Software Practice from the Inside
Ethnography Applied to Software Engineering

Kari Rönkkö

Blekinge Institute of Technology
Department of Software Engineering and Computer Science
Joshu asked Nansen: "What is the path?"
Nansen said: "Everyday life is the path."
Joshu asked: "Can it be studied?"
Nansen said: "If you try to study, you will be far away from it." (Rebs 2000)

To

Hilja, Keijo, Daniel and Johan
This thesis is submitted to the Faculty of Technology at Blekinge Institute of Technology in partial fulfilment of the requirements for the Degree of Licentiate in Software Engineering.

Contact Information:

Kari Rönkkö
Department of Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE-372 25 RONNEBY
SWEDEN
E-mail: Kari.Ronkko@bth.se
http://www.ipd.bth.se/kro
Abstract

Empirical methodologies have recently attracted increasing attention from the broader software engineering community. In particular, organisational issues and the human role in software development have been addressed. Qualitative research approaches have been identified as necessary for understanding human nature. One qualitative methodology which has become increasingly recognised in the software engineering community is ethnography. It is also the qualitative approach that is addressed in this thesis, i.e. ethnography in relation to software engineering. Ethnography emphasises the members point of view in an effort to understand the organisation of a social, cultural and technical setting. Until now, only a handful of ethnographic studies focusing on software engineering have been carried out in accordance with the original conception of ethnography; these studies have traditionally been performed by sociologists. The understanding and application of ethnography by software engineers differ from that of sociologists as it gives up the studied people's point of view in the analysis of data.

The thesis is based on two independent ethnographic studies where the 'inside' perspective which complies with the original understanding of the methodology is applied. Using these examples as a basis, the relation between ethnography and software engineering research is explored.

The objective of this thesis is to promote 'ethnographic knowledge' by giving an overview of ethnographic work within software engineering, presenting an original understanding of ethnography, comparing software engineers' understanding of ethnography with the original understanding of ethnography, demonstrating how the different implicit research attitudes of ethnographers and software engineers produce different research discourses, and finally pointing to an opportunity to combine ethnography, which contributes an ‘inside perspective’, with software engineering's need for constant improvement.
Acknowledgements

First of all, I would like to express my gratitude to my supervisor Dr. Yvonne Dittrich for her courage in taking on and supervising a thesis on such a controversial subject, also for her excellent coaching, support and feedback. I would also like to thank my examiner Prof. Claes Wohlin for his guidance, feedback and perspicacity in relating ethnography to software engineering. My thanks go to my colleagues in our research group: Christina Hansson, and especially Olle Lindeberg, whose frank and serious critique of my work has been much appreciated. I should like to thank the Department of Software Engineering and Computer Science for funding my research, and my colleagues for helping me to keep to my research focus.

I would like to thank Ericsson, especially Mattias Carlsson who made the SDL project study possible. My thanks go to the Software Engineering Program and its students in Ronneby for allowing me to perform a field study of a student project. I would also like to thank the People Computer Work Program for providing a sound theoretical basis to my research and for demonstrating the opportunities presented by multidisciplinary thinking. Special thanks go to Gunnel Andersdotter who been involved in my research through seminars, and to Björn Andersson, Dr. Berthel Sutter and Prof. Bo Helgesson who in a sense constituted the Program from my point of view at the time referred to.

I am grateful to Dave Randall whose help in the past three years with developing an understanding of ethnomethodology has been invaluable in enabling me to sharpen my theoretical reasoning.

I would also like to thank the Wittgenstein seminar group who helped me to develop an understanding of the relation between language and its use. My special thanks go to Jan Björkman who has patiently discussed this relationship with me for more than three years now.

Finally I would like to express all my love and gratitude to my wife Monica and my two sons Daniel and Johan, who have helped me to develop my current approach to work and research by providing the invaluable insight that life and living must form a whole.
Contents

**Introduction**

1. Background .................................................................................................................. 10
2. Outline of Thesis Argumentation .................................................................................. 11
3. Ethnography and Software Engineering ........................................................................ 14
   3.1 Ethnographic Research within Software Engineering ......................................... 16
   3.2 Social Science Ethnography on Software Engineering ....................................... 19
4. Research Contribution .................................................................................................. 21
5. Ethnography and Ethnomethodology ............................................................................ 24
   5.1 Ethnography versus Case Study Research ......................................................... 29
5.2 Requirement Engineering and Ethnography ............................................................. 31
6. Development Methods and Research Methodology ..................................................... 33
   6.1 Development Methods ............................................................................................ 33
   6.2 Research Methodology ............................................................................................ 35
7. Summary of Papers ....................................................................................................... 37
8. Future Research ........................................................................................................... 40
References .......................................................................................................................... 42

**Paper I: ‘Bad Practice’ or ‘Bad Methods’**

Are Software Engineering and Ethnographic Discourses Incompatible?

1. Introduction .................................................................................................................... 49
   1.1 Software Engineering ............................................................................................... 50
   1.2 Ethnography in Software Engineering ....................................................................... 51
2. Understanding the ‘Original Ethnography’ ................................................................... 53
3. Ethnographic Studies of Software Development ......................................................... 54
   3.1 Accepted Vulgar Competencies ............................................................................... 55
   3.2 Wait with Some Decisions Until a Later Stage ....................................................... 56
3.3 Should Requirements or Architecture be Developed First? ..................................... 56
   3.4 Creating and Constructing on the Same Occasion ................................................ 57
4. ‘Bad Practice’ or ‘Bad Methods’ .................................................................................. 58
5. Conclusion ...................................................................................................................... 60
Acknowledgements ........................................................................................................... 62
References .......................................................................................................................... 62
Paper II: Talking Design
Co-Construction and the Use of Representations in Software Development

1. Introduction ....................................................................................67
2. Representations in Software Development ....................................69
3. Bridge Talk ...................................................................................72
3.1 Background and Methods .............................................................73
3.2 The Setting ...................................................................................75
3.3 The Whiteboard as Lasting Common Ground .........................76
3.4 Enacting as a Resource in Communication .........................79
3.5 The Representation of the Results of the Design Meeting ........82
4. Co-Construction and Distributed Software Development .........83
4.1 The Importance of Co-Construction in Software Development ...84
4.2 Dynamics in Co-Operative Work .................................................85
4.3 Three Levels of Collaboration in Software Development ........86
5. Conclusion ....................................................................................88
Acknowledgements .............................................................................90
References ...........................................................................................90

Paper III: ‘—Yes What Does That Mean?’
Understanding Distributed Requirements Handling

Presentation of Paper III .................................................................94
1. Introduction ....................................................................................96
2. Requirements Engineering .........................................................97
3. The Project Studied ....................................................................100
4. Methods .....................................................................................102
5. Requirements in a Distributed Project ......................................105
5.1 Problematic Regarding Requirements ..................................105
5.2 Developing Solutions to the Requirements Problems ..........108
6. Conclusion ....................................................................................111
6.1 Is Completeness the Core Problem? ......................................111
6.2 Human and Social Issues through Descriptions ..................112
Acknowledgements ..........................................................................113
References ........................................................................................114
Paper IV: Plans and Accountability in Software Engineering

Presentation of Paper IV ................................................................. 117
1. Introduction .................................................................................. 119
2. Plans in Software Engineering ...................................................... 120
3. CSCW, Plans and Situated Action ................................................. 122
4. Studies of Software Engineering .................................................. 128
5. The SDL project and Observation ............................................... 129
6. Practical Problems and Practical Solutions ............................... 133
   6.1 ‘Yes ... What Does That Mean?’ - The Practical Problems of
       Requirements Work ............................................................... 133
   6.2 ‘I Mean, I Can’t Understand Them ...’ - Practical Solutions to
       Practical Problems ............................................................... 138
7. Conclusion .................................................................................... 141
8. Acknowledgements ...................................................................... 145
References ....................................................................................... 145
Introduction

1. Background

Today there is an increased acceptance of methodological diversity in software engineering as a result of a change in orientation towards real industrial problems. This change in orientation is partly explained by the fact that industry has resisted much of the research performed thus far within software engineering. In this context, empirical studies have achieved recognition in the broader software engineering community, and a need for qualitative research methodologies has been identified.

Ethnography is one qualitative method that has recently gained recognition in the software engineering community. Ethnography has its origins in anthropology and sociology. With the recognition of ethnography in software engineering it has become clear that the software engineering community lacks knowledge of this methodology. This thesis has a special contribution to make in this area by providing the software engineering community with further knowledge of ethnography. This knowledge is achieved by combining applying of the sociological and anthropological ethnography with a survey of literature in the areas of sociology and software engineering.

In this thesis it is recognised that ethnographic studies performed in and by software engineers differ from the current sociological and anthropological understanding of ethnography. The implications of this discovery are discussed in this thesis. In this thesis, sociological and anthropological ethnography is referred to as the ‘original’ understanding of ethnography, bases on Anderson's (1997) identified ‘mainstream’ historical view of ethnography. According to Andersson, the native's point of view is the heart of the ethnographic enterprise. Ethnographic methodology thus emphasises an ‘inside’ perspective in order to understand the organisation of a social, cultural and technical setting.
This thesis suggests that there is a place for the original conception of ethnography within software engineering. Incorporated in this suggestion is the problematic issue that industry does not adopt much of the research currently carried out within software engineering.

2. Outline of Thesis Argumentation

As this thesis subject is not common within software engineering the presented thesis outline is deliberately placed on the level of an argumentation outline. This is thought of as a help to the reader who is not acquainted with the subject of ethnography and its difficulties in relation to improvement proposals, i.e. method development.

The argumentation is organised as follows. Empirical studies have recently achieved recognition in the broader software engineering community (Finkelstein and Kramer 2000). This recognition has resulted in an increased interest in both quantitative and qualitative methods within software engineering. In relation to the recognition of qualitative methods, one particular quantitative legacy has been highlighted. Within software engineering today there is a tendency to quantify qualitative results thereby entailing a risk that the original sense and thus the original understanding of the qualitative research methodology ‘ethnography’ may be lost in software engineering. This thesis incorporates ethnography into software engineering. Section 3.1 presents ethnographic studies performed by software engineering researchers, along with specific research efforts related to ethnography. A lack of ethnographic knowledge in the software engineering community has been identified. Section 3.2 presents some ethnographic studies of software engineers' work performed by sociologists. These latter studies are sociological investigative ones, i.e. they do not suggest methods or improvement proposals. The aim of this chapter is to present how ethnography is perceived in and by software engineers; it provides an overview of existing ethnographic research on software engineering.

Chapter 4 presents the thesis research contributions. This thesis contributes to the understanding of the qualitative research approach known as ‘ethnography’; it applies it in its original sense to empirical
studies of software engineers. In two of the papers, the analytic program ethnomethodology informs the ethnographical studies carried out (the term ‘inform’ is often used in social science to indicate where some concept or analytic pre-knowledge affects or influences the understanding of a research methodology or a studied phenomena, see Chapter 5). The application of the ethnographic approach along with a survey of the relevant literature has yielded the following results:

1. Experience-based knowledge about the research methodology.
2. Identification of how the research results differ from the ‘expected’ outcome and discourse within software engineering.
3. An overview of existing ethnographic research on software engineering.
4. Establishment of the fact that the software engineering community understands and uses ethnography in a quantitative manner which does not comply with the ‘original’ understanding of ethnography.
5. Suggestion that original ethnography be included as an accepted methodology by the software engineering community.

In Chapter 5, the terms ‘ethnography’ and ‘ethnomethodology’ are explained. The chapter shows how ethnography, and ethnomethodologically informed ethnography lead to the asking of questions resulting in answers not normally associated with discourses in software engineering. Case study research within software engineering is the research strategy that comes closest to ethnography. Case study research is used in Section 5.1 to exemplify some predominant methodological aspects of the software engineering discourse. In Section 5.2 it is shown how requirements engineering has developed a greater awareness of ethnography and ethnomethodology than the rest of the software engineering community. What can be learned from this greater awareness is described in Chapter 6.

The purpose of Chapter 6 is to tie together the earlier presented argumentation from the preceding chapters and produce a statement of methods. This chapter demonstrates that software engineering research has matured; this is shown by the increased orientation towards ‘real’ industrial problems. One consequence of this orientation is an increased
acceptance of methodological diversity in software engineering. This is a good time for presenting the qualitative research approach of ‘original ethnography’ to the software engineering community. An interdisciplinary discourse that has existed for a long time within the research community of Computer Supported Cooperative Work is also presented; this is the discourse of how to relate ethnography to design (in this thesis the term ‘improvement’ is preferred as, intuitively speaking, it is a wider term, which also includes management and test issues). Since no generally accepted solutions of how to relate ethnography and improvement have been presented thus far, it is clear that the problems which this thesis must confront are massive.

In Section 6.1, the understanding of ethnography in relation to requirements elicitation within requirements engineering is compared to the challenge that ethnography as a research strategy must confront within the area of software engineering. A potentially unbiased observational methodological positioning is recognised and has been utilised for ethnography in software engineering research. This ‘value free’ positioning is related to the fact that one of the main reasons for software engineers to go empirical is that to date industry has been sceptical towards taking on and applying much of the research results achieved thus far. In Section 6.2 it is suggested that the software engineering community implement a participatory research approach that incorporates an ethnographic element.

In Chapter 7, summaries are presented in the form of abstracts of the different papers which make up this thesis. In Chapter 8, suggestions are made for future research. The identification of future research is one important outcome of the research conducted. One main issue for future research is identified, namely to develop and try out a research approach beginning in the orientation of practitioners’ work as the common interest. Using the suggested research approach, which includes an ethnographic element revealing work practice from the practitioner's point of view, the results will grow out of the process of method development and implementation in co-operation with industry.
The introduction is followed by a presentation of the four papers included in this thesis. Since Paper III has been submitted to an interdisciplinary research community, and Paper IV is part of an interdisciplinary book, the two papers have been complemented by presentations which place them firmly and squarely in the software engineering community.

3. Ethnography and Software Engineering

Empirical studies have only recently achieved recognition in the broader software engineering community (Finkelstein and Kramer 2000). In particular, the human role in software development has been addressed. It is recognised, particularly by practitioners, that organisational problems need to be addressed and solved if progress is to be made in the field (Seaman 1999; Finkelstein and Kramer 2000). Organisations are complex and built of and by people. Due to the complexity of the task of understanding human behaviour a need for qualitative methods has been identified; statistical and other quantitative methods are not sufficient for this task (Seaman 1999). The advantage of qualitative methods is that the researcher is forced to deal with the complexity of the problem as opposed to reducing it to a mere set of abstractions. The results of qualitative methods are displayed as words and not abstract numbers. (Ibid.)

With the growing interest in empirical studies followed an increased interest in qualitative methods. Software engineering, on the other hand, is a discipline that from the very beginning favoured quantitative methods. This legacy explains how qualitative methods are used in software engineering research. Attempts have been made to convert the results of qualitative methods into quantitative results. In software engineering one can identify a wish to mix the qualitative and quantitative in two ways: the first is as two clearly separate investigations confirming or refuting each other, a triangulation of qualitative and quantitative investigations (Yin 1989 pp.6-13; Seaman 1999); the second is to divide the same investigation into two clearly separable stages, the first involving collecting qualitative data and the second a quantitative analysis of the very same data (Seaman 1999; Singer et al. 1998; Seaman and Basili 1997a; 1997b). The above
described tendency of requiring quantitative results as in the first but especially the latter case could be one reason why there are so few researchers in software engineering using and presenting ethnographic results as these were originally defined in the area sociology, i.e. few researchers accept this kind of qualitative results in their own right. With this legacy follows the risk that the ‘original sense’ of a qualitative method such as ethnography is lost or, what is worse, misunderstood (Sim et al. 2001) or ignored. One example of a ‘possible’ misunderstanding can be seen in a paper by Seaman (1999) where she advocates the above latter described mix of the qualitative and the quantitative:

In summary, coding qualitative information into quantitative data is often useful and even necessary, but must be done carefully (Seaman 1999 p.25).

...the alternative to data analysis (which unfortunately, is sometimes practised and even in published work) is to simply write down all the researcher's beliefs and impressions based on the time they have spent in the field collecting data. This alternative pseudo-analysis method is attractive because it is certainly easier than rigorous analysis, and most researchers feel that they "know" a great deal about the setting they have studied. But it is neither scientific nor reliable, and this practice is largely responsible for the scepticism about qualitative methods that is so prevalent in our field (Ibid. p.26).

This quotation from Seaman together with her earlier mentioned claim that the complexity of understanding human behaviour requires qualitative methods, and that statistical and other quantitative methods are, in fact, not adequate for this task, demonstrate the ambiguous attitude of the software engineering community; qualitative methods are crucial to understand software practice, but they must be supported by quantitative analysis.

The above notion of what constitutes ‘scientific rigour’, i.e. attaining rigour through quantitative methods or ‘independent laws’, is not the only way of being rigorous. Qualitative sciences have a way of their own that of achieving scientific rigour (see, Chapter 5)! The above
quotation indicates there might be a lack of understanding about the nature of qualitative methods such as those used in ethnography, something that might lead to both inadequate uses of such methods, as well as the inability to recognise a good ethnographic study when you see it. Sim et al. (2001) have, in fact, pointed out that the community of software engineering lacks this kind of ‘ethnographic knowledge’.

Ethnography is a descriptive research strategy focusing on people's ‘day-to-day work practice’. Ethnography in its original sense, together with the ethnometodological view, are explained in Chapter 5. In Section 3.1 which follows, ethnographic work within the software engineering community is identified.

3.1 Ethnographic Research within Software Engineering

The first papers published within software engineering to use the term ‘ethnography’ to describe the research approach adopted have not been published until very recently (Seaman and Basili 1997a; 1997b). In the same time period, Singer et al. (1998) identified a lack of research on software engineers’ ‘work practices’:

With respect to SE work practices though, we found very little that could help us. Eleven years ago at the First Workshop on Empirical Studies of Programmers, Bill Curtis posed the question "By the way, did anyone study any real programmers?" We could add an addendum to this question, "And by the way, did anyone study any real programmers really working on real programs?" Unfortunately, with little exception, the answer to this question eleven years ago was "no", and remains "no" today (Singer et al. 1998 p.1).

To the best of the present author's knowledge, the above statement is still valid. The few software engineering researchers who use the term ‘ethnography’ are:

- **Carolyn B. Seaman, University of Maryland**. Seaman uses qualitative methods such as participant observation and interviews, which are subsequently combined with processing and coding according to a predefined scheme, i.e. a quantitative method for analysis (Seaman 1999). One way of doing this is through *prior ethnography* (Seaman and Basili 1997a, 1997b).
This kind of ethnography is used in the process of producing *Actor-Dependency Models* (AD models). AD models are thought of as ‘including’ the qualitatively collected data in a form that allows for automating some of the analysis. The ethnographical feature of the process is considered to be that of becoming familiar with the practice before the data collection which leads to AD models is used. It is worth noting that it is the whole process used to produce the AD models that is known as *prior ethnography* (Ibid.). Seaman is also committed to the idea that the analysis of qualitative results can yield new theories or hypotheses which can then be subject to validation by formal experiment.

- **Janice Singer**, National Research Council, Canada; **Timothy C. Lethbridge**, University of Ottawa; and **Susan Elliot Sim**, University of Toronto. The lack of relevant literature has motivated these researchers to perform workpractice studies of professional software engineers. These studies have been made on individual software engineers with the aim of influencing the design of new tools (Singer et al. 1998). Lethbridge, Sim and Singer have developed the concept of ‘Software anthropology’ (Lethbridge et al. 2001). This concept is a set of techniques involving observing and analysing the ‘day to day work’ of software engineers. The idea of the approach is borrowed from both cultural and physical anthropology and includes studying the work practices of individuals and teams, as well as understanding how people work or lived by looking at their artefacts (Ibid.). Ethnography has its origins in cultural anthropology (Anderson 1997). The methods used in software anthropology rely heavily on field study techniques. Singer and Sim were also two of the organisers behind the International Conference on the Software Engineering 2000 workshop *Beg, Borrow, or Steal: Using Multidisciplinary Approaches in Empirical Software Engineering Research*, aimed at helping researchers learn about methods from other disciplines. Ethnography was one of four themes in the workshop.
• **Susanne Grinter, Xerox Palo Alto Research Center.** Grinter's research focuses on supporting the process of design for new innovative technologies. She uses field studies of people's work practices with the goal of creating an understanding of people's work, especially the kind of problems that arise in relation to incremental innovations of technologies. By emphasising the term ‘field study’ instead of using the term ethnography in her descriptions of the research methodology used she points to the fact that she allows herself to take an ‘outside’ view instead of the ‘inside’ one that follows with the original use of ethnography. The ‘inside’ view is explained in Chapter 5. Grinter has also performed interdisciplinary studies combining theories from sociology, *Computer Supported Cooperative Work* and software engineering. These studies have led to recommendations for structuring developers’ practices (Herbsleb et al. 2001; Grinter 1997; Gruder and Grinter 1995).

In summing, it can be said that it is only Seaman who has published a paper in software engineering using the term ethnography to describe the research approach adopted (Seaman and Basili 1997a, 1997b). The concept *prior ethnography* used by Seaman in these publications contradicts with the original conception of ethnography as she uses field methods to collect data for later numerical modelling. Lethbridge, Sim and Singer invented the term ‘software anthropology’, a research strategy very close to ethnography (Lethbridge et al. 2001). Both involve understanding the ‘day to day work practice’ and use the same kind of field methods to achieve understanding. Singer and Sim have also identified a lack of ‘ethnographic knowledge’ within the discipline of software engineering (Sim et al. 2001). Grinter does not use the term ethnography in software engineering publications; instead she emphasises ‘field study’ although she is well aware of the original meaning of ethnography. In fact, Grinter has been invited to comment on the subject of ‘ethnography and design’ in the journal *Computer Supported Cooperative Work* (Gruder and Grinter 1995).

It is also possible that some published case studies within software engineering come close to the idea of ethnography e.g. Sim and Holt have published a paper at the International Conference on Software
Engineering (Sim and Holt 1998) in which they use ‘case study’ to cover the method description; this is the very same paper that was subsequently referenced by Lethbridge, Sim and Singer as an example of ‘software anthropology’ (Lethbridge et al. 2001). Yin identified the fact that case study research has sometimes been confused with ethnography (Yin 1998 pp.6-11). A Scandinavian software engineering research group that is also relevant to the present discussion is Lars Mathiassens’ group from the University of Aalborg. This group carries out action research which includes observations; these are not discussed in this section as the present author did not find articles where Mathiassen’s group discusses its field techniques. Unfortunately, the above group also publishes most of its work in areas other than software engineering (Mathiassen et al. 2002; Mathiassen 1998; Nielsen and Nørbjerg 2001).

So far in this thesis it has been claimed that empirical studies have achieved recognition in the broader software engineering community; a need for qualitative methods has also been identified. The legacy issues of quantifying have influenced the introduction of qualitative methods. With quantifying follows a risk that the ‘original concept’ of qualitative methods such as ethnography might be misunderstood, or, what is worse, may go unrecognised or even be ignored. In software engineering, only two papers have been identified where ethnography is stated to be the research method applied. The present author discovered some software engineering research that could be related to ethnography; Section 3.2 describes ethnographic work carried out by social scientists outside the software engineering community.

3.2 Social Science Ethnography on Software Engineering

The author could identify only a handful of ethnographic studies in sociological research which focus on ‘software engineering’. Some researchers have studied tasks in software development such as the use of representations in the design of applications (Suchman 1990; Suchman 1991). These kinds of studies have been excluded from the present discussion since they do not relate explicitly to the software engineering process as a whole. Two authors who have related
ethnography and engineering are Button and Sharrock. They have published a series of five papers on this theme between 1994 and 1996; Newman also published such a paper in 1998.

- Four of Button and Sharrock’s papers are based on the same empirical study of software engineers whose task was to develop software for a photocopier (1994, 1995a, 1996, and 1997). The objective of the first study was to develop an understanding of what it means to follow a methodology designed to approach requirements analysis (Button and Sharrock 1994). The second article examines the relationship between the formal representations of software development as described in methodologies and the actual work of software development (Button and Sharrock 1995a). The third study comprises a survey of the literature available and related theoretical discussions. It is concerned with details of the screen-based work of writing computer programs and the document-based work of reading computer programs (Button and Sharrock 1995b). The fourth study is concerned with how software engineers organise their work as project work. (Button and Sharrock 1996). The fifth publication draws on the previous four studies and illustrates an ethnomethodological view of the complex articulation of the collaborative project format (Sharrock and Button 1997).

