Open Research Online

The Open University's repository of research publications
and other research outputs

Scientists and software engineers: A tale of two cultures

Conference or Workshop Item

How to cite:

Segal, Judith (2008). Scientists and software engineers: A tale of two cultures. In: PPIG 2008: Proceedings of the 20th Annual Meeting of the Pschology of Programming Interest Group (Buckley, Jim; Rooksby, John and Bednarik, Roman eds.), Lancaster University, Lancaster, UK.

For guidance on citations see FAQs.

© [not recorded]

Version: [not recorded]

Link(s) to article on publisher's website:
http://www.cs.st-andrews.ac.uk/ jr/ppig08/index.html

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online's data policy on reuse of materials please consult the policies page.

oro.open.ac.uk

Scientists and software engineers: a tale of two cultures

Judith Segal

Department of Computing
The Open University
j.a.segal@open.ac.uk

Keywords: POP-I C. Ill-defined problems, POP-II A end-users, POP –II C working practices, POP-V B, case studies

Abstract

The two cultures of the title are those observed in my field studies: the culture of scientists (financial mathematicians, earth and planetary scientists, and molecular biologists) developing their own software, and the culture of software engineers developing scientific software. In this paper, I shall describe some problems arising when scientists and software engineers come together to develop scientific software and discuss how these problems may be ascribed to their two different cultures.

1. Introduction

One major difference between most commercial software development and scientific software development lies in the complexity of the domain. Most software engineers have some intuition as to what is required from (for example) a hotel booking system or a banking system or a payroll system; few have any intuition as to what is required from (for example) a stochastic modelling system or a quantum chemistry system or a protein crystallography system. The implication of this is that the relevant scientists must be deeply engaged in the system development, either developing it themselves in their role of 'professional end-user developers' (see below), or in providing (and explaining) requirements, giving feedback and performing user-acceptance tests.

Before I go any further, I shall define my use of the terms 'professional end-user developers' and 'culture'. The term 'professional end-user developers' (Segal, 2007) refers to people such as scientists and engineers working in highly technical, knowledge-rich domains who develop software in order to further their own professional goals and/or those of their close colleagues. Like other end-user developers, these people do not regard themselves primarily as software developers and have little or no education or training in software engineering. Unlike most other end-user developers, however, coding per se presents them with few problems as they are used to formal languages. Turning to the term 'culture', the concept of culture has many aspects. In this paper, I take the term to mean the set of values and customary behaviours of an identifiable group of people, professional end-user developers and software engineers in this case.

This paper draws on field studies I have undertaken with financial mathematicians, earth and planetary scientists, space scientists and molecular biologists. In section 2, I shall describe the culture within which I have observed scientists developing software for their own use and/or for the use of their close colleagues, and present a model of how this software is developed, a professional end-user development model. In section 3, I shall describe two sets of problems which I observed when software engineers worked closely with scientists in order to develop scientific software, and which arise from a cultural mismatch. In the first case, software engineers tried to impose a traditional software engineering culture on scientists. In the second, scientists expected software engineers to ascribe to the culture of professional end-user development. In section 4, I shall discuss the limitations of my field studies. Although my field studies have explored quite a variety of contexts, they are in no way comprehensive. I discuss whether other software development models would fit better with scientific software development than the traditional phased waterfall model that I

observed, and also whether the characteristics of the culture described in section 2 are common across all contexts of scientists developing scientific software. Section 5 consists of a summary and conclusions.

2. The culture of professional end-user development

2.1. Values

As described in Segal, 2007, the most salient characteristic of the culture I saw in my field studies of financial mathematicians and earth and planetary scientists, was the low value ascribed to software development knowledge and skill compared with knowledge of the mathematical/scientific domain. People spoke in terms of 'everybody' knowing how to develop software; of software development knowledge being merely part of the armoury of the average scientist; of the belief that a piece of software was something that could be dashed off during a lunch hour.

