Nils Brede Moe

From Improving Processes to Improving Practice

Software Process Improvement in Transition from Plan-driven to Change-driven Development

Thesis for the degree of Doctor Philosophiae

Trondheim, September 2011

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

NTNU – Trondheim
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the degree of Doctor Philosophiae

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

ISBN 978-82-471-3010-0 (printed ver.)
ISBN 978-82-471-3011-7 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2011:226

Printed by NTNU-trykk

The real voyage of discovery consists not in seeing new landscapes
but in having new eyes

- Marcel Proust

Abstract

As information technology's role in the modern economy grows in importance, society makes exponentially greater demands on the diversity and quality of the software being produced. To develop high quality software, a good software development process is important. Software process improvement is about improving software quality and reliability, employee and client satisfaction, and return on investment. From the mid 1990s onwards, agile software development has been challenging the traditional (plan-driven) view of software development. Agile software development accords primacy to uniqueness, ambiguity, complexity, and change, as opposed to prediction, verifiability, and control.

The fundamental differences between traditional and agile software process improvement and the lack of research on these differences, gave rise to the overall problem addressed by this thesis: "How does Software Process Improvement work change with the introduction of agile software development in plan-driven companies?" This thesis focus on answering the following research questions: What characterizes SPI in plan-driven companies?, What characterizes SPI in change-driven companies?, and What are the key SPI challenges when implementing change-driven development?

This thesis summarizes six years of studies in three small and medium-sized companies in Norway. The overall research method has been the same: case study and action research. Qualitative data in the form of interviews and participant observations constitute the most important source of evidence.

Through a synthesis of contributions from twelve papers, ten key findings connected to the three research questions has been identified. To summarize, software process improvement in plan-driven companies is characterized by a participative bottom-up approach when creating company best practice, focus on project management support, high individual autonomy, and long cycles of single-loop learning; the goal of reflection on projects is to improve future projects. Software process improvement in change-driven companies is characterized by supporting the whole team and not only project management, practice is improved by short cycles of single-loop learning, and the goal of reflection in projects is to improve the current project. Finally, software process improvement challenges while implementing change-driven development are to increase redundancy to create conditions for the team to self-manage, to learn how to learn, and to perceive the adoption of change-driven development as a large, long-term organizational change project.

The overall contribution of this thesis is that it shows empirically that the goal of software process improvement changes from improving processes to improving practice. However, achieving this goal is challenging when only part of the organization is involved in the change. Also contributions are deep knowledge about software process improvement in plan- and change-driven small- and medium-sized companies, knowledge about software process improvement challenges when implementing change-driven development, and increase the body of literature on longitudinal action research.

Preface – the beginning

I was introduced to teamwork, continuous improvement, learning, and reflection when I played handball in one of Norway's best clubs. Handball (also known as team handball or European handball) is a team sport in which two teams of seven players each (six outfield players and a goalkeeper) pass a ball to throw it into the goal of the other team. The game is quite fast, physical, and it involves body contact as the defenders try to stop the attackers from approaching the goal.

When I started playing, my father was the coach, and he told us that for a team to perform, all players need to master all positions on the field. When I was 16, we got a new coach. It was his first job as a coach and he demanded discipline – you always showed up on time, you were well prepared and motivated to do your best. After a few weeks, some of the most experienced and talented players left because they did not want to comply with this new way of running the team. In addition, some started to question whether the new coach was the right man for the job. However, the results speak for themselves; we started in the third division and after three years we were in the top division.

Planning and evaluation was an important part of the philosophy of the new coach. Each month started with setting a plan, each practice usually started with presenting its aim, and ended with a short evaluation: what was good and what was not so good. During training, the coach sometimes interrupted an exercise to let players know whether their performance was excellent, poor, or needed improvement. In addition to reflection as a team, I had my own training diary for personal reflection. We practiced 10 to15 hours a week, mostly together as a team. 10-20% of the time was allocated to individual training, focusing on each player's needs.

Our coach has now been working for various teams for more than 25 years, and he is still focusing on improving his skills. Important aspects of his philosophy are joint responsibility, involvement, and commitment to goals. In 2006 he was voted the coach of the year in Norway, and in 2010 he won the world championship for the women's national team.

I learned five things about teamwork through my handball experiences. You need hard work, team discipline, involvement, frequent reflection, and it takes a long time to build a cohesive team. I have been a coach myself, and I came to the conclusion that there is no single best practice to achieve success.

Acknowledgements

This thesis is the tangible result of work in which I have depended on the help, support, and inspiration of many people. First, I wish to thank my father and Thorir Heirgerisson for introducing me to the field of teamwork. Then I wish to thank my fellow researchers at SINTEF. SINTEF is a unique institution, and I am grateful for the opportunity to work in such an excellent environment and professional atmosphere. Special thanks go to Torgeir Dingsøyr and Tore Dybå for participating in several of the studies reported in this thesis and for discussing both research method and results of these studies. Also, I wish to thank Geir Kjetil Hanssen, Tor Erlend Fægri, and Børge Haugset for fruitful discussions and valuable feedback. I would also like to thank my manager, Eldfrid Øfsti Øvstedal, for always supporting my work.