- Newman describes a study of the development of middleware in a company. She identifies twelve types of actors involved in the design process. These actors comprise a diverse and shifting group located both inside the company as well as in other companies. These actors form an interconnected web that changes over time i.e. different groups of different actors, each with their own set of concerns, make up the design together. This suggests that the design process described is in no way a linear one starting with goals and requirements and ending with a complete design. Instead, the design is done concurrently with the specification. Another important issue in the paper is how to define the place of ethnographic study, due to the distribution of the setting (Newman 1998).
Common to studies by Sharrock and Button is the fact that the performed ethnography is informed by ethnomethodology. The main contribution of the papers is the elucidation of the relationship between the formal representations of programmers' work which take the form of structures or methodologies used by programmers, and the actual situated and contingent practices of programmers using those structures or methodologies. Newman demonstrates that the design was done concurrently with the specification in the highly distributed software development context described in her paper. Sharrock and Button (1994, 1995a) are referred to in Paper I and Paper IV and Newman (1998) in Paper I and Paper III.

From a software engineering point of view, the contributions of the above-described research do not extend beyond the level of ‘better understanding’. There are no suggestions, better structures or methodologies presented to improve the situation of the practitioners studied. The following chapter describes the contributions of this thesis on a general level. It discusses the ‘balancing act’ by which the belief of the qualitative scientist in rigour can be catered for while at the same time satisfying the software engineering researcher's quantitative legacy and emphasis on improvement suggestions.

4. Research Contribution

In this thesis the ethnographic research approach has been applied as it was originally conceived (Chapter 5). In relation to software engineering and quantitative methods it could be expressed as; the research focus is on understanding work practices from the point of view of the people studied; this without forcing on method improvement or tool development, i.e. it avoids instrumentation of the research approach. In software engineering, improvement and tools development are at the heart of the community, often considered to be a separate and subsequent step of an initial study. As a consequence, the tension between the ethnographic contributions and software engineering expectations become clearly demonstrated in the present thesis. Two different tensions which have been identified between the ethnographic results and expected results within software engineering,
discussed in Paper I and Paper IV. Ethnography is applied to software engineering in Paper II-IV, with a special emphasis on the ethnomethodologically informed variant of ethnography in Papers III and IV.

One contribution of this thesis to ethnographic research outside software engineering is that it presents new ‘organisational ethnographic studies’ of software engineers' work practices. The contribution to software engineering is threefold. The first is to respond to the identified lack of ethnographic knowledge within the area of software engineering (Sim et al 2001). The possibilities and limitations of ethnography are explored by applying ethnographical principles in their original form. The second is that each of the studies contributes to a deeper understanding of software engineers work practice to the software engineering community (Papers II-IV). The third is to clarify how the ethnographic results differ from ‘expected research outcomes’ within the discipline of software engineering; different research methodologies result in different questions and thus different kinds of answers. As a result, different kinds of discourses are to be found in different research communities; published papers are expected to relate to these different discourses. This third contribution is accommodated for since ethnography reveals ‘an inside view’ of the people studied. In addition to the ‘inside view’ the use of the ethnomethodological perspective also leads to specific kinds of questions being asked. This is explained in Chapter 5, and discussed in Paper I and Paper IV.

A double-edged contribution which relates to both sides can also be discerned by identifying existing tensions that have appeared as a result of taking up the challenge of publishing interdisciplinary research of the kind conducted here. In Paper II an applied theoretical reasoning is provided in direct relation to the empirical material; this reasoning is still to be found in the paper. An orthodox ethnomethodologist reviewer has suggested that such ‘outside reasoning’ is absolutely unacceptable. A reviewer with a software engineering background has claimed that the very same section is too weak i.e. too few conclusions have been drawn and there is too little discussion about how to improve distributed development in direct relation to the empirical findings. The reason given was that software engineers want to read about things they
can use in their projects. The paper which is now Paper IV was commented on in an early version in the following way by two different reviewers. One reviewer remarked that the introduction makes valuable points but was troubled by the fact that the ‘plans’ in relation to the empirical analysis are an analyst’s construction, i.e. the discussion is not sufficiently focused on the ‘inside’ view of the project members studied. The second reviewer, who is from the software engineering community, suggested that the introduction should be excluded as it has obviously become bogged down in the existing confusion about plans which characterises the discipline of Computer Supported Cooperative Work. On the other hand, this second reviewer considered the analysis of the software engineers to be solid work. Of course, these contradictory criticisms have been taken out of their review context; nonetheless, they clearly indicate the difficulty of adopting an interdisciplinary approach. The criticised section in Paper II has been left in its original form so that the readers of this thesis can make up their own minds about the contradictory critiques. Paper IV has been modified to a large extent to comply with the more refined argument presented in this paper. Paper I is also related to this methodological discussion in the sense that it has been demonstrated that one existing tension in this discussion can be related to the application of ‘an inside’ contra ‘an outside’ view.

In the interdisciplinary research community Computer Supported Cooperative Work the problem of relating ethnography to design or improvement, and ethnomethodology to method development are parts of a longer discussion, though no clear, acceptable answers have been presented (Plowman et al. 1995).

Plowman et al. (1995) identified that several studies that are primarily concerned with revealing social phenomena, conceptual and theoretical concerns are sometimes dismissed with the comment ‘so what?’ from designers. The latter seems to be looking for studies performed which match a checklist approach. As a result, the long-term perspective of sharpening concerns, issues and central questions by presenting them as theoretical and conceptual contributions is endangered (Ibid.). This discussion within the area of Computer Supported Cooperative Work gives an idea of the extent of the challenge which this thesis presents,
i.e. it gives a sense of what it brings about in effort to apply the qualitative research approach of ethnography in the community of software engineering. The following chapter explains both ethnography and ethnomethodology.

5. Ethnography and Ethnomethodology

Ethnography is a research approach taken from sociology; and with roots in anthropology (Anderson, 1997). Ethnography is also a research approach, which means very little in one sense. Ethnography has been given various interpretations (Harper 2000; Shapiro 1994). If it is difficult to know what is meant by ethnography it is equally difficult to know what contributions ethnography can make, and how it can be related to other research methodologies. In the research community of *Computer Supported Cooperative Work* ethnography has a strong position in relation to system design. In this community, ethnography has become one of the key methods for studying work practice and its influence on system design. This thesis understanding of ethnography is for precisely the reason of ‘that it has been related to system design’, borrowed from *Computer Supported Cooperative Work*. This thesis uses the term ‘ethnography’ in the original sense of the word i.e. as defined by Anderson (1997), who provides a *mainstream* historical view of ethnography. Harper (2000) has complemented the original understanding of ethnography by suggesting a fieldwork program to accompany it.

Harper (2000) has identified three basic strands for the application of ethnography within *Computer Supported Cooperative Work*. The first strand is a set of papers that uses the term to label studies of work practices using close observation; this set of papers has a diversity of structures, if any, influencing the research approach. The second strand suggests how to use the approach to develop an analytical or theoretical program as ‘sociological conceptions of organisation’, ‘sociological interpretations of negotiation’ and ‘sociological views on conversational interaction’. One example of an analytical program is ethnomethodology (explained in latter paragraphs of this chapter). Shapiro (1994) has shown how ethnography can be made to serve virtually any analytical program. The third strand of commentary that
Harper (2000) has identified is those who remark on the unwillingness of ethnographers to make any serious attempts to specify design or improvement choices. The concern of this latter category's papers is not with the actual work that goes on in the work place studied, but with the motivation behind research undertaken (Ibid.; Grudin and Grinter 1995).

As ethnography is obviously ‘an open subject’ in one sense it is necessary to provide certain underpinning assumptions when defining what it is or is not. Harper (2000) has identified one such basic underpinning assumption: *it is a method for understanding what activities mean to the people who do them* (Ibid. p.244). This is what is referred to earlier in this thesis as the ‘inside’ view. If there is no such underpinning assumption, then it is not ethnography; rather it is some other kind of observation technique or fieldwork (Ibid.). A fuller description of the same underpinning assumption is:

*The main virtue of ethnography is its ability to make visible the ‘real world’ sociality of a setting. As a mode of social research it is concerned to produce detailed descriptions of the ‘workaday’ activities of social actors within specific contexts. It is a naturalistic method relying upon material drawn from the first-hand experience of a fieldworker in some setting. It seeks to present a portrait of life as seen and understood by those who live and work within the domain concerned. It is this objective which is the rationale behind the method’s insistence on the direct involvement of the researcher in the setting under investigation. The intention of ethnography is to see activities as social actions embedded within a socially organised domain and accomplished in and through the day-to-day activities of participants (Hughes et al. 1994 p.430).*

Ethnography is often perceived as generating ‘thick’ descriptive results. This is related to how the ‘inside’ view can or will let itself be described e.g. to know what the work of a configuration manager is about, one would need to grasp and collect examples of how configuration managers take part in their own organisation; the rationale behind their actions; be able to illustrate what the role means when acting it out in their daily work practice; to recognise what kind
of acts and accompanying results make the person(s) studied feel both awesome as well as satisfied in the role as configuration manager. Enumerating work tasks, responsibilities and pinpointing these on the organisational map is not enough. It is rather a question of understanding a specific person's own understanding of his/her role and related responsibilities connected to the studied person's own understanding of the contingencies that follow when acting out his/her role in the organisation. If all these are described, the result would be a 'thick' description.

Ethnographic studies are written in an ethnographic style using the personal pronouns ‘I’ and ‘we’, i.e. they have a particular voice and are written from a particular point of view. This form of writing is a consequence of the ethnographic emphasis on fieldwork experience and not fieldwork findings. In fact, ethnography is sometimes referred to as a way of writing up and not a way of finding out (Anderson 1997). The ethnographic way of ‘writing up’ is reminiscent of the hermeneutic disciplines, while the way in which software engineers write their studies often bears a greater resemblance to studies conducted in the natural sciences. In this way ethnography seems unscientific if viewed from the natural sciences, but ethnography does not turn to the natural sciences for its guidance; it turns to the hermeneutic disciplines (Andersson 1997).

(Fuggetta 2000) suggests that the software engineering community must keep in mind that empirical studies are the means, not an end. Indicating that the same scientific objectives can be achieved through different means (Ibid.). Having in mind that the software engineering community should not automatically disqualify as "non" scientific those efforts that are not based on statistical evidence and controlled experiments ... Some of the most important contributions in computer science were not based on empirical studies (as we define them today) and statistical evidence (Ibid. p.32). As a provocation he further claims that by today's evaluation criteria, ‘some of the most important contributions’ would probably not be considered scientifically valid (Ibid.). By accident, Fuggetta has own experience from what could be considered as ‘suspect’ scientific results. In a paper which discusses experiences from conducting process assessment, the authors
established that the study's most interesting results are a secondary effect stemming from the fact that during the interviews people were encouraged to talk freely about their problems, impressions and ideas (Cattaneo et al. 2001). Such discussions made it possible to identify major problems in one particular process (Ibid.).

Ethnographers’ personal experiences from the field are proof of the fact that ethnographers acquire knowledge not available to others who lack their personal experience. This is also the reason why many ethnographers' papers contain a high proportion of empirical material; the reader is invited to judge the plausibility of the conclusions drawn. The credibility of ethnographic work for the ethnographer is related to this ‘personal experience’ together with presentations of the field material and its conclusions to the people studied. Results are also confirmed by combining qualitative sources and methods. In this thesis, for example, the results of observations and interviews were compared, and the conclusions made together with the related field material were regularly presented to the people studied.

What then comprises the ethnomethodological element in ‘ethnomethodologically informed ethnography’? The answer is that over and above the ethnographic ‘inside’ perspective, ethnomethodology provides an analytic program. Ethnomethodology as a program is concerned with revealing what ‘people's methods available to them to make sense of their immediate social surrounding and thus take action in league with their colleagues’ are. The methods of interest for this program are the commonplace and more or less taken for granted routines by which social situations are collectively produced. (Coulon 1995 p.V). By paying attention to everyday commonplace activities and regarding them as phenomena in their own right this program gives rise to practical sociological reasoning (Ibid. p.2) Applying this approach to software engineering thus introduces the ‘practical reasoning of engineers work’ to the discipline of software engineering thereby starting a new discourse. New questions and answers arise e.g. as ‘what methods are available to software engineers as social beings when they make sense of their immediate surroundings and determine what action to take?’ In this thesis a face-to-face steering group meeting and interviews with ‘talk about’ the project members
work are studied in this respect, see Papers III and IV. Ethnomethodology is also much concerned with the warrantability of data. That is to say, what justification is there for arguing that any particular thing is ‘going on’ should be evident in the presented data.

It is no easy task to reveal this kind of knowledge about ‘people's methods’ with the aid of quantitative studies steered by a specific hypothesis. This is because the use of a hypothesis assumes that you already know what you are interested in, it is also due to the fact that this kind of research question is not a question to be answered by derivation. Instead exploratory observational research is needed. On the one hand an exploratory case study including observation and interviews could be adequate; on the other hand, it is difficult to justify why forty years of ethnomethodological experience and knowledge have been left out. Where there is, as in this thesis, an interest in software engineers' methods as social beings as a phenomenon in its own right, it seems reasonable to at least have an awareness of ethnomethodology and some of its accomplishment. This thesis has gone one step further in this respect by letting ethnomethodology inform two of the performed ethnographic studies (Paper III and IV).

The fieldwork program applied in this thesis in Papers II, III and IV consists of three stages. First, developing an understanding of the flow of information that was necessary for the success of the overall progress of the project and the subprojects, i.e. to reveal what kind of information, information channels and co-ordination techniques that took place between the different subprojects and the project as a unit in the accomplishment of their tasks. Second, going through what the project members themselves perceived that the researcher needed to know about their work in order to understand it. Third, comparing what the project members ‘would like to do’ or ‘think they actually do’ with what was actually observed. The observed understanding of the project members' work was then juxtaposed with what the project members themselves described as their actions. An overall understanding developed of what project members ‘perceived they did’ as opposed to what could be observed as taking place. This overall ‘picture’ was presented to the project members studied. This fieldwork program is basically the same as that proposed by Harper (2000).
The use of ethnographic and ethnomethodologically informed ethnography has lead to questions being asked which have resulted in answers not normally associated with discourses in software engineering. If relating to case study research that was mentioned above, it becomes clearer to what extent the methodologies and the methodological discourses they produce differ.

5.1 Ethnography versus Case Study Research

Both ethnography and case study research are well suited to handling ‘messy’ reality situations where the investigator has little or no control e.g. situations with more variables of interest than collected data points are capable of revealing. Both claim that they retain the holistic and meaningful characteristics of real life events. Both can be exploratory as well as explanatory. Both have a descriptive side. Both can use the same kind of evidence, and follow the same principles of data collection. What is it then that distinguishes the case study research view described here from the view of ethnography.

Yin represents the MIT School (Yin 1989), a school often referred to in software engineering publications. Yin claims that case study research benefits from the prior development of theoretical propositions to guide data collection and analysis. Yin also claims that ideally case study research should use six sources of evidence. These are handled through three principles of data collection. The sources of evidence are documentation, archival records, interviews, direct observation, participant observation, and physical artefacts. The three principles of data collection are: to use multiple sources of evidence, to create a study database, and to maintain a chain of evidence. By following the three principles of data collection the benefits of the evidence may be maximised at the same time as these principles are the formal procedures that ensure quality control during data collection. Case studies should not be confused with qualitative research as they can be based on any combination of quantitative and qualitative evidence. Case studies can even be limited to quantitative evidence. In short, for case study research, methodological quality assurance is a matter of following the method correctly. This is the strive even though some authors have identified that case study research cannot achieve the
scientific rigour of a formal experiment (Kitchenham and Pickard 1995). (Fuggetta 2000) has pointed at the fact that research methods are the means to an objective rather than the objective itself, meaning that case studies are capable of achieving the same scientific rigour through different means than the today within software engineering predominant 'statistical evidence and controlled experiments'.

If a study focuses on a single project it should be called a case study. If the study involves many projects or a single type of project replicated many times, it can be either a case study or a formal experiment. Formal experiments require appropriate levels of replication as well as subjects and objects chosen at random within the constraints of an experimental design. Case studies are usually research in what is the typical. Formal experiments are often research in the small (Kitchenham and Pickard 1995). Both good case studies and formal experimentation start with defining an hypothesis (Ibid.). Yin (1989) has pointed out that in a case study it is not absolutely obligated to start with a hypothesis. The hypothesis is perceived as help in ensuring that you can draw both internally and externally valid conclusions. Since case studies can avoid scale-up problems they are important for the industrial evaluation of software engineering methods (Kitchenham and Pickard 1995).

Ethnography is an approach which allows understanding of those studied from an ‘inside’ perspective, i.e. social actions are seen to be embedded within a socially organised domain that is accomplished in and through the day-to-day activities of the people under observation. Among ethnographers, ethics and acceptance are considered to be the crucial conditions that give access to areas that are otherwise kept hidden; in this sense the ethnographer is perceived as a guest who is allowed to enter the realm of personal work of the people studied. As already demonstrated in this chapter, some of the characteristics of ethnography are built into the choice of analytical program. Different analytical programs ‘produce’ different outcomes. In short, methodological quality assurance for ethnographers is to be found in the idea that the researcher understands and takes action in accordance with ‘an inside’ view, which is often affected by an analytical program. The resulting evidence takes the form of so-called thick descriptions.
To sum up, within software engineering the closest corresponding methodology to ethnography is case study research. An ethnographic study can be seen as case study research; a case study cannot, however, be seen as an ethnographic study. A comparison of the different usages of the two research approaches demonstrated that using case study research in the software engineering community contributes to a methodological discourse of ‘an objective stance’ by relying on the ‘correct following of the method’. Whilst ethnography contributes to a methodological discourse involving the asking and answering of questions from ‘the studied people's own point of view’. This means that ethnography leads to ask questions which result in answers not normally associated with discourses in software engineering. In the following section, the discipline within software engineering with the most developed use and understanding of both ethnography and ethnomethodology is presented.

5.2 Requirements Engineering and Ethnography

Within requirements engineering there is an ongoing discussion about how to relate anthropological and sociological methodologies such as ethnography and ethnomethodology to requirements elicitation and analysis. To the best of the present author's knowledge, requirements engineering is the discipline within software engineering with the most developed understanding of ethnography and ethnomethodology. This understanding has developed as the result of a need for requirements elicitation techniques to handle social and organisational requirements.

In Sommerville (2001) ethnography has a section of it own placed under Requirements elicitation and analysis (Ibid. p.135). Ethnography is described as an observational technique for understanding social and organisational requirements. The ideas behind this is that individuals are not always aware of the social and organisational factors that affect their work. As a result, people often find it difficult to articulate difficulties at work. An unbiased observer is needed to reveal the social and organisational factors involved. Ethnography is identified (Sommerville 2001; Nuseibeh and Easterbrook 2000) as the answer to this challenge. It is particularly suited to revealing two types of requirements: first, requirements derived from the way people actually
work rather than the way they ought to work according to process descriptions; and second, to requirements derived from awareness of other people's work activities, i.e. revealing second nature co-operation (Ibid.). It is also interesting to note that the ethnographic references used in (Sommerville 2001 p.136) are papers which have been published within the research community of Computer Supported Cooperative Work.

In (Finkelstein 2000) Nuseibeh and Easterbrook present a roadmap for requirements engineering (Nuseibeh and Easterbrook 2000). This roadmap presents an overview of current research in requirements engineering. It is pointed out that requirements engineering needs to be sensitive to how people understand and perceive the world around them, i.e. how they interact and how the sociology of their workplace affects their actions. Here requirements engineering needs to draw on cognitive and social sciences to provide a theoretical grounding as well as techniques for coping with the situation. Ethnography and ethnomethodology are identified as the observational contextual techniques for analysing interaction and collaborative work. Ethnomethodology is also identified as an ‘objective’ methodology. Requirements capturing is not value-free, it is theory driven and biased by its methods and paradigm:

For requirements engineers, the methods and tools they use dominate the way that they see and describe. In the extreme case, this shifts the problem of validating requirements statements to a problem of convincing stakeholders that the chosen representation for requirements models is appropriate (Nuseibeh and Easterbrook 2000 p.42).

Ethnomethodologists attempt to avoid the above problem by refusing to impose modelling constructs on stakeholders; instead they seek to apply ‘value-free’ observation of stakeholders (Ibid.). The ‘value-free’ observation is the same as ‘an unbiased’ perspective; the ‘studied people's own point of view’ is not imposed. The ethnomethodological analytical program is not engaged in politics; instead it concentrates on the methods available to the people studied for making sense of their immediate social surroundings and thus to take action in league with
their colleagues. It is also interesting that the references in the text (Jirotka and Goguen 1994) are to the very authors who publish ethnographic and ethnomethodological work within Computer Supported Cooperative Work.

Chapter 6 ties together the argumentation presented in the preceding chapters; it concludes with a method statement concerning the issue of ethnography and improvements in the software engineering community.

6. Development Methods and Research Methodology

6.1 Development Methods

Finkelstein and Kramer (2000) have identified an increased disciplinary maturity in software engineering research. The orientation of software engineering research has changed towards a more carefully orientation towards ‘real’ industrial problems. This has come about as a result of the significant resistance from practitioners towards adopting much of, though by no means all, of the overly complex and heavyweight research solutions presented to date (Ibid.):

*It has taken a long time for researchers to realise that we cannot expect industry to make very large big-bang changes to processes, methods and tools, at any rate without substantial evidence of the value derivable from those changes. ... It is readily observable that research that proposes new frameworks, methods and processes are not accepted without positive evidence that they are of use rather than simply airy and unfounded speculations (Finkelstein and Kramer 2000 p.6).*

There is a lack of knowledge within software engineering about the empirical and qualitative approach of ethnography (Sim et al. 2001), including the core issue for software engineering; how this relates to the issue of improvement proposals. As a result, ‘ethnography’ needs a presentation and placement in relation to the subject of improvement in order to be accepted within the software engineering community, i.e. to reach the starting point for ‘building on others work’.
When it comes to the issue of ‘ethnography and improvement’ it is fruitful to return to the earlier requirements engineering discussion. Requirements engineering perceives ethnography and ethnomethodology as an attempt to solve conflicting perspectives by means of its ‘value free’ or unbiased observation (Section 5.2).

Two important and basic elements in requirements engineering are the concern of interpretation and understanding users' terminology, as well as the concepts and goals of stakeholders. Questions arise with regard to the nature of truth and what can be known; requirements engineering also considers the difficulty of reaching agreement with or between stakeholders (Nuseibeh and Easterbrook 2000). The technique chosen affects what can be modelled and consequently also what a requirements engineer is capable of observing (Ibid.). If transferring the reasoning about conflicting perspectives and ‘value free’ or unbiased observation to the issue of empirical research, it leads to the following:

*For software engineers, the research methods and tools they use dominate the way that they see and describe. In the extreme case, this shifts the problem of validating empirical research statements to a problem of convincing practitioners that the chosen representations for research models are appropriate (fictive).*

This adapted version of the original text of Nuseibeh and Easterbrook (2000) corresponds closely to the inconvenience that Finkelstein and Kramer (2000) have described. This is the situation in which practitioners significantly resisted to apply scientifically researched results. The point is that a similar philosophical element to that identified in requirements engineering exists in software engineering research as well. If the argument is developed further, ethnography and ethnomethodology can be used in the same manner within software engineering research as in requirements engineering, i.e. as an attempt to introduce ‘value free’ or unbiased observation of practitioners. This does not solve any technical problems. Where it does help is that it shows ‘what needs to be solved’. This is one identifiable potential characteristic of ethnography in relation to improvement or development methods in the industrial setting under observation.
6.2 Research Methodology

In an engineering discipline, empirical research aims at identifying improvements, including in particular evaluation and validation of the new outcome in relation to the initial ‘state of affairs’. This means that software engineering researchers tend to have a strong practical fieldwork program of relating to the ‘objective stance’ which influence the design of technology, methods and tools supporting software development. When it comes to ethnography, there is little or no discussion of what a fieldwork program should consist of (Harper 2000). Instead, ethnography often has to rely on its underpinning assumption, or alternatively, on its underlying assumption together with an analytic program. As the underpinning assumption of ethnography is that it is a method for understanding what activities mean to the people who do them (Harper 2000), it must be concluded that ethnography by definition has nothing in itself to say about improvements: if the underpinning assumption is not met, it is not ethnography but some other kind of observational technique or fieldwork which is being employed (Ibid.). On the other hand, this does not prevent the ethnographic detailed analysis of work providing rich material on which to ground improvement decisions, as is obviously the case in requirements engineering and Computer Supported Cooperative Work.