In these two contexts of financial mathematics and earth and planetary science, software development is something one practises at the beginning of one's professional career. As one ascends the career ladder (by passing one's professional exams or by publishing enough scientific papers), then one leaves software development behind to be done by one's juniors. The situation in which this is not the case, that is, in which a professional end-user developer – or, indeed, a software engineer working within a professional end-user organisation – develops software full-time on a long-term basis, does not appear to differ significantly in the low value afforded by the organisation to software development knowledge and skill. My current field study of molecular biologists includes several interviews with a professional end-user developer whose skill in software development had been recognised during his PhD work in molecular biology and who is now working full time developing and maintaining software for his lab. Although this software is the absolute sine qua non of the lab – without the software, there would be no lab – this man feels that there is no way someone in his position could ever become head of such a lab. His belief is that such a position would always go to a traditional bench biologist, despite the fact that traditional bench biology now plays a relatively small part in the work of the lab. I also talked with a software engineer who works for a central government research facility with the express aim of providing software support for the UK scientific community. The management of this facility consists of professional end-user developers, people who primarily consider themselves to be scientists. The developer constantly finds his promotion blocked because he has not published enough scientific papers – this despite the fact that software development, and not writing scientific papers, is his remit. The developer feels that the facility's management do not understand, and cannot judge, professional software development (as opposed to professional end-user development). He feels that the concerns and quality goals of the former are quite different from those of the latter. This is a point to which I shall return briefly in section 3.2 below.

The low value ascribed to software development knowledge and skills no doubt contributes to the difficulties that professional end-user developers have in acquiring such knowledge and skills as described in Segal, 2007, despite the fact that it is assumed that 'everybody' knows what to do, as above. Professional end-user developers have rarely had any formal software engineering education at university. However, the same is true of many software engineers, and in fact, Kelly, 2007, notes that university software engineering courses are frequently unpopular with potential professional end-user developers since they are often taught in a way which is independent of the domain and the students are unable to make links between the software engineering as taught and their chosen science.

What is more important than formal education, I think, are the learning opportunities afforded by the community of practice. My interviews indicate that software engineers acquire their knowledge and skills through a variety of means, all dependent on their being part of a community (or network) of practice of software developers. These means include working with a variety of other developers on a variety of projects and thus sharing knowledge on an informal basis, reading books and studying internet tutorials etcetera as recommended by colleagues, and going to technical conferences and short courses, the existence of which is made known through the network of practice. For the professional end-user developer observed in my field studies, this community of software development

practitioners does not exist. The primary community of practice to which the professional end-user developer belongs is that of the application domain, the science. Professional end-user developers often work on their own or in very small groups and so rarely have the opportunity to share knowledge informally. In at least one of my field studies, the perception that software development knowledge is trivial and known to everyone, meant that the management was loath to spend money on resources, such as courses, designed to improve such knowledge.

2.2. Behaviours: a model of professional end-user development

Figure 1 is, I suggest, the standard model of professional end-user development. I found it practised by all the professional end-user developers in my field studies. And a casual conversation on a train with a computational linguist elicited the information that he recognised it as the model he used in writing his latest substantial program in Python in order to analyse the dialogues of Plato.

In this model, the developer begins with just a vague idea of what is needed. S/he quickly develops a piece of software, and then sits back and reflects on the question of whether the software does what s/he wants and how it might be extended or modified, drawing in his/her colleagues if available. The developer goes round the development/evaluation loop several times until s/he decides s/he has got a suitable release. S/he then does testing of a very cursory nature. For example, a few items of data similar to the data that will be input when the software is released, is entered into the system, and the output is checked to see that it looks broadly correct – or at least not broadly incorrect. The software is then ready to become accepted as a tool for the scientific endeavour.

