Empirical Software Engineering, 10, 517–536, 2005.

When Software Engineers Met Research Scientists: A Case Study

JUDITH SEGAL j.a.segal@open.ac.uk
Department of Computing, Faculty of Mathematics and Computing, The Open University, Milton Keynes, MK7 6AA, UK

Editor: Marvin Zelkowitz

Abstract. This paper describes a case study of software engineers developing a library of software components for a group of research scientists, using a traditional, staged, document-led methodology. The case study reveals two problems with the use of the methodology. The first is that it demands an upfront articulation of requirements, whereas the scientists had experience, and hence expectations, of emergent requirements; the second is that the project documentation does not suffice to construct a shared understanding. Reflecting on our case study, we discuss whether combining agile elements with a traditional methodology might have alleviated these problems. We then argue that the rich picture painted by the case study, and the reflections on methodology that it inspires, has a relevance that reaches beyond the original context of the study.

Keywords: Case study, software engineers, scientific software, agile methodologies, tailoring methodologies.

1. Introduction

This paper describes a case study in which software engineers followed a traditional staged document-led methodology in order to develop a library of instrument-driving software components for a group of research scientists. Two problems with the development are revealed. The first is concerned with requirements: the research scientists are experienced in developing their own software in the laboratory in a highly iterative manner, and having requirements emerge in succeeding iterations. They thus did not appreciate the need to articulate requirements fully and upfront as demanded by a staged methodology, and found this articulation very difficult to do. The second is a problem of communications: using contractual documents (requirement and specification documents) together with formal minuted meetings, did not suffice to construct a fully shared understanding between the scientists and the software engineers.

Case studies can sometimes be viewed with suspicion in the academic empirical software engineering community. Glass et al. (2002), in their review of published research in software engineering, express surprise at the paucity of field/case studies therein. Although Glass et al. did not include the journal of Empirical Software Engineering in their review, a taxonomy of this journal's publications (Segal et al., 2005), paints a similar picture. But we argue in Segal (2004) and Robinson et al. (2003) that case studies are essential if we are to understand the actual *practice* (as opposed to the theory) of software development, and that such understanding is an essential prerequisite to the primary aim of empirical software engineering, which is to inform practice.

The suspicion occasioned by case studies may be based on three related fears. These are fears of: their lack of objectivity; the great difficulty—or more likely, the impossibility—of their replication, and their lack of generalisability.

Regarding the first of these fears, lack of objectivity is a threat to any research, involving either qualitative analysis, as in case studies, or quantitative, as in controlled experiments, as we discuss in Segal (2004). We concur with Lanubile (1997) who says.

> The objectivity of empirical research comes from a review process that assures that the analysis relies on all the relevant evidence and takes into account all the rival interpretations. This can be done (or not done) in both a quantitative and a qualitative analysis. [p. 102]

In our case study, we tried to minimise the threat to objectivity by collecting data from diverse sources including interviews with 11 people from, essentially, four different groups, and from published documents, as recommended by Bratthall and Jorgensen (2002); by checking and enriching our understanding with follow-up phone-calls and emails, and, finally, by presenting our interpretation to our interviewees for them to check that it 'rang true,' as advocated by Seaman (1999). This is consistent with the principles articulated by Klein and Myers (1999) for the conduct of interpretive case studies, in which emergent understanding of the rich picture is constructed by multiple interactions between the researchers and the participants.

Regarding the second fear, it is true that the rich context of a case study, describing events which took place in a particular context, at a particular place and during a particular time period, cannot be replicated. But does this always matter? The rich interpretive picture provided by a case study might communicate more successfully to practitioners than replicable, traditional scientific experiments. As Zelkowitz et al. (2002) say:

> . . . the industrial community is generally wary of laboratory research results [p. 255]

and Wenger and Lave, as quoted in Cockburn (2002), emphasise the effectiveness of context rich stories in conveying ideas:

> The world carries its own structure, so that specificity always implies generality (and in this sense, generality is not to be assimilated to abstractness). This is why stories can be so powerful in conveying ideas, often more so than an articulation of the idea itself. [p. 55]

Even within the well-known positivist scientific tradition, in which hypotheses are formulated, refined, and then tested by means of experimental and control groups, and where the results of laboratory research may be expected to be the most salient, case studies can provide valuable data. Seaman (1999) illustrates how such studies might be used to formulate and refine hypotheses, and Lee (1989) describes their use in hypothesis testing.

