Thomas Østerlie

Problems and solutions: Maintaining an integrated system in a community of volunteers

Thesis for the degree of Philosophiae Doctor

Trondheim, October 2009

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics
and Electrical Engineering
Department of Computer and Information Science

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the degree of Philosophiae Doctor

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

ISBN 978-82-471-1808-5 (printed ver.)
ISBN 978-82-471-1809-2 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2009:204

Printed by NTNU-trykk

Abstract

Motivation. Software maintenance is a significant part of the software life-cycle cost. Current research focuses on the maintenance of application software. Despite increased focus on systems integration, there is limited research on maintaining integrated systems. Before progressing with informing software integration practice, researchers therefore need to better understand the actual work of maintaining integrated systems.

Research. To this end, a study of maintaining an integrated system in practice has been conducted. The study is conducted in the context of a community of volunteer software integrators. The research combines field studies with document analysis, asking:

> **RQ1**: How is knowledge of software failures developed during geographically distributed software maintenance?
>
> **RQ2**: How do software developers build knowledge of how to replace a business-critical software system?
>
> **RQ3**: What are the characteristics of large-scale software maintenance work in a geographically distributed community of volunteers?

Contributions. The main empirical contribution offered by this thesis is insight into the social and technical processes of maintaining an integrated system in a distributed community of volunteer software integrators. It offers a view of software maintenance where multiple stakeholders with different interests continuously negotiate over *problems and their solutions*. Focusing upon scarcity of resources and contradictory interests brings out the inherently political aspects of software maintenance.

> **C1**: Knowledge of software failures is developed through a process of negotiating over possible interpretations of available data, a process that is contingent upon situational issues such as workload, priorities, and responsibilities
>
> **C2**: A collective understanding of the scope, stakeholders, and sequence of activities for rewriting software evolves in response to new problems emerging from the rewrite efforts themselves as well as environmental changes
>
> **C3**: Maintaining an integrated system in a community of volunteers is characterized by scarcity of resources, an emphasis on coalition building, and volatility of stakeholders

Two contributions to software maintenance practice are offered:

> **C4**: Recommendations for a lenient approach to coping with variability during corrective maintenance
>
> **C5**: Recommendations for an opportunity-driven approach to systems replacement

Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) for partial fulfilment of the requirements for the degree of Philosophiae Doctor.

It is the concluding report from the research project titled 'Empirical Software Engineering and Open Source Software Development'. The project has been funded by Professor Reidar Conradi by the Department of Computer and Information Science, NTNU, Trondheim.

The reported research has been conducted under the supervision of Professors Letizia Jaccheri, Reidar Conradi, and Eric Monteiro.

Acknowledgements

I would like to extend my thanks to Letizia Jaccheri for encouraging the research, for being open to questions and discussions, and for giving me the freedom to approach the research my own way. I would like to express my deepest gratitude to Reidar Conradi for his role in funding the reported research as well as in his enduring patience with seeing the research project through. His role has not been sufficiently acknowledged in the papers reporting from this research. A special thanks to Eric Monteiro for guiding and advising me on my way through the maze of academic life. His influence has been formative to the way I conduct research.

I am indebted to the intellectual community, both locally at the department as well as the broader national and international community of colleagues, that I have had the pleasure of participating in throughout the period of this research.

I would like to offer deep gratitude to my co-authors Alf Inge Wang, Glenn Munkvold, and Kirsti Berntsen with whom I shared endless hours of discussions with. I would also like to offer thanks to the COSI project team at NTNU – Øyvind Hauge, Carl Fredrik Sørensen, and Sven Ziemer – for enjoyable and productive cooperation, as well as company during the many national and international project meetings. I am also grateful for the intellectual community shared with colleagues in the software engineering research group as well as Forskerfabrikken.

I am indebted to the industry partners in the COSI industrial R&D project for the formative role they have played in developing my understanding of OSS. I would like to extend a particular thanks to the Finish COSI research team – Juho Lindman and Matti Rossi at the Helsinki School of Economics, as well as Pentti Martiin at Nokia Siemens Networks – for letting me take part in their ongoing discussion on the nature of OSS in a commercial context.

I offer my gratitude to the invaluable feedback and support provided by the community of fellow researchers attending the PhD Days workshops, the national research seminar held by the Department of Informatics at the University of Oslo.

I would also like to thank the anonymous reviewers who rejected early versions of some of the papers included in this thesis. This was important feedback for us to change and improve these papers.

Thank you to Tor Grønbech for unwavering help and support during periods of hardship. Also thank you to Kjell Bratsbergengen for being supportive and understanding during these periods. I want to thank family and friends for their enduring support throughout the period of this research. I particularly want to acknowledge the price paid and the many sacrifices made by my wife Grete in seeing this research project through. It was a longer haul than any of us had expected.

Table of contents

1. Introduction

This thesis summarizes and concludes the research project titled 'Empirical Software Engineering and Open Source Software Development'. The project is undertaken as part of the Ph.D. programme attended by the Department of Computer and Information Science at the Norwegian University of Science and Technology. As the concluding report of the research project, the purpose of this thesis is to provide the broader context for the eight previously published papers reporting from the research project. In addition to summarizing the main contributions of the papers, the thesis also present an original empirical contribution based on the totality of the reported research. With basis in the empirical contributions, the thesis also offers a set of recommendations for software maintenance practice.

The purpose of this chapter, however, is to provide the motivation for the study and to briefly summarize the research reported in this thesis.

1.1. Research motivation

Research on maintenance effort over the past 30 years suggests that more than half the total life cycle cost of software is spent on software maintenance (Calzolari et al. 1998). Research also suggests that the maintenance burden is increasing. Pigoski (1997), for instance, shows that maintenance costs have risen from 40% of the total life cycle cost in the 1970s, through 55% in the 1980s, to 90% in the early 1990s. While the latter figure may be somewhat exaggerated, many researchers report that organizations now spend more time maintaining existing software than they do developing new ones (Swanson and Dans 2000). As software maintenance is often defined as modifications of software after its initial delivery (Basili 1993), increase in maintenance costs may also be attributed the increased longevity of contemporary software (Swanson and Beath 1989).

Research on software maintenance is growing. So far, though, software maintenance research has predominantly focused on maintenance of application software (Mockerjee 2005). Banker et al. (1993) define application software as a set of software modules performing a coherent set of tasks in support of a given organizational unit and

maintained by a single team. This definition can be expanded to include standardized software products. Several teams or even an entire software organization may also be required to maintain large-scale application software. Over the past decade, however, software integration has received increased attention. This can be attributed to three developments within the software industry. With increased availability of off-the-shelf products, component-based development has become a viable alternative to traditional programming (Boehm and Abts 1999). Furthermore, individual and collaborating organizations integrate previously separate and isolated systems to give them greater market leverage (Lam and Shankararaman 2004). Software integration is also proposed as a solution to avoid replacing or modifying the growing number of business-critical legacy systems (Hasselbring 2000).

In the future, the number of integrated systems will therefore increase at the expense of application software (Boehm 2006b). Yet, there is limited research on maintaining integrated systems. However, to do research that is relevant to systems integration practice, researchers have to better understand the actual work of maintaining integrated systems. Rather than focusing on improving the process of maintaining integrated systems, the reported research therefore explores how software integrators maintain integrated systems in practice. Consequently, it seeks to inform research rather than practice. As the relevance of software engineering research is largely driven by the desire to directly address the needs of practitioners (Osterweil 2007), the reported research can therefore be considered part of the ongoing discussion about research relevance within software engineering.

Software engineering practice relies heavily upon the knowledge of individual software developers and their interactions (Ye 2006). The research reported in this thesis therefore focuses upon *software maintenance as knowledge-intensive work*. This is called *software maintenance work*. The reported study draws upon research that sees work and knowledge as interrelated (Brown and Duguid 1991), emphasising the unexpected twists and turns as software integrators have to make sense of situations that are puzzling, troubling, and uncertain (Weick 1995).

1.2. Research setting

Software maintenance as knowledge-intensive work is explored in the context of open source software development (OSSD). OSSD is well suited for studying software maintenance work, as it is often understood as a perpetual cycle of corrective, adaptive, and perfective maintenance (Samoladas et al. 2004). To this end, an interpretive field study of Gentoo has been undertaken. Gentoo is a community of volunteer software integrators who maintain and operate a software system for distributing and integrating third-party open source software (OSS) with various Unix operating systems. Chapter 4 will explain OSS more in detail. For now, however, OSS is merely software released under a license that makes the source code open for anyone interested to read and modify. The volunteers studied call themselves the Gentoo developers, and they release the software they maintain as OSS. In addition, the community provides a GNU/Linux distribution, *Gentoo Linux*, based upon the software distribution system. A GNU/Linux distribution is a collection of software applications and libraries bundled together with

the Linux operating system kernel. In a sense, it is the OSS equivalent of the shrink-wrapped Microsoft Windows installation disc. As such, the Gentoo community can be understood as the OSS world's equivalent of a vendor of shrink-wrapped software.

As of March 2006, the Gentoo community consisted of 320 official volunteer software integrators distributed across 38 countries and 17 time zones. They call themselves the Gentoo developers. None of the Gentoo developers were, to the best of my knowledge, geographically co-located. As with most volunteer OSS communities, users are an important part of the Gentoo community, contributing with problem reports as well as source code. However, it is impossible to pinpoint the number of users active in the community at any one time. It is still safe to say that Gentoo is a large-scale maintenance effort.

1.3. Research goals and questions

With basis in the view of software maintenance as knowledge-intensive work, the overall goal of the reported research is:

> *To explore maintenance of an integrated system within the context it is developed and used. In particular, to explore the intertwined social and technical factors that influence software maintenance work in a community of volunteer software integrators.*

The following three research questions have therefore been asked:

> **RQ1**: How is knowledge of software failures developed during geographically distributed software maintenance?
>
> **RQ2**: How do software developers build knowledge of how to replace a business-critical software system?
>
> **RQ3**: What are the characteristics of maintaining an integrated system in a distributed community of volunteers?

1.4. Contributions

Included with this thesis are eight previously published papers reporting from the research project. Each of the papers has been published in peer-reviewed outlets. They therefore offer single research contributions from the research project. The papers are listed along with a brief summary of their individual contributions in Section 1.4.1. This thesis also provides five contributions with basis in the results reported in the papers. A brief overview of these contributions is presented in Section 1.4.2.

1.4.1 The papers

Out of 13 scientific and two popular-scientific papers published as part of the research reported here, eight peer-reviewed papers have been included with this thesis. This

section presents the individual papers (ordered chronologically) by providing: publication details, a short summary of the paper itself, before progressing with a brief outline of its individual research contribution, concluded with a description of my contribution to the finished product.

The sequence of the papers reflects the order they were written and published, reflecting the learning process I have gone through conducting the reported research. The scope of the research was initially broad, focused upon theory before gradually becoming more empirically intensive. Each paper has gone through preliminary versions, duly commented by colleagues at the department. Earlier versions of several of the papers included have also been presented and discussed in various seminars, workshops, and conferences. Where appropriate, the revision history of the papers is provided to give a better account of the learning process.

P1. Østerlie, T. "In the network: Distributed control in Gentoo/Linux", in *Proceedings of the 4th Workshop on Open Source Software Engineering, co-located with the 26th International Conference on Software Engineering (ICSE'04)*, Edinburgh, Scotland, May 25, 2004, pp. 76-81.

Summary. The paper reports on control issues during adaptive maintenance. Drawing upon the tension between distributed versus centralized control, the paper seeks to explore alternative ways of understanding control other than the power to make decisions.

The paper was selected as one of four accepted papers to be presented at the workshop.

Research contribution. The paper offers a contribution to OSS research by exploring the tension between distributed and centralized control in OSSD. Much research on control in OSS communities focuses on who has the power to make decisions, decision-making structures, and the configuration of these. In contrast, this paper empirically illustrates how control can also be understood as the power to frame the problems that needs to be made decisions about. As such, control is distributed in that it is a function of the reciprocal influence among people and technology. Control is therefore understood as not only inherent in organizational structures or hierarchies, but locally embedded among human and technological actors in the problem framing process.

My contribution. The paper is fully authored by myself.

P2. Berntsen, K., Munkvold, G., and Østerlie, T. "Community of practice versus practice of the community: Knowing in collaborative work," *The ICFAI Journal of Knowledge Management* (II:4), December 2004, pp 7-20.

Summary. This paper explores some theoretical implications for collaborative work when technology is given a prominent role. It proposes a shift of focus from the community aspect of collaboration towards the practice aspect.

Drawing upon work within science and technology studies, we illustrate the constitutive role technology plays in everyday work.

Research contribution. The paper offers a contribution towards research on knowledge-intensive work, in that it illustrates the material aspect of knowledge in collaborative work.

My contribution. While I wrote the initial draft, each author has contributed equally with text to the paper.

Revision history. An early version of this paper appears in *Proceedings of the 27th Information Systems Research Seminar in Scandinavia (IRIS'27)*, August 14-16 2004, Falkenber, Sweden.

P3. Jaccheri, L., and Østerlie, T. "Can We Teach Empirical Software Engineering?", in *Proceedings of the 11th IEEE International Symposium on Software Metrics (Metrics 2005)*, Como, Italy, September 19-22, 2005, pp. CD-ROM.

Summary. Based on the experiences from organizing and teaching a national PhD course in empirical software engineering, the paper seeks to evaluate two different approaches to teaching empirical software engineering – classroom and seminar-based teaching. The comparison is based upon the responses to a questionnaire circulated among students attending two iterations of the course.

Research contribution. The paper contributes to software engineering education by offering the description of a PhD level course in empirical software engineering: a well-defined syllabus, as well as an evaluation of two pedagogical strategies for teaching the syllabus.

My contribution. I participated in formulating the questionnaire, analysing the responses, and in writing the related work section of the paper.

Revision history. This paper was first published as AP10 (see page 16). P10 is an abridge variant of an early revision of P3. This early revision was rejected for the *10th IEEE International Metrics Symposium (Metrics'04)*. P3 is based upon the feedback received for AP10 and the review comments received from *Metrics'04*.

P4. Østerlie, T., and Wang, A.I. "Establishing Maintainability in Systems Integration: Ambiguity, Negotiation, and Infrastructure", in *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, Philladelphia, PA, September 24-27, 2006, pp. 186-196.

Summary. This paper reports from the analysis of corrective maintenance work in the Gentoo community (Section 8.2). The paper revisits the concept maintainability in the context of software integration. The paper explores how maintainability can be understood as a function of the external environment within which the software is being maintained. Maintainability is therefore the

collective achievement of software integrators, users, failing software, and an infrastructure of diagnosis tools.

Research contribution. The paper offers a contribution to software maintenance research by empirically illustrating how the maintainability of an integrated system is continuously enacted during corrective maintenance. This supplements existing research which views maintainability as a quality attribute or an architectural strategy. In contrast, the paper presents a view of maintainability, understood as the ease with which software can be understood and modified, where corrective maintenance is understood as a process of framing the problem resulting in the reported failure.

My contribution. Paper written by me, except for the related work section that was written together with the second author who also created the figures.

Revision history. This paper is a derivative of AP12. AP12 was initially submitted as a full-length paper to the *Second International Conference on Open Source Systems (OSS'06)*, but accepted as short paper. However, the review comments received were formative for the direction of the revision to P4. An early revision of AP12 was also presented and commented during the *10th PhD Days*, a PhD seminar held by the Department of Informatics, University of Oslo, February 9-10, 2006.

P5. Jaccheri, L., and Østerlie, T. "Open Source Software: A Source of Possibilities for Software Engineering Education and Empirical Software Engineering", in *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development, co-located with the 29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, Minnesota, May 20-26, 2007, pp. 1-5.

Summary. Paper reports from and reflects upon the work on teaching master level students by using principles from action research for organizing OSS education.

Research contribution. The paper contributes towards the software engineering education with an approach to learning practice-based software engineering through action research in OSS communities.

My contribution. Paper predominantly written by first author. I wrote the related work section as well as supplemented the analysis.

P6. Østerlie, T., and Jaccheri, L. "A Critical Review of Software Engineering Research on Open Source Software Development", in *Proceedings of the The Second AIS SIGSAND European Symposium on Systems Analysis and Design*, Gdansk, Poland, June 5, 2007, pp. 12-20.

Summary. The paper is based on a discourse analysis of the software engineering research literature on OSSD. It seeks to explore why software

engineering research on OSSD keeps on portraying OSSD as a homogenous phenomenon despite the fact that recent empirical studies show great variation of software development activities among OSS communities. Four ways the literature present OSSD as a homogenous phenomenon is identified. The paper finds that the software engineering research literature's view of OSSD is based in three assumptions collectively held by the discipline: assumptions about software engineering research, assumptions about how to do software engineering research, and assumptions about the object of study.

Research contribution. The paper offers three contributions to software engineering research. First, it shows that assumptions about software engineering research may have produced a systemic bias in the research on OSSD. Second, it offers a set of suggestions for improving the situation. Third, the paper contributes with a possible approach for evaluating the effect research approaches and assumptions have on the object of study.

My contribution. The paper is the result of several years of discussion about software engineering research on OSSD with the second author. It is still written in its entirety by myself.

Revision history. A first revision of the paper was submitted to the *Third International Conference on Open Source Systems (OSS'07)*, but rejected. While strongly disagreeing with the review comments, the paper still underwent major revision to pre-empt the concerns raised by the *OSS'07* reviewers.

P7. Østerlie, T., and Jaccheri, L. "Balancing Technological and Community Interest: The Case of Changing a Large Open Source Software System", in *Proceedings of the 30th Information Systems Research Seminar in Scandinavia (IRIS 30)*, Tampere, Finland, August 11-14, 2007, pp. 66-80.

Summary. This paper reports from an analysis of the process of rewriting and replacing a core component of the Gentoo software (Section 8.1). The paper seeks to explore how the interaction between the software and its context of development and use enable and constrain the rewriting process. It shows how adaptive maintenance as a continuous process of negotiating over the scope of the changes to be made, their sequence, and which actors to be involved in the process.

Research contribution. The paper offers a contribution to software engineering research on systems replacement. Much research on rewriting and replacement focuses upon replacement strategies and planning of the rewriting effort. In contrast, this paper empirically illustrates how the plan for rewriting and replacing software is continuously unfolding. The paper offers a view of systems replacement as a process of framing the problems that the rewritten software is to resolve.

My contribution. I wrote the text. The second author contributed as discussion partner and with concrete suggestions for improving the paper.

P8. Østerlie, T., and Wang, A.I. "Debugging Integrated Systems: An Ethnographic Study of Debugging Practice", in *Proceedings of the 23rd International Conference on Software Maintenance (ICSM'07)*, Paris, France, October 2-5, 2007, pp. 305-315.

Summary. This paper reports from the analysis of corrective maintenance work in the Gentoo community (Section 8.2). It explores how software integrators debug an integrated system. We identify five characteristics of the debugging process: that it spans a variety of operating environments, it is collective, social, heterogeneous, and ongoing. The debugging process is a collective sensemaking process, influenced by both social and technical factors, rather than a purely individual, cognitive problem-solving activity.

Research contribution. The paper offers a contribution to software maintenance research by identifying the five characteristics that sets the practice of debugging integrated systems apart from existing research on debugging. This suggests that the software failure is not unproblematic as a phenomenon, but rather subject to interpretation and negotiation. This raises concerns about the appropriateness of assuming that software failures are clearly identifiable and stable phenomena. That there is a clearly identifiable relation between the errors in the code and the observed failures is too simple. In system integration the problem is more complex.

My contribution. Paper predominantly written by myself. Second author contributed the analysis and to the related work section.

Revision history. An early revision of the paper was presented and commented during the *11th PhD Days*, 21-22 September, 2006. The same revision was submitted to the *29th International Conference on Software Engineering (ICSE'07)*, but rejected. The review comments helped pinpoint significant problems with this early revision, and were formative for revising the paper.

Additional papers (AP) published as part of the research project, but not included in this thesis.

AP9. Østerlie, T., and Rolland, K.H.R. "Unveiling distributed organizing in open source software development: The practices of using, aligning, and wedging", in *Proceedings of the Workshop on Open Source Software Movement and Communities, co-located with the First International Conference on Communities and Technologies (C&T'03)*, Amsterdam, Netherlands, September 18, 2003, pp. 1-7.

AP10. Jaccheri, L., and Østerlie, T. "Empirical Software Engineering Education", in *Proceedings of the 11th Norwegian Conference on Information Systems (NokobIT'04)*, Stavanger, Norway, November 29-December 1, 2004, pp. 242-249.

AP11. Østerlie, T., and Munkvold, G. "Ordering actors, organizing work", in *Proceedings of the 28th Information Systems Research Seminar in Scandinavia (IRIS)*, Kristiansand, Norway, August 6-9, 2005.

AP12. Østerlie, T. "Producing and Interpreting Debug Texts", in *Proceedings of the Second International Conference on Open Source Systems (OSS'06)*, Como, Italy, June 8-10, 2006, pp. 335-336.

In addition, the following popular scientific papers have been published in connection with the research reported in this thesis.

AP13. Oksholen, T. "Frå vondt til verre" (eng. 'From bad to worse'), *Gemini*, issue 5, October 2005, pp. 28-29.

This article is an interview with me. With basis in the reported research and existing research on software integration, I reflect upon the implications of software integration to organizations. Gemini is a national popular-scientific magazine.

AP14. Søndenaa, T. "Forsker på open source-utviklere" (eng. 'Studies of open source developers'), *Linux magasinet*, issue 3, June 2007, p.30.

This article is an interview with me. The interview reports on the research reported in this thesis to the national community of Linux enthusiasts and practitioners. Linux magasinet was a national trade magazine targeted at the national Linux and open source community.

1.4.2 Contributions of this thesis

Summarizing the individual papers reporting from the research project, this thesis offers five contributions. Three of these contributions are empirical, and two offer recommendations for software maintenance practice. The contributions are presented in their entirety in Chapters 9 and 10, respectively. An overview of the five contributions is provided in Figure 1-1. The figure illustrates the relationship between the individual papers reporting from this research and the five contributions offered in this thesis.

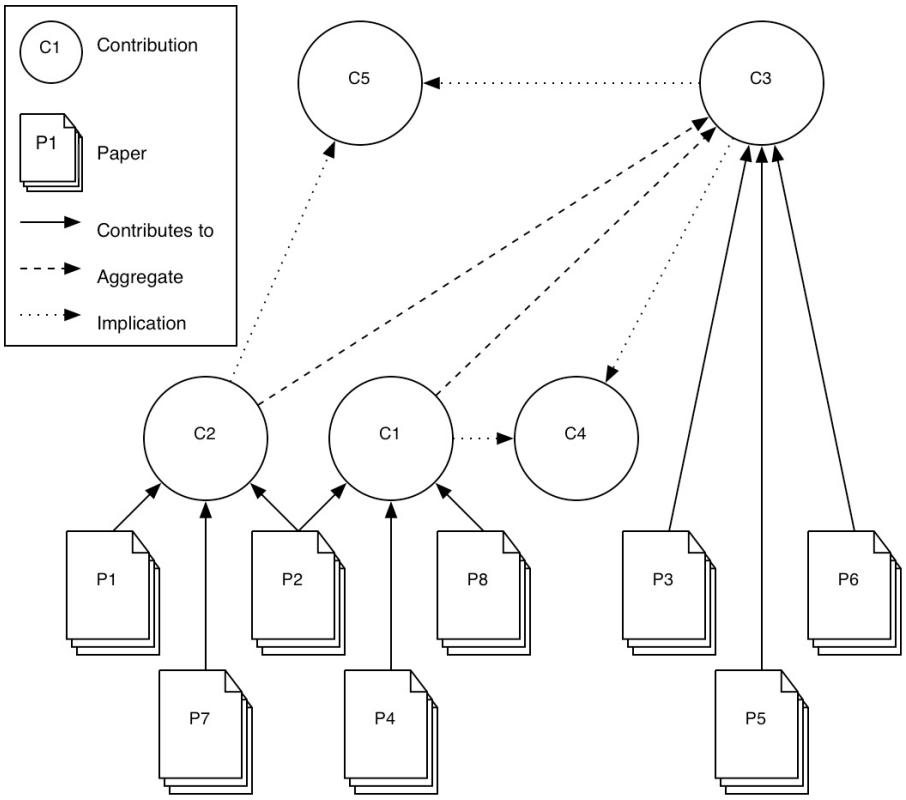


Figure 1-1 Relationship between papers and contributions

The main empirical contribution offered by this thesis is insight into the social and technical processes of maintaining an integrated system in a distributed community of volunteer software integrators. In particular, the thesis offers a view of software maintenance where multiple stakeholders with different interests continuously negotiate over *problems and their solutions*. Focusing upon scarcity of resources and contradictory interests brings out the inherently political aspects of software maintenance. Whereas more or less clearly defined problems is the basic premise of application software maintenance research, the reported research shows that the essential activity of maintaining integrated systems is problem setting: the collective process in which situations that are unclear, problematic, and puzzling are progressively clarified.

Specifically, three empirical contributions are offered. The first two contributions draw together results reported in the papers included with this thesis:

> **C1**: Knowledge of software failures is developed through a process of negotiating over possible interpretations of available data, a process that is

contingent upon situational issues such as workload, priorities, and responsibilities (in response to RQ1, documented by P2, P4, and P8).

C2: A collective understanding of the scope, stakeholders, and sequence of activities for rewriting software evolves in response to new problems emerging from the rewrite efforts themselves as well as environmental changes (in response to RQ2, documented by P1, P2, and P7).

Contribution C3 aggregates the results reported in C1 and C2 to form an original contribution from the totality of the reported research:

C3: Maintaining an integrated system in a community of volunteers is characterized by a scarcity of resources, an emphasis on coalition building, and volatility of stakeholders (in response to RQ3, documented by P3, P5, and P6).

Grounded in the empirical contributions, contributions C4 and C5 draw practical implications for software maintenance practice:

C4: Recommendations for corrective maintenance practice.

C5: Recommendations for systems replacement practice.

1.5. Thesis structure

The thesis is divided into three parts. Part I presents related work. The purpose of this part is to position the reported research within software engineering. This part introduces the topic of research relevance within software engineering. It also presents research on software maintenance and OSSD. Part II presents the reported research. Here, the interpretive research approach is presented first. Gentoo, the research setting, is then presented, before progressing with an overview of the research process and reflections upon the research. Part III presents the results of the reported research. This part consists of three chapters. The first chapter presents the empirical results of the research. Implications of the empirical results are then drawn for software integration practice. The issue of research relevance is then revisited in light of the reported research. Chapter 11 draws conclusions of the research and proposes future work. Chapter 12 offers a glossary with the key terminology used.

PART I: RELATED WORK

2. Relevance in software engineering research

The software engineering discipline can be understood as a movement of industry and academic actors to professionalize software development. Research-informed practice has therefore been a key goal of the discipline. However, since the mid 1980s there has been a recurring discussion over the relevance of software engineering research (Basili et al. 1986; Fenton 1993; Potts 1993; Glass 1994). The discussion can be related back to the goal of research-informed practice, and that the relevance of software engineering research has largely been driven by the desire to meet the needs of practice (Osterweil 2007). To make research more relevant to practice, the empirical agenda was proposed to increase the validity of research results through increased scientific rigour (Basili 1993).

This thesis asserts that in order to inform software engineering practice, researchers need to better understand what practitioners do when developing and maintaining software. While increased scientific rigour may increase the validity of the research results, it is argued that the problem is also that research results fail to address issues relevant to practitioners. Empirical studies, while scientifically rigorous, tend to focus on simplified small-scale problems that fail to grasp the complexities of software engineering practice. Software engineering researchers often know too little about these complexities to effectively inform research practice. Software engineering research therefore needs to be informed by practice before researchers can inform practice. As such, increased scientific rigour may actually contribute to further exacerbate research's lack of relevance to practice.

With basis in the above proposition, this thesis and the research reported here can be considered part of the ongoing discussion on relevance and software engineering research. The purpose of this chapter is therefore to provide the background for the empirical agenda in software engineering as response to the problem of relevance. However, it is proposed that the turn towards science within software engineering research needs to be situated in the broader societal context of professionalizing work during the 20th century. It is within this context that the empirical agenda in software

engineering can be understood not only as a direct response to problems of research's relevance to practice, but also as a standard solution of the broader movement of professionalizing work that the software engineering discipline is part of. Drawing upon a standard solution, however, researchers only address a part of the problem of relevance within software engineering.

However, before progressing further with elaborating this argument, we need a working understanding of software engineering first.

2.1. Software engineering

While several definitions of software engineering exist, the purpose of this section is not to synthesise a definite definition. Software engineering is an evolving discipline (Finkelstein and Kramer 2000), and definitions are inherently problematic when trying to grasp evolving phenomena. This section therefore seeks to establish a working understanding of software engineering rather than to formally define it. It does so by drawing upon previous works aimed at identifying the discipline.

In their review of the computing literature, Glass et al. (2004) distinguish between three broad subfields within the computing disciplines: computer science, software engineering, and information systems research. This review implies that rather than being clearly delineated, there are sliding boundaries between the three subfields. Computer science is at one end of the scale. This subfield is predominantly concerned with computer concepts at technical levels of analysis. Information systems research resides at the other end of the scale. Information systems research examines topics largely related to organizational issues. However, systems and software-specific topics are also studied. Computer science researchers expect to contribute with new processes, methods, algorithms, and products. Information systems researchers, on the other hand, expect to explore theories, concepts, techniques, and projects.

Software engineering resides between the two other subfields. Like information systems research, software engineering is concerned with systems and software-specific topics. However, like computer sciences, it does so predominantly at the technical level of analysis. While software engineering researchers to a certain extent expect to contribute with new processes, methods, algorithms, and procedures (Glass et al. 2002), there is also some focus on theory contributions (Sjøberg et al. 2008).

Finkelstein and Kramer (2000) draw upon software engineering's focus on systems and software-specific topics in locating the discipline within a broader disciplinary landscape. They propose that software engineering can be considered a subfield of systems engineering. Systems engineering is concerned with hardware development, policy and process design, as well as software engineering (Sommerville 2001). Like systems engineering, software engineering is concerned with the specification, development, implementation, and maintenance of systems.

Drawing upon the engineering aspect, Basili (1993) distinguishes software engineering from manufacturing. The purpose of engineering research is to observe existing

solutions, propose better solutions, implement these solutions, and evaluate them. Unlike hardware, however, software is often considered more complex to build and understand. As such, there is an increased need for control through tools and process models. Software engineering can therefore be understood as the disciplined development and evolution of software systems based upon a set of principles, technologies, and processes (ibid.). In this view, the purpose of software engineering research is therefore to provide and improve tools, techniques, and methods for practitioners to improve parts of the software process. In short: research-informed practice. The issue of research-informed practice will now be pursued in the context of professionalization of work.

2.2. The empirical agenda in software engineering

This section situates software engineering as part of the broader movement towards professionalization of modern work during the 20th century. As part of this movement, it is argued that the turn towards increased scientific rigour is a natural response to the problem of research relevance. To this end, the section is organized as follows. First, software engineering is situated as part of the movement towards professionalization of work. Then, the problems of research relevance and the research-practice crisis of the 1990s is presented. The chapter is concluded with presenting empirical software engineering as the discipline's response to the crisis.

2.2.1 Professionalization of software development

Software engineering researchers often trace the origins of the discipline back to the *software crisis* (Boehm 2006a). Increased hardware capacity during 1960s made larger and more complex software systems a possibility. The software crisis is therefore commonly attributed to the combination of increased largeness and complexity of software systems and the relative inexperience of software developers which led to late deliveries of systems, escalating costs and failed software projects (Friedman and Cornford 1989, p. 99).

Looking towards the production industry and aiming to build upon its success since late 19th century, software engineering was proposed as the solution to the software crisis:

> *The whole trouble comes from the fact that there is so much tinkering with software. It is not made in a clean fabrication process, which it should be. What we need is software engineering. (Ludewig 1996, p. 25)*

As such, software engineering came to be defined as "*the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software*" (IEEE 1990).

However, the dichotomy between 'so much tinkering' and 'clean fabrication process' holds a clue to an alternative explanation of the software engineering discipline's origins. This explanation contextualizes software engineering in the movement towards

professionalization modern work during the 20th century. Professionalization of work is often traced back to the tension between the practice-based education of traditional trades' and theory-based academic training (Noble 1977). Professionalization has been characterized as the dual process of institutionalization on the one hand and development of professional knowledge on the other (Schön 1991).

Professionalization has been contrasted with the experience-based knowledge of the traditional trades where customary activities are modified by trial and error. Professions, on the other hand, are identified by the application of general scientific principles, and standardized knowledge to concrete problems (Schön 1991). The development of a standardized professional body of knowledge is therefore an important part of professionalization. Schein (1972) provides a three-component model of professional knowledge (summarized in Table 2-1).

Component	Description
1. Underlying theory of discipline	Component provides the general principles upon which the body of knowledge rests.
2. Applied science / engineering	Resting upon the general principles from the underlying discipline, the component provides the applied knowledge from which the day-to-day diagnostic procedures and problem-solutions are derived
3. Practical skills and attitudes	Using the underlying applied science, the component concerns the performance of services to clients.

Table 2-1 Schein's three-component model of professional knowledge

In this context, the software engineering discipline can therefore be understood as a movement of industry and academic actors to professionalize software development, maintenance, and operations. Such a view is further corroborated by recent years' discussion on further institutionalizing the software engineering profession by licensing the title (Knight and Leveson 2002).

With the above view of the software engineering discipline, the distribution of responsibilities between research and practice is such that researchers are to "establish a scientific and engineering basis for software engineering" (Basili 1993, p. 7). Software engineering practitioners, on the other hand, are to apply this knowledge in the development, operation, and maintenance of software. As such, *research-informed software engineering practice* has been a key goal of the discipline, and the relevance of software engineering research is largely driven by the desire to meet the needs of practice (Osterweil 2007).

2.2.2 The research-practice crisis

Since the mid 1908s, studies examining the state of software engineering research have raised concerns over its lack of impact on practice. With the goal of research-informed software engineering practice in mind, this lack of impact on practice is of great concern among software engineering researchers. Glass (1994) calls this the research-practice crisis.

Reviewing the software engineering research literature to examine the validity of the claims that methods, tools, and techniques improve quality and productivity in software development, Fenton (1993) finds "very little empirical evidence to support the hypothesis that technological fixes, such as the introduction of specific methods, tools, and techniques, can radically improve the way we develop software systems". This is particularly troubling when the predominant contribution of software engineering research are such methods, tools, and techniques (Glass et al. 2002). Commenting on research's lack of impact on software engineering practice, Tichy et al. (1993) conclude that software engineering research is lacking in quality and thereby becoming less credible to practitioners. Glass (1994) traces the origins of the crisis to the different views of software development held by researchers and practitioners; research results simply fail to address issues relevant to practitioners.

Tichy et al. (1995) surveyed 400 research papers within the broader field of computer science. Based on a random sample, they find that only 20% of the software engineering papers devote more than one fifth or more of the space to validation. Papers with no research validation are typically studies where the researcher implements a technology and shows that the technology works. Glass (1994) calls this advocacy research – researchers advocating a new technology without validating its effectiveness over existing technologies or its applicability to practitioners.

Similarly, Zelkowitz & Wallace (1998) reviewed 612 software engineering research papers. The papers have been published in three leading software engineering journals and magazines at three intervals during a ten-year period from 1985 to 1995. The survey shows that in 58.7% of the papers there is no validation of the research claims or the validation is based on assertions. This figure rises to 66.8% if counting papers where validation was not applicable.

As such, two reasons for the research-practice crisis were identified:

- Lack of credibility of research
- A gap between research interests and software engineering practice

2.2.3 Empirical software engineering

A call for increased empirical research and scientific rigour rose within the software engineering research community in response to the research-practice crisis. To increase the credibility of research claims software engineering research needed to better validate its scientific claims (Zelkowitz and Wallace 1998), preferably through increased experimentation (Tichy 1998). The low ratio of validated research had to be rectified for the long-term health of the discipline (Tichy et al. 1995). Similarly, Fenton (1994, p. 199) addressed existing research's lack of understanding of measurement theory, arguing that software engineering researchers "must adhere to the science of measurement if it is to gain widespread acceptance and validity". Summarized, researchers agreed that increased scientific rigour was needed to address the lack of credibility of research results.

The subfield within software engineering that emerged from this discussion came to be labelled empirical software engineering (Basili and Harrison 1996). Its focus is the systematic evaluation of software related artefacts for the purpose of characterization, understanding, evaluation, prediction, control, management, or improvement through qualitative or quantitative analysis through the application of the scientific method (Basili and Harrison 1996; Wohlin et al. 2000; Conradi and Wang 2003). The subfield has materialized in two annual conferences (*The IEEE Symposium on Software Metrics* since 1993, and *The International Symposium on Empirical Software Engineering* since 2002; being merged to *The International Symposium on Empirical Software Engineering and Measurement* in 2007), as well as the *Empirical Software Engineering* journal since 1996 (Basili and Harrison 1996). The subfield has been further supported by a number of textbooks on the topic (e.g. Fenton and Pfleeger 1997; Shull et al. 2007).

Surveying 369 software engineering research papers in the period 1995-1998, Glass et al. (2002) finds that less than 10% of the papers report from empirical studies. However, the trend is towards more empirical studies within software engineering. Concerns about the state of scientific rigour in empirical software engineering research have recently been raised, though. Dybå et al. (2006) reviews 103 papers reporting on controlled experiments published from 1993-2002. They find the statistical power in reported software engineering experiments to fall substantially below accepted norms. Despite these concerns, recent years' evaluative reviews continued focus on research validation shows that the dominant view of empirical software engineering research is a field based on measuring the software process and its products (Segal et al. 2005).

This view is strengthened by recent year's increased attention on evidence-based software engineering (EBSE) (Kitchenham et al. 2004; Dybå et al. 2005). Inspired by the results of evidence-based medicine, EBSE is regarded as a method for systematizing existing knowledge. Through a joint undertaking of systematic literature reviews, the goal of EBSE is "to provide the means by which current best evidence from research can be integrated with practical experience and human values in the decision making process regarding the development and maintenance of software" (Kitchenham et al. 2004, p. 274). A much-used set of guidelines for such systematic reviews, however, express the need for scientific rigour as a key quality in filtering what can be considered proper evidence (Kitchenham et al. 2004). As such, EBSE enforces the dominant view of software engineering as a predominantly quantitative research field.

2.3. Rigour or relevance

We have now established software engineering as a movement of academic and professional actors to professionalize software development. This, we have seen, is part of the broader movement of professionalization of modern work in the latter half of the 20th century. Other examples of professionalization can be found throughout modern working life, for instance in other fields of engineering, law, as well as in medicine and nursing. In his study of professional work, Schön (1991, p. 22) observes that "a profession involves the application of general principles to specific problems". Such general principles are based on a systematic, scientific knowledge formalized in theories

and models. Professional work, the application of such models and theories to particular problems, is therefore a form of applied science.

Yet, the application of such formalized knowledge requires unambiguous problems. This, Schön (ibid., p.42) argues, is the premise of the dilemma of rigour or relevance:

> *In the varied topography of professional practice, there is a high, hard ground where practitioners can make effective use of research-based theory and technique, and there is a swampy lowland where situations are confusing "messes" incapable of technical solution.*

Using the field of formal modelling as a formative example, Schön observes that the rigorous application of scientific knowledge is usefully employed to solve problems in undemanding areas, while failing to yield any results in more demanding and complex areas of the swampy lowlands. The problem, however, is that the high, hard ground is often of limited relevance to everyday practice. The messes of the swampy lowlands, on the other hand, are. The systematic development of a rigorous scientific knowledge base to turn a vocation into a profession can therefore be of limited relevance to practitioners of the profession. As such, a widening gap between research and practice may develop over time.

A similar division of labour between researchers developing formalized knowledge to be applied by practitioners may be observed within software engineering (Subsection 2.2.1 above). While the empirical agenda may have increased the validity of research results, there are few indications that this has improved research's impact on software engineering practice. For instance, Glass' (2007) appeal to software engineering practitioners to keep abreast with the findings published by experimental software engineering researchers suggests that the gap between software engineering research and practice remains.

The empirical agenda addresses the issue of credibility through rigour. However, as shown in Subsection 2.2.2 above, lack of credibility is only one of two causes of the research-practice crisis. The other cause is the gap between research interests and software engineering practice. Yet, considering software engineering in the context of professionalizing work, increased scientific rigour appears as a stock response to the problem of relevance. As argued above, increased scientific rigour is not synonymous with 'relevant to practice'. Furthermore, increased scientific rigour also tends towards small-scale problems that fail to address the complexities of software engineering practice.

This observation needs to be tempered by recent research showing that it may take between 15 and 20 years from the initial publication of an idea until it is widely used in products (Osterweil et al. 2008). As such, it may be too early to evaluate the effect of rigorous research. Still, with basis in Schön's (1991) work, it is reasonable to assume that more rigorous research may also be of less relevance to practice. Similar concerns have already been raised within the software engineering discipline. In surveying the computing literature Glass et al. (2002) finds that software engineering research is limited in its scope to software technical matters related to building software, improving

the way software is built, and analyzing or implement promising new concepts. However, they ask, maybe it is time for software engineering research to broaden its scope and to seek methods that may yield richer findings?

This question is left hanging for now. It will be picked up again in 9.4.5, which relates the issue of research relevance to the reported research.

3. Software maintenance, legacy systems, and integration

Software maintenance constitutes a significant part of the software life-cycle cost. Calzorella et al. (1998) report that estimates range from 50 to 80 percent of the total life-cycle costs are spent on maintenance. Research suggests that the maintenance burden is increasing. Pigoski (1997), for instance, shows that maintenance costs have risen from 40% of the total life cycle cost in the 1970s, through 55% in the 1980s, to 90% in the early 1990s. While the latter figure may be somewhat exaggerated, many researchers report that organizations now spend more time maintaining existing software than they do developing new ones (Swanson and Dans 2000). As software maintenance is often defined as modifications of software after its initial delivery (Basili 1993), increase in maintenance costs may also be attributed the increased longevity of contemporary software (Swanson and Beath 1989). Research is therefore mainly concerned with identifying factors driving maintenance costs, as well as developing methods for managing and reducing these costs.

So far, however, software maintenance research has mainly focused upon application software maintenance (Mockerjee 2005). Over the past decade, however, software integration has received increased attention. This can be attributed to three developments within the software industry. With increased availability of off-the-shelf products, component-based development has become a viable alternative to traditional programming (Boehm and Abts 1999). Furthermore, individual and collaborating organizations integrate previously separate and isolated systems to give them greater market leverage (Lam and Shankararaman 2004). Software integration is also proposed as a solution to avoid replacing or modifying the growing number of business-critical legacy systems (Hasselbring 2000).

To this end, this chapter is organized as follows. Section 3.1 discusses software maintenance in general, with a particular emphasis on corrective maintenance and debugging as these are central topics for the reported research. Section 3.2 discusses the product of long-term software maintenance, legacy systems. How legacy systems increase the maintenance burden and different strategies for coping with them are

discussed. Section 3.3 concludes the chapter with a discussion of maintaining integrated systems.

3.1. Software maintenance

Software maintenance refers to the activities of modifying software after its initial delivery and implementation. Software maintenance therefore focuses upon the correction of defects and the modification of the software to perform new tasks or perform old tasks under new conditions (Dvorak 1994). This section provides an overview of software maintenance, the processes and activities of the processes. Particular emphasis is paid corrective maintenance and debugging, as this is important for the research reported in this thesis.

To this end, the section is organized as follows. First, the scope of software maintenance is outlined (3.1.1). Organizational level maintenance process (3.1.2) and the individual process of implementing changes (3.1.3) are then presented. The section is concluded with a more in-depth presentation of corrective maintenance (3.1.4) and debugging (3.1.5).

3.1.1 Categories of software maintenance activities

Initially conceived as the correction of errors (Canning 1972), the scope of software maintenance has come to include corrections as well as enhancements. Swanson (1976) offers a typology of software maintenance activities. This typology is based on the cause for or purpose of the maintenance to be done. The typology consists of three categories:

- *Perfective maintenance*: Performed to perfect the software in terms of its performance, processing efficiency or maintainability
- *Adaptive maintenance*: Performed to adapt the software to changes in its data environment or processing environment
- *Corrective maintenance*: Performed to correct processing, performance, or implementation failures in the software

Highly influential within software maintenance research, and has been adopted by many researchers (Chapin et al. 2001). Kitchenham et al. (1999) proposes a fourth maintenance category, *preventive maintenance*. This expands the scope of the maintenance activities to include modification of both the software and its requirements, as both perfective and adaptive maintenance requires modification of system requirements. Adaptive maintenance entails new requirements to be added, while perfective maintenance only entails the modification of existing requirements. Preventive maintenance, on the other hand, only requires modification of the software.

The maintenance categories have been used to develop profiles of the maintenance effort. These profiles have been developed to identify factors driving maintenance costs. In a much cited study, Lientz et al. (1978) finds that 17.4% of the maintenance effort was spent on corrective maintenance, 18.2% on adaptive, while 60.3% as perfective.

The remaining 4.1% was categorized as others. Yet, in a more recent study, Scach et al. (2003) finds that the distribution of corrective maintenance is more than three times that of Lientz et al.'s study. In comparison, Scach et al. find that 4.4% of the effort is spent on adaptive maintenance, while 36.4% is spent on perfective maintenance. The figures therefore indicate that corrective maintenance drives maintenance costs. Reducing the effort of corrective maintenance activities may therefore have a significant impact on overall the maintenance costs.

3.1.2 The organizational level maintenance process

While the software maintenance activities are the same, the maintenance process may differ between organizations (Swanson and Beath 1990). Kitchenham et al. (1999) differentiates between the organizational level process of administrating change requests, and the individual maintenance engineers' process of implementing specific change requests. These two processes will be presented in turn.

An organization with co-located software maintenance team or department is likely to have direct interaction between maintenance engineers and users or user representatives. Companies developing off-the-shelf software, on the other hand, often interact with users through a customer support department (Pentland 1992). Still, in all organizations software maintenance focuses upon the correction of defects and the modification of the software to perform new tasks or perform old tasks under new conditions (Dvorak 1994). It is therefore common to refer to an idealized model of the maintenance process (Figure 3-1 below).

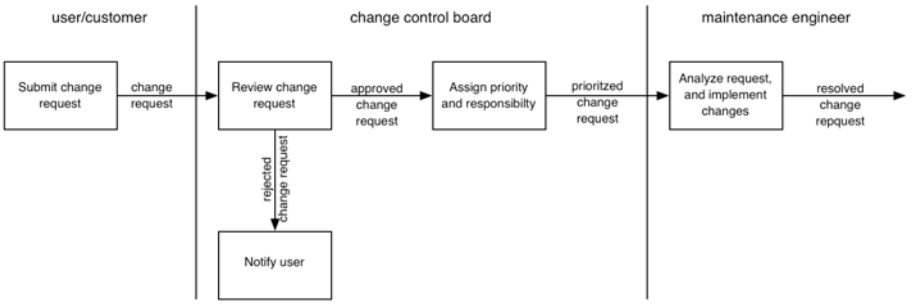


Figure 3-1 Model of the maintenance process

The change request is the point of departure for the maintenance process. In organizations with co-located maintenance team, users submit change requests. Users may range from end-user to user representatives (Swanson and Beath 1990). Commercial software companies receive change requests from customers. Change requests include both requests for adaptive, perfective, as well as corrective changes. Requests for corrections are sometimes called defect or *problem reports*. Preventive maintenance requests usually originate within the maintenance organization itself.

While older literature reports on paper-based databases for administrating change requests (Basili and Perricone 1984), it is today common to administrate change

requests with issue tracking software (Serrano and Ciordia 2005). It is often recommended that a change control board (CCB) is responsible for managing change requests (van Vliet 2000). The CCB prioritizes incoming change requests as well as assigning change requests to maintenance engineers. The administration of change requests is closely tied with strategic decisions on release planning and long-term development trajectory of the software (Ruhe and Saliu 2005).

Individual maintenance engineers, or teams of engineers, are then set to the task of implementing the requested changes.

3.1.3 Individual process of implementing changes

All forms of categories of software maintenance activities – adaptive, perfective, preventive, and corrective maintenance – require that the maintenance engineer comprehend the program to be changed, understands how the program works, and how to make the desired changes without introducing new defects or breaking existing functionality (Vans et al. 1999). To reduce maintenance costs, researchers have studied factors influencing the effort of individual maintenance engineers. In particular, researchers have focused on studying how characteristics of the software product itself influence the effort required to modify the software, as well as how maintenance engineers come to understand the software to be modified.

Studies of how the software product influences maintenance effort have focused on the relationship between maintainability and maintenance effort. Factors studied range from low-level syntactic structures such as code complexity (Gibson and Senn 1989; Banker et al. 1993), to high-level program structure such as design patterns (Voká et al. 2004).

Based on the observation that maintenance engineers spend half their time studying source code and documentation (Oman and Cook 1990), researchers have studied how maintenance engineers understand source code. A number of models have been proposed to describe the cognitive processes used to acquire program comprehension (von Mayrhauser and Vans 1995). The models show how engineers use existing knowledge of the software to build new mental models of the software being maintained. The strategies employed for building mental models vary between the models (Shneiderman and Mayer 1979; Brooks 1983; Letovsky 1986; Pennington 1987; von Mayrhauser and Vans 1995). These models have been used to compare the effect of programming languages and paradigms on maintenance effort (Corritore and Wiedenbeck 1999; Corritore and Wiedenbeck 2001). Similarly, Shaft and Vans (2006) study how the fit between individual maintenance engineers' mental models of the software on the one hand and the modification tasks on the other impacts on the maintenance effort.

3.1.4 Corrective maintenance

Corrective maintenance has been defined as the activities performed to correct defects in hardware and software (IEEE 1990). The point of departure for corrective maintenance is failing software. The core activity is therefore to correct underlying software defects. Early software maintenance research makes no distinction between

errors in code and errors in program behaviour, using the term error for both (Basili and Perricone 1984). While the terminology has been refined, there is still a lot of confusion in both the terms used and their interpretation among software maintenance researchers (Fenton and Neil 1999). However, underneath the differences in terminology and interpretations of terms, it is possible to identify a common causal model of software errors. Before progressing with a description of this model, however, it is necessary to clarify the terminology used, summarized in Table 3-1 below.

Term	Description
Mistake	Human error that is manifested in the source code as a defect
Defect	An incorrect statement of sequence of statements in the source code that may lead to an infection upon execution
Infection	An error in the program state that may lead to a failure
Failure	Externally visible deviation from correct program behaviour compared to requirements and specifications
Infection chain	A causal chain from defect to failure

Table 3-1 Key terms in the causal model of software errors

The terms mistake, defect, infection, and failure are used to distinguish between different types of errors (Zeller 2006). The mistake is a human error. The mistake is manifested in the program code as a defect. The defect is an error in the code. Upon execution, the defect produces an error in the program state, an infection. The infection may, or may not, lead to a failure.

A failure is an externally observable error in program behaviour. The infection relates to the defect as a product of an executed defect. The causal chain from human mistake to failure is called the infection chain.

The defect is also sometimes called a *latent error*. It is latent because the defect may not be executed during operation of the software, or the code may only produce an infection in very special cases. The relationship between the two is therefore contingent, depending upon the execution paths through the software. The term *error trigger*, defined as "the events that cause latent errors in programs to surface" (Sullivan and Chillarege 1991, p. 2), is also used to explain the relationship between defect and infection.

The infection is itself latent, as an incorrect program state need not produce externally observable incorrect program behaviour. There is no 1-to-1 relation between mistake and failure (van Vliet 2000). For instance, an incorrect program state that is never used during execution will not lead to a failure. Conversely, a failure may be caused by more than one defect. A single defect may also cause several failures. Furthermore, defects may be hidden so deep in the software that it is impossible to locate and the defect is identified by the correction made (Endres 1975).

While it is widely acknowledged that it is often difficult to determine what the defect really is, the corrective maintenance literature find the causal model of software errors useful. Figure 3-2 summarizes the model.

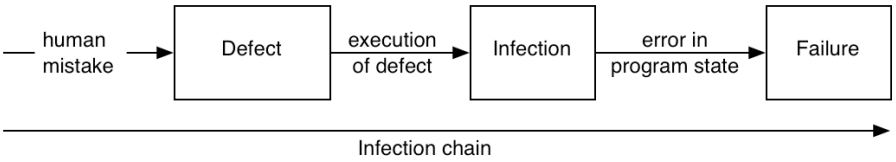


Figure 3-2 Causal model of software errors

For the maintenance organization, two concerns need to be balanced when handling submitted problem reports. On the one hand, the need to correct failures that users experience. On the other hand, the need to prioritize failures with the most impact on the largest population of users. Some failures are more critical than others. Failures where the software crashes or it corrupts critical data are typically more critical than minor flaws in the user interface, for instance. It is therefore more important to address critical failures. However, as failures are magnitudes more expensive to correct during maintenance compared to testing, the criticality of the failure needs to be traded off against the population it affects. While critical to those affected, it may not always be cost-effective to correct failures that affect only a single user or a miniscule population of users (Adams 1984). Similarly, minor failures may be prioritized if they affect a large population of users.

3.1.5 Debugging

Debugging encompasses the activities of analyzing and correcting reported failures. Analyzing the reported failure, is the activity of tracing along the infection chain from failure to defect (Cleve and Zeller 2005). The basic challenge facing any maintenance engineer is to determine the cause of reported failures (Endres 1975). As the maintenance engineer responsible for correcting failures rarely have direct access to the failing system, replicating the operating environment where the failure occurs is the first step towards analyzing the reported failure. From the maintenance engineer's point of view, the problem report therefore needs to contain sufficient information for the maintenance engineer to be able to replicate the operating environment as well as reproduce the failure (Zeller 2006).

Once reproduced, the cause of the software failure has to be located; the defect leading to the infection and consequently the failure. A common observation among researchers is that maintenance engineers spend a lot of time chasing red herrings because there is little understanding of how to systematically debug software (Martin and McClure 1983; Araki et al. 1991; Zeller 2006). To this end, software maintenance researchers have offered a number of techniques for locating faults like program slicing (Xu et al. 2005), delta debugging (Misherghi and Su 2006), and hypothesis-driven debugging (Araki et al. 1991). With basis in the source code and additional data like documentation and stack traces, these techniques offer ways of analyzing failure by either from chunking statements in the source code to higher-level abstractions or mapping knowledge of the problem domain to the source code (von Mayrhauser and Vans 1995).

Once the defect is located, the maintenance engineer corrects the failure and verifies that the failure no longer occurs when trying to reproduce it. Tracing backwards along the infection chain, debugging can be modelled as a linear process going from well-defined failures, through locating the defect, to correcting it as suggested by the grey line in Figure 3-3 below. As such, it builds upon the causal model of software errors.

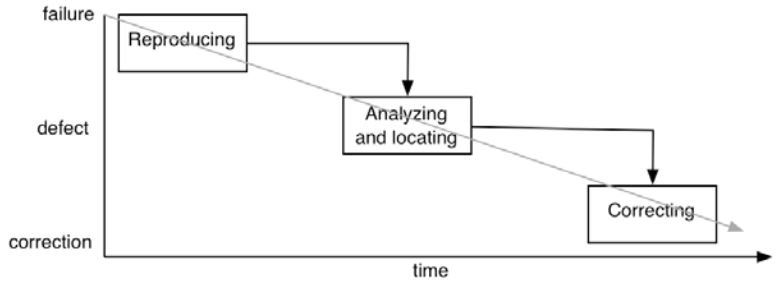


Figure 3-3 Debugging activities

3.2. Legacy systems

Software systems survive over time because they are adapted to the changes in the operating environment (Bennet 1995). If no remedial action is taken, however, the structural integrity of software systems will deteriorate (Eick et al. 2001). A legacy system is the product of software evolution (Lehman 1980). It is a software system that has survived over time, and is becoming increasingly difficult to modify (Bisbal et al. 1999). Yet, it is critical to the host organization and cannot be disposed of easily.

This chapter discusses the product of long-term software maintenance: legacy systems. To this end, it is organized as follows. First, a brief introduction to software evolution is given (3.2.1). Legacy systems (3.2.2) and the increasing maintenance costs incurred by them (3.2.3) is then discussed. The dilemma of keeping or replacing legacy systems is presented (3.2.4), before the chapter is concluded with coping strategies for legacy systems (3.2.5).

3.2.1 Software evolution

With some recent additions (Eick et al. 2001), software evolution research builds predominantly on the research reported by Belady and Lehman during the mid-1970s and early 1980s (Belady and Lehman 1976; Belady and Lehman 1978; Lehman 1979; Lehman 1980). Software evolution research takes the result of the software maintenance process as its object of study, software that has evolved over time due to maintenance. The research builds upon and seeks to explain the observation that large-scale software over time becomes increasingly difficult to modify.

Software evolution research shows that software systems that survive over time do so because they are able to adapt to an evolving operating environment (Bennet 1995). With basis in change data from IBM, Belady and Lehman propose that the direct cause of the increasing maintenance burden is that over time the software's structure

deteriorates. Software evolution research also shows that as software systems change over time, they become increasingly difficult to maintain. This phenomenon has been called systems entropy (Belady and Lehman 1976) or more recently code decay (Eick et al. 2001). As the code decays, the maintainability of the legacy system decreases and it becomes increasingly costly to modify the system.

While the direct cause of the increased maintenance burden is code decay, two dynamics leading to code decay have been identified.

First, software evolves over time (Belady and Lehman 1976). It does so in order to respond to the changing functional requirements of the host organization. Unless the software adapts to the host organization's changing environment, it will be rendered obsolete (Parnas 1994). As such, there is a direct relation between the longevity of the software and the amount of changes it has undergone.

Second, through adaptive maintenance, new functionality is added to the software. However, the nature of the changes and the process by which they are made may impact on the software structure. In some instances, the original program structure may not have been conceived with the new functionality in mind. As such, new program structure violating the original design principles has to be superimposed on the existing design to make the required changes (Lehman 1979). Developers unfamiliar with the design or with too little time to assess how best to implement modifications in accordance with the design may also violate the original design (Eick et al. 2001). Over time, software therefore often acquires layers of superimposed program structure.

Unless remedial work is undertaken to amend the program structure, the code will therefore decay (Belady and Lehman 1976). Rephrased in software engineering terminology: unless preventive maintenance is undertaken to deliberately amend the program structure, the maintainability of the software will decrease. As such, the implications for software maintenance are clear. Maintainability is not something to be established once and for all through ensuring that quality attribute requirements are met during development (Boehm 1978; Cavano and McCall 1978). Nor is it merely established through choice of appropriate architectural techniques during design (Bass et al. 2003). Rather, continuous restructuring (Mens and Tourwe 2004) of the software is required to unify the design to avoid a layering of design.

3.2.2 Legacy systems explained

Computerization has increased over the past 40 years. Many organizations find their portfolio of software systems growing, and many of these portfolios contain software systems that have been long-lived, but are still in operation (Swanson and Dans 2000). Aging systems often resists modification significantly. They therefore constitute a significant part of the host organization's maintenance burden. These systems are often called *legacy systems* (Brodie and Stonebraker 1995).

While expensive to maintain, legacy systems are often difficult to decommission. The systems and the data they contain are vital assets for the host organization. They are typically the backbone of the host organization's information flow and the main vehicle

for consolidating information (Bisbal et al. 1999; Bianchi et al. 2003). Although constituting a significant maintenance burden, legacy systems are still operational because they have remained business-critical over an extended period of time. In its exclusive form, the term 'business-critical' is used about software whose failure may result in the failure of the business using the system (Sommerville 2001, p. 357). However, in a more inclusive form the term may be used about any software that is critical to the organization. Such an inclusive view of business-critical software also encompasses software providers whose survival relies on providing the software.

Legacy systems are therefore typically distinguished by two defining characteristics:

Characteristic	Description
Business-critical	The software system provides data and functions that are critical to the organization
Aging	The software system has continued to be business-critical by evolving in response to the host organization's changing needs over time

Table 3-2 Characteristics of legacy systems

3.2.3 Increasing maintenance cost

The challenge facing host organizations is the increasing cost of adapting the legacy system to its changing environment. It has been of particular interest to identify the factors contributing to the increasing maintenance cost. Many factors have been suggested. One way of summarizing the factors contributing to the increased maintenance cost of legacy systems is to split them into internal and external factors. *Internal factors* are related to the legacy system's resistance to modification. *External factors* are related to the host organization and its environment.

3.2.3.1 Internal factors

Internal factors to increasing maintenance cost are summarized in Table 3-3 below.

Factor	Description
Deteriorating systems structure	Poor system structure (code decays) increases the maintenance effort and makes the introduction of new faults during maintenance more likely
System largeness	System largeness make program understanding a major, time-consuming maintenance activity

Table 3-3 Internal factors to increasing maintenance cost

Deteriorating system structure has two causes: code decay and outdated programming techniques. Legacy systems remain relevant and thereby business-critical over an extended period of time by adapting to the host organization's changing environment (Lehman 1979). However, through continued adaptive maintenance over time its system structure deteriorates unless work is done to maintain or reduce system complexity (Lehman 1980, p. 216); the code decays. As the code decays the effort required to modify the source code increases (Eick et al. 2001). Outdated programming techniques such as variable aliasing and the use of single, large global data structures to save memory, for instance, may also have a negative impact (Bennet 1995). Such techniques may encourage types of maintenance that quickly degrade the systems structure.

Research suggests that larger systems tend to be longer-lived than smaller ones, as they are not perceived as "not so much a burden of maintenance as they are assets expected to provide corresponding returns to maintenance over a longer time period" (Swanson and Dans 2000, p. 294). Besides, small programs are usually not difficult to maintain (Bennet 1994). While system size is an indicator of system largeness, a system is large when it lies beyond the grasp of a single individual and must be maintained by a group of people (Belady and Lehman 1978). As such, system largeness also requires the coordination of people within teams, coordination among teams within an organization, and even coordination between organizations to perform. This increases the need for communication and coordination, and may also drive the maintenance costs up.

Both deteriorating systems structure and systems largeness cause increasing maintenance cost, summarized in Figure 3-4 below.

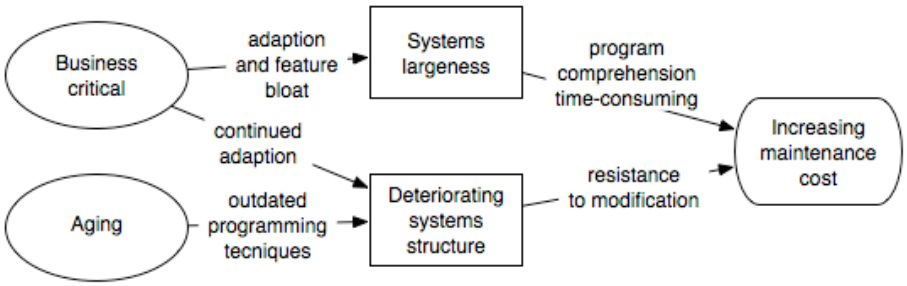


Figure 3-4 Relation between characteristics of legacy systems and internal factors

3.2.3.2 External factors

External factors for increased maintenance costs are related to the host organization and its environment. The factors are summarized in Table 3-4 below.

Factor	Description
Obsolete hardware platform	Hardware is expensive to maintain
Obsolete software platform	Legacy system lack clean interfaces and/or host organization's other software systems lack software for integrating with the legacy system
Lack of skills	The skills needed to maintain legacy systems are in short supply

Table 3-4 External factors to increasing maintenance cost

Both obsolete hardware and software platforms may be attributed system age. Obsolete hardware is less in supply, and therefore more expensive to acquire. As the functions and data provided by legacy systems are critical to the organization, there is a need for newer software systems to integrated with the legacy system. Obsolete software may make it more difficult to integrate the legacy system with new software systems in the host organization's portfolio. As such, the obsolete software platform is also attributable to the business-criticality of legacy systems.

Lack of skills can also be attributed system age. Over time knowledge of the legacy system details may be lost as the people who originally developed and maintained system leaves the host organization (Bisbal et al. 1999). Many legacy systems are also

poorly documented. Maintenance cost increases as new maintenance engineers need to learn system details. However, that engineers knowledgeable in the obsolete hardware and/or software platforms may be in short supply also may also drive the maintenance cost up.

Figure 3-5 below summarizes the relation between increasing maintenance cost and external factors.

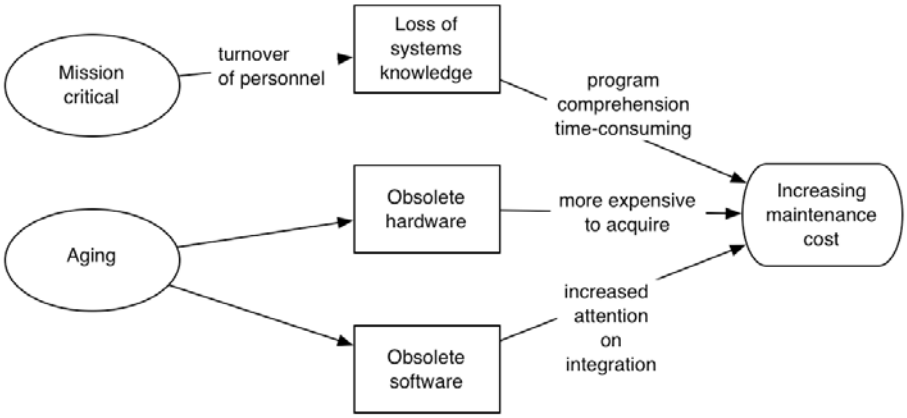


Figure 3-5 Relation between characteristics of legacy systems and external factors

3.2.4 The legacy systems dilemma

Legacy systems often pose a dilemma to host organizations. On the one hand, they constitute a significant and potentially increasing maintenance burden. On the other hand, they are business-critical and cannot be decommissioned. Herein lies the dilemma: *continued maintenance as well as systems decommission and replacement constitute a significant investment and risk for the organization.*

There are three dimensions to the dilemma summarized in Table 3-5 below: cost, risk, and adaptability.