The starting point of this thesis is the research I conducted together with Tore Dybå in 2004. Tore introduced me to the combination of software process improvement, organizational science, and socio-technical theory, which strongly influenced my research perspective. Because the work with this thesis has been done without a supervisor, discussing my findings with Tore was of outmost importance to me.

I met a number of researchers outside of my research group who inspired me in many different ways. During my stay at UNSW Sydney (Australia) I had time to get to know new theories and to write some of the most important papers included in this thesis; therefore I would like to thank Aybuke Aurum for being my host. I would also like to thank Darja Smite and Pekka Abrahamsson for introducing me to new perspectives in the field of computer science and process improvement, and for asking important questions about my studies and my results. I also thank Maria Line for all the inspiring discussions at the coffee bar.

My thanks also go to the three companies where the studies were conducted. The close and long-term cooperation with them forms the basis of my work. I thank the Research Council of Norway for funding the projects I have been working on, and Chris Wright and Ewa Huebner for proofreading this thesis.

Finally, I offer heartfelt thanks to my family for their care, support, and encouragement. Special thanks go to my wife Ela who has motivated me, given me valuable feedback, and has been my coach in the field of organizational learning. Without the ability to discuss my work with her, this thesis would never have been finished. Also I wish to thank my parents in law, Anne-Marie and Kjell, who has supported my family during

the vacations I have been working on my thesis. My children has put up with a great deal due to my overtime work and mental absence, so I owe a special debt of gratitude to Emrik, Ask, and Helma.

Trondheim, May 2011
Nils Brede Moe

Content

Abstract **i**

Preface - the beginning **iii**

Acknowledgements **iv**

Content **vi**

List of Figures **ix**

List of Tables **x**

Abbreviations **xi**

List of Papers **xii**

1 Introduction **1**

1.1 Background for the research 1
1.2 Research gaps 3
1.3 Research problem and questions 5
1.4 Research context 5
1.4.1 SPIKE 6
1.4.2 EVISOFT 6
1.4.3 EVIDENSE 6
1.5 Research approach 7
1.6 Research design 7
1.7 Claimed contributions 9
1.7.1 Contribution of thesis 9
1.7.2 Included papers 10
1.8 Thesis structure 17

2 Background **19**

2.1 Software development 19
2.1.1 Plan-driven development 20
2.1.2 Change-driven development 21
2.1.3 From plan- to change-driven development 22
2.2 Software Process Improvement 23
2.2.1 Challenges of software development and SPI 24
2.2.2 Norm-driven approach to SPI 25
2.2.3 Tailoring best practice 27

2.2.4 Change-driven approach to SPI 27
2.3 SPI and organizational issues 29
2.3.1 Software development teams 30
2.3.2 Self-managing teams 31
2.3.3 Coordinating the development process 33
2.3.4 Organizational learning and SPI 35
2.3.5 SPI and participation 37

3 Research method and design 39

3.1 Action research 40
3.2 Case study research 43
3.3 Research setting 44
3.3.1 NorSoft 45
3.3.2 EastSoft 45
3.3.3 MidSoft 46
3.3.4 Process Guide 47
3.3.5 Scrum 48
3.4 Research methods applied in the plan-driven period 49
3.5 Research methods applied in the change-driven period 50
3.6 Studies and data analysis 53

4 Results 57

4.1 SPI in plan-driven companies 57
4.1.1 Key Finding 1: Best practice mainly supports project management 58
4.1.2 Key Finding 2: Involvement affects how best practice is adopted 59
4.1.3 Key Finding 3: Individual experts approach is a simple strategy to manage projects 60
4.1.4 Key Finding 4: Post-project reflection is an important learning strategy 61
4.2 SPI in change-driven companies 61
4.2.1 Key Finding 5: Short iterations make project management easier 62
4.2.2 Key Finding 6: Change-driven development encourages frequent problem reporting 65
4.2.3 Key Finding 7: Long-term quality is in conflict with short-term progress 66
4.3 Key SPI challenges implementing change-driven development 67
4.3.1 Key Finding 8: Specialization hinders self-management 67
4.3.2 Key Finding 9: Process related problems are difficult to solve 69
4.3.3 Key Finding 10: There are major organizational barriers to self-management 70

5 Discussion 73

5.1 Q1: What characterizes SPI in plan-driven companies? 73
5.1.1 Participative bottom-up approach to creating company's best practice 73
5.1.2 Long cycles of single-loop learning and post-project reflection 74
5.1.3 Project management support focus and high individual autonomy 75
5.2 Q2: What characterizes SPI in change-driven companies? 76
5.2.1 Short cycles of single-loop learning to improve current project 76
5.2.2 Supporting the whole team and not only the project management 77

5.3 Q3: What are the key SPI challenges when implementing change-driven development? 79
5.3.1 Creating conditions for self-management 79
5.3.2 Learning to learn 81
5.3.3 Changing the organization 82
5.4 Implications for research and practice 83
5.5 Limitations 85
5.6 Recommendations for future work 86