Finally we consider the fear of lack of generalisability. In Section 5 below, we address this issue in the context of our case study by discussing how our findings, focused on a particular team, at a particular place and point in time, together with our reflections on these findings, as discussed in Section 4, might inform the practice of software engineering in other contexts.

We begin by describing the context of our case study, in Section 2, and the data arising therein in Section 3.

2. The Background to Our Case Study

In this section, we shall describe the context of the case 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, as illustrated in Figure 1: those who worked exclusively in the laboratory, denoted by LAB in Figure 1; those who had laboratory experience and who were now preparing for the first time to send an instrument up into space, denoted LABtoSPACE in Figure 1, and those who had some considerable experience of sending instruments into space, denoted SPACE. The focus of our case study is on the middle group, LABtoSPACE, 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 case study, they had little contact with the group denoted in Figure 1 as SPACE, that is, 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, situated in another town, 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 as 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

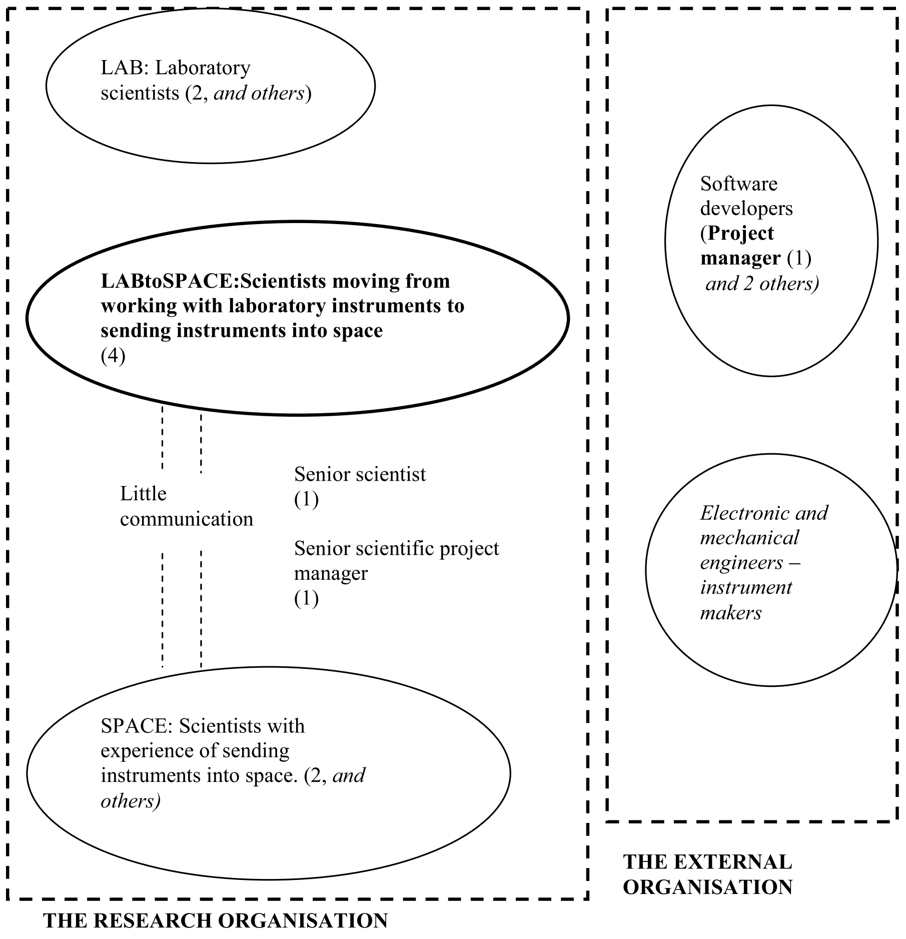
LAB: Laboratory scientists (2, *and others*)
LABtoSPACE:Scientists moving from working with laboratory instruments to sending instruments into space (4)
Little communication
Senior scientist (1)
Senior scientific project manager (1)
SPACE: Scientists with experience of sending instruments into space. (2, *and others)*
Software developers (**Project manager** (1) *and 2 others)*
Electronic and mechanical engineers – instrument makers
THE EXTERNAL ORGANISATION
THE RESEARCH ORGANISATION