Dimension	Continued maintenance	Systems replacement
Cost	Continued maintenance becomes increasingly expensive	Systems replacement a major organizational investment
Risk	Comprehensive testing difficult, and new faults may be introduced	New faults may be introduced during re-implementation
Adaptability	Responsivity to adaptation can no longer be appropriately sustained	Redevelopment is time-consuming, requiring the legacy system to be stable and irresponsive to adaption during the period of reimplementation

Table 3-5 The legacy systems dilemma

Both maintaining and replacing legacy system constitutes a significant organizational investment. The increased cost of continued maintenance may be related back to all of the external and internal factors (see Table 3-3 and Table 3-4 above). Because legacy

systems are business-critical, their failure can potentially have serious impact on business (Bennet 1995). The risk of continued maintenance is related to deteriorating system structure. The chances of introducing new faults during maintenance increases as the code decays (Eick et al. 2001). Adaptability of a legacy system may be related back to both deteriorating system structure and obsolete software. Deteriorating system structure may make adaption impossible (Bisbal et al. 1999), and obsolete software may make it hard or even impossible to integrate with new systems.

3.2.5 Coping with and replacing legacy systems

Because of the legacy system dilemma legacy systems can be understood as large software systems that host organizations don't know how to cope with but that are vital to the organization (Bennet 1995). As such, research on legacy systems has typically focused on:

- Methods for coping with the problems of legacy systems
- Models for determining when it makes more economical sense to replace a legacy system rather than keep on maintaining it

Models for the timing of systems replacement (Taizan et al. 1996) is outside the scope of this thesis, and will not be discussed further. However, methods for coping with the problems of legacy systems have been placed along the range of system evolution to system revolution (see figure ?? below).

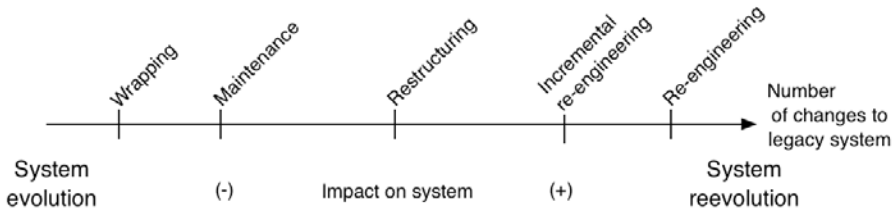


Figure 3-6 Methods for coping with legacy systems (adapted from Bisbal et al. 1999)

3.2.5.1 Coping strategies

At the far right of the scale is *re-engineering*. Re-engineering constitutes a redevelopment of the legacy system in a different programming language and/or in a different operating environment. It also often encompasses the restructuring of data. As such, software re-engineering is altering the implementation of an existing system while the basic functionality remains the same (Sneed 1995). This is time-consuming and requires a large organizational investment. Furthermore, legacy systems replacement faces the cut-over problem (presented below). Data salvage is a variant of re-engineering, which does not fit properly into the figure above. In this approach, the software itself is replaced by other systems. The data, however, remains the same.

At the other end of the scale is *wrapping*. This coping strategy seeks to encapsulate the legacy system. As such, it is a form of system integration (Hasselbring 2000). Rather

than modifying existing legacy systems, new systems interfacing with the legacy systems are introduced. By leaving the legacy system intact, wrapping seeks to bypass the problems related to continued maintenance as well as systems replacement. However, legacy systems wrapping may be expensive and difficult because of architectural mismatch between the legacy system and the systems seeking to interface with it (Sneed 2005). It is even argued that some forms of wrapping, like screen scraping, may compound the organization's maintenance problem as such wrappers do not address the serious problems legacy systems face (Bisbal et al. 1999).

Restructuring holds the middle ground between re-engineering and wrapping. It is a form of perfective maintenance that seeks to decrease the cost and risk of continued maintenance by improving the system's maintainability (Mens and Tourwe 2004). It is not an activity exclusive to coping with the legacy systems dilemma, but addresses maintenance in general. However, as restructuring seeks to mend deficiencies with the existing system, it seeks to ammend problems related to code decay without as well as bypassing problems related to systems replacement. While a viable strategy with newer systems, some of the maintenance cost of legacy systems stems from obsolete platforms. As such, restructuring only resolves the internal factors to increased maintenance cost.

3.2.5.2 The cut-over problem

Coping strategies that seek to replace existing legacy systems with new software face the cut-over problem. The cut-over problem is related to time it takes to develop the replacement system. There are two dimensions to this problem:

- Continued evolution of legacy system
- Volatility of data

Business-critical systems continued to evolve in response to the changing conditions of the host organization. However, in order to succeed with a 1-to-1 replacement of the legacy system, it needs to be functionally stable from the moment the re-engineering effort is planned until the system is replaced (Sneed 1995). The longer it takes to develop the replacement system, the harder it is for the host organization to avoid continued evolution of the system to be replaced.

Incremental re-engineering, sometimes also called *systems migration*, seeks to address the problem of continued evolution. By gradually re-engineering a few procedures at a time, each re-engineering operation takes so short time that it is possible to freeze modification of the procedures in question (Bianchi et al. 2003). While addressing the problem of continued evolution, incremental re-engineering is still susceptible to the cutover problem.

That the information contained in legacy systems may be vital assets for the host organization is a compounding factor to software re-engineering. As long as legacy systems are in use, new data will be added and existing data modified. This leads to the cutover problem. An un-concerted transition may cause data to be store in non-synchronized databases: some in the legacy system and some in the replacement system. As it is practically impossible to freeze the data in a legacy system while re-engineering

it, the data needs to be transferred to the replacement system. This may be time-consuming, and business-critical system cannot be out of operation for very long.

Software integration has been proposed as a solution to the cut-over problem (Hasselbring 2000). By integrating new information systems with the legacy system, the problems of continued evolution and as well as volatility of data are bypassed.

3.3. Maintaining integrated systems

Current software maintenance research focuses upon the maintenance of application software maintenance. Banker et al. (1993) define application software as a set of software modules performing a coherent set of tasks in support of a given organizational unit and maintained by a single team. Despite increased interest in systems integration through component-based software development (Boehm and Abts 1999), Web services (Vogels 2003), and information and enterprise integration (Lam and Shankararaman 2004), limited attention has so far been paid the implications of systems integration on software maintenance (Mockerjee 2005).

The purpose of this section is therefore to discuss software integration in relation with software maintenance. To this end, the section is organized as follows. First, software integration discussed (3.3.1). Then characteristics of integrated systems are presented (3.3.2), before the section is concluded with a discussion of challenges related to maintaining integrated systems (3.3.3).

3.3.1 Software integration

Software integration is the activity of building integrated systems. This thesis assumes an inclusive view of software integration, encompassing activities from component-based development (Boehm and Abts 1999), through middleware integration (Vogels 2003), to information and enterprise systems integration (Hasselbring 2000). While there are differences in scope and purpose between the different forms of software integration, software integration is characterized by its stakeholders.

To understand the uniqueness of software integration, it is useful to compare it with application software development. There are two stakeholders – the developer and the user – in the most simplistic form of application software development. While some software systems may have multiple user groups, the basic scenario remains: someone uses the software, someone else maintains it. Similarly, teams or collections of teams within the organization often develop large systems. This modifies the most basic model of application software development somewhat. However, the organization as an entity is still in charge of the product, though (Banker et al. 1993).

The basic premise of software integration is that software is composed of software provided by third-party organizations. Rather than developing an application, the software developer composes software by integrating third-party software. The developer is therefore called a *software integrator*. Whereas constellation of integrator-user remains in software integration, a third stakeholder is introduced: the third-party

organization providing software, i.e. *the provider*. While providers may target software to specific organizations, they typically offer software with generic functionality within a specific domain. There is a one-to-many relationship between providers and software integrators. The provider's product is offered to many integrators. However, while some systems do indeed only integrate a single product, large-scale integration involves the integration of multiple products. As such, seen from the software integrator's point of view, the provider-integrator relationship is also one-to-many. The software integrator is typically in a relationship with many providers.

3.3.2 Integrated system

Application software is developed and maintained by a single team or organization that has full access to and is in complete control of the source code. *Integrated systems*, on the other hand, are composed of software products developed and maintained by third-party organizations. Coming with more or less well-defined interfaces, the software products being integrated are usually treated as black boxes. Building upon the inclusive view of software integration, integrated systems may be composed of software components to enterprise information systems. In contrast to application software, no single stakeholder is in complete control of an integrated system.

Hasselbring (2000) identifies three distinguishing characteristics of integrated systems. These are presented in Table 3-6 below. This table is expanded somewhat to differentiate between technical and organizational aspects of the three characteristics.

Characteristic	Technical	Organizational
Heterogeneity	Difference in hardware platforms, operating systems, and programming languages. Conceptually, different programming and data models.	Software maintained by multiple organizations organizing the maintenance process differently (see 3.1.2)
Autonomy	Software may be autonomous in execution, running on different computers within different organizations. Third-party software develops independent of the integrator's product.	Providers are in control of the development trajectory of their own products, with limited or no coordination among providers.
Distribution	Integrated systems may be comprised of software executing on different computers. This is not always the case for COTS-based systems, for instance.	The maintenance effort is distributed among multiple organizations, both integrators and providers.

Table 3-6 Characteristics of integrated systems

3.3.3 Challenges for maintaining integrated systems

While systems integrators have access to the source code of the overall product they develop and maintain, they have limited, if any, access to the source code of the software they integrate. Furthermore, studies show that even when third-party software comes with source code, systems integrators tend not to spend time and effort to read and understand the product (Li et al. 2007). Consequently, one of Belady and Lehman's (1978) well-known characteristic of systems largeness – that the system is outside the

intellectual grasp of a single individual – is accentuated when maintaining integrated systems. During maintenance of integrated systems, the system is even outside the grasp of a single group of individuals, too, as not even systems integrators fully grasp the products they integrate. Whereas the software maintenance research in general, and the corrective maintenance research in particular, builds on the assumption of source code as the key for systems knowledge, this is increasingly problematic with regards to maintaining integrated systems.

Whereas a single team or organization is in control of the source code of application software (Banker et al. 1993), systems integrators have at best limited influence on the development trajectory of the third-party software they integrate. The direction, extent, and timing of changes to third-party software – in short the software's development trajectory –is under the provider's control (Hybertson et al. 1997). A variant of this is the situation where the provider goes out of business (Voas 1999). As such, maintaining integrated systems need to cater for the evolution of the third-party software. Several suggestions have been proposed (Hybertson et al. 1997; Edwards et al. 1999; Carney et al. 2000).

Again, systems integration accentuates a characteristics of systems largeness: that the system reflects within itself a variation of human interests and activities (Belady and Lehman 1978). In the case of systems integration this relates to the multi-organizational relationship between vendors and integrators. Although well known to software maintenance research for decades, the characteristics of largeness are only reflected to a limited degree within the maintenance literature. For instance, with a few notable exceptions (Vans et al. 1999), studies of program software maintenance activities are based upon small-scale activities within the intellectual grasp of a single developer. Similarly, whenever variety of interests is addressed by the legacy systems literature, which is rarely, it is only superficially. Sneed (1995), for instance, delegates variety of interests to an issue of establishing the cost of maintaining the existing portfolio, and then justifying the systems replacement by demonstrating that re-engineering will provide a long-term return of investment.

The differences between application software and integrated systems maintenance are summarized in Table 3-7 below.

Characteristic	Application software maintenance	Integrated systems maintenance
Source code	Maintenance team full access to source code	Systems integrators limited if any access to source code
Ownership of products	Maintenance team in complete control of source code	Software being integrated is developed and maintained by numerous third-party organizations
Control over products	Host organization which the development team is member of is in complete control of the development trajectory of the application software	Third-party organizations developing and maintaining software being integrated controls the trajectory of the product
Program comprehension	Small-scale maintenance activities graspable by single individual	Large systems outside the intellectual grasp of a single individual

Table 3-7 Comparison application software and integrated systems maintenance

4. Open source software and software engineering

The reported research has studied maintenance of an integrated system in practice. This has been done in the context of a distributed community of volunteer software integrators who develop and maintain an OSS product. Developing and maintaining OSS in such distributed communities is often called OSS development. Yet, OSSD is not merely the research setting of the reported study. In this chapter it serves as a concrete example of how increased scientific rigour may contribute to further exacerbating research's lack of relevance to practice. This is based on the observation that after over a decade of research, OSSD remains largely irrelevant to the broader field of software engineering. While there is a significant stream of OSS research within software engineering, this research predominantly focuses upon how to use OSS to develop new products. Software engineering research on OSSD, however, remains limited.

The argument pursued in this chapter is as follows. Driven by the unquestioned assumption that OSSD is completely different from software engineering, software engineering researchers have applied scientific rigorous methods to determine in what ways OSSD is different from software engineering (Østerlie and Jaccheri 2007a). Yet, if the goal of software engineering research is to be relevant to practice, situating OSSD in such an otherness relation removes its practical relevance. Based on the assumption that OSSD is completely different, software engineering research on OSSD therefore remains largely irrelevant as it fails to inform software engineering practice.

The above argument is pursued in three steps. Section 4.1 traces the origins of the mythologized view of OSS as radically different to software engineering. Section 4.2 provides an overview of the two major OSS research agendas pursued within software engineering studies of OSSD, and studies of developing software with OSS. Returning to the problem of relevance, Section 4.3 concludes the argument by showing how studies of OSSD may contribute to software engineering by focusing on showing how OSSD differs from software engineering. The result of continually situating OSSD in an otherness relation to software engineering is that OSSD remains a piece of curiosa that is largely irrelevant to software engineering.

4.1. Open source software in context

Many have tried to grasp what OSS *really* is (Gacek and Arief 2004). Yet, the prevailing view of OSS is still software of superior quality developed through a revolutionary new software development approach by collectives of supremely talented volunteer software developers (Fitzgerald 2006). This view of OSS is based upon the mythology presented and circulated by OSS proponents since the mid-1990s. To understand this mythologized view of OSS, we need to go back to the specific point in time from which the term and its mythology arose: the state of the computing industry in the mid-1990s.

Before progressing with this, however, a brief outline of how the mythologized view has been constructed by drawing historical lines of descent and the purpose such mythologizing has served is presented.

4.1.1 The mythologized view of OSS

Many have sought to understand OSS by tracing its historical origins (DiBona et al. 1999; Feller and Fitzgerald 2002). Some trace these origins back to the community of hackers at MIT's Artificial Intelligence (AI) Lab during the 1960s and 70s (Levy 1984). Others trace the origins back to the Unix-based ARPANET community of the 1970s and 80s (Moody 2001). Yet, others trace the origins of OSS to the convergence between the two communities in the Free Software Foundation (FSF) of the 1980s (Hannemyr 1999).

Regardless of which historical line is traced, attempts at tracing the origins of OSS all share the narrative of the hacker. The narrative of the hacker emphasises individual technical prowess and technological innovation (Himanen 2001). In this narrative, the hacker is the mythological maverick, the lone outsider who succeeds against superior odds. Fitzgerald (2006) argues that almost every aspect of this myth can be questioned. It is therefore useful to understand where such a view of OSS emerged. To do so we have to go back to the state of the computing industry in the mid-1990s and the rise of the New Economy.

4.1.2 Linux, OSS, and the New Economy

The OSS term has a definite origin in time and space. It was coined in 1997 at a meeting between Linux proponents and the fledgling Linux industry (Perens 1999). The meeting takes place at a time when Microsoft dominates desktop computing. A latecomer in the Internet-market, Microsoft is by now also amassing market shares by distributing its Web browser together with its operating systems. Once actors in an open marketplace, Microsoft's competitors are rapidly loosing ground as the latecomer tightens its grip on the browser market.

Similarly, since the late 1980s the Unix industry has been loosing ground to the more popular operating systems provided by Microsoft. However, by 1997 Linux is gaining increased popularity. Linux is a Unix-like operating system kernel developed by a

young Finish student and a ragtag band of volunteers. Without any financial backing its popularity is unlikely. At the same time, the Apache Web server dominates the Web server market. Like Linux, Apache has until then been developed by a loosely organized gang of volunteers with practically no financial backing (Østerlie 2003).

By 1997, the promises of an Internet-based New Economy are gaining foothold. Investors are looking for new, and revolutionary business models (Behlendorf 1999). In this environment with Microsoft's market dominance on the one hand, and venture capital looking for alternative investments on the other hand, Linux proponents and representatives of the Linux industry see an opportunity and cease the moment. Together they form the Open Source Initiative (OSI). The term 'open source' is proposed to overcome resistance among investors to the politicized freedom discourse of Free Software (Perens 1999). Instead, the OSI seeks to further OSS as a fruitful venue for investment.

It is in this context that we can understand the role genealogical lines of descent play in giving meaning to the term OSS. Mobilizing history in this particular way OSS proponents sought to establish OSS as both technically superior and innovative, as well as building an identity of the outsider who succeeds against superior odds. It is the story of how the David of the Linux industry will prevail against the Goliath of its time: Microsoft.

4.1.3 Open Source Software, Free Software, Free/Open Source Software?

There is an ongoing controversy between the terms "free" and "open source" software. The Free Software Foundation, the organization administrating the GNU public licences does not agree that Free Software is a subset of OSS (Williams 2002). OSI, on the other hand, continues to regard Free Software as a subset of OSS. Free Software is licensed under the GNU public licenses. These licenses seek to ensure that software remains free. In this context, 'free' means freely and publicly available. Broadly speaking, the two GNU licences seek to ensure that nobody can derive commercial products based on GNU licensed software without providing the source code of derivative works.

Similarly, OSI formulated a set of guidelines, the open source definition, to ensure that the source code remained publicly and freely available. Open source software is therefore software released under an OSI compliant license (Gacek and Arief 2004). However, unlike the GNU licenses, OSI opens the possibility of creating licenses that allows building commercial derivative products based on OSS. As such, the GNU licenses are a restrictive form of OSI compliant licenses. Free Software therefore also falls under the umbrella of OSS.

To bypass or overcome this controversy, some researchers have opted for the term FOSS, Free Open Source Software (Scacchi 2007). The standard convention within the software engineering community, however, is OSS. The term OSS is therefore used in this thesis, too.

4.2. OSS research in software engineering

Software engineering research on OSS has typically pursued one of two separate agendas. The first agenda sees OSS as a source of applications and reusable components. The other agenda studies OSS development, emphasizing the unique characteristics and attributes of developing software in distributed communities of volunteers. Each of these agendas will be presented in turn. Before progressing, it is worth noting that there is also a stream of software engineering research that uses OSS as data source to test and validate non-OSS related theories. This agenda is unrelated to OSS. It will therefore not be discussed.

4.2.1 Developing with OSS

While some researchers have focused on the use of OSS development tools (e.g. Serrano and Ciordia 2005), developing software with OSS components has been the predominant stream of OSS research in software engineering. While research on developing with OSS components plays a limited role in the reported research, it is still outlined in this subsection. This is done to provide context later in the chapter for making the argument that while research on OSSD remains largely irrelevant to the software engineering field, research on OSS has been highly successful.

Research on developing with OSS components follows a long line of software engineering research on software reuse. The reuse literature revolves around the two issues of developing *for* reuse, and developing *with* reuse. Component-based software development is a form of developing with reuse that has gained increasing attention the past decade (Boehm and Abts 1999). Assuming the integrator's point of view, much research on commercial off-the-shelf software (COTS) has focused upon evaluation and selection of COTS. The core assumption of this research is that choosing the right software is considered critical to project success (Wang and Wang 2001).

OSS offers new possibilities for component-based development (Madanmohan and Rahul 2004). The main reasons cited are availability (Wang and Wang 2001), decreased development costs (Fitzgerald and Kenny 2004), as well as shorter time-to-market, less development effort, and better system quality (Li et al. 2006). In addition, OSS code and publicly available project information opens new possibilities for software integrators to evaluate OSS components in terms of product characteristics (Li et al. 2005) as well as project characteristics (Woods and Guliani 2005; Cruz et al. 2006). As such, following the line of research on evaluating COTS, research on developing with OSS has focused on ways of evaluating such components.

It has also been argued that developing with OSS components is less risky as the source code is available (Ruffin and Ebert 2004). Yet, in a survey of off-the-shelf component adoption in the Norwegian software industry, Li et al. (2007) finds that software developers practically always treat OSS as a black box even though the source code is available. Similarly, even though the source code of abandoned OSS is available, software integrators seek to avoid the responsibility of maintaining the software they integrate.

4.2.2 Open source software development

The advantages and limitations of OSSD is a topic of much debate within the software engineering literature. On the one hand, objections have been raised about the lack of formal development processes (Wilson 1999), the effects of having little or no explicit design (Perkins 1999), as well as the limitation of the user-developer convergence (Messerschmidt 2004). On the other hand, repeated claims have been raised about the advantages of OSSD over software engineering methods. These claims include increases in development speed of development, reduction of effort, and higher quality of the end product (Dinh-Trong and Bieman 2005).

However, Østerlie and Jaccheri (2007a) shows that such a research focus is based on the unquestioned assumption that OSSD is completely different from software engineering. This subsection goes more into detail on how OSSD has been studied as completely different from software engineering. It does so with basis in the view of OSSD as a particular software development approach characterised by close interaction between users and developers as well as being Internet-based. First, the key software development key practices are presented. Then, drawing upon OSSD as an Internet-based approach to software development, issues related to the organization of the development effort. Finally, the subsection is concluded with a discussion of how existing empirical studies seek to understand in what ways OSSD is different from software engineering.

4.2.2.1 OSSD as software development approach

The software engineering literature often relates OSS to a particular software development approach. The two distinguishing characteristics OSSD are that it is *Internet-based* and based upon *close interaction between users and developers*. The software produced through OSSD is OSS. However, not all OSS is developed through OSSD. This is an important distinction. In addition to close interaction between users and developers, a defining characteristic of OSSD is that those contributing with code are also users of the software (Gacek and Arief 2004). OSSD is therefore sometimes characterized as use-driven software development (Messerschmidt 2004).

The central role of use is reflected in both OSSD's basic quality assurance practices of field testing and parallel debugging, as well as in the practice of developing requirements through use. Supporting the above practices is a process of rapid releases. Each of the four will be discussed in turn.

Field testing and parallel debugging constitute the basic quality assurance mechanism of OSSD (Huntley 2003). Instead of testing the software thoroughly to pre-empt failures, software is released 'as is'. The software is tested through use in the field. This is done with the expressed intention that users notify the developers about software failures. It is the developers' task to correct reported failures. As such, there is a clear separation of work: "[s]omebody finds the problem, somebody *else* understands it" (Raymond 1998).

This way of working is often referred to as *parallel debugging* (Feller and Fitzgerald 2002). The parallelism of the debugging effort unfolds along two axes: discovery and correction. Subjecting the software to their use profile, users test the software in parallel

to discover failures. Failures are considered a collective responsibility of the developers, who work in parallel to correct the failures.

As there is a convergence between users and developers in OSSD, developers may also report software failures they discover through use. However, the convergence between users and developers are particularly important when it comes to product evolution. Rather than engaging in requirements analysis, *new functionality is discovered through use*. Østerlie (2003) shows how innovation in the Web server technology arise in the early stages of the Apache Web server project. The Apache developers were also Web masters using the Web server to offer services to clients. By using the Web server in different contexts, the Apache developers uncovered uses that lead to innovations that today is considered part of the Web server technology. Similarly, Scacchi (2002; 2004) illustrate how new functionality is discovered in OSSD through use-related practices.

OSS is released in a *rapid release cycle*. Rapid release cycle is often found to be the third defining practice in addition to parallel debugging and discovering new functionality through use. Rapid release cycles are particularly important for enhancing the efficiency of field testing. Frequently releases of corrections play two roles. First, it is a way of avoiding double work related to duplicate failure reports. Second, through successive releases the software becomes increasingly reliable. The use of rapid release cycles has lead some researchers to consider (somewhat misguided) OSSD as a form of agile software development (Cockburn 2002).

There is sometimes made a distinction between unstable and stable releases (Erenkrantz 2003). When the software is released for field testing, the release is considered unstable. A release is considered stable when it has been subjected to subsequent rounds of field testing. Samoladas et al. (2004) interpret the frequent releases of updated software as a perpetual cycle of corrective and adaptive maintenance. As such, OSSD may be considered a form of prototyping.

4.2.2.2 Organization of software development

There has been some interest in OSS as organization of software development in distributed communities of volunteers. However, this line of research has been studied more closely within disciplines on the than software engineering. These studies have focused on issues such as the structure of volunteer communities (Crowston and Howison 2005) and the culture of such communities (Ljungberg 2000; Bergqvist and Ljungberg 2001) within information systems research, as well as developer motivation within economics (Bonaccorsi and Rossi 2003). Within software engineering, though, research on OSS as a form of organizing software development, has predominantly focused on distribution and coordination of effort.

Interest in the distribution of effort revolves around two concerns. First, is the issue of shared understanding of the software. Within software engineering an explicit system design is used to build a shared understanding of the software among those working on it. Without a design, then, the question is how such a shared understanding of the system attained? Second, is the issue of coordination. Given that OSSD is based on volunteer work, it is hard to assign tasks. Rather, volunteers undertake tasks. Who, then,

will do the boring tasks? Without a formal organization to distribute work, how is work coordinated within large communities of volunteers?

Mockus et al. (2002) finds that in the Apache community a group of 15 developers contribute with over 80% of the code. Yet, in the Mozilla community they find that groups of 22 to 35 developers contribute with the same amount of code to different parts of the software. The Apache community has an informal organization, where work is undertaken rather than assigned. However, the Mozilla community exercises a form of code ownership. With basis in these observations, it is proposed that for groups of more than 15 developers, explicit mechanisms of coordination are required. However, in groups of 15 or less, informal coordination mechanisms may work. The former proposal is corroborated by Dinh-Trong and Bieman (2005).

Summarizing research on OSS organization of software development, Crowston and Howison (2005) propose an onion layered model. This model illustrates the observation that a small group of developers contribute with most of the functionality. In a layer outside, are co-developers who contribute with corrections and some functionality. Active developers contribute with problem reports, and exist in a layer outside the co-developers. As such, although informally organized, communities of volunteers exhibit a clear distribution of work.

4.2.2.3 Empirical studies of OSSD products

Most of the empirical studies of OSSD within software engineering has focused on studying if and in what ways OSSD is different from software engineering. In particular, the claim that OSSD produces more reliable software than commercially developed software, or closed source software (CSS) has received much attention. Paulson et al. (2004) investigates the claim of higher reliability by comparing two OSS with two CSS products. Their study shows that the two OSS products have fewer defects than their CSS counterparts. While having no comparative data on the response time of problem reports, Paulson et al. (ibid.) attribute the difference in defect density to OSSD's rapid cycle of releases.

Such a claim, however, may be supported by Mockus et al.'s (2002) comparative study of the Apache and Mozilla OSS products. Like Paulson et al. (2004), they find that the two OSS products exhibit a low defect rate. This may seem to corroborate Paulson et al.'s (ibid.) assertion that defect density is caused by rapid response to user problems. However, Mockus et al. (2002) find that development of the Apache Web server and Mozilla browser exhibits different response times. Apache responds rapidly to problem reports, while Mozilla responds much slower. The difference in response time is attributed the fact that Apache is a volunteer project (it was later to be backed by IBM), while Mozilla is company-backed. Mockus et al.'s (ibid.) conclusion is therefore that commercial software development inhibits rapid response time. As such, in contrast to Paulson et al. (2004), they attribute low defect density to the use of field testing . Replicating Mockus et al.'s (2002) study on the FreeBSD kernel, Dinh-Trong and Bieman (2005) corroborates that field testing leads to lower defect density.

However, if field testing leads to lower defect rate, then OSSD should exihbit a greater degree of corrective maintenance compared to commercial software development.

Scach et al. (2002) compares the distribution of maintenance categories in two large OSS products with a CSS product. On the one hand, they find that the distribution of corrective maintenance in the OSS products is twice that usually reported within software engineering. On the other hand, they find the same distribution of maintenance activities in the CSS product. Using the distribution of maintenance categories as an indication of field testing, the evidence to support field testing as superior to pre-release testing is at best inconclusive. Consequently, claims that specific characteristics of OSSD should produce more reliable software products therefore remains unsubstantiated.

Similarly, it has been claimed that OSSD produces more maintainable software. However, evidence to support such a claim is also inconclusive. On the one hand, Capra et al. (2007) find that OSS products show lower levels of entropy than CSS. This corroborates the assertion that OSS is more maintainable than CSS. However, these findings are contradicted by Samoladas et al. (2004). Studying the maintainability of five OSS products, Samoladas et al. (ibid.) find that these products suffer from the same deterioration of maintainability as previously reported with CSS. Yu et al. (2004) find similar deterioration of maintainability in a study of 400 successive releases of the Linux kernel. However, in a later comparison of Linux with FreeBSD, NetBSD, and OpenBSD Yu et al. (2006) find that only the Linux kernel suffer from maintainability deterioration.

The conclusion to be drawn from existing empirical research is that the evidence to support claims of that OSSD or the product it produces are uniquely different from software engineering. Yet, researchers keep claiming that OSSD is uniquely different from software engineering (Scacchi 2007). The problems this causes for the relevance of OSSD to software engineering is discussed in the next section.

4.3. Rigour and irrelevance in software engineering research on OSSD

Early predictions that OSSD will revolutionize the way software is developed have failed to come through. Similarly, after a decade of research, the relevance of OSSD to the broader field of software engineering remains limited. Only a handful of studies of OSSD have yet to be published in software engineering outlets. The predominant focus of software engineering research is issues related to developing with OSS. It is therefore safe to say that OSSD research remains largely irrelevant to the broader field of software engineering.

Fitzgerald and Kenny (2004) makes a similar observation. They argue that OSSD research lacks relevance to practitioners because researchers have mainly focused inwards on the phenomenon by identifying characteristics of OSS projects and products. Reviewing research on OSSD, Feller at al. (2006, p. 274) concludes that research on OSSD "requires greater discipline and rigor – deeper research, more quantitative data, and more robust cross case-analysis". Retaining the view that current dominance of

proprietary, closed source software will come to an end, Fitzgerald (2006) argues that it is OSS, not OSSD, that will revolutionize the software industry.

This section elaborates Fitzgerald and Kenny's (2004) argument. It does so by relating the problem of relevance to the empirical agenda in software engineering. Chapter 2 showed that the main motivation of the empirical agenda is to make research more relevant to practice through increased credibility. Credibility, in turn, is to come as a product of applying scientific rigorous research methods. With basis in on our review of software engineering literature on OSSD (Østerlie and Jaccheri 2007a), this section pursues the argument that scientific rigour may in fact have made OSSD less relevant for the broader field of software engineering. As such, the argument offered here questions Feller et al.'s (2006) call for increased scientific rigour and more quantitative data as a solution to the problem of relevance.

4.3.1 The rigorous development irrelevance

Despite recent studies showing greater diversity among OSSD projects (Michlmayr et al. 2005), our analysis of the software engineering literature showed that the literature continues to describe OSSD as a homogenous phenomenon (Østerlie and Jaccheri 2007a). While such lack of precision is to be expected in early phases of exploring a novel phenomenon, we would have expected a more nuanced view after a decade of research. In our analysis, we therefore asked: under what conditions such an unbalanced view of OSSD could be maintained over time?

Through our discourse analysis of the literature, we found that such a lack of nuances stem from three sources: strategies for describing OSSD as a homogenous phenomenon, the use of predominantly quantitative research methods, and a lack of diversity in the cases studied. Contextualizing these three issues in the software engineering discipline itself, our analysis showed that they had basis in two commonly held assumptions about the software engineering discipline:

- Assumptions about the identity of software engineering
- Assumptions about how to do software engineering research

As discussed in Chapter 2, the software engineering discipline's identity is closely intertwined with the software crisis. Professionalizing software development is the software engineering discipline's response to the crisis. Software development by volunteers could be regarded as a threat to this very identity. As such, framing OSSD as completely different from software engineering reduces the applicability of OSSD outside the specific context where there is a convergence between users and developers. In so doing, we argue, the challenge posed by OSSD to the software engineering identity is neutralized. However, in the process, we find OSSD homogenized to software development by users for users.

In terms of Schein's three-component model of professional knowledge (Table 2-1S), the purpose of software engineering research is to develop prescriptive models for managing the software development process. Compared to such prescriptive models, OSSD practice comes across as completely different to software engineering. On the

other hand, what exists of research on software engineering practice also shows that these prescriptive models of software development do not reflect software engineering practice very well (Robinson et al. 2007). However, rather than questioning the assumption that OSSD is completely different from software engineering, Sub-section 4.2.2 shows how scientific rigorous methods for quantifying OSS products and process have been applied to quantify such differences. As such, through its application of scientific rigorous research, existing studies simply fail to address the unquestioned assumption of OSSD as completely different from OSSD.

Our conclusion was that existing studies of OSSD keep situating the phenomenon in an otherness relation to software engineering. This, it will be argued in the next subsection, makes OSSD largely irrelevant to the software engineering field.

4.3.2 Otherness and irrelevance

While existing software engineering research on OSSD tends to focus on the unique characteristics of software development in geographically distributed communities of volunteers, it has been observed that OSSD "is not software engineering done poorly … [it] is different" (Scacchi 2007, p. 459). This is not entirely unproblematic. Messerschmidt (2004), for instance, argues that since OSSD is completely different to software engineering, OSSD research only applies to the context of volunteer software development. The contribution of OSSD research to software engineering is therefore somewhat unclear.

Golden-Biddle and Locke (1993) reflect upon the construction of convincing contributions in reported research. They identify three factors that make a contribution convincing. One of these factors is plausibility. Plausibility balances two concerns. A research contribution has to be distinctly different from existing research, on the one hand. On the other hand, the contribution needs to establish a connection with common concerns within the discipline. Failing to establish such a connection to common concerns, the contribution will come to be regarded as irrelevant.

What is the connection between the concerns of the software engineering discipline if OSSD is completely different? While it interesting as a piece of curiosa, without such a connection OSSD studies will remain largely irrelevant to the software engineering practitioners. There is therefore no plausible contribution of OSSD.

4.3.3 Beyond the otherness relation

The title of the research project reported in this thesis is 'Empirical software engineering and open source software development'. The problem of relevance to the broader field of software engineering has therefore been central to the research reported in this thesis. The last subsection in this part of the thesis dedicated related work will draw some implications of the problem of relevance to the role OSSD plays in this research. As such, these reflections serve as a bridge to the next part of the thesis that will report on the research.

The reported research started out on the assumption that OSSD is completely different from software engineering. For me, this was motivated by an anti-bureaucratic agenda. Over time, I had come to see software engineering as a movement towards bureaucratizing software development, and thereby stifling human potential (Mumford 2006). From such a view, I found the collaboration between users and developers as a way of realizing human potential in software development. While I still retain such humanistic ideals, this chapter shows that I have come to question my initial assumptions about OSSD.

Over time, however, I have become increasingly concerned with the lack of relevance of OSSD research in software engineering. Like Samoladas et al. (2004) I have therefore come to regard OSSD as a form of software maintenance based on a perpetual cycle of corrective and adaptive maintenance. Rather than emphasising possible differences between OSSD and software engineering, I have sought to bridge the gap between OSSD and the software engineering literature by treating software integration in a distributed community of volunteers merely as the research setting for exploring the practice of maintaining an integrated system. In the reported research, OSSD is therefore studied as an instance of software maintenance.

PART II: THE RESEARCH

5. The interpretive research approach

This thesis builds upon and supplements the small, but growing body of research on software engineering practice. Various aspects of software engineering practice have been studied. These range from software design (Button & Sharock 1996; Walz et al. 1993; Curtis et al. 1988), configuration management (Grinter 1999), rapid application development with prototyping (Beynon-Davies et al. 1999), software maintenance (Sim & Holt 1999; Sim & Holt 1998), to broader studies of work practices of software engineers (Singer et al. 1997; Low et al. 1996) along with the culture and community of software engineering (Sharp & Robinson 2004; Sharp and Robinson 2002; Sharp et al. 2000; Sharp et al. 1999).

To promote awareness of practice studies, researchers have focused upon clarifying methods for collecting and analysing data of software engineering practice (Lethbridge et al. 2005). While the mechanics of method is important, exclusive emphasis on method obscures a more fundamental shift of focus in practice studies. Rather than focusing on software engineering methods, tools and techniques independent of their context of use, these studies seeks to understand how methods, tools, and techniques are used in the broader social context of the development organization (Robinson et al. 2007). Studying practice therefore requires a research approach to address its inherent social nature. To this end, the reported research draws upon interpretive research. There is currently little interpretive research within software engineering. This chapter therefore draws upon literature from information systems (IS) research in its presentation of the interpretive research approach.

Interpretive research builds upon a set of assumptions about reality and knowledge that "emphasizes the importance of subjective meanings and social-political as well as symbolic action in the processes through which humans construct and reconstruct their reality" (Orlikowski and Baroudi 1991, p. 13). Section 5.1 discusses the assumptions. These assumptions have implications for the research methods considered appropriate when doing interpretive research. This is discussed in Section 5.2. Analysis interpretive research data is discussed in Section 5.3. Section 5.4 discusses the kinds of contributions interpretive research may offer, before Section 5.5 rounds off the chapter with by discussing how to evaluate interpretive research.

5.1. Assumptions about social reality and knowledge

It is common within IS research to distinguish between three broad groups of research approaches: positivism, interpretivism, and critical research (Orlikowski and Baroudi 1991). Table 5-1 briefly outlines and contrasts the three approaches' assumptions about social reality and knowledge.

Approach	Assumptions of social reality	Assumptions about knowledge
Positivism	An objective physical and social world exist independent of humans	Knowledge consists of facts that need to be verified or falsified
Interpretivism	Social reality is produced and reinforced by humans through their action and interaction	Knowledge of reality is gained through language and action
Critical research	Social reality is historically constituted	Knowledge is grounded in social and historical practices

Table 5-1 Comparison of research approaches

Positivism emerges from the natural sciences. Interpretivism and critical research emerges from the humanities. Positivist research is concerned with simplicity, testability, and hypotheses. Seeking towards the natural sciences for its general principles (component 1 in Table 2-1), software engineering research is practically exclusively positivist. Sjøberg et al.'s (2008) guidelines for evaluating software engineering theories is a good example. They emphasise the importance of testing theories empirically, and evaluating them in terms of simplicity. Simplicity is both related to parsimony (i.e. simplicity in the number of concepts used), as well as generality and explanatory power (i.e. how much simple constructs can explain).

Interpretive research, on the other hand, draws upon phenomenology (Boland 1985) and hermeneutics (Lee 1994) for its underlying principles. Unlike positivism, interpretivism seeks to address the complexities of social reality, emphasising nuances over simplicity. It does so through human interpretations and meaning, seeking to understand how language and action brings particular social realities into being. From this follows the assumption that our knowledge of reality is also gained through language, consciousness, shared meanings, documents and other artefacts (Klein and Myers 1999). Interpretive research therefore seeks to understand phenomena through the meanings that people assign to them in order to explain why they do what they do. Rather than seeking to grasp an external reality, then, the interpretive researcher seeks to "understand *intersubjective meanings embedded in social life*" (Gibons 1987, quoted in Orlikowski and Baroudi 1991, p. 13, my emphasis).

Reality is therefore not a fixed object to be grasped independent of the social actors. Rather, social actors construct reality as they go about making sense of an ongoing flow of experience (Schütz 1967). This includes the researcher, who is seen as yet another social actor. The interpretive researcher does not seek to uncover de-contextualized facts or laws. Rather, human interpretation and meaning is seen as a product of the broader context from which they emerge (Walsham 1995). Yet, there is a continuous *codetermination* between people and the broader context they are part of. People

actively shape the context they are part of, but are in turn shaped by the context (Weick 1995).

5.2. Interpretive fieldwork

As people actively construct social reality as they go about making sense of the ongoing flow of experience, the interpretive researcher seeks the insider's view of the phenomenon under study; seeking to uncover how the research subjects themselves bring the phenomenon into being through their actions and interactions. As such, the researcher approaches the field open minded and with a non-judgmental attitude towards the research subjects' activities (Robinson et al. 2007).

Seeking ways to study phenomena as they unfold, interpretive research regards the research process as emergent. It is also emergent as it seeks to take into account the researcher's deepening understanding of the phenomena. Focusing on the intersubjective meanings embedded in social life, interpretive research therefore aims at developing a rich understanding of the research subjects' world-building activities (Walsham 1995). This requires the interpretive researcher to seek out and get close to the everyday activities of the people under study, placing himself in the midst of other people's lives and to observe them.

A common argument pursued in the software engineering literature is that fieldwork should only be undertaken when it is impossible to control the variables (Tichy 1998). In contrast, fieldwork is the interpretive researcher's preferred method. Whereas natural scientific research seeks an objective distance to the object of study, the quality of interpretive research seeks immersion with the research subjects. It is by engaging with people that the researcher understands how they make sense of and give meaning to their experiences. Fieldwork enables just this form of *immersion* in the lives of other people.

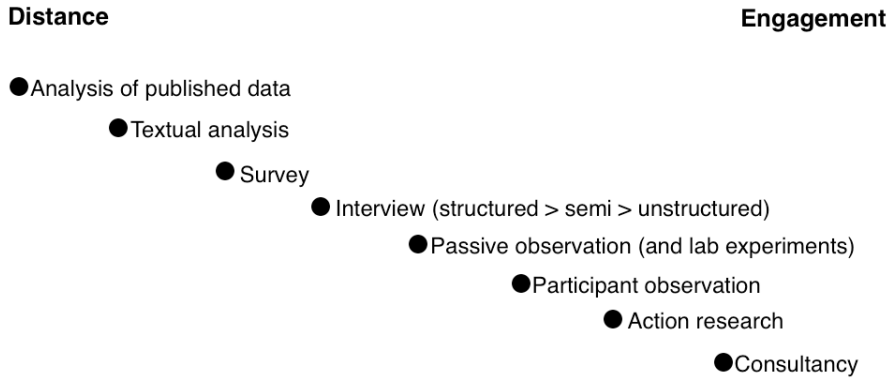


Figure 5-1 Implications of data collection methods on researcher's engagement with the field (Nandhakumar and Jones 1997)

Immersion has practical implications for the data collection methods considered appropriate during interpretive fieldwork. Nandhakumar and Jones (1997) provide an overview of the degree of immersion provided by different data collection methods. At the remote end of immersion is analysis of published data, while consultancy is the extreme variant of immersion. The more the researcher is engaged with the field the more flexible the fieldwork is to the researcher's unfolding understanding of the phenomenon under study (Lethbridge et al. 2005).

Seeking immersion, interpretive research often takes the form of in-depth case study or ethnography (Walsham 1995). There is a sliding boundary between the in-depth case study and the ethnography. Both are predominantly based upon data collection methods ranging from interview to action research. Ethnographic research relies more upon participant observation, and the in-depth case study is more interview-based.

The materials collected during interpretive fieldwork may include interviews, observations, and documents. The latter may include both documents internal to the organization, as well as press, media, and other publications (Walsham 2006). The kind of data collected is typically process data (Langley 1999). Process data deal with events, activities, and the sequence of these. Despite this primary focus on events, process data tend to be eclectic and may include both qualitative and quantitative data. This means that while interpretive research is predominantly qualitative, interpretive research is not necessarily restricted to qualitative data.

5.3. Analysis

Analysis is an ongoing process from the moment the interpretive researcher enters the field until the complete research report is written. It is an ongoing process of making sense of the fieldwork experience and the collected data. It is based on the view that "we come to understand a complex whole from preconceptions about the meanings of its parts and their interrelationship" (Klein and Myers 1999, p. 71).The analysis therefore emerges out of the broader context the researcher is part of, and both shapes and is shaped by this context. It is therefore common to talk of three dimensions of codetermination during analysis:

- As an interaction between particular observations and their appropriate historical or political context
- As an interaction between observations and theory
- As an interaction between the researcher and the research subjects

Since analysis is continuous throughout the research process, it useful to differentiate between informal and formal analysis. During fieldwork, the researcher engages in informal analysis to understand the world-building activities of the research subjects. This provides the researcher a better sense of the fieldwork experience. During periods of participant observation, for instance, informal analysis may take the form of writing out the notes that have been quickly and briefly jotted down in the notepad during the day's observation, and organizing them into more coherent field notes (Emerson et al. 1995). By relating the day's observations to previous field notes, the researcher looks

for patterns in his observations for building informal theories. These informal theories, in turn, inform how the researcher continues to perform the fieldwork. This way, the researcher can adjust the fieldwork with basis in an increased understanding of the research setting.

Formal analysis, on the other hand, usually commences upon withdrawing from the field. Whereas informal analysis is directly related to the fieldwork experience, formal analysis is related to the textual data collected during field. Seaman (1999), for instance, offers a method for formal data analysis based on coding. More generally, though, formal data analysis is a process of systematically going through the collected data, looking for recurring patterns, and incrementally generalizing from a multitude of singular observations to increasingly more generalized descriptions of activities (Fetterman 1998). Throughout this process, non-recurring details of the singular observations are omitted and recurring issues included. However, determining which details to omit in the final analysis and which to include is an iterative process of working on generalizing the descriptions while continuously returning to the more detailed analyses looking for recurring patterns that may shed light on the generalized description.

The collected data is interpreted through theory. Theory acts as a lens to bring out particular aspects of the data. Particular observations are related to generalized theoretical concepts. Lee and Baskerville (2003) calls this theoretical generalization. As there is an interaction between data and theory this theoretical lens is likely to change during the research process (Klein and Myers 1999). The theoretical framework used for planning the research is therefore likely to be different than the theory used when reporting at the end of the research process.

5.4. Research contributions from interpretive research

Interpretive research contributes with *in-depth understanding* of phenomena. This is commonly reported in the form of thick descriptions. The thick description embraces the assumption that social reality is a "multiplicity of complex conceptual structures, many of them superimposed upon or knotted into one another" (Walsham 1995, p. 71).It is anti-reductionistic, seeking to strike the balance between conveying the complexity of human actions on the one hand, and appropriate simplification to render the complexity intelligible to the reader on the other hand.

Interpretive research may offer five types of contributions, summarized in Table 5-2 below. These types of contributions differ somewhat from contributions often associated with software engineering research – improved tools, techniques and methods (Basili 1993) – aimed at contributing towards the applied knowledge component of professional knowledge (Table 2-1). The first four types of contributions in the table below contribute towards theory. Such contributions are not directly aimed at informing practice, although Robinson et al. (2007) points out that rich descriptions may indeed help inform practice. However, specific implications for particular domains

of action may be drawn from interpretive research. Such implications are less of prescriptive than methods and techniques, emphasising tendencies rather than prediction (Walsham 1995).

Type of contribution	Example
Development of concepts	Ciborra and Lanzara (1994) develops the concept 'formative context' to explain the dynamics of innovation in organizations.
Generation of theory	Orlikowski (1996) develops a theory for a situated change perspective on organizational change.
Challenge perceived views	Bansler and Bødker (1993) reveal that there is a gap between the procedures prescribed by structured analysis and the way in which it is carried out.
Contributions of rich insight	Through an ethnographic study of eXtreme Programming (XP) practice, Sharp and Robinson (2004) offers rich insight into how an organization implements XP for software development.
Implications for particular domains of action	With basis in a study of IS development in the finance sector, Walsham and Waema (1994) draw a number of implications for the relationship between design and business strategy.

Table 5-2 Types of contributions offered by interpretive research

Theory can therefore be both the input to (see 5.3 above) and output of the interpretive research process. Theory developed through interpretive research can be a refinement of that used for planning the research, or it may be a newly formulated theory grounded in the empirical data. The theory developed through interpretive research is what Langley (1999) labels process theory. Process theory seeks to conceptualize events, activities, and choices ordered over time and to detect patterns among them. The purpose is to explain the outcome and mechanics of these activities and events. Process theory therefore encompasses single concepts describing the mechanics of processes. This is in contrast to variance theory that provides explanations of phenomena in terms of dependent and independent variables. It is in this latter meaning of the term software engineering researchers talk about theory (Sjøberg et al. 2008).

5.5. Evaluation of interpretive research

The credibility of reported research results are grounded in a broader understanding of knowledge and social reality (Pozzebon 2004). Research approaches therefore address research evaluation differently. Emphasising how well the reported research represents an objective reality, positivist research is evaluated with the theoretical constructs of validity and reliability. Interpretive research, on the other hand, does not see reality as an object independent of the social actors who try to understand it. Rather, reality is constructed as social actors try to make sense of it. The researcher has no privileged access to reality, and is therefore considered one of these social actors. The credibility of interpretive research is therefore grounded in the researcher's research practice.

Interpretive researchers have approached research evaluation in different ways. Some researchers provide no explicit research evaluation in their reported research, as no universally valid judgements about the credibility of their results can be made. Garrat and Hodkinson (1998), however, argue that even though no pre-specified criteria can ensure universally valid judgements about any kind of research, reflecting upon how the

research may be evaluated can help refine and develop our collective understanding of how interpretive research is to be evaluated. Most interpretive researchers have therefore chosen to employ predefined schemas for evaluating their research results.

A number of IS researchers have employed Golden-Biddle and Locke's (1993) schema for evaluating ethnographic fieldwork. The criteria in this schema ground the credibility of the reported research in the research report itself:

- Authenticity: Was the researcher there?
- Plausibility: Does the story make sense?
- Criticality: Does the research offer something new to the research field?

The most commonly used evaluation schema within interpretive IS research, however, is Klein and Myers' (1999) seven principles for evaluating interpretive field studies. Acknowledging the emergent nature of interpretive research, the principles are not intended as a predetermined set of criteria to be mechanistically applied to the research. Rather, they form a set of guidelines to be applied appropriately with judgement and discretion both in planning and conducting interpretive research, as well as in evaluating the resulting interpretations.

Principle	Description
1. The fundamental principle of the hermeneutic circle	This principle suggests that all human understanding is achieved by iterating between considering the interdependent meaning of parts and the whole that they form.
2. The principle of contextualization	This principle requires critical reflection of the social and historical background of the research setting
3. The principle of interaction between researcher and subjects	Requires critical reflection on how the research materials were socially constructed through the interactions between the researchers and participants.
4. The principle of abstraction and generalization	Intrinsic to interpretive research is the attempt to relate the particulars described in the unique instances observed to abstract categories and concepts that apply to multiple situations.
5. The principle of dialogical reasoning	Requires sensitivity to possible contradictions between the theoretical preconceptions guiding the research and the actual findings.
6. The principle of multiple interpretations	This principle requires the researcher to be sensitive to difference in interpretations among the studied subjects.
7. The principle of suspicion	Requires sensitivities to possible biases and systematic distortions in the narratives collected from the participants.

Table 5-3 Klein and Myers' (1999) seven principles of interpretive fieldwork

Where Klein and Myers' (ibid.) seven principles focuses on evaluating the reported research in terms of the research process, Golden-Biddle and Locke's (1993) offer criteria for evaluating the research in terms of the final reports. Yet, neither of the two is directly concerned with the research results themselves. This is particularly problematic when it comes to how well research results translate outside the particular study. While I will use Klein and Myers' schema for evaluating the reported research, I will supplement this with transferability (Patton 2002) to evaluate the research results. Rather than statistical generalization of quantitative methods or the theoretical generalization of Klein and Myers' principle of abstraction and generalization, transferability builds on the logic of similarity between two contexts.

6. Theoretical framework: Knowledge-intensive work

The reported research studies software maintenance as knowledge-intensive work. It draws upon research that sees work and knowledge as interrelated (Brown and Duguid 1991). This research focuses upon work as it unfolds over time and looks to those working while many of the options and dilemmas remain unresolved. It emphasises how everyday work consists of unexpected twists and turns, and the ‘muddling through’ of practical decision-making and knowing (Orlikowski 2002). As work unfolds within a broader social and organizational context, it is structured by the rich texture of the context's constantly changing conditions. Rather than something to be taken as given, work is a collective achievement that needs to be continuously enacted (Weick 1988).

The purpose of this chapter is to present the theoretical framework used in developing the results reported in this thesis. It combines two theories to study software maintenance as knowledge-intensive work. However, before progressing with a presentation of the two theories, the term 'work' and its use in the reported research needs to be explained.

6.1. Work

Broadly speaking, the term 'work' has four different meanings (Orr 1996):

- Work as profession
- Work as employment
- Work as job description
- Work as action

In the first meaning, work is associated with an individual's profession as in 'software engineer'. In the second meaning, work is associated with the socio-economic relationship between the employer and the employee. In this meaning of the term, work is a transaction between the employer paying the employee's effort to a predefined end. The problem of the above meanings of the term is that they assume work as the activity of production without including the activities essential to production.

The third meaning of the word, work as a job description, overcomes this problem. Here work is a set of activities to be performed by the individual employee. This is often described in manuals, training courses, as well as in formal job descriptions. Here, work is seen as a set of clearly delineated activities. Brown and Duguid (1991, p. 40) calls this view of work 'canonical practice'. They criticise such descriptions for being abstract and detached, and consequently distort or obscure the intricacies of the actual work. In contrast, they propose the term non-canonical practice. Non-canonical practice is the fourth meaning of the term work. Non-canonical practice views work as it unfolds over time, and looks to someone at work on it, while many of the options and dilemmas remain unresolved. Furthermore, it sees work as a performed achievement of a collective of people and technological artefacts (Orr 1996).

It is in the latter meaning of the term 'work' that is used in this thesis.

The distinction between process and practice in software engineering is problematic. Software process is often defined as the sequence of steps performed for developing software (IEEE 1990). As such, software processes deal with prescription and formality (Highsmith 2002), and are described as software process models. Like canonical work, software processes are abstractions detached from the actual flow of work as it unfolds.

Like non-canonical work, research on software engineering practice seeks to study the process of doing tasks and how they are actually structured by the conditions of work and the world. However, the term practice is used with three different meanings within software engineering. The three meanings are summarized in Table 6-1.

Meaning	Description
Software engineering research and practice	In this context, 'practice' means software engineers developing software in contrast to software engineering researchers.
Practice as oppose to process models	"Process deals with prescription and formality, whereas practice deals with all the variations and disorderliness of getting work done." (Highsmith 2002, pp. 121-122)
Best practice	In this context, 'practice' is used to describe the best technique or method for achieving a goal. Software patterns, for instance, is a set of best practices for solving typical design problems object-oriented programming.

Table 6-1 Meanings of 'practice' in software engineering

To avoid confusion, this thesis uses the term 'software engineering practice' in the first meaning in the above table. Focus of the reported research, however, is on practice as opposed to process, the second meaning in the table. This is called 'software engineering work' to avoid confusion. Similarly, the term 'software process' is used in the meaning of an abstraction description of software development practice. The term process is used in a more general term about events, activities, and the sequence of these (Langley 1999).

6.2. Knowledge-intensive work

Much research focuses on the nature of knowledge, seeking distinctions such as tacit and explicit or codified and non-codified knowledge. Such discussions are bracketed in

this thesis. Instead, focus is upon how people express knowledge by acting knowledgeably in practice (Orlikowski 2002). To study software maintenance as knowledge-intensive work, this thesis draws upon a combination of sensemaking theory (Weick 1995) and actor-network theory (Latour 1987). Rather than providing a rigid framework to be applied to the collected data, they have informed the analysis. As such, the two theories have been used as a form of scaffolding to be removed once they are no longer needed to make sense of the collected data (Walsham 1993).

6.2.1 Sensemaking

Sensemaking theory addresses how people make sense of situations that initially makes little sense through action (Weick 1988). The central question driving sensemaking is therefore 'what is going on?' rather than 'what to do next?' (Weick et al. 2005). Action is therefore point of departure for sensemaking.

Action is driven by previous experience and presumptions; it is *retrospective*. To make sense of situations people act upon the ongoing flow of experience. Drawing upon sociological phenomenology (Schütz 1967), the distinction between singular and plural form of experience is important to sensemaking theory. Experience, in the singular form, is the ongoing stream of pure experience of the present moment. To make sense of the ongoing flow of experience we need to act upon it to chunk and classify it into experiences. Experiences, in the plural form, are therefore the product of retrospectively chunking and classifying moments from the ongoing flow of experience. Sensemaking is therefore retrospective action upon past experience (Weick 1995).

While individual action is the point of departure for analyses of sensemaking, sensemaking is still inherently *social*. The term social is understood as the behaviour of two or more actors, and action as the behaviour to which subjective meaning is attached (Schütz 1967, p. xxii). Through our actions, we therefore take part in constructing the materials to make comprehensible situations. However, people are an intrinsic part of this environment. As such, through action sensemaking is social.

This brings us back to the centrality of action as unit of analysis for sensemaking. Action always resides in the past, present, and the future. Action unfolds here and now. But we always act upon past experience. In the process however, we enact sensible environments. In so doing, we enable and constrain future actions (Weick 1995).

While sensemaking theory emphasises the codetermination between human action and the context of human action, it lacks the theoretical apparatus for unpacking and analysing how artefacts in this environment influence human action. Because the centrality of artefacts in the maintenance process, it is important to supplement sensemaking with theoretical concepts for analysing how people go about producing the materials from which to make comprehensible situations. To this end, sensemaking is supplemented by actor-network theory.

6.2.2 Actor-network theory

Originally conceived as a theory for mapping scientific controversies (Latour 1987; Callon 1999), actor-network theory (ANT) has been broadened to the study of technological development including information systems development (Monteiro 2000). The reported research draws on ANT as it offers theoretical concepts for bringing artefacts into the analysis of knowledge creation. It needs to be noted that ANT is not a stable body of theory. Rather, it is continuously revised and extended. The version of ANT used in this research has been labelled the *sociology of translation*.

The sociology of translation is particularly concerned with the development of scientific knowledge. Emerging from the sociology of knowledge (Bloor 1976), ANT refutes that scientific knowledge is the product of a privileged scientific method. Instead, *knowledge is the product of a patterned network of materially heterogeneous actors* (Law 1992). This is a rather convoluted explanation of knowledge creation. The purpose of this subsection is therefore to progressively clarify this explanation by presenting the key terms of the sociology of translation.

The *actor* is the basic concept of ANT. Actors may be human. The human actor is similar to what we call a *stakeholder* in software engineering. For ease of terminology, the term stakeholder is therefore used about human actors in this thesis. However, ANT does not limit the term actor to humans. Instead, the term is used about all physical entities involved in the knowledge-creation process. For instance, Callon (1999) treats scallops as actors in analysing how scientific knowledge of scallop farming is developed. Similarly, Latour and Woolgar (1979) brings laboratory scientists and technicians, laboratory equipment such as test tubes, reagents, and microscopes, as well as the scientific papers, patents, and conferences reporting scientific knowledge as actors in the analysis of how knowledge is created in laboratory sciences.

Rather than privileging humans, ANT therefore treat all actors involved as equally important in the analysis of knowledge creation. The term 'materially heterogeneous actors' derives from this inclusive view of the actor. This inclusive view of actors is labelled *the generalized principle of symmetry (Latour 1987)*. Like sensemaking, ANT sees knowledge as essentially social. With the generalized principle of symmetry, however, the scope of the social is expanded to include all actors: humans as well as non-humans. As many see social as exclusive to humans, it is useful to use the term 'collective' instead.

Returning to the definition of work as a collective achievement ,we see that it is the achievement of human as well as non-human actors. The analysis of software maintenance work therefore needs to include both stakeholders like users and software developers, in addition to product artefacts like source code and executing software, process artefacts like problem reports, as well as the tools used in developing and maintaining the software. They are all brought into the analysis of software maintenance work as actors.

ANT was conceived to analyse scientific controversies and the mechanisms of resolving such controversies. To this end, ANT offers the term *translation*. Translation is the

mechanism through which different actors with different interests come to reach an agreement. This is often referred to as translation of interests. For anyone wanting to establish a fact, the basic constituent of knowledge, the first moment in the translation of interests is to define actors, endowing them with interests and problems to overcome (Callon 1999). The purpose of defining and endowing actors with such interests is to establish the fact as a solution to problems faced by the actors. By accepting the fact as true, the actors will meet their interests. As such, the defined actors' interests are translated or aligned with the translating actors' interest. Put another way: actors are patterned. Facts and knowledge come about as the product of such patterned networks of actors.

Sensemaking is clear on how action folds the past, present, and future in one. The same happens when translating interests, too. ANT, however, is not particularly clear on this. Yet, defining actors, endowing them with interests and problems to overcome is both retrospective and enactive. On the one hand the particular pattern of actors do not exist a priori. Rather, they are retrospectively constructed in the moment the moment of translation. However, bringing a set of actors into being, a particular form of reality is constructed. As such, translation is also enactive. While sensemaking emphasises the collective aspect of knowledge and understanding, it is weak on theorizing conflict. This is where the vocabulary of ANT comes to the rescue.

It is worth noting that the interests endowed to actors do not exist *a priori* (Callon and Law 1982). Rather, the translating actor endows other actors with interests. However, these actors are not necessarily docile bodies in the hands of the translating actor. On the contrary, translation is a multilateral process of negotiation. The translation is in the hands of the other, the actors being translated (Latour 1987). As such, "[t]o describe enrolment is thus to describe the group of multilateral negotiations, trials of strength and tricks that accompany the interessement and enable them to succeed" (Callon 1999, p. 74).

Through mobilization, the actor-network is kept stable by making "a configuration of a maximal number of allies act as a single whole in place" (Latour 1987, p. 172). This renders the individual actors in the network invisible making them appear as a single unity, a process called black boxing. This way we see how actors are hybrids, collectives that take the form of "companies, associations between humans and associations between non-humans" (Callon 1991, p. 140). This also points to the analytical flexibility of ANT, entailing "that the 'actor' of an analysis is of the 'size' that the researcher chooses as most convenient relative to the direction of the analysis" (Monteiro 2000, p. 82).

Translation is therefore the process of enrolling a sufficient body of actors by aligning these actors' interests so that they are willing to participate in particular ways of acting. It implies definition, and this definition is inscribed in material intermediaries. These intermediaries are actors in their own right. They are delegates who stand in for and speak for particular interests; they are the medium in which interests are inscribed. The operation or translation is therefore triangular: it involves a translating actor, actors that are translated, and a medium in which the translation is inscribed.

7. Research setting: Gentoo

The setting for the reported research is the Gentoo community. This is a geographically distributed community of volunteer software integrators. They operate and maintain a system for distributing and integrating third-party OSS on different Unix operating systems. The purpose of this section is to provide an overview of the Gentoo community, its technology, and their work activities. To this end, the chapter is organized in three sections First, a brief overview of the Gentoo community is provided in Section 7.1. An overview of the Gentoo technology is provided in 7.2. The chapter is concluded with an overview of the formal organization of work within the Gentoo community in Section 7.3.

7.1. The Gentoo community

Gentoo started as a one-man effort to create a highly configurable GNU/Linux distribution, Gentoo Linux, in 2000. Over time the community and its technology has evolved. By 2006, the effort has grown into a community of geographically distributed volunteers across the world. By now, the software distribution system originally at the heart of Gentoo Linux has become the core product of the community. The software distribution system supports five different Unix operating systems in addition to GNU/Linux.

Of over 100 commercial and non-commercial GNU/Linux distributions, DistroWatch (http://www.distrowatch.com) lists Gentoo Linux among the top 10 most widely used. Debian GNU/Linux , another volunteer-based GNU/Linux distribution, is Gentoo Linux' strongest competitor. Emphasising stability of use, Debian has had problems keeping up to date with the latest updates of the software they offer. Gentoo Linux, on the other hand, has had problems moving from an expert/developer's distribution, to a more easily managed distribution.

As of March 30 2006[1], the Gentoo community consists of 320 developers. Being a Gentoo developer is a formal title within the Gentoo community, indicating that the

[1] This is the date I formally concluded the fieldwork. No new data have been collected since.

person has been formally adopted with development privileges to the community. The developers are geographically distributed across 38 countries. The distribution of Gentoo developers is summarized in Table 7-1 below.

Continent	Country	Developers by country	Developers by continent
Africa			1
	South Africa	1	
Asia			18
	China	1	
	Israel	2	
	Japan	8	
	New Zealand	3	
	Singapore	2	
	Taiwan	1	
	Vietnam	1	
Australia			3
Europe			132
	Austria	7	
	Belgium	9	
	Cyprus	1	
	Denmark	5	
	Finland	1	
	France	6	
	Germany	32	
	Hungary	1	
	Iceland	1	
	Ireland	1	
	Italy	11	
	Norway	1	
	Poland	5	
	Portugal	2	
	Romania	3	
	Russia	1	
	Slovakia	1	
	Spain	3	
	Sweden	3	
	Switzerland	5	
	The Netherlands	6	
	United Kingdoms	26	
North America			138
	Canada	8	
	USA	130	
South America			10
	Argentina	1	
	Brazil	4	
	Chile	1	
	Colombia	1	
	Venezuela	3	
Unregistered			18
Total	38 countries	320 developers	

Table 7-1 Distribution of Gentoo developers

A majority of the developers are located in Europe and North America with respectively 138 and 132 developers, for a total of 84% of the Gentoo developers. 130 of these developers are located in the USA alone, giving the country the highest population of

Gentoo developers. Second and third largest populations are found in Germany and the United Kingdoms with 32 and 26 developers respectively. 34 countries are represented by 10 developers or less. Only Germany, Italy, United Kingdoms, and USA are represented by more than 10 developers. To the best of my knowledge, no two Gentoo developers are geographically co-located.

The geographic distribution of the Gentoo developers means that they are spread across 17 time zones, from Pacific Standard Time in Western USA (UTC -8) to New Zealand Mean Time (UTC +12). Two ranges of time zones are not represented: the three pacific time zones between Western USA and New Zealand, and the four central Asian time zones between Moscow Time (UTC +3) and Western Standard Time (UTC +8). The majority of developers are found in two time zone ranges. 43% of the developers live in the four North American time zones from Pacific Standard Time (UTC -8) in the West to Eastern Standard Time (UTC -5) on the East coast. 41% of the developers live in the three European time zones from Greenwich Mean Time (UTC ±0) in the West to Moscow Time (UTC +3) in the East. The largest time zone difference is 12 time zones. This is the distance between the Russian developer the developers on the West coast of the USA.

To overcome the geographical and temporal distance, the Gentoo community has a number of Internet Relay Chat (IRC) channels, mailing lists, and Web-based forums for communication. IRC is the primary mode of communication, with the mailing lists providing a supportive role. The number of participants on the IRC channels and mailing lists far exceeds the number of Gentoo developers. In addition to the Gentoo developers, a lot of volunteers contribute to Gentoo with source code as well as problem reports. It is difficult to ascertain the number of such volunteers, as they come and go. However, it is important to note that such volunteer contributions of source code and problem reports are important to the Gentoo community.