6 Conclusion 87

References 91

List of Figures

Figure 1 Study design 8
Figure 2 Single and double-loop learning (Argyris and Schön 1996)....36
Figure 3 Study design and papers produced in the plan- and change-driven period 39
Figure 4 The CAR process (Davison, Martinsons et al. 2004)....42
Figure 5 Conceptual model for the survey tested in study 1 49
Figure 6 Example of the coding process in study 4 55
Figure 7 Defects found and closed in the plan-driven phase....63
Figure 8 Defects found and closed in the Scrum phase 64
Figure 9 Framework for organizational change (Adler and Shenhar 1990) 82

List of table

Table 1 Studies performed in the thesis and resulting papers..8
Table 2 Detailed contribution of thesis: how the key findings and the research questions (RQ) are related 10
Table 3 Papers included in the thesis and the companies involved in each study.........10
Table 4 Main differences between change-driven and plan-driven development (Nerur, Mahapatra and Mangalaraj 2005). 23
Table 5 Underlying differences between traditional and agile software development and SPI (Salo and Abrahamsson 2007)..28
Table 6 Characteristics of the CMM and Agile mindset in SPI (Aaen, Börjesson and Mathiassen 2005)..29
Table 7 Teams and data collection sources..51
Table 8 The use of Klein and Myers' principles in this field research............................52
Table 9 Relationship of various papers to the key findings and the research questions (RQ)...57
Table 10 Teams introducing agile approach 62

Abbreviations

ASD	Agile Software Development
CMM	Capability Maturity Model
CMMI	CMM Integration
EPG	Electronic Process Guide
GUI	Graphical User Interface
ICT	Information and Communication Technology
ISO	International Organization for Standardization
PMA	Post-mortem analysis
QA	Quality Assurance
RUP	Rational Unified Process
RQ	Research Question
SE	Software Engineering
SEI	Software Engineering Institute
SINTEF	The Foundation for Scientific and Industrial Research at the Norwegian Institute of Technology
SME	Small and Medium-sized Enterprises
SPI	Software Process Improvement
SPICE	Software Process Improvement and Capability dEtermination
STS	Socio-Technical System
SW	Software
SW-CMM	Capability Maturity Model for Software
TQM	Total Quality Management
UNSW	University of New South Wales

List of papers

No.	Paper
P1	An Empirical Investigation on Factors Affecting Software Developer Acceptance and Utilization of Electronic Process Guides, Metrics (Dybå, Moe and Mikkelsen 2004)
P2	Measuring Software Methodology Usage: Challenges of Conceptualization and Operationalization, Isese (Dybå, Moe and Arisholm 2005)
P3	The Use of an Electronic Process Guide in a Medium-sized Software Development Company, SPIP (Moe and Dybå 2006)
P4	Improving by involving: a case study in a small software company EuroSPI (Moe and Dybå 2006)
P5	The Impact of Employee Participation on the Use of an Electronic Process Guide: A Longitudinal Case Study, TSE (Dingsoyr and Moe 2008)
P6	Understanding Self-organizing Teams in Agile Software Development, Aswec (Moe, Dingsøyr and Dybå 2008)
P7	Understanding Decision-Making in Agile Software Development: A Case Study, Euromicro (Moe and Aurum 2008)
P8	Putting Agile Teamwork to the Test – An Preliminary Instrument for Empirically Assessing and Improving Agile Software Development XP (Moe, Dingsøyr and Røyrvik 2009)
P9	Understanding Shared Leadership in Agile Development: A Case Study, HICCS (Moe, Dingsøyr and Kvangardsnes 2009)
P10	Overcoming Barriers to Self-Management in Software Teams, IEEESW (Moe, Dingsøyr and Dybå 2009)
P11	Transition from a Plan-Driven Process to Scrum – A Longitudinal Case Study on Software Quality, ESEM (Li, Moe and Dybå 2010)
P12	A teamwork model for understanding an agile team: A case study of a Scrum project, IST (Moe, Dingsøyr and Dybå 2010)

The full citation for each paper, its relevance to this thesis, and its contribution is provided at the end of the first chapter. Hereafter, the papers will be referred to by their number.

PART I – Summary of studies

1 Introduction

1.1 Background for the research

Information technology play a significant role in all areas of modern human activity, including science, engineering, business, government, entertainment, education, energy, defense, health, and medicine. In the same way as electricity and the combustion engine made industrial society possible, information technology enables the global information society. The knowledge workers, the creative industries, and the service industries cannot exist without information systems.

As information technology's role in the modern economy grows in importance, society makes exponentially greater demands on the diversity and quality of the software being produced. Time-to-market can spell the difference between a successful product release and bankruptcy. Software process improvement (SPI) has become the primary approach to improving software quality and reliability, employee and client satisfaction, and return on investment (Mathiassen, Ngwenyama and Aaen 2005). Therefore, SPI is becoming increasingly important.

