The Open University

Open Research Online

Citation

Segal, Judith (2003). When software engineers met research scientists: a field study. Technical Report 2003/14; Department of Computing, The Open University.

URL

https://oro.open.ac.uk/90117/

License

(CC-BY-NC-ND 4.0) Creative Commons: Attribution-Noncommercial-No Derivative Works 4.0

https://creativecommons.org/licenses/by-nc-nd/4.0/

Policy

This document has been downloaded from Open Research Online, The Open University's repository of research publications. This version is being made available in accordance with Open Research Online policies available from Open Research Online (ORO) Policies

Versions

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding

Technical Report No: 2003/14

When software engineers met research scientists: a field study

Judith Segal

13th November 2003

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

When software engineers met research scientists: a field study

Judith Segal

Department of Computing

Faculty of Mathematics and Computing

The Open University

Milton Keynes

MK7 6AA

UK

j.a.segal@open.ac.uk

TEL: +44 1908 659793

FAX: +44 1908 652140

Key words: field study; software engineers; scientific software; agile methodologies.

Abstract

In this paper, we describe a field study in which software engineers, following a traditional, staged, document-led development methodology, provided research scientists with a library of software components with which to drive a scientific instrument. Our data indicate two problems. The first of these is the clash between an upfront statement of requirements as needed by the traditional methodology and the emergence of requirements as occurs naturally in a research laboratory; the second is the fact that certain project documents do not seem to fully support the construction of a shared understanding between the scientists and the software engineers. We discuss whether the adoption of certain agile practices might ameliorate these problems.

1. Introduction

This paper describes a field study in which software engineers developed a library of embedded software components, as specified by research scientists, in order to drive a scientific instrument in space. The research scientists had a long history of themselves developing similar software for their own use in the laboratory but this was the first time they had worked with software engineers or used a traditional, document-led, staged methodology. Our data, as described in section 3 below, reveal problems with requirements (a traditional methodology demands an upfront statement of requirements whereas the scientists were used to the emergence of requirements over several iterations) and documents (it was clear that the requirement documents did not fully support a shared understanding between scientists and software engineers).

Since agile methodologies appear ideal in a situation where requirements emerge over time and dispense with unnecessary documentation, we discuss in section 4 below the potential of certain practices of a well-articulated agile methodology, XP, Beck 2000, to ameliorate the problems revealed by our data.

We begin by describing the background to our field study.

2. The background to our field study

In this section, we shall describe the context of the field study, the people at its heart and the means by which the data were collected and validated.

2.1. The context

The research scientists to whom we spoke all worked at the same research organisation. They fell into three groups: those who worked exclusively in the laboratory; those who had laboratory experience and who were now preparing for the first time to send an instrument up into space, and those who had some considerable experience of sending instruments into space. The focus of our field study is on the middle group, the scientists who were preparing for the first time to send an instrument up into space. On the whole, this group had worked closely together in the laboratory for several years. Over the period of the field study, they had little contact with the latter group, those with considerable experience of space science, who had joined the research organisation only relatively recently.

In the laboratory, the scientists develop their own instruments for the collection and analysis of the relevant scientific data. Building an instrument usually involves the tailoring and combining of off-the-shelf components. In addition, the scientists develop the software both for controlling the instruments and for collecting and analysing the subsequent data.

Designing and building an instrument together with its concomitant software, is considerably more complicated when the instrument is destined for space rather than for a laboratory. There are physical considerations of size and mass; there is the problem of restricted power consumption; there are also problems of integration with the infrastructure of the spacecraft and its software, and with other experimental instruments on the same flight. Because of this complexity, the research organisation worked together with instrument builders and software engineers at another organisation in order to develop the instrument and its embedded control software. This other organisation has considerable experience of developing instruments for experiments in space, though none, we are told, has yet been so complex as the instrument at the heart of this study.