7.2. The Gentoo technology

The Gentoo community develops and maintains a software distribution system for distributing and integrating third-party OSS with Unix-like operating systems. This system is released as OSS. A *Unix-like* operating system is an operating system that behaves in a manner similar to a Unix system, but does not necessarily conform to POSIX. Gentoo supports the following operating systems: GNU/Linux, FreeBSD, OpenBSD, NetBSD, Mac OS X, and Dragonfly. Gentoo's software distribution system is made up of two parts: an Internet-based infrastructure for distributing third-party OSS, as well as the Portage package manager for integrating third-party OSS on individual computers. The Gentoo community provides its own GNU/Linux distribution, *Gentoo Linux*, based upon the technology above. A *GNU/Linux distribution* is a collection of software applications and libraries bundled together with the Linux operating system kernel. It is called a GNU/Linux distribution as much of the core software is developed by the GNU project.

Before progressing with a more in-depth presentation of the Gentoo technology, it is useful with an overview of the Unix system architecture.

7.2.1 The Unix system architecture

Unix is a time-sharing operating system developed by AT&T's Bell Laboratories during the late 1960s and early 1970s (Ritchie 1984). It became a popular operating system at universities during the 1970s. By the early 1980s the Advanced Research Projects Agency by the US Defence Department had chosen Unix as the standard operating system for its Internet node. It is common to present the architecture of an executing Unix system as a four layered model (Tanenbaum 1992), as shown in the figure below.

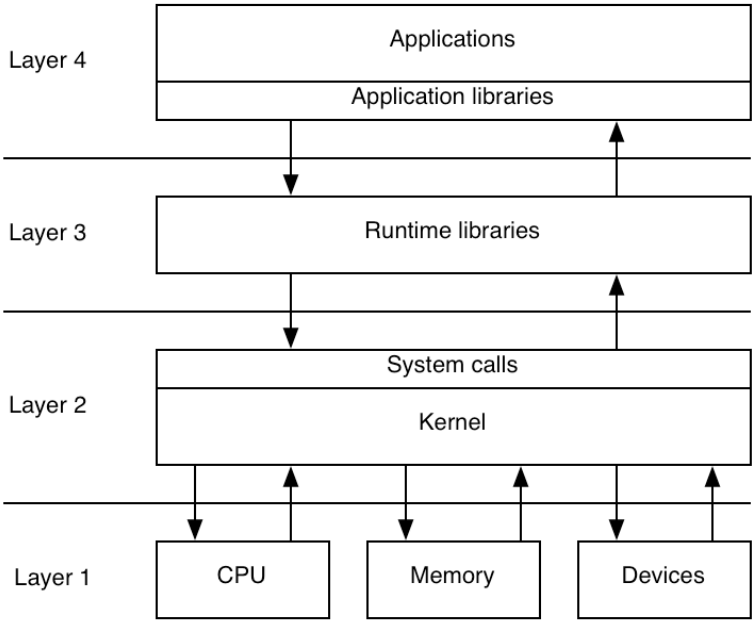


Table 7-2 The Unix system architecture

At the bottom of the figure is Layer 1, the hardware layer. On top of the hardware layer is the operating system kernel, or simply the *kernel*. The kernel manages the system resources, and communicates between the software and the hardware. Linux is an example of an Unix-like kernel. The different BSD kernels are another example. *System calls* provide an interface for software to request services from the kernel. The system calls are part of the kernel. Most Unix-like operating systems provide slightly different system calls.

Layer 3 contains *runtime libraries*. The runtime libraries offer an abstraction layer between the application software in layer 4 and the operating system. These are usually an implementation of the C library, such as the `glibc` implementation used by most GNU/Linux distributions. Runtime libraries handle the low-level details of passing information between the kernel and the application software layer. They are therefore operating system dependent.

At the topmost layer is the *application software*. This consists of application libraries as well as applications. *Application libraries* offer a collection of subroutines that multiple applications use. Unix-like operating systems have a great number of application libraries such as `Qt` a library offering functions for graphical user interfaces. The part of Unix-like operating systems that users typically relate to, are the applications. These follow to the definition of applications presented earlier in this thesis.

7.2.2 The Portage package manager

Portage is the Gentoo package manager. A *package manager* is an application that integrates software with a local computer's file system. There are many different package managers. RedHat Linux, for instance, uses a package manager called rpm. The BSD operating systems, on the other hand, use a package manager called ports. A *Gentoo system* is a computer using Portage to integrated third-party OSS with its local file system. An overview of Portage's architecture is provided in Figure 7-1 below.

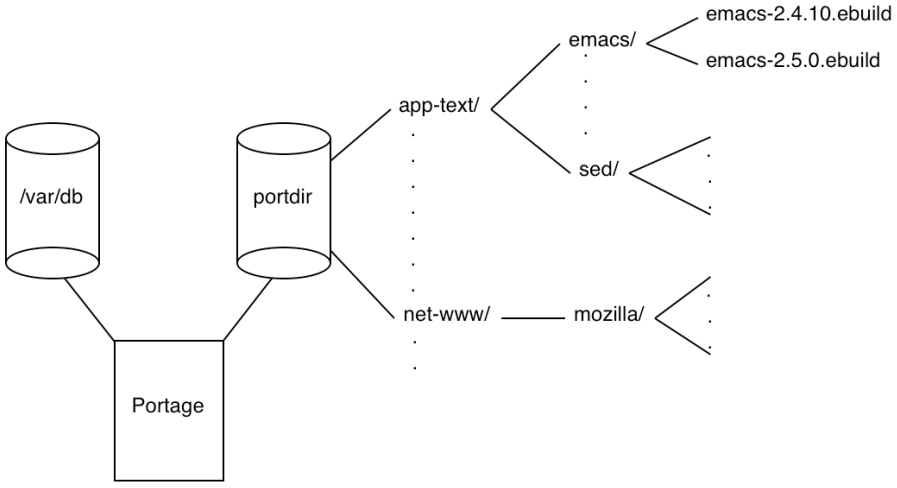


Figure 7-1 Overview of Portage's architecture

Portage is a software application executing on a local computer. It integrates software packages with the computer's file system. In the GNU/Linux terminology, a *software package*, or simply package, is third-party software that can be integrated with the local computer's file system. While many package managers integrate pre-compiled packages, Portage compiles the software locally before integrating it with the file system.

Portage uses an installation script, an *ebuild*, to integrate packages. There is one ebuild for every package supported by Gentoo. Gentoo often offers multiple versions of a package. For instance, both versions 1.5.7 and 1.6.0 of the Mozilla Web browser is supported. Multiple versions are often supported because the most recent version is considered experimental and unstable, or there may be compatibility issues with other

packages. Each version has a separate ebuild. There are ebuilds for software at layers 2 to 4 of the Unix system architecture.

All available ebuilds are stored in a local database. This database is called *portdir*. This is the variable name of the database used in the Portage source code. portdir organizes packages in *categories*. The categories are organized according to functionality. An example of such a category is app-text, which contain text-processing software packages like the Emacs text editor. Another example is net-www, which contain web browsers like Mozilla and Firefox. The portdir database is implemented as a simple directory hierarchy. Each category is a directory, and the ebuilds for a single package is organized in a sub-directory of its category. In addition to containing ebuilds, the package directories contain addition files like configuration files, auxiliary scripts, for instance.

As of March 2006, Gentoo supported 8486 packages, for a total of 23911 ebuilds. The total SLOC of ebuilds in the repository is 671971. The installation scripts make up approximately 90% of the source code in the repository. The rest of the source code in the central repository is patches and configuration scripts to be used when integrated the software package on a local computer.

The process of integrating a package with Portage follows the following process:

1. Calculating dependencies to other packages. If the package to be integrated depends on other packages, this process is repeated for each package until all required packages have been integrated with the local system
2. Download source code of the package
3. Unpack the source code in a protected sandbox
4. Configure source code
5. Compile source code
6. Integrate binaries and documentation with local file system
7. Update the Portage database to store information about the newly integrated package

Information about the software that Portage has integrated with the local file system is stored in another database, labelled by its file system path: /var/db/. /var/db/ is implemented as a simple directory hierarchy, with an identical structure to portdir. However, some addition information about the location of the package's files on the local file system is stored in this database.

7.2.3 The Gentoo software distribution infrastructure

The Gentoo software distribution system is based around a central repository of ebuilds. The portdir database stored locally on individual Gentoo systems is merely a replication of the central repository. Upon request, Portage synchronizes its local portdir over the Internet from the central repository, downloading new ebuilds and deleting ebuilds no longer supported by Gentoo.

The Gentoo developers are responsible for developing and maintaining the ebuilds in the central repository. To keep track of changes made to each ebuild, the repository and the ebuilds contained within it is under revision control using the CVS configuration management system. This leads to a two-layered versioning scheme, where individual ebuilds have revision numbers from CVS. These, though, are independent of the package's version number. The package's version number is part of the ebuild's file name (see Figure 7-1 above). Changes made to individual ebuilds are checked into the CVS repository. Once every hour, the central repository is updated with the latest updates from CVS.

The schematic outline of the distribution system's infrastructure is outlined in Figure 7-2 below.

7.2.4 Variability and Gentoo

As shown in Figure 7-2, there is no single Gentoo system. Rather, numerous instances run on computers distributed across the Internet. Distribution is a defining characteristic of integrated sytems (Hasselbring 2000). Distribution is the core factor for the potentially for immense variability among Gentoo systems. The individual state of Gentoo systems vary greatly. The dimensions of this variability will be discussed here.

Heterogeneity is another defining characteristic of integrated systems (Hasselbring 2000). The heterogeneity among the operating systems supported by Gentoo contributes to the variability among individual Gentoo systems. This variability can partly be explained with basis in the Unix systems architecture. Gentoo supports five different operating systems. Each has its own kernel, with slightly different system calls, and different runtime libraries. GNU/Linux, for instance, supports threading in the `libthread` runtime library. The BSD operating systems, however, support threading in the `libc_r` runtime library. Furthermore, the kernels run on different hardware platforms. MacOS X runs on the Apple PowerPC computer architecture. The Linux kernel is also supported for five different processor architectures.

Autonomy is another source of variability among Gentoo systems. Some operating systems are more autonomous of Portage than others. Portage controls all software at layers 2 to 4 for Gentoo Linux. Apples's own package management software, on the other hand, controls much of the software on MacOS X. Here Portage co-exists with Apple's package management software. Apple's package manager is in complete control of the software in Levels 2 and 3. It is also in control of many application libraries in Layer 4. Portage has to take this into account when calculating dependencies.

Individual Gentoo systems can be autonomously configured using *optional features* and *virtual packages*. Portage may be configured to support optional features, simply called optionals, across individual ebuilds. The IMAP mail protocol is an example of such a crosscutting feature. If IMAP is registered as an optional in the Portage configuration file, every ebuild having supporting IMAP will be compiled with IMAP support.

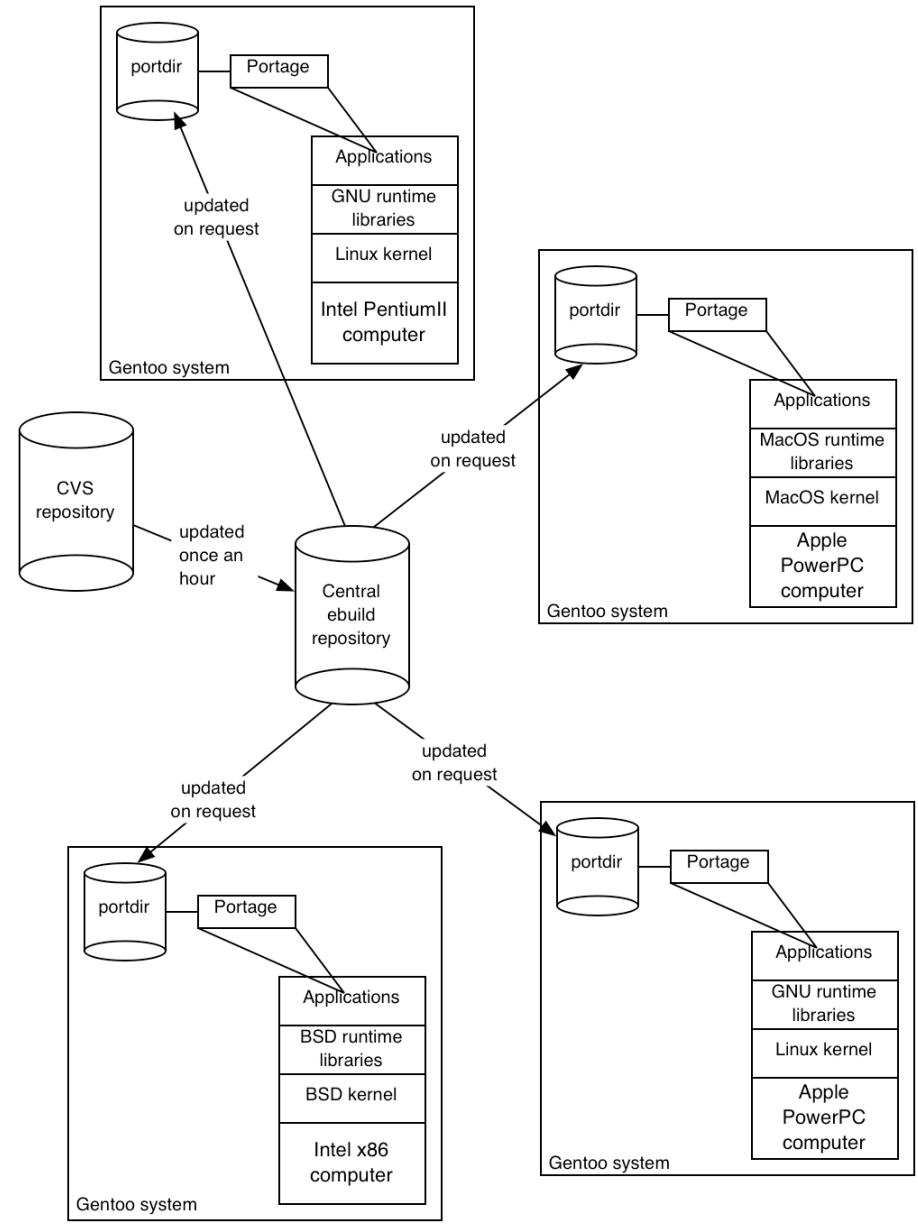


Figure 7-2 Distribution system's infrastructure

Similarly, functionality that may be provided by different packages are called *virtual packages*. For instance, Java applications only rely upon having a Java virtual machine installed on the local computer. It does not matter whether this is Sun's or IBM's Java machine. Similarly, many GUI applications rely functionality provided by either the GTK1 or GTK2 widget library.

7.3. Organization of the maintenance process

7.3.1 Community organization

Having experienced an exponential growth from a one-man project in 2000, the Gentoo community adopted a formal management structure in July 2003. The stated intent of the management structure was to resolve the chronic management, communication, and coordination issues the community was suffering under. The community was to be headed by a *management team*. The management team initially consisted of the Chief Architect along with the managers of the Gentoo projects. A project is a group of Gentoo developers formed for handling a particular general area. The Portage project, for instance, is devoted to maintaining and updating Portages core functionality and utilities. There are 32 such projects as of March 30 2006.

A *herd* is a team of Gentoo developers responsible for a collection of packages. As of March 30 2006, there were 124 such herds. Each herd vary in size from a single developer, to over 20 developers. The purpose of the herds is to ensure that there is always somebody responsible for resolving incoming problem reports. Sometimes herds and projects overlap. For example the Portage project has an associated Portage herd.

7.3.2 The maintenance process

Respective projects or herds handle maintenance modification decisions informally. The Portage project is somewhat unique, as it is responsible for maintaining and application and not ebuilds. This project therefore has a more formal decision process, where the project manager makes the decisions about modifications to the software.

The corrective maintenance process, however, is organized more formally. Upon experiencing software failures, the user submits a problem report to the Gentoo defect tracking system. The Gentoo community uses Bugzilla, an open source defect tracking system (Barnson 2007). The problem report in Bugzilla consists of a set of predefined fields for classifying software failures. It also provides a text field called 'Additional comments' for attaching comments as well as textual data like stack traces to the problem report.

Upon receiving a new problem report, the Bugwranglers, the Gentoo community's equivalent of a change control board for corrective maintenance, assigns the problem report to the responsible herd. The Bugwranglers assign the problem report with basis in the classification of the problem, without analysing the reported failure themselves. The

responsible herd assigns the problem report to a developer. The assigned developer then resolves the problem report. Figure 7-3 summarizes the process.

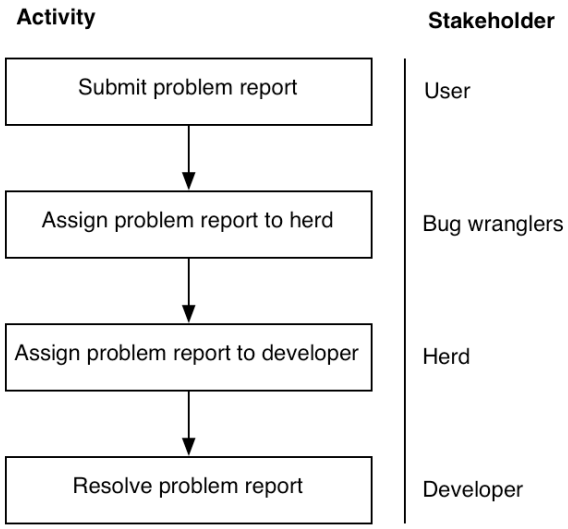


Figure 7-3 Overview of the corrective maintenance process

The corrective maintenance process reaches closure in one of five ways, summarized in Table 7-3 below. Reaching a closure with one of the five resolutions to the problem report requires and understanding of the system causing the software failure. It is the process of reaching such an understanding that has been the focus of the reported research on corrective maintenance work.

Resolution status	Description
Correction of problem	The developer corrects the reported failure
Mark with NEEDINFO flag	Further work on the problem report is pending further information
Reject as user failure	The failure is found to be caused by problems with the user's Gentoo system
Mark as duplicated	The failure has previously been reported
Forward upstream	The failure is not connected with the way Gentoo integrates software. It is caused by a defect in the third-party software.

Table 7-3 Closure of the corrective maintenance process

8. The research process

Interpretive research is a result of the researcher's embodied, situated experience (Walsham 2006). It is a process marked by a plethora of more or less conscious decisions, evolving theoretical concepts, beliefs, and practical problems (Avgerou 2005). The challenge when reporting the research is therefore to provide sufficient information to make an intelligible account of the big lines of the research project without overloading the reader with too much information. This chapter section seeks to strike a balance between reporting the different research activities performed and provide an outline of the most important decisions made during the study.

To this end, the research process is split into three distinct periods: fieldwork, study of corrective maintenance, and testing of preliminary results. The overall goal of the project, to explore software maintenance practice as knowledge-intensive work, has been constant from beginning to end. However, each of the three periods reported in this chapter have their distinct focus and place in the overall research process. Figure 8-1 below provides a timeline for the data collection activities undertaken as part of the research process.

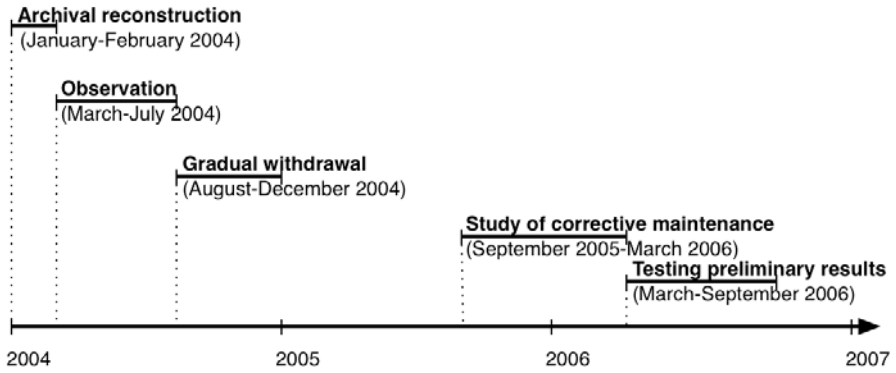


Figure 8-1 Timeline of data collection activities

The purpose of this chapter is therefore to present the research process. It is therefore organized as follows. Sections 8.1 and 8.2 present the fieldwork and the study of

corrective maintenance respectively. Section 8.3 presents the activities performed to evaluate preliminary research findings, while Section 8.4 concludes this chapter with reflections on the evaluation of the reported research.

8.1. Fieldwork

In the period from January to December 2004 I performed empirical fieldwork. During this period, the research underwent a transformation from archival reconstruction of mailing list archives, to participant-observation. While participant-observation was the primary form of data collection during this period, it is supplemented with documents, problem reports, and data from mailing list archives.

The initial focus of the reported research was to study how change requirements are developed in an OSS community. With basis in the observation that there are no formal processes for developing change requirements in OSSD (Scacchi 2002), the intent was to study this as a process of learning the requirements through use and development. Thinking in terms of knowledge-intensive work, developing change requirements was conceived as a form of working-as-learning (Brown and Duguid 1991).

The scheduled empirical research coincided with the Gentoo community's efforts to replace Portage, the Gentoo package manager, beginning in November 2003. The code had become too difficult to comprehend, making Portage difficult to maintain. The effort to develop a new package manager was given the working title of PortageNG – next generation Portage. The PortageNG effort dwindled out during January-February 2004. In its wake a series of failed attempts at replacing Portage were undertaken throughout 2004. The continued efforts at replacing Portage during 2004 therefore formed the focus of the empirical fieldwork reported here.

8.1.1 Archival reconstruction (January-February 2004)

The original plan for the research project was to use a method for reconstructing mailing list archives developed in a previous study (Østerlie 2003). During the period from January to late February 2004 I worked with reconstructing the Gentoo developers' mailing list archives to study the development of change requirements. This study focused upon the PortageNG effort, both the activities leading up to PortageNG as well as the development of it.

Archival reconstruction is a meticulous and time-consuming activity of systematically working through the mailing list archives. It involves a good deal of detective work to locate relevant e-mails and connect them, as well as relating these to documentary sources outside the mailing list. It is therefore a process of relating fragmented pieces of information gleaned from different documentary sources with each other.

By February, though, I found progress to be lacking. There was a distinct lack of activity related to developing change requirements on the Gentoo developers' mailing list. Having gotten in touch with a Gentoo developer in connection with following a lead, I inquired about the lack of such discussions on the mailing lists. I learned that the

Gentoo developers used a number of Internet Relay Chat (IRC) channels for discussing change requirements. I also learned that the PortageNG effort had dwindled out and been abandoned in late January 2004.

8.1.2 Passive observation (March-April 2004)

With this new information, I adopted a different empirical strategy. The shift of empirical strategy coincided with a shift in theoretical focus. Whereas I had originally conceived the process of developing change requirements as a form of working-as-learning (Brown and Duguid 1991), I was turning towards conceiving it as process of constructing facts (Latour 1987). I came to see change requirements as the product of constructing specific problems related to Portage and then proposing solutions for these problems. This shift of focus gained momentum when I used ANT when reporting from the fieldwork so far in late March (Østerlie 2004).

In March I started observing the Gentoo developers on IRC. While there was much activity on the IRC channels, I found only parts of it of interest to my study. As such, I also started a retrospective analysis of PortageNG's demise. Identifying key stakeholders in the PortageNG effort, I sought to investigate their explanations of why PortageNG had been abandoned. These stakeholders were approached on IRC and on e-mail. From these inquiries, I learned of a new effort, GentoolkitAPI. GentoolkitAPI was considered a less ambitious extension of PortageNG. The purpose of the GentoolkitAPI was to provide a stable API for third-party applications accessing the Portage databases.

Observing on IRC channels is an indirect form of observation, in that I had no direct access to the Gentoo developers and users. Rather, I observed their activities through the traces they left on IRC, but also in the tools they used for collaborating. It is a form of observation through artefacts. This is a challenge that I reflect upon in 8.4.1. My access to the field was the same as those I studied, as no two Gentoo developers are geographically co-located. The field, however, differs from direct observations of co-located software development.

8.1.3 Participant-observation (April-July 2004)

Some time during April, I went from observation to participant-observation. Although having been a Gentoo Linux user since 2001, I often found it hard to relate to issues that were discussed in relation to replacing Portage. I therefore decided to participate in hopes of getting more of an understanding. From previous experience as an active Gentoo Linux user, I knew that submitting new ebuilds, reporting problem reports, and resolving incoming problem reports are key activities in Gentoo. I began participation by submitting a series of ebuilds that I had written for applications I was using at the time, but were not yet supported by Gentoo. I then started contributing towards resolving bug reports when these were discussed on IRC. I also submitted some problem reports of my own.

Late April PortageAPI, an experimental API for Portage, was announced on the Portage IRC channel. Having followed the GentoolkitAPI for a while, I was curious to learn why two efforts aimed at the same target were launched. PortageAPI was an extension

of an effort to modularize Portage. Two Gentoo developers had tried and failed at this during March 2004. This modularization effort, labelled portage_mod, had failed. One of the two developers working on portage_mod was now trying a less ambitious plan for replacing Portage: writing an API to 'insulate the internals' of Portage, then rewrite the tools to access the internals through the interface, before re-engineering Portage in the end. I volunteered to developing unit tests for PortageAPI.

Throughout May and June I worked full time developing unit tests, learning firsthand how hard the Portage code was to comprehend. In the beginning of July I went on summer vacation, only to learn the PortageAPI had been abandoned upon returning some weeks later.

8.1.4 Gradual withdrawal from the field (August-December 2004)

Inquiring about the PortageAPI's demise, I was told that it was based upon the wrong assumption of Portage as a single-user application. Two developers were now rewriting Portage as a multiuser application. The multiuser application was to be based on a Unix deamon architecture. The effort was labelled the ebuild-daemon. Having invested a lot of effort in comprehending the Portage source code and writing tests for the PortageAPI, I decided that participating in the ebuild-daemon effort would be too time-consuming. While continuing to participate in resolving failure reports for a while, I gradually reduced participation throughout August and September.

By the end of September I was back to mostly observing, only asking questions to learn more about the activities of rewriting Portage. While the ebuild-daemon effort persisted, so did continued corrective and adaptive maintenance of the original Portage code, too. All in all, I found that the effort to maintain and adapt Portage had remained largely untouched by the many efforts to replace it throughout 2004. By early November 2004, yet another attempt at rewriting Portage appeared: omicron. This, however, was a complete reimplementation of the package manager. History seemed to repeat itself. One year after PortageNG had been announced as a complete reimplementation of the package manager system, yet another complete rewrite was attempted.

At the time of writing up this thesis, Gentoo is still using the original Portage code.

8.1.5 Materials collected

Throughout the period of fieldwork a number of materials were collected. These are summarized in Table 8-1 below.

Data source	Description
Fieldnotes	Each day's fieldnotes stored in a single file identified by its date for ease of reference. All files stored in a single folder.
IRC logs	Logs saved as one file per channel per day, for a total of 1027 files. For ease of reference, each file stored on the format <channel name>-<date> (e.g. gentoo-portage-2004.25.05). All logs saved in a single folder.
Documentary database	70 documents related to 1) the efforts to replace Portage, or 2) background information on the Gentoo community, stored in the documentary database. For ease of reference, each document identified by its date and a serial number (e.g. 2004.01.27-#3), and stored in a separate folder.
Mailing list archives	Archives of 31 Gentoo-related mailing lists provided by the gmane project (http://www.gmane.org). Archives date back to April 7 2001. Accessible with mail client through a NNTP interface. Mail client provides full text search of the entire archive.

Table 8-1 Summary of materials collected

I made *fieldnotes* in the period from April through December 2004. These were jotted down in a note pad by the side of the keyboard. Particularly interesting passages from the IRC channels were copied into a separate file. At the end of the day, the day's fieldnotes were transcribed on my computer. Daily fieldnotes were made in the early stage of observation. As participation intensified, the extent of the fieldnote-taking decreased, to the point where I barely made any notes during the period of most extensive participation during May-June. As I started to gradually withdraw from participation from the end of August, the extent of these fieldnotes became more sporadic. The fieldwork was becoming familiar, and I did not find it necessary to make notes of the familiar. Issues that I judged to be of interest dried up as I gradually withdrew from the field. By December 2004 I barely made any fieldnotes at all.

The Gentoo community has set up over 40 IRC channels dedicated the issues ranging from user support to Gentoo developer communication. During the period from April through December 2004 I had an IRC client connected to the following five channels:

- Development of the Portage package manager
- User support channel
- General developer discussions
- Java-related issues
- MacOS X support for Gentoo.

As there are no archives of the Gentoo IRC channels, my IRC client logged the activity on these six channels to disk, 24 hours a day, seven days a week. The period of June 20 to 29 the logs are less comprehensive due to network problems. There are no logs in the period from June 30 to July 12 2004, as I was off on summer vacation during this period.

Although I quit the archival reconstruction effort, the mailing list archives of the Gentoo developers' mailing list continued to be a source of information. While the Gentoo community does not archive its mailing lists, two independent mailing list archives are available: the Mailing list ARChive (http://www.marc.info) and Gmane (http://www.gmane.org). MARC provides archives for 31 of the Gentoo mailing lists, while Gmane archives 64 of the mailing lists. MARC provides complete archives of the

mailing lists from January 1 2001. GMane provides archives of the mailing lists from April 7 2002. While the MARC archives provide a more extensive history, I used the GMane archives because they are available with mail client through a NNTP interface. The mail client provides full text search of the entire archive. The MARC archives are only available through a Web interface with limited search capabilities.

8.2. Study of corrective maintenance work

By the end of 2004 I had completely withdrawn from the field. During the last months of fieldwork I had started working systematically through the materials collected so far. I continued this work until March 2006, interrupted during the period from June to September 2005. Drawing upon the materials collected during the fieldwork, I developed an overarching 40 pages case narrative (Patton 2002) of the many failed attempts at replacing Portage. Upon returning to the research in September 2005, I decided to supplement the study of replacing Portage with an in-depth analysis of corrective maintenance work.

This section will present this study of maintenance work. The study is presented as a sequence of activities: sampling, constructing case narratives, identification of themes and patterns. While I performed all of these activities, the study unfolded more iteratively. However, I choose this sequential form for the purpose of presentation.

8.2.1 Sampling problem reports

For better control of the data, the Gentoo community's defect tracking system's database was replicated on my computer with a simple script. Over 20.000 problem reports were downloaded. The first task was therefore to reduce the volume of data to identify information-rich problem reports for further analysis. Looking for information-rich problem builds on *intensity sampling*, which is a sampling strategy where cases that manifest the phenomenon intensely are studied (Patton 2002).

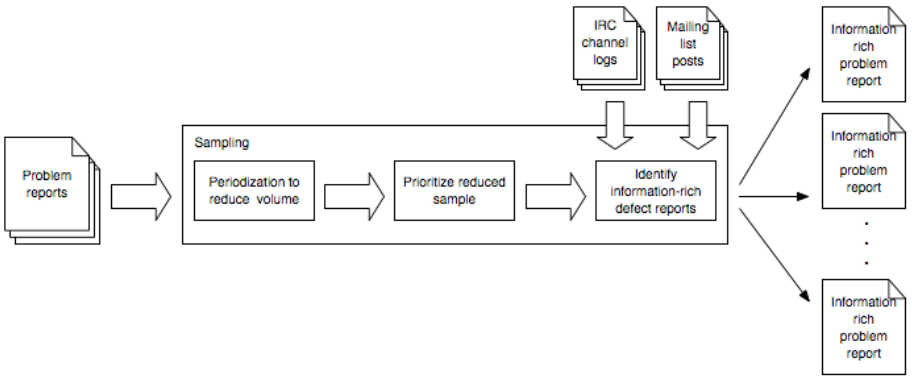


Figure 8-2 The sampling process

First the volume of data was reduced through *periodization*. From the fieldwork I knew that the Gentoo developers often discuss problem reports on the Gentoo IRC channels

and the dedicated developers mailing list. Combining several data sources was a way of identifying information-rich samples for further analysis. Supplementing the problem report with data from the collected IRC logs and the mailing list archives would also enable triangulation during analysis (Fetterman 1998). Only problem reports submitted in the period of my fieldwork were therefore included with the sample.

The volume of problem reports was still too large for in-depth analysis. The reduced sample from the previous step was therefore *prioritized* to identify the most information-rich problem reports. Again, drawing upon experience from my fieldwork I knew that the Gentoo developers and users used a text field at the bottom of the problem reports to communicating back and forth during the corrective maintenance process. An operational indicator for information-rich problem reports at this stage in the sampling process was therefore reports with an extensive back and forth dialogue in this text field. I developed a small script to count the number of comments attached to this text field along with the problem report's unique identification number. The list was sorted with the problem reports with the most extensive back and forth dialogue at the top.

This list was then used to *identify information-rich problem reports*. I knew from the fieldwork experience that the Gentoo developers practically always use the problem report's unique identification number when discussing reported problems. Going from the top of the list, I searched for the identification number in the IRC logs. Reports that had not been discussed on any of the IRC channels were discarded as not sufficiently information-rich. I also used a news client for searching for the problem report's unique identification number in the gmane.org mailing list archives. From the fieldwork, however, I knew that the most significant discussions about reported problems found place on the IRC channels. Problem reports with no e-mail discussions were therefore not necessarily excluded.

8.2.2 Assembling case narratives of corrective maintenance

The second step of the study was to assemble case narratives. At this stage, each problem report was treated as a single case. Focus for the case narrative was corrective maintenance practice: the process of reporting defects and resolving problem reports. Contrary to the focus upon classifying defects and analysing their causes that dominates the research literature on corrective maintenance (Fenton and Neil 1999), the object of study is the activity itself. Assembling case narratives consisted of three activities. The process is outlined in Figure 8-3 below.

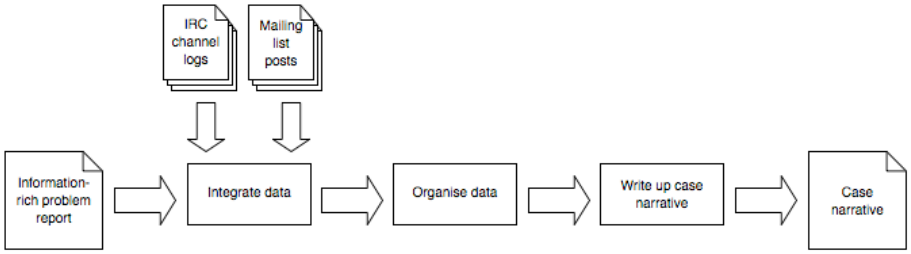


Figure 8-3 The process of assembling case narratives

First, all data from the problem report was integrated with data from the IRC logs and mailing list archives. These were collected in a single Word document. This document was then used throughout the process of assembling case narratives. Next, the data in the word document was *organized* by laying it out sequentially in time. While the data usually started with the initial problem report, there were instances where the sequence of data started with a discussion on an IRC channel or mailing list. Laying out the data sequentially also meant splitting up the problem report, providing the IRC channel and mailing list discussions in between comments attached to the problem report.

The final step was to *write up* the events of resolving the problem report as a narrative. This was a time-consuming process. Similar to Orr's (1996, p. 125) experience from analysing diagnosis work, that "[t]elling stories in diagnosis contexts makes some them extremely elliptical and barely recognizable to outsiders", I often found it hard to grasp the content of the raw data. While organizing the data sequentially provided a clear sequence of events for a problem report, I found particularly IRC discussions circular, convoluted, and full of implicit references to the problem report, documents, and stories circulating in the Gentoo community.

Writing up case narratives was therefore a dialectic process of shifting between writing the narrative and working with the raw data. In writing out the narrative, I came across statements and issues that were unclear. I found implicit references that I had to figure out by looking through the different data sources. Once uncovered, the data of these implicit references were copied into the timeline. Particularly references to installation scripts and the data provided with these were uncovered in this dialectical process.

8.2.3 Identifying themes and patterns

While writing the case narrative involves a degree of content analysis, cross case comparison is a more detailed content analysis that involves "identifying coherent and important examples, themes, and patterns in the data … quotations or observations that go together that are examples of the same underlying idea, issue, or concept" (Patton 1987, p.149 quoted in Cope 2005, p.179). The process of finding themes and patterns was a form of cross-case comparison, where I studied compared the assembled case narratives.

At this stage in the process I employed a technique called *bracketing*:

> *In bracketing, the researcher holds the phenomenon up for serious inspection. It is taken out of the world where it occurs. It is taken apart and dissected. (...) It is treated as a text or a document; that is, as an instance of the phenomenon being studied. (Denzin 1989, p.55-56 quoted in Patton 2002, p.486).*

Rather than focusing on the software failure and the nature of its corresponding defect, I bracketed these concepts analysing corrective maintenance practice as the process of submitting and resolving problem reports. The practice of corrective maintenance was reduced to the case study narrative.

A number of patterns were identified during this process, which in turn where thematized. I will now use a concrete example to give an impression of this part of the process. Østerlie (2006) is an illustration of the relation between what would form the pivotal patterns and theme reported in this thesis. I had identified a pattern that I labelled *interpreting*. It was based on the following observed pattern:

> *To make sense of failures reported in bug reports, the [Gentoo] developers discuss a number of possible sources for the failure. Of these possible explanations, I find that none are dismissed on conclusive evidence. (ibid., p. 336).*

Furthermore, focusing upon the roles of different actors in the process, I identified a pattern I labelled *producing debug texts*, wherein the role of the user is to provide the Gentoo developer with more information about the software failure occurring on the user's local system. These two patterns were then thematized to the *practice of producing and interpreting debug texts*:

> *(...) I find that none are dismissed on conclusive evidence. Instead, alternative explanations for reported failures are made more or less plausible by producing new debug texts, trying to reproduce the bug, and drawing on external texts like installation scripts, source code, documentation, and change logs. (ibid.)*

This is also an illustration of the use of an emic perspective during analysis. Emic analysis is sensitive to the vocabulary and practices indigenous to the studied subjects (Patton 2002). The term comes from anthropology, and is used in opposition to the etic perspective where categories created by the researcher are used to structure and drive the analysis. Assuming an emic perspective together with the technique of bracketing, are the two primary techniques used to study the practice of corrective maintenance from the insider's point of view.

Constructing case narratives and identifying patterns and themes was an iterative process. Analysing all of the problem reports identified as information-rich would have been too time-consuming. Instead, I started by constructing case narratives for three problem reports. Then, while identifying patterns and themes, I would go back to construct case narratives for a new problem report to see if any new patterns or themes could be identified. I iterated between these two steps several times, until the number of new themes occurring in the case narratives dwindled away. Also, as the process progressed, I opted more and more for simply organizing the raw data of a case

chronologically and writing a small executive summary rather than writing a long case narrative. With experience in studying problem reports along with the emerging picture of patterns, I was able to identify patterns more easily.

8.3. Testing preliminary research results

Since 2003, I had grown increasingly uneasy about the espoused view of OSSD as completely different from software engineering (see Section 4.3). However, after writing up Østerlie and Wang (2006), I was growing increasingly concerned about the relevance of my research in relation to software engineering. The proof of the pudding, I decided, was to test the *transferability* of the preliminary research results outside the context of volunteer software development.

Transferability is an approach for establishing the applicability of research results outside the research setting (Patton 2002). It is an indirect approach to generalization, as it is a speculation on the likely applicability of findings to other situations under similar, but not identical, conditions. This kind of generalization is case derived and problem oriented rather than statistical. To evaluate the transferability of preliminary the research results, I decided to organize group sessions with industrial software integrators. In the period from March to September 2006 I organized three such group sessions (summarized in Table 8-2).

Date	Description
Session #1: March 2006	An international network of software researchers and senior software engineering practitioners with experience from the European software industry
Session #2: May 2006	A group of senior software consultants working with systems integration in different large-scale software integration projects throughout the Norwegian software industry
Session #3: September 2006	A group of systems developers working with systems integration at the research and development department of an international telecom company based in Norway

Table 8-2 Summary of group sessions

While the groups work with software integration in the software industry, the particular selection of group selection was based on opportunity and convenience. Session #1 came about in the wake of a department-wide call for presentations for a workshop being held locally for a network of international researchers and software engineering practitioners. Session #2 was held for a group of practitioners from a former employer of mine, an IT consultancy. Session #3 came about as a former colleague from the department invited me to present my research at a monthly colloquium at his current work place.

8.3.1 Organization of the group sessions

Each evaluation session was planned to last an hour: 30 minutes for presenting the research results, followed by 30 minutes of group discussion afterward. Before starting on the presentation of the research results, the participants were told that the purpose of the session was to learn more about similarities and differences between software maintenance in Gentoo and their experiences from an industrial setting.

To provide the audience with an understanding of the specifics of the Gentoo context, the presentation started with an overview of the research setting. This was then followed by an in-depth example of corrective maintenance work. The example emphasised on the indeterminate nature of reported defects, and the iterative process of producing data about the defect and interpreting the data. As such, the key finding presented was the process of negotiation over what the defect "really is", and consequently the social and technical nature of software defects. With basis in the negotiated nature of software defects, the last part of the presentation was devoted to the issue of maintenance problems. Focusing on the process of negotiation over what the problem "really is", I illustrated the similarities between problem setting in corrective and adaptive maintenance in Gentoo.

The latter half of the session was left open to discussion and feedback from the practitioners. Apart from session #1 that was held as part of a workshop with a strict time schedule, the group discussion after the presentation lasted over 1 hour. The practitioners related the presentation to particular individual experiences. In session #2 the practitioners even started discussing among themselves. Also, in all three instances, attendants approached me after the session to discuss the issues further.

8.3.2 Materials collected

I made brief notes during each session. Immediately after the sessions I made more extensive notes based on the feedback and my own reflections.

8.4. Research evaluation

In interpretive research, the credibility of the results is grounded in the research practice the results are a product of. The evaluation of the reported results is therefore to be grounded in the way data is collected, how they are analysed, as well as in the presentation's rigour of argumentation (Walsham 1995). Rather than delegating the task of evaluation to abstractions like validity and reliability (Kirk and Miller 1986), interpretive research accounts give the reader a more active role in evaluating the reported results.

The purpose of this section is therefore to bring attention to parts of the research practice I consider important for evaluating the credibility of the reported research. Table 8-3 below provides an overview of how the seven principles for evaluating interpretive fieldwork have been addressed in this thesis. The remainder of this section reflects upon issues I consider important for evaluating the reported research, but not yet been previously addressed.

Principle	Description	Evaluation
1. The principle of hermeneutic circle	This principle suggests that all human understanding is achieved by iterating between considering the interdependent meaning of parts and the whole that they form.	• Considering observations from Gentoo in relation to issues in the broader Internet-based OSS community (Section 8.1)
2. The principle of contextualization	The principle requires critical reflection of the social and historical background of the research setting.	• Situating software engineering in the context of professionalizing modern work (Section 2.2) • Considering OSS as a product of the software industry during the 1990s (Section 4.1)
3. The principle of interaction between the researcher and subjects	Requires critical reflection on how the research materials were socially constructed through the interactions between the researchers and participants.	• The use of observation for getting to grips with the field (Subsection 8.4.1) • The form of interaction with the Gentoo developers (Subsection 8.4.4)
4. The principle of abstraction and generalization	Intrinsic to interpretive research is the attempt to relate the particulars described in the unique instances observed to abstract categories and concepts that apply to multiple situations.	• Theoretical framework (Section 6.2) • Use of theory in framing research contributions (Chapter 9 and 10)
5. The principle of dialogical reasoning	Requires sensitivity to possible contradictions between the theoretical preconceptions guiding the research and the actual findings.	• Describing how the theoretical framework changed throughout the research process (Subsection 8.4.3)
6. The principle of multiple interpretations	This principle requires the researcher to be sensitive to differences in interpretations among the studied subjects.	• The form of interaction with the Gentoo developers (Subsection 8.4.4) • Emphasis on multiple stakeholders with different interests in the research contributions (Chapters 9 and 10)
7. The principle of suspicion	Requires sensitivities to possible biases and systematic distortions in the narratives collected from participants.	• The form of interaction with the Gentoo developers (Subsection 8.4.4)

Table 8-3 Summary of research evaluation

8.4.1 Getting to grips with the field

Section 5.2 discusses how the quality of interpretive research depends on researcher's level of immersion in the natural environment of the research subjects. This is not always possible, in which case the researcher needs to make plausible that there has been enough interaction with the research subjects and archival material to compensate for the lack of direct immersion (Pozzebon 2004). Doing Internet-based fieldwork, I had the field available at my desk. However, because I accessed the field through my computer, my engagement has been through textual media. The textual media include both real-time interaction in the case of IRC and – and to a certain extent e-mail – as

well as non-interaction in the case of archival data like mailing list archives, problem reports, and Web pages.

Nandhakumar and Jones (1997) find textual analysis to offer the farthest distance to the research setting (Figure 5-1). Compared to immersion, it has the maximum distance between the researcher and the research subjects. The purpose of this subsection is therefore to shed light on the methodical decisions made in order to compensate for the limitations inherent in doing Internet-based fieldwork.

8.4.1.1 The problem of gaining entry to the field

Rosen (1991) writes that "to understand social processes one must get inside the world of those generating it". While gaining access to the Gentoo community is a matter of subscribing to a set of mailing lists and connected to the IRC channels, getting inside the world of those generating the social processes was more of a problem. Acknowledging the limitations of working with only textual data, I sought ways of narrowing the distance between myself – the researcher – and the Gentoo developers – the research subjects. I initially turned to personal e-mail for inquiring about the Gentoo developers' interpretations of unfolding activities. However, the answers I received – when I did get any reply at all – were short, lacking in detail, and schematic.

At the time, I attributed this to the e-mail medium. I therefore saw the possibility of real-time interaction offered by IRC as a way of increasing my engagement with the Gentoo developers. Yet, after having presenting the objectives of my research to the Gentoo developers I approached on IRC, I still found them uninterested in responding to my inquiries. Although having access to the research site, I found myself in a position where I was excluded from practically any form of meaningful interaction with people who could provide me with an insiders' view of the Gentoo developer community.

Having no go-between within the community to function as an icebreaker (Fetterman 1998), I opted for a two-pronged approach for gaining entry to the field. One, to make myself less of an outsider, I decided to adopt common conventions when interacting with the Gentoo developers. Two, in an effort to justify my inquiries, I decided to actively participate in the Gentoo community's software development activities.

Having observed that it was common to sign e-mails with a GNU Privacy Guard (http://www.gnupg.org) signature, I started doing the same. In addition, I started paying close attention to the form of the Gentoo developers' IRC communication. I adopted conventions such as appending words I was unsure of their spelling with a '(sp?)'. This means, did I spell that correctly, invoking an IRC bot that would correct the word if misspelled. I also adopted the practice of correcting typographic errors with sed-like syntax. For instance, if I had written 'We need to calrify this', I would correct myself by writing 's/calr/clar'. I also learned the implicit rules of which questions to ask in public and when to use private IRC sessions.

It is difficult to tell the effect adopting common conventions of interaction within the Gentoo community had on gaining entry. At least it made me stand less out as a sore thumb. Turning to participation, on the other hand, had a provably significant effect on

gaining entry to the field. However, unlike my presupposition that participation would justify my inquiries, it turned out that taking part in shared activities was the key to gaining entry. I will reflect upon this in what follows.

8.4.1.2 The importance of participation

Looking back at IRC log transcripts from the period prior to participation, I find my inquiries and questions to be abstract or general. It is even hard for myself to understand what I was asking about now, years after the fact. In comparison, I find the questions asked while being an active participant myself to be significantly more concrete and most of the time connected to particular issues of our shared activity. Concrete questions are commonly responded to with concrete answers. Through concrete questions about our shared activity, I was able probe deeper into the inner life of the Gentoo developer community. As such, participation was the key for gaining entry into the community's inner life.

By engaging in shared activities, I could ask the Gentoo developers for practical advice on how to solve common problems. As such, I was to a certain extent able to indirectly observe the Gentoo developers through following their practical advice. I also found that practical problems I was facing were often the same problems the Gentoo developers themselves were facing. Practical advice to my questions therefore often turned out to be conventions of working shared within the Gentoo developer community. As such, while I was prevented from direct immersion with the Gentoo developers in their daily work, one participant was available for direction observation: myself.

Obviously, there are limits to the usefulness of observing my own activities instead of direct observation of the Gentoo developers' activities. Experience is an important issue here. While having been a Gentoo Linux user for several years prior to commencing the fieldwork, I was also what the Gentoo developers would call a newbie. Active participation over a period of ten months is not sufficient for becoming a seasoned Gentoo developer.

Still, for developing interpretations, these shortcomings are less of a problem. My access to the social reality of the Gentoo community through texts is exactly the same as the Gentoo developers' access. They relate to each other through the same textual media that I did. However, the limitation is that I have no data on the Gentoo developers' actual activities in going about their daily work. On the other hand, I have much data on their interaction while pursuing these individual activities.

8.4.2 Interaction particulars and whole

That human understanding emerges from the iterating between particular observations and the whole they form, is a fundamental principle in interpretive research (Klein and Myers 1999). I will use scarcity of resources to exemplify such interaction in the reported research. Scarcity of resources is reported as a characteristic maintaining an integrated system in a community of volunteers in Contribution C3 (Section 9.3).

A first particular observation from the fieldwork is that the Gentoo developers avoid reproducing reported failures as long as possible. Rather, they use the 'Additional comments' field of problem reports to engage in a dialogue with the reporting user. A second particular observation was related to reproducing reported failures without disrupting one's own Gentoo system. To avoid this, the Gentoo developers try to reproduce failures in a virtual Gentoo system. They therefore maintain a 'barebones' virtual system that they copy when trying to reproduce reported failures. This copy of the virtual system is then updated with the required packages to reproduce the reported failure.

To understand the social and historic setting of Gentoo (see Guided Klein and Myers' (1999) principle of contextualization), I actively followed key forums for the larger Internet-based OSS community. Developer burnout and lack of project progress because of too high workload was a recurring issue. The burnout of Linus Torvalds, leader of the Linux kernel project, in 2002 is one of the most high-profiled such incidents (Weber 2004). In April 2004, Daniel Robins, the initiator and leader of the community, left Gentoo because of burnout.

During the study of corrective maintenance work, I developed some descriptive statistics. These showed that the number of unresolved problem reports was continuously growing (see Exhibit 9-2). Individually, the two observations on use of the 'Additional comments' field and virtual systems were seemingly unrelated. In the same way, the observations related to burnout had little relation to the previous two relations. However, considered these particular observations as a whole together with the growing number of unresolved problem reports, I realized that the reluctance for reproducing reported failures was related to scarcity of resources.

As all the software has to be compiled from source code, building a virtual system from scratch is time-consuming. It may take as much as a day depending on the computer's hardware. That is why the Gentoo developers maintain a 'barebones' setup. Yet, compiling the required software to reproduce the problem may take hours. Instead, to avoid spending their limited resources, they use the 'Additional comments' field to engage in a dialogue with the reporting user instead. By considering the particular observations as a whole, I therefore came to interpret corrective maintenance work as a continuous process of balancing the effort spent on resolving individual problem reports towards the total workload.

8.4.3 Interaction data and theory

I worked actively with theory throughout the research process. Theory use as well as theories used has changed throughout the research process. I initially used communities of practice theory (Brown and Duguid 1991; Lave and Wenger 1991; Orr 1996) as a guide for planning the research. During the fieldwork theory functioned as a form of scaffolding (Walsham 1995) for making sense of the fieldwork experience and the data collected. However, rather than seeking to apply a particular theory to the fieldwork, I used theory actively for exploring software maintenance as knowledge-intensive work.

The scaffolding was later taken down once it has served its purpose. Theory use therefore gradually underwent a shift towards the role of abstracting and generalizing from particular observations (Klein and Myers' (1999) principle of abstraction and generalization) for developing research results during analysis.

Through an interaction with the collected data, the theories used also changed. Figure 8-4 indicates significant moments in this process. The entire research process was an iterative process of interpreting collected data, reflecting upon the appropriateness of theory in relation with research findings, and revising theory (Klein and Myers' (1999) principle of dialogical reasoning).

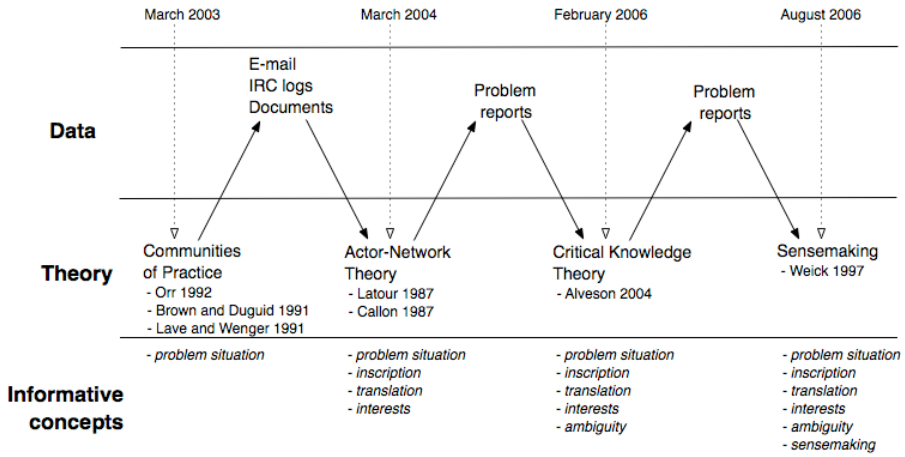


Figure 8-4 Interaction data and theory

8.4.4 Interaction with research subjects

I spent a lot of time talking with Gentoo developers throughout the period of conducting fieldwork. I did so to bringing forth different interpretations about the process of replacing Portage. This is similar to Klein and Myers (1999) principle of multiple interpretations, stressing that multiple stakeholders will express different interpretations of events and sequences of events. I would also use such conversations as occasions of testing other Gentoo developers' interpretations. Always making sure not to embarrass anyone but revealing their private interpretations, I would for instance ask: "I have heard that Portage-NG failed because people could not agree on programming language. Was it really so?". Posing questions this way often provoked the respondent into giving his own interpretation of facts. In the case above, for instance, the respondent answered:

> *The big problem wasn't the 'what' but the 'how'. waiting for the community to provide requirements without anyone leading the project doesn't work. and the 'requirements' that were provided were mostly crap. that and the AI vision killed it.*

While sometimes using the above technique to test my own preliminary interpretations, I also used key informants for giving feedback on preliminary versions of papers. This

in order to test my own interpretations. The result of such discussions varied. In commenting on an early draft of Østerlie (2004), the informant stated that while finding no factual errors he was unfamiliar with the topic and style of reporting.

In working on Østerlie and Wang (2007), another informant challenged our interpretation. The draft paper included in-depth empirical material on controversies over the cause of reported failures. The informant diligently analysed each of the controversies in an effort to determine whose diagnosis of the reported failure was correct. He argued that corrective maintenance activities could not be understood unless we also understood the real nature of the reported failure. While not being directly useful to the paper, the informant's focus on understanding whose diagnosis was correct or not made me revisit our emphasis in analysing corrective maintenance.

8.4.5 The personal journey

This research project has been a personal journey for me in several ways. My understanding of the OSSD phenomenon has evolved since I started the research reported here. My initial view of OSSD was that of something completely different from software engineering. The project follows up my research on OSSD that I did for my master thesis (Østerlie 2003). My interest in operating systems during lower level university studies led me to Linux in the mid-1990s. While this was initially a technical interest, it also exposed me to the Internet-based hacking culture. Hacking and OSSD were synonymous in my eyes. Inspired by the work of Hannemyr (1999) and Levy (1984) on the one hand, and Braverman-like analyses of formalization, automation, and down-skilling on the other hand (Orr 1996), I had come to consider hacking as a way of embracing the craftsmanship and know-how involved in developing software. In contrast, I considered software engineering with its emphasis on formalization of software development methods and processes as a way of down-skilling software developers. Influenced by the discourse on technology as a means to counter oppressive working place conditions (Bjerknes and Bratteteig 1995), I therefore saw the emancipatory potential in hacking's emphasis on craftsmanship practical know-how.

Looking back, I have to admit that my initial view of OSSD as completely different from software engineering seems almost embarrassingly naive. I no longer think that OSSD is completely different from software engineering, and I am not entirely sure that formalization of software development processes and techniques will down-skill software developers. Yet, my deep appreciation of what software developers do in practice has remained throughout the duration of the study. Some of this is obviously connected to my own background within home computing. I received my first computer aged 11. It was a low-cost ZX Spectrum clone, but no official Spectrum data tapes worked with it. I therefore had to write my own software. Since then, I have always had an enduring fascination with the practice of developing software.

I hope that my understanding of the complex disciplinary field I am moving within has evolved over the years. Yet, in many ways the same concern I brought into the research project remains: the relationship between research and practice. This relationship was initially conceived as the adversary relationship between the art and craft of hacking on the one hand, and the down-skilling intention of software engineering on the other.

Throughout the process of doing this PhD I have to come appreciate the dialectical relationship between research and practice. The importance of researchers to both try to inform practice and in turn be informed by practice.

PART III: RESULTS

9. Empirical findings

The main empirical contribution offered by this thesis is insight into the social and technical processes of maintaining an integrated system in a distributed community of volunteer software integrators. In particular, this chapter presents three empirical contributions that offer a view of software maintenance where multiple stakeholders with different interests continuously negotiate over *problems and their solutions*. Focusing upon scarcity of resources and contradictory interests brings out the inherently political aspects of software maintenance. So far, software engineering research has focused upon developing methods, tools, and techniques independent of their context of use. The politics of software development has therefore been of limited concern. Yet, with practice studies' emphasis on the social context of software engineering, the politics of software development becomes an important issue.

While the politics of software development is well-established within the related field of information systems research (Howcroft and Wilson 2003), emphasising that importance on multiple stakeholders with different interests in software maintenance practice is novel within software engineering. This thesis therefore contributes to the body of software engineering practice research with a first step towards explicitly addressing politics in such studies. This thesis also contributes to software maintenance research with a critical evaluation of the basic assumption that software maintenance is essentially a cognitive problem solving activity. This is based on the premise that maintenance engineers are faced with more or less clearly defined problems. Yet, Contributions C1 and C2 show that the essential activity when maintaining integrated systems is *problem setting*: the collective process in which situations that are unclear, problematic, and puzzling are progressively clarified. This chapter therefore concludes that the basic assumption no longer holds true when maintaining integrated systems.

This chapter is therefore organized as follows. Sections 9.1 and 9.2 present contributions C1 and C2 respectively. These contributions summarize and draw together results previously reported in the empirical papers included with this thesis (see Figure 1-1). With basis in these two contributions, Section 9.2 presents contribution C3. This contribution aggregates the findings reported in C1 and C2 to form an original contribution reported in this thesis. Section 9.5 discusses the contributions.

9.1. Debugging as collective activity (C1)

This section presents contribution C1, by summarizing the research reported in Østerlie and Wang (2006; 2007). It is based upon participant-observation of corrective maintenance, as well as document analysis of problem reports (see Section 8.2). An overview of Gentoo's corrective maintenance process is provided in 7.3.2. The contribution is summarized as follows:

> *Knowledge of software failures is developed through a process of negotiating over possible interpretations of available data, a process that is contingent upon situational issues such as workload, priorities, and responsibilities*

The contribution is offered in response to RQ1:

> *How is knowledge of software failures developed during geographically distributed software maintenance?*

An often used definition of debug is to detect, locate, and correct defects in a computer program (IEEE 1990). Debugging is therefore seen as a linear process where the maintenance engineering locate defects by tracing along the infection chain from more or less well understood problem, the failure (Zeller 2006). The solution to the problem is to correct the defect. Building upon scientific principles, it has been proposed that debugging should be hypothesis-driven, based on accurate, factual data (Araki et al. 1991).

The description of debugging offered here differs from the linear view presented above. Here debugging is found to be a cyclic process where the reported problem is not always clearly understood before there is a solution to it. This activity focuses upon understanding the reported failure, rather than locating and correcting the defect causing it. Debugging is therefore understood here as the process of finding out what the reported problem really is. This is a collective process shaped by social as well as technical factors. It is a process of trial and error, where the relevance and validity of available data is contestable. Debugging is therefore driven by plausibility rather than accuracy.

The implication of this contribution is that the software failure is not an unproblematic phenomenon during software maintenance. It is subject to interpretation and negotiation. Integrators' understanding of what constitutes a software failure is contingent upon situational issue such as workload, priorities, responsibilities, as well as available technical data. Failures are therefore not necessarily stable phenomena to be grasped with scientific principles.

To this end, this section is organized as follows. First, in the Gentoo community knowledge of software failures is predominantly based on indirect data (9.1.1). Then the cyclic nature of debugging is presented (9.1.2). Finally, the section is concluded with a presentation of the negotiated and contingent nature of debugging in the Gentoo community (9.1.3).

9.1.1 Indirect data

Understanding a reported failure is not an individual activity. Rather, it is a collective process where Gentoo users and developers together make sense of the failure. They engage in a collective sensemaking process where knowledge of the failure is primarily attained through indirect data. This data is produced by running diagnostic tools on the failing system. In Østerlie and Wang (2006) we label such textual data, along with other textual information provided in problem reports, *debug texts*. There are two reasons why knowledge of reported failures is based on indirect data:

- Reproducing reported failures is often difficult
- Software being integrated is treated as a black box

First, reproducing failures is often difficult. To reproduce a reported failure, the Gentoo developers often have to reproduce the configuration of the failing system. Yet, because of the potentially large and non-deterministic variability of Gentoo systems (see 7.2.4), reproducing a failing system's configuration may be difficult in practice. Reporting from a longitudinal study of large-scale software maintenance, Adams (1984, p. 13) makes a similar observation: that the typical failure "requires unusual circumstances to manifest itself, possibly in many cases the coincidence of very unusual circumstance". Reproducing reported failures is also difficult, or at least inconvenient, as it may be time-consuming to reproduce the configuration of a failing system (see 8.4.2).

Second, while the Gentoo developers have access to the third-party software being integrated, they usually treat it as a black box. This is related to the Gentoo developers' role as software integrators. The software being integrated is mostly developed and maintained by third-party OSS communities. Only some Gentoo developers may be familiar with the source code of the software. Reproduction, essential for locating defects in source code, is therefore less relevant when debugging black boxes.

Standard Unix development software as well as Gentoo-specific diagnostic tools are used to generate data about the failing system and the reported failure. Debug texts therefore play two roles in debugging reported failures. They stand in for the source code, as the Gentoo developers treat the software they integrate as black boxes. The debug texts are also delegated the task of communicating information about a remote failing system, instead of reproducing the failure.

9.1.2 Cyclic

The Gentoo users and developers use the 'Additional comments' field to exchange debug texts when resolving problem reports. Debug texts only provide a limited glimpse of the failing system. The data is not exhaustive, but rather open to interpretations. It is therefore not necessarily obvious what the software failure really is. Rather, users and developers are often confronted with problem situations. A problem situation is a situation where it is clear that something is not right, but it is rather unclear what the problem is (Schön 1991). As such, software failures often have to be constructed from the materials of situations that are problematic, uncertain, and puzzling. Exhibit 9-1 below is an example of such a problem situation.

Developer A: This particular Web page crashes both the Mozilla and Galeon Web browsers.
Developer B: That doesn't happen on my computer.
Developer A: I've built the applications for the Athlon T-Bird processor architecture, and both have been compiled with the GTK2 widget library. I generally assume it's my using GTK2 that messes it up.
Developer B: It might be GTK2. I've compiled both Web browsers with the GTK1 widget library on my system.
Developer D: Well, that page works on my Epiphany Web browser compiled with the GTK2 library.
Developer C: And it works with my installation of Mozilla compiled with GTK2.
Developer D: This other Web page crashes my Phoenix Web browser, but not Mozilla or Galeon.
Developer A: The Web page crashes on my Epiphany installation, as well. It seems it's my Mozilla build that's flakey.
Developer C: But boingboing.net crashes my Epiphany installation. I've compiled it for the PentiumII processor architecture, though.
Developer A: boingboing.net crashes Galeon on my system, too.
Developer B: boingboing.net working for Mozilla on my system.
Developer C: Hmm… It seems the problem is related to Mozilla compiled with the GTK2 widget library and the Xft font library. Weird thing is that boingboing works on my Galeon installation…
Developer A: Now here's a very good reason to only build for one processor architecture, stable source tree and only do point releases. Variation kills reproducibility.

Exhibit 9-1 Excerpt from Gentoo developers' IRC channel (gentoo-dev-2004.04.16)

Two factors are of particular note here. One, that it is not obvious to the user reporting the failure what information is relevant. Two, the Gentoo developers do not have direct access to the failing system. They therefore have to interact with the user to establish an understanding of the failure. However, without first having an understanding of the failure, it is difficult for the Gentoo developers too to determine what constitutes sufficient and relevant information for analysing the failure.

The Gentoo developers therefore interact with the reporting user in order to make sense of the reported failure. This interaction therefore takes the form of a cyclic process of producing and interpreting debug texts using the problem report's 'Additional comments' field for communication between the stakeholders. Table 9-1 below illustrates the frequency of this interaction between users and developers.

Number of additional comments	2002	2003	2004
1	2948	4900	9620
2	2882	4584	7913
3	1967	3451	5472
4	1311	2665	3767
5+	3652	8315	12567
Average	4.225862069	4.797742003	4.787183202

Table 9-1 Frequency of interaction in 'Additional comments' field

The table shows that most problem reports have between 1 and 4 additional comments attached to them. Drawing upon information rich samples (see 8.2), the reports analyzed during the study of corrective maintenance belong to the category of problem reports with 5 or more additional comments. While a significant amount of problem reports have 5 or more additional comments attached, they are not statistically representative for the entire population of problem reports. This, however, implies that the reason for the interaction between users and developers are contingent upon the nature of particularly problematic failures. This is not the logic pursued in the reported research.

Yet, the argument pursued here is that the interaction between users and developers come as a result of the problems of reproducing failures. This is contingent upon the autonomous and heterogeneous characteristics of integrated systems. These characteristics make it difficult for the Gentoo developers to reproduce reported failures. Instead of reproducing reported failures, the Gentoo users and developers therefore have to debug with indirect data. As such, the form of generalization pursued here is that of extreme cases (Patton 2002). It is upon studying the mechanisms of the information-rich samples that we learn about mechanisms that are to a smaller or larger degree shared by all instances of debugging in Gentoo, and possibly even when debugging integrated systems in general.