The environment in which software is designed and created is also changing. Software systems are becoming larger and more complex, commercial off-the-shelf components are playing increasingly significant role, and the already rapid pace of requirement changes is accelerating. When software professionals refer to today's software development as solving the "wicked problems" (Nerur and Balijepally 2007), they are not simply considering technical issues. Rather, software development today forces developers to interact with and consider the viewpoints of a wide variety of stakeholders, many of whom have conflicting views on the desirability of the software features and its functionality. Today's turbulent environment requires that managers continuously coordinate and adjust priorities of diverse change initiatives. Agile SPI approaches provide an answer to this development (Mathiassen, Ngwenyama and Aaen 2005).

The traditional software development environment is characterized by the product-line approach using a standardized, controllable, and predictable software engineering process (Dybå 2000). The traditional way of developing software is to involve extensive planning, codified processes, and rigorous reuse to make development an efficient and predictable activity (Boehm 2002). Increased complexity is addressed by relying on foresight to develop and impose architectures, which can often moderate adverse impact of change on the system (Boehm and Turner 2003). This view of developing software is also known as a plan-driven approach, and it is usually guided by a life cycle model such as the waterfall model or the spiral model, with the focus on the quality of the software artifacts and the predictability of their processes (ibid.). This engineering approach favors explicitly defined processes, which can be standardized both within and across organizations (Lycett, Macredie et al. 2003).

From the mid 1990s onwards, agile software development principles and methodologies have been increasingly challenging the traditional view of software development. In contrast to the plan-driven perspective, agile processes address the challenge of an unpredictable world by relying on ‘‘people and their creativity rather than on processes” (Dybå 2000; Nerur, Mahapatra and Mangalaraj 2005). The goal of optimization of design is being replaced by flexibility and responsiveness (Nerur and Balijepally 2007). Agile development is ‘‘about feedback and change” (Williams and Cockburn 2003). Erickson et al. (2005) define agility as follows:

> Agility means to strip away as much of the heaviness, commonly associated with the traditional software development methodologies, as possible to promote quick response to changing environments, changes in user requirements, accelerated project deadlines and the like. (p. 89)

The inherent differences between the plan-driven and the agile approaches require new SPI mechanisms to fit the context of agile software development. Software process improvement has its roots in general improvement strategies like total quality management, which has been tailored to software engineering, for example the Quality Improvement Paradigm (QIP) (Basili 1989), and in efforts on standardization, for example the ISO 9001 (ISO 2000) and the Software Engineering Institute’s Capability Maturity Model Integration (CMMI) (SEI 2002). Classical SPI techniques like CMMI relate software processes, standardization, software metrics, and process improvement (Hansen, Rose and Tjornehoj 2004). This focus on software processes is based on the premises that:

- The process of producing and evolving software products can be defined, managed, measured, and progressively improved.
- The quality of a software product is largely governed by the quality of the development process (Humphrey, Kitson and Kasse 1989).

This approach prescribes norms for how individuals, teams, and organizations should operate, and for how processes should be standardized and improved (Hansen, Rose and Tjornehoj 2004).

When describing traditional or plan-driven development this thesis relies on the prescriptive and norm-driven approach to SPI as described by Hansen (Hansen, Rose and Tjornehoj 2004). The prescriptive approach is more concerned with how the strategies should be formulated than with how they are actually implemented. The norm-driven approach is based on an underlying normative model of software process improvement (Aaen, Arent et al. 2001). The motivation for the research leading to this thesis was to fill the gaps in the field of SPI, which tends to be dominated by one approach (the capability maturity model (CMM) which is norm-driven) and heavily biased towards prescriptive methodologies (Hansen, Rose and Tjornehoj 2004).

1.2 Research gaps

There are several fundamental differences between traditional and agile software development from the perspective of SPI. First, while SPI in the plan-driven approach prescribes norms for how individuals, teams, and organization should operate, agile software development addresses the improvement and management of software development practices within individual teams (Lycett, Macredie et al. 2003). In agile development processes are not products, but rather practices which evolve dynamically within the team as it adapts to the particular circumstances (Aaen 2008). The empowered self-managing team should base work coordination on face-to-face communication, and is responsible for finding the best way of developing software through frequent reflection. This has been stated in three of the twelve principles of the Agile Manifesto[1]:

- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- At regular intervals the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
- The best architectures, requirements, and designs emerge from self-organizing teams.

Another difference is that plan-driven methods, such as the waterfall model, usually adopt a top-down approach to improving the software development process (Salo and Abrahamsson 2007), while the agile development represents a bottom-up approach. Furthermore, SPI in plan-driven development often emphasizes the continuous improvement of the organizational software processes for future projects, while the principles of agile software development focus on iterative adaption and improvement in the ongoing projects. Short development cycles provide continuous and rapid loops of iterative learning, to enhance the processes and to test the improvements.

