Some challenges facing software engineers developing software for scientists

Conference or Workshop Item

How to cite:

For guidance on citations see FAQs.

© [not recorded]
Version: [not recorded]

Link(s) to article on publisher’s website:
http://dx.doi.org/doi:10.1109/SECSE.2009.5069156

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
Some challenges facing software engineers developing software for scientists

Judith Segal
Empirical Studies of Software Development group
Centre for Research in Computing
The Open University
Walton Hall
Milton Keynes MK7 6AA
UK
j.a.segal@open.ac.uk

Abstract

In this paper, I discuss two types of challenges facing software engineers as they develop software for scientists. The first type is those challenges that arise from the experience that scientists might have of developing their own software. From this experience, they internalise a model of software development but may not realise the contextual factors which make such a model successful. They thus have expectations and assumptions which prove challenging to software engineers. The second type is those challenges which, while not unique to the development of software for scientists, have especial significance in the context of such development. These include the challenges of ensuring effective user engagement and of developing software for a community.

1. Introduction

It is often the case that scientists develop their own software. This can be a highly desirable state of affairs: the scientist has a deep understanding of what is required from the software and can deliver it in a timely fashion in order to address some pressing scientific problem. Sometimes, however, such end-user development is not feasible. The complexity of the software might be such that the scientists recognise that they do not have the requisite development expertise; or the software might be intended to support a whole scientific community rather than just a particular individual, lab or project; or existing ‘proof of concept’ or prototype software as developed by scientists might need engineering to become production quality code. In such cases, software engineers commonly become involved.

The aim of this paper is to describe some major challenges that face software engineers as they develop scientific software. This paper is by no means exhaustive. None of the challenges described herein are technical: they arise from clashes of expectations or from social issues such as ownership or competition. In addition, most of the challenges described were identified from my field studies (of software engineers developing software for space scientists [1] and for biologists [2]), and obviously these studies do not cover the full spectrum of software engineers developing scientific software. In particular, high performance computing developments are not considered.

This paper extends the work I presented at the first SECSE workshop [3] by explicating the model of scientists developing software in Section 2 and articulating the challenges posed to software engineers as a result of the scientists’ expectations raised by this model in Section 3. In Section 4, I go on to discuss some other challenges, such as that of effectively engaging scientists in software development and the particular challenges associated with developing software for a scientific community. I do not claim that these latter challenges are unique to the development of scientific software: unlike those articulated in Section 3, they bear little or no relation to the particular characteristic of scientists that many of them have experience of developing their own software. I do, however, claim that they have especial significance in the development of scientific software. Effective user engagement is widely recognised as being an important success factor in any software development, but I shall argue that it is even more
crucial in general when the development concerned is
of scientific, rather than of commercial, software
because of the sheer complexity of the scientific
domain. As for community software, this is becoming
more important to science as many sciences ‘go large’,
that is, involve many scientists working on the same
basic problem and sharing large quantities of data. In
Section 5, I summarise the paper.

I shall begin by describing a model of how
scientists develop software.

2. A model of software development by
scientists.

From my field studies, I have identified a pervasive
model of how scientists develop their own software, as
in Figure 1. In this model, the developer forms a
vague idea of what is required and begins coding. He
(or she) then informally evaluates the software so
produced either on his own or with the help of
colleagues, asking questions such as: does this
software do what I (or we) want? Can it be usefully
extended? He then either modifies and/or extends the
code, or does some cursory testing, usually by
addressing the question: is the output broadly what I
expect? And if the answer to this question is in the
affirmative, then the development process is over.