9.1.3 Negotiated and contingent

It is not always possible to determine what the reported failure "really is". It is not always obvious. Nor are the debug texts conclusive. Rather, the debug texts are open to interpretation. During the cyclic exchange between developers and user, a number of possible interpretations are discussed. Alternative interpretations of what the reported failure really is are constructed from combining elements from different debug texts, trying to reproduce the failure, drawing on external texts like installation scripts and change logs.

Problem solving reaches its closure when the problem is solved. In the context of debugging, the problem is solved when the defect causing the failure has been corrected. Negotiations, on the other hand, have no such clearly defined closure mechanisms. As such, interpreting debugging as a process of negotiation shifts analytical emphasis towards the closure mechanisms. The research reported here identifies two formal forms of closure: resolution or rejection of the problem report. However, these are the forms of closure, not the mechanisms leading to closure.

The question we therefore seek to address in both Østerlie and Wang (2006; 2007) is how problem reports reach their closure. In addition to being based upon the evidence put forward by the debug texts, we find that closure is contingent upon situational issues such as workload, priorities, and responsibilities. Closure mechanisms are discussed further in 9.3.2.

Østerlie and Wang (2006) illustrates this as a process where users provide debug texts and the developers interpret them. Over time the number of interpretations of the reported failure varies, until the problem report reaches its closure. This is illustrated in Figure 9-1 below.

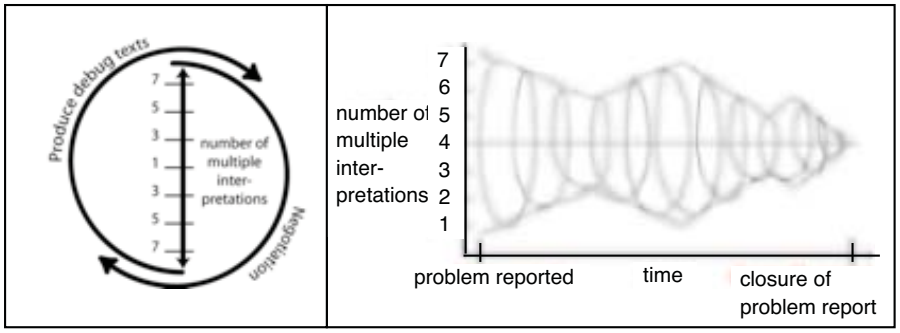


Figure 9-1 Producing and interpreting debug texts

9.2. Rewrite evolves in response to an unfolding environment (C2)

This section presents Contribution C2, by summarizing the research reported in Østerlie (2004) and Østerlie and Jaccheri (2007b). It is based upon the fieldwork conducted during 2004 as well as the materials collected during the fieldwork (see Section 8.1). The contribution is summarized as follows:

> *A collective understanding of the scope, stakeholders, and sequence of activities for rewriting software evolves in response to new problems emerging from the rewrite efforts themselves as well as environmental changes*

This contribution is offered in response to RQ2:

> *How do software developers build knowledge of how to replace a business-critical software system?*

Both Østerlie (2004) and Østerlie and Jaccheri (2007) reports from the activities of rewriting and replacing Gentoo's package manager, Portage. While Østerlie (2004) offers in-depth analysis of a single event in the process of rewriting and replacing, Østerlie and Jaccheri (2007) offers a longer case narrative, exploring the tension between the need for functional stability to rewrite Portage on the one hand, and the various social interests of the Gentoo community on the other hand.

This section builds upon these two papers, offering an interpretation of the efforts to replace Portage as a continuous process where rewriting evolves in response to an unfolding environment.

9.2.1 The problem situation

Portage provides functions and data that are critical to Gentoo. It is therefore business-critical (see Table 3-2) to the Gentoo community. Over time, however, it has become increasingly difficult to modify. By November 2003, only four Gentoo developers know

the source code well enough to maintain the package manager. This puts a lot of strain on these four developers.

While initially designed to integrate third-party software on Gentoo Linux, Portage has been adapted to work on several Unix-like operating systems (see 7.2.2). Similarly, Portage has been adapted to work with Web applications as well as regular software (Østerlie and Jaccheri 2007b). With little attention on reducing the complexity of the source code, Portage's system structure has deteriorated (Eick et al. 2001). As one developer put it, the source code is 'very fragile' as it has 'evolved rather than being designed'. Complex interdependencies between functions and modules make it difficult to comprehend parts of the source code without a complete understanding of the whole. It is therefore difficult to modify the source code without breaking existing functionality.

Portage also has complex interdependencies with third-party software that integrates with Portage. A number of third-party applications call functions in the Portage source code or access Portage's databases directly. As described in Østerlie (2004), such interdependencies often lead to problems with the third-party software when Portage database schemas are changed or when the source code is modified.

While four Gentoo developers have sufficient understanding of Portage's source code to modify it without breaking existing functionality, they have little control of the effects this may have on third-party software. Belady (1978, p. 118) defines system largeness as a "program that is too large to fall fully within the intellectual grasp of a single individual". While Portage's source code is not outside the intellectual grasp of a single individual, Portage's interdependencies with third-party software makes it outside the grasp of a single individual. As such, it exhibits a form of system largeness.

Portage has been maintained for no more than four years at the time of the study. Still, it exhibits characteristics similar to legacy systems (see Table 3-3 and Table 3-4). It suffers from:

- Deteriorating system structure
- System largeness
- Lack of skills for maintaining the software

The Gentoo community is therefore facing its own legacy systems dilemma (see 3.2.4). Portage is the core of the Gentoo software distribution system and cannot be decommissioned. Yet, only four developers able to modify the source code. If these four leave, the community stands the risk that no one is able to maintain Portage any longer.

9.2.2 An outline of the case narrative

To address this situation, the Gentoo developers want to rewrite and replace Portage. We describe three efforts to rewrite and replace the package manager in Østerlie and Jaccheri (2007b):

- *Next generation Portage (November-December 2003)*: A complete rewrite of the package manager with a modularized plug-in architecture.
- *Modularized Portage (February-March 2004)*: A modularization of the existing code base.
- *Portage API (May-June 2004)*: Preparation for a modularizing Portage by encapsulating the package manager's source code and databases from Portage-specific third-party applications.

9.2.3 The constituents of rewriting requirements

The process of rewriting and replacing Portage does not follow a clear sequence of activities from analysis of current situation, to planning the strategy for rewriting and replacing the software, to the implementation of the plan. Rather, the process is marked by a series of efforts with the shared intent of replacing Portage. Judged in terms of the assumption that software replacement can only succeed if properly planned (Sneed 1995), the efforts to rewrite and replace Portage appear as a series of false starts that fail because of poorly planning. Obviously, seen from the point of view of iterative re-engineering (Bianchi et al. 2003), the efforts may be interpreted as activities in an iterative process. Yet, research on iterative re-engineering presupposes an overall plan of action. This is not present for replacing Portage.

We therefore proposed that the process of replacing Portage may be understood as an unfolding negotiation over the scope of the rewrite, the sequence of activities, and the stakeholders to be involved in the process (Østerlie and Jaccheri 2007b). These, then, are the constituents of rewriting requirements:

- Scope
- Sequence of activities
- Stakeholders

For instance, in Østerlie and Jaccheri (2007b) we explore the tension between the need for functional stability for replacing Portage on the one hand, and the various social interests of the Gentoo community on the other hand. The key insight developed in the paper is how the efforts to rewrite Portage unfold within and are part of the continuously emerging context of development and use. Bringing this context of development and use into the analysis brings out the complex and interdependent relations Portage finds itself. These, in turn, shape the requirements for rewriting Portage.

9.2.4 Reflexivity of the unfolding environment

Instead of judging the failure to replace Portage in terms of what the Gentoo developers could or should have done, we therefore sought to understand the dynamics of the efforts to rewrite and replace Portage. Focusing upon the activities, we find that all efforts share a common goal: that of encapsulating the Portage application to reduce its coupling with third-party software. With this goal in mind, the attempts at rewriting Portage take on less of the appearance of false starts and rather appear a process where

the Gentoo developers are actively trying to get to grips with the problems associated with rewritign and replacing Portage.

Rather than false starts, the knowledge gained through the attempts at rewriting Portgage is not lost. Instead, it feess back into the new attempts at rewriting so that an understanding of the problem is built incrementally by trying to rewrite.

The scope, the stakeholders involved, and sequence of activities for replacing Portage differs between the three efforts to rewrite Portage. Yet, the challenge faced by each effort is the same, to strike a balance between the need for keeping the parts to be rewritten stable, and the need for continued adaption. This balance point, however, is continuously negotiated and renegotiated and the strategies for rewriting and replacing Portage have to respond to this.

Through their attempts to rewrite and replace Portage, the Gentoo developers both partake in creating the environment that rewriting Portage is part of, as well as reacting to this environment. There is a codetermination that rewriting and replacing Portage needs to take into consideraton.

This is exemplified in Østerlie and Jaccheri (2007b) when a Gentoo developer explains why next generation Porrage failed: " A rewrite is a MAJOR waste of extremely limited resources … The amount of time it'd take would really drag out on the developers that want new features and simplifications". The Gentoo developers learn from previous attempts, and these attempts enable and restrict further attempts. After next generation Portage, a complete rewrite was no longer possible as the Gentoo developers had learned that this scope would require too much resources.

What we see, is that the process of rewriting Portage is driven by the question "what is going on?" rather than "how to proceed from here?" (Weick 1995). It is a sensemaking process where an understanding of how to proceed with rewriting and replacing Portage emerges in response to an unfolding environment.

9.3. Three defining characteristics of maintaining an integrated system (C3)

This section presents contribution C3. Contribution C3 aggregates the findings reported in C1 and C2 to form an original contribution of this thesis. It does so by identifying three defining characteristics of maintaining an integrated system. As such, it reports on the totality of the research. The contribution is summarized as:

> *Maintaining an integrated system in a community of volunteers is characterized by a scarcity of resources, an emphasis on coalition building, and volatility of stakeholders*

The contribution is offered in response to RQ3:

What are the characteristics of maintaining an integrated system in a distributed community of volunteers?

Each of the three characteristics will now be now discussed in turn.

9.3.1 Scarcity of resources

Scarcity of resources is a key theme recurring in three of the four empirical papers included with this thesis In the context of Gentoo, scarcity of resources is related to two issues:

- Scarcity of manpower
- Scarcity of information

Scarcity of manpower is part of the analysis in all three of the papers. Particularly, in Østerlie and Wang (2007) we illustrate this with the growing gap between reported and resolved problem reports in the defect tracking system database (see Exhibit 9-2 below). Scarcity of manpower for rewriting Portage from scratch is also a recurring explanation for the repeated failures in replacing Portage (Østerlie and Jaccheri 2007b).

Date	New problem reports submitted	Problem reports closed	Open problem reports	Number of Gentoo developers
January 6 2003	269	Not available	1893	102
January 5 2004	837	428	4479	259
January 3 2005	700	390	7877	Not available
January 16 2006	799	447	9083	320

Exhibit 9-2 Weekly debugging workload (Østerlie and Wang 2007)

There is also scarcity of information. While we address this in Østerlie and Wang (Østerlie and Wang 2007), scarcity of information is particularly developed through our use of the term 'ambiguity' in Østerlie and Wang (2006). Ambiguity is to be understood as scarcity of information resources.

As such there are limits to the resources available for performing the many maintenance activities of the Gentoo community. Prioritizing which resources to bear on what problems is therefore important for the Gentoo developers. Determining what problems to spend the limited resources on is open to negotiation. While such negotiations over resources are essentially non-technical, all four empirical papers included with this thesis shows how such negotiations typically unfold in guise of technical issues. Some examples are in place to illustrate this.

Østerlie (2004) reports from the negotiations over introducing a programmable interface for third-party applications to access Portage's package database. Presented as a technical problem of preventing third-party applications from breaking whenever the database schema is modified, the reported incident can also be interpreted as a negotiation over boundaries. The base question being, where is the boundary between third-party applications and Portage to be drawn? Who is responsible for maintaining the third-party applications? The Portage developers, or the third-party application

developers? Being a process of negotiating over the technical boundaries, it is at the same time a negotiation over responsibilities. Summarized: negotiation over boundaries is a negotiation over whose resources are to be spent on resolving which problems.

Similarly, both Østerlie and Wang (2006; 2007) address the issue of negotiating over boundaries. In these papers the technical surface discussion is one of software failures and their causes. At the surface, it is therefore a negotiation over problems and possible solutions. However, negotiations over reported failures are at the same time a process of negotiating over boundaries.

There are two dimensions to the negotiation over boundaries during corrective maintenance. One, the boundary of the failure. The essential technical discussion is whether or not the reported failure really is a failure. Is the failure only located to the reporting user's computer? If the failure is bounded to the user's system, it is labelled user error. This leads us to the second dimension: boundary of responsibilities. User errors are the responsibility of the user. However, if the problem is not merely bounded to the user's system, commonly corroborated through reproducing the problem, it may be the Gentoo developers' responsibility to resolve the problem. Yet, this is conditional on whether the failure is caused by the way the software is integrated. If it is a defect in the software itself, the third-party provider of the software is responsible for resolving it. Again, it is a process of negotiating over which resources to spend on what problems.

9.3.2 Emphasis on coalition building

Emphasis on negotiation shifts the analytical focus towards closure mechanisms. The software maintenance literature, focusing upon problem solving, sees identification of the cause of the problem as closure mechanism. When analysing software maintenance in terms of negotiation, however, the closure mechanisms are rather different. While the conclusion may be technical, the closure mechanism is the building of coalitions.

The importance of building coalitions is discussed most in-depth in Østerlie and Jaccheri (2007b). A central point of that paper is to show how the repeated failures to replace Portage stem from the problem of building sustainable coalitions. We discuss this in terms of balancing between technical and community interests. Building coalitions is a process of translating interests and aligning them with one's own.

We illustrate this process by drawing translation diagrams. Table 9-2 below is an example of such a diagram, illustrating a moment in the process of rewriting Portage where a group of developers, collectively labelled the Portage-ng developers, seek to translate the interests of various actors to their own interest of rewriting Portage as a modular system. They do so by framing problems that the other actors encounter in meeting with their interests, and how they will overcome these problems through the solution proposed by themselves.

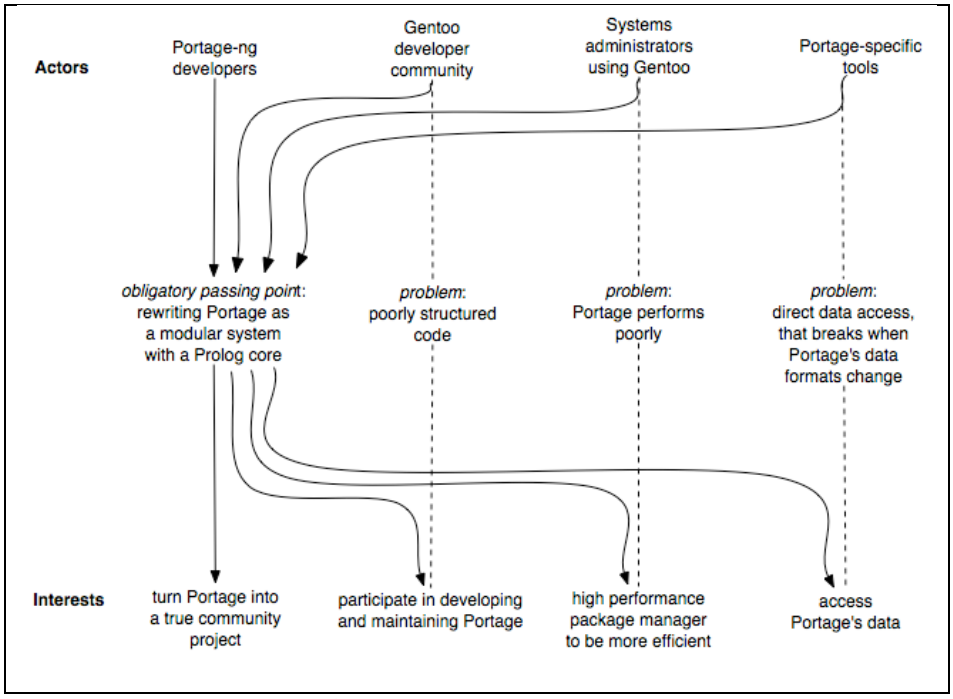


Table 9-2 Translating interests (Østerlie and Jaccheri 2007)

This emphasises the importance of mobilizing both technical and non-technical actors in building coalitions. This is best illustrated in Østerlie (2004). This paper shows that in both technical and non-technical actors are mobilized in building a coalition for developing a programmable interface for third-party applications to access Portage's database. The non-technical actors include the third-party developers and the Portage developers. The technical actors include a set of problem reports, corrupted Portage databases, as well as semi-functioning applications.

Engaging collectively with users in making sense of reported failures is reported as part of contribution C1. This collective engagement can be interpreted as a form of coalition building. Users and developers form temporary coalitions to achieve the joint goal of resolving problem reports. However, these coalitions are precarious. As discussed in Østerlie and Wang (2007) the user needs to present the failure as a likely problem related with the way Gentoo integrates software. This is an effort to establish the problem as an obligatory passing point for reaching the interests of both the Gentoo developers (fault free software) and the user (a non-failing system). The users do so by mobilizing debug texts to strengthen their claim that the reported failure is related with the way Gentoo integrates software. The Gentoo developers, on the other hand, seek to curb the workload of incoming problem reports. They do so by mobilizing data to counter the coalition of technical and non-technical data presented by the user.

On the other hand, the Gentoo developers cannot be too dismissive of incoming problem reports. That users report failures is a key quality assurance mechanism in the

Gentoo community. Users who loose confidence in the Gentoo developers' responsiveness to reported failures are likely to move on to another GNU/Linux distribution. The Gentoo developers therefore need to establish the corrective maintenance process as an obligatory passing point for reaching the users' interest, too. Corrective maintenance can therefore be interpreted as a dual process of building coalitions to resolve particular problem reports, and sustaining coalitions for users to continue reporting failures in the future.

9.3.3 Volatility of participants

Maintenance work in the Gentoo community is characterized by a volatility of participants. This characteristic amplifies the need for coalition building. Volatility of participants in the maintenance activities is most clearly visible in corrective maintenance. This is exemplified in Table 9-3 below.

Range of problem reports submitted	Number of submitters	Number of user submitters	Number of Gentoo developer submitters
1-2	2574	2480	94
3-5	687	629	58
6-10	321	273	48
11-20	145	108	37
21-100	75	39	32
101+	1	1	0
Total	3803	3530	269

Table 9-3 Participants in corrective maintenance in 2002

The table shows that almost 68% of participants in corrective maintenance work report one or two problem reports. 18% submit three to five problem reports, while 14% submitted more than five problem reports during 2002. In practice, this means that there is little sustainability in the process. Rather, coalitions formed to correct reported failures are temporary.

Similarly, the process of rewriting and replacing Portage is marked by volatility of actors. The inability of sustaining coalitions over an extended period of time is a contributing factor to the repeated failures in replacing Portage. The same are the overlapping initiatives for replacing Portage. While overlapping initiatives for replacing Portage may be an expression of a lack of coordination within the community, it is striking that relatively fresh community members undertake many of these initiatives. Coalition building becomes important in these instances in two ways. It shows existing initiatives inability to enrol new stakeholders with their initiative. However, it also shows how new stakeholders need to enrol others with their initiatives to gain entry into the community.

We see two different examples of how this is treated. Portage-C is a one-man effort to rewrite Portage in C with a modular architecture. A young graduate student undertakes it. While addressing both the issue of modularity and performance that are central to the Portage-NG effort, the Portage-C initiative is met with complete silence. It is simply ignored by the Gentoo community. While the Portage-C developer makes an effort to enrol the Gentoo developers with his effort to replace Portage, he fails.

On the other hand the Portage API is met with another form of reaction. This is also an initiative undertaken by a newcomer to the Gentoo community, and is an attempt to establish a programmable interface for third-party software to interface the existing Portage code. While clearly overlapping with the GentoolkitAPI, the Gentoolkit developers abandon their efforts and enrol with the Portage API initiative. While the Portage API developers succeed with enrolling the Gentoolkit developers, they fail to sustain this coalition in the face of yet another newcomer's efforts at rewriting Portage as a multiuser application. Still, unlike the Portage-C developer who fades out of the Gentoo community, the Portage API developer will over time become one of the central Gentoo developers.

9.4. Discussion

This section discusses the contributions presented above. While Section 8.4 evaluates the research in terms of the reflections on the research process, Subsections 9.5.1 and 9.5.2 will evaluate aspects of the reported findings. Subsection 9.5.3 draw two implications of the research for software maintenance research, while 9.5.4 concludes this discussion by revisiting the issue of rigour and relevance.

9.4.1 Problem setting: an essential activity of software maintenance

Having more or less well-defined problems is the basic premise of application software maintenance research. As such, software maintenance has been understood as individual problem solving activities and the management of these activities. In some areas of the research literature, this premise is explicit. In the case of empirical studies of maintenance tasks (Voká et al. 2004) and program comprehension (Vans et al. 1999), for instance, research subjects are provided with clearly defined problems to resolve. Similarly, the research on corrective maintenance procedures take the failure as its point of departure (Zeller 2006). The failure is the equivalent of a more or less well-defined problem. In other areas of the research literature, however, the premise is implied. Much research focuses upon managerial issues such as controlling the change process and planning how to cope with legacy systems (Sneed 1995). However to control change and plan how to cope, there needs to be a problem to be addressed. As such, it is implied that the premise is more or less well-defined problems.

Yet, the empirical findings reported in this chapter show that this premise is problematic. The point of departure of maintenance activities is not well-defined problems. Rather, the essential activity of maintaining an integrated system is a collective process in which situations that are unclear, problematic and puzzling are progressively clarified. In the case of debugging, contribution C1 shows that the Gentoo users and developers engage in a cyclic process of making sense of reported failures. In the case of systems replacement, contribution C2 shows that the problem to be resolved in rewriting Portage is continuously under question and negotiation.

Problem setting is the collective activity of making sense of situations that are unclear, problematic, and puzzling – problem situations. The term embodies a duality: 'setting' is both a noun and verb. The noun is synonymous with 'environment'. 'Problem setting'

can therefore be understood as the environment of the problem. The verb, on the other hand, is defined as 'put, lay, or stand (something) in a specified place or position'. As a verb, 'setting' is the activity of creation. 'Problem setting' is therefore to be understood as the activity making sense of problem situations by constructing the environment of the problem.

Rather than viewing software maintenance as the activities related to discovering causes of problems in the world, problem setting therefore emphasises the activities of constructing problems. Constructing the environment of the problem is an act of *intervention*. Intervening to making sense of problem situations, the situation also changes. The problems to be addressed are not only out there in the world to be discovered, but also immanent in the human activity of constructing them. The *process of problem setting is therefore inseparable from the product emerging from the process: the problem*. This is an ontological shift from an objectified world-view, towards an *in-process view* of objects (problems in this situation) as immanent in human activity.

This use of the term 'problem setting', is slightly differently than originally formulated by Schön (1991). This thesis therefore contributes to theory by elaborating on Schón's original meaning of the 'problem setting'. Schön differentiates between the process and the product of problem setting, stating that "[p]roblem setting is a process in which, interactively, we *name* the things which we will attend and *frame* the context in which we will attend to them". This explanation of problem setting uses the term 'frame' the same way as it is used in Østerlie (2004), as the construction of problems. However, drawing upon the duality of meaning of the term 'setting', 'problem setting' is interpreted as both the process of constructing problems as well as the product of the process; i.e. the environment of the problem.

9.4.2 Transferability of empirical findings

Mockus et al. (2002) raises the question of how software engineering can learn from OSSD? The premise of the question is the idea that OSSD is something different than software engineering as Scacchi (2007) puts it. This premise is problematic. Chapter 4 discusses the historic background of OSS, and how different actors have used the term to position themselves in an otherness relation to dominant market positions within the computing industry. In Østerlie and Jaccheri (2007a) we argue that the software engineering community sees OSSD as a direct threat to its identity: that of a movement of industry and academic actors to professionalize software development. We therefore seek to illustrate how this community uses OSS' otherness position to argue that OSSD is less relevant to software engineering because it is something completely different.

Returning to the original question, I would say that the question is, regardless of whether I have studied software maintenance in a geographically distributed community of volunteers or not: *how do the findings transfer outside this research setting*? This is a valid question, whether we speak of transferability of the findings to similar communities or to in-house software development departments. Another valid question is: *are the findings are limited to system integration?*

These two issues related to transferability of the findings reported in this chapter will be discussed below.

9.4.2.1 Transferability outside the research setting

The empirical findings above are based on a study of a single community of volunteers. I sought to investigate the transferability of the research findings by presenting preliminary results from the reported research to groups of professional system integrators (see Section 8.3).

Presenting for professional system integrators was a deliberate attempt to see whether the results were recognizable outside the context of volunteer software development. At this moment in the research process (March-September 2006), I was still caught up with the idea of OSSD as completely different from software engineering. The choice of professional software developers therefore builds on the logic of opposition: if the results transfer to professional system integration, which I regarded as the complete opposite of OSSD, the transferability of the results were good.

The general feedback at all three sessions was that practitioners recognized my description of corrective software maintenance work from their industrial experience. In particular, the following technical issues of familiarity were emphasised by the practitioners:

- The lack of clearly definable problems, and that the primary work when doing all forms of software maintenance during systems integration, not only corrective maintenance, is to understand what the problem really is
- The lack of traceable defects and the issue of interaction defects is something the practitioners say they are often facing
- The issue of interaction defects were the problem is in the interface between two components or systems
- The practitioners also identified with the situation where it is unclear which information is relevant to understand the problem situation at hand

The feedback substantiates that the findings presented as contribution C1 is to a certain degree transferable outside the research setting.

Participants in the third session pointed out that problem situations were often an occasion for what they called 'organizational politics'. Working in a large corporation, they used the term organizational politics about a form of blame game. The blame game was an effort to limit the work of the department. The departments' limited resources motivated this. In these situations, integrators reluctantly found themselves finding technical data that could be used to place the blame of the problem outside of their own department.

Participants in the other two sessions touched upon similar issues in the passing only. However, scarcity of, or at least limited, resources is well known within the maintenance literature. Indeed, it is one of the central concerns of the literature. Much software maintenance research is motivated by the need to reduce the maintenance burden. With unlimited resources, there would be little need for organizations to reduce

the maintenance burden. Access to and control over scarce resources is therefore important for continued survival in organizations (Morgan 1997). It is therefore likely that most organizations have to deal with the different interests of multiple stakeholders during software maintenance. The answer to the first of the above challenges is therefore that the reported findings are likely to apply in industrial software maintenance, as well.

9.4.2.2 Transferability beyond maintenance of integrated systems

This research finds the problem situation to be an occasion for struggles over limited resources within the organization. So far the problem situation has been identified as a key concern when maintaining integrated systems. Yet, a senior practitioner in the first group session argued that problem setting was the key activity in application software maintenance as well. There is no software maintenance research to corroborate this, however. Yet, some research may seem to indicate that this is the case. Martin and McClure (1983) for instance, observes that maintenance engineers often waste a great deal of effort looking for defects in wrong places. It is likely that some maintenance engineers do waste time looking for defects in the wrong places during debugging. Yet, this observation may also be grounded in the value-based view that debugging should progress from well-defined failures to their resolution. Such an interpretation is likely when considering software engineering as a movement away from trial and error-based approach of the trades towards the professionalization of software engineering through the application of scientific principles (Section 2.2).

The problem situation, however, has been emphasized in related realms of action. Schön (1991) observes that engineers face only a limited number well-defined problems in the daily work. Orr (1996) studies the use of technical manuals in diagnosing faulty of copying machines. He finds that technical manuals can only address problems that present themselves as givens. Yet, users can typically resolve such problems themselves. Maintenance engineers therefore face problem situations, which technical manuals fail to capture. Similarly, Gasser (1986) observes that users device workarounds to errors in information systems. It is therefore likely that problems reported during software maintenance are those users are unable to resolve themselves through workarounds. With basis in Schön and Orr's observations, it is therefore likely to assume that maintenance engineers are faced with problem situations during application software maintenance, too.

The conclusion of the above discussion is therefore that *the political view of software maintenance as a continuous struggle over limited resources is therefore likely to be a fruitful view of software maintenance in general*.

9.4.3 Trustworthiness of findings

How much trust can we put in the findings reported in this chapter?

Research findings never come about in a complete isolation. Rather, as Golden-Biddle and Locke (1993) argues, it resides in the tension between familiarity and uniqueness. On the one hand, reported research findings need to establish a sense of familiarity and relevance to the reader. The text seeks to establish a connection with the disciplinary

background of the readers. This is not sufficient, however, for research findings to be plausible contributions. They also need to provide a sense of distinction and innovation. As such, in establishing the plausibility research findings, the research account needs to provide readers with the means to bridge the gap between the familiar and the distinctive new of the subject matter.

While there is a small, yet growing body of qualitative research on software engineering practice, the research methods and subject matter is still esoteric within the software engineering community. Offering distinctive different results compared to mainstream software engineering research has not been a challenge in this research. Making the research sufficient familiar to the software engineering community, on the other hand, has been a significant challenge. To make the research more familiar to the reader, I deliberately adopted a style of reporting that is common within this research community. In this style of reporting the researcher's personal voice is anonymously present in the text. When directly present the voice is impersonal, indicating personal distance and objectivity. While still I believe the choice of reporting style was a necessity to address this particular community, I also see that it is not entirely unproblematic in terms of the trustworthiness of the research findings. Understanding the researcher's partiality and subjective interpretations are important in evaluating interpretive research.

This problem is further compounded by our limited use of raw qualitative data in the papers. Again, this is a function of my goal to address the software engineering community. The standard document format of software engineering conferences papers leaves little room for qualitative data. Developed for reporting quantitative research, the paper length is usually limited to 8 or 10 pages. I have therefore chosen to limit the use of empirical material in the papers. I could of course have compensated for this by a more liberal use of raw empirical material in this thesis. Yet, this is a paper collection, not a monograph. The purpose of the thesis is to summarize previously reported research, not further elaborate or substantiate it.

To compensate, I have therefore opted to make the presentation of the research process as transparent as possible (Chapter 8). I do so in two ways. First, Sections 8.1 and 8.2 describe in the procedural aspects of the research process in detail. Second, I reflect upon the how the research results have emerged in the triangular interaction with data, theory, and the research subjects (Subsections 8.4.2, 8.4.3, and 8.4.4). In both instances, I have faced the challenge of striking the balance between providing sufficient amount of information to make the process of developing the results transparent without overloading the reader with copious amounts of details.

Using more empirical illustrations may have increased the authenticity of the text. Yet, empirical illustrations alone are not sufficient to determine the trustworthiness of the findings. Methodical transparency and use of empirical data are complimentary. Still, I believe I have struck a sufficiently good balance. As the papers have been accepted in peer reviewed conferences, it seems that the software engineering community agrees that the reported research is trustworthy. Obviously, in the end it is up to the reader to evaluate how well I have succeeded establishing the trustworthiness of the reported research.

9.4.4 Implications for software maintenance research

Two implications for software maintenance research may be drawn from the reported research. First, software maintenance research on maintaining integrated systems need to shift focus away from studying maintenance only as the individual activity of solving well-defined problems towards the collective activity of constructing problems out of materials of situations that are puzzling, troubling, and uncertain. As such, it suggests a shift from individual 'problem solving' toward the collective process of 'problem setting'.

Problem setting emphasises the collective nature of software maintenance work. With this view, software maintenance work is the achieved performance of multiple stakeholders with different interests. Focusing upon the contradictory interests brings out the inherently political aspects of software maintenance work. Such aspects are rarely touched upon in application maintenance research. This research usually assumes that stakeholders shared interests. Whenever the issue of multiple interests is addressed, the task of resolving such conflicts is delegated to, or translated into, an organizational structure with clearly defined roles and responsibilities.

Yet, Østerlie (2004) shows how organizational structures is but one actor to be drawn upon as a closure mechanism. As such, it is argued that the empirical contributions offered by this research contributes towards establishing the need for software maintenance researchers to focus on issues of conflict and differing interests when studying software maintenance. While most maintenance research acknowledges that software is maintained within organizations and that the quality of the software is a function of the quality of the social relations of the organization, few researchers have drawn the consequence of this. The reported research therefore offers a first foray into this area, by identifying some of the social dynamics of maintaining integrated software.

A second implication of the reported research is therefore that existing experimental studies of individual maintenance engineers performing limited maintenance activities with basis in more or less clearly defined problems need to be supplemented with studies that emphasise the contingent, negotiated nature of software maintenance.

9.4.5 Revisiting relevance

Have focused on quantitatively determine whether and in what ways OSSD is different from software engineering, researchers have failed to establish the relevance of OSSD to software engineering in general (Chapter 4). The research reported in this thesis brackets the question of how OSSD differs from software engineering, as this is based on the false premise that OSSD is a homogenous phenomenon (Østerlie and Jaccheri 2007a). The above discussion on generalization indicates that the findings translate outside OSSD. This thesis is therefore offered as an example of how OSSD can be made more relevant to software engineering by studying it as a special case of software maintenance.

The results reported in this chapter also suggest that studies of software engineering practice may supplement scientific rigorous studies to make research more relevant to

practice. Practice studies have so far met limited understanding within software engineering (Robinson et al. 2007). Some of this may be caused by these studies tenuous relationship with the overall goal of software engineering: the professionalization of software development through the application of scientific principles. Practice studies' relationship to this goal is tenuous in two ways.

- Their relevance to practice is unclear, as they do not contribute to the applied science component of software engineering knowledge with improved methods, tools and techniques to improve parts of the software process.
- They rely upon a different underlying theory of the discipline.

There is a trend towards scientism within software engineering. Scientism is a form of methodical monism where only natural scientific methods are considered appropriate for developing valid knowledge. This is probably, as Shaw (2001) observes, a result of a discipline coming of age where we have yet to recognize what our research strategies are and how to establish their results. It is therefore worth noting that such methodical monism reifies the view of OSSD as completely different from software engineering (Østerlie and Jaccheri 2007a). The research reported in this thesis illustrates the importance of capturing the complexity of the social context to better understand what practitioners really do when maintaining integrated systems. This is in contrast with the experimental research on program comprehension that reifies the view of software maintenance as individual, cognitive problem solving (von Mayrhauser and Vans 1995).

The danger of methodical monism is therefore that the natural sciences seek to generate facts that are independent of the social context. However, for practitioners it is exactly this context that is of importance. As such, studies of software engineering practice may supplement existing focus on scientific rigour to capture the complexities of real-life software engineering. By better reflecting upon these complexities, research may become more relevant to practice.

Software engineering has developed a strong theoretical core consisting of more or less standardized terminology and well-established models. It has often-cited references with clearly defined terminology like the IEEE Standard Glossary of Software Engineering Terminology (IEEE 1990). Glass et al. (2004) also observe that software engineering research rarely draw upon reference disciplines. Both are indicators of a strong, coherent theoretical core. As such, software engineering researchers tend to approach the research with *a priori* concepts that are applied to the object of study.

Different research approaches, however, have different views on where and how concepts arise (Alvesson and Deetz 2000). Rather than approaching the fieldwork with concepts to be applied, the reported research exemplifies how familiar concepts may be revisited and supplemented with meaning that emerges from the local research setting. This research revisits and give additional meaning to maintainability (Østerlie and Wang 2006) as well as debugging (Østerlie and Wang 2007) which supplements *a priori* definitions in the research literature.

The danger of theory is that research becomes narrow, caught up in surface phenomena and conventional meanings (Alvesson and Deetz 2000). Such a narrow focus, in turn,

may make research less relevant to practice. Yet, by grounding our understanding of terminology in local meanings, researchers may hope to address concerns that are more relevant to practitioners. Practice studies are well suited for this kind of exploration of the local meaning of familiar concepts. Offering an example of how studies of software engineering practice may explore familiar concepts, the reported research may be considered a response to Osterweil' (2007) question of whether software engineering researchers should explore more.

10. Implications to software maintenance practice

With basis in the empirical findings presented in the previous chapter, this chapter draws implications to software maintenance practice. In particular, it offers a set of recommendations for corrective maintenance and systems replacement. These are offered as contributions C4 and C5. This chapter is therefore organized as follows. First, Section 10.1 offers a set of recommendations for a lenient approach to coping with variability during corrective maintenance. Here it is suggested that rather than pre-empting problems related to variation, it may be more beneficial to address problems as they arise. Section 10.2, offers a set of recommendations for an opportunity-driven approach to systems replacement. Here, it is argued that rather than careful planning requiring stable coalitions over time, it may be more beneficial to emphasise the process of planning. The shift of focus towards planning emphasises a contingent and opportunistic approach to systems replacement.

10.1. Recommendations for a lenient approach to coping with variability during corrective maintenance (C4)

The configuration among individual installations of an integrated system may vary greatly. Variability is therefore a significant concern during maintenance of integrated systems. Existing research recommends technical solutions to control and reduce such variability (Crnkovic and Larsson 2002). This section draws implications of the empirical findings reported earlier for coping with such variability in practice. It offers a set of recommendations for a more lenient approach. With basis in Contribution C3, these recommendations seek to strike a balance between investing scarce resources in pre-empting future problems through increased control of variability, with the effort required to handle such problems as they arise. Hybertson et al.'s (1997) makes a similar argument, offering a set of simple heuristics for modest tracking of third-party software in integrated systems. Building on C1, this section supplements these heuristics with a set of recommendations for organizing the corrective maintenance process to better handle problems of variability when they occur.

10.1.1 Curb up-front investment of effort for coping with variability

An expressed goal of the Gentoo community is that individual systems should be continuously updated rather than reinstalled. This, combined with frequent updates of the third-party packages supported by Gentoo would suggest the potential for immense variability among Gentoo systems. However, Gentoo's response to the issue of variety is diametrically opposite of that recommended by the research literature. While Portage supports dependency handling among packages, controlling variability is not much of a concern to the Gentoo developers. The mechanism for enforcing dependencies between software packages is only used when software depends on a specific version of another package.

With basis in the reported research, however, the following is recommended:

> ***Recommendation 1****:When manpower is scarce, cope with problems related to variability as they arise rather than invest in controlling and limiting variability among installations of an integrated system.*

Increased control is a strategy that seeks to pre-empt anticipated problems. This requires an up-front investment of effort. However, instead of spending a lot of effort to pre-empt *potential* problems, resources are spent on addressing actual problems. The recommendation for a lenient approach to coping with variability resembles the implications Adams (1984) draws for software reliability. Reporting from a study of nine software products over a 10 years period, Adams (1984) observes that most reported problems in large-scale software only occur once. Commenting upon Adams' study, Littlewood (1986) observes that the reason for this observation is related to variation of configuration of individual computers running the software. With basis in the observation that only a fraction of the defects will impact on a large population of users, Adams recommends not spending effort eliminating all defects during testing but rather to address the high-impact failures reported during use. The resemblance between the two strategies is to spend effort when required, rather than to pre-empt potential problems.

This is obviously not an argument against testing software before releasing it. Nor is it an argument not to impose certain degree of control on the variability. Rather, it is an argument against what may be an overzealous attempt to control problems that may not be that difficult to keep in check during maintenance. The Gentoo community complements the lenient approach for coping with variability with a closer integration of users in the corrective maintenance process. This leads to the next recommendation.

10.1.2 Support for remote debugging

The research literature emphasises the need to reproduce software failures on the maintenance engineer's system (Zeller 2006). Detailed and often complex schemas for describing software failures have been proposed to support the reproduction of failures. Configuration management systems are used to limit the variability of application software. Each release of an application is numbered, and the release number corresponds with a set of revisions of the source files in the configuration management

system. Failures therefore relate to a particular release, and the release may therefore be used for tracing the differences in source files since the last failure free release.

Similarly, Carney et al. (2000) stress the importance of having the entire integrated system under configuration management for controlling variability. The result is to release the integrated system as a monolith as it is common for application software. Yet, this approach presumes that the integrator is in complete control of the software being integrated. While this may be possible when developing an integrated system from off-the-shelf components, this is not the case when integrating information systems across organizational boundaries (Hasselbring 2000). Yet, without such control, the integrated system may be difficult, if not even impossible, to debug (Voas 1999).

With the above assumption in mind, the Gentoo community's lenient approach to variability should therefore be problematic in a process where software quality relies more on field-testing and peer review than rigorous testing prior to release (Huntley 2003). The Gentoo developers, however, handle the situation by engaging in a collective process with the users. With basis in the reported research, the following is therefore recommended:

> ***Recommendation 2****: When variability among installations is great, supplement the problem report with alternative communication channels to support remote debugging rather than developing complex classification schemas to support reproduction of failures.*

The problem report seeks to decouple the user experiencing the failure from the maintenance engineer responsible for correcting the reported failure both geographically and temporally. The Gentoo developers seek to interface the failing system both more directly and more indirectly. More directly by engaging with the user. More indirectly by not engaging with the software failure on their own system, but through debug data produced by the user. This is in contrast to the completely decoupled corrective maintenance process espoused by the OSS literature, where someone finds and reports the failure while somebody else corrects it (Huntley 2003).

However, based upon the reported research, simply adding an additional clear text field like Bugzilla's 'Additional comments' may not suffice. The empirical data shows the need for developers and users to communicate more directly. In the case of Gentoo this is handled through IRC channels. Any form of chat-like communication would probably suffice. However, the empirical data show three important features of IRC which are important when debugging:

- Real-time communication facilitates tighter interaction between stakeholders
- The IRC client provides a brief history of previous statements so that people may catch up on threads of discussion
- The possibility of paging particular individuals so they can join the conversation

As software users are seldom co-located with the software developers within the same organization, providing a more direct means of communication between users and

developers is not necessarily limited to geographically distributed software development teams.

10.1.3 Support for bootstrapping

An implicit assumption of existing schemas for classifying software failures is that the reporting user has already understood what the failure is. In the documentation for the Bugzilla defect tracking system (Barnson 2007), for instance, each of the fields for describing software failures are thoroughly presented. However, the part of the problem report that is most used by the Gentoo community, the Additional comments field, is described by a single sentence: "Here you may add additional comments". Yet, it is exactly this field that is of most help during problem situations.

To encourage users to use defect tracking systems, it may be useful to substitute comprehensive classification schemas that seek to describe failures exhaustively with a schema where only a minimum of information about the problem situation is required of the user. Instead, the problem report could facilitate communication between users and developers akin to the way the 'Addition comments' field is used by the Gentoo community.

> ***Recommendation 3****: Provide users with simplified schemas for reporting failures in order to bootstrap the corrective maintenance process.*

The reported research may therefore suggest that a focus on schemas for reporting software failures may be counter constructive when maintaining integrated systems. Because users and developers have to deal with problem situations, it is difficult to determine the value of the predefined fields of the problem report (discussed in 9.1 above). A similar issue arose during one of the practitioner presentations (see 8.3). The practitioners' experience with defect tracking systems was that non-technical users did not use them for reporting software failures. Rather, the users were overwhelmed by the problem reporting schemas, and instead chose to report the problem by phone or e-mail instead.

As such, the research may suggest that the function of the problem report may be limited to bootstrapping the corrective maintenance process.

10.2. Recommendations for an opportunity-driven approach to systems replacement (C5)

The research literature emphasises the need for thorough planning when replacing legacy systems. Sneed (1995, p. 24), for instance, stresses that the success of systems replacement "depends to a great degree on proper planning". Acknowledging the need for an operational system during reengineering, more recent literature proposes incremental approaches to systems replacement (Bianchi et al. 2003). Yet, a well-planned process is still emphasised as a key to success even here (Sneed 2005). This section draws implications for systems replacement practice. With basis in Contribution C2, a set of recommendations for an opportunity-driven approach to systems

replacement is offered. Rather than emphasising the importance of a plan, these recommendations suggest that software developers should focus on the activity of continuous planning throughout the process of replacing the system. These recommendations are offered as Contribution C5 from the reported research.

10.2.1 Long-term goals, short-term plans

Thinking in terms of coalition building, the long-term plan requires a coalition that is stable over time. Yet, the volatility of participants in the process of rewriting Portage made such stable coalitions difficult (see 9.3.3). In contrast to the literature, then, the view of systems replacement offered by contribution C2 emphasises that systems replacement is contingent and depends upon building coalitions. With basis in this view of systems replacement, the following recommendation is offered:

> ***Recommendation 1****: During systems replacement, formulate long-term goals and focus on plans that require temporal rather than long-term coalitions.*

This recommendation is based upon two observations of the systems replacement process:

- Emergent understanding
- Translation of plans into action

First, the issue of emergent understanding. This is an issue developed in Østerlie and Jaccheri (2007b) as well as in both Østerlie and Wang (2006 and 2007). Rather than seeing knowledge as discoverable, these papers argue that understanding a situation requires intervention. Through intervention the situation changes (Weick 1995). As such, knowledge about systems replacement and the process of replacing a system is not something that can be completely grasped beforehand. Rather, understanding is emergent and contingent. As such, it is important to adapt to the changing environment.

Second, the issue of translating plans into actions. The problem with plan-based approaches to systems replacement is that it considers translating the plan into practical action trivial. Acting according to plan requires stable coalitions. The plan will only be translated into practical action when it is in the actors' interest to do so. However, by understanding that systems replacement unfolds as part of a continuously emerging context of development and use, we see the problem with keeping coalitions stable over time. Actors' interests change, and new problematic situations will arise.

10.2.2 Opportunity-driven

Rather than focusing on the plan and its execution, a practical implication of the reported research is to emphasise planning. 'Planning' is proposed here as an activity immanent in the process of replacing systems. Regarding planning as a continuous activity throughout the process of replacing a system seeks to address that understanding is emergent and that plans are negotiated entities. An important aspect of the carefully laid plan is to retain control of the systems replacement effort. Panning, on the other hand, deemphasises the need for controlling the entire replacement process.

Yet, if control of the process is regarded as important in order to succeed, how to succeed without control? This leads to the next recommendation:

> ***Recommendation 2****: Seek smaller, temporary coalitions rather than overall control.*

The practical implication for systems replacement is therefore to seek smaller, temporary coalitions. While replacing the entire software system in one go has clear attractions, the likelihood of succeeding can be quite small. Emphasising the contingent character of systems replacement, planning seeks a shift of focus towards building and sustaining coalitions. Coalitions need to be built and sustained in order to meet the stated goals of systems replacement. This, in turn, emphasises the need for seizing upon opportunities for building new coalitions as they arise within the organization.

The problem that legacy systems' need to adapt even in periods of rewriting, can be related to the sustainability of the effort. Incremental reengineering therefore seeks to make the rewriting steps so small that it is possible for the legacy system to adapt during maintenance.

While a technical problem, the problem of sustainability can also be understood within the context of coalitions. The continued problem with replacing Portage is closely related to the problems of sustaining coalitions. Interpreted in terms of coalition building, complete systems replacement requires a stable coalition over time. Sustainability of coalitions is closely related to stability of actors, but also a stability of actors' interests. In organizations with many actors with different interests, the initial problem will be to enrol a sufficient number of actors in a coalition. Then, sustaining the coalition is an even further problem.

As such, regardless of the technical challenges related to systems replacement, the practical implications of a pluralist perspective on software maintenance organizations is that it can be more realistic to seek temporary coalitions with a few actors that only need to be sustained over a shorter period of time.

11. Conclusions and future work

This thesis concludes the research project titled 'Empirical software engineering and open source software development'. The project has been conducted as part of the Ph.D. programme attended by the Norwegian University of Science and Technology. The thesis has summarized the empirical study undertaken as part of the project. This has been done in three parts. In part one of the thesis, the reported study was situated within the software engineering discipline. The ongoing discussion of relevance and software engineering research is presented in here. Part two of the thesis presented the empirical study performed. In addition to reporting on the research process, the interpretive research approach as well as the research setting is introduced in this part. The final part is dedicated the results of the reported research. Two kinds of results were reported. First, the empirical contributions offered by the research were reported. With basis in the empirical contributions, implications for software maintenance practice are drawn.

The purpose of this chapter is to briefly outline the conclusions of the reported research, discuss limitations of the research, as well as indicate potential avenues for future work.

11.1. Conclusions

The goal of the reported research has been to inform software engineering research. This is a response to the research community's worries over lack of relevance to practice (see Chapter 2). The reported research is based on the view that we do not fail to inform practice because our research lacks credibility. Rather, we as a research community fail to inform practice because we know too little about practice to study issues relevant to practitioners. To meet our collective goal of informing practice, the software engineering research community first needs to be informed by practice. To this end, I have conducted a study of software maintenance practice. The study was conducted in the context of a community of volunteer software integrators. Three research questions were posed (Section 1.3).

Research questions 1 and 2 dealt with software maintenance as knowledge-intensive work. They were concerned with how system integrators in a geographically distributed community of volunteers build knowledge for two kinds of software maintenance

activities: corrective maintenance and software replacement. Whereas most software maintenance research is based on the premise that maintenance activities follow from more or less clearly defined problems, I found problem setting to be an essential activity of maintaining an integrated system. Problem setting is the collective process in which situations that are unclear, problematic, and puzzling are progressively clarified.

The shift from problem solving to problem setting broadens the scope of software maintenance activities. Problem solving is concerned with software maintenance as an individual cognitive activity. Existing maintenance research has therefore focused upon understanding the cognitive mechanics of problem solving and developing tools for supporting individual developers. Yet, problem setting broadens the scope to include both the social and technical processes involved when maintaining an integrated system.

Building upon this observation, I answered *research question 3*. This research question was concerned with the characteristics of large-scale software maintenance work in a geographically distributed community of volunteers. With basis in the social and technical processes identified in response to research questions 1 and 2, I found problem setting to be a process where multiple stakeholders with different interests continuously negotiate over problems and their solutions. I call this multilateral software maintenance. Maintaining an integrated system in a community of volunteers is therefore characterized by a scarcity of resources, an emphasis on coalition building, and volatility of stakeholders. Focusing upon scarcity of resources and contradictory interests brings out the inherently political aspects of multilateral software maintenance.

The following conclusions can therefore be drawn with basis in the reported research.

A conclusion for software maintenance is that researchers need to acknowledge the multilateral character of systems integration. This means that the basic premise of application software maintenance – that a single team or organization is in control of the development trajectory of software (Banker et al. 1993) – is no longer tenable. Rather, multilateral software maintenance means that no single stakeholder is in control of the entire integrated system. Instead, multiple stakeholders with different, sometimes conflicting interests are in control of the development trajectory of different parts of the integrated system.

As far as existing research acknowledges that multiple and conflicting interests may exist during software maintenance, the methods proposed invariably leaves it to a single stakeholder to resolve such conflicts. Yet, it is the very lack of such central authority that characterizes multilateral software maintenance. Instead of relying upon maintenance methods that assumes that there is some form of central control, multilateral software maintenance calls for pragmatic strategies based on building coalitions. This is exemplified in contributions C4 and C5.

A conclusion for software engineering is that we as a community have mostly missed out on the opportunity for learning from the experiences of OSSD so far. This thesis proposes that software engineering is a community of industry and academic actors with the shared goal of professionalizing software development. Learning from experience to build a body of professional knowledge is an important part of professionalization. Yet,

as we show in Østerlie and Jaccheri (2007a) software engineering researchers continue to treat OSSD as different. In so doing, we as a community are missing out on an important opportunity for learning that may offer significant contributions to the professional body of knowledge we are collectively building. By continuing to quantify artefacts of OSSD, we fail to move beyond the dichotomy between OSSD and software engineering. Moving beyond this dichotomy is important for us to start learning from OSSD. To do so, software engineering researchers also need to study how OSSD is developed and maintained in practice.

The need to study practice is not limited to software engineering research on OSSD, but applies to software engineering research in general. This research shows that the concept of problem solving only partially addresses the activities of maintaining an integrated system. Yet, software maintenance as mainly an individual problem solving activity remains the basic premise of much research. The reported research empirically demonstrates that the individual problem solving is only part of the collective activities of problem setting. As such, while existing experimental research on software maintenance is scientifically rigorous, it is in the case of software integration based on the mistaken premise of more or less well-defined problems. To develop research that is relevant for practice, the theory applied in scientifically rigorous experiments needs to be calibrated with research on actual software development practice.

11.2. Limitations

This thesis reports from a study of software maintenance work in a single geographically distributed community of volunteers. While this research strategy has the potential of developing in-depth data, it also faces two potential limitations:

- The results may be inapplicable outside the context of the particular case
- With basis in a community of volunteers, the results may have little relevance to commercial software development which is the main concern of software engineering

While I have made efforts to test the transferability of the reported research to industrial system integration (see 8.3 and 9.4.2), the results could have been made more credible by doing a comparative study of system integration in a commercial organization and a geographically distributed community of volunteers. Although predominantly descriptive, this study proposes 'problem setting' as a way of conceptualizing the maintenance of integrated systems (see 9.4.1). A comparative study would support better development of this concept. However, time and resources did not permit that such a comparative study to be undertaken.

Practically unlimited access to data is an advantage of studying geographically distributed communities of volunteers. This gives immense amounts of data. Yet, for me there was also a significant limitation to doing such research: I had no immediate access to those I studied. Although being available through e-mail and IRC, I found in-depth communication with the research participants limited. While I did discuss drafts of some papers to selected community members (see 8.4.4), I think the results would

have benefited from arranging group sessions with selected Gentoo developers similar to those held for professional system integrators (see 8.3). With the possibility of commenting both upon concrete events and situations as well as my interpretations of these, I believe such group sessions could have provided important feedback to improve the results of this study.

11.3. Future work

The motivation of the reported research is that there is very little research on maintenance on integrated systems. While there is a clear shift of focus towards software integration, the software maintenance research community does not seem to have fully grasped the implications of such a shift for their object of study. As such, studying the maintenance of integrated systems appears a fruitful avenue for further research.

Offering a view of software maintenance work where multiple stakeholders with different interests continuously negotiate over problems and their solutions, this thesis offers an outline of a political perspective on software maintenance. Following a turn towards studying maintenance of integrated systems, further development of such a political perspective should be particularly relevant to software maintenance research. Existing research is based on an implicit understanding of the social as harmonic. This may not be an altogether misleading assumption when studying application software maintenance where a single team or organization is in complete control of the software. However, in the context of integrated systems where no one organization or actor is in complete control, a political perspective's emphasis on multiple stakeholders with different interests seems particularly fruitful.

Furthermore, while the reported research does study maintenance in the context of development and use, the implications of this relationship remains unresolved by the reported research. Yet, it is implied that there is a reciprocal relationship between the two. In the future, it would be particularly interesting to study this relationship even further. In developing a political perspective on software maintenance the interaction between use and development is particularly important, as individual and groups of users are important stakeholders in the maintenance process.

The context of the reported study is a distributed community of volunteers. In the future, it would be interesting to explore a political perspective on software maintenance in the context of a large commercial organization. Whereas the Gentoo community's organizational structures are not particularly strong, the theoretical perspective offered by Østerlie (2004) to draw hierarchies into the analysis of the process of problem setting as merely yet another actor would be particularly interesting in the context of formal hierarchies.

Some practitioners participating in the group sessions (see Section 8.3) indicated that making sense of problem situations is also a central activity in application software maintenance. It may therefore be interesting to test whether the premise of more ore less clearly defined problems is valid in application software maintenance, too.

12. Glossary

Application library	A collection of subroutines that multiple applications use.
Application software	A set of software modules performing a coherent set of tasks in support of a given organizational unit and maintained by a single team.
Business-critical	Software whose failure may result in the failure of the business using the system.
Component	A unit of code that integrators can combine with other components and integrate into a system in a predictable way
Debugging	The activity of diagnosing problematic situations related to failing software.
ebuild	Installation script used by Portage to integrate software packages with Gentoo systems.
Gentoo system	A computer using Portage to integrated third-party OSS with its local file system.
GNU	GNU is a recursive synonym for 'GNU is not Unix', and is used as the brand for the Free Software Foundation's Unix-like operating system.
GNU/Linux distribution	A collection of software applications and libraries bundled together with the Linux operating system kernel. It is called a GNU/Linux distribution as much of the core software is developed by the GNU project.
Integrated system	A software system composed of black-boxed software. The black-boxed software may range from software components to enterprise information systems.
Kernel	Short for *operating system kernel*.
Legacy system	Software system that is expensive to maintain, but still operational because it is business-critical.
Open source software	Software released under a license compliant with the Open Source Definition.
Operating system kernel	The kernel manages system resources, and communicated between the software and the hardware.
Optional	Short for *optional feature*.
Optional feature	A global configuration option in Portage that enables optional features across individual ebuilds. IMAP support is an example of such an optional feature.

Package	Short for *software package*.
Package manager	A software application that integrates software with a local computer's file system.
portdir	Portage's package database.
Portage	Gentoo's package manager.
Problem report	A standardized schema for reporting software failures
Problem setting	The collective process where problematic situations are progressively clarified.
Problematic situation	A situation that is puzzling, troubling, and uncertain. It is a situation where it is unclear what the problem really is.
Process data	Data of events, activities, and the sequence of these.
Process theory	Theory that seeks to conceptualize events, activities, and choices ordered over time and to detect patterns among them. The purpose is to explain the outcome and mechanics of these activities and events.
Runtime libraries	Libraries that handle the low-level details of passing information between the kernel and the application software layer.
Software integration	The process of developing integrated systems.
Software package	Third-party software that can be integrated with a computer's file system.
System calls	Part of the operating system kernel that provides services for to request services from the kernel.
Unix-like	An operating system that behaves in a similar manner to a Unix system, but does not necessarily comply with POSIX.
/var/db	The database Portage stores information about the packages that have been integrated with the computer's file system.
virtual package	Functionality that may be provided by different packages. The functionality of the Java virtual machine, for instance, may be provided by Sun's Java VM as well as IBM's Java VM.

References

Adams, E.N. "Optimizing Preventive Service of Software Products," *IBM Journal of Research and Developmen* (28:1), January 1984, pp. 2-14.

Alvesson, M., and Deetz, S.A. *Doing Critical Management Research*, SAGE Publications, London, UK, 2000, p. 232.

Araki, K., Furukawa, Z., and Cheng, J. "A General Framework for Debugging," *IEEE Software* (8:3), May 1991, pp. 14-20.

Avgerou, C. "Doing Critical Research in Information Systems: Some Further Thoughts," *Information Systems Journal* (15:2), April 2005, pp. 103-109.

Banker, R.D., Datar, S.M., Kemerer, C.F., and Zweig, D. "Software Complexity and Maintenance Costs," *Communications of the ACM* (36:11), November 1993, pp. 81-94.

Bansler, J., and Bødker, K. "A Reappraisal of Structured Analysis: Design in an Organizational Context," *ACM Transactions on Information Systems* (11:2), April 1993, pp. 165-193.

Barnson, M.P. "The Bugzilla Guide - 3.1 Development Release," http://www.bugzilla.org/docs/tip/html/, 2007. Last accessed: 21 March 2007.

Basili, V. "The Experimental Paradigm in Software Engineering," in: *Experimental Software Engineering Issues: Critical Assessment and Future Directions,* B.A. Kitchenham, N. Fenton, H.D. Rombach, W.W. Agresti and A. von Mayrhauser (Eds.), Springer Verlag, Heidelberg, 1993, pp. 1-12.

Basili, V.R., and Harrison, W. "Editorial," *Empirical Software Engineering* (1:1), January 1996, pp. 5-10.

Basili, V.R., and Perricone, B.T. "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM* (27:1), January 1984, pp. 42-52.

Basili, V.R., Selby, R.W., and Hutchens, D.H. "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering* (12:7), July 1986, pp. 733-743.

Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, Addison Wesley, Boston, Mass., 2003, p. 560.

Behlendorf, B. "Open Source as a Business Strategy," in: *Open Sources: Voices from the Open Source Revolution,* C. DiBona, S. Ockman and M. Stone (Eds.), O'Reilly & Associates, Sebastapol, Calif., 1999, pp. 149-170.

Belady, L.A., and Lehman, M.M. "A Model of Large Program Development," *IBM Systems Journal* (15:3), 1976, pp. 225-252.

Belady, L.A., and Lehman, M.M. "The Characteristics of Large Systems," in: *Research Directions in Software Technology,* P. Wegner (Ed.), The MIT Press, Cambridge, Mass., 1978, pp. 108-138.

Bennet, K. "Legacy Systems: Coping with Success," *IEEE Software* (12:1), January 1995, pp. 19-23.

Bergqvist, M., and Ljungberg, J. "The Power of Gifts: Organizing Social Relationships in Open Source Communities," *Information Systems Journal* (11:4), October 2001, pp. 305-320.

Berntsen, K., Munkvold, G., and Østerlie, T. "Community of Practice Versus Practice of the Community: Knowing in Collaborative Work," *The ICFAI Journal of Knowledge Management* (II:4), December 2004, pp. 7-20.

Bianchi, A., Caivano, D., Marengo, V., and Visaggio, G. "Iterative Reengineering of Legacy Systems," *IEEE Transactions on Software Engineering* (29:3), March 2003, pp. 225-241.

Bisbal, J., Lawless, D., Wu, B., and Grimson, J. "Legacy Information Systems: Issues and Directions," *IEEE Software* (16:5), September 1999, pp. 103-111.

Bjerknes, G., and Bratteteig, T. "User Participation and Democracy: A Discussion of Scandinavian Research on Systems Development," *Scandinavian Journal of Information Systems* (7:1), 1995, pp. 73-98.

Bloor, D. *Knowledge and Social Imagery*, University of New South Wales, Chicago, Ill., 1976, p. 156.

Boehm, B.W. *Characteristics of Software Quality*, Elsevier, Amsterdam, North-Holland, 1978.

Boehm, B.W. "A View of 20th and 21st Century Software Engineering", in *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, May 20-28, 2006a, pp. 12-29.

Boehm, B.W. "Some Future Trends and Implications for Systems and Software Engineering Processes," *Systems Engineering* (9:1), January 2006b, pp. 1-19.

Boehm, B.W., and Abts, C. "Cots Integration: Plug and Pray?," *IEEE Computer* (32:1), January 1999, pp. 135-138.

Boland, R.J. "Phenomenology: A Preferred Approach to Research in Information Systems," in: *Research Methods in Information Systesm: Ifip Wp 8.2 Colloquium Proceedings,* E. Mumford, R. Hirschheim, G. Fitzgerald and A.T. Wood-Harper (Eds.), North-Holland Publishing Co., Amsterdam, Netherlands, 1985, pp. 193-201.

Bonaccorsi, A., and Rossi, C. "Why Open Source Software Can Succeed," *Research Policy* (32:7), July 2003, pp. 1243-1258.

Brodie, M.L., and Stonebraker, M. *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*, Morgan Kaufmann Publishers, San Francisco, Calif., 1995, p. 210.

Brooks, R. "Towards a Theory of the Comprehension of Computer Programs," *International Journal of Man-Machine Studies* (18:6), June 1983, pp. 543-554.

Brown, J.S., and Duguid, P. "Organizational Learning and Communities-of-Practice: Towards a Unified View of Work, Learning, and Innovation," *Organization Science* (2:1), February 1991, pp. 40-57.

Callon, M. "Techno-Economic Networks and Irreversibility," in: *A Sociology of Monsters: Essays of Power, Technology, and Domination,* J. Law (Ed.), Routledge, London, UK, 1991, pp. 132-164.

Callon, M. "Some Elements of a Sociology of Translation: Domestication of the Scallops and the Fishermen of St. Brieuc Bay," in: *The Science Studies Reader,* M. Biaglio (Ed.), Routledge, New York, NY, 1999, pp. 67-83.

Callon, M., and Law, J. "On Interests and Their Transformation: Enrolment and Counter-Enrolment," *Social Studies of Science* (4:12), November 1982, pp. 615-625.

Calzolari, F., Tonella, P., and Antoniol, G. "Dynamic Model for Maintenance and Testing Effort", in *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, Bethesda, Maryland, November 16-20, 1998, pp. 104-112.

Canning, R.G. "That Maintenance 'Iceberg'," *EDP Analyzer* (10:10), October 1972, pp. 1-14.

Capra, E., Francalanci, C., and Merlo, F. "The Economics of Open Source Software: An Empirical Analysis of Maintenance Costs", in *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM'07)*, Paris, France, October 2-5, 2007, pp. 395-404.

Carney, D., Hissam, S.A., and Plakosh, D. "Complex Cots-Based Software Systems: Practical Steps for Their Maintenance," *Journal of Software Maintenance: Research and Practice* (12:6), November-December 2000, pp. 357-376.

Cavano, J.P., and McCall, J.A. "A Framework for the Measurement of Software Quality", in *Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues*, 1978, pp. 133-139.

Chapin, N., Hale, J.E., Khan, K.M., Ramil, J.F., and Tan, W.-G. "Types of Software Evolution and Software Maintenance," *Journal of Software Maintenance: Research and Practice* (13:1), January-February 2001, pp. 3-30.

Ciborra, C.U., and Lanzara, G.F. "Formative Contexts and Information Technology: Understanding the Dynamics of Innovation in Organizations," *Accounting, Management and Information Technologies* (4:2), April-June 1994, pp. 61-86.

Cleve, H., and Zeller, A. "Locating Causes of Program Failures", in *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, St. Louis, MO, May 15-21, 2005, pp. 342-351.

Cockburn, A. *Agile Software Development*, Pearson Education Limited, Boston, MA, 2002, p. 278.

Conradi, R., and Wang, A.I. "Introduction," in: *Empirical Methods and Studies in Software Engineering,* R. Conradi and A.I. Wang (Eds.), Springer Verlag, Berlin, 2003, pp. 1-6.

Corritore, C.L., and Wiedenbeck, S. "Mental Representations of Expert Procedural and Object-Oriented Programmers in a Software Maintenance Task," *International Journal of Human-Computer Studies* (50:1), January 1999, pp. 61-83.