[1] http://agilemanifesto.org
[2] SPI based on Knowledge and Experience

Because adopting agile approach entails changing many aspects of the organization including its structure, culture, and management practice (Nerur, Mahapatra and Mangalaraj 2005), it requires significant organizational changes which take a long time to implement (Pyzdek 1992; Vinekar, Slinkman and Nerur 2006). Longitudinal studies on adoption of agile practices are needed to understand the differences between SPI in plan-driven and change-driven environments. In an extensive review of literature on agile software development Dybå and Dingsøyr (2008) found seven studies addressing how agile development methods are introduced and adopted in companies. However, none of the studies on adoption focused on SPI. In addition, the conclusion of the study of Aaen et al. (Aaen, Börjesson and Mathiassen 2005) is that there is no recognized SPI model supporting the agile approach.

There were however a few studies focusing on software process improvement and the introduction of agile development. One such study is the work by Salo and Abrahamsson (2007). They argue that because of the fundamental differences between traditional and agile software development, there is a need to define new SPI mechanisms for agile software development. Salo and Abrahamsson suggest an iterative improvement process for conducting SPI within agile software development teams. However, they only studied student projects lasting from eight to eleven weeks. A study by Qumer and Henderson-Sellers (2008) suggests a framework which can be used to create, modify, and tailor situation-specific agile software processes. The model includes among others an agility measurement model and an agile adoption and improvement model. However, this framework was only explored in a limited way; the first case study only applied a few agile methods for a short period, and the second case study only involved two developers.

The field of software engineering is largely influenced by and based on the needs of large organizations like the US Department of Defense (Fayad, Laitinen and Ward 2000) and by military applications. However, most software companies are small. Even in the USA, 94% of the software industry consist of companies with fewer than 50 employees (ibid.). Because problems of big organizations are in many ways different from the problems of small organizations, most current research in software engineering is not automatically useful for resolving problems faced by small organizations. Therefore, it is necessary to study small and medium-sized companies.

1.3 Research problem and questions

The fundamental differences in SPI between traditional and agile software, and the lack of research in this area, gave rise to the overall problem addressed by this thesis:

How does Software Process Improvement work change with the introduction of agile software development in plan-driven companies?

To narrow the focus of the investigation, the research problem addressed by this thesis can be summarized by the following research questions:

RQ 1: What characterizes SPI in plan-driven companies?

RQ 2: What characterizes SPI in change-driven companies?

RQ 3: What are the key SPI challenges when implementing change-driven development?

Because of the importance of studies on small and medium-sized companies, the answers to these questions in this thesis are based on the study of such companies.

1.4 Research context

This thesis culminates about six years of studies in three small and medium-sized companies, and shows how these companies changed from a plan-driven to a change-driven approach to software development.

The research presented in this thesis has been conducted within the context of three research projects: SPIKE[2], EVISOFT[3] and EVIDENSE[4]. In SPIKE and EVISOFT, the target group was small and medium-sized software companies, research and development departments in large companies, as well as universities, colleges, and consulting companies, which develop and offer SPI services. Action research (Susman and Evered 1978) was the preferred research method, and it involved me helping the organizations in improving their software development processes according to company-specific quality goals. In EVIDENSE the goal was to develop a theoretical

[2] SPI based on Knowledge and Experience
[3] EVidence based Improvement of SOFTware engineering
[4] EVIDENce-based Software Engineering

understanding of the trade-offs between agile methods and plan-based methods in effective software development

1.4.1 SPIKE

SPIKE (SPI based on Knowledge and Experience) was a national SPI program partially funded by the Research Council of Norway from 2003 to 2006. Ten small and medium-sized software companies participated in the program together with the Norwegian University of Science and Technology, the University of Oslo, and SINTEF. SPIKE focused on context dependent methods and guidelines for software process improvement with the aim to find the right method for the specific project. Key areas were incremental and evolutionary methods, object oriented analysis and design, electronic process guides, knowledge management, estimation and project management, and evolutionary projects.

1.4.2 EVISOFT

EVISOFT (EVidence based Improvement of SOFTware engineering) – was a user-directed innovation program funded by the Research Council of Norway (grant 174390/I40), which started in 2006 and ended in 2010. Three research institutions (SINTEF, Norwegian University of Technology, and the University of Oslo), together with ten large and medium-sized Norwegian ICT companies cooperated to find methods and technologies for producing software with the right quality, within a given timeframe, and at the right price. EVISOFT focused on agile and evolutionary methods, model based development, estimation and risk management, and component based development.

1.4.3 EVIDENSE

EVIDENSE (EVIDENce-based Software Engineering) was a strategic internal project at SINTEF ICT funded by Research Council of Norway (grant 181658/I30), which started in 2007 and ended in 2009. The objective of EVIDENSE was to develop a better theoretical and empirically based understanding of the trade-offs between agile methods and plan-based methods in effective software development. EVIDENSE focused on project management, collaborative processes, and team learning. These issues are of the utmost importance for building scientific knowledge on software development as well as for innovation and value creation in the ICT industry.

The action research projects in the companies involved in EVISOFT and SPIKE constitute the main empirical basis of this thesis.