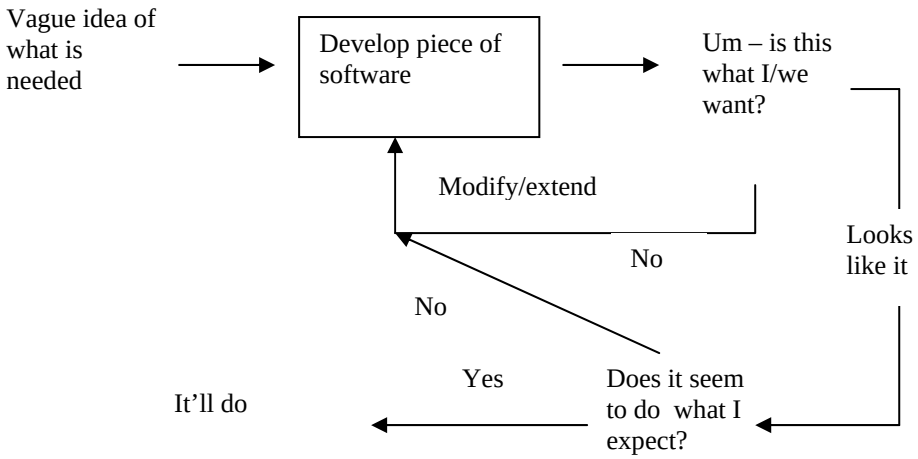

Figure 1. A model of professional end-user development (from Segal and Morris, 2008)

The salient characteristics of this model are, firstly, the lack of an upfront requirements model; secondly, the intertwining of evaluation and the identification of emergent requirements ('Is this what I/we want?'), and finally, the cursory nature of the final testing. This model would not be taught in any software engineering course – and yet, to judge by its pervasiveness, it works. But only in a very particular context, as I shall now discuss.

Starting off with a vague idea of what is needed depends on the developer having sufficient knowledge of the domain.

The reliance on feedback depends on the developer being embedded in the user community. Many of us will have experienced problems in getting potential users to engage in a software development in

order to give informed and reasoned feedback. Getting such feedback is much easier if you, as the developer, are just asking your mate at the next desk/bench 'Have a look at this. What do you think?'

I have several suggestions as to why testing is so cursory. The first is to do with the low value placed on software as opposed to that placed on the science: the software is valued only insofar as it progresses the science. I suggest that scientists regard the software in the same light as any other instrument for enabling their scientific endeavours. It is argued by many philosophers and historians of science, see for example, Chalmers, 1982, that scientists assume that their instruments work unless confronted by absolutely incontrovertible evidence. Perhaps this assumption also holds for their software: the innate quality of the software is not questioned unless it becomes clear that the software is not supporting the science. The second is to do with the developer being embedded in the user community. If a scientist does find faults in a piece of professional end-user developed software, then the developer is readily at hand (either the scientist him/herself or a close colleague) to make amendments. The third suggestion is to do with the nature of scientific software and concerns the great difficulty of validating software (such as scientific software) in which the domain is only poorly understood and, in fact, the aim of the software is to advance the understanding of the domain, see, for example, Carver et al. 2007. In this case, there is simply no way in which the scientists can know whether the output from the software is correct: s/he just has to rely on her/his gut instincts that the output is not absolutely wrong.

3. Clashing cultures: some problems that arise when scientists and software engineers work together

In this section, I shall describe, firstly, a situation in which software engineers tried to impose a traditional software engineering culture on scientists, and secondly, a situation in which a scientist assumed software engineers were working within a professional end-user development culture, as described above.

3.1. Why can't scientists be more like software engineers?

The discussion in this section is based on the field study described in Segal, 2005. The context of the field study is thus: the scientists were familiar with writing their own software in the lab to drive instruments such as spectrometers and to analyse the data coming from the instrument. They were now about to embark on a very risky endeavour: rather than pick up space material and bring it back to earth to be analysed in the lab, they were going to send an instrument up into space to do the analysis in situ and relay the results back to earth. They brought in software engineers to write a library of components which they could use to drive the instrument, and themselves had a model of the instrument in the lab which they could use to reify their requirements.

The software engineers followed a waterfall-type phased model of software development as recommended by the European Space Agency. The scientists in their lab followed the model of professional end-user development as described in section 2.2 above. The first problem lay with requirements and is illustrated in Figure 2. The software engineers needed an upfront requirements document; the scientists expected most of the requirements to emerge.

Other problems stemmed from the scientists being used to working within the lab, where informal face-to-face communication flourished. They were thus not used to either writing or reading formal project documents, such as requirement documents, and were thus not aware of the contents of such documents, and hence did not fully know which requirements had – or had not - been implemented. Their user acceptance testing was as cursory as that described in section 2.2.