Corritore, C.L., and Wiedenbeck, S. "An Exploratory Study of Program Comprehension Strategies of Procedural and Object-Oriented Programmers," *International Journal of Human-Computer Studies* (53:1), January 2001, pp. 1-23.
Crnkovic, I., and Larsson, M. "Challenges of Component-Based Development," *Journal of Systems and Software* (61:3), April 2002, pp. 201-212.
Crowston, K., and Howison, J. "The Social Structure of Free and Open Source Software Development," *First Monday* (10:2), February 2005.
Cruz, D., Wieland, T., and Ziegler, A. "Evaluation Criteria for Free/Open Source Software Products Based on Project Analysis," *Software Process: Improvement and Practice* (11:2), March-April 2006, pp. 107-122.
DiBona, C., Ockman, S., and Stone, M. (Eds.) *Open Sources: Voices from the Open Source Revolution*. O'Reilly, Sebastapol, Calif., 1999, p. 272.
Dinh-Trong, T.T., and Bieman, J.M. "The Freebsd Project: A Replication Case Study of Open Source Development," *IEEE Transactions on Software Engineering* (31:6), June 2005, pp. 481-494.
Dvorak, J. "Conceptual Entropy and Its Effect on Class Hierarchies," *IEEE Computer* (27:6), June 1994, pp. 59-63.
Dybå, T., Kampenes, V.B., and Sjøberg, D.I.K. "A Systematic Review of Statistical Power in Software Engineering Experiments," *Information and Software Technology* (48:8), August 2006, pp. 745-755.
Edwards, H.M., Mallalieu, G.M., and Thompson, J.B. "Some Insights into the Maintenance of Legacy Systems within Small Manufacturing and Distribution Organization in the Uk", in *Proceedings of the 23rd International Computer Software and Applications Conference (COMPSAC'99)*, Phoenix, Ariz., October 27-29, 1999, pp. 13-20.
Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., and Mockus, A. "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Transactions on Software Engineering* (27:1), January 2001, pp. 1-12.
Emerson, R.M., Fretz, R.I., and Shaw, L.L. *Writing Ethnographic Fieldnotes*, The University of Chicago Press, Chicago, Ill., 1995, p. 254.
Endres, A. "An Analysis of Errors and Their Causes in System Programs," *ACM SIGPLAN Notices* (10:6), June 1975, pp. 327-336.
Erenkrantz, J.R. "Release Management within Open Source Projects", in *Proceedings of the 3rd Workshop on Open Source Software Engineering, co-located with the 25th International Conference on Software Engineering*, Portland, Ore., May 3, 2003, pp. 51-55.
Feller, J., Finnegan, P., Kelly, D., and MacNamara, M. "Developing Open Source Software: A Community-Based Analysis of Research," in: *Social Inclusion: Societal and Organizational Implications for Information Systems,* E.M. Trauth, D. Howcroft, T. Buttler, B. Fitzgerald and J.I. DeGross (Eds.), Springer, Boston, Mass., 2006, pp. 261-278.
Feller, J., and Fitzgerald, B. *Understanding Open Source Software Development*, Addison-Wesley, London, UK, 2002, p. 211.
Fenton, N. "How Effective Are Software Engineering Methods?," *Journal of Systems and Software* (22:2), November 1993, pp. 141-146.
Fenton, N. "Software Measurement: A Necessary Scientific Basis," *IEEE Transactions on Software Engineering* (20:3), March 1994, pp. 199-206.

Fenton, N.E., and Neil, M. "A Critique of Software Defect Prediction Models," *IEEE Transactions on Software Engineering* (25:5), May 1999, pp. 675-689.
Fenton, N.E., and Pfleeger, S.L. *Software Metrics: A Rigiorous & Practical Approach*, PWS Publishing Company, Boston, Mass., 1997, p. 638.
Fetterman, D.M. *Ethnography: Step by Step*, (Second ed.), SAGE Publications, Thousand Oaks, Calif., 1998, p. 166.
Finkelstein, A., and Kramer, J. "Software Engineering: A Roadmap", in *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, Limerick, Ireland, June 4-11, 2000, pp. 5-22.
Fitzgerald, B. "The Transformation of Open Source Software," *MIS Quarterly* (30:3), September 2006, pp. 587-598.
Fitzgerald, B., and Kenny, T. "Developing an Information Systems Infrastructure with Open Source Software," *IEEE Software* (21:1), January-February 2004, pp. 50-55.
Friedman, A.L., and Cornford, D.S. *Computer Systems Development: History, Organization and Implementation*, John Wiley & Sons, Inc., London, UK, 1989, p. 438.
Gacek, C., and Arief, B. "The Many Meanings of Open Source," *IEEE Software* (21:1), January 2004, pp. 34-40.
Gasser, L. "The Integration of Computing and Routine Work," *ACM Transactions on Office Information Systems* (4:3), July 1986, pp. 205-225.
Gibson, V.R., and Senn, J.A. "System Structure and Software Maintenance Performance," *Communications of the ACM* (32:3), March 1989, pp. 347-358.
Glass, R.L. "The Software-Research Crisis," *IEEE Software* (11:6), November 1994, pp. 42-47.
Glass, R.L., Ramesh, V., and Vessey, I. "An Analysis of Research in Computing Disciplines," *Communications of the ACM* (47:6), June 2004, pp. 89-94.
Glass, R.L., Vessey, I., and Ramesh, V. "Research in Software Engineering: An Analysis of the Literature," *Information and Software Technology* (44:8), June 2002, pp. 491-506.
Golden-Biddle, K., and Locke, K. "Appealing Work: An Investigation of How Ethnographic Texts Convince," *Organization Science* (4:4), November 1993, pp. 595-616.
Hannemyr, G. "Technology and Pleasure: Considering Hacking Constructive," *First Monday* (4:2), April 1999.
Hasselbring, W. "Information Systems Integration," *Communications of the ACM* (46:6), June 2000, pp. 33-38.
Highsmith, J. *Agile Software Development Ecosystems*, Addison Wesley, Boston, Mass., 2002, p. 404.
Himanen, P. *The Hacker Ethic: And the Spirit of the Information Age*, Random House, New York, NY, 2001, p. 232.
Howcroft, D., and Wilson, M. "Paradoxes of Participatory Practices: The Janus Role of the Systems Developer," *Information and Organization* (13:1), January 2003, pp. 1-24.
Huntley, C.L. "Organizational Learning in Open-Source Software Projects: An Analysis of Debugging Data," *IEEE Transactions on Engineering Management* (50:4), November 2003, pp. 485-493.

Hybertson, D.W., Ta, D., and Thomas, W.M. "Maintenance of Cots-Intensive Software Systems," *Journal of Software Maintenance: Research and Practice* (9:4), December 1997, pp. 203-216.

IEEE "Ieee Standard Glossary of Software Engineering Terminology," 610.12-1990, IEEE.

Jaccheri, L., and Østerlie, T. "Empirical Software Engineering Education", in *Proceedings of the 11th Norwegian Conference on Information Systems (NokobIT'04)*, Stavanger, Norway, November 29-December 1, 2004, pp. 242-249.

Jaccheri, L., and Østerlie, T. "Can We Teach Empirical Software Engineering?", in *Proceedings of the 11th IEEE International Symposium on Software Metrics (Metrics 2005)*, Como, Italy, September 19-22, 2005, pp. CD-ROM.

Jaccheri, L., and Østerlie, T. "Open Source Software: A Source of Possibilities for Software Engineering Education and Empirical Software Engineering", in *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development, co-located with the 29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, Minnesota, May 20-26, 2007, pp. 1-5.

Kirk, J., and Miller, M.L. *Reliability and Validity in Qualitative Research*, SAGE Publications, Thousand Oaks, CA, 1986.

Kitchenham, B.A., Dybå, T., and Jørgensen, M. "Evidence-Based Software Engineering," In Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on, 2004, pp. 273-281.

Kitchenham, B.A., Travassos, G.H., von Mayrhauser, A., Niessink, F., Scheiderwind, N.F., Singer, J., Takada, S., Vehvilainen, R., and Yang, H. "Towards and Ontology of Software Maintenance," *Journal of Software Maintenance: Research and Practice* (11:6), November-December 1999, pp. 365-389.

Klein, H.K., and Myers, M.D. "A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems," *MIS Quarterly* (23:1), March 1999, pp. 67-93.

Knight, J.C., and Leveson, N.G. "Should Software Engineers Be Licensed?," *Communications of the ACM* (45:11), November 2002, pp. 87-90.

Lam, W., and Shankararaman, V. "An Enterprise Integration Methodology," *IT Professional* (6:2), March-April 2004, pp. 40-48.

Langley, A. "Strategies for Theorizing from Process Data," *The Academy of Management Review* (24:4), October 1999, pp. 691-710.

Latour, B. *Science in Action*, Harvard University Press, Cambridge, Mass., 1987, p. 274.

Latour, B., and Woolgar, S. *Laboratory Life: The Construction of Scientific Facts*, SAGE Publications, New York, NY, 1979, p. 294.

Lave, J., and Wenger, E. *Situated Learning: Legitimate Peripheral Participation*, Cambridge University Press, New York, NY, 1991, p. 138.

Law, J. "Notes on the Theory of the Actor-Network: Ordering, Strategy, and Heterogeneity," *Systemic Practice and Action Research* (5:4), August 1992, pp. 379-393.

Lee, A.S. "Electronic Mail as a Medium for Rich Communication: An Empirical Investigation Using Hermeneutic Interpretation," *MIS Quarterly* (18:2), June 1994, pp. 143-157.
Lee, A.S., and Baskerville, R.L. "Generalizing Generalizability in Information Systems Research," *Information Systems Research* (14:3), September 2003, pp. 221-243.
Lehman, M.M. "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle," *Journal of Systems and Software* (1), 1979, pp. 213-221.
Lehman, M.M. "Programs, Life Cycles, and Laws of Software Evolution," *Proceedings of IEEE* (68:9), September 1980, pp. 1060-1076.
Lethbridge, T.C., Sim, S.E., and Singer, J. "Studying Software Engineers: Data Collection Techniques for Software Field Studies," *Empirical Software Engineering* (10:3), July 2005, pp. 311-341.
Letovsky, S. "Cognitive Processes in Program Comprehension", in *Proceedings of the First Workshop on Empirical Studies of Programmers*, Washington, DC, June 5-6, 1986, pp. 59-79.
Levy, S. *Hackers: Heroes of the Computer Revolution*, Penguin Books, London, UK, 1984, p. 455.
Li, J., Conradi, R., Slyngstad, O.P.N., Bunse, C., Torchiano, M., and Morisio, M. "An Empirical Study on Decision Making in Off-the-Shelf Component-Based Development", in *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, Shanghai, China, May 20-28, 2006, pp. 897-900.
Li, J., Conradi, R., Slyngstad, O.P.N., Torchiano, M., Morisio, M., and Bunse, C. "A State-of-the-Practice Survey of Risk Management in Development of Off-the-Shelf Software Components," in: *IEEE Transactions on Software Engineering*, 2007, p. 35.
Li, P., Herbsleb, J.D., and Shaw, M. "Finding Predictors of Field Defects for Open Source Software Systems in Commonly Available Data Sources: A Case Study of Openbsd", in *Proceedings of the The 11th IEEE International Software Metrics Symposium (Metrics'05)*, Como, Italy, September 19-22, 2005.
Lientz, B.P., Swanson, E.B., and Tompkins, G.E. "Characteristics of Application Software Maintenance," *Communications of the ACM* (21:6), June 1978, pp. 466-471.
Littlewood, B. "Why Did Ed Adams See So Many Small Bugs?," *Software Reliability and Metrics Newsletter* (1:4), 1986, pp. 31-34.
Ljungberg, J. "Open Source Movements as a Model for Organizing," *European Journal of Information Systems* (9), 2000, pp. 208-216.
Ludewig, J. "Software Engineering - Why It Did Not Work", in *Proceedings of the Dagstuhl Seminar on History of Software Engineering*, Schloss Dagstuhl, 1996, pp. 25-27.
Madanmohan, T.R., and Rahul, D. "Open Source Reuse in Commercial Firms," *IEEE Software* (21:6), November/December 2004, pp. 62-69.
Martin, J., and McClure, C. *Software Maintenance: The Problem and Its Solutions*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1983, p. 472.
Mens, T., and Tourwe, T. "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering* (30:2), February 2004, pp. 126-139.
Messerschmidt, D.G. "Back to the User [Open Source]," *IEEE Software* (21:1), January 2004, pp. 89-91.

Michlmayr, M., Hunt, F., and Probert, D. "Quality Practices and Problems in Free Software Projects," In First International Conference on Open Source Systesm, Genova, Italy, 2005, pp. 24-28.
Misherghi, G., and Su, Z. "Hdd: Hierarchical Delta Debugging", in *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, May 20-28, 2006, pp. 142-151.
Mockerjee, R. "Maintaining Enterprise Software Applications," *Communications of the ACM* (48:11), November 2005, pp. 75-79.
Mockus, A., Fielding, R.T., and Herbsleb, J.D. "Two Case Studies of Open Source Software Development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology* (11:3), Jul. 2002, pp. 309-346.
Monteiro, E. "Actor-Network Theory and Information Infrastructure," in: *From Control to Drift: The Dynamics of Corporate Information Infrastructures,* C.U. Ciborra and O. Hanseth (Eds.), Oxford University Press, Oxford, UK, 2000, pp. 71-83.
Moody, G. *Rebel Code: Inside Linux and the Open Source Revolution*, Perseus Publishing, Cambridge, MA, 2001, p. 342.
Morgan, G. *Images of Organization*, SAGE Publications, Thousand Oaks, CA, 1997, p. 485.
Mumford, E. "The Story of Socio-Technical Design: Reflections on Its Success, Failures and Potential," *Information Systems Journal* (16:4), October 2006, pp. 317-342.
Nandhakumar, J., and Jones, M. "Too Close for Comfort? Distance and Engagement in Interpretive Information Systems Research," *Information Systems Journal* (7:2), April 1997, pp. 109-131.
Noble, D.F. *America by Design: Science, Technology and the Rise of Corporate Capitalism*, Alfred A. Knopf, New York, NY, 1977, p. 384.
Oman, P.W., and Cook, C.R. "Typographic Style Is More Than Cosmetics," *Communications of the ACM* (33:5), May 1990, pp. 506-520.
Orlikowski, W.J. "Improvising Organizational Transformation over Time: A Situated Change Perspective," *Information Systems Research* (7:1), March 1996, pp. 63-92.
Orlikowski, W.J. "Knowing in Practice: Enacting a Collective Capacity in Distributed Organizing," *Organization Science* (13:3), May-June 2002, pp. 249-273.
Orlikowski, W.J., and Baroudi, J.J. "Studying Information Technology in Organizations: Research Approaches and Assumptions," *Information Systems Research* (2:1), 1991, pp. 1-29.
Orr, J.E. *Talking About Machines: An Ethnography of a Modern Job*, Cornell University Press, Ithaca, NY, 1996.
Østerlie, T. "The User-Developer Convergence: Innovation and Software Systems Development in the Apache Project," in: *Department of Computer and Information Science*, Norwegian University of Science and Technology, Trondheim, 2003, p. 123.
Østerlie, T. "In the Network: Distributed Control in Gentoo/Linux", in *Proceedings of the 4th Workshop on Open Source Software Engineering, co-located with the 26th International Conference on Software Engineering (ICSE'04)*, Edinburgh, Scotland, May 25, 2004, pp. 76-81.

Østerlie, T. "Producing and Interpreting Debug Texts", in *Proceedings of the Second International Conference on Open Source Systems (OSS'06)*, Como, Italy, June 8-10, 2006, pp. 335-336.

Østerlie, T., and Jaccheri, L. "A Critical Review of Software Engineering Research on Open Source Software Development", in *Proceedings of the The Second AIS SIGSAND European Symposium on Systems Analysis and Design*, Gdansk, Poland, June 5, 2007a, pp. 12-20.

Østerlie, T., and Jaccheri, L. "Balancing Technological and Community Interest: The Case of Changing a Large Open Source Software System", in *Proceedings of the 30th Information Systems Research Seminar in Scandinavia (IRIS 30)*, Tampere, Finland, August 11-14, 2007b, pp. 66-80.

Østerlie, T., and Munkvold, G. "Ordering Actors, Organizing Work", in *Proceedings of the 28th Information Systems Research Seminar in Scandinavia (IRIS)*, Kristiansand, Norway, August 6-9, 2005.

Østerlie, T., and Rolland, K.H.R. "Unveiling Distributed Organizing in Open Source Software Development: The Practices of Using, Aligning, and Wedging", in *Proceedings of the Workshop on Open Source Software Movement and Communities, co-located with the First International Conference on Communities and Technologies (C&T'03)*, Amsterdam, Netherlands, September 18, 2003, pp. 1-7.

Østerlie, T., and Wang, A.I. "Establishing Maintainability in Systems Integration: Ambiguity, Negotiation, and Infrastructure", in *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, Philladelphia, PA, September 24-27, 2006, pp. 186-196.

Østerlie, T., and Wang, A.I. "Debugging Integrated Systems: An Ethnographic Study of Debugging Practice", in *Proceedings of the 23rd International Conference on Software Maintenance (ICSM'07)*, Paris, France, October 2-5, 2007, pp. 305-315.

Osterweil, L.J. "A Future for Software Engineering?", in *Proceedings of the 2007 Future of Software Engineering*, Mineapolis, Minesota, May 20-26, 2007, pp. 1-11.

Osterweil, L.J., Ghezzi, C., Kramer, J., and Wolf, A.L. "Determining the Impact of Software Engineering Research on Practice," *IEEE Computer* (41:3), March 2008, pp. 39-49.

Parnas, D.L. "Software Aging", in *Proceedings of the IEEE International Conference on Software Engineering (ICSE'94)*, Sorento, Italy, May 15-21, 1994, pp. 279-287.

Patton, M.Q. *Qualitative Research & Evaluation Methods*, (3rd ed.), SAGE Publications, Thousand Oaks, CA, 2002, p. 598.

Paulson, J.W., Succi, G., and Eberlein, A. "An Empirical Study of Open-Source and Closed-Source Software Products," *IEEE Transactions on Software Engineering* (30:5), April 2004, pp. 246-256.

Pennington, N. "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs," *Cognitive Psychology* (19:3), July 1987, pp. 295-341.

Pentland, B.T. "Organizing Moves in Software Suport Hot Lines," *Administrative Science Quarterly* (37:4), December 1992, pp. 527-248.

Perens, B. "The Open Source Definition," in: *Open Sources: Voices from the Open Source Revolution,* C. DiBona, S. Ockman and M. Stone (Eds.), O'Reilly & Associates, Sebastapol, CA, 1999, pp. 171-180.
Perkins, G. "Culture Clash and the Road to World Domination," *IEEE Software* (16:1), January/February 1999, pp. 80-84.
Pigoski, T.M. *Practical Software Maintenance: Best Practices for Managing Your Software Investments*, Wiley Computer Publishing, 1997.
Potts, C. "Software-Engineering Research Revisited," *IEEE Software* (10:5), January 1993, pp. 19-28.
Pozzebon, M. "Conducting and Evaluating Critical Interpretive Research: Examining Criteria as a Key Component in Building a Research Tradition," in: *Information Systems Research: Relevant Theory & Informed Practice,* B. Kaplan, D.P. Truex III, D. Wastell, A.T. Wood-Harper and J.I. DeGross (Eds.), Kluwer Academic Publishers, Amsterdam, 2004, pp. 275-292.
Raymond, E.S. "The Cathedral and the Bazaar," *First Monday* (3:3), 1998.
Ritchie, D. "The Evolution of the Unix Time-Sharing System," *AT&Bell Laboratories Technical Journal* (63:6), 1984, pp. 1577-1593.
Robinson, H., Segal, J., and Sharp, H. "Ethnographically-Informed Empirical Studies of Software Practice," *Information and Software Technology* (49:6), 2007, pp. 540-551.
Rosen, M. "Coming to Terms with the Field: Understanding and Doing Organizational Ethnography," *Journal of Management Studies* (28:1), January 1991, pp. 1-24.
Ruffin, M., and Ebert, C. "Using Open Source Software in Product Development: A Primer," *IEEE Software* (21:1), January/February 2004, pp. 82-86.
Ruhe, G., and Saliu, M.O. "The Art and Science of Software Release Planning," *IEEE Software* (22:6), November-December 2005, pp. 47-53.
Samoladas, I., Stamelos, I., Angelis, L., and Oikonomou, A. "Open Source Software Development Should Strive for Even Greater Code Maintainability," *Communications of the ACM* (47:10), Oct. 2004, pp. 83-87.
Scacchi, W. "Understanding the Requirements for Developing Open Source Software Systems," *Software, IEE Proceedings* - (149:1), February 2002, pp. 24-39.
Scacchi, W. "Free and Open Source Software Practices in the Gaming Industry," *IEEE Software* (21:1), 2004, pp. 68-72.
Scacchi, W. "Free/Open Source Software Development: Recent Research Results and Emerging Opportunities", in *Proceedings of the The 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE '07)*, Dubrovnik, Croatia, 2007.
Scach, S.R., Jin, B., Yu, L., Heller, G.Z., and Offut, J. "Determining the Distribution of Maintenance Categories: Survey Versus Measurement," *Empirical Software Engineering* (8:4), December 2003, pp. 351-365.
Schein, E. *Professional Education, Some New Directions*, McGraw-Hill, New York, NY, 1972.
Schön, D.A. *The Reflexive Practitioner: How Professionals Think in Action*, Ashgate Publishing Limited, Aldershot, UK, 1991, p. 374.
Schütz, A. *The Phenomenology of the Social World*, Northwestern University Press, New York, 1967, p. 255.

Seaman, C.B. "Qualitative Methods in Empirical Studies of Software Engineering," *IEEE Transactions on Software Engineering* (25:4), July/August 1999, pp. 557-572.
Segal, J., Grinyer, A., and Sharp, H. "The Type of Evidence Produced by Empirical Software Engineers", in *Proceedings of the Proceedings of the 2005 workshop on Realising evidence-based software engineering*, St. Louis, Missouri, 2005.
Serrano, N., and Ciordia, I. "Bugzilla, Itrackers, and Other Bug Trackers," *IEEE Software* (22:2), March-April 2005, pp. 11-13.
Shaft, T.M., and Vessey, I. "The Role of Cognitive Fit in the Relationship between Software Comprehension and Modification," *MIS Quarterly* (30:1), March 2006, pp. 29-55.
Sharp, H., and Robinson, H. "An Ethnographic Study of Xp Practice," *Empirical Software Engineering* (9:4), 2004, pp. 353-375.
Shaw, M. "The Coming-of-Age of Software Architecture Research", in *Proceedings of the The 23rd International Conference on Software Engineering*, Toronto, Ontario, Canada, May 12-19, 2001, pp. 656-664.
Shneiderman, B., and Mayer, R. "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results," *International Journal of Computer & Information Sciences* (8:3), June 1979, pp. 219-238.
Shull, F., Singer, J., and Sjøberg, D.I.K. (Eds.) *Guide to Advanced Empirical Software Engineering*. Springer Verlag, London, UK, 2007, p. 388.
Sjøberg, D.I.K., Dybå, T., Anda, B.C.D., and Hannay, J.E. "Building Theories in Software Engineering," in: *Guide to Advanced Empirical Software Engineering,* F. Shull, J. Singer and D.I.K. Sjøberg (Eds.), Springer London, London, UK, 2008, pp. 312-336.
Sneed, H.M. "Planning the Reeingeering of Legacy Systems," *IEEE Software* (12:1), January 1995, pp. 24-34.
Sneed, H.M. "An Incremental Approach to Systems Replacement and Integration", in *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, Manchester, UK, March 21-23, 2005, pp. 196-206.
Sommerville, I. *Software Engineering*, (6th ed.), Pearson Education Limited, Harlow, UK, 2001, p. 693.
Sullivan, M., and Chillarege, R. "Software Defects and Their Impact on System Availability: A Study of Field Failures in Operating Systems", in *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, Montreal, Canada, June 25-27, 1991, pp. 2-9.
Swanson, E.B. "The Dimensions of Maintenance", in *Proceedings of the 2nd International Conference on Software Engineering*, San Francisco, CA, October 13-15, 1976, pp. 492-497.
Swanson, E.B., and Beath, C.M. *Maintaining Information Systems in Organizations*, John Wiley & Sons, Inc., New York, NY, 1989, p. 255.
Swanson, E.B., and Beath, C.M. "Departmentalization in Software Development and Maintenance," *Communications of the ACM* (33:6), June 1990, pp. 658-667.
Swanson, E.B., and Dans, E. "System Life Expectancy and the Maintenance Effort: Exploring Their Equilibrium," *MIS Quarterly* (24:2), June 2000, pp. 277-297.

Taizan, C., Siu Leung, C., and Teck Hua, H. "An Economic Model to Estimate Software Rewriting and Replacement Times," *IEEE Transactions on Software Engineering* (22:8), August 1996, pp. 580-598.
Tanenbaum, A.S. *Modern Operating Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992, p. 728.
Tichy, W.F. "Should Computer Scientists Experiment More?," *IEEE Computer* (31:5), May 1998, pp. 32-40.
Tichy, W.F., Habermann, N., and Prechelt, L. "Summary of the Dagstuhl Workshop on Future Directions in Software Engineering: February 17–21, 1992, Schloß Dagstuhl," *ACM SIGSOFT Software Engineering Notes* (18:1), January 1993, pp. 35-48.
Tichy, W.F., Lukowicz, P., Prechelt, L., and Heinz, E.A. "Experimental Evaluation of Computer Science: A Quantitative Study," *Journal of Systems and Software* (28:1), 1995, pp. 9-18.
van Vliet, H. *Software Engineering: Principles and Practice*, John Wiley & Sons, Inc., New York, NY, 2000, p. 726.
Vans, A.M., Mayrhauser, A., and Somlo, G. "Program Understanding Behavior During Corrective Maintenance of Large-Scale Software," *International Journal of Human-Computer Studies* (51:1), 1999, pp. 31-70.
Voas, J.M. "Disposable Information Systems: The Future of Software Maintenance," *Journal of Software Maintenance: Research and Practice* (11:2), March/April 1999, pp. 143-150.
Vogels, W. "Web Services Are Not Distributed Objects," *IEEE Internet Computing* (7:6), December 2003, pp. 59-66.
Voká, M., Tichy, W.F., Sjøberg, D.I.K., Arisholm, E., and Aldrin, M. "A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns: A Replication in a Real Programming Environment," *Empirical Software Engineering* (9:3), September 2004, pp. 149-195.
von Mayrhauser, A., and Vans, A.M. "Program Comprehension During Software Maintenance and Evolution," *IEEE Computer* (28:8), August 1995, pp. 44-55.
Walsham, G. *Interpreting Information Systems in Organizations*, John Wiley and Sons, Chichester, 1993, p. 286.
Walsham, G. "Interpretive Case Studies in Is Research: Nature and Method," *European Journal of Information Systems* (4:2), June 1995, pp. 74-81.
Walsham, G. "Doing Interpretive Research," *European Journal of Information Systems* (15:3), June 2006, pp. 320-330.
Walsham, G., and Waema, T.M. "Information Systems Strategy and Implementation: A Case Study of a Building Society," *ACM Transactions on Information Systems* (12:2), April 1994, pp. 150-173.
Wang, H., and Wang, C. "Open Source Software Adoption: A Status Report," *IEEE Software* (18:2), March/April 2001, pp. 90-95.
Weber, S. *The Success of Open Source*, Harvard University Press, Cambridge, MA, 2004, p. 312.
Weick, K.E. "Enacted Sensemaking in Crisis Situations," *Journal of Management Studies* (25:4), July 1988, pp. 305-317.
Weick, K.E. *Sensemaking in Organizations*, SAGE Publications, Thousand Oaks, CA, 1995, p. 231.

Weick, K.E., Sutcliffe, K.M., and Obstfeld, D. "Organizing and the Process of Sensemaking," *Organization Science* (16:4), July-August 2005, pp. 409-421.
Williams, S. *Free as in Freedom: Richard Stallman's Crusade for Free Software* O'Reilly, Sebastapol, CA, 2002, p. 225.
Wilson, G. "Is the Open-Source Community Setting a Bad Example," *IEEE Software* (16:1), January/February 1999, pp. 23-25.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., and Wesslén, A. *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, Boston, Mass., 2000, p. 204.
Woods, D., and Guliani, G. *Open Source for the Enterprise: Managing Risks, Reaping Rewards*, O'Reilly, Sebastapol, CA, 2005, p. 217.
Xu, B., Qian, J., Zhang, X., Wu, Z., and Chen, L. "A Brief Survey of Program Slicing," *ACM SIGSOFT Software Engineering Notes* (30:2), March 2005, pp. 1-36.
Ye, Y. "Supporting Software Development as Knowledge-Intensive and Collaborative Activity", in *Proceedings of the 2006 International Workshop on Interdisciplinary Software Engineering Resarch*, Shanghai, China, 2006, pp. 15-22.
Yu, L., Schach, S.R., Chen, K., Heller, G.Z., and Offut, J. "Maintainability of the Kernels of Open-Source Operating Systems: A Comparison of Linux with Freebsd, Netbsd, and Openbsd," *Journal of Systems and Software* (79:6), June 2006, pp. 807-815.
Yu, L., Schach, S.R., Chen, K., and Offut, J. "Categorization of Common Coupling and Its Application to the Maintainability of the Linux Kernel," *IEEE Transactions on Software Engineering* (30:10), October 2004, p. 694.
Zelkowitz, M.V., and Wallace, D.R. "Experimental Models for Validating Technology," *IEEE Computer* (31:5), May 1998, pp. 23-31.
Zeller, A. *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufman Publishers, San Francisco, CA, 2006, p. 448.

Appendix: Papers P1-P8

Paper 1

Østerlie, T. "In the network: Distributed control in Gentoo/Linux", *in Proceedings of the 4th Workshop on Open Source Software Engineering, co-located with the 26th International Conference on Software Engineering (ICSE'04)*, Edinburgh, Scotland, May 25, 2004, pp. 76-81.

In the network: Distributed control in Gentoo Linux

Thomas Østerlie
Department of computer and information science
Norwegian University of Science and Technology
thomas.osterlie@idi.ntnu.no

Abstract

This position paper reports on the findings of an empirical pilot study of Gentoo Linux. Gentoo Linux is an open source Linux distribution developed by a geographically distributed community of volunteers. The reported findings are based on the analysis of a specific episode using actor network theory. With basis in the analysis, it is argued that control in this specific episode can be interpreted as both distributed and local at the same time. Control here being the power to define a problem and make the decision about the appropriate solution to the problem defined. Control, it is argued, is distributed in that it is the function of reciprocal influence among several human and non-human actors. Furthermore, it is argued that control can be interpreted as not inherent in organizational structures or hierarchies, but locally embedded among actors in the decision making process.

1. Introduction

Geographical distribution is one of the distinct characteristics of open source software development. Open source software development has been connected with teams of geographically distributed developers ever since Raymond's first description of the bazaar [1]. Despite the geographical distribution of developers, Raymond describes control in the bazaar as centralized, headed by the 'benign dictator'. Using open source software development as an example of computer-supported distributed work, Moon and Sproull [2] argue that an enabling condition for the success of the Linux kernel are the "capabilities a single leader brings to a project". They argue that the "clear locus of decision-making, singular vision, and consistent voice" are important in controlling this kind of collaborative effort. This supports Raymond's notion of the 'benign dictator'. Control in these two works is therefore understood as centralized.

Mockus and Herbsleb [3] describe the Apache open source web server community in two contradictory ways. On the one hand there is a formal organizational structure for making decisions about code integration. On the other hand, they report that work is not assigned but that individual developers choose what to do themselves. "The choices are constrained, however, by various motivations that are not fully understood." Understanding control as the power to define problems and their appropriate solutions, and thereby making decisions about what tasks to prioritize, Mockus and Herbsleb's description points to a tension between centralized and distributed control.

Picking up on Mockus and Herbsleb's observation, this paper raises the question whether control always is centralized in open source software development? How can we understand the tension between distributed and centralized control?

The paper is organized as follows. Section 2 presents the empirical findings. The section contains a short presentation of the Gentoo Linux case, details of the method employed, and a detailed presentation of the reported episode itself. Section 3 discuses how control can be interpreted in the reported episode. The conclusion draws implications of the discussion, and formulates directions for future work.

2. The case

This section presents the empirical findings. For context, an overview Gentoo Linux is presented first. Then the methods of data collection and analysis that form the basis for this position paper are described. The reported episode is described afterwards, after which the episode is analysed in terms of the mechanics of framing the problem to be solved and what actors take active part in framing the problem.

Gentoo Linux is an open source Linux distribution developed by a geographically distributed community of volunteers. Aiming for advanced users, the distribution is a mix between Linux from scratch and a regular Linux distribution. Gentoo Linux provides the minimum of support for installing a bare bones Linux system. In this way the user can build an installation from the bottom up, tuning it to his exact needs; be it a workstation installation, a secure server, or a gaming system. That is why Gentoo Linux is also called a meta distribution.

Portage, Gentoo Linux' software distribution system, is the technology that makes this possible. Portage keeps

track of the third party software, also called packages, available for Gentoo Linux at any one time. At the time of writing there are over 6000 packages available. Portage also keeps track of which packages have been installed on the local system. Information about installed packages is stored in a database. For each installed package this database contains information such as the absolute path for every files installed by the package, the compiler flags the package was built with, and the package's license.

When installing new packages, Portage compiles the software on the local system. The user can therefore fine-tune such things as compilation flags and additional software support. This information is stored in a set of configuration files.

2.1. Method

The episode reported in this position paper is part of the empirical evidence collected during a pilot study of Gentoo Linux. Data for this pilot study was collected with a number of methods. Archival data was collected from the Gentoo web site at `http://www.gentoo.org`, and from the Gentoo mailing list archives accessed through the `news.gmane.org` service. The IRC logs that form the basis of the analysis which this position paper is based on, were downloaded from Gentoo's home pages. In addition, the pilot investigation involved participatory observations with a software consultant using Gentoo Linux as development platform, and a semi-structured interview with one of the Gentoo Linux developers. The interview was performed according to the guidelines laid down in [4]. Ethnographic field notes [5] were taken in connection during the participatory observation and later written out as a full field report

The episode reported in this position paper is primarily based on the IRC log of the Gentoo managers' meeting from December 15 2003. Using actor network theory, an analysis was performed on basis of the log supplemented by the interview. Actor network theory is a method borrowed from the field of science and technology studies. It is a method for analysing the relationship between the technological and the social [6,7]. Unlike traditional software engineering methods that teaches us to categorizes entities into classes such as roles, instances, technical artefacts, organizational artefacts, just to mention a few, actor network theory attributes symmetry to all entities in the network by promoting them to actors. This reflects the basic assumption that all entities in the network are capable of acting upon each other.

Central to actor network analysis is identifying the actors and associations between them. Thinking of actors as nodes and associations as connections between the nodes, the network appears. The network is composed of heterogeneous nodes—technical and non-technical, human and non-human, etc.—that are associated for a period of time. However, the actor network is reducible neither to an actor alone, nor to a network. In addition, the network is seen as constantly shifting, and not as a representation of the original or final state.

In actor network theory the network is an analytical structure constructed by the analyst. Instead of thinking of the actor network as a representation of things out there, it is a conceptual frame, a perspective to interpret social and technological processes. The episode reported in sections 2.2, 2.3, and 2.4 is related as interpreted through the perspective of actor network theory.

2.2. The episode

The Gentoo managers' meeting is a biweekly meeting for Gentoo developers to coordinate activities. The managers' meeting is arranged over the Internet, using IRC. During the Gentoo manager's meeting December 15 2003 [8], the issue of third party utilities operating on Portage's database and configuration files is discussed. Some of these utilities mangle the configuration files, while other utilities no longer work because the Portage database format has changed. One of these utilities, `qpkg`, a utility for querying Portage's database, has accumulated over 20 unresolved bug reports in Gentoo Linux' bug tracking system. The source of all these problems is identified to be code that is out of synchronization with the rest of the system. This kind of problem has been resolved before by introducing the maintainer role. The maintainer is responsible for keeping specific parts of code in synchronization with the rest of the system. The conclusion is that the code in question is outdated because it has not been assigned a maintainer.

An additional response to the problem is to introduce an abstraction layer, an API, on top of Portage's database and configuration files. All utilities accessing the configuration files and database must do so through this API. Two Gentoo developers are assigned to develop and maintain this API.

There is dissent among the participants at the meeting about priorities. Gentoo Linux' chief architect proposes to base the API on Portage's own code. The two developers in charge of the API, while agreeing that this would be a good idea, argue that there are other factors that are more important to take into account when resolving the problem. Especially the issue of missing maintainers for utilities accessing the Portage database and configuration files. The `qpkg` utility is used as an example of these difficulties. The utility was included in the distribution by a developer who later left the project. `qpkg` implements its own code for accessing Portage's database. Responsibility for the utility was handed over to someone else when the original developer left Gentoo Linux. This second developer went on leave, and `qpkg` was left un-maintained. The problem, while technical in symptoms, is something more and something else. It is also symptomatic for the problems to be addressed by the API developers, in that `qpkg`, like the other utilities, implement its own code for accessing Portage's database and configuration files directly. Without any guarantee for

how long the developers for these utilities will stick around Gentoo Linux, the situation that the API is to address is to keep the way utilities access Portage's database and configuration files synchronized even after the original developers leave.

2.3. Framing the problem

The decision to introduce an API on top of Portage's database and configuration files is an answer to a problem the Gentoo developers want to solve. Thinking in terms of actor networks, the problem can in fact be conceptualized as an actor. However, it is not an actor that exists before the meeting starts. It is actually a constructed actor. The problem is "a list of ... trials … hooked to a name of a thing and to a substance" [7, p.122]. The way the problem is given substance, its framing, is the topic of this section.

In the transcript from the Gentoo manager meeting December 15 2003, one of the developers participating in the meeting states that there are a "slew of util[itie]s lying about". He associates these with mangled Portage configuration files, in that the utilities "hack, slash and mutilate the ... config[uration] files". Then he associates the Portage database with the "util[itie]s lying about", as "these util[itie]s misreads /var/db [the Portage database, author's comment], so as not to be consistent with [P]ortage". Another problem with the "util[itie]s lying about" is that they have overlapping functionality, and none do their tasks particularly well:

"*we don't need five half-working use flag editors. we need one really good one*"

The problem is framed by the developer associating different actors, framing a problem in such a way that the other developers understand it as their problem, too. Figure 1 illustrates how the different actors are associated in framing the problem.

Having framed the problem as a shared problem, its cause is established. The cause of the problem is that the utilities lying about have not been properly updated, as "a few of the existing tools [the same as the utilities lying about, author's comment] don't work with portage 2-0.50 due to API changes [in Portage, author's comment]". That is also why the `qpkg` utility does not work any longer, since there are "20+ bugs [reports] about `qpkg`" that remain unresolved in the bug tracking system. The technical cause of the problem is outdated code, but this is more a representation of the larger problem:

"*now I have 20+ bugs about qpkg assigned to me, it's a mess, and nobody wants to touch it. Who is responsible to maintain it now?*"

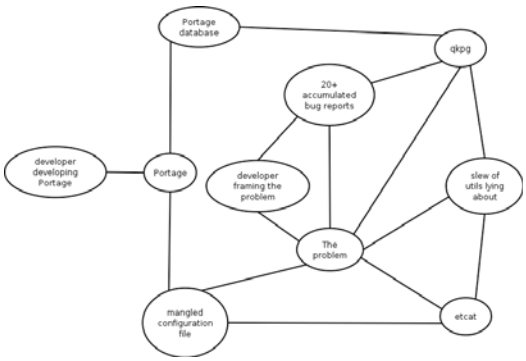


Figure 1 The problem framed

The symptom is that the utilities lying about have not been updated, but this is caused by the fact that there are no one maintaining the "slew of util[itie]s lying about". In this way, the maintainer replaces the problem in the actor network, providing a solution to the situation.

Control is exercised in deciding what activities are to be undertaken, how and when. There are hundreds of unresolved bug reports in Gentoo Linux' bug tracking system. In making the decision about which of these bug reports are to be resolved, decisions about what activities to prioritize are made. Framing the problem can therefore be understood as the power to determine the activities to be undertaken. From this follows that the task of identifying who is in power in the episode above, is the task of identifying who has the power to frame problems.

2.4. Who frames the problem?

At first glance, the problem facing the developers seems to be framed by one of the developers participating in Gentoo manager meeting. As a response to the problem the maintainer role is introduced. The maintainer role, as an actor decoupled from a person, was once constructed to resolve similar situations. In framing the problem at hand in this particular way, the answer to introduce a maintainer becomes a given. Following this line of thinking, one can go as far as saying that the maintainer role participates in shaping the problem. If you have a hammer, all you see are nails. The knowledge among discussion participants that this role exists can be considered constitutive to the problem framing. Looking at the episode this way, the maintainer role is turned from passive to active in framing the problem.

It is highly unlikely that every bug experienced by Gentoo Linux users is reported in the bug tracking system. However, the bugs that are used to frame the problem are those reported in the bug tracking system. Bugs are given priority, severity, status, and assigned to a given person or group of persons for resolution. A bug is resolved when it is fixed or labelled invalid. As long as a bug remains unresolved but assigned to a developer, the

bug is a reminder to the assignee. In this sense, bug reports are also active in framing the problem.

Framing the problem is not a function of a single developer or a closed group of developers. Instead, it can be interpreted as the function of a number of actors, both human and non-human. Neither is the power to frame the problem one-sided in that one actor forces other actors to do something they do not want to. Instead, framing is a reciprocal relationship between the Gentoo developers, the maintainer role, and the bug reports.

3. Discussion

This discusses how control can be interpreted in the above episode above. Three aspects of control are discussed. First the implication of the episode in terms of control and organizational hierarchies is discussed. Then we discuss how control can be interpreted as distributed among human and non-human actors. Finally, it is argued that actor network theory makes the interpretation of control as reciprocal among actors likely.

3.1. Relation of control and organizational hierarchy

Gentoo Linux is split into projects and sub-projects. Herds consisting of maintainers are responsible for keeping a set of packages up to date. This is how the Gentoo developers describe their organization in terms of hierarchies and distribution of roles. However, by conceptualizing the way the Gentoo developers talk about the organization during the Gentoo Managers' meeting as an actor network, another view appears. In framing the problem that the API resolves, the maintainer is introduced as an actor in the network. In contrast Gentoo Linux' chief architect does not get through his idea to base the API directly off Portage.

Looking at the organizational hierarchy, the architect is placed farther up than the developer. If control and organizational hierarchies were related, the chief architect would have the power to make his view the prevailing. In the episode above, this does not happen, though. Why not?

Control can be understood as local in the way actors enrol other actors and are enrolled themselves in the immediate actor network. If control was inherent in the hierarchy, the chief-architect should have gotten his view through. That he does not get his view through can be explained by him never enrolling the chief architect role, considered an actor in an actor network analysis, in the immediate actor network.

The implication of the above interpretation is that there need not be an inseparable relation between organizational hierarchy and control. Control can be locally embedded among actors in the immediate network. The actors brought together by the hierarchy have no essential relation to each other, but can instead be understood as dispersed actors temporarily brought together through the hierarchical ordering. By viewing of actors as inherently dispersed, thinking of the organization as an actor network shows that the hierarchical description of organization is just that: a hierarchical description of organization, an abstraction. As such organizational hierarchy need not be inherently connected with control.

3.2. Control is distributed and heterogeneous

In saying that a corrupted configuration file is the same as a missing maintainer, technical (the corrupted configuration file) and organizational (the maintainer) actors are treated as equals. By treating all actors symmetrically this way at the same level of analysis, control can be interpreted as the mutual relationship between heterogeneous actors. Control is not the relationship between action and structures of signification, legitimization and domination [9], but in the direct relationship between actors in the network. A possible implication of this interpretation is that control is no longer purely social, but a function of human and non-human actors, of technological and non-technological actors, of organizational and non-organizational actors. Control becomes orthogonal. It is a function between all actors in the network, regardless of classification schemes. Actors are no longer higher or lower in the organizational hierarchy, technical or non-technical, human or non-human; they are all and the same: actors in the network.

3.3. Control as reciprocal

In saying that control can be understood as local to the immediate network of actors, control becomes both the actors' ability to frame problems, and the ability to limit other actors' framing activities. Control can therefore be understood as more than the traditional control relation within a set of actors

A_B
C_D
D_B
A_E

but as a relationship where actors reciprocally control each other, understood as the relation of

(A, B, C, D, E)

In the latter relationship lies the argument that control is distributed. Control can't be reduced to an actor A's ability to overcome actor B's and thereby exert control over B, as implied in the relationship A_B. It is not one-sided, but distributed. A must not only overcome B's resistance, but the resistance of the other actors in the immediate network. In this sense, in exerting control over B, A exposes itself to the controlling power from the other actors.

4. Conclusion

This paper has argued that traditional notions of control may be inadequate in describing distributed

control in Gentoo Linux. Control, it is claimed, need not be limited to the people who seem to be making decisions. Rather, control can be interpreted as distributed among both human and non-human actors. In reported episode, control is distributed among a number of Gentoo developers, the maintainer role, and bug reports. In this sense, control is not distributed in terms of geographical distribution, but distributed as in shared among a handful of human and non-human actors.

While Gentoo Linux is geographically distributed, the interpretation of distributed control is not connected with the geographical distribution. It is, rather, connected with the distribution of elements within an actor network. The key points of distributed control are:

a) that control need not be inherent in the organizational hierarchy, but can be interpreted as embedded in the immediate actor network
b) that control need not be inherent in structures, but can be distributed among actors,
c) that control can't always be reduced to a function of human agency, but may at times be understood as the function of all actors in the network such as tools and organizational roles
d) that control can be a reciprocal relationship between a set of actors

Thinking of distribution this way, similar analysis of distributed control could therefore be equally applicable in geographically co-located software development efforts, too. Distribution is not geographically, but instead understood as distributed among actors.

In arguing that control is distributed in Gentoo Linux, this position paper addresses only the mechanics of control through following the construction of networks through enrolling. The rules of this construction are left untouched. How is it that some actors in the network inscribe stronger behaviour than others? What are the rules for enrolling actors, and what are the rules for excluding actors as valid to be enrolled? These issues need to be addressed in future studies.

The decision to do an API on top of the Portage database and configuration files were only a month and a half old when this pre-study was done. At the time of writing, the API has still to be integrated in a large scale. It is available in Gentoo Linux, but very few utilities actually use the API. A point of future study is to follow up how the implementation of the API and its integration with utilities goes. How is access through the API enforced? How are bugs connected with not using the API handled? What are the effects of introducing the API? Does it lead to lesser problems for utilities integrating with Portage's database and configuration files?

7. References

[1] E.S Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly, Sebastapol, 1999.

[2] J.Y. Moon, and L. Sproull, "Essence of Distributed Work: The Case of the Linux Kernel", First Monday, 5:11, 2000.

[3]A. Mockus, and J.D. Herbsleb, "Why Not Improve Coordination in Distributed Development by Stealing Good Ideas from Open Source?", Proceedings of the 2nd Workshop on Open Source Software Engineering, IEEE, 2002.

[4] S. Kvale, *InterViews: An Introduction to Qualitative Research Interviewing*, SAGE Publications, New York, 1996.

[5] R.M. Emerson, R.I. Fretz, L.L. Shaw, *Writing Ethnographic Fieldnotes*, University of Chicago Press, Chicago, 1995.

[6] M. Callon, "Some elements in a sociology of translation: domestication of the scallops and fishermen of St. Brieuc Bay". *Power, Action and Belief*, Routledge, London, 1986.

[7] B. Latour, "Technology is society made durable", *A Sociology of Monsters. Essays on Power, Technology and Domination*, Routledge, London, 1991.

[8] Gentoo Managers' Meeting Log, `http://www.gentoo.org/proj/en/devrel/manager-meetings/logs/2003/20031215.txt`, last accessed March 1 2004.

[9] A. Giddens, *The Constitution of Society: Outline of the Theory of Structuration*, Polity Press, Cambridge, 1984.

Paper 2

Berntsen, K., Munkvold, G., and Østerlie, T. "Community of practice versus practice of the community: Knowing in collaborative work," *The ICFAI Journal of Knowledge Management* (II:4), December 2004, pp 7-20.

Community of Practice versus Practice of the Community: Knowing in collaborative work

Kirsti E. Berntsen *Glenn Munkvold* *Thomas Østerlie*

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

NTNU-IDI, Sem Sælands vei 7-9, NO-7491 Trondheim, NORWAY

keb@idi.ntnu.no *glm@idi.ntnu.no* *thomasos@idi.ntnu.no*

***Abstract**: How do software developers, field service technicians, and medieval cathedral builders accomplish collaborative work? This paper looks at how they learn from each other by building and sharing knowledge across time and space.*

To illustrate this, we first present Community of Practice (CoP) as a way of understanding collaborative work which puts focus on the community and its social interaction. CoP, introduced by Lave and Wenger (1991), is based on the fundamental belief that dividing theory from practice is unsound. Hence CoP contradicted traditional theories of learning, where learning and working often are conceived as separate processes. Using Orr's (1996) rendition of service technician's work, it is shown that stories act as repositories of accumulated wisdom in keeping track of facts, sequences and their context. Representations made by a CoP to aid their work, are termed Reifications which can be stories, tools, artefacts etc. Practice is seen as a duality of Participation and Reification which both require and enable each other. We find however, that CoP based analyses tend to focus on the human actors in that you start out by looking for the communities and what defines them. We also present examples of alternative approaches that illuminate the technology and artefacts that are present in collaboration. Berg(1997) uses Actor-Network Theory (ANT) to illustrate the responsibility awarded to artefacts in the process of documenting a hospital-patient's fluid balance. Hutchins(1995) describes navigation as a joint accomplishment of artefacts and people. Turnbull(1993) sees a wooden template as a chief enabler of building gothic cathedrals without use of structural mathematics. Facets of knowledge/knowing is discussed, their accumulation and transfer by stressing the value of both the social and the technical approach.

***Keywords**: collaborative work, communities of practice, actor-network theory, role of technology, knowledge sharing*

1. Introduction

What is it that software developers do when building software systems? And what is it that field service technicians do when fixing broken copying machines? For that matter, what did medieval cathedral builders do when raising tall stone cathedrals across Europe? What do software developers, field service technicians, and medieval cathedral builders have in common? In the context of this paper, the answer is they achieve their goals through *collaborative work*: they build and share knowledge and learn across time and space.

Researchers in different academic fields have made attempts to describe and explain collaborative work. The IS researcher wants to understand the collaborative efforts involved in developing software (Naur, 1992). The ethnographer (Orr, 1996) wants to describe and understand how field service technicians collaborate on fixing broken copying machines, and the historian (Turnbull, 1993) wants to know how cathedral builders managed to raise a multitude of tall stone cathedrals all across Europe in a relatively short period of time.

Let's turn the coin and rephrase the questions posed above. How are software systems built? How are broken copying machines fixed? How is the building of gothic cathedrals achievable? There is of course no single answer to these questions, but they raise the issue concerning the constituents of collaboration.

This paper discusses how different research traditions have opened the black box of collaborative work, trying to explain collaborative work with different approaches. This is not an exhaustive literature review on the topic, but rather the beginnings of one.

The paper is structured as follows. First, we present Community of Practice (CoP) as a way of describing and understanding collaborative work. After discussing the contribution to understanding collaborative work provided by the thinking around CoP, we discuss the approach's shortcomings in addressing the role of technology in collaborative work. We then present alternative approaches to describing and discussing collaborative work which are specific on the role of technology. After discussing these approaches' contribution to understanding collaborative work, we conclude by drawing the implications that such an approach has on the way we think about collaborative work and the sharing of knowledge and knowing.

2. Programming as theory building

Naur (1992) argues that software development is more than just production of a program and certain texts. Successful software development is a question of having the appropriate theory, as in a mental model, of the software system. With certain kinds of large programs, the continued adoption, modification, and

correction of errors depends on knowledge possessed by a group of developers who are closely and continuously in connection with the software system. The developers' knowledge transcends that which is recorded in the documentation: they possess a theory of the software. "[A] person who has or possesses a theory ... knows how to do certain things and in addition can support the actual doing with explanations, justifications, and answers to queries, about the activity of concern" (ibid., p. 229). The notion of theory was proposed by Ryle (1949) in an effort to describe the difference between intellectual and intelligent behaviour. Ryle claims that intelligent behaviour is the ability to do certain things without having any concrete knowledge to build this behaviour on.

Naur's perspective on software systems development is that of the individual developer. While his contribution is significant in that it provides argumentation for viewing software systems development as a knowledge intensive activity, it fails to address the dynamics of collaborative work. Even though he argues that the theory of the software system must be shared by a group of developers, the theory is still embedded in the individual. By not being specific on the description of *how the theory is shared*, Naur only manages to point out that software development is in fact collaborative work. The context surrounding the development of software is not included in Naur's discussion.

The question, then, becomes: how is knowledge shared, across time and space, and how does context play a role? The related topic of how knowledge is built or acquired across time and space will be touched upon in our discussion.

3. Communities of practice

The way people work differs from the abstract ways organizations describe that work in manuals, training programs, organizational charts, and job descriptions (Brown and Duguid, 1991). Communities of practice (Wenger, 1998) is a concept used to better understand the activities and processes going on in work, and what kinds of social engagements provide a better context for learning and innovation to take place.

CoP was first introduced by Lave and Wenger (1991). It is based on the fundamental belief that dividing theory from practice is unsound. Hence CoP contradicted traditional theories of learning, where learning and working often are conceived as separate processes. Instead, CoP argues that learning should be contextualized by acknowledging its presence and allowing it to continue to be an integrated part of work. Based on the fieldwork of Orr (later published as Orr, 1996) Brown and Duguid (1991) illustrate how formal descriptions of work and learning often are abstracted from actual practice, and how knowledge is socially constructed through informal interaction. Orr did his fieldwork by observing a group of Xerox repair technicians who met regularly in informal, common areas trading stories and insights around their work (repairing different types of copying machines). The workers actually made a point out of

spending more time in each other's company. This slack initially seemed like an excellent opportunity for productivity improvements. However, Orr's fieldwork shows that these activities were actually a very important part of becoming, being and remaining a good technician. It was central to how the technicians learned, how they improved their skills, how they formed bonds as a community of practice, and how they transferred and honed their knowledge and expertise amongst themselves.

The creation and transformation of knowledge in the Xerox case is related to social interaction among technicians. Taking form as storytelling, the knowledge transfer made the technicians capable of sharing not only the type of knowledge that could be read out of books, but also the type of knowledge not explicitly stated in the company's instruction manuals. The practice included sharing both the explicit and the tacit/implicit. What was said and left unsaid thus served as an intrinsic part of solving the problem. According to Brown and Duguid (1991) stories act as repositories of accumulated wisdom and it allows people to keep track of the sequence of behaviour and of their wisdom, in keeping track of the facts and their context. In a highly situated and improvisational approach, the technicians were able to construct a shared understanding out of bountiful conflicting and confusing data.

Communities of practice rely on the informal depiction that each member generates of it: who is part of the community, which are the different modes of participation that are accepted, who knows what, what cultural tools are used to mediate communication and interaction, and so forth. The depictions of the community are iterative and evolve continuously as community members share experiences, take action and interact with each other, as well as the outside world which is reasoned about. A shared understanding is negotiated and emerges from scattered pieces of knowledge and knowing. The differentiation between knowledge and knowing is described by Cook and Brown (1999, p.381) in that "[k]nowledge and knowing is seen as mutually enabling (not competing). We hold that knowledge is a tool of knowing, that knowing is an aspect of our interaction with the social and physical world."

In general, Wenger (1998) defines a CoP along three dimensions:

1. a joint enterprise that is continually renegotiated by the members of the community
2. mutual engagement, that bind the members together into a social entity
3. a shared repertoire of common resources that the members have developed over time (routines, vocabulary, artefacts, experiences, stories, etc.).

The resources developed by the community can somehow be considered the accumulated knowledge and knowing of the community.

This informal, narrative and contemplative nature or aspect of a CoP, does not preclude that a community may also make formal representations, checklists,

tools etc. as well as to define concepts and ideas, to aid them in their endeavours of work (ibid., pp. 62-71). These representations are termed *Reifications*. Practice can be seen as a duality of *Participation* and *Reification* in which both require and enable each other. "Participation is not merely that which is not reified (ibid., p.66). On the contrary, they take place together. … There is no reification without participation … [and vice versa]". The reifications/artefacts play a key role since they are often used as explicit representations of the informal model that is shared among the members. Reifications may also function as *boundary objects* through which different communities can relate to each other. A boundary object has a common denominator that each community can identify and relate to, but may play different roles and have extra meanings within the CoP, in line with the context and joint enterprise of that particular CoP.

Discussion of shortcomings

In CoPs the relation between the subject and the "world" assume that the subject adapts to the surroundings by means of participating in communities of practice. The artefacts and technology which aid their existence remain self-evident and in the background. Practice - implicitly understood as knowing, which means doing and learning how to do, is explained, understood and interpreted by means of the human subject.

In order to see the artefact in the theory of CoP, the artefact must either be the central joint enterprise, or a boundary object. Brown and Duguid's example of the Xerox technician's CoP has the artefact, its representations and interactions within the customers organizations as "The central joint enterprise" around which the CoP evolves. The machine/artefact is also a boundary object that connects their CoP to their customers' communities of practice.

CoPs allow the artefact a place on the agenda in a more or less informal fashion as reifications of human action. They play a critical role in cultivating and coordinating knowledge but are only considered to be frozen reifications that must be interpreted by the human actors. A similar point has been made by Prout (1996 in Timmermanns and Berg 2003, p.9) saying that "Work is constructed as done on and through machines, but not by them".

4. Illuminating the elusive technology

A relevant question is then: Does the theory of CoP adequately cover the relevant aspects of collaborative work? The poignant catch here is the word *relevant*. The relevance of various theories depend on the direction of interest in the application of theory. Wenger states in his introduction (1998) that his purpose is "… to propose … what I call a social theory of learning … which comes close to developing a learning-based theory of the social order. In other words, learning is so fundamental to the social order we live by that theorizing

about one is tantamount to theorizing about the other." No wonder, then, that CoP has become widely used, outside its original scope of learning.

CoP has been widely adopted within both communities studying organizational knowledge as well as within management theory. Contu and Willmott (2003) contend that many of these renditions have disregarded or failed to see, some aspects of Lave and Wenger's (1991) original work such as: "... embryonic appreciation of power relations as media of learning" (Contu & Willmott 2003, p. 283) in that the topic of power relations in a situated learning context often is not addressed by those who embrace the concept of CoP into their own discourses. There may be many reasons for this end result, Contu and Willmott (ibid.) reason about both the present oversight of power relations and for the subsequently necessary re-inclusion of power relations into the situated learning discourse.

We intend to show that in a similar fashion, other embryonic appreciations also tend to disappear when using CoP for theorizing on communities that include artefacts as reifications. Wenger's concept of the boundary object that mediates understanding between communities, albeit sometimes very selective understandings, is both illuminating and useful. Various artefacts and technologies may constitute such boundary objects, along with other reifications such as narratives, rules and norms, etc. The concept is a powerful one for grasping constituents of communication and collaboration between different communities in illustrating that it allows them to cooperate without a unilateral (universal) consensus on activities, purposes and priorities. However, the deeper aspects of the reifications as resources within the community and across communities is little expanded in CoP. CoP divulges some aspects of artefacts in communities, but remains ignorant or uninterested in others.

It is our observation that common concepts concerning the humane inhibit the inclusion of non-human aspects into our discourses of societies, organizations and activities. And so we mostly turn a blind eye to the technologies we interact with. When we do address technology, acknowledging its presence, it tends to be in an instrumental dichotomous fashion where the humans are either in total control or at its mercy. We wish to expand our concepts of both the artefacts and the humane, to stretch the dichotomy into a duality ascribing more than structure or mediation to the artefacts. Wenger does describe such a duality, but the focus of Cop is still mainly on the social aspects.

5. Making technologies explicit

Marc Berg uses actor-network theory (ANT) to take a closer look at artefacts within work practices, both the IT system and other artefacts. Berg's studies show that some qualities of technology as artefact may be seen as universal in holding both knowledge and a transformational power of informal practical world aspects into formal representations.

Marc Berg (1997) takes a detailed look at practice in a hospital intensive care unit. His case describes each minute part of a work process which aims at documenting a hospital-patient's fluid balance, which is a sum of what fluid goes in and what comes out. In observing and recording each minute detail of the particular process, separate elements are identified. This hybrid consisting of several people, various artefacts, routines and experiences comprises everything that is needed for the activity of measuring a patient's fluid balance to proceed. The formal tools, the artefacts, come to life only as part of real life activity.

The shape of the bag of diffusion liquid with its quantity scale gives input to the nurse for the number to be entered into the fluid balance spreadsheet. The granularity of the scale defines the level of accuracy. The size and shape of the drinking cup and the urine container also re-represents (as in representing again) the separate liquid in- and outputs of the patient's body into formal representations. These formal representations can again be entered into the spreadsheet. The person entering the number needs no knowledge of medical theory, diagnosis, treatment, or purpose for performing this specific task. The only interpretation necessary by the human is reading the quantity scale in order to enter it into the spreadsheet. "The task of producing formal representations is delegated to the mundane artefacts which perform, in Latour's terms, 'the practical task of abstraction'" (Berg 1997, p.144)

Berg focuses on the interrelationships between the artefacts and the human workers in saying that through these interlockings, new competencies can be achieved and higher levels of complexity in work tasks can be achieved. People can be seen as communicating/interlocking via the tools without intimate knowledge of the other parts of the process chain. The distributed nature of the activity, shared between the artefacts and human actors effect a distribution of control and responsibility across the heterogeneous ensemble of humans and artefacts. The individual actors have no overview of the complete process, and cannot affect global workarounds based on an overall picture. The humans are not in control of the overall task. On the other hand, neither are any of the artefacts. The human actors introduce workarounds in performing their own particular tasks pertaining to the unexpected contingencies of either their colleagues or the artefacts. Another shape or functionality, in effect a different inscription in the involved artefacts, would however shape the human actors tasks differently.

Another point of Berg is that the ensemble of humans and artefacts–the actor network–cannot bee seen as stable once the artefacts are in place. In line with the view of artefacts and humans as equal actors in producing the end result of an activity or process, then all actors within the network are affected when changes occur in the forces influencing the network. Most work processes have aspects of drift in which work is continually redesigned to adapt to the particular circumstances. This drift also introduces the need to continually adapt the use and/or functionality of the artefacts. A quaint analogy of this need

for adapting artefacts can be related to perhaps our most archaic artefact of all– the hammer. A modern-day hammer comes in various shapes and sizes– adapted to each craft's particular need. The cleft in today's carpenter hammer arose from the need to pull out misplaced iron nails. This functionality was inconceivable in the times of wooden pegs.

While Berg places technology as embedded locally, Hutchins (1995) is concerned with the "circulation" of cognition in collaborative work. Traditionally human cognition has been placed within the mind of the individual, as previously exemplified by Naur's notion of programming as theory building. A basic idea in distributed cognition is that human activity does not take place solely in the heads of people, but that the environment–social, physical, and artefactual–provides a cognitive context from where cognition actually should be delineated. Looking at the practice of navigating ships, Hutchins (1995) develops a methodological and analytical framework for understanding how cognitive achievements can be conceptualised as a joint accomplishment of artefacts and people. According to Hollan et al. (2000) in distributed cognition, one expects to find a system that can dynamically configure itself to bring subsystems into coordination to accomplish various functions. At the core of Hutchins' argumentation lies an assumption of equality between people and artefacts in structuring practice. In this way the centre of attention in collaborative activities are the interdependencies between people, and between people and artefacts.

Similarly Turnbull's (1993) study of medieval cathedral building can be understood in terms of collaborative work. Medieval cathedrals were built in a discontinuous process by different groups of masons. Turnbull's challenge is to explain how masons could build these tall buildings without knowledge of structural mechanics. During the 13th century 50 cathedrals were raised throughout Europe. Turnbull envisions the cathedral building site as an "experimental laboratory" in which the key elements were the template, geometry, and skill" (p.322). The argument is that the collective work of cathedral builders was not one of human ingenuity alone, but also manifest in artefacts. Turnbull shows how wooden templates for building arches circulated between building sites, acting as accumulations of every design decision that had to be passed on. Because a template is easy to replicate, it could circulate among builders at a site, and among building sites across Europe. In this way, knowledge of gothic cathedral building, as manifested in the template, could circulate and spread. Also, argues Turnbull, the template has an organizing effect, having the power to organize large number of workers. Turnbull's approach is specific on the role technology plays in transferring knowledge and indirectly coordinating collective work.

6. Discussion

We have so far discussed different approaches to describing and understanding collaborative work. The approaches were presented in two parts. We first presented CoP as an approach to describe and understand collaborative work, arguing that this approach conceals or fails to address many of the inscribed qualities of the technology. We then presented different examples making technology more visible. We focused on describing these approaches as dissimilar in terms of the role technology play in their way of describing and understanding collaborative work. In this section, we attempt to extract similarities in the topics these approaches handle. We see two topics running through all the works presented above:

- knowledge accumulation and knowledge transfer
- different facets of knowledge

6.1 Knowledge accumulation and transfer

Knowledge accumulation is a question of where knowledge is stored. While stored gives mechanistic associations, it is not intended in this way. Rather, it is used to describe that different knowledge is embedded in different actors. It is a question of who/what has knowledge. The who/what dimension follows from the differences between the different approaches presented above. The communities of practice approach, exemplified by Julian Orr's (1996) ethnographic study of field service technicians and copying machines, views knowledge as embedded in the practices of human actors. It is the field service technicians and the human users of the copying machine that has knowledge of the machines. The user knows the specifics of a given machine, while the field service technicians know the general problems associated with series and models of machines as well as possibly having knowledge of the history of the specific machine.

The distinction between knowing and doing is not made explicit. The epistemological assumption in CoP is that doing or knowing is socially situated. Knowledge is an intrinsic property of people's engagement in communities of practice. Accumulation of knowledge is attributed to the human actors in a "collective mind of the community". Application of the knowledge is solely explained by means of human agency.