1.5 Research approach

The papers included in this thesis represent contributions stemming from research at three organizations; each paper has different, specific research objectives and hence distinct perspectives for analysis. The overall problem formulation and the associated research questions of this thesis are addressed by synthesizing (Cruzes and Dybå 2010) individual contributions of the studies performed earlier. The synthesis did not alter the original elements, and the aim was to contribute through combined analysis from a more evolved perspective. This thesis relied on interpretive synthesis (Dixon-Woods, Agarwal et al. 2005). Through an interpretive synthesis, the concepts identified in the primary studies were subsumed into a higher order theoretical structure. The main product is not aggregation of data, but of theory. The primary concern has been the development of concepts and of theories integrating these concepts. In the interpretive synthesis the concepts were not specified in advance, but were derived from the combined data reported by the primary studies (Dixon-Woods, Agarwal et al. 2005). The perspective defined by the research problem and the research questions allowed for interpretation of the papers included in this thesis within a new conceptual frame.

In addition to software process improvement in software development, this thesis covers the areas of organizational issues in SPI. There are two reasons for this. First, organizational issues are of great importance in SPI (Børjesson and Mathiassen 2004; Hansen, Rose and Tjornehoj 2004; Dybå 2005; Mathiassen, Ngwenyama and Aaen 2005). Second, introducing agile software development requires the organization to change its culture, strategy, and structure (Vinekar, Slinkman and Nerur 2006). The magnitude of such changes is relatively large, the level of learning required is high, and the time to adjust is long (Adler and Shenhar 1990).

The position taken in this thesis regarding organizational issues is strongly influenced by socio-technical theory (Trist 1981). Its central concept is that organizations are both social and technical systems, and that the core of the software organization is represented through the interface between the technical and human (social) system. Instead of a search for global best practices, this thesis points to the importance of organizational learning as described by (Argyris and Schön 1996).

1.6 Research design

Five studies in three companies were conducted over six years, resulting in twelve publications (Figure 1). EastSoft was located in the southeast of Norway, MidSoft in the middle of Norway and NorSoft in the North of Norway. Studies 1 and 2 were performed in the plan-driven period, where the SPI focus was on the whole development at the department and organizational level. Studies 3 and 4 started in the plan-driven period and continued into the change-driven period. After Scrum training, the processes in projects in study 3 and 4 were tailored to agile methods, and the projects were classified as change-driven development. Study 5 was conducted fully in the change-driven period. A different SPI focus (the organization versus the team) in the two periods called for a different research approach, from studying how developers were using technology to studying how developers worked together. This resulted in two methodologically different phases of the research. An overview of the purpose of each study and the resulting papers is given in Table 1.

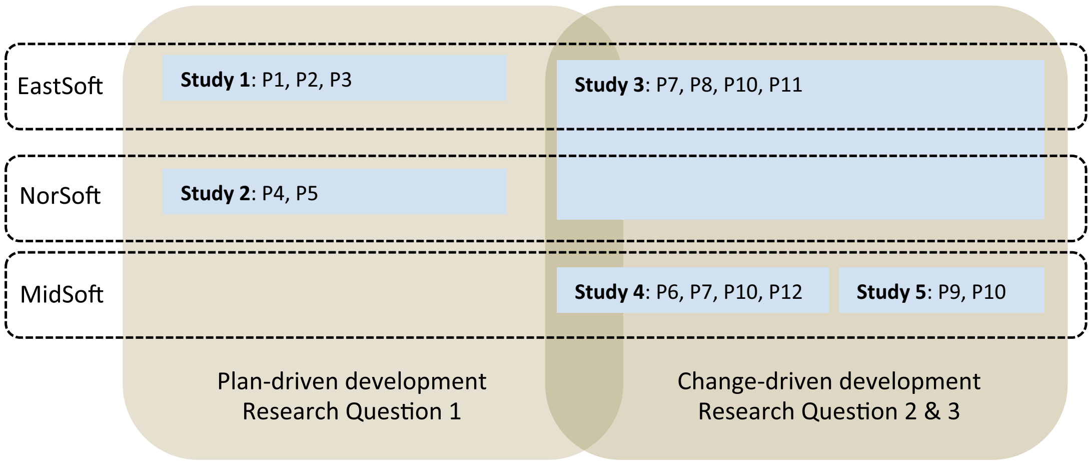

Figure 1 Study design

Table 1 Studies performed in the thesis and resulting papers

Study	Purpose	Paper
Study 1	To understand factors, which influence the usage and acceptance of the electronic process guide (EPG) in the organization by inspecting documents, analyzing usage logs and surveys, and by interviewing the users.	P1, P2, P3
Study 2	To understand the importance and the effect of participation in SPI, with a particular focus on process workshops as a technique for involving participants when creating an EPG. The researchers and the company designed the process workshop, and the effect of the workshop was evaluated by interviews and inspection of access logs.	P4, P5
Study 3	To understand the process of the introduction of change-driven development in the middle of long-term projects. Observations and interviews were the primary source of evidence. The nature of the study made it possible to study defect data from the same project before and after Scrum was introduced	P7, P8, P10, P11
Study 4	To understand the process of the introduction of change-driven development at the beginning of a project. An important part of this study was also to document the change-driven approach. The change-driven perspective was explored by observations, interviews, and inspection of documents.	P6, P7, P10, P12
Study 5	To understand change-driven development by looking from inside of the project. To get a deeper understanding of the phenomena investigated in this study action research included working as a software developer. In this study, in addition to ethnographic observations, interviews were important.	P9, P10