Using ethnography to inform improvement is often considered to be time-consuming, local and context-related, although it does not necessarily have to be time consuming on every occasion it is applied (Hughes 1994). On the other hand, such a time-consuming application of ethnography involves less risk of ending up in the somewhat distanced situation of expecting ‘big-bang changes’ of industrial practitioners (Section 6.1; Finkelstein and Kramer 2000).

One methodological question which software engineering is confronted with is whether it is fruitful to adopt ‘original ethnography’, i.e. if it is fruitful to try to understand the development practice from within and to relate any conclusions reached to that perspective. In this thesis this question is described in the form of two different methodological assurance discourses exemplified by the case study approach that takes an ‘outside’ objective stance, and the ethnographic approach that relies on the underlying idea of an ‘inside’ stance or research perspective.
Different research methodologies result in different questions and different forms of answers. This gives rise to different kinds of discourses in the different research communities to which papers to be published are expected to relate.

Finkelstein and Kramer (2000) argue that the change towards empirical research was followed by an increased acceptance of methodological diversity in software engineering. This occurred at the same time as research was still expected to build on the work of others, i.e. there exists less tolerance for reinventing the wheel. Using existing standards and proven research contributions are considered to be the best form of research (Ibid.). This could be one explanation of the legacy issue identified in Chapter 3. It could also be a potential obstacle to accepting the ‘inside view’.

The participants’ perspectives are important, but without a rich body of general knowledge to relate to, efforts become local problem solving and not research. The duality between research and practice informing each other becomes lost. There are lessons to be learned from participatory design and evolutionary development. Method development could be organised in a co-operative way, with implementation and evaluation cycles, making long-term learning among practitioners and researchers possible (Dittrich 2002). With this co-operative strategy, the starting point for research would not be to develop a theory to be implemented; rather, the whole co-operative process would show what is suitable or not suitable. The research results would ‘grow’ out of the process of method development and implementation in co-operation with practitioners. And, as mentioned (Section 6.1) the strong resistance from practitioners to adopting research solutions (Finkelstein and Kramer 2000) would be addressed. In short, on this overall level, a methodological research approach is recommended, beginning with the ‘orientation of practitioners’ work’ as the common interest. This ‘common interest’ achieved by applying an ethnographic approach, for example, may be leading to an ‘unbiased’ view of what is to be developed.
The papers in the remainder of this thesis provide the first elaborate ground for the next step in the development of the suggested research approach. Paper I identifies and discusses a tension that follows as a result of applying ethnography as it was originally conceived to software engineering. Papers II, III and IV are all examples of the application of ethnography to software engineering. In Papers III and IV the analytic program ethnomethodology is informing the ethnographic studies.

7. Summary of Papers

The abstracts of the papers are presented below:

**Paper I: ‘Bad Practice’ or ‘Bad Methods’ Are Software Engineering and Ethnographic Discourses Incompatible?**

*Kari Rönkkö, Olle Lindeberg and Yvonne Dittrich*


**Abstract**

Organisational problems in industry have evoked increased interest in empirical methodologies in the broader software engineering community. In particular, the human role in software development has been addressed. Qualitative research approaches are identified as necessary for understanding human nature. The qualitative approach addressed in this article is that of ethnography in relation to software engineering. Ethnography emphasises the members' point of view in order to understand the organisation of a social, cultural and technical setting. Today, with very few exceptions, it is sociologists who have performed the majority of ethnographic studies of software development — but how useful are these studies to software engineers? Ethnographic studies present problems from the observer's point of view. One implication of presenting studies from an ‘inside’ perspective is that they lend themselves to being regarded as revealing ‘bad methods’, i.e. which do not work in complex work situations. Taking from a software engineering point of view it is just as easy to point to the opposite interpretation of ‘bad practice’, i.e. a bad application of existing methods. The objective of this paper is to promote ‘ethnographic knowledge’ by revealing the different implicit research attitudes of ethnographers and software engineers, and to point to possibilities which combine studies that contribute an ‘inside perspective’ on software method improvement.
Paper II: Talking Design  
Co-Construction and the Use of Representations in Software Development  

Yvonne Dittrich and Kari Rönkkö

Abstract
Software development, just like all design projects, entails visualising something that does not exist as yet. Representations thus play an important role: even if they only describe aspects of the software developed, they mediate the common design work. Using the evaluation of a short section of a design meeting, we give a new focus to ‘talking design’ in relation to documents and diagrams. The meeting was part of a distributed software development project. An inevitable result of the distribution of project work is the visualisation of co-operation efforts. The objective of this paper is to demonstrate one aspect of why distributed software development can easily result in communication problems. The common object of the design is not a given fact; it must be collectively constructed. It is demonstrated that software development as a collective construction process largely takes place on the level of co-construction. By making a detailed analysis of a design meeting, we show how such co-construction is managed with the aid of face-to-face dialogue, enactment and whiteboard drawings. The question is posed: if important parts of design are collectively constructed during such meetings, and in such a way, what are the implications for co-operation, coordination and division of labour with regard to software development projects?

Paper III: ‘—Yes What Does That Mean?’  
Understanding Distributed Requirements Handling  

Kari Rönkkö

This paper is revised version of a chapter in Social Thinking - Software Practice 2002, editors. Dittrich, Y., Floyd, C and Klischewski, R. MIT Press.

Abstract
Requirements engineering is a process comprised of the requisite activities for creating and maintaining requirements documents. Different documents are produced at different stages of the development process. In most systems, requirements change. People develop a better understanding of what they want to do; companies and projects reorganise; when people change positions their successors may have a different understanding of problems and new ideas; modifications are made to hardware, software and organisational structures. Requirements engineering is concerned with managing changes in requirements. This paper describes requirements problems in a distributed software development project and describes them from the
project members’ ‘own point of view’. The research methodology used is that of empirical and qualitative ‘ethnomethodologically informed ethnography’. It results in a discussion divided in two parts. The first part is concerned with the precise benefits and drawbacks of using the methodology. The second part illustrates how the central problem of requirements engineering is not completeness but the production of collaborative theory building and mutual intelligibility. This central problem is compared with one particular requirements engineering perspective presented in a book where ethnomethodology and requirements engineering are discussed.

**Paper IV: Plans and Accountability in Software Engineering**

*Kari Rönkkö, Dave Randall and Yvonne Dittrich*

Submitted to the Journal *Computer Supported Cooperative Work*

**Abstract**

Based on empirical material from the area of distributed software engineering, this article discusses the issue of plans and planning in the *Computer Supported Cooperative Work* community as contrasted with the approach taken in software engineering. It suggests that insights concerning the nature of ‘plans’ and ‘planning’ in *Computer Supported Cooperative Work* are valuable insofar as they draw attention to features of the distributed software engineering process largely overlooked by the software engineering community—features of particular significance for distributed software engineering projects. The analysis of the empirical material allows us to show how particular kinds of problem arise and are dealt with in one project, problems that are best thought of as problems of ‘organisational memory’. We suggest that developing support for software engineers depends on recognising their practical problems and that this in turn requires a turn to qualitative empirical studies that they have been hitherto unwilling to make.
8. Future Research

The identification of further research is one important outcome of performed research. The performed research in this thesis has identified difficulties and a possibility of relating the ‘original’ understanding of ethnography to the software engineering community. The picture below gives an overview of this thesis contribution together with a direction of future development work.

Figure 1. Overview of this thesis contribution and suggested future research.

1. Has been applied on SE
Demonstrates:
• A tension between the ‘original ET’ and the existing SE research view
• Identified that existing ET studies in SE have a different research view
• A potential ‘inside’ or ‘unbiased’ empirical research approach

2. Future Research
How can ET complement existing SE research?
• Develop Research Methodology
• Develop Development Methods
In figure 1 a rectangle is framing the scope of the licentiate thesis, and it shows how software engineering and the ‘original ethnography’ today are separated from each other. Through my own applying of the ‘original ethnography’ within software engineering a first attempt to bring them together is made (1). Identified results from this work of ‘bringing together’ are:

- A tension between the ‘original ethnography’ and software engineering exists in the sense that the former relates to hermeneutic disciplines and the latter to natural sciences.
- Existing ethnographic studies in software engineering have a different research view, including coding of qualitative information into quantitative data. In this manner the qualitative information is made to relate to the prevailing tradition of ‘objective’ evaluation and validation within software engineering.
- The ‘original ethnography’ is suggested to have a place within software engineering as a potential ‘inside’ or ‘unbiased’ empirical research approach.

Future research is suggested (2). The advantage of the ‘original ethnography’ lies in the ‘sensitising’ of social phenomenon, i.e. to understand real-world character of activities in context. This understanding starts from the studied people's point of view. To understand such ‘social phenomenon’ provide context related possibilities to identify weaknesses and hence areas for improvement. In relation to ‘original ethnography’ there exist a need to develop and try out a research methodology and development methods that fit to software engineering. Some inspiration and experiences in this challenge might be found in (Hughes et al. 1994; Randall et al. 1994).

The suggested main issues for future research concern the structuring and organising of a research approach beginning with an ‘orientation of practitioners’ work’ as common interest for both researchers and industry, with a serious ethnographic ‘unbiased’ part. This means to develop further concepts and structures for the usage of ethnography in relation to software engineering research, this as a co-operative effort together with industrial partners. The ‘inside view’ following from the
application of the ‘original ethnography’, could be perceived as a promoter for accepted research solutions, addressing the ‘development methods’ problem of significant resistance from industrial to adopt much of the performed research solutions’ (Chapter 6). Methodological improvements would be developed based on the inside perspective revealed by ethnographic studies in co-operation with the practitioners. In such future co-operative strategy the starting point for the research would not be to develop a theory to be implemented, instead the whole co-operative process would both lead to adequate research questions, as well as to demonstrate what fits or not. In short, both a development method and research methodology including the ‘inside’ perspective have to be made concrete and tried out together with an industrial partner. Other related issues for future research are:

- To clarify how and ‘exactly’ at what points the reached scientific rigour from the hermeneutic discipline (that the ‘original ethnography’ relates to) differs from the rigour of the natural sciences (that software engineering relates to). Feyerabend (1993) might be one controversial inspiration source.
- In relation to this, formulate and state by which means the ‘original ethnography’ can provide an own acceptable evaluation and validation criteria within software engineering.
- To try out ‘original ethnography’ by starting in compliance with the prevailing tradition within software engineering, i.e. to start with a defined problem definition or hypothesis, and approach the problem through applying the ‘original ethnography’.

References


‘Bad Practice’ or ‘Bad Methods’
Are Software Engineering and Ethnographic Discourses Incompatible?

Kari Rönkkö, Olle Lindeberg and Yvonne Dittrich


Abstract
Organisational problems in industry have evoked increased interest in empirical methodologies in the broader software engineering community. In particular, the human role in software development has been addressed. Qualitative research approaches are identified as necessary for understanding human nature. The qualitative approach addressed in this article is that of ethnography in relation to software engineering. Ethnography emphasises the members' point of view in order to understand the organisation of a social, cultural and technical setting. Today, with very few exceptions, it is sociologists who have performed the majority of ethnographic studies of software development -- but how useful are these studies to software engineers? Ethnographic studies present problems from the observer's point of view. One implication of presenting studies from an 'inside' perspective is that they lend themselves to being regarded as revealing 'bad methods', i.e. which do not work in complex work situations. Taking from a software engineering point of view it is just as easy to point to the opposite interpretation of ‘bad practice’, i.e. a bad application of existing methods. The objective of this paper is to promote 'ethnographic knowledge' by revealing the different implicit research attitudes of ethnographers and software engineers, and to point to possibilities which combine studies that contribute an 'inside perspective' on software method improvement.
1. Introduction

How can or shall software development be best understood? Software development today is by no means a coherent or unanimously agreed upon concept with clearly defined methods or process models for practical applications. Research methodologies to for understanding software development are chosen depending on the problem at hand. The orientation of software engineering research has recently changed towards real industrial problems, and an increased interest in empirical methods has followed (Finkelstein and Kramer 2000). This implies an increased acceptance of a diversity of empirical methods (Ibid.).

Ethnography is one empirical and qualitative methodology that has attracted interest (Sim et al. 2001). It contributes an understanding of the social aspects of everyday working life from the members' own points of view. The organisation of social or cultural settings is described from an inside perspective in the continuing; this paper adopts the term ‘inside view’ throughout. The authors have identified a lack of ‘ethnographic knowledge’ within software engineering (Ibid.). Today, with few exceptions it is sociologists who have performed the majority of ethnographic studies on software development. But how useful are these studies to software engineers? Ethnography has a history within sociology which carries with it certain basic assumptions. These assumptions or underpinnings have implications for the research attitude of the ethnographer. The implicit assumptions of ethnography are important if ethnography is to be adopted and used in software engineering. This paper has a valuable contribution to make to the community of software engineering by revealing attitudes that follow with the original understanding of the methodology ethnography. It demonstrates how ethnographers' attitudes clash with the attitudes of software engineers. In this paper, the ‘original’ understanding of ethnography is used interchangeably with Anderson's (1997) historical mainstream description of ethnography. Anderson's definition is summarised in one sentence by Harper (2000), see Chapter 2. The present article further points to possible ways to address this clash and to make use of ethnographically informed studies in Software Engineering research. First, the methodology is positioned in relation to research issues in software engineering.
1.1 Software Engineering

Since the founding of Software Engineering as a discipline in 1969 it has focused on the effort to re-design practice in accordance with the model of other engineering disciplines. Software engineering as a research discipline has focused on methods for product development, it has been successful in developing programming concepts, analysis and design methods, and process models widely used today. Parallel with this development a development of programming environments, CASE- and modelling tools, document repositories and configuration management systems have been produced. Despite the success of this software engineering development, many of the hitherto accepted research results have gained little acceptance in industry. Most researchers acknowledge that practice in industry does not comply with much, by no means all, of currently available research results (Finkelstein and Kramer 2000).

While standard concepts, process and method guidelines were initially missing, today the problems have shifted …we must pay attention to the complex interrelation of a number of organizational, cultural, technological, and economical factors. (Fuggetta 2000 p.28). Both software engineering researchers and practitioners have expressed the need to address and solve organisational problems if the software engineering field is to advance (Finkelstein and Kramer 2000; Fuggetta 2000; Seaman 1999).

This introduces new problems since organisations and cultures are complex and built by and made up of people. How can organisations and cultures be studied? Seaman (1999) has suggested that, due to the complexity of understanding human behaviour, qualitative methods are needed; statistical and other quantitative methods are not adequate for this task. Qualitative methods have the advantage that the researcher is forced to deal with the complexity of the social aspects. The results of qualitative methods are produced in the form of words, not numbers which by means of abstraction has obscured the complexity of mundane software practice (Ibid.). Ethnography is one such qualitative methodology that has gained some recognition in software engineering. One example of this recognition is that at the International Conference of Software Engineering 2000 a workshop was arranged entitled Beg,
borrow and steal: Using Multi-disciplinary Approaches in Empirical Software Engineering Research (Sim et al. 2001). The workshop accepted twenty-three papers, of which one quarter fell under the category of ‘ethnography’.

1.2 Ethnography in Software Engineering

This section demonstrates different understandings of ethnography, as related to different research attitudes in software engineering. Three different software engineering approaches ‘relating to ethnography’ are presented in this section. The first is related to the idea of collecting qualitative data in an initial research phase that is subsequently quantified in the analysis phase (Seaman and Basili 1997a; 1997b). This does not qualify as an ethnographic study in its original sense. The second view takes its inspiration from the same roots from which ethnography originated (Lethbridge et al. 2001). The third view is to relate software engineering to the original understanding of ethnography. Among the studies already carried out which address ethnography in its original sense can be mentioned (Dittrich and Rönkkö 2002; Rönkkö 2002a), and discussions of how to relate the original understanding of ethnography to software engineering include (Dittrich 2002; Rönkkö 2002a and 2002b).

The two studies by Seaman and Basili (1997a; 1997b) are examples of improvement-oriented ethnographic research as well as of the tradition of quantifying research results. These two studies are the only ethnographic studies found that clearly use the term ethnography in terms of methodology too; prior ethnography ¹ This way of understanding and using ethnography differs from the ‘mainstream’ use and the general understanding of ethnography within sociology (Andersson 1997; Harper 2000) as it is directed towards processing qualitatively collected data according to predefined schemes instead of as it is opposed to producing descriptive inside view of the studied practices.

¹ The authors might have missed some articles. It seems likely that the authors have found most of the articles on the subject as no further studies are referenced in the literature.
A study by Lethbridge et al. (1997) seems to contain an initial ethnographic field study, even though the publication is not a descriptive ethnographic contribution but rather a quantitative presentation. Lethbridge et al. (1997) call the qualitative study a ‘work practice’ study, i.e. not an ethnographic study. They divide the study into two separate parts: a qualitative field study and a quantitative analysis aiming at improvement proposals. It is not clear whether the initial work practice study takes on the original ethnographic stance of an inside view or not. Obvious, according to the ethnographic definition (Harper 2000) the second part of this paper is not ethnography; i.e. the quantitative analysis together with its numerical presentation of the results does not qualify as ethnography in its original sense. It would have been interesting from a methodological point of view if the authors would have presented whether and how it is ethnography in its original sense that have influenced the scheme for the quantitative analysis. Lethbridge, Sim and Singer (2001) have also suggested a research approach influenced by the same roots as that of ethnography.

Based on a set of smaller studies (Dittrich and Rönkkö 2002, Rönkkö 2002a) the authors explored how to apply ethnography in a more original sense as part of software engineering research. Based on this experience, a research approach has been suggested which has been tentatively called cooperative method-development. (Dittrich 2002) Ethnographical studies provide the ‘inside view’ as input to identify and develop possible improvements together with the practitioners involved. This way of relating to the original ethnography will be further explored in the conclusions.

The above described literature survey of software engineers’ ethnographically influenced approaches shows that software engineers’ tend to have a common conviction that the analysis even of even qualitative data benefits from quantitative analysis according to predefined schemes. This mathematical way of analysing is inconsistent with the original understanding of ethnography. The difference in the software engineers’ method of analysis revealed by the authors implies a different kind of research result to that of the original sociological one.
What then is the original sociological understanding of ethnography? What would be the consequences of presenting ethnographic studies in the original understanding of the term to software engineers? The first question is answered in the Chapter 2. In Chapter 3 examples of ethnographic studies are provided. The question of consequences is discussed in Chapter 4, highlighting the different interpretations leading either to ‘bad practice’ of methods in use or ‘bad methods’ not suited for real world circumstances. Chapter 5 concludes this paper; two ways of benefiting from the original understanding of ethnography in the field of software engineering are suggested.

2. Understanding the ‘Original Ethnography’

Ethnography originated from the need to study foreign cultures (Anderson 1997). It was Malinowski who in 1915 invented the professional stranger, the ethnographer. For Malinowski, the purpose of fieldwork was to become intimately familiar with the setting studied. This was done by learning language and culture, and by living according to studied setting's regime. This intimacy leads to the subjective understanding that is necessary to integrate and correlate the found data in the form of synoptic charts, detailed descriptions of day-to-day activities and narratives. The idea behind creating such ‘ethnographic accounts’ was that things are not always what they seem to be, i.e. appearances are not the whole story. The native or participant him/herself did not always have the ‘overview’ that the ethnographic account could present (Ibid.).

So far it can be concluded that the ethnographic account is a post hoc representation of the field studied. Obviously, it is also a representation or account that involves interpretation and analysis from the very first stage. This personal way of executing research is legitimated by the ethnographers' ways of ‘knowing’ what others do not and cannot know as they lack the personal field experience of the ethnographer. This is the ethnographic qualitative warrant or scientific trademark; the ethnographer writes with a voice and from a point of view. This approach seems unscientific if viewed from the natural sciences. But this way of thinking does not turn to the natural sciences for its
guidance; it turns to the hermeneutic disciplines (Andersson 1997). This also explains why it is the fieldwork experience, and not the fieldwork findings in themselves which is of greatest importance. Historical ethnography is often described in basic textbooks as a body of procedures and techniques that anyone can learn and apply. This has been questioned; is it the following of the procedures and techniques that actually leads to an adequate ethnographic account? As consequence of this doubt, the techniques for identifying, listing and integrating activities from the studied peoples world are not much emphasised anymore, instead ‘the native's point of view’ has been identified as the heart of the ethnographic enterprise (Ibid.). For a more extensive review of ‘mainstream ethnography’ from its origins to the present day, see Anderson (Ibid.).

Harper (2000) has described ethnographic method based on one basic assumption: it is a method for understanding what activities mean to the people who do them (Ibid. p.254). If this assumption is not met it is not ethnography, it is some other kind of technique or field method. From this assumption also follows that an ethnographic account is a descriptive one. This is because the account needs to present the circumstances of the phenomenon studied: their life, their meaning, their purpose and point, i.e. the things that give the observed conduct the meaning it has. If one does manage to describe all these aspects, then one has a so-called thick description. One problem in producing such descriptions is that some activities are so ordinary and mundane that it becomes difficult to know what exactly needs explanation. The suggested solution to this problem is to keep to the idea of presenting the world as perceived by those within that world. (Ibid.).

3. Ethnographic Studies of Software Development

What can the original descriptive ethnographic research approach contribute to software engineering? One way to answer this question is to look at existing ethnographic studies of software development practice. The following demonstrates small sections of larger studies that have software development as a common theme. The sampling of papers could be labelled ‘convenience sampling’ (Robson 1997, p. 141); the set of studies we discus is not intend to cover the entire field
of software development. Neither would presenting those studies contribute to the argument. From these small sections we demonstrate the different nature of the research findings presented in comparison with what is expected in software engineering. The studies captured in these smaller sections were originally carried out in accordance with aims which do not necessarily correspond with the way in which the studies have been applied in this paper. In order to shed some light on the different attitudes possible to apply to software development some of the studies has been reinterpreted in some parts and used in a way for which they were not intended. This reinterpretation is perfectly in line with one of the main points of this paper, namely that studies often lend themselves to different kinds of interpretations.

3.1 Accepted Vulgar Competencies

Button and Sharrock (1995) present a study of code writing practices in The Mundane Work of Writing and Reading Computer Programs. One of the conclusions in this paper is that in software engineering ‘the vulgar competencies of common-sense knowledge and practical reasoning’ are accepted as essential for software development. In fact, programs are written to format the code automatically in accordance with the computational process to support the application of these ‘vulgar’ competencies. A programmer who tries to understand a program uses any available information, jumping from high-level documentation to low level code reading, following functions or calls to other related parts. Software engineering methods that demand documentation and code comments have the unintended side effect of making the work that went into the writing of the code visible to others that later try to make sense of the program in order to change it. Hence, the ‘vulgar and common sense practices of writing and interpreting code’ have documentation support related to the code. The authors contrast this with the natural and social sciences' practice where the practical work that sets constraints on scientific results goes unrecognised, and with mathematics, where in the final product, -the proof-, there is no longer any visible trace of how the mathematician succeeded in finding the proof.
In software engineering there is a tendency to formalise the division of labour. It is not always clear whether the ‘vulgar practice aspect’ is taken into consideration, or not.

3.2 Wait with Some Decisions Until a Later Stage

In the same paper by Button and Sharrock (1995) the focus is on the detailed practices applied when writing code and not the overall development process. Nevertheless, these practices have implications for the validity of basic assumptions in software engineering. One example is the programmer’s practice of sometimes using temporary names for variables (e.g. temp-1) instead of immediately providing an intelligible name. This is because programmers are uncertain about how to name the variable until they have completed the major part of the program and the role of the variable has been clarified. They then change the temporary name to an intelligible one. The authors describe the activity of selecting informative names in terms of the programmer engaging in constructing a taxonomy of a semantic domain, for their practical and situated purposes. They exemplify by asking what to call the function of turning on a warning light on a photocopier: should it be called a fault indicator or a problem indicator? The correct name cannot be chosen without taking all other warning lights into consideration.