Conversely, in Marc Berg's (1997) study of cooperative work in hospitals, knowledge is explicitly accumulated along a process chain. This process chain consists of humans as well as technology in a chain of distributed links. The separate artefact links in the process chain also have knowledge inscribed in them. The various liquid vessels have the appropriate size, shape and measurement scales appropriate for their appointed task of collecting liquids and turning them into a numeral representation. The vessels know, as Mol

(2003) would put it. This is similar to Turnbull (1993) who argues that knowledge of building cathedrals is based on the key elements of the template, geometry, and skill (p.322). The template, however, plays an important role in accumulating knowledge outside humans. It "encapsulated every design decision that had to be passed down to the man doing the carving in shop and quarry" (ibid.). The way the artefact accumulates knowledge, is a primary explanatory factor in Turnbull's work, as the building of gothic cathedrals was a discontinuous process. It is this discontinuity that is missed by solely looking towards humans as knowledge accumulators.

Narration is an important aspect in the communities of practice approach to collaborative work. The narrative is a way of transferring knowledge. Knowledge is transferred through social interaction, through narratives, through talking about machines. Turnbull, Hutchins, and Berg on the other hand, see knowledge transfer as the circulation of artefacts among people and among communities. In this line of thinking knowledge is shared through circulating artefacts among people. Which is it? Which of these approaches are correct? Is knowledge accumulated in people and shared through social processes, or is knowledge accumulated in artefacts are shared through the circulation of artefacts? Our argument is that both are valid, important and dependant of each other.

6.2 Facets of knowledge

In line with Nonaka and Takeuchi's (1995, p. 235-240) assault on what they term "false" dichotomies we argue that the dichotomy of human versus artefact is such a false dichotomy. "The dynamic and simultaneous interaction between two opposing ends of 'false' dichotomies creates a solution that is new and different. In other words, A and B create C, which synthesizes the best of A and B. C is separate and independent of A and B, not something 'in between' or in 'the middle' of A and B" (ibid., p. 236). Rather the concepts of knowledge accumulation and knowledge transfer must be seen in the light of the dynamic integration of three of the synthesized "false" dichotomies that Nonaka and Takeuchi put forward (p.237) namely explicit versus tacit knowledge, body versus mind, and individual versus organization. Nonaka and Takeuchi, however, do not include the artefacts in their theorizing. This is in line with Cook and Brown (2003, p.381) who state that: "Organizations are better understood if explicit, tacit, individual and group knowledge is treated as four distinct and coequal forms of knowledge (each doing the work the others cannot), and if knowledge and knowing are seen as mutually enabling (not competing).

In accepting Berg's argument that knowledge and knowing is distributed among actors, and that no single actor has the complete picture of the collaborative work process, we argue that knowledge can be accumulated in both humans and artefacts. In this way, knowledge and knowing can be shared through the

circulation of artefacts and accessed, interpreted and applied by people. CoP stresses that the interpretation and application is activated through social interaction. This, for us, is the consequence of applying Berg's argument to the topic of knowledge and knowing accumulation and sharing in collaborative work. What we are saying is that a medieval mason, although skilled at building brick walls and columns, is unable to raise a gothic cathedral without the template. Conversely, a person not skilled in masonry is unable to build a cathedral no matter how many templates he is in possession of. Using CoP alone to analyze this example fails to appreciate the qualities of the artefacts. Focusing on the technology renders the social barely visible.

Based on the above discussion, it may be argued that the CoP approach is mainly concerned with the social aspects regarding establishing and sharing of knowledge/knowing. As Wenger (1998, p.141) puts it "knowing is defined only in the context of specific practices, where it arises out of the combination of a regime of competence and an experience of meaning", while Turnbull and Berg are more concerned with how knowledge is made durable and transferable across social contexts.

The body versus mind dichotomy can be seen as an illustration of the skills that the human has acquired as opposed to the abstract depictions or representations we have of those skills. Knowledge/knowing as read from text books can be seen as knowledge transfer in an abstract manner. Know-how may be analyzed and put into words and numbers in order to externalize its content and make it explicit. In the process of abstraction and transfer, something is lost. Nonaka and Takeuchi give the name tacit knowledge to the part of know-how that cannot be externalized. Wenger (1998) states that "[c]lassifying knowledge as explicit or tacit runs into difficulties, however because both aspects are always present to some degree ... what counts as explicit depends on the enterprise we are involved in" (p. 69) . In other words, that which may be inexpressible and tacit in one CoP may be "easily" expressible in another CoP whose joint enterprise is different. In order not to confuse Polanyi's (1983) use of the term tacit knowledge with that described by Nonaka and Takeuchi, which we discuss in the following, we use the term implicit knowledge of that which may be difficult to express.

Only some part of knowledge/knowing is transferable in an abstract and explicit way. CoPs alleviate the problem by strategies that achieve Learning by doing, socializing and telling stories, which will indirectly include extra dimensions in knowledge transfer without needing the same level or type of abstraction. The narratives include the context of each situation that indirectly may infer these implicit aspects. The scope of interpretations increases when we abstract. In doing, socializing and telling stories we can direct, align, combine, and recreate our understandings to get a clearer picture, in order to narrow or redirect the scope. Through stories people build up a repertoire for improvisation. Narratives are reactivated by adding new elements. They naturally integrate the implicit elements as well as the explicit and are tuned to balance between

content and context. In seeing texts, mathematics and books as examples of the embodiment of formal abstractions, we can infer that these abstractions in the form of artefacts like books, represent knowledge made durable in a way that allows explicit knowledge accumulation and transfer. The transfer of implicit knowledge is seen to be more cumbersome. However we believe that the "simple" artefact as exemplified by the mason's wooden template is the embodiment of part of the gothic architects acquired implicit knowledge/knowing. The use of the technology of a template is an embodiment of parts of the explicit knowledge that does without the formal mathematical kind of abstraction. In lack of a CoP with a narrative way of transferring some of the implicit aspects, the template will perform a similar job. The template accumulates and transfers knowledge/knowing in a less formal and less abstract fashion which is durable, scales and transfers differently and perhaps better, than structural mechanics and mathematics.

We find that Wenger's theory of CoP with its *reifications* misses out on this formative aspect, that technology may hold in that it fails to recognize that different characteristics of different technologies as exemplified by the book, the template, and the liquid container.

In leaving the dichotomies of the explicit versus tacit (implicit), body versus mind, and individual versus organization behind in regards to knowledge transfer and accumulation, we argue that the dichotomy of humans versus artefact can be left behind, too.

7. Conclusion

In the introduction the same question were asked in two different ways. By rephrasing the questions our intention was twofold. First, to illustrate how different types of questions focus our attentions differently, and thus lead us towards different approaches in our understanding of collaboration. Second, to "implicitly" prepare the reader on the content of the rest of the paper, and hopefully provoke the reader to reflect a bit on the issue. In short the first type of questions emphasised the community aspect of collaboration–the "what" questions–while the second type of questions were directed towards the practice part of collaboration–the "how" questions. Our intention was not to favour any of the approaches, but to stress the importance of both and illustrate how they accent different aspects to our understanding of collaboration.

To sum up we demonstrate how a focus on the technology might provide different insights to the CoP example of Orr's service technicians and how the social position of CoP gives additional insights to the examples of Turnbull's templates and Berg's liquid vessels.

Turnbull illustrates that technologies as abstractions, in this case as a wooden template, can hold and transfer knowledge as design information between communities with similar community skills/knowing in effect communities that

have the skill to build with brick and mortar. The template works as a boundary object that traverses the community boundaries through both time and space, and comes across with a similar meaning, close enough to enable another master builder to decide to build a gothic rather than a Romanesque church. If this story looses sight of the technology, the artefact, then the transferral of knowledge becomes a mystery. The powerful qualities of this simple artefact are vital to the whole "plot". It scales better than the numerical mathematics, on which we rely today, in that it transcends language barriers and non-existent structural mathematics and it is durable in withstanding wear and tear. It travels well. So, just any technology will not do. Technologies have different characteristics which relate differently to different societal factors. Which technology is best at any point in time and setting will depend of the whole dizzying network of factors that make up and influence our social world, including the artefacts and what reifications we may establish in our communities. In analyzing possible relationships between the social and the non-human, and focusing at least equally on both, we may identify aspects of technology that grant us to be better equipped in reaching our goals.

Berg describes a use of technology where the artefacts are links in a production chain. Loose the liquid-container's specific qualities and the process is seriously hampered. The containers design is a product of knowing how best to collect and transfer the liquid in question into abstractions suitable for their entry into the liquid chart. Now this particular example is not so advanced as to render it impossible to establish a workaround if the vessel should disappear, but it clearly illustrates the distribution of responsibility and control, power and action into the separate links. The end link of the chain need have no suitable knowledge of what the whole process is about, let alone the differing links within the chain. There is no social interaction involved in the production of the end result in relation to a specific patient. The activities of the communities that designed the different artefacts may be long gone and the resulting process chain can scarcely be described as a community. However, if one look at the human actor as constituent of a particular link in the chain, CoP would see this actor as a part of a community where probably several people carry out that same activity for different patients. The liquid vessels would be the boundary object mediating the interaction with the next human actor in the chain. In effect the CoP based analyses focuses on the human actors because you start out by looking for the communities and what defines them.

Orr's service technicians discuss the technology in their community through sharing stories. Through these narratives of humans and artefacts, the technicians iterate, rephrase, recombine various bits of knowledge and experience to build new knowledge, knowing and tactics in coping with the machines. Their stories are their common stored knowledge, which sit in their collective memory and make sense in light of different contexts and experiences. Wenger uses this example to stress the importance of the community's collective work of producing the knowledge that enables them to carry out their work. However, through these stories, the machines gain a life of their own. The fact

that contexts vary, different machines of the same make behave both similarly and differently, is constantly contributing to and feeding the activity of the community. In this case the artefact need not be seen as a boundary object mediating meaning between communities, but also an actor with its own agenda, albeit based on their initial design. The qualities of the machines are highly relevant not only as the focal point of the CoP of service-technicians but also as part of the community, or as actors in the CoP as ANT would allow.

References

Berg, M. (1997). On Distribution, Drift and the Electronic Medical Record: Some Tools for a Sociology of the Formal, Proceedings of the Fifth European Conference on Computer-Supported Cooperative Work, ECSCW'97, Kluwer, pp. 141-156.

Brown, J.S. and Duguid, P (1991), Organizational Learning and Communities of Practice: A unified View of Working, Learning and Innovation, Organization Science, Vol. 2, No. 1, pp. 40-56.

Cook, S.D.N and Brown, J.S. (1999), Bridging Epistemologies: The Generative Dance between Organizational Knowledge and Organizational Knowing, Organization Science, Vol. 10, No. 4, pp. 381-400.

Contu, A. and Willmott, H (2003), Reembedding situatedness: The importance of power relations in learning theory, Organization Science Vol. 14, No. 3, pp. 283 – 296.

Hollan, J., Hutchins, E., and Kirsh, D. (2000), Distributed cognition: Toward a new foundation of human-computer interaction research, ACM Transactions on Computer-Human Interaction, Vol. 7, No. 2, pp. 174-196.

Hutchins, E. (1995). Cognition in the Wild, Cambridge, MA: MIT Press.

Lave, J, and Wenger, E. (1991), Situated learning: legitimate peripheral participation. Cambridge: Cambridge University Press.

Mol, A. (2003), The body multiple: Ontology in Medical Practice, Durham, UK: Duke University Press.

Naur, P. (1992), Programming as theory building, in P. Naur Computing: A human activity", ACM Press.

Nonaka, I. and Takeuchi, H. (1995), The Knowledge-Creating Company, Oxford, UK: Oxford University Press.

Orr, J. (1996), Talking about machines: An ethnography of a modern job, Cornell, CA: Cornell Univ. Press.

Polanyi, M, (1966) *The tacit Dimension*, Routledge and Kegan Paul, London.

Ryle, G. (1949), *The Concept of Mind*, London, UK: Penguin.

Timmermans, S. and Berg, M (2003), The practice of medical technology, Sociology of Illness & Health, Vol. 25, No. 3, pp.97-114.

Turnbull, D. (1993), The ad hoc collective work of building Gothic cathedrals with templates, string and geometry, Science, Technology and Human Values, Vol. 18, No. 3, pp. 315-340.

Paper 3

Jaccheri, L., and Østerlie, T. "Can We Teach Empirical Software Engineering?", in *Proceedings of the 11th IEEE International Symposium on Software Metrics (Metrics 2005)*, Como, Italy, September 19-22, 2005, pp. CD-ROM.

Can we teach Empirical Software Engineering?

Letizia Jaccheri and Thomas Østerlie
Department of Computer and Information Science (IDI)
Norwegian University of Science and Technology (NTNU)
letizia.jaccheri@idi.ntnu.no, thomas.osterlie@idi.ntnu.no

Abstract

We report about an empirical software engineering course for PhD students. We introduce its syllabus and two different pedagogical strategies. The first strategy is based on individual learning and presentations. The second relies also on social activities to support learning and knowledge sharing. The syllabus, which has been used for three iterations of the course, is available at our web site together with student essays, evaluation data, and other documentation produced during course runs.

1. Introduction

Empirical software engineering (ESE) is a sub field of software engineering which aims at applying empirical theories and methods for the measuring, understanding, and improvement of the software development process in organizations. ESE is by its nature a multi-disciplinary field as software engineers, industry actors, statisticians, pedagogues, and psychologists have traditionally been cooperating.

In this paper, we report on a PhD level course in empirical software engineering that has been run three times. The course held during the autumn of 2002 was based on individual presentation. During the spring of 2003 the course was based on group work held. This was replicated when the course was held in 2004.

The main objective of empirical software engineering education is to train software engineers in empirical evaluation of the tools, techniques, and technologies used in software engineering. It is in this context, that we see the importance in discussing the strategy for teaching empirical software engineering.

We are of the opinion that ESE is relevant for both practitioners and researchers. For practitioners it is about evaluating tools and techniques for use in concrete cases [18]. While it is equally important to teach ESE to each of these two groups., we report on a course for teaching ESE to software engineering researchers in this paper. We believe that our findings are equally applicable for teaching ESE to practitioners.

There are some fundamental challenges for an ESE research project to succeed. First, researchers in general and PhD students in particular must be well acquainted with existing methods. Second, ESE research is a major undertaking and it is a cooperative activity within a research group. Third, the research needs to be relevant outside the research community. The research group must therefore have access, knowledge of and familiarity with the software industry in order to study concrete and real situations, and to generate industry relevant research questions. Lastly, the research problems must also be relevant within the academic field of software engineering. Research problems therefore have to be significant to both the international research community and the local industry. This means that research problems and questions must be shared and understood by the all members of the research team and by the industrial actors. These challenges need to be addressed by a PhD level course in empirical software engineering.

Our course is an attempt to make our PhD students acquainted with the state of the art within ESE as well as reflect on investigations done by others and in which they have possibly participated.

Our course has been run three times and its syllabus, program, and evaluation is available at [16]. We have evaluated the pedagogical effects of the course by exploiting Bloom´s taxonomy of learning (which is well known and used by the software engineering community) and qualitative methods for data collection and analysis [29] as applied in the ESE field.

This paper is structured as follows. Section 2 introduces the aspects of software engineering education which have been relevant to our work and some learning issues in research education. Section 3 describes our course, its syllabus, pedagogical goals

11th IEEE International Software Metrics Symposium (METRICS 2005)
1530-1435/05 $20.00 © 2005 **IEEE**

and the two different strategies. Section 4 is about the evaluation of the course. Discussion and conclusions are given in section 5.

2 Background

Software engineering, as a field, has, among others, two supporting disciplines–software engineering education and ESE. Software engineering education focuses on training "software professionals" for the industry [25], while ESE focuses on evaluating the tools and techniques used, developed, and intended for use in the industry through empirical validation. There is an element of training required in making the transition from trained software professional, with or without working experience, to become an ESE researcher.

2.1 Software Engineering Education

The software engineering education literature moves along a dual axis; one axis for education content and one for pedagogy. We use the two axes to reflect on the current state within software engineering literature in this section.

The education content axis is delimited by two extremes: industry driven and principles driven. Meyer [25] argues that the contents of software engineering education must be driven by the principles on which software engineering is based:

"What matters is teaching [the students] fundamental modes of thought that will accompany them throughout their careers and help them grow in this ever-changing field. The ones who blossom are those who can rise beyond the tools of the moment in harmony with the progress of the discipline. ([25] p.29)"

On the other end is the work of Lethbridge [24]. Based on a survey of 168 software engineers, he finds significant differences between curricula taught at colleges and universities and the actual knowledge required in the industry. Lethbridge argues for aligning existing curricula with skills required by the industry. Where Meyer [25] is specific on the need to distance education from the industry's immediate, short-term requirement, Lethbridge writes little of the long-term requirements that Meyer addresses with his principles.

In this sense, their approaches can be classified as addressing short-term requirement vs. addressing long-term requirements. Guidelines for Software Engineering Education [4] adopt a middle ground approach to education contents. They address the long-term issues and are based on the body of knowledge for software engineering [1]. This body of knowledge, however, is based upon expert opinions within the industry.

Along the second axis, there are two strategies: the first strategy is based on lectures and individual learning. The second strategy is based on learning by doing, also known as project-based learning. Both Meyer [25] and [4] favor project-based learning. Unlike the education content, they argue in favor of project-based learning to "prepare our students for the real challenges they will face" ([25] p.33). They also argue that it is easier to learn from personal mistakes rather than mistakes related by a lecturer.

2.2 Learning issues in research education

Provided that it is possible to teach somebody how to become a good researcher, there are three kinds of courses that can be offered as part of research education:

1. General courses on research methods at both undergraduate and post graduate level are usually offered by social science faculties. These courses address research issues such as scientific method and nature of evidence, advocacy versus evidence-based approaches, writing and reviewing research proposals, how to use bibliographies and citation searches, project planning, selecting results and places to publish, outlining and structuring research papers, the peer review process, presenting posters and papers at conferences, publishing in academic and engineering journals, etc..[5]
2. Courses on research methods in computer science address some of the research issues above, such as scientific method and nature of evidence, customized to the IT field. At our department, for example, there is a common introductory course for all PhD students, which addresses general research issues in IT like those discussed for example in [12].

Table 1 Bloom's taxonomy of learning customized to ESE

Level	Definition	Sample verbs	Sample behavior	Sample behavior ESE
Knowledge	Student recalls or recognizes information, ideas, and principles in the approximate form in which they were learned.	Write List Label Name State Define	The student will define the 6 levels of Bloom's taxonomy of the cognitive domain.	The students will be able to define the content of the different papers.
Comprehension	Student translates, comprehends, or interprets information based on prior learning.	Explain Summarize Paraphrase Describe Illustrate	The student will explain the purpose of Bloom's taxonomy of the cognitive domain.	The students will explain the purpose of the give methods and investigations.
Application	Student selects, transfers, and uses data and principles to complete a problem or task with a minimum of direction.	Use Compute Solve Demonstrate Apply Construct	The student will write an instructional objective for each level of Bloom's taxonomy.	The student will be able to use one method for experimentation.
Analysis	Student distinguishes, classifies, and relates the assumptions, hypotheses, evidence, or structure of a statement or question.	Analyze Categorize Compare Contrast Separate	The student will compare and contrast the cognitive and affective domains.	The students will compare and contrast different methods and investigations.
Synthesis	Student originates, integrates, and combines ideas into a product, plan or proposal that is new to him or her.	Create Design Hypothesize Invent Develop	The student will design a classification scheme for writing educational objectives that combines the cognitive, affective, and psychomotor domains.	The students will design an investigation by choosing and perhaps combining different methods.
Evaluation	Student appraises, assesses, or critiques on a basis of specific standards and criteria.	Judge Recommend Critique Justify	The student will judge the effectiveness of writing objectives using Bloom's taxonomy.	The students will judge the effectiveness of using empirical software engineering methods.

3. Courses like the one we describe in this paper, which address empirical research methods in software engineering. These courses do not exist in isolation but in education and research context that may or may not include general research courses or courses specific for IT research. As far as we know, there is at least one paper that reports on teaching ESE [15] as part of a software engineering course. Undergraduate students work in projects, and the teachers play the role of the customer. In one project the customer is a hypothetical company wanting the students to perform empirical studies (mainly experiments) to evaluate different alternative techniques, e.g., different kinds of reviews. Students are not asked to plan the studies but only to perform them, as planning and running would take too much of the course. The syllabus of the empirical software engineering part is [32].

2.2 Software engineering education and research education at IDI

The software engineering group at IDI has thirty years experience with project based software engineering education. One of the author of this paper, Jaccheri, has more than ten years experience with teaching project based software engineering, software quality and metrics issues to software engineering students, both in Italy and in Norway, as well as reflecting and writing about this [17] [18] [19]. IDI has 7 years experience with an introductive course for IT researchers, which is mandatory for all new PhD students. Jaccheri has had the main responsibility for this course for one year. IDI graduates twenty PhD candidates each year and the software engineering group has graduated a total of 19 software engineering PhD students.

3. Empirical software engineering course

The course reported in this paper is offered to PhD students within software engineering. While the course has been run three times, we mainly report on the first two times as the third time we ran the course as a replication of the second time. The first and the second times had some common characteristics and some different characteristic. In the following section we discuss the common characteristics and afterwards we introduce the characteristics that were different for the two.

3.1 Common characteristics

In the following section we discuss the common characteristics for the course.

3.1.1 Students

The empirical software engineering course counts as 7.5 European Credit Transfer System (ECTS). This is equivalent to 12 hours work for 15 weeks, 2 hours in class and 10 hours of other learning activities, for a total of 180 hours. The course was run once during the autumn of 2002 at the Norwegian University of Science and Technology (abbreviated NTNU) for PhD students by the software engineering group, and once during the summer of 2003 at Simula Research Laboratory in Oslo for both University of Oslo and NTNU PhD students.

Some of the students worked in empirical software engineering research projects, while others worked in other kinds of software engineering projects. All students had a general course about research methods in IT as part of their curriculum. Some students even had a course on research methods in general. The students age, gender, nationality, and scientific background differed.

3.1.2 Syllabus

The course has had the same syllabus throughout. Here, we refer to a consolidated bulk of literature, which is used as syllabus for the course.

The syllabus is divided into three parts: motivation, method, and actual investigations.

- **Motivation**: In [33] and in [31], motivations for the existence of the ESE field are given. These papers provide also a classification of existing software engineering research papers according to the kind of empirical method used in the respective research.
- **Methods**: [32] provides an introduction to the field, with special emphasis on experiments. [22], [9], [28], [3], [8], [29], [2], and [14], provide concrete methods for performing, and analyzing investigations. [27] and [6] are about Data Analysis Methods.
- **Actual studies**: [13], [23], [11], and [28] are about concrete investigations.

The course ended with a final oral exam with the teacher and an ESE expert outside of the university as examiner.

3.2 Differing characteristics

Here we introduce the characteristics that differed.

3.2.1 Pedagogical goals and teaching strategy for Autumn 2002

In the autumn 2002 iteration of the course, the goal was to make the students acquainted with the contents of the syllabus. The course was held in a classroom context where students were met two hours a week for 13 weeks (totally 26 hours in class).

At each meeting, one student presented one paper from the syllabus. The teachers responsible for the course provided feedback and stimulated discussion around the paper.

3.2.2 Pedagogical goals and teaching strategy for Summer 2003

In the summer 2003 version of our course, the pedagogical goals were:

1. *given our syllabus which introduces a possible overview of empirical software engineering knowledge, let our students know about the syllabus at a level with is as high as possible according to the Bloom´s taxonomy [7].*
2. *establish a Norwegian network of young researchers within the field of software engineering.*

Bloom's taxonomy is reported in Table 1. The last column of the table (Sample behavior ESE) gives our interpretation of the taxonomy when applied to the ESE domain.

The second iteration of the course was organized as a three days event (21 hours). Here we list in chronological order the main tasks of the course.

a. Students were informed about the syllabus and the course web page [16]. One month before course start and were asked to read the entire syllabus before the course started.
b. Short introduction to the course content by the teacher.
c. Introduction to the field of empirical software engineering by discussing examples of interaction with the Norwegian software industry by two Norwegian research managers.
d. Group work with the goal of extracting the main issues from the syllabus. Each group had to find and list research hypotheses, data, and their analysis. Moreover each group had to summarize one method to plan and perform investigations, and to summarize one investigation.
e. Practical exercises coordinated by a performance and theater instructor with the goal to introduce students to performance work and to stimulate cooperation among students.
f. Production of a five minutes movie advertising the field of empirical software engineering.
g. Choosing an actual study in the syllabus, characterize the investigation presented in the chosen study according to motivations, method, measurements, and data analysis method. The goal of this exercise was to simulate the planning and execution of an empirical investigation.
h. Group work with the goal of obtaining a deep and critical understanding of the issues presented in the syllabus, also in light of the experience acquired during the previous exercises. Each group had to: choose one research hypothesis and its motivation (business, education, others); list and comment investigation planning and risks (what can go wrong); define (or reuse) guidelines for designing and running empirical investigations; choose one actual investigation.
i. Essay writing. Each student had to write an individual essay. The essay assignment was "*Extract from [22] and from the other papers in the Methods part of the syllabus, the most important guidelines for designing and running empirical investigations. Characterize the investigation presented in the chosen actual study, according to these guidelines. This means that you have to comment about the motivations, the method, the measurements, and the data analysis method*". The essay was supposed to be handed in two weeks after the seminar and one week before the oral exam. Recalling that students have 180 hours allocated to this course, we expected that students worked full time writing the essay during these two weeks.

4. Course evaluation

Based on running the course twice we wanted to evaluate it in order to plan its third iteration. One goal of this evaluation was to get general suggestions for improvement. We wanted to reflect about the two ways of teaching ESE–individual presentations or group assignments.

Educational evaluation is a sub field of educational research for which there is an extensive bibliography along with national and international standards [26]. To the best of our knowledge our course is one if the few offered internationally within the field. The number of new software engineering PhD students in Norway is of the magnitude of ten. While there is a need to teach such a course and to reflect over its

learning effects and the benefits and risk of different pedagogical strategies, we are aware that time is not mature for a formal evaluation of the course involving professional pedagogues and psychologists of the education service at our university.

However, we have designed and run an evaluation of the two first iterations of the course. Our evaluation attempts to reflect about how much the students had learnt from attending the course.

We decided to implement our investigation as an e-mail-based questionnaire [29] that we circulated to all students attending the courses.

We decided to use Bloom's taxonomy. The columns Level, Definition, Sample verbs, and Sample behavior are taken directly from the Bloom taxonomy of educational objectives [7]. Column "Sample behavior ESE" is our contribution.

We decomposed the level of learning reached by students with respect to the Bloom taxonomy in three categories, one for each part of the syllabus: motivation, method, and actual studies (see section 3.1.2 Syllabus). We wanted to know how much each student has learnt from each part of the syllabus.

The questionnaire had to be formulated in such a way that we could assess what the students have learnt and how well they have learnt. To this end we formulated the two questions:

- *When did you take the Empirical software engineering course? (Autumn 2002 or Summer 2003?)*
- *Have you ever participated in an Empirical software engineering investigation? If yes, prior or after attending the course?*

Our goal was to measure how much of the course contents the students had learnt from attending the empirical software engineering course. We formulated the following question in order to measure how much of the course contents the students had learnt:

- *In your own words, what were the primary objectives of the Empirical software engineering course?*

We did not ask directly what the students had learnt, but what they thought were the primary objectives of the course. We did this to make sure the students answered what they had learnt from the course contents, not what the teachers were supposed to teach them. To see how well the course succeeded in teaching the students the overall goal, to run empirical evaluations, we asked the following:

- *On the basis of what you have learned in this course, would you be able to plan and run an empirical investigation?*

Finally, we formulated the final question:

- *Do you have further comments on the course?*

If we had asked the wrong questions, we hoped the students would tell us so in answering this question. We were also interested in feedback on how to better the course, and thought this question provided an opportunity for such feedback.

Based on lists of the participants in the ESE course, we circulated the questionnaire to all participants by e-mail. We included an introductory letter explaining that we wanted to use the data for both improving the course and as material for a paper, that all responses would be treated confidentially, and a date within which we would like to receive the responses by. The responses to our questionnaire were on free form and are available at [16].

We e-mailed the survey to the seventeen students attending the two courses. In the end, due to data discard, we had ten responses to analyze.

The data from the questionnaire and our analysis can be found at [16].

- The "stories" written by students provide a valuable piece of knowledge for those evaluating our course. To the question "In your own words, what were the primary objectives of the Empirical software engineering course?".

Table 2: Coding conventions.

	Word/phrase in free form question	**Translated meaning**
Coding key 1	Learn insight overview aware	• Bloom's taxonomy: knowledge level • Course topic: all
Coding key 2	Run experiment	• Bloom's taxonomy: application level • Course topic: actual studies
Coding key 3	Plan and run experiment	• Bloom's taxonomy: synthesis level • Course topic: actual studies
Coding key 4	Tradition	• Bloom's taxonomy: depends on other coding keys • Course topic: motivation

To the question "On the basis of what you have learned in this course, would you be able to plan and run an empirical investigation?" students write answers like:

- *"Yes. But more concrete skills about how to design questionnaire and how to analyze data should be learned".*
- *"Not without more input, but I would have a better starting point for the planning phase. A lot of the work in the planning phase would consist of deciding how the investigation should be formed, and I think I have a better understanding about the basic concepts and where I could find out more about them."*

To analyze our data, we employed the method of coding [25] for extracting qualitative data and map them into Bloom's taxonomy of learning. Based on the responses and our interpretation of Bloom's taxonomy, we therefore formulated a set of coding keys to ensure identical coding of free form replies for both the taxonomy and the course topics (Table 2).

- Coding key 1 enabled us to translate sentences like "Provide some insight into typical methods used in the field" into level *knowledge* for topic *method.*
- Coding key 2 enabled us to code sentences like *"Yes - it would give me an important starting point, but I would always need to confer and discuss afterwards possible investigations and their plans with people with experience"* into level *application* for topic *actual studies*.
- Coding key 3 says that whenever a student declares to have knowledge to plan an experiment, the level for actual studies is *synthesis* since Create, Design, Hypothesize, Invent, and Develop are sample verbs for synthesis level in the Bloom taxonomy. Example is a sentence like: "Yes, I can design an empirical study now".
- Coding key 4 says that whenever a student acknowledges that the goals of the course encompass the traditions of the field, we assigned a value to the *motivation* field. Example: in response to the question "In your own words, what were the primary objectives of the empirical software engineering course?" one student replied *"To given an overview of the main methods and traditions of empirical software engineering".*

Along with the coding keys, we made extra rules for coding that can be found at [16] together with the data set.

There is a slight difference between the levels obtained for autumn 2002 and summer 2003. However, taken in consideration the number of subjects (10) and the measurement scale (ordinal), it does not make sense to try statistical generalization.

5. Discussion and Conclusions

The ESE course is now mandatory for all new software engineering PhD students at NTNU. We had only 4 new software engineering PhD students in 2004. For this reason, the teacher (Jaccheri) decided to organize the 4 students as a group with which she interacted as a performing member for the third iteration of the course.

The transition from 2003 to 2004 benefited from the evaluation we report in this paper. The 2004 iteration is organized in a similar way to 2003 and it is based on group activities. The most important change with respect to 2003 is the introduction of a "summary writing" activity that proceeds the seminar. This is an individual activity during which each student is supposed to write a structured summary of the main characteristics of the content part. Students get the assignment one month before the seminar. In this way, the 2004 iteration is more oriented toward individual learning that the 2003 iteration but still much more group oriented than the 2002 one. The learning goal of this summary activity is that students must have reached a knowledge level of the Bloom´s taxonomy before seminar starts. The next version (fourth) of the course will run during autumn 2005.

The main contribution of this work is the description of a course in the field of empirical software engineering. First, the presented syllabus [16] can be used as a basis for a dialog in the ESE community about which topics are of most importance when educating the future researchers in the field. Second, the two presented pedagogical strategies can be discussed further to find out which one or which combination of the two is better suited for which context. Another contribution is our customization of Bloom´s taxonomy of learning to the ESE field.

Like every other course, there are at least two axes: one axis for education content and one for pedagogy. These two axes can be used also to reflect over this course or similar ones. Which part of the content are we satisfied with and why? Which pedagogical strategy work better in which situation? According to our evaluation, students are generally satisfied with syllabus and the way it is structured into motivation, methods, and actual studies. Students appreciate examples provided by the papers in the investigation part. Concerning the method part, there is need to focus more attention on case studies and interpretative studies. Teachers and examiners are satisfied with the

essays and we believe that the guidelines provided in [22] are a good tool to characterize investigations

Other ESE courses are offered at other universities, like the one documented in [15]. However, if one asks colleagues about how they learnt to become researchers in software engineering, or more specific empirical software engineering or measurement, the answer is often vague. Experienced researchers rely on tacit knowledge which is learnt and shared locally by research groups and internationally by research communities. A contribution of this paper is that it can be an example for those who provide and want to provide similar courses.

Concerning our evaluation, we are aware that the evaluation of the two pedagogical strategies has limitations if one regards it as a piece of educational research evaluation. First the sample size (10) is too limited. Second we ask students to evaluate their own learning perception. Third, though we map the initial knowledge of each student, we do not take gender, age, language, and country of origin into consideration. Students have different age, gender, nationality, and scientific background. While heterogeneity enhances learning challenges and possibilities, it poses serious problems to a formal evaluation of the learning effects of the course.

However, the evaluation, in addition to content and pedagogy, are relevant to the ESE community and to the software engineering community. This because despite several empirical studies being conducted with students as subjects [10], there is a lack of frameworks for evaluating the learning effects of software engineering courses. Our work is of interest for the measurement community as it can serve as an example for those who want to evaluate the pedagogical effects of courses where such experiments take place.

One of the goals of our course is that of establishing a Norwegian network of young researcher in the field of software engineering. From the students' questionnaire answers and for living and working together with these students, we evaluate the project based version of the course to be successful to respect to this goal.

There remain three open questions. First, that of comparing our course with other similar courses offered internationally. Secondly, one question which is not attacked by our work is: is it possible to teach PhD students how to cooperate with external actors? The final issue that we have not discussed is empirical software engineering education for practitioners and not only for researchers.

6. Acknowldge

We thank all PhD students who participated to the courses and who answered our questionnaire. We are in debt to Reidar Conradi and Monica Divitini who set up the first version of the syllabus and organized the first version of the course. We thank Tore Dybå for useful discussions and suggestions on the syllabus part. Special thanks go to Dag Sjøberg and the Simula center group for giving us the possibilities to run the second and third version of our course.

7. References

[1] A. Abran, J. W. Moore, P. Bourque, R. Dupuis, and L. L. Tripp, "Guide to the Software Engineering Body of Knowledge, Trial Version SWEBOK, A Project of the Software Engineering Coordinating Committee," IEEE, May 2001.

[2] E. Arisholm, B. Anda, M. Jørgensen, and D. I. K. Sjøberg, "Guidelines on Conducting Software Process Improvement Studies in Industry," presented at 22nd IRIS (Information Systems Research Seminar In Scandinavia) Conference, Keruu, Finland, 1999.

[3] D. E. Avison, F. Lau, M. D. Myers, and P. A. Nielsen, " Action research," Communications of the ACM, vol. 42, num. 1, pp. 94-97, 1999.

[4] D. J. Bagert, T. B. Hilburn, G. Hislop, M. Lutz, M. McCracken, and S. Mengel, "Guidelines for Software Engineering Education Version 1.0 (1999)," CMU/SEI CMU/SEI-99-TR-032, 1999.

[5] J. Bell, Doing your research project. Buckingham . Philadelphia: Open University Press, 230 pp., 2000.

[6] A. Birk, T. Dingsøyr, and T. Stålhane, "Postmortem: Never Leave a Project without It," IEEE Software, vol. 19, num. 3, pp. 357-390, 2000.

[7] B. S. Bloom, Taxonomy of educational objectives: The classification of educational goals: Handbook I, cognitive domain. New York, Toronto: Longmans, Green. 1956.

[8] L. Briand, E. Arisholm, S. Counsell, F. Houdek, and P. Thevenod-Fosse, "Empirical Studies of Object-Oriented Artifacts, Methods and Processes: State of the Art and Future Directions," Empirical Software Engineering, num. 4, pp. 385-402, 1999.

[9] L. C. Briand, S. Morasca, and V. R. Basili, "An operational process for goal-driven

definition of measures," IEEE Transactions on Software Engineering, vol. 12, num. 4, pp. 1106- 1125, 2002.

[10] J. Carver, L. Jaccheri, S. Morasca, and F. Shull, "Issues in Using Students in Empirical Studies in Software Engineering Education," presented at International Software Metrics Symposium, Sydney, Australia., 2003.

[11] R. Conradi and T. Dybå, "An Empirical Study on the Utility of Formal Routines to Transfer Knowledge and Experience," presented at 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, Vienna, Austria, 2001.

[12] P. J. Denning, "Computer Science: The Discipline," Encyclopedia of Computer Science, 2000.

[13] T. Dybå, "An Instrument for Measuring the Key Factors of Success in Software Process Improvement," Empirical Software Engineering, Kluwer Academic Publishers, vol. 5, pp. 357-390, 2000.

[14] N. Fenton, "Software Measurement: A Necessary Scientific Basis," IEEE Transactions on Software Engineering, vol. 20, num. 3, pp. 199-205, 1994.

[15] M. Höst, "Introducing Empirical Software Engineering Methods in Education," presented at Conference on Software Engineering Education and Training CSEE&T, Covington, Northern Kentucky, USA., 2002.

[16] L. Jaccheri and T. Østerlie, "Empirical software engineering Course," Trondheim - Oslo, Norway, http://www.idi.ntnu.no/emner/empse/, last accessed February 2004.

[17] L. Jaccheri, A software quality and software process improvement course based on interaction with the local software software industry, in the Tutorial section of Computer Applications in Engineering Education Journal, John Wiley and Sons, Inc. page 265-272, Mar 2002.

[18] L. Jaccheri and P. Lago, Metrics aspects of a software engineering course project, September 1997, in Proc. of Inspire II, 2nd International Conference on Software Process Improvement - Research into Education & Training, Sweden.

[19] L. Jaccheri and P. Lago, "Applying Software Process Modeling and Improvement in Academic Setting", April 1997, Proc. of the 10th ACM\IEEE-CS Conference on Software Engineering Education and Training, Virginia Beach.

[20] J. Kirk and M. L. Miller, Reliability and Validity in Qualitative Research, vol. 1. Newbury Park, California: SAGE Publications Inc., 1986.

[21] B. Kitchenham, "DESMET: A Method for Evaluating Software Engineering Methods and Tools," Keele University, Technical Report TR96-09, August 1996.

[22] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, " Preliminary guidelines for empirical research in software engineering," IEEE Transactions on Software Engineering, vol. 28, num. 8, pp. 721 -734, 2002.

[23] O. Laitenberger and J.-M. DeBaud, "Perspective-based Reading of Code Documents at Robert Bosch GmbH," Information and Software Technology, vol. 31, num. 11, pp. 781-791, 1997.

[24] C. T. Lethbridge, "On the Relevance of software education: A survey and some Recommendations," Annals of Software Engineering, vol. 6, pp. 91/110, 1998.

[25] B. Meyer, "Software Engineering in the Academy," IEEE Computer, vol. 34, num. 5, pp. 28-35, 2001.

[26] Meredith D. Gall, Walter R. Borg, Joyce P. Gall, ISBN: 0-321-08189-7, Allyn & Bacon, 2003, 656 pp

[27] J. Miller, J. Daly, M. Wood, M. Roper, and A. Brooks, "Statistical Power and its Subcomponents: Missing and Misunderstood Concepts in Empirical Software Engineering Research," Information and Software Technology, vol. 39, num. 4, pp. 285-295, 1995.

[28] M. Morisio, M. Ezran, and C. Tully, "Success and Failure Factors in Software Reuse," IEEE Transactions on Software Engineering, vol. 28, num. 4, pp. 340-357, 2002.

[29] C. B. Seaman, "Qualitative Methods in Empirical Studies of Software Engineering," IEEE Transactions on Software Engineering, vol. 25, num. 4, pp. 557-572, 1999.

[30] D. I. K. Sjøberg, B. Anda, T. Dybå, M. Jørgensen, A. Karahasanovic, E. F. Koren, and M. Vokac, "Conducting Realistic Experiments in Software Engineering," presented at First International Symposium on

Empirical Software Engineering, Nara, Japan, 2002.
[31] W. F. Tichy, "Should Computer Scientists Experiment More?," IEEE Computer, vol. 31, num. 5, pp. 32-40, 1998.
[32] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, Experimentation in Software Engineering: An Introduction: Kluwer Academic Publishers. 2000.
[33] M. V. a. Zelkowitz and D. R. Wallace, " Experimental Models for Validating Technology," IEEE Computer, vol. 31, num. 5, pp. 23-31, 1998.

Paper 4

Østerlie, T., and Wang, A.I. "Establishing Maintainability in Systems Integration: Ambiguity, Negotiation, and Infrastructure", in *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, Philladelphia, PA, September 24-27, 2006, pp. 186-196.

Establishing Maintainability in Systems Integration: Ambiguity, Negotiations, and Infrastructure

Thomas Østerlie, Alf Inge Wang
Norwegian University of Science and Technology
thomas.osterlie@idi.ntnu.no, alf.inge.wang@idi.ntnu.no

Abstract

This paper investigates how maintainability can be established in system integration (SI) projects where maintainers have no direct access to the source code of the third-party software being integrated. We propose a model for maintainability in SI focusing on post-release activities, unlike traditional maintainability models where focus is on pre-release activities. Our model describes maintainability as a process characterized by ambiguity and negotiation that is supported through an infrastructure of debugging and coordination tools. Further, we describe how the process going from a software failure to establishing the fault causing the failure can be managed in SI. The results presented in this paper are based on observations from an ethnographic study of the Gentoo open source software (OSS) community, a large distributed volunteer community of over 320 developers developing and maintaining a software system for distributing and integrating third-party OSS software packages with different Unix versions.

1. Introduction

It has been repeatedly established over the past 30 years that more than half of the total life-cycle cost of software systems goes into software maintenance activities. The figures vary between 50 to 80 percent of the total life-cycle cost [6]. This research indicates that the maintenance burden has been increasing over the decades rather than decreasing. To face this challenge, *maintainability* has been proposed as a software quality measure. This measure is used to assess how easy it is to maintain a system and what decisions to make in design of a system to limit the maintenance costs. Existing research on maintainability builds on the premise of application software that is maintained by a single team of developers with full access to and control over the source code. However, with increasing attention on systems integration (SI) through component-based development [5], Web services [22], and information and enterprise systems integration [14], this may no longer be a valid premise. A number of distinguishing characteristics of SI diverge from application software: systems integrators usually have limited or no access to the source code of the software being integrated, and control over the development and maintenance of the software being integrated is carried out by numerous third-party organizations [11]. Given these differences, we ask: **how can maintainability be established in SI?**

In this paper, we seek to explore a possible solution to this problem; a solution that rests upon the premise of software maintenance as knowledge-intensive work. By studying the activities involved with reporting software failures and determining their related faults, we propose that corrective maintenance in SI unfolds within an environment of *ambiguity* [1]. Ambiguity is an uncertainty where the correct meaning of a phenomenon cannot be established given sufficient or appropriate information. Instead, ambiguity involves uncertainties that cannot be resolved or reconciled due to the absence of agreement on boundaries, clear principles, or solutions. Ambiguity means that multiple meanings or several plausible interpretations of the observed phenomenon exist, and their meaning can only be established through *negotiation*. The process of establishing certain interpretations of ambiguous phenomenon as stable scientific facts has been a primary concern within the field of science studies. In these accounts, this process is seen as unfolding within an *infrastructure* of experimental tools, scientific artifacts, social interaction, and practices [15]. It is an infrastructure of scientific facts; the behind-the-scenes, messy or boring items that form a crucial part of how facts are made.

Building upon the notions of ambiguity, negotiation, and infrastructures, we propose that *maintainability can also be understood as a function of the external environment within which the software is being maintained.* Maintainability is a function of the

infrastructure of tools used during maintenance, the texts produced by these tools, knowledge about the system embedded in the tools, and tools for supporting and coordinating interaction between developers. This supplements existing models that focus on maintainability as a function of characteristics of the application software. The proposed explanation is based on an empirical study of maintenance work in a large-scale open source software (OSS) integration project. OSS is well suited for studying software maintenance, as OSS development is often understood as a perpetual cycle of perfective and corrective maintenance [20].

Limiting our inquiry to the issue of maintainability in connection with corrective maintenance in SI, we study the activities involved with reporting software failures and determining the related fault. Through a detailed narrative analysis of these activities, we propose a model for the corrective maintenance process that supports our suggestion for establishing maintainability in SI.

The rest of the paper is organized as follows. Section 2 reviews existing research on maintainability and approaches to establishing maintainability during pre-release activities. Section 3 describes the research methods employed and the materials collected during our field study. Section 4 describes a detailed narrative analysis of the activities with reporting software failures and determining the corresponding faults. Section 5 concludes the paper by discussing our findings in relation to establishing maintainability in SI.

2. Related work

Intended as an indicator of the costs of maintaining a software system, maintainability can be broadly defined as the ease with which a software system can be understood and modified [10]. By making the software more maintainable, i.e. increasing its maintainability, organizations should be able to reduce the maintenance effort and free needed resources for more new system development. Maintainability can be viewed from different perspectives. In this section we presents two of these:

- establishing and assessing maintainability using software quality models; and
- making a system maintainable by using design techniques when creating the software architecture of the application

We then conclude the section by discussing the issue of maintainability in relation to OSS development.

2.1. Quality-based approaches

McCall [18] provides an overall description of approaches to developing software based on software quality frameworks. At the outset of a software development effort quality factors are identified based on the specifics of the software being developed. Maintainability is one such quality factor. Once the important factors are identified, they are specified as requirements of the systems development by providing their definition, identifying supporting software attributes, and providing measurements to assess their attainment. The software development is then periodically measured in a quantitative fashion to assess if the software product is capable of meeting its identified requirements. Based on this assessment of the software product's quality, decisions are made as to efforts needed to improve the software product. This process is repeated until the quality requirements, in this case the requirements for maintainability, are met and the product can be released.

Several approaches to assessing the software product's maintainability have been proposed. McCall [18], Martin & McClure [17], Boehm et al. [4], and ISO9126 define maintainability as a quality factor in their quality models. Wherein McCall limits maintainability to include only corrective maintenance, both Boehm et al., Martin & McClure, and ISO9126 provide definitions that encompasses both corrective, perfective, and adaptive maintenance. Boehm et al. defines maintainability to include the quality criteria testability, understandability, and modifiability. Martin & McClure argues for an expanded view of maintainability, arguing that its definition needs to be expanded with a high degree of reliability, portability, efficiency, and usability in addition to the attributes provided by Boehm et al. Landing on the middle ground, ISO9126 defines maintainability as analyzability, changeability, stability, and testability. In all of the above models, the quality criteria are broken into a set of metrics for measuring code characteristics.

2.2. Architecture approach

In the software architecture domain, software maintainability is a quality of the end-system the developer can obtain by carefully choosing the correct structures and making the correct decisions when designing the system. Different terms are used to describe maintainability.

In Bass et al. [2], maintainability is described in terms of the quality attributes modifiability and testability. Modifiability describes the costs of changing the system. Typical changes can be both changes of functionality as well as changes of non-

functional properties of the system like performance, availability, change of operating system etc. Testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing. To obtain a high level of modifiability and testability in a system, the developers must consider both architectural and non-architectural aspects. The architectural aspects typically concerns *important design decisions* that affect the way the software is organized, structured and decomposed.

Non-architectural aspects typically concern implementation details, graphical layout of user-interfaces etc. Bass et al. use the term *architectural tactics* for important design decisions that affects the software architecture. Several such tactics have been collected over the years based on experiences from several software projects. Examples of tactics to obtain high maintainability involves recommended design guidelines for object-oriented systems like maintaining semantic coherence, hide information, restrict communication paths, use of intermediary, etc. There are also similar tactics to obtain a high level of testability in a system.

2.3. OSS and maintainability

The OSS development cycle have three distinguishing characteristics. First, source code is made available on the Internet, released early, long before all functionality is in place and faults have been eliminated. Second, by releasing the software early, developers around the world can contribute code, adding new functionality and improving the present functionality. This is often called parallel development [9]. Parallel debugging is the third characteristic of the development cycle, wherein failure reports and fixes are submitted to the project. This process has been characterized as a perpetual cycle of perfective and corrective maintenance.

Seeing OSS development as software maintenance, the question can be raised whether the success of OSS development can be explained by the software's maintainability? In determining the categories of maintenance work in two large OSS products, 53.4% of all changes to the source code of these products stem from corrective maintenance [21]. Given that the cost of corrective maintenance are at least an order of magnitude more expensive to fix than those found during testing [7], the question concerning OSS success and maintainability becomes even more pressing.

In measuring the maintainability index of five OSS projects, Samoladas et al. [20] finds that OSS code quality suffers from the very same problems observed in closed source software (CSS) projects. Maintainability deterioration over time is a common phenomenon in CSS, and they project that is reasonable to expect this as OSS products age, too. In a comparison of OSS and closed source software products, Paulson et al. [19] investigates the claim that OSS succeeds because of code simplicity. Measuring the overall project complexity, average complexity of all functions, and average complexity of functions added, they find that for all three metrics the OSS projects had higher complexity than the CSS projects. Similarly, they compare the perfective maintenance of OSS and CSS by measuring the growth rate of the projects. They find that OSS and CSS have similar growth rate. Albeit based on a limited population, the inference from combining the conclusions of Samoladas et al. and Paulson et al. is that the maintainability of OSS and CSS is mostly the same.

Paulson et al. also reports that faults are found and fixed more rapidly in OSS projects. Holding to the definition of maintainability as the ease with which a software system can be understood and modified, questions may be raised with basis in these findings as to how to understand maintainability? It seems that commonly used maintainability metrics do not correspond to the actual facts of maintainability as measured in ease of which software systems can be understood and modified.

3. Methods and materials

This paper reports on one of the authors' study of software maintenance in a large OSS community. The study is based on the view that to better understand software engineering, "it is imperative to study … software practitioners as they solve real software engineering problems in real environments" [16]. As such, the study has been conducted as ethnographic fieldwork, expanding on a growing body of ethnographic studies of software engineering practice. Ethnography is a research method where the researcher participates with the subjects being studied. Through longitudinal observations of naturally occurring activities, the researcher builds an increased understanding of the object under study. However, if we want to understand how software is developed in practice, it is important not to start out assuming what we want to explain. Therefore the ethnographer does not give any prior significance to particular features of practice. Giving primacy to the empirical data, ethnography is a systematic approach for reaching empirically validated conclusions.

In Section 3.1 we present the research setting. In Section 3.2 we present the data collection process. In

Section 3.3 the data analysis process is presented, and in Section 3.4 we discuss the validity of our findings.

3.1. Research setting

This paper reports on an ethnographic study of the Gentoo OSS community. As of March 2006 Gentoo is made up of over 320 developers distributed across 38 countries and 17 time zones. We use the term community here about those involved with Gentoo, as users play an important role in OSS development [9]. Enumerating the number of users in the community is difficult because there are no lists of purchased licenses or registered users available.

Gentoo is a large systems integration project. Broadly speaking, the Gentoo community develops and maintains a software system for distributing and integrating third-party OSS software packages with different Unix versions. The software is distributed in the form of installation scripts, one script for every supported version of each package distributed. As of March 30 2006 Gentoo distributes one or more versions of 8486 software packages, for a total of 23911 installation scripts. As well as integrating software for 5 different hardware architectures for the GNU/Linux operating system, the installation scripts can also integrate software with both the MacOS X, FreeBSD, and OpenBSD operating systems. Such heterogeneity is a defining characteristic of integrated systems [11].

In distributing software developed by other OSS projects, the development and maintenance of these packages are outside the control of the Gentoo community. Such autonomy is also a distinguishing characteristic of integrated systems [11], but also manifest a variety of human interests and activities. In defining largeness of software systems, Belady & Lehman [3] find variety to be a distinguishing characteristic. In terms of largeness, the software distributed is outside the scope of a single individual and require not only one group of people to develop and maintain the software, but numerous groups; both the Gentoo community developing and maintaining the installation script and the third party OSS communities who develop and maintain the software distributed. Furthermore, the installation scripts developed and maintained by the community is also outside the grasp of a single individual. Gentoo is organized into 124 teams, each responsible for a discrete set of installation scripts.

There are complex interactions between parts of Gentoo, both technologically and socially. Complex interaction is another characteristic of largeness. Technologically these interactions manifest themselves in the specific relations between different packages and that the same package is supported both on different hardware platforms and for different operating systems. This is made further complex by the introduction of virtual packages, identical functions that are provided in different packages. Socially, the complex interactions are not only between members of the Gentoo community or among the teams, but also in the interface between the Gentoo community and the OSS communities developing the software distributed by Gentoo.

So far, we have used the term Gentoo without any clear definition. This is done on purpose, as the term itself is ambiguous. The term has three meanings. First, it is used for talking about the Gentoo community of developers and users. Second, it is used about the Gentoo GNU/Linux distribution. Sometimes the term Gentoo Linux is used to specify this. Third, Gentoo is a software system for distributing OSS software packages for different Unix implementations. The distributed packages are developed by third-party OSS projects, and the Gentoo community develops and maintains installation scripts for these packages. These scripts are made available through a central repository.

The term Gentoo is ambiguous; it is particularly problematic to draw a clear boundary between Gentoo Linux and the Gentoo software distribution system. At the heart of Gentoo Linux is the Gentoo distribution system. Historically, however, the distribution system has grown out of the Gentoo Linux distribution. The term Gentoo is used interchangeable between the two, and often used by developers as a means to avoid drawing the problematic boundary between the two. Technically speaking, there are both installation scripts and other files distributed by the Gentoo distribution system that are particular to Gentoo Linux. However, most installation scripts distributed are not specific for the GNU/Linux distribution.

The lack of consensus on boundaries is a trait of ambiguity. Both variety and complex interactions produces unclear technological boundaries and ambiguity in the Gentoo software product.

3.2. Data collection

The first author conducted the ethnographic fieldwork. We therefore present this section in first-person view. Participant-observation is the primary method for data collection in ethnographic fieldwork [12]. In this study this meant that I observed the Gentoo developers online through dedicated Gentoo IRC channels, dedicated mailing lists, the Gentoo Web site, and Web-based front-ends for Gentoo's defect tracking system and revision control system. My

participation included submitting and assisting in resolving bug reports, submitting installation scripts, as well as participating in a large restructuring effort of Gentoo's package manager. I used both Gentoo Linux and MacOS X with a Gentoo installation as operating systems on my workstations during the period of fieldwork. I made no formal interviews with participants in the Gentoo community, but conducted informal talks with participants on a regular basis to test my own informal theories.

Throughout the period, I made daily field notes [12]. In addition, the Gentoo IRC channels were logged to disk throughout the period of study; one file each day for each IRC channel totaling 1027 files. 71 documents were collected throughout the period and organized in a documentary database. I also surveyed online data sources that provide static data. These include the Gentoo bug tracking database, the Gentoo mailing list archives, and the Gentoo revision control system. As the Gentoo Web site is under revision control, I did not organize relevant documents from this Web site in the documentary database. Instead, I decided to rely on Gentoo's revision control system.

3.3. Data analysis

Ethnographic data analysis is an ongoing process from the moment the field worker enters the field until the complete research report is written. During field work the data analysis is informal. Upon withdrawing from the field, the data analysis is gradually formalized. Informal data analysis is a continuous activity through out the period of fieldwork. Because this analysis is closely connected with the daily details of fieldwork, there are no clear links between this analysis and the topics discussed in this paper. We have therefore opted for a more general description of the activities of informal analysis, and instead present the details of the formal analysis as this is closely connected with the topic of this paper.

Informal analysis takes the form of writing out the notes that have been quickly and briefly jotted down in the notepad during the day's observation, and organizing them into more coherent field notes. By relating the day's observations to previous field notes, I continuously looked for patterns in my observations for building informal theories. These informal theories, in turn, inform how I continued performing the fieldwork. This way, I was able to better fit the way I performed my fieldwork with basis in an increased understanding of the research setting.

Upon withdrawing from the field the first author spent a year working systematically through the collected data, looking for recurring patterns. Once the recurring patterns are identified and formulated, formal data analysis commenced. Formal data analysis is a process of incrementally generalizing from a multitude of singular observations to increasingly more generalized descriptions of activities. Throughout this process, non-recurring details of the singular observations are omitted and recurring issues included. However, determining which details to omit in the final analysis and which to include is an iterative process of working on generalizing the descriptions while continuously returning to the more detailed analyses looking for recurring patterns that may shed light on the generalized description.

During formal analysis we identified a set of bug reports in the Gentoo bug tracking system. The bug reports were identified to capture the width of bug reports submitted. The selection criteria were based on the field notes and experiences from the fieldwork. Upon identifying a set of relevant bug reports, we went through each report, reconstructing a time line for the bug report based on the bug report activity log feature provided by the defect tracking system. Into this time line we also placed discussions about the bug from the Gentoo IRC channel logs collected during the period of fieldwork, the Gentoo mailing list archives, and the Gentoo Web forums. In this timeline we simply cut and pasted from the various data sources. With basis in this, we wrote an executive summary of the bug report's life cycle as well as writing out a complete narrative of the bug report's life cycle with explanations.

With basis in these narratives, one for every bug report in the set, we started relating our data to theory. At this stage we focused on establishing relevance and context of our observations. We tried a number of theories for interpreting our data; ranging from social theories on decision-making, via theories on distributed cognition from the field of computer supported cooperative work, to more standard software engineering theory on software maintenance. From this analysis the focus on maintainability, which led us to the last part of the formal analysis, which is to write up the results and analysis presented in section 4.

3.4. Research validation

Ethnographic research does not follow a step-wise process. Rather, the data collection requires flexible responses to the specific circumstances of the moment. This flexibility also means that the research design changes in the face of in-field realities that the researcher could not anticipate at the outset of the study. It is therefore difficult to ground the study's validity in the procedural rigor of controlled

experiments. Instead, the validity is established through a rigor in argumentation by following the seven principles for conducting fieldwork [13] as shown in Table 1.

4. Results and analysis

Following the definition of maintainability as the ease with which a software system can be understood and modified, we are focusing on the aspect of system comprehension in this paper. In this section we discuss systems comprehension in relation to each of the three concepts raised in the introduction – ambiguity, negotiation, and infrastructure – relating them to the empirical data collected and existing literature. The main point is that systems comprehension is a collective process of generating a consensus-based comprehension of the system and how it causes the observed failures.

4.1. Ambiguity

Some software systems fail. A software failure is an externally observable error in the program behavior. Software failures are caused by software faults that are triggered under specific circumstances during execution. Upon experiencing a software failure that cannot be corrected locally, Gentoo users submit a bug report to the Gentoo defect tracking system (http://bugs.gentoo.org). The bug report is analyzed by Gentoo developers and resolved either by rejecting the reported failure as a real failure, by correcting the fault causing the failure, or by forwarding the report upstream. As the Gentoo developers repackage software developed by external OSS projects, forwarding bug reports upstream means that the failure is not caused by Gentoo specific code or interaction of components distributed by Gentoo, but found to be caused by faults in the third-party software. This is the overall description of Gentoo's corrective maintenance process.

Gentoo uses Bugzilla, a Web-based OSS defect tracking system. In Bugzilla, failures are reported as bug reports in a standardized Web form. Bugzilla provides a standardized schema for describing the failure and for administration of bug reports. This schema is mostly used for assigning bug reports and tracking their status. A recurring pattern in the use of Bugzilla is that the Gentoo users and developers use the optional text field at the end of the bug report, named *Additional comments*, during corrective maintenance. Why is that?

Table 1. Research validation

Principle	Description	Validation
1. The fundamental principle of the hermeneutic circle	This principle suggests that all human understanding is achieved by iterating between considering the interdependent meaning of parts and the whole that they form.	Discussion of the iteration between the day's findings and previous field notes during informal data analysis, and the process of working on generalized descriptions while returning in detailed analysis, Section 3.3.
2. The principle of contextualization	This principle requires critical reflection of the social and historical background of the research setting	Discussion of the shift from application software to SI, Section 1. Relating Gentoo to SI and discussing of the historical relationship between Gentoo Linux and distribution system, Section 3.1.
3. The principle of interaction between researcher and subjects	Requires critical reflection on how the research materials were socially constructed through the interactions between the researchers and participants.	Discussion of interviews during participant observation, Section 3.2.
4. The principle of abstraction and generalization	Intrinsic to interpretive research is the attempt to relate the particulars described in the unique instances observed to abstract categories and concepts that apply to multiple situations.	Presentation of ambiguity, negotiation, and infrastructure as theoretical constructs, Section 1. Relating the analysis to these constructs, Section 4.
5. The principle of dialogical reasoning	Requires sensitivity to possible contradictions between the theoretical preconceptions guiding the research and the actual findings.	Discussion of establishing relevance and context of observations, Section 3.3.
6. The principle of multiple interpretations	This principle requires the researcher to be sensitive to difference in interpretations among the studied subjects.	Central topic throughout analysis and conclusion, Sections 4 and 5. Multiple interpretations the process of negotiation is discussed in Section 4.3.
7. The principle of suspicion	Requires sensitivities to possible biases and systematic distortions in the narratives collected from the participants.	Discussion of no clear principles for resolving bug reports, Section 4.2.

It need not be obvious what the failure "really is". Reporting failures is a balancing between providing too little information and too much information, but sufficient and relevant information [23]. However, it is difficult for a user to determine what sufficient and relevant information is when it is not obvious what the failure really is. Instead, the process of describing the failure is often a series of exchanges where the developers ask the user reporting the failure to generate more information about the failure. These exchanges may span over days, weeks, or even months before the bug report is resolved, and this is what the *Additional comments* field of the bug report is used for.

Martin & McClure [17] argue that programmers doing corrective maintenance "do not know where to look and often waste a great deal of effort looking in the wrong place". The exchanges back and forth between Gentoo users and developers may seem like a process of trial and error like that described by Martin & McClure. However, the view that corrective maintenance is a question of following the infection chain from the observed failure to its fault, presupposes that the observations of the software failure are unambiguous. However, as Endres [8] notes, "[t]here is, of course, the initial question of how we can determine what the error really was". He equates the error with the correction made, noting that this is not always correct but sometimes the bug lies too deep to be grasped or corrected. In SI the most significant problem is that failures are caused by external packages that the Gentoo community cannot control. Typically, this would lead to rejecting the bug report [23], but in Gentoo this problem is so prevalent that the developers have to bypass it.

The software being integrated by Gentoo is developed and maintained by other OSS projects, While some Gentoo developers may be quite familiar and knowledgeable of the source code of the components they integrate, most treat the software being integrated as a black box. It is therefore usually not possible to trace the infection chain of the failure. Instead the Gentoo developers use standard Unix tools and diagnostic tools developed specifically for Gentoo to generate indirect information about the failure. Along with the textual information provided in the bug report, we call the output of these tools *debug texts*. It is often impossible to establish what the failure "really is" from these indirect observations. However, during this exchange between users and developers, the users iteratively provide developers with additional debug texts in an attempt to reconcile the data. During this process multiple interpretations of what the failure "really is" are constructed from combining elements from the different debug texts. "Ambiguity means that a group of informed people are likely to hold multiple interpretations or that several plausible interpretations can be made without more data or rigorous analysis making it possible to assess them" [1]. As such, these failures can be considered ambiguous because what the failure "really is" cannot be established given sufficient information. Instead, this information gives rise to several plausible interpretations of the failures.

With ambiguity the possibility of clear cause-effect relationships and exercised qualified judgment becomes seriously reduced. Cast another way, the understandability of the software becomes seriously reduced. Instead, an understanding of the software failure and its corresponding fault is established through negotiation.

4.2 Negotiation

Gentoo as a software system lies outside the intellectual grasp of a single individual, requiring several organized groups of people to develop and maintain it (see section 3.1). As no single individual can have complete systems comprehension, understanding failures and their corresponding faults becomes a collective activity where individual Gentoo developers' partial comprehension is combined. This is further accentuated by the fact that there is no single Gentoo installation, but thousands of Gentoo installations where software failures occur. As such, the users' knowledge of local system configuration is an important part of the knowledge required to generate a comprehension of the software failure. An understanding of the failure is therefore reached through an iterative process where the user produces new debug texts and the developers generate interpretations of these texts by negotiating over the meaning of the texts. These negotiations often lead to new requests for debug texts in an iterative cycle until a consensus interpretation of the failure is reached. As such, negotiation is the collective process of sharing existing system comprehension and generating new through the production of debug texts. However, this is also a process of reducing the number of interpretations to reach a closure of the bug report. Through consensus interpretations are made invalid.

During negotiation there is often a wide variety in interpretations of the source of failures. It is often hard to find the source of failures resulting from unpredictable interaction of several packages, and as "deciding upon who is to blame is a political process" [23]. Complex interactions among the packages provided by Gentoo produce similar situations in Gentoo. Such interaction effects can also be observed in the interface between the software distributed by Gentoo and the underlying operating system. Varying

standards of system calls among Unix versions can also increase the complexity of the failure. This is a sort of interaction effect akin to architectural mismatch [5]. Finally, failures may also be caused by specific configurations of the user's system. Common to the above failures is that it is hard to locate the fault. The failures are ambiguous in the sense that they lack clear boundaries.

Negotiation is the approach for overcoming this problem. As such it is very much like the political process described by [23]. If it cannot be resolved technically, the fault is located through consensus. However, there are no clear principles for doing so. For instance, one might assume that failing to reproduce a failure would be an indication that the fault is with the user's local configuration and be grounds for rejecting the bug report. Sometimes irreproducibility means the rejection of a bug report. At other times, irreproducible failures or even failures found to be caused by user configuration are resolved. What we see is that the criteria for resolving or rejecting failures varies from bug report to bug report. This is but one of many examples of a pattern of no clear principles to determine what constitutes a valid failure or for resolving unclear boundaries in failures.

Such a lack of clear principles is another trait of ambiguity, and can be seen as the result of several people with differing priorities and practice doing corrective maintenance. This is a reasonable explanation and can in part explain the lack of clear principles. However, the explanation should not overshadow the interpretation that some of this lack of principles is also a product of the ambiguity of software failures as a result of the complexity and variability of Gentoo. This can explain the uncertainty, complexity, instability of principles, and uniqueness in the way bug reports are handled. The lack of clear principles raises issues of power, but this is outside the scope of this paper.