1.7 Claimed contributions

1.7.1 Contribution of thesis

The objective of the thesis is to investigate how Software Process Improvement work change with the introduction of agile software development in small- and medium-sized plan-driven companies. Based on the objective of this study and the answers provided to the research questions posed in this thesis, I claim that the thesis has unique contributions for theory and practice in Software Process Improvement. The main contributions are:

- **Increased awareness of both importance and challenges of improving practice.** SPI has traditionally focused on improving process descriptions and identifying best practice. The overall contribution of this thesis is that it shows empirically that the goal of SPI changes from improving processes to improving practice. However, achieving this goal is challenging when only part of the organization is involved in the change.
- **Deep knowledge about SPI in plan- and change-driven small- and medium-sized companies.** The field of software engineering is largely influenced by and based on the needs of large organizations. Therefore, an important contribution of this thesis is to the body of knowledge on SPI in plan- and change-driven small- and medium-sized companies. This knowledge constitutes answers to research question Q1 and Q2.
- **Knowledge about SPI challenges when implementing change-driven development.** Change-driven development is found to be a strong infrastructure for SPI. However, I found several key SPI challenges implementing change driven development. These key challenges answer research question Q3.
- **Methodological contribution: longitudinal action research.** Despite the relevance of action research in the software industry, the method is seldom used in the field of software engineering (SE) and information systems (IS). Therefore an important contribution of this thesis is to increase the body of literature on longitudinal action research studies.

Moreover, twelve papers have been included in this thesis and from the papers, and through a synthesis ten Key findings emerged (Table 2). These key-findings also contribute to the body of knowledge in SPI, however they are on a more detailed level than the contributions mentioned above.

Table 2 Detailed contribution of thesis: how the key findings and the research questions (RQ) are related

No	RQ	Key finding	Papers
1	1	Best practice mainly supports project management.	2, 3, 5
2	1	Involvement affects how best practice is adopted.	3, 4, 5
3	1	Individual experts approach is a simple strategy to manage projects.	6, 7
4	1	Post-project reflection is an important learning strategy.	1, 4
5	2	Short iterations make project management easier.	11
6	2	Change-driven development encourages frequent problem reporting.	7, 10, 12
7	2	Long-term quality is in conflict with short-term progress.	7, 9, 10, 11, 12
8	3	Specialization hinders self-management.	6, 8, 9, 10, 12
9	3	Process related problems are difficult to solve.	6, 8, 10, 12
10	3	There are major organizational barriers to self-management.	6, 7, 9, 10

1.7.2 Included papers

Twelve published papers are included in this thesis.
Table 3 shows the papers and companies in which the studies were performed.

Table 3 Papers included in the thesis and the companies involved in each study

No	Paper	Company
P1	An Empirical Investigation on Factors Affecting Software Developer Acceptance and Utilization of Electronic Process Guides, Metrics (Dybå, Moe and Mikkelsen 2004)	EastSoft
P2	Measuring Software Methodology Usage: Challenges of Conceptualization and Operationalization, Isese (Dybå, Moe and Arisholm 2005)	EastSoft
P3	The Use of an Electronic Process Guide in a Medium-sized Software Development Company, SPIP (Moe and Dybå 2006)	EastSoft
P4	Improving by involving: a case study in a small software company EuroSPI (Moe and Dybå 2006)	NorSoft
P5	The Impact of Employee Participation on the Use of an Electronic Process Guide: A Longitudinal Case Study, TSE (Dingsoyr and Moe 2008)	NorSoft
P6	Understanding Self-organizing Teams in Agile Software Development, Aswec (Moe, Dingsøyr and Dybå 2008)	MidSoft
P7	Understanding Decision-Making in Agile Software Development: A Case Study, Euromicro (Moe and Aurum 2008)	MidSoft EastSoft
P8	Putting Agile Teamwork to the Test – An Preliminary Instrument for Empirically Assessing and Improving Agile Software Development XP (Moe, Dingsøyr and Røyrvik 2009)	NorSoft, EastSoft
P9	Understanding Shared Leadership in Agile Development: A Case Study, HICCS (Moe, Dingsøyr and Kvangardsnes 2009)	MidSoft
P10	Overcoming Barriers to Self-Management in Software Teams, IEEESW (Moe, Dingsøyr and Dybå 2009)	MidSoft, NorSoft, EastSoft
P11	Transition from a Plan-Driven Process to Scrum – A Longitudinal Case Study on Software Quality, ESEM (Li, Moe and Dybå 2010)	EastSoft
P12	A teamwork model for understanding an agile team: A case study of a Scrum project, IST (Moe, Dingsøyr and Dybå 2010)	MidSoft

The relevance of the papers to this thesis and my contribution to each paper are described next.