This observation presents a problem with one particular view of SE, the top-down view. The top-down strategy for program development is based on the idea that the program is broken down into parts that can be implemented independently of each other. The observations in the study make it painfully clear that naming is a global problem. The names of different conventions must be the same throughout the program to avoid confusion.

3.3 Should Requirements or Architecture be Developed First?

In her article Here, there, and nowhere at all Newman (1998) describes a study of the development of middleware in a fortune 500 company. The actors involved in the design of this software are a diverse and shifting group, coming from both inside the company and from other companies (she lists 12 types of actors). These actors form an interconnected web which changes over time. The main problem taken
up in the article is how to define the place where the ethnographic study should be carried out. This is not our concern here. However, it is interesting to note that the design is made by shifting groups of different actors, each with her/his own set of concerns. This suggests that the design process described is in no way a linear one starting with goals and requirements and ending with a complete design. Instead, it is done concurrently with the specification.

From a software engineering point of view, the situation described does not at all fit into one ‘desirable framework’ where the development starts with requirements analysis. What Newman is describing is a situation where the internal technical design of the middleware is constructed at the same time as the description of what the middleware will actually do. For a software developer this is not very surprising, the architecture of the middleware will make some things easier and other things nearly impossible, it is not possible to make the tradeoffs necessary before the technical design is conceptualised. Newman also describes a situation where there is no clear customer but a whole set of actors with different agendas.

3.4 Creating and Constructing on the Same Occasion

In their article *Artificial intelligence as craftwork* Suchman and Trigg (1993) concentrate on the use of whiteboard sketches. One conclusion is that the work of AI involves a series of transformations or re-representations, starting with observations of the social world and ending in executable code. In the article, a design talk between two AI researchers is analysed. The AI researchers use a whiteboard to sketch on and use gestures to show the dynamics. The figures on the whiteboard are seen as both representing a logical formalism and a Lisp construction. They do not clearly differentiate between these levels but make reference to the level most relevant for the moment.

One issue in this paper that is interesting in relation to software engineering is stepwise formalisation. In a sense, it is possible to conclude that the AI researchers follow a traditional stepwise formalisation model since they do not start writing the actual code until
the formalism is elaborated. In another sense, they completely break with the stepwise formalisation model since they allow reasoning on code level to settle how the formalism should look.

4. ‘Bad Practice’ or ‘Bad Methods’

By comparing the improvement oriented software engineering view with the presented ethnographic cuts, contradictory views and discourses can be identified. With an improvement orientation in mind these studies could be read as consciously suggesting weaknesses in methods, ‘bad methods’ overlooked by software engineering research. On the other hand, the studies can be interpreted as reporting ‘bad practice’ of incompetent software developers. In the following, the study results are reconsidered, together with a suggesting for one possible software engineering response:

- From the ethnographic inside view the first study suggests more awareness of programmers’ practical needs to support vulgar competencies of common-sense knowledge and practical reasoning.
  Another interpretation would suggest that these aspects are not important enough to be the subject of software engineering research.
- In the second study the naming problem was taken up, it can be read as supporting the idea that you must sometimes wait to make some decisions until a later stage. Methods should support this need for to wait with some decisions.
  Another interpretation would be that the design should already have clarified which distinctions to make. Delays in naming should not be necessary.
- The observations of the middleware project might suggest that it is not always possible to specify requirements before the architecture is defined. Neither is it possible to decide on the architecture without having specific requirements. The study suggests that methods must allow for requirement analysis and design to be intertwined.
An alternative interpretation would be that the project management did not establish a stable enough project environment. An evolutionary software development model would solve part of the problem.

- In the fourth study, it is possible to conclude that the developers follow a stepwise formalisation approach since they do not start writing the actual code before the formalism is elaborated. However, they also break completely with the stepwise formalisation model since they allow reasoning at code level to settle how the formalism should look like.

The other interpretation is ‘so what’ this phenomenon is part of mundane problem solving following from the complexity at hand.

In the descriptive studies we cite above, respect for the observed developers is emphasised. The ethnographic work is characterised by a humble attitude. The researcher acts as a novice in the field in order to learn about it from the participants own points of view. The material gathered is intended to represent the field as seen and understood by those who live and work within it. This research focus leads to descriptions of software development practice, as the practitioners themselves perceive it. From this point of view the people observed are regarded as the most competent experts in their field. The consequence of the ethnographers’ humble attitude and focus on revealing software development problems from the studied peoples own point of view is by definition doomed to end up in a lot of revealed and described ‘bad methods’ findings.

Let us compare how such ethnographic findings, and the ‘humble’ research attitude that lead to such findings, relate to a software engineering research attitude. In improvement oriented software engineering the goal of research is to find new methods, rules and perspectives to apply to software development based on the assumption that it is always possible to improve current practice. The latter ‘always possible to improve’ attitude implies that ‘an outside’ view is applied on the study in question, a view that does not easily fit in with the ethnographic aim of providing a description with a special emphasis on those observed. From the improvement oriented software engineering
view practitioners are likely to talk in terms of ‘bad practice’ where suggested methodologies are not followed. New and better methodologies and structures have to be developed to handle complex work situations. With this attitude, software engineers are, by definition doomed to interpret ethnographers’ ‘description of found methods problems from the studied people’s point of view’ as a criticism of developed methodologies. That is, ethnographers’ presenting ethnographic studies that over and over again reveal weaknesses in software engineering methods, i.e. weaknesses missed or not understood by software engineers’ developing methods.

The studies provide evidence to support both the ‘bad practice’ perspective as well as the ‘bad methods’ perspective. It seems that either the practitioners are not following the methods or that the methods developed are not capable of handling real world problems. Of course, experienced researchers and practitioners are already aware of the problems in the field presented by ethnographers’ descriptions. Probably most software engineers would categorise those described problems as normal exceptions in ordinary software development. Software engineers may also point to the fact that methods are thought of in terms of giving support, not as rules to be rigorously followed. In a methodological discussion relating to ethnography and design it has been shown that several studies that are primarily concerned with revealing social phenomena sometimes are dismissed with precisely a so what attitude by designers (Plowman 1995).

5. Conclusion

Ethnographers study what is actually going on in a community from the point of view of people being studied. Software engineering research aims at improving and thereby presenting possible improvement directions for the way in which software development is carried out. The different research attitudes lead to different interpretations of the original ethnographic descriptions of software development practice. One can either assume that ethnographic descriptions present ‘bad practice’ of methods usage which should be changed, or ‘bad methods’ not well suited for that must be improved. This is an issue that will probably never receive a clear ‘yes’ or ‘no’ answer as it depends too
much on the applied views on the research material presented. Different existing implicit research attitudes among ethnographers and software engineers makes a difference for the interpretation of ethnographic studies. One conclusion could be that ethnography and software engineering are in a sense doomed by definition to end up in the attitude and interpretation clash described here.

This prediction is to some extent counteracted by the author's own work (Chapter 5). However, quantification within software engineering implies the loss of what actually could actually be the most important contribution of ethnography: the inside perspective on software development practice. The quantification of ethnographic field material leads to results that are in agreement with the accepted discourse in software engineering, at the same time as it contradicts the original understanding of ethnography. The question that needs to be asked is, what will be missed by not taking in the discourse of original ethnography? The original descriptive ethnographic material provides not only an additional set of data but makes other aspects of software development visible; a different set of research questions leading to different answers might result. Conceptual concerns and theoretical reasoning, and lessons learned from ethnographic work are in danger of never entering the software engineering field. Ending the conclusion it is below suggested two possible ways of benefiting from to the original understanding of ethnography in the field of software engineering.

One suggestion is the cooperative method-development (Dittrich 2002). The co-operative approach makes it possible to take the ‘inside perspective’ into account throughout the research process. In this approach, the ethnographic studies provide with an unbiased ‘inside view’ as the common ground for the development of improvements together with the practitioners involved. The implementation of these changes can then be studied in the same way. A set of consecutive cycles shows what parts of the method innovations can be implemented successfully and why others fail to be applicable. Experiences have shown that ethnographic methods change character when applied in the context of software engineering research. A software engineering researcher might be a novice in a specific software project, but she can never be a novice in the same way as an ethnographer in a different
culture. These kinds of methodological concerns can only be explored if the researcher takes the attitude and underpinnings of both sides in this cross-disciplinary adaptation of methods seriously.

Another possible suggestion concerning how to use original ethnography is as a means of promoting methods development, to shed light on how methods are actually used in different industrial contexts compared to what researchers’ original aims were with regard to the methods developed. This means, to relate the original understanding of ethnography to a discussion of what is actually meant by methods in relation to methods following. From this point of view it is important to discuss what is meant by descriptions of how to do things in a method; is it a rule to be followed always? something that we most normally follow but in every single case? is it rather a goal or ideal to strive for, recognising that the strict following of methods is unpractical? It is only after discussing what is really meant by methods that it can be decided if the discrepancies between methods in use and researchers intentions with the methods exist or are a problem at all, and what does what mean. If ethnographic studies are regarded from this point of view they will not only give us a deeper understanding of practice but also of the relationship between practice and methods.

**Acknowledgements**

We would like to thank Prof. Claes Wohlin for providing valuable critique to an early version of this paper.

**References**


Paper II
Talking Design
Co-Construction and the Use of Representations in Software Development

Yvonne Dittrich and Kari Rönkkö

Technical Report

Abstract

Software development, just like all design projects, entails visualising something that does not exist as yet. Representations thus play an important role: even if they only describe aspects of the software developed, they mediate the common design work. Using the evaluation of a short section of a design meeting, we give a new focus to ‘talking design’ in relation to documents and diagrams. The meeting was part of a distributed software development project. An inevitable result of the distribution of project work is the visualisation of cooperation efforts. The objective of this paper is to demonstrate one aspect of why distributed software development can easily result in communication problems. The common object of the design is not a given fact; it must be collectively constructed. It is demonstrated that software development as a collective construction process largely takes place on the level of co-construction. By making a detailed analysis of a design meeting, we show how such co-construction is managed with the aid of face-to-face dialogue, enactment and whiteboard drawings. The question is posed: if important parts of design are collectively constructed during such meetings, and in such a way, what are the implications for co-operation, co-ordination and division of labour with regard to software development projects?
1. Introduction

Hersleb et al. (2000) has statistically demonstrated that distribution in software development is responsible for delays. One important problem is the lack of mutual awareness between subgroups in different sites. In this article this subject is studied qualitatively. It is demonstrated aspects of 'why' software development is difficult to achieve in spatially distributed settings. For software development - as in every design task - the co-construction of the common object of design and implementation plays an important role. Software development practice has developed using face-to-face communication– and hence depends on this form of communication. Using an analysis of a design meeting, we argue that the success of distributed software development not only depends on the development of suitable technical support but also on a change in development practices. The objective of this paper is to provide with a qualitative understanding of the development practice that is the subject for improvement.

We used interaction analysis to understand what took place during this design meeting. Interaction analysis is a social science method which allows people's interaction to be described and analysed in considerable detail, as the transcription in this paper of two minutes of a two-hour meeting illustrates. Analysis of such detailed descriptions makes visible members' actions of which they themselves are often not aware. Indeed, people would be incapable of pursuing engaged practical action if they were to maintain awareness of all small details of their action (Garfinkel 1967/1996). By applying interaction analysis in this article we have been able to establish the critical role of whiteboard, utterances, sounds and enactment's which form the occasional, transient and flexible representations used to envision the software to be developed.

The use of the method interaction analysis provides an insight into how certain practice is achieved and maintained through everyday activities. It highlights meaningful aspects that are not easily perceivable from the outside. As such, it is concerned with a different dimension of the sociality of life and work to that of the quantitative research, discussed for examples in (Basili 1996), (Basili, Green 1994).
During the period January-June 1998 we observed a software development project taking place in Ronneby/Sweden and in Oulu/Finland. Both teams consisted of students doing a project course within the framework of their respective software engineering education programmes. Both project courses are designed to provide realistic settings which allow the students to experience and reflect on software development under industrial conditions. The students develop software for real customers. An experienced software engineer, who takes on the role of ‘Head of Department’, supervises them. In our case, the project was successful: a functioning program satisfying the requirements was delivered, and both projects contributed to this. However, the co-operation between the various parties proved unsatisfactory. Despite the fact that the two sub-projects were only loosely connected, the distribution of tasks and the mutual expectations and sub-goals were neither clearly defined nor communicated. Half way through the project, communication almost broke down, although it gradually improved as a result of joint effort. In the end, both sub-projects contributed to the finished product.

For a discussion of communication and co-ordination issues derived from the project, see Johansson, Dittrich, Juustila 1998. In this article we analyse a design meeting which took place in Ronneby. We focus on the use of representations in the design process as well as the media of communication. We argue that, despite the emphasis on documents and other lasting design artefacts in software engineering, attention must be turned to the more flexible and transitory means of representation occurring in face-to-face communication in order to provide adequate co-operative design support methods. The compatible understanding achieved in this way provides the background for the interpretation of the more permanent documents.

\footnote{In this article we use the term ‘compatible understanding’ instead of ‘shared’ or ‘common understanding’. Whereas we cannot know whether two people share an understanding or have a common one, compatibility of understanding is necessary for the co-ordination of co-operative work. Compatible understanding is common enough for all practical purposes. Incompatibility of understanding can be observed in everyday practice and is often a starting point for re-negotiation.}
The extent to which generalisations can be made on the basis of a student project is, of course, debatable. More experienced software engineers would perhaps have prevented the communication breakdown for example. However, even in real life industrial settings, written and oral representations are used for co-operative design; the study can thus be seen as a pointer to interesting issues to be explored in future software development research.

The following section returns to the discussion of the use of documents and representations in software engineering. Thereafter, we focus on a specific meeting in Ronneby, a design meeting at which the overall architectural design of the software was collectively constructed. As this section is dedicated to an interaction analysis of a short sequence of the meeting we use the following section to relate our observations to discussions in software engineering and CSCW.

2. Representations in Software Development

In developing a compatible understanding of an artefact of high complexity which does not already exist, representations play an important role in meetings and other face-to-face communication as well as in asynchronous communication. It comes as no surprise that representations are considered important for software development irrespective of the backgrounds of researchers and practitioners. In the following paragraphs software engineering history will be revisited in order to establish what kind of representations that has been both emphasised and criticised in mainstream software engineering.

In the *Software Engineer's Reference Book* (McDermid and Rook, 1991), which can be seen as a compilation of a historically existing mainstream perspective on software development, the entire software development process is described as a transformation of representations of future software until the final representation, the executable program, is complete. The different representations are taken to mark milestones in the development process, interfaces between different phases that are thought to be largely independent of one other. This so-called Waterfall Model was questioned as early as 1986 by Parnas and
Clement in their article ‘A rational design process – why and how to fake it.’ Software developers do not work in this way. Parnas and Clement aimed, nonetheless, to produce a documentation of the product as if it could be developed in such a rational way, and provided a supplement in the form of the history of argumentation and change around the design. This was intended as an aid to developers in communicating the reasoning behind the design, allowing reconstruction of the design decisions later on.

Naur (1985) added another perspective to the role of documents. He claims that the development of a theory, relating the design of the interface and the software itself to its anticipated use, plays an important role in software development. Such theories, he argues, surface in the communication of design details and guide the actual work, but they cannot be communicated through documents alone. If this is true – and Naur provides some convincing arguments – then one might ask the question, ‘what is the role of various types of representations and documentation during the development phases?’

In the context of participatory and evolutionary development, discussion of the role of documents and other design artefacts should adopt a different perspective. Common artefacts such as mock-ups, prototypes or scenarios should serve as boundary objects between language and the practices of the developers, and of users of the software under development. (Ehn 1993) Moreover, design artefacts can support the development of a common practice and common language between users and developers (Dittrich 1997). Other authors emphasise the same mediating qualities of design artefacts and documents between developers. (Keil-Slawik 1992) The tools and materials approach by Bükle et al. (1995) compiles a whole development methodology on the use of documents as a means for the developers to make sense of the work practice in which the software should be embedded as well as to design and develop the software itself. The conceptualisation of the documents and other design artefacts changed in this context: they are no longer regarded as interfaces transferring information between phases and groups but as common artefacts mediating the unfolding process and as a necessary ingredient in co-operative development.
From this point of view, one can question the use of the term ‘representation’. ‘Representation’ implies that there is something to be represented. Representations of tasks to be supported, the contexts in which they are to be used as well as the human abilities which are to be modelled and automated are all, of course, deployed during the design process. Notations used for representations in the design of computer applications are designed to suit this specific purpose, modelling aspects of the functional context for their encoding and implementation on a machine or in describing the existing work practices with the aim of promoting a better understanding on the part of the developers. There are other notations and languages used in software developing which focus on the structuring of the future application. The overall design of the computer application, the division into modules and their interaction, class-diagrams and similar documents are used to deploy the implementation technology in a skilful way in order to construct the desired functionality for the user. Formal notations as well as those invented ad-hoc serve to envisage, discuss and ultimately implement a computer application which has not yet taken form. In a very real sense they do not represent anything as such; they facilitate the development of what they describe. Using a concept from Bardram (1998), this aspect of the use of documents and other representations can be called co-construction; the common object is defined in relation to the tasks to be supported and the means of implementation.

Software engineering literature often emphasises relatively permanent representations in order to preserve design discussions and decisions as resources for the further refinement and implementation on which it is based. This paper suggests that there exist a need for emphasising and

3 Lucy Suchman gives examples of this kind of purposeful reconstruction from the area of cognitive psychology and artificial intelligence (Suchman 1990, Suchman & Trigg 1991). In other places, she highlights the differences in representations as a means and as results of work and representations of work practices for design (Suchman 1995).
understanding the nature of ‘the more fleeting and occasional representations’ that take place in face-to-face activities, this as a means to be able to improve on distributed software development.

During an empirical study of co-operation in a distributed software development project we had the opportunity to videotape and analyse a design meeting. At this meeting, the collaborative construction of the overall architecture was proposed, negotiated and agreed upon. The resulting document, the minutes of that meeting, can only be regarded as a reminder for the participants. Other means were used to present software in the future in order to make it visible for the group and to enable discussion of the overall design: drawings on the white board provided an easy-to-change medium for representations, and allowed participants to envisage the programs’ behaviour as well as the properties of proposed solutions. Such methods were necessary if the software engineers were to reach a compatible understanding of the common object - in this case, the overall software architecture. Based on our analysis of design talk in the following section, we consider not only the distinction between oral and written representations but also their use with regard to different aspects of co-operative design work.

In the following section we briefly describe the observed project and the study and research method chosen. We then introduce the design group. We analyse in detail two minutes of one, two-hour meeting, which we call ‘bridge talk.’ Thereafter, we discuss what the implications are for co-operation and communication and the use of representations in software development as well as for computer support for this kind of work.

3. Bridge Talk

In this section, we show how participants in a design meeting used representations in their discussion of co-operative design. They struggled with the alignment of perspectives, resulting in a somewhat laborious development of an agreed implementation. In face-to-face negotiation and re-negotiation, the participants explored, resolved and together constructed an overall design. In the analysed sequence, they jointly constructed the problem of the organisation of the interaction
between the server and client in their application and reached a decision as to how to solve this. With the detailed analysis of a very short sequence we aim to demonstrate the role played by semi-persistent media for representation, such as talk and enactment in this kind of co-construction. The ‘bridge’, which characterises this kind of talk, is a module of the program the participants explore during the sequence analysed here. In the following section, we discuss the implications of such practices of co-construction for the understanding of co-operation, co-ordination and division of labour in software development projects.

3.1. Background and Methods

The bridge talk took place in the context of a 'large team project' within the framework of an undergraduate course in software development in Ronneby. Approximately fifteen third-year students in software development, three economics students and two students taking part in a programme called 'people, computers, and work' were involved in the 'large team.' All members worked together in as realistic conditions as possible. Each student had a budget of 700 hours to 'spend' on the project. The budget for this specific project was 13300 man-hours. The students adopted the role of a software developing company with an industrial partner as a customer; the supervisor played the role of head of department; the group itself decided on negotiations, contracts and commitments as well as the organisation, planning and execution of the contract.

We observed one of these projects, in which students were co-operating with a small programming project in Oulu, Finland. The programming project teams in Oulu consisted of 4 students, each with a budget of 200 hours. Otherwise, the rationale behind the course was identical to that in Ronneby. Working for an external customer allows for realistic conditions. In our case, the Ronneby project members adopted the role of customer. The goal was to learn and teach about software engineering in the large.

In order to evaluate the experience, the project was subject to two case studies, one in Ronneby and the other in Oulu. During the development of the software - which took 5 months - the empirical research-material was gathered by interviewing, observing, audio-taping and video-taping
project members at different meetings e.g. ‘Head of Department’ meetings, project meetings, inspection meetings and video and telephone conferences. An alternative approach was to hang around in the project room and take coffee breaks with the team members in order to get a grasp of the evolution of concepts and the difficulties in communicating them. We were given access to the written communication between project members in the form of emails that were digitally ‘carbon copied’ to us when sent to the receiver. All documents produced were placed in their common workspace with the aid of a BSCW server, to which we were also granted access. We visited Oulu once, half way through the project, in order to interview the Finnish team and to share observations with the teachers and the researcher in Oulu. The major part of the empirical research was done in the first period of the project, the pre-study phase.

The customer in our project was a medium-sized Swedish software company specialised in strategic gameware. The task of the project was to develop a software system to simulate management education in companies. The co-operation and co-ordination were left to the project members. In Ronneby, the students decided to make this the task of the project leader; in Oulu, the group considered the project too small to give any one person the responsibility for it. The Oulu students jointly took over the responsibility for receiving and answering mails and co-ordinating and keeping contact. The form of co-operation – Oulu was to play the role of a subcontractor – and the content of the subproject were defined at an early stage. The Oulu students had access to all project documents. They used the common virtual workspace for storing their own documents.

Co-operation proved, however, to be a problem e.g. it was not clear which of the groups was responsible for the design of the database and which was to provide the interface between the parts. Mutual requirements for the interface between the two parts were exchanged at a late state in the project. Documents were difficult to locate in the

---

4 BSCW stands for 'Basic Support for Co-operative Work', an application which has been developed by the GMD in Bonn. (Bentley et al. 1997)
complex structure designed by part of the Ronneby project and were very much tailored to local needs. Professional terms such as ‘project report’ were interpreted differently in the two countries due to differences in software development ‘culture’. This was not recognised by the students and caused misunderstandings and friction between the project groups. In the end, the sub-system for which the Oulu group was responsible could only be finished because the students spent additional time on the project and were awarded extra study credits.

Despite a good technical infrastructure – the students were able to meet in video conferences, they used e-mail and chat, a BSCW was installed to provide a common work space, and telephones were provided – the difficulties in co-operation and co-ordination were not resolved until very late in the process. The mutual access to an elaborated set of documents made no difference in this respect.

The video recording and subsequent Interaction Analysis of a design session helped us to understand the source of the above difficulties and to reformulate the question; we stopped wondering why participants did not communicate and started to admire their success in spite of the considerable geographical distance between the two groups.

3.2 The Setting

The project team in Ronneby, consisting of twenty students, had to split up into sub-groups. They started out with two design teams with three software engineering students each, a quality team consisting of software engineering students, and the People Computer work students and economic students, who comprised a team of their own. For the implementation phase, the participants split up the group in accordance with the sub-parts of the software. In the Ronneby team, the co-operation and co-ordination between the sub-groups worked rather well, and the final product was developed within the time allocated.

In the taped session, the two design teams, who had developed proposals for a common overall design solution independently of each other, met together with the project manager in order to combine their contributions and develop a common design. They met in a smaller group room which was equipped with a whiteboard and a table with
chairs around it (see figure 1). The meeting had already been in progress for one hour when we transcribed our two minutes. During this time, each team presented its design proposal to the other team and both teams were already one hour into a discussion of the common design. In the following analysis of two minutes of this session, we have focused on the means of representation that served as common references or common artefacts during the design meeting.

The following two sub-sections together illustrate what we will later develop as routine co-construction work. This has major implications for the establishment of co-operation among members. The third sub-section focuses on the resulting documentation of the meeting. At the end of each sub-section we briefly highlight what we consider to be important for future discussion. Questions and reflections are taken up in the following section.

3.3 The Whiteboard as Lasting Common Ground

One of the means of representation was the whiteboard which formed part of the equipment of the room. Throughout the session it was used to draw, change, erase and redraw complete or partial design solutions.