One might be tempted to see the process of negotiation as a way of reducing or overcoming ambiguity. Yet, at its very heart lies the need for ambiguity. It is not uncommon that developers refuse to assist in helping to resolve bug reports even though the fault can be identified within their area of responsibility. When this happens, ambiguity plays a role in getting the bug report back on track again. If there were no room for interpretation, there would be no way of proceeding with resolving the bug report. However, with multiple interpretations it is possible to pursue another interpretation in order to resolve the bug report.

4.3. Infrastructures

In the above analysis we have moved from the ambiguity generated in the technical domain to the social processes of interpretation and negotiation to cope with and handle this ambiguity. In this section we will once again return to the technical domain, albeit with a definite connection to the social. From the above analysis we see that knowledge and systems comprehension may be understood as a product or an effect of various materials. It occurs in the form of debug texts, in the skills for using the debug tools embodied by the Gentoo users and developers, and in the knowledge about the system and typical failures embedded in the debug tools. Not only is systems understanding the product of these materials along with the tools and people generating them, but through knowledge about the system and frequently occurring failures embedded in the tools the tools themselves participate in generating the possible interpretations. As such, corrective maintenance is made possible by this network, or infrastructure, of tools and people [15].

We find that the Gentoo infrastructure of debug tools consists of two groups of tools. Tools in the first group are standard Unix tools like, for instance, `strace` for tracing system calls and signals or `ldd` for printing shared library dependencies. These are debug tools known to most Unix developers. The other group of tools is the custom tools specifically made for Gentoo. Among these are tools that are distributed as part of Gentoo, tools available from private home pages of developers and super users, and tools available from an unofficial repository for Gentoo tools. Debug tools are also proposed and discussed on the IRC channels, and it is common for people to submit debug tools they have developed as bug reports in the Gentoo defect tracking system.

The infrastructure of debug tools is used for generating debug texts. As such, their role is to generate data and to support the negotiation over possible interpretations of these data. We include the Bugzilla defect tracking system as part of the infrastructure of debug tools, too, since it both supports the communication among developers as well as being used for marking duplicate bug reports. Duplicates often provide valuable information on invariants of a software failure.

While the Gentoo developers are not explicit on the process of developing and maintaining the Gentoo-specific debug tools nor on the importance of this job, in practice they are performing a process where knowledge about typical error situations and typical diagnostic actions are inscribed in tools. As typical

failures change over time, tools are made obsolete and new tools are added either in the official distribution or on the unofficial locations such as home pages and the tools repository. It is quite common to see references to Web pages with tools on the developers' IRC channel. This devising of relevant debug tools and the demise of irrelevant tools is a continuous process contingent upon the current reported failures.

4.4. A proposed maintainability model

We see, then, that developing and maintaining Gentoo involves ambiguity both in product as described in the research setting and in process as described in the results and analysis above. This ambiguity of process and product manifests itself in the corrective maintenance activities. Tracking down the source of failures is a process of generating systems comprehension through the production and interpretation of debug texts. We see from the above analysis that tracking down the bug need not be all that simple in practice. It need not be obvious what the bug "really is". Rather, it is subject to interpretation and negotiation. A number of possible interpretations are discussed, and none are dismissed on conclusive evidence but rather made less plausible. Alternative explanations for what the failure "really is" are constructed from combining elements of the different debug texts. The explanations are made more or less plausible both by producing new debug texts, trying to reproduce the failure, drawing on external texts like installation scripts and change logs, or simply by refusing to enter a discussion over possible interpretations.

What we see then, is that reaching an agreement as to what the failure really is, is made with both ambiguous and inconclusive evidence and is more or less open throughout the process. Finding the source of the problem is a process where the person reporting the failure and those trying to understand it work together to find relevant pieces of information and producing additional debug texts. Making the software maintainable can therefore be interpreted as a collective process including both the person submitting the bug report, those trying to understand and resolve the problem, as well as the tools involved in producing the various debug texts being interpreted. The software is made maintainable by iteratively producing debug texts, extracting fragments of information from these texts and assembling these fragments into meaningful combinations.

With basis in this, we propose a model to describe the corrective maintenance process to support our explanation of maintainability. We present two views of this model. Figure 1 shows the cyclic process of producing new debug texts and generating new interpretations through negotiation. The vertical arrow in the middle of the cycle illustrates the number of interpretations.

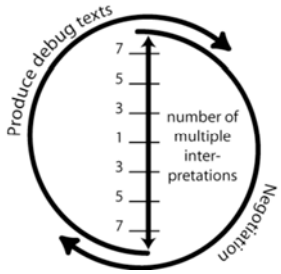


Figure 1. Cyclic view of the corrective maintenance process

Through iterations of the process, the number of interpretations may contract or expand. This is shown in Figure 2. This figure provides a temporal view of the process from the bug report is submitted until it is closed. The number of interpretations is a function of both the level ambiguity and the degree of consensus among developers. Reaching the point of closure can therefore be achieved through the elimination of ambiguity or simply by reaching a consensus about how to resolve the bug report by possibly rejecting it without any technical basis. These are the extremes. More commonly, though, bug reports reach their closure through reducing the ambiguity and reaching a consensus.

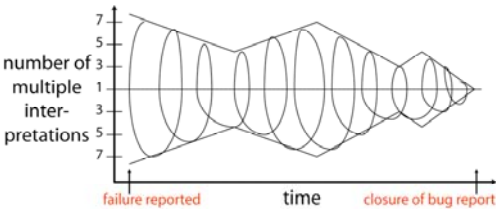


Figure 2. Temporal view of the corrective maintenance process

5. Conclusion

With this basis, we return to our research question: how is maintainability established in systems integration? We find that maintainability is established through the development, operation, and maintenance of a debug infrastructure. This infrastructure mostly supports interaction between developers, like the way

Bugzilla, IRC, and mailing lists are used in Gentoo. The infrastructure must also consist of tools that generate relevant debug information. This is done by constantly evaluating the usefulness of existing debug tools towards the typical failures reported. For Gentoo, we see that this is a continuous process of developing new tools, revising existing tools, and the demise of tools that are no longer useful.

With basis in this we may rephrase our solution to the problem of establishing maintainability in SI. Maintainability in SI may be established through an infrastructure that bridges both the geographical and knowledge gaps between actors in the corrective maintenance process.

6. References

[1] Alvesson, M., *Knowledge Work and Knowledge-Intensive Firms*, Oxford University Press, Oxford, 2004.

[2] Bass, L., P. Clements and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, Boston, Massachusetts, 2003.

[3] Belady, L.A. and M.M. Lehman, "Characteristics of Large Systems", *Research Directions in Software Technology*, P. Wenger (ed.), pp.108-138, MIT Press, Cambridge, Massachusetts, 1978.

[4] Boehm, B.W, J.R. Brown and J.R. Kaspar, *Characteristis of Software Quality*, TRW Series f Software Technology, Amsterdam, North Holland, 1978.

[5] Boehm, B. and C. Abts "COTS Integration: Plug and Pray?" *IEEE Computer,* pp. 135-138, January 1999.

[6] Calzolari, F., P. Tonella and G. Antonioli, "Dynamic model for maintenance and testing effort", *Proceedings of the International Conference on Software Maintenance, ICSM'98*, pp. 104-112, 1998.

[7] Dalal, S.R., J.R. Horgan and J.R. Kettering, "Reliable software and communication: Software quality, reliability, and safety", *Proceedings of the 15th Conference on Software Engineering, ICSE'93*, pp.425-435, 1993.

[8] Endres, A., "An Analysis of Errors and Their Causes in Systems Programs", *Proceedings of the 1975 Conference on Reliable Software*, pp. 327-336, 1975.

[9] Feller, J. and B. Fitzgerald, *Understanding Open Source Software Development*, Addison-Wesley, Boston, Massachusetts, 2002.

[10] Gibson, V.R and J.A. Senn, "Systems Structure and Software Maintenance Performance", *Communications of the ACM*, pp. 347-358, March, 1989.

[11] Hasselbring, W., "Information Systems Integration", *Communications of the ACM*, pp. 32-38, June, 2000.

[12] Fetterman, D.M., *Ethnography*, Newbury Park, CA: Sage Publications, 1998.

[13] Klein, H.K. and M.D. Myers, "A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems", *MIS Quarterly*, pp.67-93, January, 1999.

[14] Lam W. and V. Shankararaman, "An Enterprise Integration Methodology", *IT Professional*, p. 40-48, March/April, 2004.

[15] Law, J., "Notes on the Theory of the Actor Network", 1992, http://www.lancs.ac.uk/fss/sociology/papers/law-notes-on-ant.pdf

[16] Lethbridge, T.C., S.E. Sim and J. Singer, "Studying Software Engineers: Data Collection Techniques for Software Field Studies", *Empirical Software Engineering*, pp.311-341, July, 2005.

[17] Martin, J. and C. McClure, *Software Maintenance: The Problem and Its Solutions*, Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[18] McCall, J. A. "Quality Factors. In Marciniak", *Encyclopedia of Software Engineering, Vol. II*, John J. (Ed.), John Wiley & Sons, New York, pp. 958-969, 1994.

[19] Paulson, J.W., G. Succi and A. Eberlein, "An Empirical Study of Open Source and Closed-Source Software Products", *IEEE Transactions on Software Engineering*, pp. 246-256, 2004.

[20] Samoladas, I., I. Stamelos, L. Angelis and A. Oikonomou, "Open Source Software Development Should Strive for Even Greater Code Maintainability". *Communications of the ACM*, pp. 83-87, 2004.

[21] Scach, S.R., B. Jin, L. Yu, G. Z. Heller and J. Offutt, "Determining the Distribution of Maintenance Categories: Survey versus Measurement", *Empirical Software Enginering*, pp. 351-365, 2003.

[22] Vogels, W., "Web Services are not distributed objects", *IEEE Internet Computing,* pp. 59-66, November/December, 2003.

[23] Zeller, A., *Why Programs Fail: A Guide to Systematic Debugging*, Elsevier, Amsterdam, 2006.

Paper 5

Jaccheri, L., and Østerlie, T. "Open Source Software: A Source of Possibilities for Software Engineering Education and Empirical Software Engineering", in *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development, co-located with the 29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, Minnesota, May 20-26, 2007, pp. 1-5.

Open Source Software: A Source of Possibilities for Software Engineering Education and Empirical Software Engineering

Letizia Jaccheri, Thomas Østerlie
Norwegian University of Science and Technology
letizia@idi.ntnu.no, thomas.osterlie@idi.ntnu.no

Abstract

Open source projects are an interesting source for software engineering education and research. By participating in open source projects students can improve their programming and design capabilities. By reflecting on own participation by means of an established research method and plan, master's students can in addition contribute to increase knowledge concerning research questions. In this work we report on a concrete study in the context of the Netbeans open source project. The research method used is a modification of action research.

1 Introduction

Open source software (OSS) has shaped software engineering education programs over the past decade [9]. OSS development poses serious challenges not only to the commercial software industry, but also to academic institutions that educate software engineers. Motivated by their passion for programming, some of our students love participating in OSS projects; some have even been participating in OSS projects for years.

With increased focus on adoption and use of OSS, the Norwegian industry and public sector is looking for software engineers with OSS skills and knowledge. "Highly qualified personnel are the principal product of universities, and play a major role in developing absorptive capacities in firms" [3]. We therefore see it as our role to develop educational programmes to make software engineering students acquainted with the theoretical aspects of OSS development, as well as with the technical and social skills to participate in OSS projects. This way, we are feeding back OSS v.2 engineers [6].

For students OSS is an arena for learning, and the industry needs software engineers acquainted with the theoretical and practical aspects of OSS development. Our research question is therefore: How can we make use of OSS communities for formal education purposes? How do we, as software engineering teachers and researchers, tackle this challenge so that we provide a sound and motivating environment for software engineering education?

We have developed an approach for teaching master level students by using principles from action research [2] for organizing OSS education. We design assignments for master level students, who have to act as both developers and researchers in OSS projects. As developers they have to participate in a given OSS project. As researcher, the student has to define one or several research questions based on the existing literature on OSS development, to be addressed by participating in an OSS project.

We have been running these kinds of projects since 2002 and all the reports are available online [4]. In this paper, we report from one such project. We will present a case that will provide the choice of the project, the research questions and the answers we found to them. More important, we will summarize lessons learned from combining an education and empirical approach to OSS research. We discuss how the choice of the research questions, the research methods, the literature, and the choice of the OSS project are dependent on each other.

Evaluation of education results is always a challenge, especially when there is no baseline to compare to. It is always difficult for the student to have a double set of goals in mind. The first set of goals are related to participation (understanding the code, understanding the dynamics in the community, finding a way to contribute). The second set of goals are related to observation and research. Some students have never participated in a research project before and they struggle to understand the connection between their research questions and the actual learning process. The research results, which are a secondary product of the student projects, could be of interests to the educational community if we manage to aggregate them in a consolidate framework.

The structure of this paper is as follows: Section 2 provides the foundations of this work at the intersection between software engineering education and empirical software engineering. Section 3 presents our case and Section 4 provides discussion and our conclusions.

First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07)
0-7695-2961-5/07 $20.00 © 2007 IEEE

2 Context

2.1 Empirical software engineering education

We teach a five year Master of Technology program in computer science at IDI[1], NTNU[2]. For the past 30 years, an important part of this graduate program has been to offer students realistic and industry-relevant software engineering projects in close collaboration with the Norwegian software industry [7].

There is an increased use of OSS in Norwegian software companies, and some companies even participate in OSS projects. We therefore see the need for educating students with the knowledge and skills to participate in such projects. This way the industry can give back to OSS communities. To this end we provide project assignment for our master students to participate in an OSS projec in their fifth, and final, year. The main constraint and source of feedback, in addition to teacher supervision, is the interaction with this OSS project. The students may influence what kind of technology to work with. The goals of the project include defining relevant research questions, the study of existing literature on OSS development, selecting an OSS project based on characteristics defined by the student, and to participate actively in this project. Students admitted to these projects must attend two supporting courses with exams.

2.2 Research-based education

At IDI we have a tradition for combining software engineering research and education in various forms. On the one hand we have been designing and running experiments where students act as subjects. On the other hand, our best master students have been given the chance to work as junior researchers in our research projects [1].

In the approach for teaching OSS to master level students we report on in this paper, we combine these two perspectives. Students are subjects of the investigation by particpating in the OSS project under study. Students are researchers, too, as we formulate project assignments so that the student together with the supervisor and other senior researchers contribute to the definition of research questions, data collection and analysis. This can be understood as a form of action research (AR) [2].

AR is a research method that may be well suited for empirical research in the context of education. It originates from the social sciences, and is used for learning from experience by intervening into various systems. One orientation of AR is Canonical Action Research (CAR), proposed by Davison et al. [2].

[1]The Department of Computer and Information Science
[2]The Norwegian University of Science and Technology

CAR is an iterative process consisting of two main components: carefully planned and executed cycles of activities, and a continuous process of problem diagnosis. This has the dual function of improving practice in an organization through a change process, while also contributing to knowledge about the object of the study.

3 The case

In this section we present an example from one student study. This study was based on an assignment designed according to the principles laid out above. The study is available at [4]. The assignment did not constrain the research questions asked by the student. The only constraint was that AR should be used as methodology for the study. In what follows, we first present the canonical description of each phase in the research project, followed by a presentation of how this step was performed in the study we report from.

3.1 Diagnosis

This phase involves the researcher diagnosing the organizational situation from the information that is available. The objectives of the CAR project will control what is studied here, along with experiences from previous CAR iterations. Goals of the diagnosis phase include determining causes of a problem, and study the environment in order to allow proper actions to be planned.

The assignment was for the student to participate in and contribute to OSS development in order to better understand how firms can benefit from using OSS. The goal was to determine the effects of using formal techniques in OSS projects, like explicit planning, ownership, inspection and testing in OSS projects, as they occur in commercially controlled OSS projects. Through the study, it was intended to see if the commercial use of OSS leads to a more manageable process.

The research goal and hypotheses were created from a literature survey that was done in the first two months of the project.

3.1.1 Research questions

Two research questions were formed:

Q1: *Are developers who are not directly hired by the controlling organization able to affect the decision processes?*

The rationale for this research question was to uncover how commercially operated OSS projects view volunteer developers. In case of any confusion of roles, especially with regard to paid vs. non-paid developers, the use of OSS may be less suitable.

0-7695-2961-5/07 $20.00 © 2007 IEEE

Q2: *How much of the decision process is open to the whole community, and to what extent are decisions taken inside the organization that is controlling the open source project?*
For OSS projects where decisions are not multilateral, participants may feel there are conflicts in the community, as described in [8].

3.1.2 Project selection

As the number of potential OSS projects to choose from is large, project selection was initiated to find a project that suited the study well. Project selection and research questions are related, as the studied artifact must be suitable for the research goal. OSS communities have widely different differing characteristics; they vary in size, have different goals, and may have reached different levels of maturity. As the project aimed at investigating commercial ties in OSS development, the selection process aimed to identifying a project where this connection was clear.

After compiling a list of known OSS projects, each project was evaluated according to the following criteria:

- Should consist of 10-50 active developers
- Community allows entrance in a supporting role
- Formal techniques (project planning, etc.) are used in the OSS development
- Implementation is done in either Java or C++, to which the researcher is acquainted.
- Available public mailing lists, chat, and bug tracking.
- Software has general usefulness for researcher

Based on this evaluation, the student chosen the Netbeans project.

3.2 Planning

In CAR, the planning phase should generate a course of action for collecting data. The planned actions should be generated to manipulate the object in order to better understand it.

A data collection strategy was developed during the initial planning phase of the student project. With qualitative data analysis, the goal was to capture as much interaction with other people as possible, thoughts and opinions during the project. The following elements were emphasized in the plan:

1. Identities and roles of people that participate in these discussions.
2. The process which is used for accepting or discussing contributions, and how decisions for inclusion of are made.
3. Communication around changes will be useful for later analysis, to see how decision-making is done.
4. Information about how a contribution fits into schedules and personnel allocation is interesting.

3.3 Intervention

In CAR, intervening in an organization requires that a plan for collecting data is present. Proper data collection techniques should be applied before, during, and after the intervention.

The study was executed with two iterations of AR. The researcher started with little knowledge of the decision processes in the Netbeans project. A meritocratic leadership was assumed to exist in addition to the maintainer organization's influence on the product. Actions that were planned for the first iteration included finding open bugs, making significant changes in order to fix the issue, and following through the inclusion of the change into the main version.

Finding issues that were easy to work with was harder than expected. A total of three bugs were addressed in this phase, but with regard to the number of code lines the contribution was very small.

From the interactions in this process, Netbeans was found not to significantly differ from meritocratic hierarchies in other OSS communities. However, one surprising discovery was that most contributors seemed to be employees of the maintainer organization.

3.4 Evaluation

In CAR, results from the intervention phase should be analyzed in the context of the current understanding of the problem and the goals of the research.

Data analysis was performed according to a pre-defined plan that involved considering the observations in context of theory. Davison et al. state that theory provides a basis for delineating the scope of data collection and analysis [2]. Assessing findings in a broader context also increases confidence in the results.

Tangible results that were found from the project included findings in the Netbeans community that there may be problems in attracting a large volunteer work force. However, the Netbeans project does implement the OSS model well, and values all outside contribution. Further investigation will be needed to see if these tendencies are universal to other OSS projects where commercial organizations are maintainers.

0-7695-2961-5/07 $20.00 © 2007 IEEE

The planned time schedule was found to be unproblematic. However, it would be preferable to have more time for participating in the project. Joining an OSS project, getting familiarized with the project artifacts, while also contributing to it, takes considerable effort.

3.5 Reflection

Reflecting is the last phase of CAR, in which the researcher reflects on the results of an iteration can determine whether additional iterations are necessary, or the lessons learned can be used to further refine research questions. If the goals of the project have been accomplished, then it could be decided to terminate further investigation.

Actions for the second iteration would focus on participating to one module within Netbeans, and looking closer at the artifacts surrounding it. The "JavaCVS" module was selected. Only one bug lead to a successful resolution during this iteration, which incidentally was unrelated to the JavaCVS module. Lessons learned during this supported the notion that few participants outside of Netbeans were active.

After this iteration, the action research cycle was ended, as the time constraints were exhausted, and sufficient information to discuss research questions had been collected.

In retrospect, there are many ways in which participation to OSS for education can be made smoother. First, focusing on OSS as a social discipline can help the researcher to get access to the project artifacts, and contribute to valuable knowledge both about culture and product. By applying action research, the researcher should go to length to collaborate with other people during the project execution, for instance through discussing ideas and technical solutions in mailing lists or newsgroups.

Second, a good recommendation is to focus research on one restricted domain, like a particular module or functional area. While this was not extensively practiced in this case, it is beneficial to commit to one particular role in order to get a more likely "open source situation".

Netbeans is evaluated to be a good choice as it is maintained by a larger software company, Sun Microsystems, that also invests significant resources to sustain it. Netbeans is a development tool that is used to aid in the development of Java-based applications.

Experience from the project, however, show that the project selection criteria may not have been optimal. The following was noted after the completion of the project:

Maturity: Selecting an OSS project that has a low level of maturity may have the disadvantage of being significantly different from an ideally run OSS project. However, if an OSS project is mature, well-tested, and close to a release, much of the remaining tasks will be polish. If the goal of the project is to contribute to an OSS project, the researcher should at least be aware of possible difficulties. In this case with Netbeans, contributing to it was difficult due to the difficulty of understanding complex bug reports.

Size of project: Larger OSS projects may suffer from awareness problems. Entering an open source project consisting of thousands of source files requires either excellent skill, or good documentation.

4 Discussion and conclusions

The final goal of this work is providing guidelines on how to exploit open source software for education and empirical purposes. At the time of writing we can provide several examples of projects that exploit open source software for education and empirical purposes, all available at sumaster.

There are four main axes around which to organize an evaluation of our goal:

1. Research questions: Working on the research questions is a time-consuming task that required a good understanding of the domain. Here there is a trade-off between learning and research issues. While students appreciate the freedom of the assignment as a positive learning experience, it is more effective from a research perspective to provide students with predefined research questions. These can be taken from related literature or from previous research projects the teacher/researcher has been working with. There is a relationship between research questions and projects. For example in the case reported in this paper, the research questions are about the interaction between professionals and volunteers in the OSS projects and this makes it necessary to select a project in which commercial actors play a significant role. From the point of view of the industrial professional who evaluated our work and also from discussions with other researchers, it was found that it is better not let students to choose research questions.

2. Research methods: Action research has worked well to balance student's learning, and the output of the study. A deep understanding of the problem itself is not necessary before intervention in projects.

   The effort necessary to contribute to an open source project should not be underestimated. A common problem for both projects, was that the students started with too ambitious goals, and therefore may have run into some difficulties.

   The different iterations used by the researcher to evaluate the problem may take considerable time and energy. The effect of this, is that learning about open

0-7695-2961-5/07 $20.00 © 2007 IEEE

source development in general, will be a continuous process throughout the entire intervention period.

For the sake of presentation and discussion we have presented our case according to the five phases in action research (diagnosis, planning, intervention, evaluation, and reflection). We are still discussing how the different phases overlap with each other. Take for example the project selection phase which we regard as a sub-phase of diagnosis. In other action research projects, the choice of the projects to work with may happen before the whole research process is started. The same is valid for research questions (or goals) which can be less open to be decided inside the AR cycle than in our case. Evaluation and reflection are two related phases that could be merged together.

3. Literature: The open source literature is cross-disciplinary and this influenced the choice of the research questions which in turn influenced the size and the nature of the project.

   A literature review must be performed to update the content of the supporting course and provide a theoretical background that reflects the evolution of the OSS research field. This is an ambitious task as OSS research efforts have been published in the main software engineering conferences and journals in the last few years. There is also an increasing number of books on this subject that have been published in the last couple of years. This makes the task of maintaining a map of open source literature a challenging one.

4. Choice of the open source project: In communicating with OSS projects, problems in the last project included entry difficulties and problems handling the size of the project. The technical competence needed to contribute was here higher than anticipated. More participation in mailing lists and newsgroups may have been helpful to react to this problem.

Guidelines for using OSS in education should stress that it is a social and complex discipline. Learning both empirical methods and getting an introduction to the open source is difficult. As mentioned, we have an ongoing project that is run according to the same principles in the case reported here and we plan to propose the same kind of projects, both 500 hour and 1000 hour projects in the next academic year.

Concerning the research method, we are satisfied with the use of action research and we believe that this paper is a valuable description of how to use this method. Concerning the research questions, there will always be a phase in which the student and the supervising researcher select new ones starting from consolidated questions in the general literature or provided by this family of projects. The choice of the open source project is an interesting topic of discussion. While it is in the interest of the teacher/research to decide the project in which the student work, we have to keep in mind that one of the principles of OSS participation is motivation and interest. Students choosing such assignments are do so for the love of participating in OSS projects. By letting the student participate in this project, the teacher/researcher could get valuable insights in the specific project.

The perspectives of the industry here is similar to that of the researcher in that industrial actors are naturally interested in letting students work on the OSS projects they support to increase activities in these projects. By replicating these kinds of projects we aim to develop a characterization of both OSS projects and research issues.

References

[1] Jeffrey Carver, Maria Letizia Jaccheri, Sandro Morasca, and Forrest Shull. Issues in using students in empirical studies in software engineering education. In *IEEE METRICS*, pages 239–, 2003.

[2] Robert M. Davison, Maris G. Martinsons, and N. Kock. Principles of Canonical Action Research. *Information System Journal*, 14(1):65–86, 2004.

[3] Giovanni Dosi, Patrick Llerena, and Labini Mauro Sylos. The relationships between science, technologies and their industrial exploitation: An illustration through the myths and realities of the so-called 'european paradox'. *Research Policy*, 35(10):1450–1464, Dec. 2006.

[4] Reidar Conradi et. al. Software engineering group home page. http://www.idi.ntnu.no/grupper/su, January 2007.

[5] Roy T. Fielding. Shared leadership in the apache project. *Communications of the ACM*, 42(4):42–43, April 1999.

[6] Brian Fitzgerald. The transformation of open source software. *MIS Quarterly*, 30(3), 2006.

[7] M. Letizia Jaccheri. Software quality and software process improvement course based on interaction with the local software industry. *Computer Applications in Engineering Education*, 9(4):265–272, 2001.

[8] Chris Jensen and Walt Scacchi. Collaboration, leadership, control, and conflict negotiation in the netbeans.org community. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS 05)*, page 196b, 2005.

[9] Mary Shaw. Software engineering education: a roadmap. In *ICSE - Future of SE Track*, pages 371–380, 2000.

Paper 6

Østerlie, T., and Jaccheri, L. "A Critical Review of Software Engineering Research on Open Source Software Development", in *Proceedings of the The Second AIS SIGSAND European Symposium on Systems Analysis and Design*, Gdansk, Poland, June 5, 2007, pp. 12-20.

A Critical Review of Software Engineering Research on Open Source Software Development

Thomas Østerlie
Norwegian University of Science and Technology
thomas.osterlie@idi.ntnu.no

Letizia Jaccheri
Norwegian University of Science and Technology
letizia.jaccheri@idi.ntnu.no

ABSTRACT

This paper asserts that the software engineering (SE) research literature describes open source software development (OSSD) as a homogenous phenomenon. Through a discourse analysis of the SE research literature on OSSD, it is argued that the view of OSSD as a homogenous phenomenon is not grounded in empirical evidence. Rather, it emerges from key assumptions held within the SE research discipline about its identity and how to do SE research. As such, it is argued that the view of OSSD as a homogenous phenomenon may constitute a systematic bias in the SE research literature. Implications of this are drawn for future SE research to avoid reproducing this bias.

Keywords

Software engineering, open source software development, literature review.

INTRODUCTION

Over much of the past decade, researchers have studied the open source software (OSS) phenomenon. After two annual conferences on open source systems (Damiani et al., 2006, Scotto and Succi, 2005), numerous special issues within multiple research fields (Adam et al., 2003, Clarke, 2006, Feller et al., 2002, Scacchi et al., 2006, von Krogh and von Hippel, 2003), as well as several cross-disciplinary paper collections on OSS (Feller et al., 2005, Koch, 2004), it is fair to say that OSS research is maturing as a multi-disciplinary field defined by its object of study, the OSS phenomenon. Researchers have approached the phenomenon from a diversity of angles; among these motivations of OSS developers (Lakhani and Wolf, 2005), social organization of OSS communities (Crowston and Howison, 2005), OSS business models (Karels, 2003), as well as OSS development (OSSD). OSSD is the topic of this paper.

Software engineering (SE) publications have been a major channel for OSSD research. After working with the SE research literature on OSSD for almost a decade, we have grown increasingly concerned with what we find to be a black and white view of OSSD. This paper therefore starts with the following assertion: *the SE research literature describes OSSD as a homogenous phenomenon*. Such a description of OSSD is problematic. Recent empirical studies show great diversity in the phenomenon. Michlmyer et al. (2005), for instance, observe "how greatly development practices and processes employed differ across [OSSD] projects". Yet, describing OSSD as a homogenous phenomenon loses this diversity. While it is reasonable that early research lacks nuances, a more nuanced view is expected as research matures. However, this paper asserts that this is not the case for SE research on OSSD. The following research question is therefore asked: *under what conditions can the view of OSSD as a homogenous phenomenon be made and maintained over time?*

This paper seeks an answer to this question through a critical literature review of published SE research on OSSD. In particular, it seeks an answer to the question by examining how underlying assumptions about both the field of SE as well as about the object of study, OSSD, enables and constrains how SE researchers can describe OSSD. As such, the methodology of this paper is discourse analysis (Phillips and Hardy, 2002). This is therefore not a study of the OSSD phenomenon itself, but rather how it is described in the SE research literature.

This paper makes three contributions. First, it contributes to SE research on OSSD by arguing the case for a potential systematic bias in existing research: that of treating OSSD as a homogenous phenomenon. Second, it motivates the need for diversifying our approach to studying OSSD. Whereas existing reviews of SE research focus on increased scientific rigour and validation of research (Fenton, 1994, Zelkowitz and Wallace, 1998), there has been little focus on how approaches to SE research and the assumptions espoused by these approaches influence the object of study. To the SE research community at large this paper therefore contributes with a possible approach for evaluating the effect research approaches and assumptions have on the object of study. Third, although limited to a survey of SE research on OSSD, the paper may hopefully inspire similar reflections on the implications of research approaches within other parts of SE research.

The remainder of the paper is structured as follows. First the methods and materials that this review is based on are presented. We then ground the assertion that the SE literature treats OSSD as a homogenous phenomenon in

an analysis of the SE research literature. The research question is revisited in the discussion where we show how the view of OSSD as a homogenous phenomenon emerges from three different assumptions. The paper is concluded by drawing implications of the analysis for SE research on OSSD.

METHODS AND MATERIALS

This literature review is approached with discourse analysis. Discourse analysis is a method for studying individual texts for clues to the nature of a discourse. It is the study of how interrelated texts, the practices of their production, dissemination, and reception – collectively labeled the discourse – brings phenomena into being. The phenomenon studied here is OSSD. Discourse analysis examines how language constructs phenomena, rather than how it reflects and reveals them. As such, it embodies a strong constructivist philosophy, and is not just a method but also a methodology.

Although discourses are inscribed and enacted in individual texts, the discourse itself exists beyond these material manifestations: "discourses are shared and social, emanating out of interactions between social groups and complex societaly [sic] structure in which the discourse is embedded" (Phillips and Hardy, 2002). As such, discourse analysis seeks to understand the context within which the discourse is embedded and emerges from. Discourses are therefore analyzed along three dimensions: texts, discourse, and context.

Discourses have no clear boundaries. "We can never study all aspects of a discourse, and inevitably have to select a subset of texts for manageability" (Phillips and Hardy, 2002). The remainder of this section describes our method for selecting this subset of texts to analyze.

Stage 1: Publication Selection

During the first stage, publications outlet for SE research on OSSD had to be identified. Webster and Watson (2002) presents two approaches for identifying relevant literature to review: 1) search through leading journals within the field, and 2) with basis in known literature go backwards by reviewing citations and forwards using research indexes to look for papers citing the known literature. This review follows the first approach, using the selection of six leading journals identified by Glass et al. (2002).

Stage 2: Selection of Texts

Once the journals had been identified, individual publications on OSSD research were identified. The selected journals were accessed through digital libraries. The digital libraries were used to identify individual papers by searching for publications with the keyword 'open source'. The journals are available through different digital libraries. Table 1 lists the journals reviewed with the provider of the digital library. As the digital libraries are continuously updated with new publications, the date of the search is also provided in the table. There are slight variations in the searchable fields supported by the digital libraries. Although these variations have minor impact on the papers identified at this stage, a list of the searchable fields supported by the digital library has been included for reference in Table 1.

Stage 3: Refining the Paper Selection

Searching for the keyword 'open source' in the above digital libraries returned a total of 120 papers. At this stage the subset of papers identified by the digital libraries were manually refined. As some of the digital libraries do not support searching for phrases, some of the returned papers were not on OSSD. Rather, they had been returned

Journal	Date of search	Digital library	Searchable field(s)
Information Software and Technology	February 21 2007	Science Direct (www.sciencedirect.com)	Title, abstract, keywords
Journal of Systems and Software	January 30 2007		
Software Practice and Experience	January 30 2007	Wiley InterScience (www.interscience.wiley.com)	Full text, abstract, article title, author, author affiliation, keywords, references
IEEE Software	January 30 2007	IEEE Xplore (ieeexplore.ieee.org)	Full text, document title, author, abstract
IEEE Transactions on Software Engineering	January 30 2007		
ACM Transactions on Software Engineering and Methodology	January 30 2007	The ACM Digital Library (portal.acm.org)	Title, abstract, author, full text (where available)

Table 1 Journals with corresponding digital libraries

as both the word 'open' and 'source' was found in the searchable fields. To remove such papers from the subset of texts to analyze, the papers were searched for the phrase 'open source'. Papers without this phrase were removed from the subset of texts to analyze.

A number of the papers identified were either a) reports on design research where the product has been released as OSS, b) research where OSS is used as a data set to validate non-OSSD methods or techniques, or c) opinion pieces. As these are not studies of OSSD, they were also removed from the subset of texts to analyze.

52 papers were left after two rounds of refining the subset of texts. This is summarized in Table 2.

Journal	Total papers	Not studies of OSSD	OSSD papers analyzed
Information Software and Technology	7	6	1
Journal of Systems and Software	13	8	5
Software Practice and Experience	15	14	1
IEEE Software	62	23	39
IEEE Transactions on Software Engineering	8	7	1
ACM Transactions on Software Engineering	15	10	5
Total	120	68	52

Table 2 Papers selected

Writing Up the Discourse Analysis

Two interests had to be balanced in writing up this review. With the reader and evaluator in mind, it is important to be as concrete as possible in building a credible case for the assertion that the SE research literature describes OSSD as a homogenous phenomenon. In practice this means making direct references to individual texts. However, it is counter to the goal of the analysis to point out problems, faults, or shortcomings of individual research texts. It is not the goal of the analysis to single out individual researchers and attack their research. Furthermore, discourse analysis is concerned with individual texts only in the way they provide clues to the nature of the discourse.

To balance these two interests, only texts that are often cited by other research and can therefore be considered formative to OSSD research are quoted in the analysis below. The danger of such an approach is that the analysis may seem anecdotal and poorly grounded. Yet, the purpose of discourse analysis is not to bring evidence or establish truths by bringing forth deep or hidden structures in a body of texts. Rather, the analysis in this paper is one of many ways of reading the body of SE research texts on OSSD. As such, the analysis provides a particular lens to view the texts with. The best validation of the analysis is therefore for the reader to approach the same body of literature with the provided lens to determine whether or not the discourse analysis provides a fruitful way of understanding the literature.

ANALYSIS

The purpose of this section is to illustrate in what ways OSSD is described as a homogenous phenomenon in the SE research literature. Four ways are identified: 1) statements about the OSSD model, 2) statements that OSSD is different from SE, 3) studies critically addressing early claims that OSSD produces superior software, and 4) studies of OSS adoption in commercial software development. Each of these approaches is discussed in turn.

Statements About the OSSD Model

Raymond's (1998) seminal paper on describes two different approaches to developing software: the organized cathedral and the buzzing activity of the self-organizing bazaar. The bazaar model of software development has a number of distinguishing characteristics: openness, self-organizing, creative, rapid cycle of releases with frequent incremental updates of the source code (Raymond, 1998). With the advent of the Open Source Initiative (Perens, 1999), the bazaar model of software development is renamed the open source software development model. Espoused in this early period of advocacy literature is the view of OSSD a specific approach to developing software.

Statements about such a specific approach to developing software appears in different forms in the SE literature. Some authors talk of the OSSD model, others about the OSSD cycle, while others talk about the OSS paradigm of software development. While it is sometimes noted that there is variation in this specific approach to developing software, the "basic tenets of OSS development are clear enough, although the details can certainly be difficult to pin down precisely" (Mockus et al., 2002). It is therefore possible to talk about a generic OSSD model (Feller and Fitzgerald, 2002). As such, statements about a specific OSSD model in the SE research literature reproduce the advocacy literature's view of OSSD as a homogenous phenomenon.

A variation of this is to make statements about salient characteristics of OSS or OSSD. SE research paper

frequently describe OSSD as geographically distributed software development, that work is not assigned but undertaken, that there are no plans, that OSS is developed by communities of volunteers, or of there being a particular social organization to OSSD. Statements about salient characteristics with OSSD are made with general significance. They apply to all instances of OSSD, assuming that OSSD is a specific approach to developing software. Such statements about salient characteristics with OSSD espouse the view of OSSD as a homogenous phenomenon.

Making such statements about OSSD as a specific approach to developing software serves two functions in the research literature: to generalize bottom-up and top-down.

By generalizing from the *bottom-up*, single instance of OSSD are made to stand in and represent the larger phenomenon of OSSD. This form of overgeneralization within the OSS research is also observed by Crowston & Howison (2005): "most research on FLOSS [Free/Libre, Open Source Software] has been case studies of particular projects, [and] has so far allowed the perception that there is a distinctive FLOSS organizational pattern and set of practices to go largely unquestioned". To generalize from a single instance of OSSD to the larger phenomenon requires homogeneity of the phenomenon, that all instances of OSSD are comparable.

Top-down generalization is mainly used to motivate research on OSSD. A typical top-down generalization can be formulated as "Our interest in studying this particular instance of OSSD originated in the popularity gained by the open source model in the last few years through the delivery of successful products such as Linux, Apache, and Mozilla". The effect of top-down generalization is to motivate research on a single instance of OSSD by grounding it in the larger phenomenon. By mobilizing well-known successful instances of OSSD, it is assumed that all instances of OSSD are worth studying. Again, this form of generalization assumes homogeneity of the phenomenon; that any instance of OSSD can stand in for the larger phenomenon.

Although bottom-up generalization is most prevalent in early research SE literature on OSSD, the most recent observation is found in a research publication from 2006. Top-down generalizations, however, are in one form or another more prevalent throughout the period of the reviewed literature.

Statements that OSSD is Different From SE

Describing OSSD as different from other forms of software development has been a common theme since the early advocacy literature. To begin with it was the cathedral versus the bazaar (Raymond, 1998), it was hacking as opposed to the mechanical forms of commercial software development (Hannemyr, 1999), and later that OSSD is "different from proprietary, or traditional, or commercial or whatever other forms of software development it is that exist besides [it]" (Crowston and Howison, 2005).

Similar statements about dichotomous relations between OSSD and other forms of software development are reproduced in the SE research literature on OSSD. These statements are made in three ways. The first two ways are direct ways of stating the dichotomous relationship between OSSD and SE. First, as direct statements that OSSD is different from SE. SE is not always referenced directly, but referenced as " the usual industrial style of software development" or "usual methods applied in commercial software development". The implication is clear, however, that OSSD is different from SE.

The second way of placing OSSD in a dichotomous relationship with SE is similar to the above approach, but instead of saying that OSSD is different from SE, authors say that OSSD is not an engineering method. The implied comparison is still OSSD versus SE. All such statements are based in a basic black and white schema: that of OSSD on the one hand and SE on the other.

The third way of making statements that OSSD is different from SE, is indirect. It is indirect in that it makes no reference to SE, "the usual style industrial style of software development", or variations thereof. Instead, the comparison is implied by describing OSSD in terms of work not being assigned, no explicit system-level design, and no project plan, schedule or list of deliverables. OSSD is here characterized by reversing salient characteristics of SE: that in SE work is assigned, there is explicit system-level design, and there is a project plan, schedule or list of deliverables. As such, OSSD is placed in a dichotomous relationship with SE reproducing the two broad categories of OSSD on one hand and SE on the other.

By situating OSSD in a dichotomous relationship with SE implies homogeneity of OSSD; that it is meaningful to situate the phenomenon at large in contrast to SE.

Myth-Busting Studies

Early OSS advocacy literature makes claims about the superiority of OSSD compared to commercial software development. In an effort to develop a deeper and more refined understanding of the OSSD phenomenon, researchers have put these myths about OSSD to the test by comparing OSS with close source software (CSS). These studies aim at providing a more correct understanding of the OSSD phenomenon by challenging empirically unsubstantiated claims. Among these are claims that OSSD compared to CSS produces more maintainable software, simpler designs, software with lower defect density, software with higher quality and reliability, and that OSSD fosters more creativity.

There are two common denominators of these studies. One, the research approach is to generate quantitative measures from products of the software process,

particularly source code and defect reports. Two, these studies compare OSS with CSS either explicitly in the research questions or in discussing the findings.

The earliest of these myth-busting studies date back to 2002, with a predominance of such research published from 2004 and onwards. While most of the tested myths are debunked, the studies' significance in the context of this paper is that they build upon the basic dichotomy of OSS in contrast to CSS. In the process of refining our knowledge of OSSD, these studies reproduce a black and white view of OSSD as a homogenous phenomenon by performing comparisons with the two generic categories of OSS and CSS.

OSS Adoption in Commercial Software Development

A number of studies on OSSD adoption in commercial software development have been published recent years. These studies focus on the adoption of OSS components or OSS tools in commercial software development. Numerous researchers have pointed out the tight relationship of OSSD and commercial software companies (Koru and Tian, 2005). However, the relationship between OSS and commercial actors remains largely unexplored. The studies on OSS adoption therefore aim to broaden our understanding of the OSSD phenomenon by investigating this relationship.

The problem with this literature is two-fold. One, it assumes that OSS is essentially different from commercial off-the-shelf software and therefore requires a unique approach for evaluation. Two, although studying OSS in a commercial setting, these studies do not challenge the view of OSSD as completely different from SE. Instead, they focus on how commercial companies make use of OSSD products. Little, if any, attention is paid to the development of OSS in a commercial context. By omission these studies therefore reproduce the view of OSSD as completely different from commercial software development or SE; a view grounded in the assumption of OSSD as a homogenous phenomenon.

DISCUSSION

The analysis above illustrates the ways in which OSSD is described as a homogenous phenomenon by the SE research literature. The purpose of this section is to address the research question by discussing the conditions under which the statement that OSSD is a homogenous phenomenon can be made and sustained in the context of SE research. As such, this part of the paper broadens the analysis from the discourse itself to its context: SE research.

Assumptions About Software Engineering Research

Glass (2003) observes that "[f]or most of SE's history, authors have eagerly told practitioners what they ought to be doing … [b]ut rarely have those 'ought' been predicated on what practitioners actually are doing". Singer et al. (1997) observe that there is little in the SE research literature about what it is that the software engineers do on a day-to-day basis, the kinds of activities they perform, and the frequency with which these activities take place. While there exist a strain of empirical studies of SE in practice, this has had little or no impact on the mainstream SE research literature. It is therefore unproblematic to state that OSSD is different from SE: OSSD practice does differ from prescriptive models for software development.

SE is a movement of industry and academic actors to professionalize software development by applying engineering to software through the "application of systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" (IEEE, 1990). The idea of a software crisis is central to this movement. Practically every SE textbook discusses the software crisis, and both SE professionals and researchers keep discussing the continued software crisis (Glass, 2003). Professionalizing software development is the SE movement's answer to the crisis – to a certain extent even its reason to be. The success of OSSD – software developed by volunteers – can be seen as a direct challenge to the very identity of SE, defying the central claim that professionalizing software development will resolve the software crisis.

That the SE research literature maintains the claim that OSSD is different from SE can be interpreted as a way of meeting this challenge. Refuting the general applicability of OSSD outside the specific context where there is a convergence between user and developer can be interpreted as a direct answer to the challenge (Messerschmidt, 2004). Another approach is to characterize OSSD as the inverse of SE as illustrated in the above analysis. Similarly, in comparing OSSD practice with predictive software development models, publishing SE researchers bypassing the problematic issue that the SE research discipline actually knows little about the field they are trying to address: SE in practice.

As such, maintaining the claim that OSSD is different from SE and CSS development serves the purpose of strengthening the SE research discipline. Yet, the effect of this is that OSSD is treated as if it was a single, homogenous phenomenon. And the question remains: how different is OSSD and SE practice?

Assumptions About How To Do Software Engineering Research

The predominance of empirical studies of OSSD reviewed for this paper, are based on either source code measurements or measures extracted from defect tracking and revision control systems. Of the empirical studies reviewed, only three were not based on measurements of products of OSSD. One of these was an ethnographic study (Scacchi, 2004), one a questionnaire survey (Ajila and Wu, 2007), and the third based on undisclosed observational research (Breuer and Valls, 2006). The

dominant approach for SE research on OSSD is therefore to measure the products of the software process.

This approach to studying OSSD grows out of a problem particular to the situation of SE during the 1990s: researchers' observation of a widening gap between software engineering research and practice (Glass, 1994). The software engineering research community was becoming increasingly concerned with its lack of impact on practice. Researchers looked for ways to address this. Tichy et al. (1993) concluded that instead of informing practice, SE research was lacking in quality and thereby becoming less credible for industry. Similarly, in a review of the SE research literature, Fenton (1993) found "very little empirical evidence to support the hypothesis that technological fixes, such as the introduction of specific methods, tools, and techniques, can radically improve the way we develop software systems".

The diagnosis of the problem situation is outlined in a number of surveys of the SE research literature. In a survey of 612 SE research papers, Zelkowitz and Wallace (1998) found that 58.7% of the surveyed papers had no validation of research claims or the validation was based on assertions. Similarly, in a survey of 400 research papers within the broader field of computer science, Tichy et al. (1995) found only 20% of the SE papers devoted more than one fifth or more of the space to research validation. Glass (1994) labels research lacking in validation advocacy research – researchers advocating a new technology without validating its effectiveness over existing technologies or its applicability to practitioners.

A call for increased empirical research and scientific rigour within the software engineering research community rigour rose in response to the problem situation. To bridge the gap between theory and practice, researchers had to move from a research-and-transfer model to an industry-as-laboratory approach (Potts, 1993). Software engineering research needed to better validate its scientific claims (Zelkowitz and Wallace, 1998). The low ratio of validated research had to be rectified for the long-term health of the field (Tichy et al., 1995). However, validation was only one aspect of this increased concern with scientific rigour. Scientific rigour also require better understanding of measurement theory. Fenton (1994) argues that software engineering researchers "must adhere to the science of measurement if it is to gain widespread acceptance and validity".

Quantitative data based on measuring products of the software development process (i.e. source code and data extracted from defect tracking and revision control systems) are well suited for doing comparative research. The myth busting studies make use of this, by comparing OSS and closed source software (CSS) to verify claims made by early OSS advocates that characteristics of OSS differ from CSS. The myth busting studies can be understood as an amalgamation of the OSS and SE discourses in that the scientific approach of empirical SE is applied on open issues raised by the OSS advocacy literature. While the advantage with measurement-based research is the ability to compare, the problem in this case is that the basis of the comparison is the product of OSSD on one side and the product of what is called CSS development on the other. While the studies have been performed with the highest scientific rigour, the amalgamation between the OSS and SE discourses reproduces the very broad distinction of OSS and CSS.

Operating with only two broad categories absolves the researcher from discussing the comparability of the categories. The question is how comparable measures based on products of the software development process are. How comparable is the defect density of a single-user application developed by two OSS developers, the mean number of developers on the SourceForge.org OSS portal, with that of a large multi-team development effort like the Linux kernel, for instance? This is a problem that cannot be met only by "greater discipline and rigour – deeper research, more quantitative data, and more robust cross case analysis" (Feller et al., 2006). The problem itself a product of the research methods employed on OSSD. As such, it beckons a call for increased multiplicity of research approaches.

Assumptions About the Object of Study

Table 3 summarizes the instances of OSSD studied empirically in the analyzed subset of texts. It is striking how a handful of instances of OSSD keep recurring.

OSSD case	Number of studies
Mozilla	4
Linux kernel	4
Other (unspecified)	3
Apache	2
FreeBSD	2
SourceForce.net	1
OpenBSD	1
NetBSD	1
Debian	1
FreshMeat.net	1
KOffice	1
The GNU Compiler Collection	1
OpenOffice	1

Table 3 Summary of OSS cases studied

Have early descriptions of OSSD been turned into prescriptions for choosing instances of OSSD to study? Is that why there are so few cases? Of the cases studied, all comply with the description of OSS projects as mainly volunteer, adhering to the rapid release and fix software development cycle. There are no empirical studies of OSSD in an industrial setting. Studies on OSS adoption are disregarded, as they are not studies of OSSD in an industrial setting, but rather how OSS is used in a commercial setting.

Fitzgerald (2006) raises concerns about the possibility of a broadening gap between the focus of OSSD research and the OSSD phenomenon itself as OSSD is shifting from geographically distributed software development in communities of volunteers towards development by commercial actors. To meet the concern, Fitzgerald (ibid.) proposes that "the open source phenomenon has undergone a significant transformation from its free software origins to a more mainstream, commercially viable form – OSS 2.0".

Is this altogether new? Perens (1999) reports that the Open Source Initiative, and the OSS term itself, originated in a meeting between advocates and the fledgling Linux industry in 1997. The goal of the meeting was to make free software a viable alternative for the mainstream software industry by de-politicizing it. Commercial interests were always strong in the Apache community (Behlendorf, 1999), even prior to IBM deciding to adopt Apache as its official Web server and hiring many of the Apache developers in 1998. Cygnus Solutions is an early commercial actor building upon and driving development of the GNU Compiler Collection (Tieman, 1999). Similarly, RedHat Software, Inc. has developed and maintained OSS for their GNU/Linux distribution since 1995 (Young and Rohm, 1999). In an effort to meet the stiff competition from Microsoft, Netscape released the source code of their web browser as the Mozilla OSS browser in 1998 to differentiate themselves from the competition.

While all empirical studies of Mozilla reviewed for this paper do note the commercial heritage of the source code and that Netscape hires most of the core developers of the Mozilla project, none have studied the relationship between the company and the community. The Mozilla studies are good examples of how the gap between OSSD research and the OSSD phenomenon that Fitzgerald (2006) is concerned about has already developed within SE research on OSSD. Although recent research suggests that commercial interest in OSS is increasing (Ghosh, 2007), this can hardly be argued as a shift in the phenomenon itself. Rather researchers' focus on community-based OSSD has overshadowed the commercial ties, which were never been truly explored. As such, the premise of Fitzgerald's (2006) problem can be understood as a product of existing research's focus on OSSD as geographically distributed, community-based software development.

IMPLICATIONS FOR SE RESEARCH ON OSSD

Through a discourse analysis of the SE research literature, this paper has argued that the assertion that SE research describes OSSD as a homogenous phenomenon is not grounded in empirical research. The research question 'under what conditions can the view of OSSD as a homogenous phenomenon be made and maintained over time?' is answered by situating the OSSD discourse in context of SE research at large. It is argued that the conditions are to be found in assumptions about the SE research field, how to do SE research, and about the phenomenon of OSSD itself. As such, treating OSSD as a homogenous may be a potential bias running throughout the SE research literature on OSSD.

However, this is not a black-and-white picture. Some researchers raise issues about diversity of OSSD practices. However, the full impact of such observations has yet to materialize in SE research on OSSD. This section concludes the paper by drawing implications of this for SE research on OSSD.

Usefulness of the OSS Term

As shown in the analysis, SE researchers often use the term OSSD to make generic statements about a particular approach to software development. However, this is problematic and does to a certain extent assume that OSSD is a homogenous phenomenon. Gacek and Arief (2004) notes that the only common characteristic of OSSD is that software product is released under an license compliant with the Open Source Definition. As such, the usefulness of the term OSSD is limited and espouses a certain view of the phenomenon.

Researchers may avoid this problem by being specific about the instances of OSSD studied instead of relying upon generic descriptions of OSSD. Being specific on the salient characteristics of the studied instances is a basis for discussions on the generalization of research findings. Here are some issues worth focusing on when being more specific about the studied instance of OSSD.

Sizes. How many developers are involved? What kind of software is developed, and how what is its size?

Commercial and/or community. Some OSS projects are completely community driven, other are controlled by companies, and other in turn are community-based with strong commercial ties. Issues worth considering are therefore: Is the case studied community driven or headed by a company? How many of the community members are hired to contribute, and how many are volunteers? What is the distribution of volunteers and hired developers?

Geographical distribution. One of the issues motivating OSSD has been that of studying successful examples of distributed software development. Many cases of OSSD are geographically distributed. Issues worth discussing when writing up research are: What is the geographical

distribution of the developers? Are any groups of developers geographically co-located? How many groups of co-located developers exist? Does the geographical co-location have any impact on the organization of the project? What is the impact of the geographical distribution on coordination within the project? What tools are used for bridging the geographical gap between developers?

Developer demography. While there exist much research on the motivation of OSS developers, we know little about who they are. Apart from Dempsey et al.'s (1999) study of the distribution of contributors to the UNC MetaLab's Linux Archives by studying the domain of their e-mail addresses, there is a distinct lack of research about who OSS developers are. Future research could focus on improving our understanding of who the people developing OSS are.

Implications for Method

We have illustrated how the dominant approach for studying OSSD within SE reproduces the view of OSSD as a homogenous phenomenon. Leading OSSD researchers call for "greater discipline and rigour – deeper research, more quantitative data, and more robust cross case analysis" (Feller et al., 2006). However, the problem is not caused by a lack of methodical discipline or rigour, but rather with the taken-for-grantedness of the phenomenon studied. As such, more cross case analysis may indeed worsen the problem.

Instead, there is a need for diversifying approaches to studying OSSD. The phenomenon needs to be approached with methods that can shed further light on the practice of OSSD, not only on the products of the process. It may be worth looking towards recent studies of OSSD practice within the field of computer supported cooperative work (Ducheneaut, 2005). This research uses ethnographic methods. While studying the product of OSSD may give the impression of homogeneity of the phenomenon, studies of OSSD practice can challenge this by looking at the specifics of practice may reveal if such is really the case.

Implications for Case Selection

There is a poverty of OSSD cases studied, both in the distribution of individual cases but also in that they are all studies of community-based OSSD. There is no research on OSSD in an industry setting. Little attention is paid to the relationship between commercial organizations and OSS communities. How do commercial actors participation in OSS communities impacts on their internal development processes? Future research should address this by studying such instances of OSSD.

Furthermore, an implication of the problem with using top-down generalization for case selection is that the rationale for case selection has to be grounded in salient characteristics of the selected case. Case selection needs to address two questions: What are the salient characteristics of this case that makes it worth researching? What dimensions of the OSSD phenomenon can it shed further light on?

REFERENCES

1. Adam, F., Feller, J. and Fitzgerald, B. (2003) Logicels Libres: Implications pour les Organisations, *Systems d'Information et Management,* 8, 1.
2. Ajila, S. A. and Wu, D. (2007) Empirical Study of the Effects of Open Source Adoption on Software Development Economics, *Journal of Systems and Software,* Forthcoming.
3. Behlendorf, B. (1999) Open Source as a Business Strategy In *Open Sources: Voices from the Open Source Revolution*, (Eds, DiBona, C., Ockman, S. and Stone, M.) O'Reilly & Associates, Sebastapol, CA, 149-170.
4. Breuer, P. T. and Valls, M. G. (2006) Raiding the Noosphere: The Open Development of Networked RAID Support in the Linux Kernel, *Software-Practice and Experience,* 36, 4**,** 365-395.
5. Clarke, D. (2006) Foreword: Special Issue on Free, Libre, and Open Source Software, *Knowledge, Technology & Policy,* 18, 4**,** 3-4.
6. Crowston, K. and Howison, J. (2005) The social structure of free and open source software development, *First Monday,* 10, 2.
7. Damiani, E., Fitzgerald, B., Scacchi, W., Scotto, M. and Succi, G. (2006) In *Second International Conference on Open Source Systems*Springer, Como, Italy, pp. 351.
8. Ducheneaut, N. (2005) Socialization in an Open Source Software Community: A Socio-Technical Analysis, *Computer Supported Cooperative Work (CSCW),* 14, 4**,** 323-368.
9. Feller, J., Finnegan, P., Hayes, J. and Lundell, B. (2006) *Panel: Business models for open source software, Towards an understanding of the concept and its implications to practice*, http://oss2006.dti.unimi.it/slides/businessModelPanel.pdf, Last accessed: January 4 2006.
10. Feller, J. and Fitzgerald, B. (2002) *Understanding Open Source Software Development,* Addison-Wesley, London.
11. Feller, J., Fitzgerald, B., Hissam, S. A. and Lakhani, K. R. (Eds.) (2005) *Perspectives on Free and Open Source Software,* The MIT Press, Cambridge, Mass.
12. Feller, J., Fitzgerald, B. and van der Hoek, A. (2002) Editorial: Open Source Software Engineering, *IEE Proceedings - Software,* 149, 1**,** 1-2.
13. Fenton, N. (1993) How Effective Are Software Engineering Methods?, *Journal of Systems and Software,* 22, 2**,** 141-146.

14. Fenton, N. (1994) Software Measurement: A Necessary Scientific Basis, *IEEE Transactions on Software Engineering,* 20, 3**,** 199-206.
15. Fitzgerald, B. (2006) The Transformation of Open Source Software, *MIS Quarterly,* 30, 3**,** 587-598.
16. Gacek, C. and Arief, B. (2004) The Many Meanings of Open Source, *IEEE Software,* 21, 1**,** 34-40.
17. Ghosh, R. A. (2007) *Study on the: Economic Impact of Open Source Software Software on Innovation and Competiveness of the Information and Communication Technologies (ICT) Sector in the EU (Final Report)*, ENTR/04/112.
18. Glass, R. L. (1994) The Software-Research Crisis, *IEEE Software,* 11, 6**,** 42-47.
19. Glass, R. L. (2003) The State of the Practice of Software Engineering, *IEEE Software,* 20, 6**,** 20-21.
20. Glass, R. L., Vessey, I. and Ramesh, V. (2002) Research in software engineering: an analysis of the literature, *Information and Software Technology,* 44, 8**,** 491-506.
21. Hannemyr, G. (1999) Technology and Pleasure: Considering Hacking Constructive, *First Monday,* 4, 2.
22. IEEE (1990) *IEEE standard glossary of software engineering terminology*, 610.12-1990.
23. Karels, M. J. (2003) Commercializing Open Source Software, *Queue,* 1, 5**,** 46-55.
24. Koch, S. (Ed.) (2004) *Free/Open Source Software Development,* Idea Group.
25. Koru, A. G. and Tian, J. J. (2005) Comparing High-Change Modules and Modules with the Highest Measurement Values in Two Large-Scale Open-Source Products, *IEEE Transactions on Software Engineering,* 31, 8**,** 625-642.
26. Lakhani, K. R. and Wolf, R. G. (2005) Why Hackers Do What They Do: Understanding Motivations and Effort in Free/Open Source Software Projects In *Perspectives on Free and Open Source Software*, (Eds, Feller, J., Fitzgerald, B., Hissam, S. A. and Lakhani, K. R.) The MIT Press, Cambridge, Mass., 3-23.
27. Messerschmidt, D. G. (2004) Back to the user [open source], *IEEE Software,* 21, 1**,** 89-91.
28. Michlmyer, M., Hunt, F. and Probert, D. (2005) In *First International Conference on Open Source Systesm*Genova, Italy, pp. 24-28.
29. Mockus, A., Fielding, R. T. and Herbsleb, J. D. (2002) Two Case Studies of Open Source Software Development: Apache and Mozilla, *ACM Transactions on Software Engineering and Methodology,* 11, 3**,** 309-346.
30. Perens, B. (1999) The Open Source Definition In *Open Sources: Voices from the Open Source Revolution*, (Eds, DiBona, C., Ockman, S. and Stone, M.) O'Reilly & Associates, Sebastapol, CA, 171-180.
31. Phillips, N. and Hardy, C. (2002) *Discourse Analysis: Investigating Processes of Social Construction,* Sage, Thousand Oaks, CA.
32. Potts, C. (1993) Software-Engineering Research Revisited, *IEEE Software,* 10, 5**,** 19-28.
33. Raymond, E. S. (1998) The Cathedral and the Bazaar, *First Monday,* 3, 3.
34. Scacchi, W. (2004) Free and open source software practices in the gaming industry, *IEEE Software,* 21, 1**,** 68-72.
35. Scacchi, W., Feller, J., Fitzgerald, B., Hissam, S. A. and Lakhani, K. R. (2006) Guest Editorial: Understanding Free/Open Source Software Development Processes, *Software Process: Improvement and Practice,* 11, 2**,** 95-105.
36. Scotto, M. and Succi, G. (2005) In *First International Conference on Open Source Systems*Springer, Genova, Italy.
37. Singer, J., Lethbridge, T. C., Vinson, N. and Anquetil, N. (1997) An Examination of Software Engineering Work Practices, *Center for Advanced Studies Conference (CANCON)*,
38. Tichy, W. F., Habermann, N. and Prechelt, L. (1993) Summary of the Dagstuhl workshop on future directions in software engineering: February 17–21, 1992, Schloß Dagstuhl, *ACM SIGSOFT Software Engineering Notes,* 18, 1**,** 35-48.
39. Tichy, W. F., Lukowicz, P., Prechelt, L. and Heinz, E. A. (1995) Experimental Evaluation of Computer Science: A Quantitative Study, *Journal of Systems and Software,* 28, 1**,** 9-18.
40. Tieman, M. (1999) Future of Cygnus Solutions: An Entrepreneur's Account In *Open Sources: Voices from the Open Source Revolution*, (Eds, DiBona, C., Ockman, S. and Stone, M.) O'Reilly & Associates, Sebastapol, CA, 71-90.
41. von Krogh, G. and von Hippel, E. (2003) Open Source Software: Introduction to a Special Issue of Research Policy, *Research Policy,* 32, 7**,** 1149-1157.
42. Webster, J. and Watson, R. T. (2002) Analyzing the Past to Prepare for the Future: Writing a Literature Review, *MIS Quarterly,* 26, 2**,** xii-xxiii.
43. Young, R. and Rohm, W. G. (1999) *Under the Radar: How Red Hat Changed the Software Business - and Took Microsoft by Surprise,* Corriolis Group, Scottsdale, AZ.
44. Zelkowitz, M. V. and Wallace, D. R. (1998) Experimental Models for Validating Technology, *IEEE Computer,* 31, 5**,** 23-31.

Paper 7

Østerlie, T., and Jaccheri, L. "Balancing Technological and Community Interest: The Case of Changing a Large Open Source Software System", in *Proceedings of the 30th Information Systems Research Seminar in Scandinavia (IRIS 30)*, Tampere, Finland, August 11-14, 2007, pp. 66-80.

Balancing Technological and Community Interest: The Case of Changing a Large Open Source Software System

Thomas Østerlie and Letizia Jaccheri
Norwegian University of Science and Technology

Abstract. This paper studies the process of rewriting and replacing critical parts of a large open source software (OSS) system. Building upon the notions of installed based and transition strategies, we analyze how the interaction between the OSS and the context within which it is developed and used enables and constrains the process of rewriting and replacement. We show how the transition strategy emerges from and continuously changes in response to the way the installed base is cultivated. By demonstrating a mutual relationship between the transition strategy and the installed base, we show how the transition strategy in this particular case changes along three axes: the scope of the rewrite, the sequence to replace existing software, and the actors to be involved in the process. The paper is concluded with some implications for how to study the process of rewriting and replacing OSS.

Keywords. Open source software development. Rewrite and replacement. Transition strategy. Installed base.

Introduction

Parallel development, a rapid release schedule, actively involved users, and prompt feedback are described as key characteristics of open source software (OSS) development (Feller & Fitzgerald 2002). Empirical studies of OSS

development have therefore primarily focused on the cyclic process of corrective and adaptive maintenance (German 2005), its organization (Crowston & Howison 2005), and analysis of the products of this process (Paulson et al. 2004, Samoladas et al. 2004, Mockus et al. 2002). Describing the process of rewriting the FreeBSD kernel, Jørgensen (2001) shows that unlike the discretely delineated tasks of adaptive and corrective maintenance, rewriting OSS is a longitudinal process that does not lend itself well to parallel development, rapid release schedule, and active user involvement. While we know that large and successful OSS products are rewritten–for instance the original Apache code was rewritten and replaced with a modular design in 1995, and several large subsystems of the Linux kernel, like virtual memory handling, have been rewritten and replaced throughout the kernel's life cycle–we find that rewriting and replacing is an underdeveloped topic within OSS research.

Building upon Jørgensen's (2001) work, we study the repeated attempts at rewriting and replacing a core OSS system. The empirical basis for this paper is a study of the Gentoo Linux distribution. The background for the study is that the Gentoo package manager, the core of the Gentoo Linux distribution "is very fragile [because it has] evolved rather than being designed", as one of the Gentoo developers puts it. Studying the attempts at rewriting and replacing the package manager provides an excellent opportunity to study the problems associated with rewriting and replacing critical parts of a large OSS system. To this end, we ask: How does the interaction between the OSS and the context within which it is developed and used enable and constrain the process of rewriting and replacement? In this paper we analyse this by studying the relationship between the installed base and transition strategies (Hanseth and Monteiro 1998) in the process of rewriting and replacing the Gentoo package manager.

The remainder of the paper is structured as follows. The second section motivates the study of rewriting and replacing OSS through the notions of transition strategies and installed base. These two terms are elaborated. The third section outlines the case; presenting the research setting, as well as describing three attempts at rewriting and replacing the package manager. In the fourth section we discuss the case along two dimensions that surface in the case: the issue of resources and transition strategies as a process. The final section contains concluding remarks, where we describe how we have addressed the research question and implications of our findings to the study of rewriting and replacing OSS.