As far as the software is concerned, the software engineers at this other organisation were charged with providing a library of embedded software components which implement commands to control the instrument's hardware components. After the spacecraft is launched, the research scientists intend to use ground support software, which they themselves have developed, in order to send instructions to the instrument consisting of a sequence of calls to these primitive commands. The ground support software is also used for driving models of the instrument and the spacecraft's on-board data management computer in order to test the sequence of calls, and will be used to analyse the scientific data when the instrument reaches its destination.

2.2. The people, the data collection and validation

In order to collect data, we conducted semi-structured interviews with eleven different people, of whom ten worked at the research organisation and the eleventh was a software engineer at the other organisation. These interviews were audio taped and fully transcribed: the quotes presented in section 3 below are from these transcriptions, unless otherwise stated. In addition, there were eleven follow-up meetings, which were not taped, but were fully noted. We also had email and phone conversations, and looked at formal documentation pertaining both to this and to other space projects.

At the research organisation, the people we talked to included a senior manager, who is a distinguished research physicist with no current direct involvement in software; the senior scientific project manager during the phase when the instrument was being developed for delivery to the spacecraft, and the junior scientific project manager who took over from him after the delivery; the research scientists, who had varying experience of writing software in the laboratory and now had to turn their attention to specifying their requirements for the software engineers, and 'the project programmer', who was charged with developing the ground support software. Despite his title, the latter had very little experience in software

development. All these people had backgrounds in physics, chemistry or engineering; all would describe themselves as physicists, chemists, engineers or space scientists, rather than software developers.

We also spoke, by way of comparison and contrast, with research scientists who worked only with instruments in the laboratory which they developed themselves and for which they wrote their own software, and with a couple of space scientists who had recently joined the research organisation and already had some experience of working with software engineers in order to develop instruments for space.

As regards the software engineers, we spoke to the project manager with overall responsibility for the delivery of the embedded instrument software to the research organisation. This man has over 30 years' experience of software development.

Once the evidence had been amassed and the arguments of the following sections marshalled, we did what Seaman, 1999, refers to as 'member checking'. That is, we went back to the people who had provided us with most of our data and ensured that our interpretation of the data was correct and that our arguments 'rang true'.

We shall now describe the problems revealed by our data.

3. The problems revealed by the field study data

3.1. Software development: the theory

In an email to the investigators, the project manager at the research organisation described the theoretical steps for developing software for a new experimental instrument in space, as follows:

1. The research scientists define the capabilities of the instrument.
2. The scientists then define a system configuration, that is, the components necessary to satisfy the capabilities defined in 1.
3. They follow this by defining how the components will operate.
4. The electronic and software requirements are then defined in user requirements documents (URDs), which specify the way that the components of the system configuration will operate in order to obtain the required data. At this point, it is determined which requirements will be implemented in hardware and which in software.
5. The software engineers respond to the software user requirements document (the SURD) with a software specification document (SSD) and associated definitions documents
6. The scientists review the SSD against the SURD.
7. The software engineers write the code, testing it against the SSD
8. When the code is finished, the scientists test it against the SURD.

The scientists determine the instrument's capabilities, components, electronic and software requirements needed for steps 1 – 4 above, by creating a 'demonstration model', based in our field study on modifications of laboratory instruments. Essentially, this model demonstrates the functionality required of the instrument to be sent into space.

We shall now describe how the practice differed from the theory.

3.2. The practice: firming requirements

We see in the above theoretical description that there is a distinct stage (step 4) at which requirements are explicated before the software is developed. It is possible to change requirements after this stage, but it isn't easy.

This is totally different from what happens when instruments and their accompanying software are developed in the laboratory. As illustrated by the following quotes from research scientists, development in the laboratory proceeds iteratively, and requirements are in a continual state of change, rather than being explicated at any point.