Figure 1.: Diagram showing how participants were arranged around the table, with a whiteboard in the room. On the whiteboard (here laid out on the floor for convenience) we can see the ‘game creator’ represented to the left, the ‘game engine’ in the middle and the ‘client’ to the right. The issue of the discussion in the sequence analysed is in the space in-between the broken line, where there is a need for a ‘bridge’.

The following two sub-sections together illustrate what we will later develop as routine co-construction work. This has major implications for the establishment of co-operation among members. The third sub-section focuses on the resulting documentation of the meeting. At the end of each sub-section we briefly highlight what we consider to be important for future discussion. Questions and reflections are taken up in the following section.
The students used common notations in software engineering: boxes, text, and arrows to indicate visualised parts, connections between these as well as to record meaningful names.\footnote{These by no means provide a formal notation. They can be understood as an ad-hoc graphical notation to make the envisioned structures visible.}

At the point where we started the transcription, everybody seemed to be satisfied and to be able to grasp each other's proposals. It becomes clear, however, that there are still conflicting perspectives with respect to the understanding of the white board representation as the Chief of Software Architecture said:\footnote{The transcription is 'talked language', an expression form that often is fragmented and locally dependent. This is the usual way to transcribe talk within Interaction Analysis. Transcription notation system developed by Gail Jefferson. (For further information see (Heritage 1984).) The original language in the transcription is Swedish. CSA: Chief Software Architect, PM: Project Manager, C, V: Software Engineers.}

CSA: \(\rightarrow\)What! Put something in-between, there is no need for anything in-between, if we, 
 \hspace{1cm} [if ]we have 
PM: [If] 
C: [If], if we have TCP IP (0.2) 
CSA: No, even if we have RPC, there is no need for anything in-between. (0.4) 
V: \(\rightarrow\)No need for anything in-between? (0.2) 
CSA: Between ((points with his right hand at the whiteboard drawings, first at the engine then at the client)) engine and client. Okay we need a communication class, \(\rightarrow\)but that we will need in whatever solution we choose. (0.2) 
V: ((moves his chair a bit backwards, then leans back in his chair at the same time as he puts both hands behind his neck)) Hmm ((makes smacking noise)) ye, yes (0.2) 
C: \(\rightarrow\)Then we do not need that bridge which we had [then ] 
V: [Hm::] (0.9)
V: No eh:: the reason that I wanted the bridge (0.2) is namely:
First, this discussion we are having now. (0.2) We don't know
what technique is good. But it won't ((moves his head as if he's
saying ‘no’)) be any problem because ((looks in a searching
way at the whiteboard)) we will export object type COM, so
that the technique ((an opening gesture with his arms)) in this
way lets say TCP IP or RPC and they won't function. So then
we can exchange without digging in the client, (0.2) OR without
digging in the engine=

CSA: =Hm::→What you are really saying is, your bridge ((looks
towards the ‘bridge’ on the whiteboard)), it could just as well be
connected with the engine. ((V nods his head as though he is
saying ‘yes’)) It is just that you have put a COM interface in-
between them.

V: Yes ((CSA nods his head as though saying ‘yes’, looking at V))
[and because]

CSA: [That, that   ]

The first statement in the transcript enables CSA to introduce a new
reflective loop allowing the reconsideration of a design detail the
participants had earlier termed ‘the bridge’. He uses the whiteboard to
indicate to the others what he is speaking about. As he takes up
implementation technologies that are familiar to the participants he
questions whether this part should be an independent module. To
answer, V also uses the whiteboard - even when his argument for the
independence of this module is based on maintainability and
changeability, which bear no relation to the drawing on the board. ‘The
bridge’ module for V provides the encapsulation of the technical
solution for the interface between ‘game engine’ and ‘client’,
‘bridging’ the ‘gap’ indicated by his distant hands. The whiteboard and
gestures are used to visualise possible future solutions for V’s
colleagues. The whiteboard serves as a common memory and a stable
ground, a mutual frame of reference for discussion purposes. The
whiteboard adds an additional dimension to the ordinary, fast-flowing
and continuous talk. It provides a space where problems can be pointed
out, solutions are made visible, and where they become the subject of
evaluation and reflection. With the help of the board, which is visible to
all members of the group, not only is a representation of the design of
the future system developed but also the meaning of this representation. The connection between the representation and its meaning develops simultaneously. ‘What do the blocks, arrows and names stand for?’ ‘Why is the solution a good solution?’ ‘What alternatives have been discarded?’ are examples of the questions that are answered parallel to the joint development of the representation.

By providing a shared indexical space, the whiteboard structures the participants' mutual orientation. It facilitates their further development as well as the asking of relevant questions. In this way, problems or agreements are brought within reach; they can be pointed at and talked about even in the future. Talking, and the directions it takes, are determined by the participants and can change with every new turn in the discussion. Talking about something that does not yet exist is a difficult endeavour. The whiteboard provides new possibilities in this process. Holding or occupying the floor in front of the whiteboard, keeping on drawing, or keeping the whiteboard-pen to oneself can be strategies to sustain the discussion in a desired direction, while handing over the whiteboard pen to a specific member can be used to steer the continuing discussion. The whiteboard can be seen as an ‘enricher’ of the possibilities for discussion opened up by new co-ordination opportunities.

A number of questions arise in relation to forming an understanding of software development: what is the role of meetings in software development? Are they just to inform and co-ordinate activity, or do they provide a possibility for co-constructing the common object of the project? What is the result of such meetings, the final drawing or the compatible understanding developed during the meeting? Can the local design talks taking place with the help of such semi-permanent media be replaced in distributed development?

3.4 Enacting as a Resource in Communication

At the point where the first transcription ended, it appears that the participants had reached a mutually compatible understanding again. However, another project member from the same team as CSA expressed confusion:
But what, the COM interface, COM is an object ((forms his hands as if he is holding the COM object in-between them over the table)), an object that is COM or a COM component ((each time he refers to the COM he moves up and down what he is holding in front of himself)), but what ((looking in a searching way at the whiteboard)), what exactly is the COM component? →Is it the bridge ((moves what he is virtually holding in front of himself to the right)) that is the COM component [or is] it ((now moves what he is holding to the left))

V: [no ]
C: a part of the game engine, or? (0.1)

C, who is a member of CSA's team, realised that he did not understand the new turn the conversation was taking. So he forced the participants into giving one more round of explanations by asking questions, hoping that he would also be able to grasp the new understanding expressed by CSA. To express and clarify his confusion, he used his hands to demonstrate in an attempt to communicate what he understood of the role of the program part called COM, although he did not know where to put it. The talk continued:

CSA: A part of the game engine=
C: =⇒A part of the game engine? ((first looks at CSA then directly back to V as if he is the one to answer the question))
V: The whole engine is a COM object and it can in its turn, the file can access new COM objects ((moves his right hand forwards)), you have your game engine ((representing this with an open hand held in front of himself)), Schach ((moving his right hand upwards fast as if he was grasping something of the same size as an apple in the air, and holds it)), then they say give me values, ((performing the same movement and grasping, but now with his left hand)) prrt, the value says, give me a company, ((same movement and grasping again as the first time with the right hand)) schach, my company says give me sub-scenario one, ((performing the same movement and grasping again, but now with his left hand)) one, then ask for then, now I have these two ((holding the two virtual ‘values’ in front of himself, and looking at them)), tell everything to me and then the game
engine starts ((making the movements in turn with both hands, left hand backwards and right hand upwards, again and again)) going [backwards and forwards there. ]

C:       [\(\rightarrow\)And it is the bridge that sits ] and ((repeats V's earlier hand gesture with his right hand, but with smaller movements)) pick up and asks after all these=

V:  =Yes, and the bridge goes from using these ((does the hand movements just as above, again and again)), these COM objects, then interprets them to whatever ((kind of throw away things movement with his right hand, repeated faster and faster a few times)) communication channel ((C nodding 'yes' with his head, up and down)) it wants=

V explains the design by referring to the dynamic behaviour thought which will characterise software in the future as it is implemented by a computer. As there is nothing there yet to demonstrate such behaviour,\(^7\) V enacts the program part under discussion in a metaphorical way using his hands. As if the body language were not enough, V adds sounds to his enactment, to put more 'life' into it. The body language expression of the envisaged behaviour is taken over by C as he tries to formulate his understanding. V then takes it up again as he confirms the rephrasing of the purpose of the bridge in the interaction between client and engine. The static representation on the whiteboard is not sufficient to present the dynamic aspects of the software under construction. In order to confirm and unfold earlier enacted contributions to their talk, the developers also imitated each other's enactment's as references to the subject of their discussion, as if compatible understanding were linked to these ‘objects of enactment,’ the body language expressions.\(^8\)

\(^7\) Even if there were an implementation, the behaviour of the program on this level would not be visible and it was thus necessary to visualise it in some way in order for it to become the subject of discussion. Software in this respect can be regarded as invisible in a more fundamental sense. (Brooks 1987)

\(^8\) Garfinkel and Burns 1979 described a similar phenomena as the *ghost* entry, gestures at the board that were never marked but still referred to (cited by Suchman 1990, p. 317).
In another episode later in the meeting, enactment was used along with the whiteboard in order to introduce dynamism into an otherwise static medium.

Software development methods often focus on documents, on representations of work-like plans and project management tools and on representations as (intermediate) products. How do such ‘normative’ representations relate to the kind of design and co-construction discussions presented here? How can more permanent and static representations be incorporated into lively design meetings of both the scheduled and informal type? What does it take to keep the meaning of a design alive?

3.5 The Representation of the Results of the Design Meeting

The meeting continued until the participants thought they had reached a compatible understanding of their object, the collaboratively constructed overall design. The minutes from the meeting included the same representation as that which emerged on the white board. This was hardly accompanied by any descriptive text.

![Diagram of the minutes](image)

**Figure 2:** A diagrammatic representation of the minutes.

The minutes were a memory support compiled exclusively for the team members present at the meeting. In the discussions, the frame of reference necessary for the interpretation of the graphical representation and its use later on in the development was established
parallel to the evolution of the envisaged structure it represents. But how can project members who were not present at the meeting interpret the minutes and contribute to a compatible understanding? How was it possible for the project members who were not present to co-operate and to co-ordinate work that depended on knowledge developed at this meeting? The Ronneby project, which was the larger of the two, had the advantage of members being physically within reach of one another, making face-to-face communication possible. They could establish and re-establish their compatible understanding of the common product by means of face-to-face communication; they met in the corridor or had a discussion during a coffee break, where they could arrange new meetings or perhaps develop an understanding by overhearing others talk, just to give a few examples. While the Finnish team was forced to put their trust in the representations and information technology, neither of which proved to be sufficient.

4. Co-Construction and Distributed Software Development

The software engineers' task of developing and agreeing on the overall design solution was not considered special in any way. It was part of their daily, ordinary work, an important meeting of course, but just another meeting nonetheless. The actual work of developing a compatible understanding of, and agreement on the jointly constructed design object was taken for granted and passed unnoticed. Participants are not necessarily aware of this, and it is regarded as mundane work. Project members would not be able to function or make progress in the project if they were to consider every such step in detail.

Design meetings such as the one we picked are a common feature of software development. The use of black- or whiteboards, the 'enactment as representation' of the behaviour of the various parts of the design, is so common that it passes unnoticed. When it is highlighted as in the transcript above, every software developer will be able to recognise the common practice. It is perhaps because of such professional blindness that this kind of co-construction of the common object of work and its role in the co-operation involved in software development is not properly understood.
The Importance of Co-Construction in Software Development

The sequence of the design meeting we analysed in the previous section is but a minor example showing how software developers together designed part of a computer application by developing representations of the future system which were meaningful for the participants both at the time and later on. We have shown how the participants used talk, graphical representations on the whiteboard, and enactment's to visualise, communicate, reflect, and discuss their ideas about the future system.

However, software development is not just a co-construction of meaningful representations since the software source code, which is ultimately developed, must have a use in a given context. Distributed coding and testing must also be co-ordinated so that the partial products can be incorporated into the final application. In the division of labour and the co-ordination of this, design representations such as the one developed during the meeting described above have an important role to play. They provide a division into parts of software in the future, with defined interfaces between these parts. The latter provide meaningful units for the division of labour, whereas the interfaces can be regarded as points where the implementation of one part must be co-ordinated with the neighbouring modules and the groups responsible for these.

Communication and co-ordination are widely recognised problems in software engineering. They also proved to be problematic for the project under discussion. However, by analysing the videotape of the ‘bridge talk’ we were able to understand the problems involved in co-operation and communication in a new way. We began to regard co-operation in a more vertical way. The problems of distributed software development may not only be the result of difficulties in co-ordinating co-operative work over a distance but reflect a difficulty in understanding the means of distribution and co-ordination of the co-operative effort along the lines of the design representation compiled as the outcome of our meeting. To be able to implement a part successfully and to determine which small-scale design decisions one's colleagues need to know about, one has to understand the rationale behind the distribution as well as the interplay between the parts. It was
precisely these issues that were considered in the section we have analysed in detail. The fact that these were not present in the minutes of the meeting indicates lack of awareness of this aspect of their work.

As the same interplay between co-construction and the co-ordinated distributed implementation is repeated in more detail for each part and sub-part down to the basic units such as classes or procedures, this relation between co-construction, co-operation, and co-ordinated work must be regarded as an important aspect of software development. In the following subsections we present one way of looking at the dynamics of co-operative work, returning finally to software engineering.

4.2 Dynamics in Co-Operative Work

Bardram (1998) considers the dynamics of co-operative work. His conceptual frame is based on Activity Theory. This theory identifies a triple level hierarchical structure of collaborative activity, as co-operative work is called in this school. On the first level, the co-construction, the collaborative object is either unstable or does not exist and must therefore be collectively constructed, i.e. co-constructed. Questions such as: What is the meaning of this problem? How did the problem emerge? Why are we trying to solve it? Who benefits from the solution? are asked on this level. On the second level, the Co-operation, the actors have achieved a common object for their collaborative work. The actors are engaged in balancing and adjusting their own actions to those of their partners to achieve the common task. The third level, the co-ordination, is that of the routine flow of interaction, where work is conducted in harmony with surrounding activities. Here the flow of work goes on without questioning or discussing, in accordance with tacitly assumed traditions and norms. Bardram argues that in practice collaborative work takes place on all three levels as dictated by the requirements of the situation. Figure 3 shows the dynamics between the different levels. Collaboration shifts between the three different levels of collaboration, i.e. co-ordination, co-operation and co-construction. Problems within routine work lead to breakdowns in the co-ordination level and become subject to co-operation; breakdowns on the co-operation level in turn become the subject of co-construction.
Co-Construction
Reflection on the Implementation: Stabilising
Object of work the object of work.

Co-Operation
Reflection on the Routinisation: Stabilising
Means of work the Means of work

Co-Ordination

Figure 3: Levels of collaborative activity (Bardram 1998, p.92)

For Bardram, normal, everyday work constitutes a co-ordinated activity. Co-operation and co-construction are exceptions caused by problems disturbing the normal flow of work, and they require special treatment. We do not believe that this is the case for software development. Co-construction may not be the only kind of activity, neither is it an exception caused by a breakdown. Co-construction, co-operation, and co-ordinated activities are incorporated into one another in a different way.

4.3 Three Levels of Collaboration in Software Development

In software development projects, routine division of labour is produced as part of the design process of the software. Design and development projects do not start out with a given object on which to work with. The common object is the subject of construction throughout the entire process. Expertise and routine in terms of to how to run such projects will, of course, be acquired over time. With regard to each project, the new product, a suitable way of dividing labour and the team, and a working co-ordination of the single tasks must be achieved. Even if mechanisms and routines exist to support co-ordination, they must be adapted and customised for the specific project in hand. The process must also be designed and implemented. (Floyd 1992)

The emphasis put on the different aspects of co-operation may change during the project. Independent of the overall process model, co-construction must be regarded as the main task during the requirements
and design phases. During implementation, co-operation and co-ordination may be most clearly visible from the developers' perspective. However, as the design is not finished until the software is ready, the division of tasks to be co-ordinated is not stable. Tasks are redistributed and designs are changed because of unexpected technical constraints or evolving requirements. One could say that co-construction is part of the normal, everyday, and in this respect, routine work of software developers.

The design talk we have analysed here is an example of such routine co-construction. During the talk, the participants developed a compatible understanding of how software on an architectural level should look. As a result of the various arguments, the rationale behind the design, the advantages and disadvantages and the risks were also shared. Thus not only were the foundations for the division of further design and implementation laid but the background to co-ordinating the co-operative tasks was also developed. The issue of what other tasks might be influenced if one has to change one's design as well as whom to contact etc. was also considered. In this co-constructive task, only the whiteboard, talk and enactment were used as means of representation. The representation outlined in the minutes showed the attained solution but gave no indication as to the ‘whys’ and ‘hows’. A simple reading of the minutes could not communicate the result of the co-construction or the compatible understanding of what should be built. To make sense of the representation one had to be present during its construction or needed help with interpretation. In the Ronneby team, this problem was solved through everyday communication, which remained unrecognised and undocumented. A sub-group which did not take part in the product design but which designed the quality assurance routines and quality criteria for the different documents was introduced into the co-operative design and implementation without any major problems. The Finnish team, however, was left behind. Despite being able to read all the documents and follow changes by means of a common BSCW workspace, the members were not able to participate in co-construction or its results in a satisfactory way.
Different representations and forms of representations serve a variety of purposes. This has to some extent already been acknowledged in the literature. Process-related documents – plans, review routines, progress reports, for example – help to organise and co-ordinate co-operative work. Product-related documents – stating requirements, design documents on different levels, documentation, test specifications and test protocols – describe attributes of the software under construction. We suggest that one takes into account another kind of representation and consider more transient means of communication. In our example, talk and enactment, two transient forms, were used for highly interactive co-construction. Drawing on the whiteboard, a less fluid medium than talk and enactment, was used to structure the discussion as well as to keep certain aspects of the object under construction present during the meeting. On the other hand, this is a flexible enough medium to avoid constraint on the evolving and sometimes meandering discussion. Written or computer-based documents yielding very permanent representations were used to document results, to structure the design, and to organise the common work over a longer period of time. In addition, the different mediums were interwoven; oral communication was used to explain documents and computer-based drawings such as the ‘result’ of our talk helped the participants to remember what was going on during the design meeting. If different kinds of representations are to be interwoven in such a way, the role of documents for software development must be reconsidered as well as the possibilities for distributed software development and its support by computer supported co-operative work (CSCW) applications.

5. Conclusions
In this article, we focussed on the role of ‘talking design’, of fleeting means of representations – like utterances, whiteboard drawings, and enact – for the co-construction of the common object of work in software development. We highlighted a specific aspect of software development that provides a difficulty for its spatial distribution. To argue our point we first referred to the historical discussion of the role of representations in software development. We then provided an
interaction analysis of a part of a meeting, showing the work of co-construction in the discussion around different design solutions. We discussed our findings using concepts from software engineering literature as well as computer supported co-operative work.

It was presented a triple level hierarchical structure of collaborative activity to help understand the different kinds of collaboration that exist during software development. In addition to the emphasis on documents and other lasting design artefacts in software engineering, the more flexible and transient means of representation occurring in face-to-face communication must be considered in order to provide adequate methods for supporting co-operative design. The two lower levels in the collaboration hierarchy ‘Co-operation and co-ordination’ depend on a compatible understanding of the common object of work. Before co-operation and co-ordination is possible, there must take place co-construction leading to the issues of co-operation and co-ordination. Meetings in software development provide one structured possibility for co-construction of that common object of work. Whether and how local design talks taking place with the help of such semi-permanent medias as the whiteboard and enactment's can be replaced in distributed development is still an open question.

Our observations suggest that co-construction within software development takes place above all in face-to-face situations. Distribution of labour seems necessary, but how reliant is software development on face-to-face work, and at which occasions. In other words, awareness of the prerequisite for the continuing ‘routine work steps’ in the form of co-operation and co-ordination is crucial in especially distributed software development.

Computer Supported Cooperative Work literature provides examples of project rooms supporting synchronous and asynchronous communication (Roseman & Greenberg 1996), applications for simulating face-to-face communication, and even accidental meetings ‘in the corridor’ (Nkanishi et al. 1996). However, whatever technical means are deployed, the medium will influence the ‘what’ and ‘how’ of communication and co-operation. This does not mean that we must give up distributed development. We all recognise examples of
distributed co-constructive practice: the scientific discourse or the organisation of conferences, for instance. In addition to the development of well-adapted technologies, a new and different practice supporting the work of co-construction must also be developed for distributed software development,

Acknowledgements

We wish to thank Monika Alriksson and Timo Petman, who assisted us in our fieldwork. Together we tried to make sense of what we were observing. Our thanks go to Conny Johansson and Antti Juustila who made the co-operative project possible and provided critical comments. Thank you to the project members, who with a rare openness allowed us to spy on them. We thank our colleagues Sara Eriksén, Claes Wohlin and Andy Crabtree for their support and comments on earlier versions of this paper.

References


Paper III

‘—Yes, What Does That Mean?’
Understanding Distributed Requirements Handling

Kari Rönkkö

This paper is a revised version of a chapter in Social Thinking - Software Practice 2002, editors. Dittrich, Y., Floyd, C and Klischewski, R. MIT Press.

Presentation of Paper III

Paper III is a revised version of a chapter in the book Social Thinking - Software Practice. The latter seeks to promote discussion of the interrelation between social science based on approaches elucidating the diversity of social aspects of software development and software practice. Paper III is one of five contributions which comprise the basic framework of the ‘grounding part’ in that book. Grounding in this context means to promote a broader understanding of the software development process with the aid of qualitative empirical research. This includes a broader understanding of theoretical research concepts, adapting social thinking in order to develop and improve software development methods. The first part of the title, social thinking, refers to scientific reflection guided, informed and/or inspired by social scientific approaches. The second part of the title, software practice, refers to software development, design, use and related management as shapers of current technology and changers of the way in which we engage in social relations at work or at home, in groups or in society (Dittrich et al. 2002). In Paper III, the social thinking element taken from sociology comprises the research methodology ethnography, which is combined with the informing analytic research program ethnomethodology. The software practice element in this paper is the software engineers studied, whose job it is to manage requirements in a distributed software development project.
The book *Requirements Engineering: Social and Technical Issues* (Jirotka and Goegen 1994) is, as far as the author is aware, the only existing interdisciplinary work that combines requirements engineering and ethnomethodology. The software engineering or the requirements engineering perspective demonstrated in Paper III is taken from this interdisciplinary book (McDermid 1994). The requirements engineering perspective was chosen because this book was, and still is, to the best of the present author's knowledge, the only existing interdisciplinary literature to include both requirements engineering and ethnomethodology. Paper III is primarily concerned with illuminating what the methodological gains might be for software engineering by adopting the sociological research approach known as 'ethnomethodologically informed ethnography'. Requirements engineering has developed since McDermid first outlined his requirements perspective in 1994; today it is considered to be a multidisciplinary activity (Nuseibeh and Easterbrook 2000). In Paper III ethnomethodologically informed ethnography is related to software engineering research by applying it to industrially distributed requirements management. Ethnography and Ethnomethodology today are acknowledged requirements elicitation techniques in the community of software engineering (Nuseibeh and Easterbrook 2000; Sommerville 2001; see also the Introduction to the present thesis, Section 5.1).

During the process of revising the text for this licentiate thesis some headings have been shortened, and minor changes and corrections have been made to the text. The changes in the text do not influence the overall contents or general contribution of the study. The starting point of this study was not to test general assumptions or hypotheses on the basis of general method and model descriptions; rather, it was intended to be an explanatory study. It is also worth noting that Papers III and IV are written using an ethnographic style involving the personal pronouns 'I' and 'we'; both papers adopt a particular voice and point of view. In this way, the papers comply with the ethnographic emphasis on fieldwork experience rather than fieldwork findings. The ethnographic way of writing is influenced by the hermeneutic disciplines while the writing of software engineers is much influenced by the natural sciences.
Abstract

Requirements engineering is a process comprised of the requisite activities for creating and maintaining requirements documents. Different documents are produced at different stages of the development process. In most systems, requirements change. People develop a better understanding of what they want to do; companies and projects reorganise; when people change positions their successors may have a different understanding of problems and new ideas; modifications are made to hardware, software and organisational structures. Requirements engineering is concerned with managing changes in requirements. This paper describes requirements problems in a distributed software development project and describes them from the project members' ‘own point of view’. The research methodology used is that of empirical and qualitative ‘ethnomethodologically informed ethnography’. It results in a discussion divided in two parts. The first part is concerned with the precise benefits and drawbacks of using the methodology. The second part illustrates how the central problem of requirements engineering is not completeness but the production of collaborative theory building and mutual intelligibility. This central problem is compared with one particular requirements engineering perspective presented in a book where ethnomethodology and requirements engineering are discussed.