Figure 1. People involved in the case study, with those at the focus in bold. The numbers given in brackets are the numbers interviewed. Italics denote people not interviewed. In the case of the software developers, this was because the developers in question had left the organisation.

sequence of calls to these primitive commands. The ground support software is also used for driving models of both 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

As recommended by Bratthall and Jorgensen (2002) we collected data from many sources.

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 upwards of 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, as in Figure 1, 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;
- the junior scientific project manager from the LABtoSPACE group who took over from the senior project manager after delivery of the instrument;
- two further research scientists from the LABtoSPACE group 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;
- 'the project programmer' in the LABtoSPACE group, who was charged with developing the ground support software. Despite his title, the latter had very little experience in software development: his main interest was in space science per se.

We also spoke, by way of comparison and contrast, with

- two research scientists in the LAB group, who worked only with instruments in the laboratory which they developed themselves and for which they wrote their own software;
- two space scientists from the SPACE group who had recently joined the research organisation and already had some experience of working with software engineers in order to develop instruments for space.

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.

As regards the software engineers at the external organisation, 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. There were two further programmers/developers who worked on the instrument software: both of these had left the organisation by the time of the case study.

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 Case Study Data

3.1. Software Development: The Theory

In an email to the investigators, the scientific 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 case study on modifications of laboratory instruments. Essentially, this model demonstrates the functionality required of the instrument to be sent into space.

Figure 2 illustrates this theoretical software development.

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.

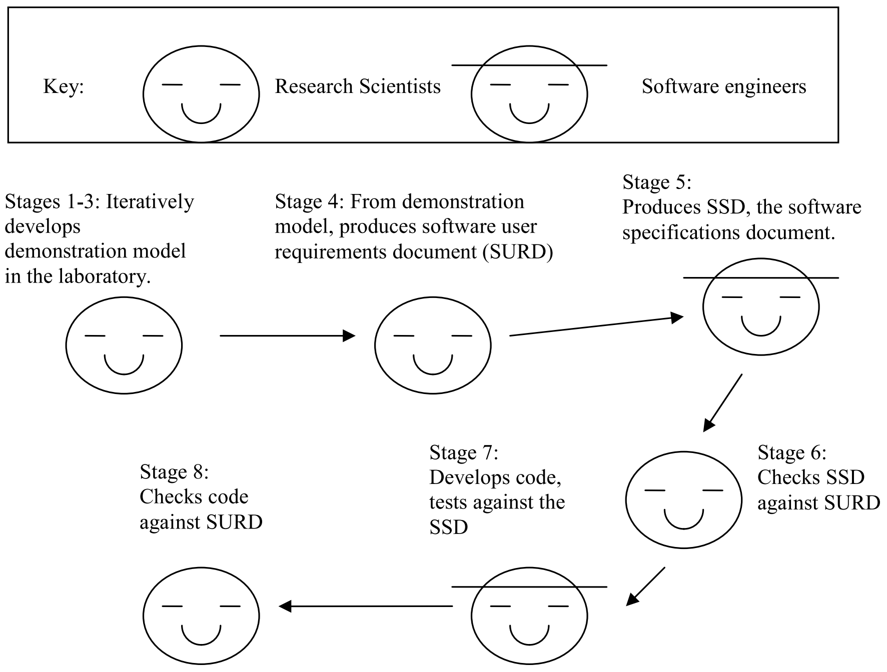
Key:
Research Scientists
Software engineers
Stages 1-3: Iteratively develops demonstration model in the laboratory.
Stage 4: From demonstration model, produces software user requirements document (SURD)
Stage 5: Produces SSD, the software specifications document.
Stage 6: Checks SSD against SURD
Stage 7: Develops code, tests against the SSD
Stage 8: Checks code against SURD

Figure 2. The theoretical stages of the software development as understood by the scientific project manager.