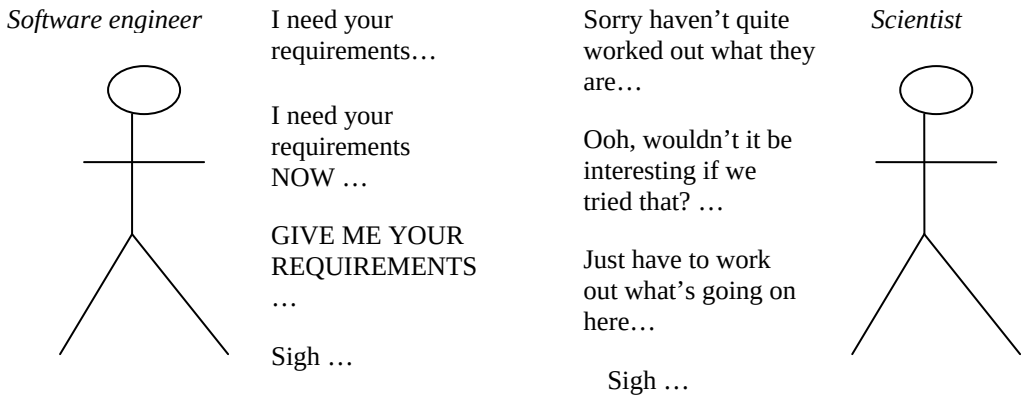

Figure 2: Why can't a scientist be more like a software engineer? upfront versus emergent requirements (from Segal and Morris, 2008)

3.2. Why can't software engineers be more like scientists?

In this section, I describe, in somewhat simplified terms, an aspect of a hitherto unpublished field study in which molecular biologists were employing software engineers to write some community software. The molecular biologists had all been at one time professional end-user developers, and some were still developing their own software. However, the community for which the software is intended is somewhat diverse and the software itself is considerably bigger than any that a professional end-user developer would tackle, and hence it was felt necessary for the scientists to employ software engineers.

The first problem again lies with requirements. The molecular biologist heading the project said as he handed over a list of features to the project manager of the software engineers: 'We know exactly what the requirements are and here is a list of them.' Of course, the features were at too high a level for the software engineers to begin to implement. Figure 3 illustrates a hypothetical (but very realistic) instance.

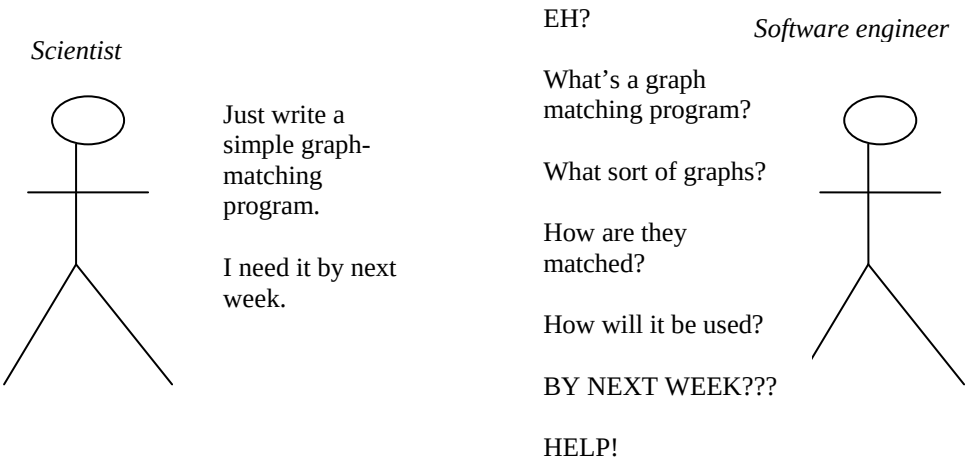

Figure 3: Why can't a software engineer be more like a scientist?