Judging by its pervasiveness, this is a very
successful model. I claim that the following contextual
factors are a prerequisite to its success:

- the developer has a deep understanding of the
domain and what is required. This is
necessary both for the start of the process (the
developer can form a vague idea of what is
needed, that is, understands the high-level
requirements), and its termination (the
developer has the gut instinct that comes with
scientific expertise to judge whether the output
from the software is acceptable);
- the developer is either the sole user of the
software or is embedded in, and co-located
with, a cohesive community of users. It is thus
easy for him to say to his colleague at the next
bench ‘come and have a look at this’, or to
discuss ideas informally over coffee or lunch
and so address the question at the core of the
iteration: ‘Is this what I (or we) want?’;
- The software produced is designed to address a
particular problem for a particular group at a
particular point in time.

Provided all these contextual factors are in place,
then this model has the potential to produce an
effective piece of software in a timely fashion. If not,
for example, if the software is designed to satisfy the
needs of a heterogeneous group of users or to support
users over a period of time, then this model is no
longer appropriate, as issues such as maintainability
and negotiating requirements have to be addressed.

3. The impact of this model on
development challenges.

In this section, I shall describe some challenges that
face software engineers as they develop software for
scientists in the context where the scientists themselves
have experience of software development. These
scientists’ model of software development is as
represented in Figure 1, and the fact that they may not
be aware of the contextual factors which are necessary
for the success of this model can lead to many
challenges, including the following.

3.1. The adoption of an appropriate
development model.

Figure 1 represents an iterative incremental model,
and this echoes the way that many scientists do their
science. That is, they try something out in the
laboratory, reflect on it and possibly modify/extend it.
In [1], I describe the significant problems and
frustrations which arose when software engineers
engaged with scientists to develop scientific software
using a waterfall-type process model. The software
engineers wanted an up-front requirements
specification; the scientists persisted in using iterative,
incremental methods in order to discover their
requirements. Much frustration ensued. In addition,
the scientists were used to the face-to-face
communication which is implicit in Figure 1, and thus
found the use of documents as communication
artefacts to be both alien and ineffective.

It seems plausible, therefore, that software
engineers should always use an iterative, incremental
development model together with informal face-to-face
communication when developing software for
scientists so as to echo the latter’s existing work
patterns when developing their own software.
However, the adoption of such a model poses its own
problems, as I shall discuss in Section 4.
3.2. The challenge of establishing requirements

In the development model of Figure 1, the requirements are either already known to the developer given her (or his) knowledge of both the scientific domain and the domain of use (though they may not be fully articulated) or easily obtainable in an informal fashion (the developer just asks her colleagues). Thus, scientists may not appreciate that the gathering of requirements at both the high (functional) and low (user) level is often a significant part of software development.

In one of my field studies [2], significant problems were caused by the scientists assuming that the software engineer needed only functional requirements (for example, ‘we need to record experiments in a structured way’). The underlying assumption here was that translating these high level to more tractable low level requirements (for example, by articulating the specific way in which experiments are structured) is trivial, as it would be in the context described in Section 2. On the contrary, the software engineers, lacking either the experience of executing experiments or the wherewithal to ask scientists informally, found that this translation was highly problematic and very resource intensive.

The challenge here is for the software engineer to persuade the scientists that the establishment of requirements in contexts other than that described in Section 2, can be a complex, resource-intensive process.

3.3. The challenge of testing.

In the model described in section 2, testing is user acceptance testing and is done informally during the core loop (‘Is this what I/we want?’) and at the end of the process (‘Decide it’ll do’). In both cases, it relies on the scientist’s gut instincts that the behaviour of the software is consistent with the science. And of course the software engineer does not have such gut instincts.

The challenge here is for the software engineer to persuade the scientist that unit testing is necessary as well as acceptance testing, and that the latter can only be done effectively with the active involvement of the scientists incorporating the software into their normal work practices and then reporting back.

3.4. Different perceptions of software development time.

Scientists used to developing software according to the model in Section 2, are not accustomed to thinking of software as being for disparate groups or supporting a community over time. They are therefore not accustomed to addressing issues such as comprehensibility of the code, maintainability, modifiability or portability. Depending on the purpose of the code, software engineers expend a great deal of resource in addressing such issues. In addition, as discussed above, addressing both the establishment of requirements and testing are significant and resource intensive concerns for software engineers whereas to
scientists, they are just a natural integral part of the
development process (Figure 1).