> 'I think with a lot of systems we have – they evolve… you don't design them a hundred percent working instrument from scratch. You come up with an idea. And then you use it, and then you realise that… it needs tweaking'

> 'Generally what tends to happen with me is, I do a first attempt… start writing programs, start using it, realise it's not quite what I wanted, and then have a second attempt…. '

> 'I would say I am programming experimentally… I did many experiments in programming something and [if] I see it doesn't [work] then I rearrange something'

This is true even in situations where a scientist is specifying software to be developed by a fellow scientist:

> '… and the fact that we were working closely together means that you can have plenty of iterations down the line. So you don't have to come up with everything initially' [scientist describing how he specified software for a close colleague to develop].

> 'I've been doing the programming, which allows him to do some experiments in the lab. And I've done it and [he says] "I need it to be able to do this" [I] go away and do it and come back. [he says] "This didn't work. Change it…. I wanted it to be like this…" ' [scientist describing working with a senior colleague]

The research scientists developed the demonstration model needed in order to articulate the requirements (as in 3.1) in the iterative manner to which they were accustomed, as described above, to the frustration of the software engineers awaiting the requirements documents.

> '.. being like a bunch of scientists, we [thought] we could change everything up until the last minute… [the software engineers] were just saying "Sort the requirements out now! Do it now! You haven't got time!" ' [scientist]

The requirement documents were thus delivered somewhat later than is ideal within the theoretical framework of 3.1. and this late delivery increased the time pressures on the project.

3.3. Problems with communication

Implicit in the staged development described in 3.1. is the use of documents to construct a shared understanding of the domain, as communicated by the research scientists, and the software development, as communicated by the software engineers. The software user requirements document, the SURD, tells the developers what the scientists want; the software specification document, the SSD, tells the scientists how the developers are going to implement their needs in software.

Our data reveal three problems. The first concerns understanding the role, and the subsequent writing, of the requirements document; the second lies with a model of knowledge possession in which domain knowledge is only possessed by the scientists, and the final problem concerns the deployment of both the requirements and software specification documents as communication artefacts. We shall now look at each of these problems in more detail.

We shall firstly consider the difficulty of writing the requirements document. Our data indicate that this is due, in part at least, to the scientists not understanding the role of such a document. Previously, the only documents they had written had been scientific papers. But scientific papers do not have the contractual /management role of a requirements document: rather, they are written to inform, to persuade and to convince. It took the scientists a long time to work out that project documents perform different roles from scientific papers, and require a different domain of discourse. To expand further on this, a document which is basically a contract, that is, either a statement of needs or a statement of how those needs are going to be satisfied, between people of different disciplines, has a different structure and employs a different vocabulary from a document which seeks to inform and persuade people in the same discipline as the document writer(s). In our field study, the software engineers were instrumental in helping the scientists

draft the requirement document so that it could easily be transformed into a software specification document. As the scientific project manager said:

> ‘ a lot of it is just trying to learn the language of the programmers in terms of how they write documents. .. we were talked through: [we said] “These were the kind of things we want to be able to do. ...” And they would say “Well. This is how you can write it in language that we can then respond to" ’

We now consider the fallacy of considering domain knowledge to be the exclusive possession of the scientists, and the failure of the requirements and specification documents in constructing a shared understanding between the scientists and developers.

Given that the software engineers were known to have much experience of developing software for space instruments, it is not surprising that the scientists expected them to provide some domain knowledge in the shape of, for example, useful features for controlling the instrument. The software engineer was aware both of this expectation, and his organisation's failure to meet it, as in the quote below.

> '…. I think if you talk to [one of the scientists] he might tell you there are things that he really would have liked that didn't find their way into the user requirements document, that we didn't think about until afterwards.… perhaps we should have been more ready to come back to the [research organisation] and say "Are you sure that you don't want it to do this?"…. Perhaps we should have said "Now are you sure you don't want these [features]?" and then I think [the scientist] would have said "Well actually, yes we do want them" ' [software engineer]

He was also aware that the absence of these features in the requirements document did not suffice to inform the scientists of their absence in the software:

> 'There are certainly lots of things which in retrospect would have been useful if they'd been incorporated in the software. They were left out and I have the feeling that perhaps [the research organisation] weren't really aware they were being left out. *Although, of course, if you read the user requirements document, nowhere.. is it stated that [these features] will be present. … [the research organisation] were sort of assuming that these [features] were going to be available. And we were sort of assuming they knew they weren't'* [the software engineer, our emphasis].