The scientist's injunction to write a piece of software with a particular piece of scientific functionality is perfectly reasonable *provided* that the developer is a professional end-user developer. In this case, the developer knows the domain, has some intuition as to how a simple graph-matching program might work and might be used; can develop a first prototype and ask his/her peers for feedback, and

generally follow the professional end-user development model. The poor software engineer, however, with no – or at best, weak – understanding of the domain, has great difficulty in proceeding.

Figure 3 illustrates another clash between the expectations of professional end-user developers and software engineers. This is to do with the time that software development takes, which in turn depends on the different values and behaviours espoused by professional end-user developers and software engineers. In general, professional end-user development takes less time. The software project manager in this field study told me that, as a rough rule of thumb, his team took three times longer to produce a piece of software than the scientists expected. There are several potential reasons for this. The first concerns requirements. The establishment of requirements in professional end-user development, as illustrated in Figure 1, is absolutely integrated with the software development. In addition, the context in which professional end-user development flourishes, as described in section 2, is one in which the developer is a faithful representative of the user group, which implies that the user group is homogenous and not split into subgroups with diverse goals and behaviours. For the software engineer developing software for a diverse community (as in this field study), establishing requirements is a time consuming and difficult task. Potential users have to be persuaded to tear themselves away from their current endeavours and engage with the development of a system which they may well never use in its mature state (such potential users are often on short term research contracts). The software engineers have to ensure that the diversity of the user community is properly represented; that clashes between different branches of the community are resolved, and so on. The second concerns those issues which reflect the values of software engineering as opposed to those of professional end-user development. Foremost among these is testing. In 2.2, I discussed the fact that the cursory testing which is a feature of professional end-user development may be due to the fact that the emphasis is on the science which the software is intended to support and not on the software per se. Software engineers, on the other hand, should ideally identify the quality goals for any piece of software, and allocate testing time in accordance with these goals. For example, a quality goal might be robustness, in which case much time must be spent testing that the software does not fall over given a variety of inputs. Other issues which do not usually impact on professional end-user development include portability and maintainability. There might also be security issues when a diverse user community is involved, for example, issues of data access when users from different branches of the user community use the same system.

4. The limitations of my field studies

I have undertaken a variety of field studies (Segal, 2007) in quite a variety of settings. The application domains have been in financial mathematics, earth and planetary scientists, and molecular biologists; the scientists have developed their software either in partnership with software engineers or on their own; the software developed has included software to drive instruments, model financial markets, and to store, analyse and support the interpretation of data. Across this variety, I have found a number of commonalities, such as the low value ascribed to software development knowledge and skill compared with domain knowledge and skills, and the ubiquity of the professional end-user development model.

My field studies are in no way comprehensive however. For example, the software engineers in my studies never adopted agile methodologies, which, relying as they do on iterative feedback loops and face-to-face communication (see http://agilemanifesto.org/), might appear to offer more to scientific software development than the more traditional, phased, waterfall-type methods. There are experience reports in the literature of software engineers successfully engaging scientists in agile development, see, for example, Bache, 2003, and Kane, 2003. However, I am not aware of any objective field study data in this area, and, given my recent experience of co-editing a special issue of IEEE Software on developing scientific software, I am wondering whether, when scientists refer to themselves as following an agile methodology, they are not just following the iterative feedback model of Figure 1.

In addition, my field studies did not cover high performance computing systems (HPCS), that is, systems in which many processors act in parallel. Such systems are commonly used in science to

simulate natural phenomena which are too big or too small or too dangerous or too complex to be investigated in vivo. There has been a lot of interest in researching HPCS in the USA recently, spurred by a large, multi-phased, ongoing DARPA project (see www.highproductivity.org). This project was instigated in response to a concern that scientific productivity using HPCS systems did not seem to improve commensurate with the rate at which the capabilities of the hardware improved. The aim of the project is thus to improve scientific productivity by a factor of ten, by dint of improvements in both software and hardware. The exact nature of the concept of 'scientific productivity' appears not to have been completely explicated, however.

The DARPA project has generated many field studies of scientists being deeply involved in the development of software simulations, see, for example, Carver et al., 2007, Basili et al., 2008. The contexts in which HPCS are used by scientists vary greatly, and Basili et al., 2008, allow that their field studies are not comprehensive – and also acknowledge that even within their field studies, they found a great deal of variation. However, their field studies demonstrate similarities with mine. For example, they found that the science, rather than the software, was paramount, and they found the same reliance on emergent requirements and difficulties with testing as did I.