Methodologically, the paper is based on an interpretive case study (Klein & Myers 1999) of the Gentoo OSS community. The data was primarily collected during a ten months programme of participant-observation conducted from March to December 2004. Since the OSS community is geographically distributed, participant-observation took the form of observing and participating on the Internet Relay Channels (IRC) that the community use for communication, by

submitting and resolving failure reports, as well as contributing with code. Throughout the period of fieldwork the IRC channels we participated on were logged to disk; one file each day for each IRC channel totalling 1027 files. A key informant also provided us with his IRC logs, stretching back to April 2003. No formal interviews of participants in the OSS community were undertaken, although informal talks with participants–both on e-mail and on IRC–were conducted on a regular basis to test our informal theories about the fieldwork. 71 documents were collected throughout the period and organized in a documentary database. Online data sources that provide static data were surveyed. These include the Gentoo bug tracking database, the Gentoo mailing list archives, and the Gentoo revision control system. As the Gentoo Web site is under revision control, relevant documents from this Web site were not organized in the documentary database. Instead, we decided to rely on Gentoo's revision control system. This archival material provided us with data from 2002 to the end of 2005. A more thorough presentation of the research is provided in (Østerlie and Wang 2006).

Theory

Jørgesen (2001) describes the process of implementing symmetric multi-processing, a significant new feature, in the FreeBSD operating system kernel. Although the paper describes in detail the practical arrangements for making the significant change and folding it into the main code base, the paper tells little about the context and rationale for organising the process this way. However, the paper provides little information about how the OSS developers decide upon the specifics of this process of going from one version of the software to other. We expand upon Jørgensen's (ibid.) work, by examining how OSS developers make such decisions. We do so by analysing the OSS an information infrastructure (II) (Hanseth and Monteiro 1998), studying the process of rewriting and replacing the Gentoo package manager in terms of transition strategies and installed base.

Transition strategies

The transition strategy is a plan outlining how to go from one stage of the II to the other (Monteiro 1998). However, the transition strategy is caught in a dilemma, "where the pressure for making changes … has to be pragmatically negotiated against the conservative forces of the economical, technical, and organizational investments in the … installed base" (ibid., p. 230). Controversies over a transition strategy are therefore negotiations about *how* big changes can–or have to– be made, *where* to make them, and *when* and in *which* sequence to deploy them.

Whereas Jørgensen (2001) describes the sequencing when rewriting a clearly delineated part of the software, thinking in terms of transition strategies enables us to study the larger process of rewriting software encompassing what is to be rewritten and the scope of the changes, important factors in the process of rewriting the Gentoo package manager.

Installed base

The installed base can be defined as the interconnected technologies and practices that are institutionalised in an organization (Hanseth and Monteiro 1998). Adopting this view, we see that changes cannot be made to software artefacts in isolation, but must always take into account the other elements of the installed base that the artefact is connected to.

This points towards two important elements when thinking in terms of installed base. One, II's must evolve by extending and improving the existing installed base, or *cultivating the installed base* as it is called (ibid.). Two, as II's grow, it becomes increasingly hard to extend and improve it because of the many elements that have to be changed in the process. This is called the *inertia of the installed base* (ibid.).

Actor-network theory

Like II, actor-network theory (ANT) is the underlying ontology for this study as well. We therefore mobilise a limited ANT vocabulary inscribed in and circulated by Callon (1986) and Latour (1987) for the case description and analysis of this paper. Well aware of recent movement toward fluids and fiery objects both within ANT and IS research, we choose to mobilise this vocabulary as it translates well our interest in bringing forth the chronic tension of multiple and at times contradictory interest in cultivating the Gentoo installed base.

A major focus of ANT is to provide a way of tracing and explaining the process of how networks of actors, *actor networks*, become more or less stable through the alignment of interest. Particular to ANT is that the notion of actors encompasses both human and non-human actors such as software technologies, documents, and so on.

The process wherein networks of aligned interest are created and maintained, is called *translation*. Through the process of translation the translating actor defines other actors, endowing them with interests and problems to be overcome. By framing a problem in such a way that it determines a set of actors, the translating actor defines and aligns the other actors' interests with his own (Callon 1986). The problem is framed in to establish the translating actor as an *obligatory passing point* by enrolling and mobilising the other actors to pass through this point to achieve their interests.

Translation is therefore the process of enrolling a sufficient body of actors by aligning these actors' interests so that they are willing to participate in particular ways of acting. It implies definition, and this definition is *inscribed* in material intermediaries (Latour 1986). These intermediaries are actors in their own right. They are delegates who stand in for and speak for particular interests; they are the medium in which interests are inscribed. The operation or translation is therefore triangular: it involves a translating actor, actors that are translated, and a medium in which the translation is inscribed.

The Case of Rewriting and Replacing Portage

GNU/Linux distributions, complete operating systems that integrate the Linux operating system kernel with a collection of software libraries and applications, are an intrinsic part of the success of Linux. Since the beginning of the Linux kernel development in the early 1990s, communities of OSS developers have created GNU/Linux distributions. As GNU/Linux distribution consists of thousands of different software libraries and applications, distribution developers primarily repackage third-party OSS, doing whatever adaptations required for the third-party software to function on their specific GNU/Linux distribution. At the time of writing, there are over 300 Linux distributions, large and small–some developed commercially, others developed by volunteers–registered with the DistroWatch (2006) Web site. In this paper we report from a study of the OSS community developing the Gentoo Linux distribution, rated by DistroWatch among the ten most widely used distributions.

Starting out as a one-man volunteer project in 2000, by 2003 the number of volunteer Gentoo developers had grown to over 200. The number of third-party software libraries and applications, collectively labelled packages, supported by the Gentoo Linux distribution had also grown. From being a GNU/Linux distribution, Gentoo had over time been turned into a generalized software system for distributing OSS software packages for different Unix operating systems like BSD and MacOS. By 2003 Gentoo suffered increasingly from growth pains.

Organizationally, they Gentoo developers addressed the growth pains by introducing a formal management structure in June 2003: "The purpose of the new management structure is to solve chronic management, coordination and communication issues in the Gentoo project" (GLEP 4). Technically, by mid-2003 growth pains were putting a strain on the Gentoo package manager, Portage, the software that integrates packages on local Gentoo systems. It is from the repeated attempts at rewriting and replacing the package manager that we report in this paper. Although all of the Gentoo developers can agree that the package manager needs to be rewritten and replaced, this turns out to be problematic. After numerous attempts, the Gentoo developers give up. Why is it that they fail to rewrite and replace the package manager? We provide an overview of these

attempts in the rest of this section, before we address the above question during the discussion in section 4.

First attempt

It is mid-November 2003. Four developers make a forceful declaration of intent during the biweekly Gentoo managers' meeting: "We are aggressively working on plans for next generation Portage, which is not going to simply be a rewrite or a new version but beyond people's wildest expectations". The source code of the current version of Portage "is very fragile [because it has] evolved rather than being designed". It has become difficult to comprehend and maintain, preventing the Gentoo developer community at large from participating in developing and maintaining the package manager. Currently, only a "small group [of Gentoo developers] really know how to make significant contributions to the code".

To enrol the Gentoo developer community with the rewrite effort, the four developers provide an architecture diagram (see Figure 1). The diagram graphically lays out the main parts of the package manager, the interface between these parts of the system, and which features will be supported as components.

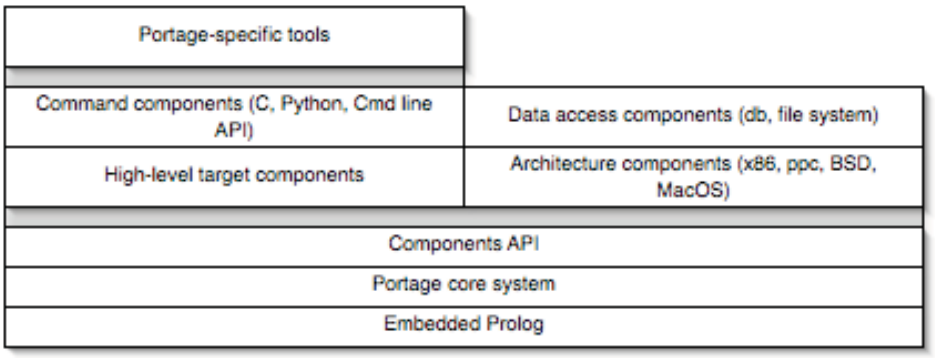


Figure 1 Portage-ng architecture diagram

By rewriting Portage with a core system and "a solid API for components [where] major parts that are now core Portage are going to be implemented as components", the four developers explain, "components can be developed by different teams [of Gentoo developers", turning Portage into "a true community project". To achieve this end, they continue, Portage "is not just to be 'robust enough' but incredibly reliable".

The architecture diagram serves to meet the interests of two other actors. Performance of the package manager has been a point of discontent among the people administrating Gentoo systems. Furthermore, a number of Portage-specific applications that are part of the Gentoo software distribution operate directly on

Portage's database and configuration files. A recurring problem with changing the format of these configuration files and databases, is that some of the Portage-specific tools cease to function. To meet these interests, the four developers are developing a prototype of the core system.

The prototype is realized in GNU Prolog, as this programming language can meet the above interests. Prolog can provide "robust, provably correct code". GNU Prolog has an API for components to be written "in C for performance when needed". However, the final choice of realization language is to grow out of the requirements. "Right now," the four developers explain, "we are at the blueprint stage … the plan is to get a solid blueprint, then make it a community project at the earliest possible point". While the four develop the blueprint and the prototype, they enrol the Gentoo developer community at large to formulate requirements for the rewrite.

No one in the community questions the rationale for rewriting Portage from scratch with a modular architecture. However, the choice of Prolog for a prototype produces resistance. How can Prolog resolve the problem of performance, when "Prolog could be very slow"? one developer asks. Also, how can Portage be turned into a true community project when only very few Gentoo developers are familiar with the predicate-logic programming paradigm of Prolog? The choice of realization language will produce a high entry-barrier, some developers argue.

The promised Prolog prototype fails to manifest, and in mid-December 2003 a competing prototype realized in Ada appears. Throughout November and December the four developers planning to rewrite Portage keep on trying to enrol the Gentoo developer community with their plan by pointing out time and again that the choice of realization language is to emerge from the requirements. However, instead of formulating requirements, the Gentoo developer community delve into endless discussions about the best programming language for rewriting Portage.

By February 2004 all activities on this attempt to rewrite Portage have ceased.

Second attempt

On February 18 2004 a new CVS module called Portage-mod is imported into the Gentoo CVS repository with the following note attached: "All current work between me and George moved from remote cvs to Gentoo cvs!". Where Portage-ng is a complete rewrite of Portage from scratch, Portage-mod is an effort to take the existing Portage code and modularize it. Niles, a Gentoo developer, is heading the effort with help from George, a newcomer to Gentoo and not yet an official Gentoo developer.

While Niles is modularizing the existing Portage source code, George will help writing unit tests. According to the README file imported with the CVS

module, the plan is that the "[d]evelopment of a package structure should facilitate the later development of an consistent Portage API, development of this API is part of this project and development should … begin once Portage modularization is done and a unit testing framework is done."

Development on Portage_mod is undertaken in parallel with the continued development and maintenance of Portage. When the code is modularized, the plan is to rework changes made to Portage during the period of modularization into the modularized version. However, it turns out that the changes made are too significant to achieve this, and this second attempt at rewriting and replacing Portage is laid to rest.

Interlude

"I have a feature request for you", Bob states on the Portage developers' IRC channel. It is mid-April 2004. Bob is a newcomer to the Gentoo community, having only recently been adopted by the Gentoo community to introduce web application support for Gentoo. "The configuration tool for web applications need to edit the Portage database," he continues, "so that a single web application may be installed multiple times on different locations in the file system. " The Portage developers cannot see the purpose of such functionality. A discussion ensues. In the end Bob argues that if the Portage developers cannot provide this functionality for him, he cannot provide support for web applications in Gentoo. Reluctantly the Portage developers agree with Bob about a technical solution to address his requirements.

Third attempt

In wake of the second attempt at rewriting Portage, the remaining developer from that effort sets out to write an API on top of the existing implementation of Portage. There is unanimous support for this effort among the other Gentoo developers. The effort, while a continuation of parts of the second attempt at rewriting Portage, also enrols the interests of two other developers who have been working to establish an API to insulate Portage-specific applications from Portage's configuration files and databases. This will solve the recurring problem of these applications breaking when the format of the configuration files and databases are changed. Furthermore, the API will insulate the core functionality of Portage, so that after the API is in place modularization of Portage may find place without disrupting users.

Work on this third attempt at rewriting Portage ceases after a month and a half. The developer working on the API explains the situation:

> The whole API was designed around a single using application [that] would instigate the reading of the configuration, etc. … that doesn't fit in at all with distributed computing and/or remote management [which is something] people will ask for and/or want to implement

> themselves down the track. [It is therefore] better to preempt it now than find we've shot ourselves in the foot later.

The new approach for Portage is to completely rewrite it with a core running as a Unix daemon with user applications calling the daemon remotely.

Upon the first author ending the fieldwork in December 2004, there are two independent efforts at rewriting Portage. One effort by a young engineering student who has rewritten the core Portage functionality in C, who fails to attract the Portage developers' attention. Another effort by one of the Portage developers to use experience from Portage to write an independent package manager. This, he specifies, is "not a Portage killer, but rather an independent implementation". However, in the future, his package manager may come to replace Portage. As of writing this paper in November 2006, a new version of Portage 2.0.51 is released, being simply the same code as in 2003 only with bug fixes and feature enhancements.

Although all of the Gentoo developers can agree that the package manager needs to be rewritten and replaced, after numerous attempts they give up. Why is it that they fail to rewrite and replace the package manager?

Discussion

A number of problems are raised in connection with rewriting Portage. Complex interdependencies between both modules and functions within the software makes it is difficult to understand parts of the software without a complete understanding of the whole. Interdependencies also make it difficult to make changes without breaking existing functionality. Because of this, only four Gentoo developers know the source code well enough to make changes. Combined with the recurring problems of third-party applications, many of which operate directly on Portage's different data bases with their proprietary data structures, ceasing to function after changes have been made to Portage, the number of developers who can make meaningful changes to Portage limits its continued development and maintenance of Portage.

This is the situation that the Gentoo developers time and again present and draw upon for motivating and explaining the interests and interest groups for rewriting the Portage code and to justify their suggested solutions. The texture of the situation remains largely unchanged throughout the period. The problems they frame and the interests the Gentoo developers construct all emerge from this context. In this section we will look closer at how this context enables and constrain the process of rewriting and replacing Portage.

Mobilizing resources, balancing interests

Why do the repeated attempts at rewriting Portage fail? Towards the end of April 2004, the Gentoo developers describe the first attempt at rewriting Portage as "hot air", "vaporware", and "mostly a buzzword". A predominant explanation for the repeated failures is exemplified by the following quote:

> A rewrite is a MAJOR waste of extremely limited resources. Unless Gentoo gets MANY more Portage devs OR can manage without a Portage update for 6-12 months, a rewrite won't happen in any reasonable time … In the mean time, what happens with the existing implementation? Do you [have people] work on it? Or do you let it sit idle/stagnant. The amount of time it'd take would really drag out on the developers that want new features and simplifications … Resources are why the rewrites failed.

The issue of limited resources is the recurring explanation. The demise of both next generation Portage and Portage modularized are explained in terms of the strain on developer resources. However, given the number of Gentoo developers, the programming resources within the community are significant. It is these resources the next generation Portage developers want to tap in by turning Portage into "a community project". It is therefore not because resources themselves are scarce that the rewrite efforts fail. The problem facing those who want to rewrite Portage can be framed by Glass (1999, p.104)'s befuddlement: "I don't know who these crazy people are who want to write, read and even revise all that code without being paid anything for it at all." Similarly, based on the observation that the interests, needs, and know-how of OSS community members varies greatly, Bonaccorsi & Rossi (2003, p.1244) asks: "[h]ow is it possible to align the incentives of several different individuals"?

It is this selfsame problem the various efforts to rewrite Portage is facing: how to align the interests of the community at large in order to mobilize the resources for rewriting? In the first attempt at rewriting Portage, turning the package manager into "a true community project" goes through the four developers who will rewrite Portage with a core system and "a solid API for components [where] major parts that are now core Portage are going to be implemented as components". By framing a set of problems and actors whose interests are blocked by these problems, the four developers tries to mobilize resources (Callon 1986) for rewrite and replace Portage. These translations are summarized in Figure 2 below.

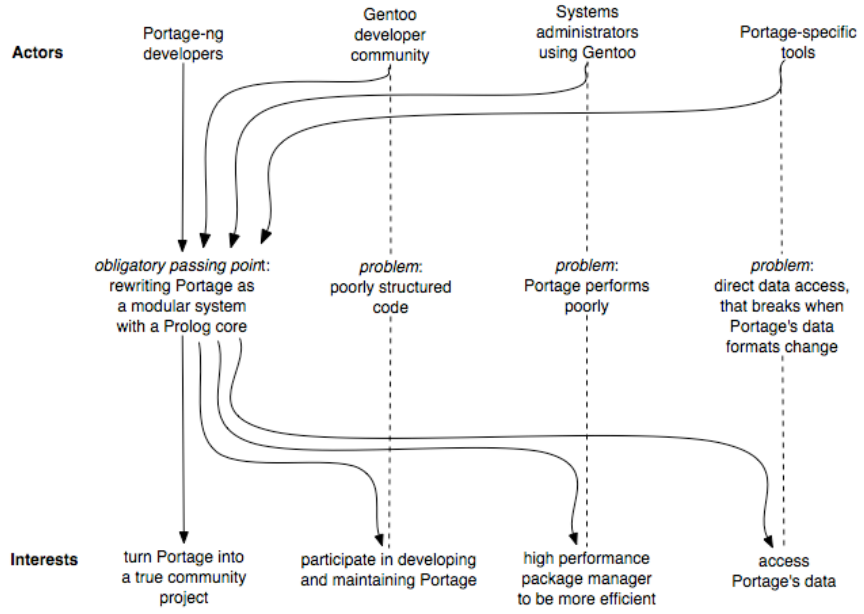


Figure 2 The Portage-ng developers' translations

However, it is not only a question of mobilizing any odd resources. The problem of the next generation Portage developers is that they want to mobilize particular resources. By translating interests into modules that clearly delineated boundaries between actors and their interests, and by inscribing these as boxes in an architecture diagram, the four developers make the architecture diagram stand in for their translations, making them more durable. Through the use of boxes, labels, and clearly separating between boxes, the architecture diagram provides an overview of dependencies between various parts of the architecture; in other words: it inscribes a sequence of work.

By saying that the programming language for realizing next generation Portage is to emerge from the requirements, they are mobilizing resources to do the requirements work first, while leaving to the small next generation Portage team to write the core system first. As such, the resources they want to mobilize are for writing the plugins. However, the effect of proposing Prolog in the design and for the prototype is that resources are spent in discussing implementation language details and problems with using Prolog. While the Prolog prototype is intended to act as a focal point for mobilizing resources for developing plugins, as it fails to materialize there is no mobilization and resources become scarce.

However, the explanation that resources is the reason why the rewrites failed has to been seen in as deeply embedded in and emerging from the context. It is

worth noting that although a number of objections over the plan for the first attempt at rewriting Portage, nobody questioned the feasibility of the effort. Yet, six months down the line, the Gentoo developers argue that lacking resources is why the effort failed. What has happened?

Resources are scarce because there is a competition for resources within Gentoo, as well as the constant need to attract new developer resources. The whole Gentoo effort relies on the sustained interest of users and developers. As observed with many large OSS projects, the key process for quality assurance is users reporting failures to the developers (Feller & Fitzgerald 2002). As Mockus et al. (2004) observes: the number of people reporting software failures greatly exceeds the number of developers. The sustained interest of user is therefore important for the Gentoo community.

The mechanism for sustaining this interest lies in the continued improvement and enhancement of the software, "improvements and simplifications" as put in the above quote. What we see throughout the period is therefore that the existing Portage application continues to change. Attracting new developers is a concern for the community, as the number of unresolved failure reports is continuously growing for Gentoo. Adding functionality to Portage is also seen as a way of recruiting new developers. A concrete example is the way Bob is recruited to the community by the promise that he can implement web application support for Gentoo. However, being a member of the community involves responsibilities, and resolving failure reports is one of these responsibilities. So, recruiting new developers by adding new features to Portage is not only a way of enhancing the software, but also a way of mobilizing resources for addressing the growing number of failure reports.

When the Gentoo developer above questions how the Gentoo community can manage without a Portage update for 6 to 12 months, he is alluding to constant need for balancing between the need for technical stability for rewriting Portage on one hand, and the need for adding new functionality to attract new development resources and keep existing developers interested in the project.

Transition strategy as a processes

Whereas in Jørgensen's (2001) description of the process of rewriting the FreeBSD kernel the scope of the changes and the sequence of actions seem unproblematic, we see that rewriting and replacing Portage is a continuous process of negotiating over the scope of the changes to be made, their sequence, and which actors to be involved in the process. It is about formulating a transition strategy (Monteiro 1998) for the transition from one version of the package manager to the other.

Formulating this transition strategy is a process of continuously balancing numerous interests. On the one hand there is the interest in keeping stable the

features of the software to be rewritten. On the other hand, use of the software to be rewritten continues to evolve and users have interest in the existing software to evolve accordingly. A balance must be struck between these interests. However, this balance point is continuously negotiated and renegotiated, and any attempt to rewrite the software has to remain flexible to these changes.

As much as formulating a transition strategy is about imposing stability of the entire package manager, it is a negotiation over what parts to keep stable and what to change. We see this in the focus in the attempts to rewrite Portage: going from a complete rewrite of the whole artefact, to a modularization of the existing code, to the introduction of an API on top of the existing code. It is a longitudinal process of translation spanning months, during which the identity of actors and the boundaries of what is to remain stable with Portage and what can change are continuously negotiated. The actors' margins of manoeuvre, their possibilities of making incontestable statements about the efforts to rewrite and replace, is delimited through this process of translation.

When one of the Portage developers in hindsight says that rewriting Portage from scratch "is a MAJOR waste of extremely limited resources", the statement tells us nothing about why next generation Portage failed. Nor does it tell us anything general that rewriting software from scratch requires a lot of resources. Rather, the statement bears testament of how the Gentoo developers' margins of manoeuvre is limited by the installed base. There is no longer room to state that it is possible to rewrite Portage from scratch. Again, this does not provide us with the means to make generalized statements that rewriting software artefacts from scratch is never feasible because of a continuously changing installed base.

Furthermore, what we see is that to better control the process of rewriting and replacing, the boundaries of the involved actors are limited. From encompassing the entire Gentoo developer community with the rewrite of next generation Portage, the scope of involved actors are seriously reduced in both Portage modularized and the attempts at writing an API on top of the existing code. When a Gentoo developer in hindsight explains that "waiting for the community to provide requirements … doesn't work", the statement tells us nothing about why next generation Portage failed. Nor does it leave us any margins of manoeuvre to make generalized statements about the number of actors involved that can be involved in successfully rewriting and replacing information systems. Rather, what it does tell us is that how the inertia of the installed base limits the Gentoo developers' margins of manoeuvre in making statements about the number of involved actors in the process of rewriting and replacing software.

What we can generalize, however, is this. The formulation of a transition strategy is constituted through a continuous negotiation with the installed base. This process of negotiation is a process of balancing the interests of the involved actors – both technical and non-technical. It is a process initiated by the construction of problems and actors with interests, but it is also a process from

which new problems emerge. With new problems, existing actors change and new actors emerge. As interests "are what lie *in between* actors and their goals, thus creating the tension that will make actors select only what, in their own eyes, helps them reach these goals amongst many possibilities" (Latour 1987, pp. 109-110), new relationships between actors change. As actors and their interests change, so does that which lies in between them: the interests. As such, rather than being an end product in itself, the transition strategy is continuously formulated and reformulated through a process of continuously emergent problems, actors, and interests enables and constraints the task of rewriting and replacing Portage.

Concluding remarks

In this paper we show how a transition strategy for rewriting and replacing OSS emerges from and continuously changes in response to the installed base. There is a mutual relationship between transition strategies and the context of use and development. The way transition strategies changes the context feeds back to change the transition strategy. We show how this mutual influence changes the transition strategy along three axes: the scope of the rewrite, the sequence to replace the package manager, and the actors to be involved in the change process.

While the entire Gentoo community can agree upon the need to replace the existing system, we show how the existing system's ability to continuously meet the community's interests are greater than the perceived benefits of replacing the system. Although the introduction of an API on top of Portage redirects existing connections to Portage, the transition strategies of the Portage developers were unable to redirect new connections to the existing Portage code, like those made for web application support. We show that battling the inertia of the installed base, then, is not only about changing existing connections from the software being replaced towards its replacement (Hanseth and Monteiro 2002). It is also about the ability to redirect new connections to the installed base to the replacement software throughout the process of rewriting and replacement.

In order to understand and analyse processes of rewriting and replacement, it is therefore important to understand the rationalities and logics in play by different actors. It is important not only to take the actors' own explanations of the world for real, but also to understand the logic and rationality of their explanations in the eyes of the other actors without giving any undue privilege to either view. Furthermore, statements of the world need to be contextualized, when were they made and in response to what, in order for the information systems researcher not to be locked into single actors' views as true and thereby seeing other actors' views as false. As information systems researchers it is also important not to lock on to and give priority to some actors' techno-economic rationalities, but rather to

remain sensitive to our own academic techno-economic bias and challenge this through careful analysis of the statements made by those we study.

References

Bass, L., Clements, P., and Kazman, R. (2003). Software Architecture in Practice, Addison-Wesley, Boston, Massachusetts.

Bianchi, A., Caivano, D., Marengo, V., and Visaggio, G. (2003). Iterative reengineering of legacy systems. IEEE Transactions on Software Engineering, 29(3), 225- 241.

Bonaccorsi, A. and Rossi, C. (2003). Why open source software can succeed. Research Policy, 32(7), 1243-1258.

Callon, M. (1986). Some elements of a sociology of translation: Domestication of the scallops and fishermen of St. Brieuc Bay. In The Science Study Reader (Biagioli, M. Ed.), Routledge, New York, New York.

Crowston, K. and Howison, J. (2005). The social structure of free and open source software development. First Monday 10(2).

DistroWatch (2006). The top ten distributions: A beginners' guide to choosing a (Linux) distribution, http://distrowatch.com/dwres.php?resource=major. Last visited: November 25 2006.

Feller, J. and Fitzgerald, B. (2002). Understanding Open Source Software Development. Addison-Wesley, London.

German, D. (2005). Software engineering practices in the GNOME project. In Perspectives on Free and Open Source Software (Feller, J., Fitzgerald, B., Hissam, S.A., and Lakhani, K.R. Eds.), p. 211, The MIT Press, Cambridge, Massachusetts.

Glass, R. (1999). Of Open Source, Linux and Hype. IEEE Software, 16(1), 126-128.

Hanseth, O. and Monteiro, E. (1998). Understanding Information Infrastructures. Unpublished manuscript, available at http://heim.ifi.uio.no/~oleha/Publications/bok.html.

Jørgensen, N. (2001). Putting it all in the trunk: Incremental software development in the FreeBSD open source project. Information Systems Journal, 11(4), 321-336.

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

Latour, B. (1987). Science in Action. Harvard University Press, Cambridge, Massachusetts.

Mockus, A., Fielding, R.T., and Herbselb, J.D. (2002). Two case studies of open source software development: Apache and Mozilla. Transaction son Software Engineering and Methodology, 11(3), 309-346.

Monteiro, E. (1998). Scaling information infrastructure: The case of the next generation IP in Internet. The Information Society . 14(3), 229-245.

Paulson, J.W., Succi, G, and Eberlein, A. (2004). An empirical study of open-source and closed-source software products. IEEE Transactions on Software Engineering, 30(4), 246-256.

Samoladas, I, Stamelos, I, Angelis, L., and Oikonomou, A. (2004). Open source software development should strive for event greater code maintainability, Communications of the ACM, 47(10), 83-87.

Østerlie, T. and Wang, A.I. (2006). Establishing maintainability in systems integration: Ambuguity, negotiation, and infrastructure. In Proceedings of the 22nd International Conference on Software Maintenance, 24-26 September 2006, Philadelphia, Pensylvania,

Paper 8

Østerlie, T., and Wang, A.I. "Debugging Integrated Systems: An Ethnographic Study of Debugging Practice", in *Proceedings of the 23rd International Conference on Software Maintenance (ICSM'07)*, Paris, France, October 2-5, 2007, pp. 305-315.

Debugging Integrated Systems: An Ethnographic Study of Debugging Practice

Thomas Østerlie, Alf Inge Wang
Norwegian University of Science and Technology
{thomas.osterlie, alf.inge.wang}@idi.ntnu.no

Abstract

This paper explores how software developers debug integrated systems, where they have little or no access to the source code of the third-party software the system is composed of. We analyze the practice of debugging integrated systems, identifying five characteristics that set it apart from existing research on debugging: it spans a variety of operating environments, it is collective, social, heterogeneous, and ongoing. We draw implications of this for software maintenance research and debugging practice. The results presented in this paper are based on observations from an ethnographic study of the Gentoo OSS community, a geographically distributed community of over 320 developers developing and maintaining a software system for distributing and integrating third-party software packages with different Unix versions.

1. Introduction

Software maintenance constitutes a significant factor (between 50 and 80 percent) in the total life-cycle costs of software systems [1]. Research suggests that software developers spend much of the maintenance effort simply trying to understand the software [2]. Current research is based on the premise that source code is the primary data source for understanding the software during debugging. Models of software errors proposed in the software engineering literature are based on the premise that software failures can be traced back to faults in the source code [3].

However, with increased attention on systems integration these are problematic premises. In component-based development [4], Web services and service-oriented architecture, along with information and enterprise systems integration [5], systems integrators have limited, if any, access to the source code of the integrated software. Even when integrating with open source software (OSS) components, research suggests that few systems integrators actually access the source code [6]. As such, systems integrators face the situation of having to debug systems without the source code to build an understanding of the problem upon. We therefore ask: without the source code, *how do systems integrators make sense of problems when debugging integrated systems*?

Debugging integrated systems is largely unexplored in the research literature. The debugging process must be understood before it can be improved upon. This motivates a shift of focus from improving the debugging process, towards exploring how software developers debug integrated systems in practice. To this end, we have explored the practice of debugging integrated systems through an ethnographic study of the Gentoo OSS community. Gentoo is a geographically distributed community of volunteer systems integrators maintaining and operating a software distribution system for distributing and integrating third-party OSS with various Unix operating systems. Similar to existing studies of community-based OSS development [7], debugging is a core activity in the Gentoo community's software development process, too. The community is therefore well suited for studying the practice of debugging integrated systems.

The shift of focus towards debugging practice requires that we draw upon research on practice. In this study we therefore draw upon research on practice and problems in organization science. This research shows that in real-world practice problems do not present themselves to practitioners as given [8]. Rather, problems have to be constructed from the materials of problematic situations that are puzzling, troubling, and uncertain. The process of constructing well-defined problems out of problem situations is often called sensemaking [9]. We will use sensemaking as a theoretical lens for exploring the practice of debugging integrated systems.

This paper contributes to debugging research by identifying five characteristics that sets the practice of debugging integrated systems apart from existing research on debugging: it spans a variety of operating

environments, it is collective, social, heterogeneous, and ongoing.

The remainder of the paper is organized as follows. Section 2 presents existing work on debugging, illustrating the central role of source code as data source for debugging. The theoretical lens of sensemaking is also presented here. Section 3 describes the research methods employed and materials collected during the ethnographic fieldwork. Section 4 describes the overall debugging process in the Gentoo community. Section 5 is an analysis of debugging in Gentoo applying the theoretical lens of sensemaking. We conclude the paper by discussing the implications of our findings for software maintenance research as well as debugging practice in Section 6.

2. Related work

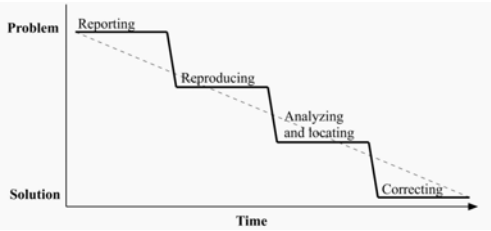
In this section we will illustrate how debugging can be understood as a linear process from problem to its solution. Such a linear model requires that the debugging developer can trace a causal chain from the software failure to its corresponding fault. To this end, we argue, source code is critical. Without the source code, tracing such causal chains becomes harder. This motivates our use of sensemaking as theoretical lens for analyzing how software developers understand problems when debugging integrated systems.

2.1. Debugging

Debugging is the process of locating and correcting the cause of an externally visible error in the program behavior [3]. Araki et al. [10] proposes a model for systematic debugging where debugging is viewed as a process of developing hypotheses about the cause of errors, expected program behavior, and how to modify the program to correct errors, and to refute or verify these hypotheses. Zeller [3] proposes a similar model. These models may be summarized as a stepwise process from a well-defined problem to its solution (as illustrated in Figure 1).

Much of the existing research on debugging focuses on the process of locating the cause of errors. Broadly speaking, three approaches have been suggested [11]. The *bottom-up* approach to debugging involves reading program statements in the source code and chunking these into higher-level abstractions. In the *top-down* approach, software developers reconstruct knowledge about the problem domain and map this to the source code. A mixed model approach has also been suggested.

Figure 1 Linear model of debugging



Several techniques, like delta debugging [3], and tools, like Eden [10], have been proposed to support the process of locating the cause of errors in the source code.

2.2. Sensemaking

The subsection above illustrates how existing research describes debugging as a linear problem solving process, progressing from a well-defined problem to its solution (illustrated by the dashed line in Figure 1). For systems integrators, however, problems do not present themselves as given. Rather, problem situations are ambiguous and open to multiple interpretations [12].

Research within organization science shows that interaction among actors increases in ambiguous situations. In a study of field service technicians repairing copying machines, Orr [13] shows that to make sense of a faulty machine the technicians engage in an ongoing dialogue about the machine with the customer. Similarly, in a study of modern professionals, Schön [8] finds that the daily work of practitioners is not about problem solving, but rather about problem setting; the kind of work professionals undertake to make a situation that is initially ambiguous, puzzling, troubling, and uncertain into something that makes sense.

Confused by ambiguity people engage in sensemaking [9]. The basic premise of sensemaking is that a person or a group's collective experiences of a problem situation are progressively clarified. Rather than starting with well-defined problems, sensemaking is a framework for analyzing how practitioners make sense of a situation that initially makes little sense. In contrast to problem solving starting with well-defined problem, the question driving sensemaking is not 'which of the available means are best suited to solve the problem?′ but rather 'what is going on?'. To make sense of a problem situation, people act on basis of previous experience. By actively engaging with the problem situation, understanding emerges as people make retrospective sense of what occurs by enlarging small cues from the available data and forming a

structure to provide meaning. Another central premise of sensemaking is therefore that action precedes understanding.

3. Methods and materials

This research is based on the first author's ethnographic study of the Gentoo OSS community. This section briefly describes the research setting and the ethnographic fieldwork this study is based upon. A more detailed description of the research including a more thorough discussion on research validation can be found in [12].

3.1. Research setting: Gentoo

Gentoo is an OSS community of volunteers maintaining and operating a software distribution system for distributing and integrating third-party OSS with various Unix operating systems. In addition, the community provides a GNU/Linux distribution, Gentoo Linux, on top of the software distribution system. The community consists of 320 official developers distributed across 38 countries and 17 time zones[1]. To the best of our knowledge, none of the developers are geographically co-located. As with most OSS communities, users are an important part of the Gentoo community, contributing with problem reports as well as source code. However, it is impossible to tell how many users are active in the community at any one time.

For the remainder of the paper we will use the term Gentoo about the Gentoo software distribution system, Gentoo Linux about the GNU/Linux distribution provided on top of Gentoo, and the Gentoo community when talking about the community of volunteer systems integrators. These volunteers call themselves Gentoo developers.

Gentoo distributes third-party OSS packages in the form of installation scripts. The installation scripts are stored in a central repository. One script exists for every version of each of the 8486 supported packages, for a total of 23911 installation scripts. The total SLOC of installation scripts in the repository is 671971[2]. The installation scripts make up 90% of the source code in the repository. The rest are mainly patches and configuration files. The installation scripts are written and maintained by the Gentoo community. While some Gentoo developers may be quite familiar and knowledgeable of the source code of the components they integrate, most treat the software being integrated as a black box. Up to six different Unix versions may be supported by a single installation script: GNU/Linux, FreeBSD, OpenBSD, NetBSD, MacOS X, and Dragonfly. For GNU/Linux, five different processor architectures may also be supported in the script.

The repository is mirrored on every Gentoo system. A Gentoo system is a computer system using Gentoo for integrating third-party OSS on the local system. The Portage package manager is the application that integrates third-party packages locally on Gentoo systems, calculating dependencies to other packages, downloading the source code, as well as configuring, compiling and integrating the package with the Gentoo system's live file system.

3.2. Ethnographic fieldwork

Data was collected during a ten months period of participant-observation. Participant-observation is the predominant method for ethnographic fieldwork [15]. In this study, participant-observation meant that the first author participated in the Gentoo community by submitting and resolving problem reports, interacting with the Gentoo users and developers on Internet Relay Chat (IRC) and e-mail, as well as participating in a major restructuring effort of the Portage package manager.

During the ten months period of participant-observation the first author wrote field notes at the end of each day of fieldwork [16]. In addition to the field notes, four of the Gentoo IRC channels were logged to file, one file per day for each channel, totaling1027 files.

Ethnographic research does not follow a step-wise process [17]. Rather, ethnographic data analysis is an ongoing process from the moment the field worker enters the field until the complete research report is written. During the field work the data analysis was informal. Upon withdrawing from the field, the first author spent a year working systematically through the collected data, looking for recurring patterns. This formal data analysis was a process of incrementally generalizing from a multitude of singular observations to increasingly more generalized descriptions of activities. Throughout the process, non-recurring details of the singular observations were omitted and recurring issues included, leading to the analysis presented in this paper.

4. The Gentoo debugging process

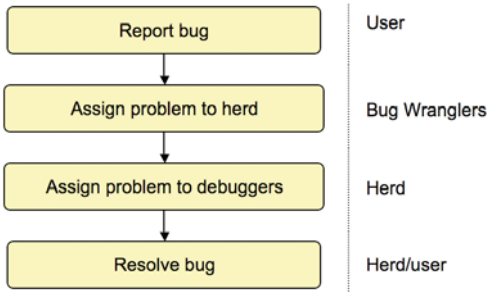
As reported in previous studies of community-based OSS development [7], debugging is a central activity

[1] Unless otherwise stated, all provided figures are of March 30, 2006, the day the fieldwork was concluded

[2] Data generated with SLOCCount [14]

for the Gentoo community, too. New installation scripts are made available in the repository after marginal quality assurance. Instead, users are expected to report problems. As such, debugging plays a central role as a quality assurance mechanism in Gentoo. The debugging process (illustrated in Figure 2) is managed through an installation of the Bugzilla defect tracking system [18]. While Bugzilla is the name of a product, unless otherwise noted we will use the term Bugzilla about Gentoo's installation of this system for the ease of reference.

Figure 2 Overview of the debugging process



4.1. Roles

The distribution of roles in the Gentoo community's debugging process is similar to that reported in existing research on community-based OSS development [7]. Users submit the majority of problem reports in the Gentoo community. The *Bugwranglers* is the name of the change control board responsible for assigning newly submitted problem reports to the relevant herd. A *herd* is a team of Gentoo developers responsible for a collection of third-party packages. There are 124 such herds, varying in size from a single Gentoo developer to over 20 developers. The herd is responsible for resolving problem reports.

4.2. Responsibilities

Gentoo integrates software from hundreds of different third-party providers. When debugging, the Gentoo developers are responsible for problems related to the way the third-party OSS packages are integrated. They are not responsible for resolving defects in the third-party software. Similarly, the Gentoo developers are not responsible for problems related to the configuration of a particular Gentoo system. In the latter case, user support is handled on dedicated IRC channels, mailing lists, and Web forums, not through Bugzilla.

4.3. Submitting and assigning reports

Users submit problem reports when they have run out of resources locally to resolve a problem. New problem reports are submitted through a standardized Web-based form. The form defines a number of fields to describe the problem, including a short description of the failure situation, the operating system and hardware platform of the failing Gentoo system, the component where the problem has occurred, the package's version number, as well a longer description of the problem situation including steps to reproduce, which software packages are affected, the reproducibility of the problem, any error messages generated when the software fails, as well as a standardized systems information of the user's system generated by running a Gentoo-specific tool.

When a new problem report is submitted to Bugzilla, an e-mail is sent to the Bugwranglers' mailing list. The Bugwranglers will assign newly submitted problem reports to the relevant herd.

4.4. Resolving problem reports

Once the Bugwranglers have assigned a problem report to a herd, an e-mail is sent to the herd's mailing list. Herds have different ways of distributing work. Many developers scan incoming problem reports to see if they immediately can resolve the report. Other herds formally distribute problem reports among themselves.

Resolving a problem report does not necessarily mean that the problem itself is resolved. This is one of the ways defect reports are resolved. The other ways are to mark the report as a duplicate, to mark it with the flag NEEDINFO meaning that the user has to provide additional information about the system or software failure itself, to reject the problem report as the problem is a user problem, or to mark the problem report as upstream. The latter option is used when the reported problem is caused by a defect in the third-party software itself.

Reaching the closure with one of the above five resolutions to problem reports requires an understanding of the system causing the software failure. In the next section we will analyze how this understanding is produced.

5. Results and analysis

With basis in the overview of the Gentoo debugging process above, we will now revisit the research question posed in the introduction: how do systems integrators make sense of problems when debugging integrated systems? We do so with the theoretical lens

of sensemaking. Our focus in this analysis is therefore on *what people do*, rather than prescribing what should have been done to improve the debugging process. We do so by identifying five characteristics of debugging an integrated system; (C1) it spans a variety of operating environments, (C2) it is collective, (C3) social, (C4) heterogeneous, and (C5) ongoing.

5.1. C1: Variety of operating environments

Zeller [3] states that to fix a problem, the developer must first be able to reproduce it. Although many reported problems are reproducible, the Gentoo developers often face problems they are unable to reproduce, or at least problems that are not easily reproduced. This is illustrated in Exhibit 1.

Exhibit 1. Excerpt from Gentoo developers' IRC channel (gentoo-dev-2004.04.16)

Developer A: *This particular Web page crashes both the Mozilla and Galeon Web browsers.*
Developer B: *That doesn't happen on my computer.*
Developer A: *I've built the applications for the Athlon T-Bird processor architecture, and both have been compiled with the GTK2 widget library. I generally assume it's my using GTK2 that messes it up.*
Developer B: *It might be GTK2. I've compiled both Web browsers with the GTK1 widget library on my system.*
Developer D: *Well, that page works on my Epiphany Web browser compiled with the GTK2 library.*
Developer C: *And it works with my installation of Mozilla compiled with GTK2.*
Developer D: *This other Web page crashes my Phoenix Web browser, but not Mozilla or Galeon.*
Developer A: *The Web page crashes on my Epiphany installation, as well. It seems it's my Mozilla build that's flakey.*
Developer C: *But boingboing.net crashes my Epiphany installation. I've compiled it for the PentiumII processor architecture, though.*
Developer A: *boingboing.net crashes Galeon on my system, too.*
Developer B: *boingboing.net working for Mozilla on my system.*
Developer C: *Hmm… It seems the problem is related to Mozilla compiled with the GTK2 widget library and the Xft font library. Weird thing is that boingboing works on my Galeon installation…*
Developer A: *Now here's a very good reason to only build for one processor architecture, stable source tree and only do point releases. Variation kills reproducibility.*

All of the four Web browsers mentioned in the exhibit are based upon Mozilla's rendering engine. This rendering engine is integrated on a Gentoo system along with Mozilla. As such, the installation scripts of the other three Web browsers have dependencies to Mozilla.

As developer A observes in Exhibit 1, the variety of operating environments makes reproducing problems difficult. This is similar to Littlewood's [19] explanation of Adams [13] observation that in large-scale software maintenance most reported problems are irreproducible: irreproducibility is an effect of the variety of operating environments in the population of systems. There are three dimensions of variety of operating environments among Gentoo systems: operating system, configuration of individual packages, and system evolution.

Gentoo distributes software for six different operating systems. Although most installation scripts in the Gentoo repository do not support for all of the operating systems at once, many packages support multiple operating systems. However, operating systems work in different ways, as illustrated in Exhibit 2.

Exhibit 2. Excerpt from Gentoo developers' IRC channel (gentoo-dev-2004.10.10)

Developer A: [making reference to a stack trace attached to the problem report being discussed] why is it that the thing [the linker] can't find pthread? is that because of a missing -pthread [flag being passed to the linker]
Developer B: sounds like glibc was upgraded [glibc is a Unix runtime library]
Developer A: an upgraded glibc still has pthreads alright?
Developer A: without that symlink the system grinds to a halt
Developer B: needs -lpthread I guess.
Developer C: well the cross-platform way is to use gcc's -lpthread, because not all systems have libpthread. bsd has libc_r for example

In the above exhibit, understanding the problem is made difficult by the way different operating systems support, in this particular case, multi-threading. Both of the above exhibits illustrate that reproducibility is not only made difficult by the variety in operating environment among Gentoo systems, but the variety of operating environments also makes problem understanding difficult.

Exhibit 1 illustrates how debugging is made further complicated by the way individual packages are configured upon integration with a Gentoo system. Such configuration of individual packages is the second dimension of variety among Gentoo systems. There are two dimensions to individual package configuration: *optionals* and *virtuals*.

Packages can have optional functionality that may be compiled into the package when it is integrated on a Gentoo system. This local configuration of individual packages is similar to what Carney et al. [4] describes as installation-dependent products in COTS development, a form of modification of a generic software product that is intended by the provider but may still vary from system to system. With virtual configuration different third-party packages may provide the same functionality. Exhibit 1 illustrates this, as both GTK1 and GTK2 may provide widget library support for the four Web browsers in question.

Packages depend on other packages. Although such package dependencies are inscribed in the installation scripts, these dependencies are only convenient for reproducing a freshly setup Gentoo system. However, many Gentoo systems have been running for a long time. New versions of packages are continuously being added to the Gentoo distribution system's repository, while old and unsupported versions of packages are

being removed. Yet, how up-to-date every package on a Gentoo systems is, varies greatly. This is the fourth dimension of variety in operating environments among Gentoo systems: system evolution.

Although the number of combinations of packages on a single Gentoo system is finite, package configurations and the effects of system evolution often makes it practically impossible to replicate the system configuration required to reproduce the problem. The situation debugging Gentoo is therefore similar to Araki et al.'s [10] observation of debugging concurrent programs. Because the state of concurrent programs may be non-deterministic, programmers often say that debugging is almost completed when they have figured out how to reproduce the problem. Similarly, the Gentoo developers spend a great deal of time understanding the reported problem. Similar to Schön's [8] observation, problems do not present themselves to the Gentoo developers as given, but have to be constructed from the materials of problematic, uncertain, and puzzling situations.

5.2. C2: Collective

When problems do not present themselves as given, the Gentoo developers need to establish what is going on. A fundamental aspect of sensemaking is that a person or a collective's experiences of a situation are progressively clarified [9]. By collectively engaging with the reported problem, comparing configurations of libraries, processor architectures, and applications, the Gentoo developers collectively work towards an understanding of the problem situation as seemingly "related to Mozilla compiled with the GTK2 widget library and the Xft font library" (see Exhibit 1). By extracting cues from the environment, information about processor architectures, widget libraries, which Web pages crashes which browser, the developers collectively makes sense of the problem situation.

In a study of field service technicians diagnosing and repairing copying machines, Orr [13] describes how technicians and users collectively make sense of faulty machines. Although provided with detailed guidelines for diagnosing and repairing copying machines, service technicians were often faced with confounding machine behavior going beyond the official documentation. To make sense of the faulty machine behavior, the service technicians interact with the customer to create a context for the behavior. By recreating the machine's history, its past quirks and problems, the customer and service technician engage in a process of constructing a context where the service technician can make sense of the faulty machine. Repairing the machine is not a process of finding the problem causing the faulty behavior and then repairing it. Rather, the problem is to understand what the problem is. By interacting with the customer and the faulty machine, the service technician creates a setting where the faulty behavior makes sense and can be resolved.

As the variety of operating environments often makes it difficult for the Gentoo developers to reproduce reported problems, we find the Gentoo developers and users collectively working together to make sense of reported problems. They typically use the problem report for interacting, adding new comments to the *Additional comments* field at the bottom of the problem report as illustrated in Exhibit 3.

Exhibit 3. Excerpt of problem report illustrating use of *Additional comments* field

Description: **Opened: 2003-01-02 02:41**
Basically, I can't even configure the package and it fails complaining 'required file `../depcomp' not found'. I have re-integrated the autoconf and automake packages, but I still get the same problem.
[error output provided]
[systems information provided]

Comment #1 From *Developer A* 2003-01-02 04:32:56
Which version of automake and autoconf do you use?

Comment #2 From *User* 2003-01-02 04:43:33
[version information about automake and autoconf packages on local system provided]

Comment #3 From *Developer A* 2003-01-02 04:52:19
Seems to be the required versions. Could you please try -r16 and/or -r18 of if the xmms package to see if it works for you?

Comment #4 From *User* 2003-01-02 04:58:44
Nope, both die in exactly the same way. I did have this installed at first, but then it tried to update the package a while ago and it just wouldn't install properly. Is there a package that xmms requires that might be broken?
[new error message provided]

Comment #5 From *Developer A* 2003-01-02 05:23:06
Could it be you are running out of disk space or memory and swap?

Comment #6 From *User* 2003-01-02 05:26:27
[information about available diskspace on local system's hard drive partitions provided]
I doubt that diskspace or memory is a problem, although that tmpfs device is a tad full!!
[information about local system's memory use provided]

Comment #7 From *Developer A* 2003-01-02 05:33:02
Version 1.3 of the xmms installation script is latest. which version do you have? Attach the output of the command 'head /usr/portage/media-sound/xmms/xmms-1.2.7-r15.ebuild'

Comment #8 From *User* 2003-01-02 05:35:59
Version 1.3 [output of running head command provided]

Comment #9 From *Developer B* 2003-01-02 06:05:58
This has to do with the version of automake/autoconf being used by the emerge process. My feeling is that xmms is using a version that is not compatible with its config process. I will, therefore, adjust xmms's ebuild to make sure it calls the correct version. Please stand by for an updated ebuild that you can test.

I have never seen this kinda thing with the xmms package.

Comment #10 From *Developer A* 2003-01-18 14:06:31
What about just adding a --add-missing to:
[script provided]

Comment #11 From *User* 2003-01-19 18:04:44
Ok, really strange... I just integrated the KDE-3.1_rc6 package and xmms installed without any problems.

Comment #22 From *Developer A* 2003-01-19 19:02:59
Well, I did add the --add-missing to the -r18 of the xmms installation script anyhow.

Exhibit 3 illustrates how Gentoo developers and users collectively work together to make sense of reported problems. 17 days pass from the date the problem is reported until it is resolved illustrating that collective debugging can be a longitudinal process.

The exhibit illustrates a typical exchange, where the developer asks the user to generate new data about the failing Gentoo system. This often entails the user running one or more diagnosis tools, producing output texts that are attached to the failure report. As illustrated by Exhibit 1, developers often use IRC for discussing problem reports in detail. There is a mailing list that is used for this, too. The user is often asked several times to generate new information, in a cyclic process between users producing data and developers interpreting the available data [12].

This observation is somewhat different to existing reports from community-based OSS development. Huntley [20], for instance, argues that debugging is a task that in nature lends itself to distribution, as finding problems is "a task that can be performed by thousand or even millions of end-users without any involvement of the core development team". Since most software failures are limited in scope, he continues, involving only a small fraction of the code, a well-controlled debugging process can be distributed among large number of programmers. We, on the other hand, observe that such a distinct separation between describing and understanding problems is problematic.

Contrary to the clear separation between problem description and analysis on Figure 1, Exhibit 3 illustrates that the process of making sense of the problem is not decoupled from the process of describing the problem. To make sense, the Gentoo developers and the user must act by engaging with the problem. This may seem like a process of trial-and-error, but from a sensemaking perspective action precedes understanding [9]. By collectively engaging with the problem, the Gentoo developers and user create materials from which they may construct the problem. As such, the debugging process observed in the Gentoo community is more a process of creating the problem retrospectively, rather than being driven by a process of formulating hypotheses and rejecting or verifying them. It is therefore a process driven by plausibility rather than accuracy [9]. Failed efforts to solve problems feed back into the debugging process with new materials to set the problem anew, or with requests for new information from the user. The problem and its causes are constructed in retrospect, once the solution is in place.

5.3. C3: Social

Debugging plays a key quality assurance mechanism in Gentoo. Installation scripts are released with only a minimum of quality assurance; with the expectation that problems related to way software is integrated on Gentoo systems will be reported. Exhibit 3 illustrates how debugging can be a longitudinal, although low-intensity activity. Although a low-intensity activity, Exhibit 4 shows the sheer number of problem reports submitted to Bugzilla on a weekly basis exceeds the number of problem reports the Gentoo developers are able to close. The exhibit is based on Bugzilla statistics published by the Gentoo developers in the Gentoo Weekly Newsletter [21]. The developer count is generated from the Gentoo developer list [22].

Exhibit 4. Weekly debugging workload

Date	New reports	Reports closed	Open reports	Number of developers
January 6 2003	269	Not avail.	1893	102
January 5 2004	837	428	4479	259
January 3 2005	700	390	7877	Not avail.
January 16 2006	799	447	9083	320

The increasing gap between new and closed problem reports may be partly explained by the way the Gentoo community uses Bugzilla; problems reported on outdated versions of packages are ignored and never marked as resolved, and the Gentoo developers use problem reports for tracking issues as well. Yet, despite a steady increase in the number of Gentoo developers, the workload of debugging exceeds the capacity of the Gentoo developers as the increasing number of open problem reports show. There is therefore a need for the Gentoo developers to prioritize among problem reports.

The problem report provides a field for rating a problem's severity. However, an understanding of problems often is retrospective (Section 5.2). Knowing the severity of a problem is therefore also retrospective, and prioritizing is therefore problematic without starting to make sense of the reported problem.

In this situation, the Gentoo developers have to balance between several interests. On the one hand, they have to prioritize problem reports that may potentially affect the many Gentoo systems. That reported problems are reproducible imply that the problem may affect many systems. Prioritizing reproducible problem reports comes at the expense of

irreproducible problem reports, or reports on problems that occur only on one or few systems.

Commenting on similar tradeoffs for prioritizing problem reports reported by Adams [23], Littlewood observes that with a large population of operating environments there may always be one or more problems that are unique to a particular user's operating environment. However, the user would be extremely disgruntled if the problem was not resolved, as the problem would be recurring at appreciable rates in his environment.

Similarly, because of the variety of operating environments among Gentoo systems, many reported problems will not be reproducible and are particular to a single or a small group of Gentoo systems. For the debugging process to function properly as a quality assurance mechanism, the Gentoo developers have to keep users interested in submitting problem reports in the future. The developers therefore have to balance the need for resolving problem reports that will increase the reliability of Gentoo for the most users, with debugging problems they are unable to reproduce or problems that are particular to a single user's system.

To curb the workload, enforcing the boundaries of one's responsibilities is an important part of debugging practice. Although the responsibilities are clearly delineated in theory (section 4.2), establishing responsibilities is more of an open question. Zeller [3] views such a lack of clarity as a political process of deciding who is to blame. Assuming the perspective of sensemaking, however, determining responsibilities is an inherent part of the retrospective process of making sense of problems. Exhibit 5 is a dialogue aggregated from a problem report and discussions about this problem report on the Gentoo developers' IRC channel.

Exhibit 5. Enacting responsibilities

Statement	Researchers' commentary
***Reporting user**: I have installed my system from scratch*	The problem is related to the way Gentoo integrates software, and therefore the Gentoo developers' responsibility
***Developer A**: [making reference to the systems information provided with the problem report] Is using an x86 profile for an amd64 machine troublesome?*	The reported problem is related to the way the user's Gentoo systems configuration; therefore the user's responsibility
***Developer B**: [making reference to the installation script] Turning off the optional esound support might solve the problem.*	The problem may be related to how the package integrates with the esound package, and the third-party provider's responsibility.
***Developer A**: [making reference to the compiler error provided with the problem report] Why is it that the thing can't find pthread? is that because of a missing -pthread*	The problem is related to the use of the pthreads library, and therefore the responsibility of another herd.
***Developer B**: sounds like the glibc library was upgraded*	Related to the user's system configuration, and his responsibility

By extracting cues from the situation ('installed my system from scratch'), from the systems information and error messages provided with the problem report, as well as information from the installation script, the Gentoo developers and the user bridge the ideal division of responsibilities (Section 4.2) and the concrete details of the problem. They produce a reality of responsibilities by their actions. However, this construction of reality is in itself constrained by their understanding of responsibilities. The model of responsibilities precedes the discussion of the particular problem, acting as a guide for extracting cues from the data. As Weick [9] puts it, sensemaking is enactive of sensible environments.

Although debugging is a technical activity, the above analysis shows how social issues like keeping users interested and determining responsibilities are closely intertwined with the technical activities of debugging.

5.4. C4: Heterogeneous

Heterogeneity is one of Hasselbring's [5] three characteristics of systems integration: "heterogeneity comes from different hardware platforms, operating systems, database management systems, and programming languages". This is similar to what the variety of operating environment among Gentoo systems (section 5.1). Similarly, Belady & Lehman [24] presents variety as a root cause of program largeness. While Hasselbring's notion of heterogeneity is technical, Belady & Lehman's understanding of variety includes both the social and the technical.

Similar to Hasselbring, we find heterogeneity to be a characteristic of the Gentoo debugging process, but like Belady & Lehman our view of heterogeneity transcends the technical. While the purpose of the debugging process is to keep Gentoo running, we find that keeping Gentoo running is not solely a technical endeavor. Rather, to keep Gentoo running requires maintenance of both the technology and the community. The debugging process is therefore heterogeneous in the sense that it serves a variety of interests and activities, where the social and the technical are closely intertwined (Section 5.3).

Section 2.1 shows that existing research is based on the premise that source code is the primary data source for debugging. Debugging Gentoo is heterogeneous in the respect that instead of relying on source code, understanding of problems is constructed from a heterogeneous ensemble of data sources: problem reports, debug data generated by the failing software and various diagnosis tools, as well as discussions on IRC, mailing lists and Web forums (Sections 5.1 and 5.2).

5.5. C5: Ongoing

Although debugging is a central activity in the Gentoo community, it is not the only responsibility the Gentoo developers have. Within the community, the developers are responsible for keeping abreast with the latest developments for the third-party OSS packages of their herd–writing new installation scripts and updating existing scripts to incorporate patches made available outside of the packages' release cycles–as well as being active on the IRC channels and mailing lists discussing and assisting other developers. In addition, the Gentoo developers have outside responsibilities like daytime jobs, and school.

As such, debugging is one activity in the ongoing flow of activities making up the day of the Gentoo developers. While it may be a low-intensity activity (see Section 5.3), debugging is not an activity the developers can devote all their attention to as illustrated by Exhibit 6.

Exhibit 6 Extract from the Gentoo developers' IRC channel (gentoo-dev-2004.07.17)

> ***Developer A****: Have you ever taken a look at bug 33877 ?*
> ***Developer B****: Yes, but there's a contention for my time. Getting Java working well has been a higher priority.*

The amount of problem reports to be addressed makes debugging a time-consuming activity. Although reflecting upon alternative interpretations of the problem situation (see Exhibits 1,2, and 3), the resources available for rigorous analysis of the problem situation are limited. Instead, the Gentoo developers often act to get a better understanding of the problem. As such, they engage in sensemaking rather than problem solving.

To cope with these constraints the Gentoo developers have to be pragmatic. Problem solving is the selection of the best-suited means to an established end. While the debugging literature presupposes that the end to be met is to correct the reported problem, we find the debugging process is equally much about establishing such ends. Schön [8] argues that by focusing on problem solving, we ignore the problem setting: "the process by which we define the decisions to be made, the ends to be achieved, the means that may be chosen".

As such, solving reported problems is but one of many outcomes of the debugging process. The process of problem setting need not conclude that there is a problem. The overarching goal of the debugging process is to reach a closure for problem reports. Resolving a problem report is not synonymous with solving the reported problem. It may be, but problem reports are also resolved by providing users with workarounds for the reported problem, by concluding that the problem is local to the user's system, or by concluding that the problem is in the third-party software.

6. Discussion and concluding remarks

In this paper we have explored how software developers debug integrated systems. We identify five characteristics of the debugging process: it spans a variety of operating environments, it is collective, social, heterogeneous and ongoing.

This description differs from the debugging process described in the research literature. It is less of a linear process going from a well-defined problem to its solution, and more of a cyclic process where the problem is not always understood before there is a solution to it [12]. The debugging process is a collective sensemaking process [9], influenced by both social and technical factors, rather than a purely individual cognitive problem solving activity [2]. In contrast to researchers' advocating a hypothesis-driven debugging processes [3, 10], we find the Gentoo community's debugging process to be driven by plausibility rather than accuracy.

This suggests, then, that the software failure is not unproblematic as a phenomenon, but rather subject to interpretation and negotiation. Software developers' understanding of what constitutes a software failure is contingent upon situational issues such as workload, priorities, responsibilities, as well as technical data. Furthermore, this research illustrates that software failures are not necessarily stable.

This has implications for software maintenance research on integrated systems, as it raises concerns about the appropriateness of assuming that software failures are clearly identifiable and stable phenomena. That there is a clearly identifiable relation between the errors in the code and the observed failures is too simple. In system integration the problem is more complex.

Although apprehensive about generalizing from a single case study, we contend that our findings may have implications for debugging practice. An important problem with debugging integrated systems is to understand what the problem is. It is therefore difficult to determine what data is relevant prior to engaging with the problem. Comprehensive schemas for classifying problems as proposed by various defect classification standards, but also found in many defect tracking systems including Bugzilla, are of limited use. Instead, defect tracking systems need to support interaction between the reporting user and the software developer resolving the reported problem. Users have

little understanding of what is relevant for debugging the system. As such, defect tracking systems need to provide reporting users with simple guidelines for describing the problem situation and what information to be provided for bootstrapping the debugging process.

References

[1] F. Calzolari, P. Tonella, and G. Antoniol, "Dynamic model for maintenance and testing effort," in *International Conference on Software Maintenance, ICSM'98* Bethesda, Maryland, 1998.

[2] C. L. Corritore and S. Wiedenbeck, "Mental representations of expert procedural and object-oriented programmers in a software maintenance task," *International Journal of Human-Computer Studies,* 50(1): 61-83, 1999.

[3] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA: Morgan Kaufman Publishers, 2006.

[4] D. Carney, S. A. Hissam, and D. Plakosh, "Complex COTS-based software systems: practical steps for their maintenance," *Journal of Software Maintenance: Research and Practice,* 12(6): 357-376, 2000.

[5] W. Hasselbring, "Information Systems Integration," *Communications of the ACM,* 46(6): 33-38, June 2000.

[6] J. Li, R. Conradi, O. P. N. Slyngstad, C. Bunse, M. Khan, M. Torchiano, and M. Morisio, "Validation of New Theses on Off-The-Shelf," in *11th IEEE International Metrics Symposium*, 2005, p. 26.

[7] K. Crowston and J. Howison, "The social structure of free and open source software development," *First Monday,* 10(2): 2005.

[8] D. A. Schön, *The Reflexive Practitioner: How Professionals Think in Action*. Aldershot, UK: Ashgate Publishing Limited, 1991.

[9] K. E. Weick, *Sensemaking in Organizations*. Thousand Oaks, CA: SAGE Publications, 1995.

[10] K. Araki, Z. Furukawa, and J. Cheng, "A General Framework for Debugging," *IEEE Software,* 8(3): 14-20, May 1991.

[11] A. von Mayrhauser and A. M. Vans, "Program Comprehension During Software Maintenance and Evolution," *IEEE Computer,* 28(8): 44-55, August 1995.

[12] T. Østerlie and A. I. Wang, "Establishing Maintainability in Systems Integration: Ambiguity, Negotiation, and Infrastructure," in *The 22nd IEEE International Conference on Software Maintenance* Philladelphia, PA, 2006.

[13] J. E. Orr, *Talking About Machines: An Ethnography of a Modern Job*. Ithaca, NY: Cornell University Press, 1996.

[14] D. A. Wheeler, "SLOCCount", http://www.dwheeler.com/sloccount/. Last accessed April 12 2007.

[15] D. L. Jorgensen, *Participant Observation: A Methodology for Human Studies*. Thousand Oaks, CA: SAGE Publications, 1989.

[16] R. M. Emerson, R. I. Fretz, and L. L. Shaw, *Writing Ethnographic Fieldnotes*. Chicago & London: The University of Chicago Press, 1995.

[17] D. M. Fetterman, *Ethnography: Step by Step*, Second ed. Thousand Oaks, CA: SAGE Publications, 1998.

[18] M. P. Barnson, "The Bugzilla Guide - 3.1 Development Release", http://www.bugzilla.org/docs/tip/html/. Last accessed 21 March 2007.

[19] B. Littlewood, "Why did Ed Adams see so many small bugs?," *Software Reliability and Metrics Newsletter,* 4): 31-34, 1986.

[20] C. L. Huntley, "Organizational Learning in Open-Source Software Projects: An Analysis of Debugging Data," *IEEE Transactions on Engineering Management,* 50(4): 485-493, November 2003.

[21] "The Gentoo Weekly Newsletter", http://www.gentoo.org/news/en/gwn/gwn.xml . Last accessed March 20 2007.

[22] "Gentoo Linux Active Developer List", http://www.gentoo.org/proj/en/devrel/roll-call/userinfo.xml. Last accessed April 12 2007.

[23] E. N. Adams, "Optimizing Preventive Service of Software Products," *IBM Journal of Research and Developmen,* 28(1): 2-14, January 1984.

[24] L. A. Belady and M. M. Lehman, "The Characteristics of Large Systems," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, Mass.: The MIT Press, 1978, pp. 108-138.