Besides the failure of expectation above, which might be regarded as errors of omission in the software, there were also errors of commission. It was reported that one of the software engineers, having some knowledge of the proposed experiment, added some features to the software on his own initiative. The scientists were uneasy about this, feeling that the result of this initiative was that they were left uncertain as to what the software actually did. Again, this points to a failure of the requirements document as a document for communicating shared understanding: these added features were fully documented therein.

Given that the software developers coached the scientists as to the form of the requirements document, as we saw above, we assume it conveyed the right sort of information to form the basis of the software specification document. But the quote below indicates that the software specification document did not satisfactorily convey information to the scientists, to the detriment of step 6 of the theoretical development detailed in section 3.1 (that the scientists should review the specification against the requirements document):

> ‘Did we read the specifications? – I think it’s very difficult to read a list of specifications that goes from specification one to specification four hundred. You know, it’s a very very dry language is a software specification document [*laughs*]' [scientific project manager]

3.4. Problems with testing

The theoretical model of software development, described in 3.1. above, has three stages of testing: stage 6, in which the scientists review the SSD, the software specification document, against the SURD, the software users requirement document; stage 7, where the software engineers test the code against the SSD, and stage 8, where the scientists test the code against the SURD. In these testing stages, the SSD and SURD are being used as contractual documents, in the sense that the software engineers contract to supply software according to the SSD, and the scientists test that the contract has been fulfilled by comparing the

software with the SURD, having first checked that the SSD accurately reflects the SURD. Our data demonstrate that the documents did not fulfil their contractual role. The quote above from the scientific project manager, leads us to doubt that stage 6 was done exhaustively, that is, that the mapping from SURD to SSD was fully checked. The scientist who gave the instrument its final testing prior to its delivery to the spacecraft, reported that he had never read the SURD, so stage 8 was not carried out. As to stage 7, where the software engineers tested the code against the SSD, their project manager admitted that time ran out; testing fell victim to the time pressures created (in part, at least) by the requirements problems described in 3.2. It is clear that the testing methodology laid out in section 3.1 was not adhered to and that this was due, in part at least, to a failure of the documents as contractual instruments.

There are two further comments we should like to make about testing. First and foremost is the fact that, just because one particular testing methodology was not adhered to, does not necessarily mean that the testing done was inadequate. The second comment concerns the impossibility of comprehensively testing this particular instrument and its software at this stage. This is because the physical context in which the instrument will work, the radiation levels, temperatures, the composition of the atmosphere, although predicted by various scientific models, are not known for certain. The final testing of the instrument and its software will thus have to come when it reaches its destination. This testing will therefore be heavily dependent on how well knowledge of the system can be communicated over time, which is an issue we hope to explore further in the future.

3.5. Discussion

The data described above reveal the following problems:

- The constraint imposed by most traditional software methodologies, that requirements be as fully explicated as possible before design and development, is antithetical to the way that research scientists work.

- Requirements and software specification documents did not suffice to construct a shared understanding between the research scientists and the software engineers, leading to some uncertainty as to what the software should deliver. Also, contrary to theoretical expectations, these documents played no part in the scientists' testing.

When we presented these findings to those research scientists who had provided most of the data, their response was that this was the first time that they had worked with software engineers and that it had constituted a valuable learning experience. They have subsequently developed another instrument under similar conditions, and claim to have taken on board the lessons learnt: their perceptions are that their new demonstration model did not undergo as many iterations as the one discussed above so that the requirements were delivered to the software engineers earlier, and that they now are much better at structuring the various project documents.