The consequence of this is that developing a piece
of scientific software by software engineers takes far
longer than scientists expect, used as they are to the
timely delivery of software in the model of Figure 1,
and this can be the cause of much frustration. The
challenge here is for the software engineer to manage
the scientists’ unrealistic expectations of how long
software development takes.

3.5. The challenges of developing production
quality software given a prototype.

A common situation in which software engineers
are brought in to develop scientific software is when
‘proof of concept’ prototype software as developed by
scientists is re-engineered to production quality code.
This didn’t happen explicitly in my field studies, but
the experience of trying to incorporate user-developed
code into community code described in [2], leads me
to suggest that such software should be used in the
nature of a throwaway prototype, a means of reifying
the requirements, rather than a first step towards
implementation. This suggestion is made on both
technical and social grounds. Technically, the fact that
such software will almost certainly have been
developed according to the model of Figure 1, with
only cursory testing and no cognisance of the problems
of maintainability etcetera, means that it is unlikely to
meet the quality goals of production quality software.
Socially, there are issues of ownership: if the prototype
code is incorporated into the production code, then the
experience of [2] is that the scientist who developed
the prototype code is going to be very loath to allow
any radical changes, for example, of software
architecture, though such changes might be absolutely
necessary in order to ensure robustness or provide
support for a community.

4. Some other challenges of importance to
the development of scientific software.

In this section, I consider some challenges which
are not unique to the development of scientific
software, but which are of especial salience to such
development. The first of these is that of effective user
engagement, which I will argue is of especial
importance when the users are scientists. The second
is a set of challenges concerned with the development
of community software.

4.1. The challenges of user engagement.

User engagement is at the core of many current
software development approaches, such as user-
centered design and development, participatory design
and development and the various agile methods. The
problem of enabling effective user engagement with
respect to these methods is an on-going research topic,
see for example [5].

Whichever development approach is used, I argue
that user engagement is more important when the
software is being developed for scientific, rather than
for commercial, purposes. This is because of the
limitations of a software engineer’s knowledge. A
software engineer probably has some intuition as to the
requirements of, and possible test cases for, (say) a
hotel reservation system, but is unlikely to have the
same sort of intuition when the software is intended to
support (say) molecular biology. And, of course, the
success of an iterative, incremental model, as seems to
fit best with scientists’ work patterns (see 3.1), is
heavily dependent on users being effectively engaged
and giving meaningful feedback at the end of each
iteration.

As we saw in Section 2, where the developer is a
potential user or has a deep understanding of the
science and is co-located with users, then this user
engagement comes for free, as it were. But for
software engineers, the problem of getting users to
engage can be very real. As has been said by many
researchers, see, for example, [6] and [7], scientists
just want to do science. One can understand,
therefore, that they may be very loath to interrupt their
work in order to explicate requirements for a system
which cannot benefit them immediately (since the
requirements have to be implemented). In any case,
they might not be clear as to what their requirements
are. Many writers argue that it is very difficult for
users to know their requirements in the absence of an
artefact such as a prototype, [8], and a clear
understanding of how the software might impact on
their work, [9].

As for acceptance testing, in 3.3., we noted that this
is ideally done by the user integrating the software into
his/her normal work practices. But new software does
not usually slot seamlessly into existing work
practices: there is at least some learning to be done and
some perturbation of practice. In other words,
acceptance testing represents a cost to the user, and the
user might well not want to pay that cost especially if
the immediate benefit of using the software is not
clear.

One way of encouraging user engagement might be
to design the software development so that the output
of the first iteration immediately delivers scientific benefit, albeit limited. This has several advantages:

- It increases the trust of scientists in the ability of developers to deliver what they want;
- It provides them with a piece of software which they can use and on which they can reflect in order to modify or extend their requirements, see for example [8];
- It makes their user engagement more effective by enabling them to envision how the software might impact their work [9].