1. Introduction

—Yes what does that mean? Who is it written for? Why do they need that? What roughly are its content, and what do they want it to look like? Key elements are missing there! These were the reactions of a manager, talking about his requirements. I mumbled something like, Key elements are missing? The manager continued: ... and from this point to derive who has introduced these requirements. It is the customer, but the customer is an unknown quantity. The organisational model in this project, a document-driven software development approach, succeeded in providing enough guidance for structuring the overall software development process, but it was less successful in facilitating project members' everyday co-ordination needs in handling the requirements. This paper is based on a qualitative study of a distributed project within an internationally prominent supplier of
equipment for telecommunications systems and related terminals. The paper deals with two main issues. I show how the central problem of requirements engineering and software development is not completeness, but collaborative theory building and mutual intelligibility. Then I use the study to illustrate how a qualitative approach can help realise these co-operative goals. By addressing these issues, the paper also contributes to meeting the challenge raised by Nørbjerg and Kraft (2002), the challenge of taking the context of software practice into account, especially, the importance of informal co-operation.

The structure of the remainder of the paper is as follows. First, I discuss contradictory ideas from the field of requirements engineering and software development. Second, the project is presented. Third, I describe the epistemological grounding of the methods used to study the project. Fourth, the project members' struggles with the complex organisation of work today are highlighted. Finally, conclusions derived from both the project members' struggles and the application of the qualitative methodology are presented.

2. Requirements Engineering

Requirements handling is a much-discussed topic. In analyzing the complex ways in which social and technical factors affect the development of software Jirotka and Goguen (1994) refer to the problem of requirements as a high-cost feature of the software development process in the latter stages of the development life cycle. They suggest that one possible interpretation for this is insufficient knowledge of the effects of the various available requirements methods used in the development process. They claim that we in fact only have begun to realise how requirements are actually construed and used in large software development projects (Jirotka and Goguen 1994 pp.2-3).

Within software engineering, the overall approach for handling the role of software requirements and the whole development process could be described as devoted to the idea of rigor and control. Methods have been developed to guide and control software development work. In models of this kind, requirements are the sole starting point for the
development process. Developers should be able to handle them without further knowledge of the use context or the history of a project: a formulaic approach to requirements is emphasised. According to McDermid and Rook (1993 p.29), no single interpretation of a language exists, pointing to the fact that possible misinterpretation of requirements during software development will always exist. Their contribution to this somewhat obdurate problem remains the proposal to develop an adequate formal language to minimise the misinterpretation possibilities. McDermid (1994) notes that no methods capable of adequately deal with requirements for socio-technical systems during software development seem to exist. He also points out that requirements are negotiated, not captured, warning us against the belief that well-defined requirements exists that are waiting to be discovered. He introduces three different approaches in requirements capturing and analysis, naming them orthodoxy, fundamentalism, and heresy.

The orthodoxy approach emphasises global holistic considerations and a uniform view of requirements. The problem is that the process leading to the creation of specifications starts in the wrong room, with requirements other than the fundamental ones. The requirements initiators are forced to provide more detail than is desirable in order to define requirements. This approach leads to low flexibility in design and implementation, because the over specification obstructs change. The system presupposed is effectively influencing the design decisions before the determination of individual requirements takes place (McDermid 1994 p.25).

The fundamentalist approach addresses the issue of over specification through focusing on a small number of key requirements for a system so-called cardinal-point requirements. This approach seems to solve some of the problems of the orthodox approach, but it introduces others. The cardinal-point requirements are typically very room resulting in disparate and under specified requirements, and often necessitating large amounts of non obvious problem-domain knowledge in order to be interpreted correctly. There is no guarantee that the requirements are consistent, and problems can arise from under
specification. There is also the risk of not bringing enough problem-domain knowledge into the requirements; system developers often lack this kind of knowledge (McDermid 1994 p.27).

The essence of the heretical approach is captured in the idea that it is impractical to produce complete, consistent, and implementable requirement specifications. Potential conflicts always exist that can only be unambiguously identified once there is an overall idea of the design. This view conflicts with ideas such as those represented by the Waterfall model, and thus it requires another process model enabling iterative and evolutionary development work. An advantage of the heretical approach is that conflicts between non-functional requirements and fundamental objectives can be recognised, which is not easy in the orthodox or fundamental paradigm. With the heretical strategy there is more freedom to change requirements and to achieve an implementable compromise. It is easier to validate the design with customers than it is to reach agreements on requirements, since customers agree to what they are actually going to get, not to what is an analyst assumes that they have required. In the most fundamental heresy, there is no fully elaborated requirements specification and requirements are finalised with the produced design. Ehn (1988) represents the Scandinavian participatory design approach (Ehn 1988 p.35). Since they are critical to existing requirements analysis, especially the orthodox approach, McDermid still acknowledges the requirement approaches as being effective if used under appropriate circumstances. He sees requirements as a problem of appropriate choice and implementation of a normative method (McDermid 1994 p.37). Work-related problems, in this view, still solved through putting trust in rigor and control applied in the use of adequate methods.

Other authors have offered perspectives on requirements handling more closely connected to human and social issues. Concerned with what it means to program, Peter Naur claimed in his article Programming as Theory Building as early as 1985 that programmers build theories relating the design documentation and the software itself to its anticipated use. In other words the text resulting from software development are outside the reach of what can be determined by rules (Naur 1985 p.57). In her article Software Development as Reality
Christiane Floyd claimed that we do not analyse requirements; we construct them from our own perspective. She bases her argument on constructivist thinking. She suggests that methods in use are orientation aids affected by our personal priorities and values and by our interaction with others constructing requirements (Floyd 1992 p.95).

During the case study that provides the empirical basis for this paper, we tried to understand how software developers actually deal with requirements during design and implementation. The following section introduces the project. I then discuss the methodological approach we used in greater detail. After the analysis of the empirical material, the issues discussed in the preceding paragraphs will be revisited.

3. The Project Studied

How to approach widely distributed software development projects through qualitative methods is a well-known difficulty (Newman 1998, Harper et al. 2000 pp.76-77) One reason is that the project members' roles often change with every project phase, that is, there is no recurrent stability. Another problem pertaining to the distribution of work is how and where to identify what is relevant for the study. An additional problem is how to recognise and make sense of distributed work taking place at different locations at the same time. The project studied was divided into four subprojects distributed over five different locations in Sweden. The sub projects in turn were divided and distributed. Contracted companies were involved in the project. Consequently, it was difficult to define the setting and approach of the study.

The company involved produces advanced products and systems for wired and mobile communications in public and private networks for customers in more than 100 countries and has a long history in the telecommunications field. The project studied was performed in and by a Swedish component of that organisation.
The project aimed at developing a graphical programming environment, including training and methods. The environment was supposed to handle the company's existing telecom code used in their telephone exchanges. The high-level programming language supported by the graphical environment is Specification and Description Language (SDL). The main project was named *SDL-Project*. SDL makes tool-supported code compilation into lower-level languages possible. This means that an SDL description can be translated into an executable application without manual coding, leading to shorter development time and increased quality\(^1\).

The SDL project included the SDL ToolCore subproject handling development of the code generator together with features of the tool. The following were also included: Training subproject handling and developing SDL training; Methods subproject handling the co-ordination of all SDL methods as well as standard methods; and SDL Tool subproject caring for signal handling, configuration management, release handling, function change, test port, and test methods. Two associated projects existed. The main technical orderer was located in Germany, and subprojects had technical orderers located in Sweden, Germany, the United States, and Spain.

The study began in the feasibility phase, and was completed during the 3\(^{rd}\) (execution) phase. The field material is built up of maintaining and taping project meetings such as system group meetings, project leader meetings, steering group meetings and the execution phase's kick-off day. By hanging around in the local field where the main projects management and the SDL Tool subproject were located. Non-taped interviews were held with code developers, and taped interviews with the main project manager, four sub-project managers, the product owner, the configuration manager, the project's quality responsible, and the main customer (main technical orderer). Because of distance, the interview with the main customer located in Germany took place by telephone. All other project members were interviewed at their local workplaces, with the concrete artefacts they used at work within reach to refer to. As can be seen, the study is somewhat top heavy, with the majority of the members studied being in some kind of management position. To place the study in the organisation and get responses on the
findings, a steering group was created consisting of two people from the university, the SDL Tool subproject manager, the product owner, and the maintenance project member. In that group, the decision was made to give the study a subproject perspective. The group had regular follow-up meetings where fieldwork and early writing results during the field study were discussed. This paper is the product of later retrospective reflections on the material. The next section discusses the methods used, their epistemological grounding, and the problems perceived when relating them to software engineering.

4. Methods

The empirical study was conducted during a five-months period through applying quick and dirty (Hughes et al. 1994) ethnomethodologically informed ethnography, supplemented by interviews as a way of uniting perceived field experiences. I use the term informed to refer to the approach that Hughes, Randall and Shapiro (1992) pioneered, as the first serious attempt I am aware of, to connect ethnomethodology (EM) with design issues in work practice. Recent reflections concerning the usefulness of EM-informed ethnography can be found in Harper, Randall, and Rouncefield (2000 pp.66-71). EM perceives the division of labour between the project members as routinely manifested in their own meaningful orientation to their work. Technology and work are, in this view, treated as technology-in-use, which is perceived as indivisible. EM is a highly descriptive framework with the goal of providing an alternative procedural description of achieved and achievable phenomena without sacrificing the describable, recognisable recurrences of ordinary activities. In this way, the perceived or recognised 'structure' in itself becomes the achieved phenomenon. This perspective may not seem obvious with respect to software engineering, where developed structures are valued in and of themselves, and where the goal of the research is to develop these new structures in the form of processes, methods, rules, and insights generally applicable to software development work. As a consequence, from an EM point of view, software engineering approaches also tend to obscure, 'misrepresent' or ignore the existence of a 'real world of work'. Instead the perception of work is derived from the model or structure of work rather than from
the activities in the work setting itself. EM-informed ethnography focuses on the activities themselves embedded in the socially organised domains—the locus of decision making—not in the structures developed.

It seems important to specify the epistemological foundation that the ethnography carried out in this project rests on, as a way of integrating other research perspectives (Anderson 1996 p.16). With respect to methodological assumptions, I take it for granted that phenomena in the social world are not constituted in such a way that they can be retrieved again by social scientists to enable scientific experiments, in contrast to the situation in the natural or physical sciences. Thus social scientists' conclusions remain speculative. I also assume that no one scientific method exists, in the sense ‘the method’, that is better suited to reveal the social organisation of the social setting's activities.

From an EM informed ethnographic viewpoint, a crucial question is whether researcher(s) have spent enough time in the field. Tracing patterns and identifying themes in a rigorously scientific manner is resolved through the use of the documentary method (Garfinkel, H. 1996b chap.3). The documentary method cannot be evaded; there is no ‘time out’. This method is employed without exceptions to establish the correspondence between the phenomena actually witnessed and the underlying phenomena studied. Investigators interpret what appears to happen based on documentary evidence from the corpus of their experienced knowledge from within that setting.

It makes a difference what the documentary method is applied to; whether to a description of the phenomena or to already-developed structures. In the latter case, the findings are also influenced by some scientists view, and by the circumstances occurring when the coded results are worked out, that is, circumstances surrounding the need to use scientific language to furnish a scientific way of to create consensus and action (Garfinkel 1996b p.24).

In both cases there exist the danger of ground an analysis upon the perceived coded structures instead of on the actual phenomenon. The problem is that the coded results, the secondary scientific constructions, can be taken to be a part of the actual social organisation they purport
to describe. That leaves us at the point where we started; the crucial point is whether the researchers have spent enough time in the field to grasp what is going on. To be able to interpret the ‘story’, it is necessary to have worked up enough feeling for how a ‘competent member’ in that field uses the documentary method.

To show the validity of my use of EM-influenced epistemology, I have to show how I handled this methodological problem. I have to explain my field material and its credibility from my own perspective. I have to show how and where I have spent time in the field (necessary in order to be able to make sense of the members' activities). And I have to keep to the EM commitment to show the ‘raw material’ in the resulting text (the justification I have for arguing that any particular thing is ‘going on’ should be evident, as ethnomethodology is highly concerned with the warrantability of data). I address these points in the following section, which present the result of EM-informed ethnography. Later I draw conclusions from the material shown.

Most of the ‘raw’ evidential material included in this paper is drawn from interviews and is not in itself the direct representation of the members' methods in action (which the transcribed meeting would be an example of). The interview material shown is actually retrospective 'talk about' the interviewees' methods in use. This conflicts to some extent with the EM idea, though ethnography is anything but a unified method and is perhaps not even a method at all as Shapiro (1994) has pointed out. Perhaps it is best regarded as an umbrella term for various analytic frameworks; in other words, it is a qualitative methodology. Yet, it is important to show the epistemology influencing the ethnography performed despite the tensions in the usage of the approach.

The next section presents part of a discussion from a steering group meeting as well as comments by subproject leaders, ending with suggestions for improving the project.
5. Requirements in a Distributed Project

The project members had problems getting organisational support in coming to terms with the scope of the requirements they were supposed to implement, despite the company's traditional way of managing projects. Through a document-driven waterfall-like model (or perhaps because of it), some requirements could only be traced with major difficulty; others could not be traced at all. The SDL-project preceded a reorganisation caused by a strategic overall reshuffling of the international firm's line-organisation. This reorganisation resulted in a reduction in scope of some of the subprojects. As a result, and also because these cut downs and partly caused of the complexity of the work distributed it became difficult to adhere to the project's original plans.

5.1 Problematic Regarding Requirements

5.1.1 Steering Group Meeting

At the time of the change from feasibility phase to execution phase, the different subprojects were not in phase with each other's and the project plan. Some parts of the necessary requirements specifications not yet finished were discussed during a steering group meeting:

**Standard methods-subproject leader:** At this point I had a question, how shall we describe it? Because the description in the implementation proposal (IP), because in the IP we describe, we do not have these requirements so to say, we will remove very much of the requirements, the issue will be totally different.

**Chairman:** To reach formalism in the...

**Standard methods-subproject leader:** Yes it has to be described in some way...

**Chairman:** Yes.

**Standard methods-project leader:** Shall it be done in the IP or in some other way?

**Customer-organization representative:** Draw it in an updated IP.

**Main technical orderer:** I do not believe we have the competence to exactly describe it...

**Standard methods-subproject leader:** No.
Main technical orderer: In order to support the project, we must have a new suggestion on what has to be done, we need new suggestions on that...

Chairman: In the project we really would want to be able to take the toll gate decision; can we handle the issue in some way that will not delay that decision any more?

Main project leader: The fact that this delays both the delivery plan, it is surely effected whatever case, this is what worries me.

Chairman: Not in all of the...

Main project leader: Yes in all, we will have to remove standard methods from delivery five and six, we have an open IP, we also have an open project specification, and an open main project specification.

Chairman: Is there no way we can close them, and still have them with us? Not to get stuck in a lot of...

Customer Organization representative: That is my opinion too, on how to solve this, because this is actually not the only ‘one’ we have, there is the training, we could also include the code generator. In my opinion it would not be wrong to close them, to give TG2 on the prerequisite that actually is the existing situation.

Main project leader: hmm.

Customer organization representative: Then we will have a TG2B later, or whatever you call it. (Steering group meeting March 5, 1999)

Despite the fact that the expected foundation for the TG2 decision did not exist, the suggestion from the customer-organisation representative was exactly that: to make the TG2 decision. The situation that occurred together with the perception agreed upon in the meeting actually became the foundation on which the TG2 decision was made. In this way, the decision reached in this meeting was not about the completeness of any requirements specification, but about collaborative theory building and mutual intelligibility. The result of the discussion was the development of a sustainable arrangement that would allow the project to continue without a rigorous implementation plan and clearly delineated requirements specifying the work to take place.

In the following sections I will elaborate on the difficulty the project members articulated in coming to terms with their requirements, and will also discuss a related issue: their problems in tracing requirement
originators. That brings to the surface the way project members described their requirements-handling work as a collaborative activity. In the first subsection I give two examples of how members perceived their trouble in handling requirements. In the next subsection I sketch solutions proposed by the members to the requirement problems. I then conclude the section by discussing what I suggested to help the situation.

5.1.2 ToolCore-Subproject Leader Problem

How to handle requirements was planned as a straightforward issue. The idea of their model was to deepen the conception of requirements through references in the documentation. On the first level, the requirements were put together to form an overview; on this level, there were a few lines of explanation connected to each requirement and references to other documents. If a reference was pursued more detailed explanations, its origin, the customer organisation, reference names, and other related documents should be found. But when discussing requirements with the project members, it often seemed that there was something about the model that did not make sense:

Yes there is, as I said before there is a reference pointing to another document where it is possible to read more exhaustively, at least sometimes, not every time, sometimes there only exist a few lines of text. This has caused a lot of major problems (ToolCore-subproject leader)

5.1.3 ToolCore-Subproject Developer Problem

When asking a project developer in the same subproject how he managed to grasp the requirements he was supposed to implement, he answered:

I don't, I mean I can't understand them only from the text in the requirements specification. What I do is, I contact the technical person that stands behind that specific requirement. The problem with this is that the person's name is never mentioned in the requirements specification, I mean the actual persons that once figured out the requirement (developer in the ToolCore-subproject).
Despite the fact that he was involved as an expert on compiler requirements in the formal project, he could not put his trust in the project's model for handling requirements.

5.1.4 Training-Subproject Leader Problem

The subproject leader in the Training project expressed the same confusion about coming to terms with requirements:

Yes what does that mean? Who is it written for? Why do they need that? What roughly are its contents, and what do they want it to look like? Key elements are missing there! And from this point to derive it to who has introduced these requirements. It is the customer, and the customer is an unknown quantity (Training-subproject leader).

5.2 Developing Solutions to the Requirements Problems

The project members deemed the reference-document plan unsatisfactory. When travelling around talking to project members, I stumbled on a number of different ways to solve this conflict, solutions developed independently by the different project members. They actually seemed to be unaware that this was a common problem. These meetings with the project members also showed that, on some occasions when the project members succeeded in locating requirements initiators, they actually did not always have the time to help them:

They perhaps thought that yes, yes, I will look at that problem soon and get back to you later, but they did not always get back to us. This has made things really difficult for us (Toolcore-subproject leader).

5.2.1 ToolCore-Subproject Leader Solution

The ToolCore subproject leader developed a sophisticated strategy of turning the problem around, actually making it the requirements-originator organisation’s problem:
In my subproject, during the autumn in the feasibility, I solved this. I chose to solve this problem as follows; if we did not manage to get a requirement clear, we simply decided to produce a solution. Out of that solution the customer then had to understand how we perceived the requirements (ToolCore-subproject leader).

5.2.2 ToolCore-Subproject Developer Solution

The code-developer from the same subproject presented another solution. On his own initiative he had established an informal network of contacts including technical persons all over the world, persons involved in the use of the earlier version of the code compiler now being tried out in real work settings:

I have to do my own research, starting to contact people, asking around to find out where this requirement originated. That means finding the technical individual responsible for every requirement, finding the people who originally wrote down the requirements, and other technical people as well, people who might have an interest in the specific requirements. I start asking around with the help of e-mail and the telephone, asking questions like: What does x mean, and why? Then negotiations arise concerning the requirements. It is very seldom that I don't use my contacts; sometimes there are up to six people involved in a discussion concerning a requirement, people from all over the world. It often takes days to get an answer (developer in the ToolCore-subproject).

The same technicians who were part of this informal network actively contacted the developer as well. A network of technical engineers had established outside both the project's formal structure and its model for documentation.

5.2.3 Training-Subproject Leader Solution

The Training-subproject leader approached the problem by travelling to his technical orderer, and ‘refused to leave’ before they had tracked down and talked to all the requirements-initiators together:
Requirements could come from separate individuals who later on had just disappeared. There is especially one requirement that has endured despite major protests from both the contractors and me; it is a requirement about a higher degree of simulation. But there is already a great deal of simulation. There is simulation all the time, you do a little design piece, then you simulate that, almost like doing a compiling test when coding, it is the same way with the high-level specification language. We started to sort out where this requirement came from, and who it could be attributed too. It turned out to be someone from Finland; it was a woman who did this off the top of her head at a meeting (Training-subproject leader).

This requirement was eliminated after the conversation with the Finnish requirements initiator. In fact fifteen of the original twenty-five requirements were removed as a result of spending weeks with tracing and negotiation work.

5.2.4 Proposed Method Changes as a Result of the Study

As a consequence of the study methodological refinements were proposed. These refinements included additions to the company's requirements reference model: names, references, and contact information regarding the requirements initiators or other individuals who could take responsibility for the requirements should be given together with other details pertaining to the requirements specifications. If requirements sources are unknown, difficult to track down or not available, that should be noted in the field reserved for contact information. The company, of course, already provided such information, but not on this level. As could be seen from the examples, the contact information previously available was insufficient. The new suggestion would save many hours time, in future distributed software development projects.

Implementing the solution suggested above is not as easy as it might appear. Crucial investments, responsibilities and organisational sacrifices have to be handled. In a large international company even a small change in the world-wide project management-model becomes a large effort. Changes of course have to be explained and changed in existing manuals and training programs, and distributed in the
organisation. How to change experienced project people who do not use manuals and are not the subject for training programs? Other things are, that employed now and then change their division and organisation, and employed in key positions perhaps do not wish to be, or even have the time to be disturbed. The ToolCore-subproject leader touched the problematic when expressing: I believe that the individuals who once wrote the requirement had a good grasp at that time, but when time passes. And of course they do not always detail the requirement in a large amount of text… As in this case, four lines. If you go back half a year later and ask this person again, then he or she perhaps does not even remember, even worse, they might have changed their opinion. A proposed change in methods of the type just outlined would benefit from following-up research aimed at showing how it adapts to and incorporates ‘ad hoc’ features not easily described or captured in a method recommendation.

6. Conclusions

6.1 Is Completeness the Core Problem?

By focusing on the actual ‘work’, this paper has highlighted mundane achievements during software requirements handling in a distributed software development project. These are achievements that otherwise risk passing unnoticed during the members' ordinary daily interactions.

In McDermid's (1994) view, the difficulty with requirements handling seems to have to do with finding the ‘right’ model to control the requirements process. He suggests a hybrid approach: an orthodox model incorporating features from the fundamental approach, as they address different problematic aspects of requirements handling. The present study led to a somewhat different insight into the developers' needs. It seemed as if the most prioritised need was to interact with someone who could take responsibility for being requirements initiator, having adequate authority to negotiate a requirements interpretation. From this standpoint, the complete specification of requirements would not actually make the difference hoped for. This seemed to be the aim
of McDermid's (1994) work. In other words, developing structured processes that lead to a complete requirements specification is perhaps not the core problem to be solved.

6.2 Human and Social Issues through Descriptions

A methodological problem occurred when relating the EM-informed ethnography to software engineering. EM is a truly descriptive epistemological framework, and as such it has nothing to say about design. Software engineering is about improving the software development practice. To make a difference in software engineering it is necessary to provide design proposals on the perceived problems. Replacing activities in work practice with ‘structures’ or tables conflicts with the EM epistemology. EM-informed ethnography describes the social world as it unfolds, avoiding any transformation of the perceived activities taking place. EM politely refuses to lend itself to any imaginary work, claiming that the ‘instances’ collected from the field speak for themselves and do not need to be organised around a core theory or cognitive model. EM is suspicious of the opposite tendencies, that is, to transform the ‘raw material’ by making it confirm to preconceived formats (Lynch, M. Bogen, D. 1996 pp.266-267). Without disputing such achievements, EM asks what more is there to be discovered that the artificial tend to obscure (Garfinkel 1996b p.6).

The conclusions I draw from applying this approach are the following.

First, it is possible to relate ‘human and social issues’ to software engineering without doing a transformation of the ‘raw’ evidential material.