We should point out, however, that this new instrument is very similar to the instrument developed in the course of the field study (hence, perhaps, the fewer number of iterations in the development of the demonstration model). We should also note that emerging, rather than upfront, requirements seems to be the backbone of instrument/instrument software development in the laboratory, as evidenced by the quotes in 3.2. One space scientist, with some considerable experience of working with software engineers to develop experimental instruments for space and not part of the core team considered in this paper, viewed the difference between emergent and up-front requirements as illustrative of a culture gap between scientists and (the more traditional) software engineers, as evidenced in the following quote:

> 'on the science side, we're not good enough at defining the specifications. But I suppose we have more of a view… that you write it and then test it and then improve it a little bit having looked at the output and so on. Whereas the software developer would rather write it and then view the thing as more or less finished. But it's really only the *start* of the development process once the software is running in conjunction with the experiment….' [our emphasis]

In the remainder of the paper, rather than considering whether research scientists can adapt themselves to the constraints of working with a traditional, staged, document-led, software development methodology, we wish to explore whether another radically different methodology can be gainfully used by software engineers working with research scientists. We shall consider agile methodologies, which seem to match better than traditional methodologies with how research scientists develop software in their natural environment, the laboratory. As we shall discuss in more detail in section 4 below, agile methodologies both cater for emergent requirements and de-emphasise documentation.

4 Would an agile software development methodology have been better?

Agile software development methodologies are based on the principles laid out by the Manifesto for Agile Software Development, 2001. Agilists (supporters of this manifesto) value response to change over following a plan; individuals and interactions over processes and plans; working software over comprehensive documentation, and customer collaboration over contract negotiation. The first value in particular encourages agilists to adopt practices which support iterative development and emergent requirements; the other values lead them to question the necessity of traditional product documentation. XP, eXtreme Programming, Beck, 2000, as summarised in Maurer & Martel, 2002, is perhaps the best known agile methodology with a set of well-articulated practices.

Before we analyse the potential of these practices to ameliorate the problems identified in section 3, we shall firstly describe the results of a trawl of recent literature in order to find accounts of the adoption of XP in contexts similar to our own.

4.1. The practice of XP in contexts similar to our field study

Our field study is concerned with scientific research and embedded software. Our literature trawl found only a few accounts of the adoption of XP by organisations with similar concerns. Wood and Kleb, 2003, evaluated the use of XP at NASA in a project in which the writers developed a software testbed for evaluating the performance of some numerical algorithms. Overall, their evaluation was positive, and, according to the writers, endorsed the use of XP for mission-critical software development at NASA's Langley Research Center. Ronkainen and Abrahamson, 2003, discussed whether agile methodologies could be tailored for use in developing embedded software concurrently with hardware (co-design) in the telecommunications industry, where the meeting of hard real-time requirements (for example, of co-ordination and performance) is the number one priority. They noted that embedded software development has to face many of the same problems that agile methodologies are designed to address, but identified several problems with existing methodologies. These include the maintenance of communication between multiple distributed development teams; the need for up-front architecture and design in embedded software, and the transition to production code given the potential risk to performance caused by refactoring. Finally, Bache, 2003, wrote very enthusiastically of her experience of using XP practices in developing software for experimental chemists at AstraZeneca:

> 'In my experience, pairing a scientist with a software professional is a smart move, and then using an agile process like XP turns it into a winning combination'.

We should note that these accounts were all written by practitioners musing on their own experiences for the benefit of other practitioners. We must thus be aware of potential weaknesses in the evidence produced. The writers might forget what actually happened; might shape the argument to accord with some social, organisational or political pressure, and might not record tacit knowledge or taken-for-granted procedures.

Having considered the (sparse) recent literature on introducing XP in contexts similar to ours, we now turn our attention back to our own study. We consider separately the topics of emergent requirements, communication and (very briefly) testing. For each topic, we identify and describe XP practices which have the potential of ameliorating the problems identified in section 3 above, and then discuss some issues involved in adopting these practices in contexts similar to that of our field study.