There is however a problem: without effective user engagement, how can such a first iteration be produced? This is a question which requires further consideration.

4.2. The challenges of developing community software.

The development of community software (that is, software intended for people with the same broad scientific interests, as opposed to for individual scientists, labs or projects) has had a huge impact on some branches of science. For example, the development of genomic and protein databases has enabled significant discoveries in molecular biology [10]. Nonetheless, such development is fraught with challenges. Most of these are related to issues of cooperation and collaboration. Such challenges include the following:

- The community may not be a true community, in the sense that they may not have any history of collaboration during which they have developed effective collaborative practices. Instead, the ‘community’ may be a set of disparate research groups who have come together for the purposes of gaining funds for a development which they perceive as being potentially beneficial to all. Even when this is not the case, problems of cooperation and collaboration abound when it comes to software development, as Star and Ruhleder found when they studied software intended to support the cohesive and long-standing community of biologists studying the model organism, the worm c. Elegans [11].
- The ‘tragedy of the commons’, [12], is a frequently occurring phenomenon in which software optimised for the needs of individual groups may not be optimised for the needs of the community. As is pointed out in [10], for software to be successfully deployed by an individual scientist, project or lab, it must support the work practices of that scientist, project or lab. On the other hand, in order to support a community, and in particular, to support a community over a period of time and for unanticipated uses, the software must be independent of particular work practices, as with the genomic and protein databases. This ‘tragedy of the commons’ impacts on the agreement and prioritisation of common requirements where, of course, there is the temptation for each group to press for the implementation of the requirements which suit them best.
- In the case of databases, there are problems with sharing data. Concerns of ownership and reputation, not wanting to publicise one’s data prematurely, and balancing cooperation with keeping a competitive edge, are all issues which need addressing, see, for example, [10] and [11].

- Issues of terminology have to be resolved. Coming from a background in algebra, I was surprised to discover that biologists appear to accept ambiguities in their terminology. I was used to a named algebraic structure (for example, a ring, field, group) being very clearly defined in terms of elements, operators and axioms. In biology, this state of affairs does not hold. For example, a named protein might not be the same as another protein having the same name – or might be the same as a protein having a different name – there is no accepted naming convention for proteins. Such terminology ambiguity is of especial concern when the software being developed is a database: how does one know that data entered by different groups into a particular named field all have the same semantics? Resolving these ambiguities is a non-trivial issue as ontology developers recognise [13], and is made more complex by issues of power, as in, who has the authority to say that this term is defined in this way; usage (for example, a lab which has been used to using a term in this way is going to find it difficult to adopt that); ownership, and other social concerns.

In addition to the barriers described in 4.1, all these challenges, unless successfully met, present further
impediments in the context of community software to successful user engagement.

5 Summary

In this paper, I have described two sorts of challenges facing software engineers as they develop software for scientists: those arising from the particular phenomenon of many scientists having experience of developing their own software, and those which are more general but of especial salience to scientists. I do not claim that this represents an exhaustive list of challenges. And I certainly do not claim to provide an exhaustive set of solutions. Solutions to the first set of challenges as described in Section 3 might well lie with software engineers understanding the scientists’ expectations as generated by the model in Figure 1, and carefully managing those expectations. Seeking solutions to the second set is currently an active research topic in Software Engineering and will doubtless continue to be so for many years.

My aim in writing this paper was to explicate those challenges which are well understood and to inspire debate about those which are not. I hope that this aim will be achieved.

Acknowledgements

I should like to thank Chris Morris for sharing with me the perspective of a software engineer struggling with the challenges described above over a period of years, and my colleagues in the ESSD group at the Open University, Marian Petre, Hugh Robinson and Helen Sharp, for their unwavering support and for many interesting conversations and discussions. And, of course, I should like to thank all the participants in my field studies, without whom there would be nothing.

6. References