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 three research scientists, two from the LAB and one from the LABtoSPACE group, 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

> [LABtoSPACE scientist describing how he specified software for a close colleague to develop in the laboratory].

> 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..." [LAB 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!" [LABtoSPACE 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 software development knowledge is only possessed by the software engineers, and the final problem concerns the deployment of both the requirements and software specification documents as communication artefacts. We shall now look at 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 case study, there was a series of seven fully minuted meetings over the course of a year, attended by the scientists, the software engineers, and the electronics and engineering experts who were to develop the instrument hardware. The purpose of these meetings was to firm up both hardware and software user requirements documents (as in stage 4 of 3.1) and to determine the hardware components which would comprise the instrument. In the quote below, the scientific project manager describes how the software engineers helped the scientists formulate the software user requirements document, the SURD, in such a way that it was then easy to translate into a SSD, a software specifications document:

> 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"

As we shall now describe, the requirements and specification documents (together with, presumably, the minutes of the relevant meetings) failed to construct a complete shared understanding between the scientists and developers.

Neither the software engineers nor the scientists assumed domain knowledge to be the exclusive possession of the scientists. The software engineers were known to have much experience of developing software for space instruments, and thus 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 organisa-*

> *tion] 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 will touch on later in this paper, in Section 4.

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 (together, presumably, with other documents, such as the minutes of meetings) 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 case 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 seem to be the backbone of instrument/instrument software development in the laboratory, as evidenced by the quotes in 3.2. One scientist from the SPACE group, 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 next section, we discuss software methodologies in the light of the findings of our case study.

4. Software Methodologies and Our Case Study

In Section 3 above, we described the problems of using a traditional, staged, document-led software development methodology in the context of our case study. In this section, we consider other approaches to software development.

We shall firstly discuss the importance of tailoring a published methodology (the base methodology) to a particular context, and then briefly describe agile methodologies which we originally believed would provide the most promising options for a base methodology in our case. It became clear that we needed to combine agile with traditional elements; we go on to discuss how this combination might have been achieved, and, finally, discuss which agile elements might usefully have been adopted.

We begin by discussing the importance of tailoring a methodology to a context.

4.1. The Importance of Tailoring a Methodology to a Context

The findings of our case study focus on the differences between the traditional staged methodology as articulated by the scientific project manager in 3.1 and illustrated in Figure 2, and what actually happened. The fact that theory and practice diverged would come as no surprise to anybody who has reflected on the actual practice of software development. Such development takes place within a very rich context—sometimes referred to as an 'ecosystem'—and is influenced by many factors: the product which is being developed; the constraints under which the development is taking place; the individual characteristics, preferences and skills of the developers; the nature of the customers; the organisational culture, etcetera. Because of this rich diversity of context, many writers advocate that any software methodology cannot be used 'out of the box' (or, out of the textbook), but must be tailored to the project in hand. See, for example, the arguments advanced by Robinson et al. (1998), Cockburn (2002), Glass (2002) and Boehm and Turner (2004). These arguments are supported by empirical evidence, such as the survey conducted by Fitzgerald (1998). He found that of the software professionals surveyed, fewer than half used any published methodology, and of those who did, very few followed the methodology rigorously but rather tailored it to the project in hand.

Tailoring methodologies is an art. It helps if the base methodology—the published methodology which is to be tailored—has a close fit to the requirements of the particular project. Our case study findings point to the importance of emergent requirements and communication in our context. These findings led us to look for a potential base methodology in a class of methodologies, agile methodologies, which directly address the problems of emergent requirements and communication and which have aroused great interest recently.

We shall now briefly describe the underlying values of agile methodologies; how they support emergent requirements and communication, and how they grew out of existing practices focussing on iteration and communication between customers and developers. We then discuss the fact that the adoption of an agile methodology doesn't completely suffice in the context of our case study.

4.2. Agile Methodologies

4.2.1. A Brief Description

Agile methodologies are based on the principles laid out in 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;
- customer collaboration over contract negotiation. (see http://agilemanifesto.org/, accessed August, 2004).