4.2 Emergent requirements

One of the main arguments for the use of agile methods is that they are intended to address the issue of emergent requirements. Rather than taking the view that changes in requirements are exceptional, as in traditional methodologies, or that they can be largely predicted, and hence allowed for by a judicious choice of software architecture, agilists consider that unpredictable changes in requirements are the norm. As Beck, 2000, says, in a quote which seems to mirror exactly the scientists' view of requirements, as quoted in 3.5:

> 'This is an absolute truth of software development. The requirements are never clear at first. Customers can never tell you exactly what they want. The development of a piece of software changes its own requirements. As soon as the customers see the first release, they learn what they want in the second release… or what they really wanted in the first. And it's valuable learning … that can only come from experience' [p.19]

XP supports this model of emergent requirements by short release cycles and by having a customer on the developers' site, so that the customer can respond quickly to the current piece of software and collaborate with the developers in scoping and prioritising the features for the next release. Requirements are firmed by the customers writing acceptance tests (with the aid of developers) for these new features.

Of course, with this model of software development, it is necessary for the software to be easy to change. In XP, this is facilitated by keeping the software simple, which, in turn, is facilitated by several XP practices. Two of these practices are purely human-centred: pair programming and collective ownership. With pair programming, the assumption is that peer influence and the need to communicate the intent of the software to the other person in the pair, encourages simplicity, see Williams et al., 2000. Collective ownership implies that anyone who spots an opportunity to simplify a design is expected to make use of that opportunity. Other practices which support simplicity include: designing only for current needs; refactoring, and adherence to agreed coding standards. Change may also be indirectly supported by automated testing, in that developers might be more inclined to make changes when they know that any bugs which are thus introduced can be quickly identified.

In our context, the customer (scientist) is not interested in the software per se, but rather in the instrument together with its embedded software. As in Ronkainen and Abrahamson, 2003, discussed in 4.1. above, there are many stakeholders in our study involved in instrument development besides the scientists: there are the software and hardware developers; the electronic experts; the managers of the spacecraft. There are also many interfaces, not just between the user (the scientist) and the software, but also, for example, between the software and the instrument, and the instrument software and the spacecraft's on-board computer. We need to investigate whether it is possible to develop an instrument iteratively within such a complex environment. At the very least, this requires good communication and coordination between the scientists, software and hardware developers, and electronic experts.

Although it is difficult to ignore hardware in our study, we are primarily concerned with the role of software. We see in theory no reason why the software developers might not adopt practices such as pair programming (which we will discuss in more detail in 4.3. below), collective code ownership and adherence to coding standards in order to make the software more comprehensible and hence easier to change during development. As to the XP practices of refactoring and automated testing, we speculate that these may be adopted most easily when software is developed using an object-oriented approach and a sophisticated development environment. In our case, the approach was not object-oriented and the development environment was far from sophisticated. Refactoring and automated testing might thus be difficult to adopt.

4.3. Communication

There are essentially two ways to achieve communication: via face-to-face interaction, or via communication artefacts such as documents. The proponents of agile methodologies favour the former for communication within a development team. The following is given as one of the principles of the agile manifesto:

> 'The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.' [that is, as opposed to using project documentation. http://agilemanifesto.org, accessed 17th February, 2003]

In XP, communication within the development team is facilitated by the team being (ideally) in the same open-plan office; communication between the development team and the customers is facilitated by having a customer on the developers' site, and by the use of a metaphor to tie together the task domain with the software architecture.

Where face-to-face interaction is not possible, for example, where information has to be conveyed over time or over distance, code and the test suite are favoured over traditional project documents. For example, Beck, 2000, talks of the reliance of XP on

> 'oral communication, tests and source code to communicate system structure and intent' [Beck, p.xvii. It should be understood, though, that this reliance doesn't preclude some element of more conventional documentation]

claiming that

> 'code can be used to communicate – expressing tactical intent, describing algorithms, pointing to spots for possible future expansion and contraction' [ibid, pp 44-45]

This contrasts with what many see as the over-emphasis on project documentation in traditional methodologies. For example, DeMarco, in DeMarco & Boehm, 2002, refers to document bloat as being endemic in the software industry, and Paulk, 2002, agrees, describing over-documentation as a 'pernicious problem'.