However, some of their findings were different from mine. For example, the relatively low status of software development that I found universal, was not always found in their case studies. I was told that sometimes physicists who developed HPCS thought of themselves as forming an elite among physicists, and, moreover, their opinion of themselves was based not on what they could bring to physics but rather on their adeptness in employing programmers' tricks to support parallel processing. This is totally counter to my findings. I was given a possible explanation for this phenomenon, which is that these physicists regard physics as having essentially three branches of equal worth: theoretical, experimental, and in silico (that is, software simulations). This does not appear to be the case in my field studies where (except in the case of the financial mathematicians) software is seen as a supporting tool for scientific enquiry rather than as providing a model of science which can be queried directly. In the case of the financial mathematicians in my field studies who were developing models, software development tends to be undertaken by students (in the professional sense, that is, people who had not yet passed a long series of professional exams), and this may account for the lack of value afforded to it. Given the importance of simulations in science, it is clear that HPCS represent a domain of scientific software development into which I am going to have to look more closely.

5. Summary and conclusions

My field studies have identified two characteristics of a culture of scientific professional end-user development: the low value given to software development knowledge and skill compared to domain knowledge, and a model of professional end-user development. Judged by its pervasiveness, this latter is very successful, though only in a particular context. I have identified the characteristics of this context as being the following:

- The developer is embedded in the user community.
- The user community is cohesive.
- The requirements are not fully established at the outset.
- The value of the software lies in the extent to which it progresses science..

I have reported the clashes which occurred when software engineers tried to impose their culture of traditional software development onto scientists and vice versa.

My field studies, of necessity, illustrate only some of the variety of scientific software development. Other field studies have investigated the situation in which high performance computing systems are developed for simulation purposes. These studies have confirmed my findings of the primacy of science over software, the importance of emergent requirements and the difficulties of testing.

This research is important because I take it as a given that software engineers cannot hope to provide effective tools, technologies and methods for improving scientific software development without first

understanding the cultural context, the values and customary behaviours, in which this development takes place. As I describe in section 3, lack of understanding of this context can lead to major problems. There is still much work to be done in this area. A complete research agenda would, I argue, encompass the following:

1. The identification of the salient dimensions against which contexts of scientific computing might be characterised. Such dimensions might include the following specific to scientific computing: whether the scientists are developing the software on their own, and if not, the degree to which software engineers are involved, and the value ascribed to software development in the user community. Other dimensions might include: whether the user community is homogenous or diverse, the size of the development team, the longevity of the code, etcetera.
2. The identification of those established techniques in software engineering which might assist scientific software developers.
3. The establishment of a mapping between software techniques identified in 2 and contexts characterised along the dimensions identified in 1.
4. The identification of the means by which scientists might be made aware of those software engineering techniques and tools which might be relevant to their development.

This latter point is especially significant given the difficulties of sharing software development knowledge among professional end-user developers, as discussed in section 2.1 above.

This research agenda might appear daunting but I hope that this paper and others like it might contribute a significant first step.

6 Acknowledgements

I should like to acknowledge my deep gratitude to all those software engineers and scientists who took part in my field studies. They were invariably patient and reflective, and themselves contributed great insights.

7. References

Bache E. 2003, 'Building software for scientists: a report about incremental adoption of XP', 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, to appear.

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. Int'l Conf. Software Eng.* (CSE 2007), IEEE CS Press, 2007, pp. 550–559.

Chalmers, A.F., 1982. *What is this thing called science?* Open University Press, Milton Keynes, UK

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

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

Segal J., 2005, 'When software engineers met research scientists: a case study', *Empirical Software Engineering*, 10(4), 517-536

Segal J, 2007, 'Some problems of professional end user developers', VLHCC, IEEE Symposium on Visual Languages and Human-Centric Computing, 2007, pp111-118

Segal, J and Morris, C, 2008, 'Developing scientific software', IEEE Software, to appear.