the context.

We have pointed to several distinct contexts, among which are scientific end-user development, the High Performance Computing context, and

collaborations between professional software developers and scientific end-user developers. Each of these involves particular opportunities and challenges which we have discussed. It is clear, however, that there is still a long way to go in identifying the different contexts of scientific software development, and then either matching each context with suitable approaches, methods, techniques and tools, or constructing suitable methods, techniques and tools where no such exist.

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 Software*, *25*(4), 44–51. doi:10.1109/MS.2008.93

Bache, E. (2003). *Building software for scientists – A report about incremental adoption of XP*. Poster presented at XP2003, Genoa, Italy.

Basili, V. R., Carver, J., Cruzes, D., Hochstein, L., Hollingsworth, J. K., Shull, F., & Zelkowitz, M. V. (2008). Understanding the high performance computing community: A software engineers' perspective. *IEEE Software*, *25*(4), 29–36. doi:10.1109/MS.2008.103

Beck, K. (2000). *Extreme programming explained: Embrace change*. Boston, MA: Addison-Wesley.

Carver, J., Kendall, R., Squires, S., & Post, D. (2007). Software development environments for scientific and engineering software: A series of case studies. *Proceedings of the 29th International Conference on Software Engineering* (pp. 550-559). Minneapolis, MN. May 23-25, 2007.

Carver, J. C., Hochstein, L. M., Kendall R. P., Nakamura, T., Zelkowitz, M. V., Basili, V. R., & Post, D. E. (2006, November). Observations about software development for high end computing. *CT Watch Quarterly*, 33-38.

De Roure, D., & Goble, C. (2009). Software design for empowering scientists. *IEEE Software*, *26*(1), 88–95. doi:10.1109/MS.2009.22

Easterbrook, S. M., & Johns, T. C. (2009). Engineering the software for understanding climate change. *Computing in Science & Engineering*, *11*(6), 65–74. doi:10.1109/MCSE.2009.193

Glass, R. (2002). Searching for the Holy Grail of software engineering. *Communications of the ACM*, *45*(5), 15–16. doi:10.1145/506218.506231

Hatton, L. (1997). The T experiments: Errors in scientific software. *IEEE Computational Science & Engineering*, *4*(2), 27–38. doi:10.1109/99.609829

Hook, D., & Kelly, D. (2009). Mutation sensitivity testing. *Computing in Science & Engineering*, *11*(6), 40–47. doi:10.1109/MCSE.2009.200

Jeffrey, P. (2007). *ABC Transporter debacle*. Retrieved April 18, 2010, from http://xray0.princeton.edu/ ~phil/Facility/Guides/ ABCtransporter.html

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

Macaulay, C., Sloan, D., Jian, X., Forbes, P., Loynton, S., Swedlow, J. R., & Gregor, P. (2009). Usability and user-centered design in scientific development. *IEEE Software*, *26*(1), 96–102. doi:10.1109/MS.2009.27

McInerney, P., & Maurer, F. (2005, November-December). UCD in agile projects: Dream team or odd couple? *Interaction*, 19–23. doi:10.1145/1096554.1096556

Miller, G. (2006). A scientist's nightmare: Software problem leads to five retractions. *Science*, *314*, 1856–1857. doi:10.1126/science.314.5807.1856

Morris, C., & Segal, J. (2009). *Some challenges facing scientific software developers: The case of molecular biology*. 5th International IEEE Conference on E-Science.

Pitt-Francis, J., et al. (2008). Chaste: Using agile programming techniques to develop computational biology software. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 366*(1878), 3111-3136.

Rogers, E. M. (2003). *Diffusion of innovations* (5th ed.). Free Press, Simon and Schuster.

Sanders, R., & Kelly, D. (2008). Scientific software: Where's the risk and how do scientists deal with it? *IEEE Software*, *25*(4), 21–28. doi:10.1109/MS.2008.84

Segal, J. (2001). Organisational learning and software process improvement: A case study. In K.-D. Althoff, R. L. Feldmann, & W. Muller (Eds.), *Advances in learning software organizations, Lecture Notes in Computer Science, 2176*, 68-82. Springer.

Segal, J. (2005). When software engineers met research scientists: A case study. *Empirical Software Engineering*, *10*(4), 517–536. doi:10.1007/s10664-005-3865-y

Segal, J. (2007). *Some problems of professional end user developers*. VLHCC, IEEE Symposium on Visual Languages and Human-Centric Computing, (pp. 111-118).