In our field study, there are various communication activities in which we are interested:

1. Communication between the research scientists and the software engineers;
2. Communication between the software engineers and the managers of the space-flight;
3. Communication over time. This is of particular salience in our context, since the final acceptance test of the software, its sending back of the experimental data, will only occur some years after it was written.

Regarding the first of these, communication between the research scientists and the software engineers, given our findings of section 3, it seems natural to explore ways other then documentation for constructing a shared understanding and firming requirements. Given that research scientists are used to developing their own software, and with the endorsement of Bache, 2003, as discussed in 4.1. above, we suggest that paired programming, with one of each pair being a research scientist and the other being a software engineer, might be the most effective way of constructing a shared understanding. To counteract any argument that paired programming is too costly, we should point to evidence in the literature, see, for example, Cockburn and Williams, 2000, Wood and Kleb, 2003, that the productivity of having pairs program the same task is very similar to that of using a single programmer, and any (small) extra cost is justified by the quality of the code so produced. In all fairness, however, we should point out that the aforementioned articles refer to pairs in which both partners are software developers but for whom the sharing of domain knowledge is not an issue. In addition, we suggest that the communicative role of documentation in forming a contract between scientists and developers might better be effected by the research scientists writing acceptance tests as described in 4.2. above. The same scientist could both pair-program and take the XP role of customer (writing acceptance tests and clarifying requirements): Wood and Kleb (op.cit) found that this combining of roles worked well.

The second type of communication identified, that between the software engineers and the spacecraft managers, poses no problems. Proponents of agile methodologies are not against documentation per se, but just against unnecessary documentation, see, for example, Cockburn, 2000, Boehm, 2002, and Ambler, 2002. A specification of (for example) the interface between the instrument software and the spacecraft's on-board data management system, might, indeed, be necessary documentation. Given that the need for such a document is clear and that it is an instrument of communication between software engineers on both sides, we do not feel that there will be any difficulty in either writing or comprehending it.

The third type of communication, over time, does pose real problems, we feel. We are sceptical of Beck's claim, quoted above, that code can express tactical intent, or point to spots for possible future expansion and contraction. Perhaps he means comments in the code, rather than the code itself. We intend to explore further in future work the issue of sharing knowledge over time by way of the community of practice, see, for example, Brown and Duguid, 2000, and information artefacts.

4.4. Testing

XP espouses a test-driven model of software development. Unit tests are written before features are implemented.

> 'Development is driven by tests. You test first, then code. Until all the tests are run, you aren't done' [Beck, 2000, p. 9]

> 'Software features that can't be demonstrated by automated tests simply don't exist' [ibid. p.45]

As we saw in 4.2., the customer, aided by the developer as necessary, writes functional tests. The full test suite is run at each change to the software system.

In the context of our study, we note (like Ronkainen and Abrahamson, 2003) that test-driven iterative development is far more complex in the case of embedded software than in non-embedded. In order to make a scientist's acceptance test pass, not only must software be developed but also the concomitant hardware (or a simulation thereof). We also noted in 4.2. that automated testing is unlikely to be supported by unsophisticated development environments, such as that used in our field study.

4.5. Discussion: taking our work forward

We return now to the question posed at the head of this section: Would an agile software methodology have been better? Given our discussion in the preceding sections, the answer to this question appears to be by no means straightforward. We make no apology for suggesting that some XP practices (such as pair programming, having a scientist collocated with the software developer acting both as a pair programmer and a customer; adherence to coding standards) might be easier to adopt and hence more useful in our context than others (refactoring; automated testing). In so doing, we are following in the spirit of Glass, 2002, Cockburn, 2000 and Robinson et al., 1998, among others, who all advocate that *any* software development methodology should be tailored to the project in hand. In particular, Boehm and his colleagues, Boehm, 2002, Boehm and Turner, 2003, have explored the melding of agile with plan-driven methods.