Second, the raw material gleaned from the field is actually closer to the social phenomena than the interpreted versions are. I do not deny that the raw material has to be interpreted at times, as the discussion of the documentary method discussion earlier revealed. I also do not deny that we need certain formats or frameworks that make it possible to discuss important themes in creative work. But I argue that social practices as they actually occur cannot be discerned or judged through rationalisations built on imaginary work. Garfinkel has captured the
issue metaphorically by saying that it is: …very much like complaining that if the walls of a building were gotten out of the way, one could see better what was keeping the roof up (1996b p.22).

Third, EM-informed ethnography in itself seems to suit the task of studying an ongoing work practice, but it does not support the envisioning of possible future ways of developing software. It is merely a starting point by making visible the actual activities that have occurred; in other words, it brings to light the work of today that is to be changed.

Fourth, I am not totally convinced that EM-informed ethnography is the best methodological choice when studying distributed software development. The use of the interview material seems to bear this point out. Much of the field material included in this paper is project members talk about their own work-methods, i.e. not by the researcher perceived ‘methods’ in use. Within EM this is referred to as talkaboutable or storyable. This situation occurred because, as a researcher, I found their talk about their requirements extremely interesting and did not have an opportunity to observe the occasions where they actually did what they talked about in the interviews.

To the extent that the EM descriptive view has been borne out in this paper in some sense, it has been done in the spirit of regarding the result as one possible version of many others (Garfinkel 1996b). I also point to the humble attitude expressed by the founder of EM; I hope that there is room in this discussion for those studies which take the importance of witness able recurrent phenomenal fields of detail seriously and as a primary issue, in whatever other respects they may differ (Garfinkel 1996a p.6).

Acknowledgements

I would like to express my appreciation to the other contributors of the grounding section of the book Social Thinking - Software Practice: Yvonne Dittrich, Urs Andelfinger, Jacob Nørbjerg and Philip Kraft, who have questioned, clarified and made useful suggestions on various points in the paper. Thanks also Bo Helgeson, Jeanette Blomberg, and
Dave Randall who in their roles as course leaders in the doctoral course *Work Practice and Technology II* provided helpful comments on an early version of the paper. Thanks as well to the course participants, especially Hans Tap who critiqued the EM material.

**Notes**

1. This description is borrowed from the company’s own Web site.
2. Garfinkel (1996b p.78) has expressed that *The method consists of treating an actual appearance as ‘the document of,’ as ‘pointing to,’ as ‘standing on behalf of’ a presupposed underlying pattern. Not only is the underlying pattern derived from its individual documentary evidences, but the individual documentary evidences, in their turn, are interpreted on the basis of ‘what is known’ about the underlying pattern. Each is used to elaborate the other.” To decide a correct correspondence is a question of producing through interpretive work a correspondence that members of a community of cobelievers would agree upon, that is, on “common sense knowledge of social structures* (Garfinkel 1996 pp.96, 76).
3. Lynch and Bogen (1996 p.281) note that *Strange as it may sound, one must establish a right to have seen something and to have seen it that way (as storyable). What the members talked about in the taped conversations was the perceived real-world problems that bothered them. This was talk about the actual unfolding-work-practice side of ‘structures’ such as plans and methods.*

**References**


Presentation of Paper IV

In the volume *The Future of Software Engineering* (Finkelstein 2000) Fuggetta provides a representation of the ‘software process’ theme. He defines this subject as the research that *deals with methods and technologies used to assess, support, and improve software development activities* (Fuggetta 2000 p.27). The issue of software development is not just a matter of introducing effective tools and environments, neither is it a matter of selecting a reasonable lifecycle; rather it is a question of paying attention to the complex interrelation of cultural, organisational, technological and economic factors (Ibid).

Fuggetta actually goes one step further than ‘paying attention to these complex interrelations’: he asks if we are not, in fact, trying to model and automate something that intrinsically can neither be automated nor modelled. In this context he also criticises the software engineering community for failing to analyse sufficiently and learn from work accomplished by others in other research communities, something that he believes has slowed down the rate of innovation in software engineering (Ibid.). Fuggetta has also issued the challenge of taking considerable investments in finding and evaluating similarities with other research areas instead of focusing on what seems to be artificial differences, which he claims is the major problem today (Fuggetta 2000).

Paper IV discusses an issue which is little debated in software engineering despite the fact that it is an unavoidable key theme in all kinds of software development processes, and much research within software engineering in fact produces models to support the subject.
The subject is ‘what is a plan’, and what is entailed in ‘planning’? The empirical study used to inform this discussion is the ‘original conception’ of ethnography. The starting point of this study was not to test general assumptions or hypotheses on the basis of general method and model descriptions; instead, it was an ‘inside’ explanatory perspective which emphasises ‘practical’ theorising directed towards such practical issues as ‘how did things end up this way?’, and ‘what is going on in this specific setting?’

This paper is based on an analysis of the same empirical material as that presented in Paper III. This paper invites the reader to reflect on how plans are developed, referred to and used by software engineers in co-ordinating their work. Plans in software engineering are used to control action. They are also used as tools for making work within the project visible to others involved in the same project. This latter aspect is clearly apparent in the analysis included in the paper. In the project studied it is demonstrated how a document becomes a plan as it is developed and negotiated by and related to by all the different project members who co-ordinate and relate their contributions both to the common project and the customer organisation. The documents combined with praxis made the plan a plan. An occurred problematic status in the project was dealt with ‘as problems with documentation’; the nature of ‘planning this situation’ proved to be an issue of accountability.

The analysis presented is also used to challenge the discourse in the Computer Supported Cooperative Work concerning the nature of ‘plans’, and suggests that much of the apparent confusion relating to the relationship between plans and situated actions is caused by implicit disagreement about whether the problem is a conceptual or an empirical one. The paper argues that, from an empirical perspective, plans can take a variety of forms, which may be applied in different ways and at different periods of time.
Abstract

Based on empirical material from the area of distributed software engineering, this article discusses the issue of plans and planning in the Computer Supported Cooperative Work community as contrasted with the approach taken in software engineering. It suggests that insights concerning the nature of ‘plans’ and ‘planning’ in Computer Supported Cooperative Work are valuable insofar as they draw attention to features of the distributed software engineering process largely overlooked by the software engineering community - features of particular significance for distributed software engineering projects. The analysis of the empirical material allows us to show how particular kinds of problem arise and are dealt with in one project, problems that are best thought of as problems of ‘organisational memory’. We suggest that developing support for software engineers depends on recognising their practical problems and that this in turn requires a turn to qualitative empirical studies that they have been hitherto unwilling to make.

1. Introduction

In this article we present some field material from a study of a distributed software tool and method development project in a telecommunication company. We compare and contrast Computer Supported Cooperative Work and software engineering interests in the study of such projects and how their different conceptions of ‘planning’ methodologies are implicated in these interests. We argue that the analytic focus of ethnographic enquiry into software engineering allows us to identify and reflect on different questions, which have to do with the nature of ‘organisational memory’. Moreover, we suggest that the particular conception of ‘plans’ and ‘planning’ developed within Computer Supported Cooperative Work, far from being irrelevant to the more prescriptive interests of the software engineering community, may help focus on problematic features of distributed software engineering that may otherwise be substantially overlooked. These have to do with how documentation is developed, referred to and used by project members to co-ordinate their work fails to provide relevant information about ‘how’ and ‘who’ – in short to cope with the complexity of a highly distributed design project. We further suggest that so-called
qualitative studies of software engineering and elsewhere (e.g. Button and Sharrock 1994) become more rather than less significant as we move to more distributed environments for software development.

The background to this is the apparent contrast between the understanding of ‘plans’ and ‘planning’ to be found in the software engineering community and the somewhat different version which typifies Computer Supported Cooperative Work interests. We do not here wish to stress the superiority of one conception over another, rather to argue that these different conceptions lead to very different strategies for understanding the software development process and in turn to the identification of different kinds of ‘problem’ to be dealt with. That is, we want to suggest that the analytic issues surrounding Suchman’s (1987) original formulation of the relationship between plans and situated actions have been vibrant in Computer Supported Cooperative Work, but less so in the software engineering community perhaps because the latter remain unconvinced as to how these issues inform ‘better’ plans and planning. We attempt to show below that empirical enquiry into software engineering can inform the software engineering community, and are likely to prove more important as we move into complex distributed planning environments. We argue that general questions of the form, ‘do plans prescribe action?’, while important, matter less than understanding the degree to which, and in what circumstances this relationship can be problematic (Bardram, 1997; Schmidt, 1997). Ultimately, and in keeping with general arguments from Conklin (1993), these issues can identify the kinds of support for software engineering that are required.

2. Plans in Software Engineering

Software development, as a design practice, is highly discretionary in that its outcomes cannot be easily predicted. At the same time- as with any engineering activity – it is a highly ‘planful’ one as well. This might be no coincidence. Planning and plans in software engineering are used to cope with complex tasks, allowing members to co-ordinate different activities across time and space in order to produce something which will be accepted as ‘achieving a common goal’, and all in the obvious context of both budget and time constraints. In software
engineering methodologies, plans, planning and project management are therefore an important theme. In order to cope with the growing complexity of software development projects, project models and planning tools were of course developed to co-ordinate the co-operative effort. The first project models were, therefore, highly ‘planful’ and drew on models from other engineering disciplines; and specified regular moves from high-level, relatively abstract plans to implementation (McDermid and Rook 1993). This conception resulted in detailed so-called ‘waterfall models’; wherein several intermediate steps and representations were designed to refine the path toward implementation. Other developments concerned the possibility of computer support for software engineering. Thus, Osterweil (1987) in what is generally agreed to be one of the most influential papers on software engineering of the last twenty years (and which arguably led to the development of CASE tools) argues that software process descriptions can be understood as steering the development process in exactly the same way as a program steers a computer. Critics, such as Lehman (1987), nevertheless see the search in terms of a series of process models, emphasising instantiation in terms of relevant concepts, available technologies, specific implementation environments, process constraints and so on (Ibid. p.14).

Other contributions such as iterative models (Boehm 1988) and evolutionary models (Floyd et al. 1989) have been developed as remedies to a perceived rigidity, and which explicitly take into consideration the specific interplay of software and its embedding context (Lehman 1980; Boehm 1988). This, largely because experience from real world project from the very beginning taught a very different lesson from that anticipated within ‘structured’ design. It turned out, that plans really do underdetermine outcomes. Thus, Parnas and Clements (1986) discuss the mismatch between the ideal and the practice, and propose a different way to address the issues which takes into account the ‘the imperfectness of the world’ and the human way of doing things. They propose to fake the rational design process; that means to follow it as closely as possible in order to have a common means of co-ordination and orientation (Parnas and Clements 1986 p.252) and to produce documentation as if the final product had been produced in a rational way. They even propose to extend it with a
description of design alternatives and the reasons for disposing of them. As such, they argue that documentation is an important resource throughout the project and during maintenance and further development (Parnas and Clements 1986 p.256). There are important issues here in that no one in the software engineering community would doubt for a moment that such methods of co-ordinating are vital resources, but at one and the same time many would share Parnas and Clements’ doubts about the ‘rationality’ of the process. Parnas and Clement propose a different take on the problem by proposing a form of documentation that helps software developers to co-ordinate their efforts within team and across time.

There are, of course, many competing views of the software engineering process within that community, and we do not wish to over-simplify evidently complex discussion. Nor do we wish to pretend that software engineers are not aware of the practical ways in which their work either does or does not orient to ‘planfulness’ in some way. Rather, our interest here is in two different orientations to plans and planning. Unsurprisingly, for software engineers, the business of understanding plans is a prescriptive business. That is, their methods are intended to result in better and more certain results, as well as aid with specific problems of project management.

3. CSCW, Plans and Situated Action

In our view, the Computer Supported Cooperative Work (CSCW) community has taken a more analytic and less prescriptive view of plans and planning. The focus, in other words, has been on empirical studies of the software engineering process, evaluation of the consequences of new plan-like methodologies (Bowers et al. 1995) and on discussions centring on the nature of plans themselves. The ‘qualitative’ nature of many of these studies may explain why it is that lessons from Computer Supported Cooperative Work have not in the main been taken up within the software engineering community. Thus, Fugetta (2000 p.31) argues, Consequently, we have basically assumed that it was inappropriate and even impossible to reuse the approaches and results produced by other communities (e.g. workflow and CSCW).

\[9\]Leaving aside such forays into prescriptiveness as the Suchman/Winograd debate
Indeed, this attitude has caused a major problem. If Fugetta is right and software engineering can learn from Computer Supported Cooperative Work, then we need better empirical accounts which identify how. We are in no doubt that current approaches to software process improvement leave us with the problem of how plans are created, used and developed in everyday project practice so that they serve as a medium for co-ordination and organisation of highly complex development processes. As we suggest below, Computer Supported Cooperative Work has already taken some steps towards identifying the pertinent issues.

We have suggested that interest in ‘plans’ in software engineering is less analytical than it is prescriptive. In other words, the interest has largely to do with how to make and maintain better plans than have existed before. Nevertheless, these policy statements do seem to be predicated on certain kinds of assumption which, on the face of it, would be met with some hostility within Computer Supported Cooperative Work, and which have to do with the way plans will in some sense determine what gets done. In contrast, discussion in Computer Supported Cooperative Work has centred on the relationship between plans and ‘what gets done’ and the degree to which it matters that plans can be conceptualised as taking a variety of forms, including their possible cognitive, documentary and artifactual character. Indeed, much of the discussion that has hitherto taken place within Computer Supported Cooperative Work is predicated on examining the similarities or differences between these forms.

The problem of the co-ordination of work in the distributed organisation is of course a central concern of Computer Supported Cooperative Work, a concern sometimes, and mistakenly, expressed through a typical contrast between the ‘plan’, construed as a formal model of a structure of tasks within an organisational model of some description as against the occasioned, or ‘situated’ practices of members, and obtaining from Suchman’s (1987) seminal work, Plans and Situated Actions. Suchman’s original critique was initially founded on the misconceptions inherent in the ‘computational’ theory of mind dominant in cognitive science at that time, and which was leading research efforts in Artificial Intelligence. Assessing the value or
otherwise of various ‘computational’ models (see Button et al. 1995; Dreyfus 1997) is no part of our interest here, but we would want to make one simple and largely uncontroversial point, which is that Suchman’s argument (1987 Chap 1) concerns ‘plans’ as mental representations, and has no consequences whatsoever for any other conception of ‘plan’. Here, Suchman rejects the view that human action is or can be determined by cognitive models which may consist in goals/means hierarchies; representational schema and/or fixed semantic and syntactic criteria. Thus: The basic premise is twofold: first, that what traditional behavioral sciences take to be cognitive phenomena have an essential relationship to a publicly available, collaboratively organized world of artifacts and action… (1987 p.50) and, There are no logical formulae for recognizing the intent of some behavior independent of context (1987 p.64). Moreover, and importantly, Suchman makes it clear that neither can interaction be understood in this manner- that it must involve more than the deciphering of the plans of others through mechanisms of ‘shared symbolism’. Put simply, Suchman is rejecting the use of a technical term, ‘plan’, by suggesting it cannot fully accomplish human action.

Even so, Suchman’s work has proven controversial; not, we argue, for the insights described above, but because continued confusion about the status of her argument may well explain why software engineering has, for the most part, not taken up the prospect of qualitative empirical study of the software process. One reading of the relationship between plans and situated actions is that plans ‘necessarily’ underdetermine situated actions because, in Wittgensteinian terms, ‘no rule dictates its own application’. This view permeates much ethnomethodological thinking about these matters, but is also a position which has produced some critical response, most notably from Nardi (1996).

---

10 In a very simple sense, the major lesson of the emphasis on ‘situatedness’ is that understanding organizational work should entail ‘going out and looking’- the ‘ethnographic’ stance- rather than relying on idealized conceptions of ‘process’, ‘plan’, ‘role’ or what have you. This is arguably now foundational to the Computer Supported Cooperative Work community, but remains mysterious, as Fugetta (ibid) shows, to others.
Nardi’s position is that the failure of situated action theory lies in its behaviouristic assumptions, which lead this ‘theory’ into presuming that plans are retrospective reconstructions. Nardi further, and quite correctly, points to the refusal of parties to situated action analyse to engage with cognitive notions of goal or intention. She further points to the failure of situated action analysis to deal with persistent structures such as artefacts, institutions, and cultural values ..., and suggests a tension exists in such analyses between their concern for the emergent, contingent, improvisatory and that which is routine and predictable. Thus, the ‘appearance of routines in situated action models opens a chink in the situated armour with respect to mental representations.’ Nardi, then, has a clear conception of what is entailed in the ethnomethodological conception of situatedness vis-à-vis the plan. It is a more or less behaviourist position, relying on objectivist positions concerning some kinds of data rather than others, and dealing contrastively, indeed tensely, with plans and situated actions. Plans in this view are evidence of some form of cognition such as mental representation (a position Nardi evidently wishes to defend). The weakness of ‘situated action analysis’ is according to Nardi that it cannot deal with the intentionality of plans, and is thus forced to describe them as retrospective reconstructions. Critics of Suchman like Nardi read her as suggesting that plans and, conversely situated actions, are different types of phenomenon, such that plans constitute one way of doing things, and ‘situatedness’ another.

Readers may, at this point, be wondering why the debate about plans is relevant to the practicalities of software engineering at all. We reiterate that it may well explain the indifference of software engineers to empirical studies of their work. Readings like Nardi’s might confirm the suspicions of the software engineer, who already knows that the ‘plan’ ought to help but that it will prove fallible at certain moments and who might question, entirely legitimately, what is to be gained by emphasising situated action at the expense of the plan. The answer, of course, is that there is no reason to accept what Bardram (1997) rightly

---

11 We leave aside here the fact that Suchman quite explicitly states that plans may exist before or after action. The consequence is that rather than plans being retrospective reconstruction’s, they may be organizationally purposeful constructions. The difference is hugely significant.
calls a *false dichotomy*. The critique of cognitivism that Suchman presents is clear and unambiguous - it specifies, no more and no less, that the principles that underpin some forms of cognitive science cannot fully account for human behaviour. It is important to be clear, because the source of the confusion lies in the conflation of quite ordinary terms like ‘thinking’, ‘memory’, ‘plan’ and so on - terms with which everyone is familiar- with the same terms as they are sometimes used within cognitive science- to express a scientific and causal view of the relationship between ‘things which exist in functional relationships’ in the mind and what goes on in the world.\(^\text{12}\) In this way, when Suchman suggests: *When situated action becomes in some way problematic, rules and procedures are explicated for purposes of deliberation and the action, which is otherwise neither rule-based nor procedural, is then made accountable to them* (1987 p.54) there is no particular reason to infer that Suchman believes that actors in the course of their work do not orient to rules in one way or another, for it is absolutely evident that they do. Again, we can simply see such statements as suggesting that actors do not need to ‘habitually’ refer to rules, implicit or explicit, in order to determine what to do, neither does the existence of rules determine what people will do. Rather, the relevance of the rules will be occasioned by the context of their use.

\(^{12}\) Ethnomethodology, for good or ill, is concerned with critiquing a model of the relationship between the mind and the world, not with denying that people plan and think. No variety of ethnomethodology that these authors are familiar with would subscribe to a view which stated in response to any ordinary language proposition about memory, thinking, or for that matter planning, ‘nonsense, human beings do not [think, remember, plan]’. The point that ethnomethodologists have in fact consistently made over a number of years in different contexts is that thinking, remembering, planning or what have you are not necessary but occasioned phenomena. An interest in ‘situatedness’ is not founded on a belief that human beings never think, remember, reflect or plan but that when they do, they do so in quite specific circumstances. Equally, and related to this, there is no need to infer from ethnomethodological interest in ‘situatedness’ that such analysis cannot analyse and account for stable and persistent phenomena. The apparent ‘tension’ between the contingent and the routine turns out to be no kind of tension at all. Seen from the point of view of actors, it is a perfectly ordinary and commonplace suggestion that ‘things crop up’ whilst also recognizing that this kind of work is ‘just routine’.
This alternative conception, one which Bardram (ibid) for instance is insistent upon, treats ‘plans’ as multi-faceted. He points out that ‘plans’ can refer not only to cognitive phenomena, but also to artefacts of one kind or another. Moreover, instructions, lists, process models, schedules, etc are extremely valuable as mechanisms for giving order to work (1997 p.18). He goes on to show how various documents are used and socially constructed in and through the intersubjective understanding and use of members of a community. Bardram makes a useful move in suggesting that the status of physical artefacts should be seen in terms of the way in which they mediate human activities, and thus that plans can usefully be conceptualised as planning. That is, he argues that plans are situated actions; that planning is something that people do and thus is itself irredeemably situated. If so, then the consequence is clear. As Schmidt (1997) has argued, no particular conceptual apparatus needs to be imposed on the relationship between plans and situated actions, because it will be empirically determined. Thus and for instance, he is at great pains to show that it makes no sense to argue that situated actions are only ever very loosely coupled with plans. That is, for Schmidt, seeing plans as underdetermining action can in practice (but not as a matter of logical consequence) lead one to failing to recognise the ordinary but important ways in which various kinds of plan get used. This rendering, whereby we are invited to see the use of plans as maps and/or scripts has important consequences, for it opens the way for a detailed and nuanced view of plans, planning and planfulness. Examining the practical ways in which ‘plans’, which may include different types of artefacts and documentation, get used ought to tell us something useful about the software engineering process. This, as we argue below, is particularly pertinent to problems encountered in distributed software engineering for it may help us both understand the practices around different documents that are used as plans, but also to understand the occasions when, for participants, the documents ‘fail’. We will argue that the analytic perspectives associated with the above perspective enable us to identify in some detail the problems that arise in a software engineering project that result specifically from what Conklin (1993) calls a product oriented rather than process oriented form of documentation.

A number of empirical studies which examine the existence of methodologies, artefacts and other plans in conjunction with the fact of ordinary and practical contingencies arising have been conducted, including for brief mention Bardram (ibid), Bowers et al, (1995). Of more specific interest to us are those studies which have software engineering as their focus, and more precisely attempting ‘to understand the relationship between methods and systems development’ (Button and Sharrock, 1994). Button and Sharrock focus on the way in which the work of software engineers consists of making the method organisationally accountable in the practices of making it work and consequentially, how the development methodology which was adopted was used for purposes other than its strictly technical one. As a result the technical requirements were frequently subordinated to other, organisational, demands (p.218). Some features of these organisational demands include those of project management. Thus, Recognising that meeting deadlines- and enabling others to meet theirs'- so that the project would not be ‘held up’ was the dominant practical priority, and the engineers would suspend the operation of the methodology to ensure that software could be provided. They had not given up on the objective of producing good software, but, for the moment, the working out of the software design according to the method had to be postponed in recognition of overriding organisational realities (p.221).

Button and Sharrock further recount how various contingencies arose which led to progressive changes to the character of the project in question, and in particular how the adequacy of documentation became an issue. Tellingly, ‘A contrast was drawn between ‘how things were’ and with how they were to be. How things had been, the lack of proper documentation had in part caused the problem that was now faced, thus ensuring that there would be proper documentation accompanying the development would help mitigate the possibility of a similar outcome in the future (p.225). That is, the issue of ‘proper’, or organisationally relevant, documentation, was central to the development of the project in question. This work serves to illustrate a number of important points, two of which are particularly germane to our study, and the first of which relates to empirical enquiry. It is that it is perfectly possible to provide detailed and sophisticated empirical accounts of software
engineering which reveal a great deal about how participants orient to documentation in (for them) quite ordinary and mundane ways and which do not merely rehearse the argument that plans turn out to be inadequate. Secondly, they show in a range of ways how the history of any given project is organic. That is, and unsurprisingly to experienced members of such projects, how original aims, tools, methods and even programming languages can change during the evolution of the project. Indeed, their maxims for requirements analysis, notably maxims 6 and 7 that, a requirement is a gloss for a swarm of changing contingencies’, and, ‘capturing a requirement is like capturing a butterfly, once it’s pinned down it’s no longer what you chased, it’s dead (Ibid. p.239) are emblematic of most of our concerns. Our interest below is to extend these themes into a ‘distributed’ software engineering project, and to specifically address the theme of how documentation turns out to be ‘proper’ or not in that context. This, as we argue below, is particularly pertinent to problems encountered in such projects, for it may help us do the analytic work of understanding the practices around different documents that are used as plans, but also to provide a basis for the ‘prescriptive work’, i.e. the development of better ways of planning and better roles for it, that interests the software engineering community.