All these values, and especially the first, support the emergence of requirements; the second and last values emphasise the importance of communication.

Examples of agile methodologies include the highly articulated and disciplined eXtreme Programming, XP, (Beck, 2000) DSDM (Dynamic Systems Development Methods), see http://www.dsdm.org/, accessed August 2004, and the Crystal family of methodologies (Cockburn, 2002). Each of these methodologies represents both a reaction against the prescription of traditional methodologies and an evolution of tried and tested practices in such software development approaches as rapid application development, prototyping and incremental development. For example, the discipline of XP includes established practices such as pair programming and test-led development: the novelty of XP is that it claims to meld these into a coherent, mutually supporting whole. DSDM, originating in the early 1990s, was developed by a consortium which had the explicit objective of unifying and developing the various existing Rapid Application Development (RAD) frameworks, see http://www.dsdm.org/tour/history.asp, accessed August, 2004. The family of Crystal methodologies was developed by Cockburn as the result of reflection son his own experience of tried and tested practices and interviews with other practitioners.

Agile methodologies support emergent requirements by means of customer feedback on iterative, incremental development. We were struck by the similarity between the following quote, from Beck (2000) and the scientists' view of requirements, as in Section 3.5. above:

> 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]

As to communication, one of the principles of the agilists is the following:

> The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. [http://agilemanifesto.org/principles.html, accessed August 2004]

That is, there is an emphasis on an informal oral, rather than a more formal written, tradition for communicating and sharing knowledge within the team. Boehm, 2002, expresses it thus:

> The main difference [between agile and more traditional methodologies] is that agile methods derive much of their agility by relying on the tacit knowledge embodied in the team, rather than writing the knowledge down in plans [p. 66]

It is widely acknowledged, however, that reliance on a knowledge base distributed in people's heads and communicated orally, can be dangerous. Potential problems include: the risk of losing such knowledge over time, and when people leave the project team; the possibility of ambiguities in what is presumed to be a shared understanding, and the fact that information is not available for external review (Boehm, 2002; Cockburn, 2002; Paulk, 2002). The externalisation of (some of) this knowledge in documentation might go some way towards alleviating these problems. One of the quandaries facing practitioners of agile methodologies, as discussed in Cockburn (2002) and by Scott Ambler, in http://agilemodeling.com/essays/agileDocumentation.htm, accessed August 2004, is to determine what documentation is really necessary, while at the same time avoiding over-documentation [described as endemic in the software industry, by DeMarco, in DeMarco and Boehm (2002) and as a 'pernicious problem' by Paulk (2002)].

4.2.2. Agile Methodologies and Our Case Study

On revisiting our case-study, we realised that taking as base methodology an agile methodology, with its focus on communication rather than conformance to specification, and then tailoring it, would not completely fit the bill. This is due to the fact that although our case study focussed on the communication interface between the research scientists and the software developers, there are two other interfaces, both invisible to the scientists, which we need to consider. The first is a set of interfaces between the hardware components of the instrument and the instrument software. The choice of components, and hence the determination of the interfaces, was decided in a somewhat iterative fashion in the same series of meetings which determined the software user requirements document. It should be noted that, unlike the research scientists who were in a different town, the developers of the instrument hardware were in the same building as the software developers, so any ambiguities or uncertainties could be negotiated relatively easily. The second interface is that between the instrument and the spacecraft's on-board management computer, enabling the passing of commands from the scientists to the instrument in space (telecommanding) and the passing of data from the instrument back to the scientists (telemetry). Here, the requirements specification was articulated upfront and remained stable over the period of development; in addition, it was critical that the specification be met. These are precisely the conditions which are consistent with the application of a traditional staged methodology. Hence we have one facet of the software development, that concerned with the telecommanding and telemetry, which seems an ideal candidate for a traditional methodology, and another facet, that concerned with eliciting and satisfying the research scientists' requirements, which seems an ideal candidate for agile methods.

Drawing heavily on Boehm and Turner (2004) we shall now discuss how agile and more traditional methodologies might be combined.