Segal, J. (2008). Scientists and software engineers: A tale of two cultures. *Proceedings of the Psychology of Programming Interest Group, PPIG 08*. Retrieved from http://www.ppig.org/papers/20th-segal.pdf

Segal, J. (2009a). Software development cultures and cooperation problems: A field study of the early stages of development of software for a scientific community. *Computer Supported Cooperative Work*, *18*(5/6), 581–606. doi:10.1007/s10606-009-9096-9

Segal, J. (2009b). *Some challenges facing software engineers developing software for scientists*. 2nd International Software Engineering for Computational Scientists and Engineers Workshop (SECSE '09), ICSE 2009 Workshop, (pp. 9-14). doi: 10.1109/SECSE.2009.5069156

Segal, J., & Morris, C. (2008). Developing scientific software. *IEEE Software*, *25*(4), 18–20. doi:10.1109/MS.2008.85

Segal, J., & Morris, C. (in press). Scientific end-user developers and barriers to user/customer engagement. *Journal of Organizational and End User Computing*.

Singer, J., Vigder, M., Segal, J., & Clarke, S. (2009). Point/counterpoint. *IEEE Software*, *26*(5), 54–56. doi:10.1109/MS.2009.135

Thew, S., Sutcliffe, A., & Procter, R., De Bruijn, McNaught, J., Venters, C., & Buchan I. (2009). Requirements engineering for e-science: Experiences in epidemiology. *IEEE Software*, *26*(1), 80–87. doi:10.1109/MS.2009.19

Wilson, G. V. (2006). Where's the real bottleneck in scientific computing? *American Scientist*, *94*(1), 5–6.

Wood, W. A., & Kleb, W. L. (2002). Extreme programming in a research environment. In D. Wells & L. Williams (Eds.). *XP/ Agile Universe 2002, Springer LNCS 2418*, 89-99.

ADDITIONAL READING

In our introduction, we noted that one chapter cannot possibly say all there is to say about the challenges facing the developers of software for a scienific community, and mentioned a body of CSCW literature regarding the social and political influences on community software development and adoption. We did not refer to this literature above because this chapter is grounded in our

field studies and experience, and the challenges we discuss in particular are those arising from the culture of scientific end-user development. Nevertheless, we should like to highly recommend Grudin, 1994, to the interested reader. Grudin discusses the following challenges (among others) relating to community software:

Many of the references above were written so as to be accessible to practitioners and not merely to enhance the academic reputation of the authors. This is especially true of articles in the journals IEEE Software and Computing in Science and Engineering. We also recommend that you look at Greg Wilson's software carpentry site, http://software-carpentry.org/, which at the time of writing (2010) is being redeveloped in the light of an increased understanding of the matching of various methods, techniques and tools to the various scientific development contexts.

- Cost and benefit might not apply equally among members of the group;
- An individual's best interests might not match with that of the group;
- Software cannot take cognisance of tacit group knowledge (for example, of the different strengths and weaknesses of members of the group);
- Successful practice involves improvisation and software supporting processes must take heed of this;
- Individual work often forms the backbone of community endeavour and community software should recognise this.

Grudin, J. (1994). Groupware and social dynamics: eight challenges for developers. Communications of the ACM, 37(1), 92–105. doi:10.1145/175222.175230

In addition, there is currently much interest in sharing and deploying scientific knowledge via ontology building. Although many of the related publications focus on tools for building ontologies, there are several which highlight how social and political challenges, including those articulated by Grudin, are played out in this context. Journals like IJHCS (the International Journal of Human Computer Studies) and conference series like those of e-social science are good places to look for these.

KEY TERMS AND DEFINITIONS

Culture: by the term 'culture', we mean here the normal behaviour, values and assumptions that distinguish one group of people from another.

High Performance Computing: This term is normally taken to describe computing systems with the substantial processing power needed to (for example) run complex scientific simulations.

Scientific End-User Developers: Scientists who develop software in order to address their own scientific problems.

Software Engineering: in this chapter, this refers to all the technical aspects of developing code including design and testing.

Software Engineer/Professional Software Developer: we have tried to avoid the use of the term 'software engineer' because of the various arguments currently raging as to what the term involves. Instead, we have used 'professional software developer'. By this, we mean someone whose focus is on the software (rather than on the science) and who is aware that software development involves rather more than mere coding.

Techniques: practices which support different aspects of software development such as testing.

Tools: software tools which support development.