5. The SDL Project and Observation

The cases we recount are taken from a software tool & method development project. The study in question was an ethnographic one, and took place during a five months period within a major company making equipment for telecommunications systems and related terminals. The company involved produces advanced products and systems for wired and mobile communications in public and private networks for customers in more than 100 countries and has a long history in the telecommunications field. The project studied was performed in and by a Swedish component of that organisation. The project aimed at developing a graphical programming environment, including a set of tools and methods (the latter would include documentation templates for different tasks, and guidelines how to apply them and so on) for its application and training. The graphical programming language supported by the environment is the Specification and Description Language (SDL). SDL makes tool-
supported code compilation into lower-level languages possible. This means that an SDL description can be translated into an executable application without manual coding.

The SDL project included the SDL ToolCore subproject handling development of the code generator together with features on the tool. The other subprojects were: the Training subproject handling and developing SDL training; the Methods subproject handling the co-ordination of all SDL methods (standard methods were developed according to the world-wide organisation ‘s documentation standard); and the SDL Tool subproject caring for signal handling, configuration management, release handling, function change, test port, and test methods. The project was performed in and by a Swedish part of the organisation, with the main customers placed in Germany and the end users in USA, Spain, Sweden and Germany. In Sweden, several departments of the company were situated in different locations and a sub contractor was also involved.

The project was undertaken using the company’s standardised model for project management. The project model was conceived within the company as meeting the need for a uniform terminology and a common view of the work procedures in projects within the organisation. It describes not only the different phases of the project, but also the procedures for decision-making, and thus served as a tool for the project sponsors (customers). Indeed, many customers actually required that the project management model should be followed. The model describes what to do in the different phases of the project, what documentation should exist, and what decision points and roles within projects should be established. It did not, however, describe or put any restrictions on the content of the project itself. Very roughly, the project model consisted of four phases, as follows:

- **The Prestudy phase**, where the feasibility from both technical and commercial viewpoints is established, and expected outcomes and time and cost limits for the project are established. The precondition for this phase is an **assignment specification document**, And the sponsor (i.e. primary customer) is responsible for this document.
• The second phase is the **Feasibility phase**. The purpose of this phase is to form a basis for the future project and to prepare for a successful execution phase. Preconditions for this phase are: a **Tollgate 1**; in effect a decision to start the feasibility phase; and the **assignment specification document**, as well as a **Requirements Specification** for the project; and an **Implementation Sketch**, defining what will be required in terms of resources, costs and time.

• The third phase is the **Execution phase**. The purpose of this phase is to execute the project as planned. Preconditions for this phase are: an updated **Requirements Specification**; an **Implementation Proposal**, describing how each project should implement their project assignments; and an approved **Project Specification**- the basis for contracts and all reporting during the project. A **Tollgate 2** decision signals the start of this phase. The execution phase also includes **Tollgate 3, 4, and 5** decisions. **Tollgate 3** is the first control point where continued execution is considered.

• The fourth phase is the **Conclusion phase**. The purpose of this phase is to break up the project organisation, see to all outstanding matters, and to compile a record of all experiences. The precondition for this phase is a **Tollgate 5**.

The project also included a whole set of roles and groups, below are presented the ones necessary for understanding the empirical material:

• **Main Project leader** (main PM) responsible for co-ordination of all subproject activities in order to achieve the overall project goal, and to deliver the product to the customer in time. A **project assistant** supported the main project leader, together with four **Subproject leaders** (sub PM) responsible for the different subprojects. The main- and the subproject leaders had a weekly telephone conference meeting in order to co-ordinate and inform each other concerning status in the different subprojects.

• **The Sponsor** who, together with the **Technical Orderer (TO)** was a figure from within the customer organisation. The main project had a main TO with co-ordination responsibility for sub
TO’s connected to each sub-project. The TO was supposed to represent the customer, give feedback and act as liaison during the project.

- **Product Owner** (PO) belonging to the line organisation, responsible for the long-term perspective and development of the product. The Product owner was included in the steering group and in the product management group.
- **The Steering Group**, which consisted of a chairman from the line organisation, the main Technical Orderer, the Sponsor, the Product Owner, and four experienced people from within the company (two representatives from the customer organisation and two from their own organisation). This group took all TG decisions.
- **The Main Project Management group** that consisted of the main project manager, the assistant project manager, the quality co-ordinator, and the configuration Manager.

The study began in the feasibility phase, and was completed during the 3rd (execution) phase. Field material was gathered at various meetings, and informal interviews were held with various project members, including code developers, the main PM, the four sub PM’s, the PO, the CM, the project's QP, and the main TO.

The material we analyse here comes from informal interviews regarding the update of requirements during the feasibility phase and a small part of a project management meeting where an important decision regarding the start of the execution phase was at stake. We have selected data to highlight the issue of how requirements documentation is followed up, problematised and revised according to the way in which organisational members construe the existing state of affairs and more specifically how it is that certain aspects of this documentation problematise its status as ‘proper’ documentation.
6. Practical Problems and Practical Solutions

6.1 ‘Yes ... What Does That Mean?’- The Practical Problems of Requirements Work.

According to the company wide project model, the requirements specification produced in the presstudy phase should be used as an input to specify the final product, write an implementation proposal and budget the further process in the feasibility phase. During the latter phase the requirements specification changed character from a collection of users’ needs to a starting point in the implementation process. The ‘what’ and the ‘how’ had to become more concrete. As the requirements specification also documented an agreement between different actors regarding the scope of the project, a plan regarding the ‘what’ of the development, it could not be changed without renegotiations. In that context, the document was ‘used ’ as a means to re-negotiate what the product should look like. One aspect of the difficulties is that this organisation and with it the project was distributed across a number of locations. In the material, requirements as expressed in the requirements specification document are the primary focus of the talk. We examine how, in the face of spatial and temporal distribution, communication, negotiation and (re-)formulation are done. Ambiguities are resolved, decisions are made, and changes accounted for. Our interest lies in precisely how these troubles arise in the first place, highlighting problems encountered during the process of producing usable requirements, and then how troubles are typically dealt with. One of the core features of the practices of project members however, as we shall show, is that there is no single strategy for resolving these problems. The interviews involve various project members. Our extracts are selected to point to the specific problems of interpretation that project members encounter. Hence:

There’s another thing when you have this amount of requirements ... you discover, when you have an understanding of the technique to be developed, and you read them all through ... that these requirements do not form a coherent whole. One person has done something here and another something there; there’s one person’s requirement from Germany, and another’s from USA and so on. Then they have brought
all the requirements together in a document and sent that to us, without having an umbrella view regarding the requirements. I went directly to the customer, I said this is problematic, if we implement this, then okay you will get a lot of features that in themselves are good, but they don’t form a whole. If I had been the customer I would at first pick out a goal, and then, okay this is what we are trying to reach, this is what we want with the product, out of this we now can pick out the requirements. This has not been the case, every individual has picked their own requirements in this project. The reason for this is of course that there hasn’t been enough time to handle it in another way. (ToolCore-subproject leader)

The first trouble, then, is the prospect of lack of coherence. The requirements specification document here stands proxy, as does any other document in organisational use, for knowledge and/or assumptions about the nature of the organisation itself, and equally importantly for inter-organisational relations. The Tool Core sub PM construes the apparent diversity of requirements as evidencing the problematic history of the project and its lack of coherence. To this project member, these distributed requirements did not constitute a solution to the project’s problem; what to design and build. For project members there was an issue of organisational accountability insofar as it was obvious that the disparity of requirements at this stage did not provide a clear foundation for the implementation. The complaint therefore can be heard as a complaint about responsibility. On the face of it this is surprising, given that the project had a set of clearly defined organisational roles. The problems, however, had to do with the size and distribution of the project. The lack of coherence was a function not of a badly managed project but of the inability of members to account for who might have been involved in the production of requirements and why they might have made the decisions they made, especially given that the individuals concerned may well have been from different organisations. That is, written requirements stand proxy for the purposes of the persons who formulated them. In the absence of any personal knowledge of these people, a set of assumptions about the motivation for any given formulation have to be made, and such ‘motivational’ work is extremely difficult in the distributed context.
These uncertainties have an associated consequence, which is that where there are ambiguities in the documentation there will also be ambiguities about their motivation. That is, requirements as stated do not always provide a warrant for deciding one way or another what requirements should be included. This issue of organisational accountability has powerful consequences. The mundane trouble of coming to terms with diverse suggestions concerning requirements then necessitates interaction with various different people in different kinds of roles. At this point, the disparate and individual collection of requirements begins to evidence possible conflicts of interest:

I try to understand the requirements. Sometimes I get additional information by contacting the Technical Orderer. Some parts I discussed with the Product Management, in order to understand the meaning behind. Other parts I discussed with the Technical Orderer for the methods subproject and sometimes the main Technical Orderer as well. Sometimes in order to avoid misunderstandings I was in contact with both and some questions I discussed with the System group. I also discussed certain matters with other designers in order to reach mutual understanding. (Tool/Methods-subproject leader)

Different accounts and expectations among the distributed members have to be made visible in order to enable decisions to be taken and to justify these decisions and their rationales. The point here is that justification is made precisely because all those with interests in the project require an account. The Training sub PM also talked about how he had to produce his sub-projects requirements handling plan through an evolving consensus:

… the requirements came in the middle of February from the customer. I was expected to have an IP ready in the end of February. They came by mail so I called up the people behind the first developed course and discussed the requirements with that person and asked: what do you think of this, that and that, and I think this about this and so on. In the end we reach some kind of picture about this, and also about what should be taken away, and things we don’t understand. Then back to the customer with the feedback, ‘we don’t understand this, and we don’t think this is relevant, have you really examined this requirement with
your people? In this manner, back and forth for some weeks. This ended up with the original twenty-five requirements in the end being reduced to ten. Perhaps all project leaders don’t do it this way, some accept the requirements directly and put them in their implementation proposal, then they have the problems when they are supposed to handle these thirty requirements instead. Perhaps not everybody has a relationship with their Technical Orderer that allows them to be so blunt. But I need to reach a consensus about what to do with the customer. Before we do it. (Training-subproject leader)

Again, the point here is not that requirements documents cannot be understood, but they need to be understood in a way that adequately accounts for what various people ‘really’ want. Some discounting work is done in reducing the requirements from 25 to 10. This discounting, again, is a matter of producing the intelligibility and rationality behind the suggested documented requirements. It is evidently a negotiative process. This is not to say that the business of ‘finding the right account’ is unproblematic, for it is not. Some requirements turn out to be more troublesome than others to handle. This is evident in the following brief piece of data, taken from a steering group meeting directed at the obligatory TG2 decision. The Main Technical Orderer brings up the issue of test methods:

1. **Main technical orderer**: On the subject of test methods …. It’s not actually that clear what exactly will be part of ‘test methods’ … it’s hard to say …
2. **Customer organisation representative**: Because it is based on that prototyping.
3. **Main technical orderer**: Yes that’s right, and the prototype runs in our part of the organisation, so I hope that the result on the prototype will be of ‘our’ type.

The obvious question here is what is meant by our part of the organisation and do other participants recognise its implications?\(^\text{13}\) Regardless, members are clearly orienting to their own, ‘stakeholders’ sense of the requirements at issue. The problem for the customer

\(^{13}\) We cannot answer our own question here, for that would require a lengthy organizational ethnography that few have attempted.
representative and the Main Technical Orderer, both of whom are members of the customer organisation, is that these methods have been developed with no reference to ‘their’ requirements. What had been originally planned for- using a standard methodology with generic applicability- had been somewhat changed so as to handle only the customer organisation ’s specific needs. This change in essence meant a substantial reduction in the scope of the project. In a nutshell, the customer representatives here are expressing doubts about the form of documentation used in the prototypical project and in turn expressing a preference for our type of prototype (line 3), i.e. one which reflected their local practices (The context here is that another user-organisation, connected to one of the sub-projects, has already proffered one solution; a solution that does not fit with the customer organisation ’s software and practice.) In any event, the troubles arising here arise because practices and rationales which are taken for granted in one part of the project are opaque to members in other parts. The main obstacle concerning these requirements seemed to be directly connected to the troubles of locating persons who might have initiated a given requirement. Again, identifying the person who demands a particular requirement is an accountability issue, as well as being a resource for clarifying and agreeing on what requirements might ‘really’ mean. Thus:

They [requirements] came from separate individual people who later on just disappeared. There is one special requirement there that has lived long despite big protests from both the contractors and me … it’s about a higher degree of simulation, meaning simulation knowledge in the training courses. But there’s already a huge amount of simulation … at every moment there’s a simulation …. you do a little design piece then you simulate on that, almost like doing a compiling test when coding … it’s the same with the high level specification language. When we started to sort out where this requirement came from, and who is claiming it? It turned out to be an individual person from Finland, it was a woman who did this off the top of her head at a meeting. (Training-subproject leader)
Despite protests from both contractors and the Training sub PM the Technical Orderer refused to remove this requirement. This, not out of misplaced awkwardness, but because the requirement could not be removed until the requirements initiator was traced down and consulted. One obvious feature of these difficulties is that troubles are functions as much of historical as of geographical distribution. They relate as much to the fact that tracing organisational members is difficult when time has passed as it is due to the fact that several organisations are involved.

6.2 ‘I Mean, I Can’t Understand Them …’ - Practical Solutions to Practical Problems

The comment about the ‘woman from Finland’ above indicates the way in which any requirement stands proxy for someone’s intentions and therefore how the someone in question must be tracked down, their opinion sought, and a decision made. The requirement in question was eventually divined to be a verbal ‘mistake’ that had somehow been captured in the requirements specification. The work of finding the originator for this requirement was nevertheless important, since it involves establishing that there are people-who-are-accepted-as-knowledgeable and who are thus able to justify the implementation or otherwise of the requirements. Tracking down requirements initiators were ranked as one of the most important issues to deal with. Important and problematic though this sometimes turned out to be, however, it was never a full solution, because in the light of the kinds of opacity we have rehearsed above, attention to possibly conflicting opinions is equally important. As the Training-subproject leader put it:

*This has been one of my big missions, to identify and to put an opponent on each requirement ... saying who can take the responsibility of being the requirement initiator and who might challenge it ... people I can have a dialogue with.*

Thus, the issue of identifying the origin of any given requirement is an issue of identifying individuals, identifying what they ‘really’ meant, and identifying who might hold to a different version. Of course, this
tracing of responsibility is not always possible, and if projects are to be completed, decisions perforce must be made. The ToolCore-subproject leader made one alternative strategy clear as follows:

Yes … sometimes when there’s a problem you can just go to other documents and you might get a more exhaustive version there. But sometimes there’s only a few lines of text anywhere … no matter where you look … this has caused a lot of major problems. In my sub-project, during the autumn in the feasibility phase, I solved this. I chose to solve this problem by simply saying, OK, if we don’t have a clear requirement, then we’re going to produce a solution anyway. The point is the customer could see how we understood the requirements within the solution we proposed, and if they didn’t like it … well, we get to understand what the customer’s requirements are …

Here, The ToolCore-subproject leader has adopted a pro-active strategy which involves deliberately taking a view of what requirements ‘really’ mean from one organisational position and proceeding on this basis with the express intention of prompting a reaction from other members of the project, in this case customer representatives. For this person, the strategy evidences the typical problems encountered insofar as customers don’t always understand his perspective and he doesn’t always understand theirs. To be clear, this is not a default strategy. It occurs specifically in a context where he has been unable to ‘track down’ the requirements originator or disambiguate the requirements specification by any other means. His strategy enforces one of two outcomes and we are perhaps entitled to judge that he has a preference for the first. Nevertheless, this approach has a direct corollary in an observation made by the main technical orderer:

On the other hand, people don’t always try to specify a requirement by asking what it means … they don’t always do it this way. Instead they try to interpret it by themselves …this without being absolutely sure … then the work of implementation takes place based on the interpretation. This can be troublesome, if this belief is strong and really thought to be the solution. But often they have actually not understood the problem, or perhaps two sides understand different things in one and same sentence.
That is, the strategy in question is high-risk because it might well way to deal with the problems we recount above, specifically those of lack of coherence and coming to terms with diverse suggestions. In any event our task here is not to judge the efficacy of this strategy but to report that it, along with others, exists.

Strikingly, one software engineer, Arne, working on the code compiler in the ToolCore-subproject had evolved a fairly sophisticated set of strategies for dealing with these problems. They involved using a network of informal contacts. When asked how he went about interpreting the requirements described in the Specifications, the very same requirements that he was supposed to implement in the compiler, answered:

*I don’t … I mean, I can’t understand them, not from the Requirements Specification document on its own. What I do is, I contact the technical person that is responsible for that particular requirement. Of course, the problem with this is that the person’s name is never mentioned in the Requirements Specification. I mean the person who originally figured it out.*

Arne continued,

*I have to do my own research, starting to contact people … asking around to establish where a given requirement might have its origin. That means finding a responsible technical person behind every requirement, finding the person who originally wrote it down, and finding other people as well who might have an interest in these specific requirements.*

In effect, what this individual had done was progressively establish a wholly informal network consisting of individuals located around the world, many of whom had prior experience of the code compiler. He explained:
I start asking around with the help of e-mail and telephone ... questions like; what does X mean, and why? Then negotiations start about the requirements in question. It is very seldom that I don't use my contacts, and we might have, say, six people involved in a discussion concerning the relevant requirement, people from all over the world. Of course, this gives rise to other problems because of the time difference. It can mean the process is slow ... it often takes days to get an answer. It shouldn't be this way ... but it is ...

One point is well worth making about this informal network, which is that it existed entirely independently of project management. Indeed, the project manager was unaware of its existence at that stage. It was also extensive and ongoing. It was common for Arne to receive communications from people asking him what solutions he had finally adopted following on from discussions that had been engaged in some time previously.

These problems, to re-iterate, had to do with searching for clarity and tracing responsibility across a widely dispersed and historically transformed organisational context. Their resolution is, and always must be, an organisational achievement. Documentation (nor any other tool) on its own is not adequate to this task since the documentation cannot specify its own ambiguity, nor is it likely to fully circumscribe all organisational contingencies. Nevertheless, documentation act as evidence of organisational accounting- that someone has taken responsibility; that progress is being made; clarity and precision achieved and that what is being done is becoming available to others. In order to answer the challenge of prescription rather than analysis the problem can be construed as that of identifying what kind of documentation or tool is likely to provide for and support this organisational accounting work satisfactorily.

7. Conclusion
We have glossed software engineering interests above as ‘prescriptive’, insofar as producing ‘better’ methods is a core feature of their work, and suggested that this emphasis, in contrast to Computer Supported Cooperative Work’s rather more analytic interests, may explain the
absence of a serious dialogue between the two communities. There is, however, an additional, and somewhat paradoxical, feature to be accounted for, which has to do with a common critique applied within software engineering to the whole business of doing ‘qualitative’ empirical studies of the business of software engineering (Seaman 1999). This, at root, rejects what is seen as a subjectivist, single-case approach on the grounds that it lacks objectivity and is generally unscientific. Put (too) simply, criticism of the empirical approach resides in the belief that it cannot provide validated answers to questions put. We return at this point to Computer Supported Cooperative Work’s analytic interests, for the critical divergence here is not, in our view, between science and subjectivism but between rationalistic conceptions of purpose, plan and intention and an empirically grounded, behavioural approach to the problems software engineers encounter and the solutions they manage. Put simply, the issue is not the problem of a valid answer, but that of a valid question. Thus and for instance we can ask questions about the necessity of the informal network of contacts referred to above, such as why it is that such a substantial network was in existence, and why is it that it was largely invisible to the official project management, and indeed why it is that these informal arrangements are not apparently visible to software engineers who develop tool support? The questions arise in the first place not only because some fieldwork has been done but also because a particular analytic stance is taken about its focus and to its attendant results. This stance has to do with the investigation of ordinary, practical and mundane issues encountered by project members as they go about their work— the problems they encounter and the strategies they adopt. Of course, it could be that strategies of this kind are in fact extremely rare in software engineering projects, but we suspect not. In any event, it seems well worth further investigation.

For the purposes of clarity, our task here is not to point to the fact that documents and other ‘planful’ artefacts necessarily underdetermine what is to be done, but to identify the specific failures that result in organisational accounting work. We are only too aware of the fact that, throughout the project we examine, documentation was necessary and vital. Indeed, we were specifically interested at the outset in how different documents served as planning and co-ordination tools. The
requirement specifications, which were clearly inadequate on some occasions for some purposes, nevertheless served as a co-ordination mechanism between the subprojects and the customers, as well as - together with the project plan - to co-ordinate co-operation within the project. The issue here is not whether the requirements determine its implementation, but whether they provide enough support for it. The documents in question, stand proxy for the work of negotiating varied assumptions about organisational purpose, structure and process. As we have seen this is seldom unproblematic and often entails elaboration of one kind or another. The need for subsequent work is not a result of the failure of the document, but of the failure of the assumptions that surround it. Knowledge and/or assumptions about the organisation and its customers embedded in the specification turns out not to be the only knowledge required. The organisational knowledge's that are required often, in turn, turn out on occasion to be opaque in origin, ambiguous, difficult to recover and will vary in their consequences. We have tried to show in our description of work done around the recovery of clear requirements that the consequences can be significant. Since the requirements needed to be understood in a way that adequately accounted for what various people ‘really’ wanted, the project members organised co-ordination across space and organisational boundaries. In this process the project members were, in effect, left to their own creative devices.

To further situate enquiries of this kind, Conklin (1993) refers to the way in which, ... an effective organisational memory would be able to answer such often asked questions as “Why did we do this?” and “How did such and such come to be the case?” Rarely is that possible now (p.561). He further argues that the failure of project and other documentation lies in its historical failure to capture context, rationale or process and suggests a move towards a new, ‘process oriented’ paradigm which might include for instance tracing such matters as meetings, action items, conversations, decisions etc. In essence, we can only agree. Nevertheless, and to re-emphasise the point about empirical studies, it is not enough for us to recognise in general that ‘product oriented’ documentation is inadequate. Empirical study should enable us to identify quite specifically what inadequacies are entailed, on what occasions and how consequential they are. Difficulties experienced, we
have pointed out, drive the planning or co-ordination work that has to be done. Specific problems, as we have tried to show, relate to lack of coherence, coming to terms with diverse suggestions, understanding the opaque practices and rationales of others, locating persons who might have initiated a given requirement, and generally dealing with problems of historical as well as of geographical distribution. These spatial and temporal facts engender the kinds of disambiguation work that has to be done here, and reflect quite ordinary knowledges and skills deployed by organisational members. Nevertheless, they need to be recognised before they can be supported. The kinds of analytic orientation that uncover, for instance, the work of ‘knowing who’ have already been attested to (see for example Randall et al, 1996; Harper et al, 2000). Indeed, in the last few years the empirical ‘turn’ in Computer Supported Cooperative Work has alighted upon problems of organisational knowledge and memory, and has prompted discussions about the nature and distribution of expertise in more than one context. In addition, some work has explicitly focused on the way in which some of these issues can be conceptualised as strategies for information searching (see McDonald and Ackermann, 1998; Groth and Bowers, 2001). Such work, taken together, indicates the importance of forms of organisational knowledge and skill (or on occasion, their absence) which have to do with ‘knowing your way around’. That knowledge and skill includes specific knowledge concerning who can be relied on to provide accurate or at least satisfying organisational accounts, finish tasks, furnish procedures etc. We believe the importance of these matters in the software engineering project we describe is abundantly clear. In our case study, absence of knowledge concerning who was responsible for formulating requirements, and their rationale for so doing engendered many of the difficulties spoken of by our respondents and resulted in sometimes elaborate strategies for resolving difficulties.

We have not attempted to discuss the details of software support for these issues here, but believe very strongly that the direction indicated by Conklin and others is centrally relevant to the software engineer’s problem. Some kind of support for engineering rationale, ranging from the very simplest of solutions, which entails providing names, organisational roles and contact details for individuals associated with requirements gathering in projects of this kind up to relatively
sophisticated engines of the kind sometimes seen as ‘organisational memory’ solutions, will become increasingly relevant as projects become more distributed. The general character of such solutions is such that they might allow software engineers to search for rationales, conversations, accounts of organisational practices, justifications and perspectives and so on.

8. Acknowledgments

Our thanks to the project members who through their openness and frankness made this study possible. Thanks also to anonymous reviewers for helpful comments.

References