4.3. Melding Agile and Traditional Approaches

Boehm and Turner (2004) base their blending of agile and traditional methodologies on a risk-based strategy. Such a strategy is based on the following: for each process component, one has to consider whether it is riskier to apply more of the process component or to refrain from applying it; for each artefact, one has to consider whether it is riskier to be detailed or to refrain from detail. The following is a process component example: for non-critical software in a situation where time-to-market is a high priority, it is riskier to do very comprehensive testing than not to do it, given the risk that during the period of time devoted to testing, one's competitors might beat one to the market. As to artefacts, one example is that, in a situation where software is easy to change and requirements are volatile, it is riskier to have a detailed requirements specification than not, given the cost of rework involved in changing the specification at each change of the requirements.

Boehm and Turner articulate some common risks associated with a reliance on agile methods, and how such risks might be alleviated by plan-based elements (that is, elements of a traditional, document-led methodology). They also consider the converse: how plan-based risks might be alleviated by agile elements. An example of an agile-based risk is one they refer to as A-Scale. A-Scale is the risk inherent in using an agile methodology with a large team or on a critical project. In the former case, informal communication and coordination, and reliance on undocumented knowledge residing in people's heads, might be impossible; in the latter, conformance to documents and processes, antithetical to an agile approach, might be essential. A-Scale may be alleviated by the judicious use of documentation. An example of a plan-based risk is P-Change, the risk of rapid change when this has to be accompanied at some non-trivial cost by associated changes to detailed specifications. This might be alleviated by a consideration of the extent to which detailed specifications are necessary in a particular context, as discussed above.

Essentially, Boehm and Turner's blending process begins by determining whether the specific project is close to the agile or plan-based home ground. In the former case, the project has a small, talented team; the software is not life-critical; the environment (in particular, the set of requirements) is dynamic, and the culture thrives on 'chaos.' (We should note that the term 'chaos' is that used by Boehm and Turner; we would prefer the phrase 'exploration and innovation.') The planning home ground, on the other hand, involves more critical software; a larger team; a stable environment, and a culture thriving on order. If the project is near the agile home ground, Boehm and Turner suggest using an agile method with any agile risks mitigated by introducing plan-based elements, as we described in the case of A-Scale above. If the project is near the plan-based home ground, then they suggest using a plan-based methodology mitigated by agile elements where necessary, as in P-Change. If the project is in the middle, they suggest encapsulating the software architecture into an agile part and plan-based part. This is the situation in our case-study, with the development of the software concerned with eliciting and satisfying the research scientists' requirements, being the agile part, and that concerned with the telemetry/telecommanding, the plan-based. In the latter case, a waterfall-like methodology might fit the bill; in the former, in the light of our discussion of tailoring in 4.1, we do not focus on any particular agile methodology but rather consider which agile elements might have been useful in our context.

4.4. Introducing Agile Elements into Our Development Context

The first agile element which might usefully have been deployed in our development is, we suggest, pair programming. We alluded in Section 3.4. above to the fact that the project has a long time-scale: the instrument is delivered to the spacecraft; the spacecraft is launched; but then there is a long time interval before it reaches the point at which the instrument begins sending back serious scientific data. Knowledge of the instrument, the software and the science clearly has to be preserved over this interval. Documents are clearly essential to this preservation. But documentation has its limitations. There are problems with finding, and then comprehending, the precise information needed. For example, the software engineer in our case study, describing another project for which he found himself having to maintain software he had not himself developed, described the difficulties he had both with finding the relevant documentation, and then, having found it, with understanding the particular notation used. There are also problems with the amount of information which can be documented: the document writer has to make some (explicit or implicit) assumptions as to what information might be useful for the potential reader of the document. These, and other problems, lead us to believe that, despite the frailty of human memory, knowledge in people's heads can usefully complement information articulated in documents. However, as we mentioned in Section 4.2.1 above, knowledge in people's heads may be lost over the passage of time and when people leave the project team, as happened with the two software developers in our case study. This loss might be alleviated by pair programming. In our case study, the two developers and their project manager might have developed the code together in pairs, thus sharing knowledge between them and reducing the loss when the two developers left. To counteract any argument that paired programming is too costly, we should point to evidence in the literature, see, for example, Cockburn and Williams (2001) and 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.