In order to explore these issues further, we need to conduct a field study of the adoption of our suggested XP practices by software engineers working with research scientists, in a context similar to that described in this paper. As we saw in 4.1., there are currently few published accounts of such adoption and such that there are, are written by practitioners reflecting on their own experiences, with all the potential problems of subjectivity. We intend that our proposed field study should have some ethnographic elements: at the very least, it should (like the study described in this paper) involve an investigator who is removed enough from the situation to view everything as 'strange' and to question any tacit assumptions. A discussion of the issues involved in studying the practice of software engineering can be found in Robinson, Segal and Sharp, 2003.

5. Summary and conclusion

This paper begins with an account of the problems seen when software engineers worked with research scientists in order to provide a library of embedded components for an experimental instrument within the framework of a traditional, document-led, staged, software development methodology. The data revealed several problems, foremost among which were the clash between a culture in which requirements emerge and one in which they are articulated up-front, and a failure of the project documents to promote shared communication between the scientists and the software engineers. Given these problems, it seems natural to explore whether agile methodologies, with their support for emergent requirements and their de-

emphasis on documentation, might have provided a better framework for software development than the traditional methodology used. While agile methodologies clearly have some potential to ameliorate the problems identified in our field study, questions such as how to tailor XP practices so that they fulfil this potential in a concrete setting, can only be answered by further field studies.

Acknowledgements.

The author would like to thank Jane Foody for conducting and transcribing most of the interviews, and, of course, the research scientists and software engineers involved for their generosity in giving freely of their time and for their most thoughtful comments and feedback.

This research was partly funded by the Open University Research and Development Fund, grant number 795.

References

Ambler, S.W. 2002. Agile Documentation. http://agilemodeling.com/essays/agileDocumentation.htm, accessed August 20th, 2003

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

Beck, K. 2000. eXtreme Programming Explained: Embrace Change. Addison Wesley.

Boehm, B. 2002. Get ready for agile methods, with care. IEE Computer 35(1): 64-69.

Boehm, B and Turner, R. 2003. Using risk to balance agile and plan-driven methods. IEE Computer 36: 57-66.

Brown, J.S. and Duguid, P. 2000. The Social Life of Information. Harvard Business School Press.

Cockburn, A. 2000. Balancing lightness with sufficiency. Cutter IT Journal 13(11): 26-33.

Cockburn, A. and Williams, L. 2000. The costs and benefits of Pair Programming. XP 2000.

DeMarco, T. and Boehm, B. 2002. The agile methods fray. IEE Computer 35(6): 90-92.

Glass, R. 2002. Searching for the Holy Grail of Software Engineering. Comm ACM 45(5): 15-16

Manifesto for agile software development, http://agilemanifesto.org/, accessed 22nd October, 2003.

Maurer, F. and Martel, S. 2002. Extreme Programming Rapid Development for Web-Based Applications. IEEE Internet Computing 6(1) 86-90.

Paulk, M. 2002. Agile methodologies and Process Discipline. Crosstalk, The Journal of Defense Software Engineering 15(10): 15-18

Robinson, H. Hall, P. Hovenden, F. and Rachel, J. 1998. Postmodern Software Development. The Computer Journal 41(6): 363-375

Robinson, H. Segal, J. and Sharp, H. 2003. The case for empirical studies of the practice of software development. to appear, WSESE

Ronkainen, J. and Abrahamson, P. 2003. Software development under stringent hardware constraints: do agile methods have a chance? XP and Agile processes in Software Engineering, Marchesi M, Succi G (eds.), Springer, LNCS 2675, 73-79.

Seaman, C. 1999. Methods in empirical studies of software engineering. IEEE Transactions on Software Engineering 25(4): 557-572.

Williams, L. Kessler, R. Cunningham, W. and Jeffries, R. 2000. Strengthening the field for pair programming. IEEE Software 17(4): 19 – 25

Wood, W.A. and Kleb, W.L. 2003. Exploring XP for Scientific Research. IEEE Software, 20(3): 30-36.