As to improving the communication between the research scientists and the software developers, the ideal solution, we feel, would be for a research scientist to join in with the pair-programming. There are precedents for this in the practitioner literature. Bache, 2003, writes 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.

Wood and Kleb (2003) found that combining the roles of customer and pair programmer worked well in their context of developing a software testbed for some numerical algorithms at NASA.

We accept, however, that this ideal solution might not be practical. Research scientists may have neither the time nor the desire to devote much energy to software development (particularly since expertise in software development is not seen either as one of their core professional skills or as enhancing their chances of professional advancement). In addition, although the research scientists have some experience of developing software, they are not as experienced as the software developers and may not be familiar with the particular language and programming environment used, so there is a cost associated with training.

The less ideal, but still potentially very useful, solution is to have more informal (unminuted), regular, frequent, face-to-face meetings. We believe that the importance of such meetings in resolving ambiguities, and in constructing and maintaining a shared vision between the scientists and the software developers, should not be underestimated. We also believe that attending such meetings would not have imposed an unreasonable load on the scientists. We maintain that only one scientist would have needed to be present at each meeting, given the strength of community within the small LABtoSPACE group and hence the ease of sharing information within the group. We further maintain that this need not have been the same scientist every time. All the scientists in the group, barring the relatively inexperienced project programmer (see 2.2), would be recognised by Boehm and Turner (2004) as being CRACK customers (that is, Collaborative, Representative of the customer base; Authorised to take decisions; Committed to the development, and Knowledgeable of the scientific needs). Any one of them could thus have represented the group.

The above discussion addresses the sharing of knowledge and improving communication. The issue of emergent requirements is complicated by the fact that the case study involved embedded software and the associated hardware became available only very shortly before the instrument had to be delivered. We can only suggest that the advantages of an incremental, iterative development, described by Cockburn (2002) as ‘a critical success factor in delivering software’ [p. 190], outweigh the cost of constructing a simulation environment. Such an environment should enable the software to be developed and tested iteratively, provide opportunities for continual feedback from the scientists, and hence support the emergence of requirements.

5. Summary and Discussion

In this paper we describe a case study in which software engineers developed a library of software components for a group of research scientists. We identify two problems with the traditional, staged, document-led software development methodology used. These problems are that the methodology doesn’t support the emergence of requirements, and that project documents do not suffice to construct a shared understanding between the scientists and the developers. Reflecting on our case study, we then discuss whether a methodology consisting of a traditional staged (plan-driven) part and an agile part might have been tailorable to fit our context, and which agile elements might usefully have been deployed.

We now address the question which we broached in the introduction concerning the generalisability of our findings: of what interest is this case study to people who are not directly involved in it?

Firstly, we argue that the case study is of interest because of the differences it reveals between the articulated practices and tacit assumptions of a particular methodological framework, and the actuality of developing software within that framework. For example, the methodology articulated in Section 3.1. and Figure 2, has the scientists alone producing the user requirements document. In fact, the interviews reveal that the scientists and the software developers worked together to produce this document in a series of meetings, as evidenced by the first quote from the scientific project manager in Section 3.3. and confirmed in subsequent conversation with the software project manager. In addition, as discussed in Section 3.4, the SURD and SSD did not play the part in the scientists’ testing that was articulated in the methodology. As to the assumptions tacit in the methodology, we believe that one such assumption is that project documents suffice to construct a shared understanding, and our evidence is that this simply isn’t true. Without a detailed case study, the very existence of discrepancies between theory and practice, let alone any understanding of the nature of such discrepancies, might have gone unnoticed.

Secondly, we hope that the rich picture painted by our case study might evoke recognition in those readers who develop software for customers in similar circumstances, and inspire them to reflect on their own development practice. We expect that many (if not most) practitioners have experienced situations in which the communication between customer and developer left something to be desired. Following our discussion in Section 4, we suggest that such practitioners reflect on how they might improve their informal communication channels, perhaps by adopting agile practices, such as pair programming, or perhaps by merely changing where they sit, as suggested in Cockburn (2002). We expect that many (if not most) practitioners have experienced situations in which customers find it difficult to articulate an upfront statement of requirements and instead expect requirements to emerge. We suggest that incremental, iterative software development may be the only effective response to such expectations. We expect that many (if not most) practitioners have experienced situations in which one group of stakeholders, like the group looking after the spacecraft's infrastructure in our case study, needs conformance to a detailed requirement specification, whereas another group, like the scientists, prefers emergent requirements. We suggest that in this case, the software development be partitioned into a largely plan-based and a largely agile part. Our suggestions are intended to spur reflection: as argued in 4.1, any implementation of them should be tailored to fit the particular context.

Finally, we hope we have convinced the reader of the value of case studies in empirical software engineering. We suggest that without some understanding of the nature of, and reason for, the compromises made when software is developed in the real world within some methodological framework, there is little hope that empirical software engineering can achieve its primary goal, of improving software engineering practice. We do not wish to deny here the value of an empirical software engineer's reflection on his/her own practice experience. However, we think it goes without saying that case studies, such as the one described in this paper, in which understanding emerges iteratively from many sources of information, and in which the researcher is unfamiliar enough with the situation to probe taken-for-granted knowledge and assumptions, paint a much richer and more objective picture.

Acknowledgments

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. She would also like to thank the reviewers, whose insightful comments inspired her to restructure the original paper.

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

References

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*. Boston, MA: Addison-Wesley.

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

Boehm, B., and Turner, R. 2004. *Balancing Agility and Discipline*. Boston, MA: Addison-Wesley.

Bratthall, L., and Jorgensen, M. 2002. Can you trust a single data source exploratory software engineering case study? *Empirical Software Engineering* 7: 9–26.

Cockburn, A. 2002. *Agile Software Development*. Boston, MA: Addison-Wesley.

Cockburn, A., and Williams, L. 2001. The costs and benefits of pair programming. In: G. Succi and M. Marchesi, (eds.), *Extreme Programming Examined*. Reading, MA: Addison-Wesley.

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

Fitzgerald, B. 1998. An empirical investigation into the adoption of system development methodologies. *Information and Management* 34: 317–328.

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

Glass, R. L., Vessey, I., and Ramesh, V. 2002. Research in software engineering: an analysis of the literature. *Information and Software Technology* 44: 491–506.

Klein, H. K., and Myers, M. D. 1999. A set of principles for conducting and evaluating interpretive field studies in information systems. *MIS Quarterly* 23(1): 67–93.

Lanubile, F. 1997. Empirical evaluation of software maintenance technologies. *Empirical Software Engineering* 2: 97–108.

Lee, A. S. 1989. A scientific methodology for MIS case studies. *Management and Information Systems Quarterly* 13(1): 33–50.

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. In: A. Jedlitscha and M. Ciolkowski, (eds.), *Proceedings of the 2nd Workshop in the Workshop Series on Empirical Studies in Empirical Software Engineering*, pp. 99–108.

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

Segal, J. 2004. The nature of evidence in empirical software engineering. In *Proceedings of the International Workshop on Software Technology and Engineering Practice (STEP) 2003*, IEEE Computer Society Press, pp. 40–47.

Segal, J., Grinyer, A., and Sharp, H. 2005. The type of evidence produced by empirical software engineers. In *Proceedings of the Workshop on Realizing Evidence-Based Software Engineering, ICSE-2005*, http://portal.acm.org/dl.cfm

Wood, W. A., and Kleb, W. L. 2003. Exploring XP for scientific research. *IEEE Software* 20(3): 30–36.

Zelkowitz, M. V., Wallace, D. R., and Binkley, D. W. 2002. Experimental validation of new software technology. In: N. Juristo and A. M. Moreno, (eds.), *Lecture Notes on Empirical Software Engineering*, World Scientific Publishing Co, pp. 229–263.

Judith Segal is a Lecturer in Computing at The Open University in the UK, where she is a member of the Empirical Studies of Software Development research group. Her research interests include exploring the questions of how software engineers might best support professional end user developers (that is, end user developers from rich technical domains, such as mathematicians and scientists, who have no problems with coding per se), and how empirical software engineering might best inform software development practice.