P1: Dybå, T., **Moe, N. B.** and Mikkelsen, E. M. (2004). An Empirical Investigation on Factors Affecting Software Developer Acceptance and Utilization of Electronic Process Guides. Proceedings of the International Software Metrics Symposium (METRICS), Chicago, Illinois, USA, 220–231.

Relevance to this thesis: The objective of this paper was to investigate the factors affecting software developer acceptance and utilization of electronic process guides (EPGs) - a tool for describing and communicating work processes in software organizations. The paper identified factors that have a significant and positive effect on software developer acceptance and utilization of EPGs. The results showed that perceived usefulness is the fundamental driver in explaining current system usage and future use intentions. Furthermore, perceived compatibility, perceived ease of use, and organizational support were the key determinants of perceived usefulness. Organizational support included Post-project support, which comprised facilitation of post-project reviews and evaluation of the infusion to gather lessons learned regarding the deployment of the EPG as well as on the use of the EPG in the projects. The paper contributes to key finding 4.

My contribution: Tore Dybå was the one responsible for the study design and analysis. I was responsible for the data collection, and Edda Mikkelsen helped me distributing the 120 questionnaires. 97 usable responses were received after using a lot of effort chasing the missing respondents, resulting in a good overall response rate of 81%. I was also in charge of the literature review on process guides. The discussions and writing was done collaboratively.

P2: Dybå, T., **Moe, N. B.** and Arisholm, E. (2005). Measuring Software Methodology Usage: Challenges of Conceptualization and Operationalization. Fourth International Symposium on Empirical Software Engineering (ISESE), Noosa Heads, Australia, IEEE Computer Society, 447 - 457.

Relevance to this thesis: The purpose of this paper was to better understand EPG usage in the plan-driven period by comparing subjective (self-reported usage) and objective (number of hits on the EPG, and documents produced by the projects) operationalization of usage. All project documentation for all projects was studied to investigate if the self-reported usage level and usage-logs corresponded to the actual use-level. There was a difference between the subjective self-reported methodology usage construct on the one hand, and the objective template usage and computer-recorded usage constructs on the other hand. This paper confirmed that projects produced deliverables according to the EPG, which demonstrated that processes and checklists supporting management activities were used. The paper contributes to key finding 1.

My contribution: Tore Dybå was responsible for the study design. I was responsible collecting the data. Together with Tore Dybå, I inspected and analyze about 1,000 documents in 23 projects in order to measure the ratio of actual template usage. I was also responsible for analyzing the number of server hits on the EPG template pages. Tore Dybå and Erik Arisholm were responsible for the rest of the analysis. Discussing the results of the analysis and writing of the paper was done collaboratively.

P3: **Moe, N. B.** and Dybå, T. (2006). The use of an Electronic Process Guide in a medium sized Software Development Company. Software Process Improvement and Practice 11(1): 21-34.

Relevance to this thesis: This paper describes the findings that emerged from a part of Study 1, where we interviewed 19 developers and project managers. The analysis was guided by grounded theory. It was found that the EPG provides mostly management support, and no or little support for the developers. Also, while the goal was to involve the users in developing the EPG, the interviewees did not report being involved. Also, the use-level was found to be low. The paper contributes to key finding 1 and 2.

My contribution: As the principal author I was in charge of the study design. I conducted all the interviews, which was the data source for this article. I was also responsible for analyzing the interview material, by coding and re-coding it in NVivo. Discussing the results and writing of the paper was done collaboratively.

P4: **Moe, N. B.** and Dyba, T. (2006). Improving by involving: a case study in a small software company. EuroSPI 2006, Joensuu, Finland, 158 – 169.

Relevance to this thesis: The paper describes how long-term participation can be realized in various SPI initiatives using several participation techniques like search conferences, survey feedback, autonomous work groups, quality circles, and learning meetings. The paper describes how post project learning in the form of postmortem review was used to collect experience after the project was finished, and further the use of quality circles when organizing process workshops to create the electronic process guide. The paper report from a part of Study 2 and contributes to key finding 2 and 4

My contribution: As the principal author I was in charge of the study design. I was the only author partaking in introducing and assisting the company in four of the five participative techniques introduced: search conferences, autonomous work groups, quality circles, and learning meetings. Tore Dybå was the one responsible for the last technique (survey feedback). The literature review on participation, discussing the results of the analysis, and writing of the paper was done collaboratively.

P5: Dingsøyr, T. and **Moe, N. B.** (2008). The Impact of Employee Participation on the Use of an Electronic Process Guide: A Longitudinal Case Study. IEEE Trans. Softw. Eng. 34(2): 212-225.
Relevance to this thesis: The paper studied the long-term effect on participation. Through collecting data from three rounds of interviews and 19 months of usage logs, we found that employees who were involved in developing the EPG showed a higher degree of usage, used a larger number of functions, and expressed more advantages and disadvantages than those not involved. Also, the study confirms that the EPG supports project management. The paper contributes to key finding 1 and 2
My contribution: I participated in the whole process, from planning of the study to analysis, and reporting. We interviewed half of the developers each and I was the one responsible for collecting and analyzing the access logs on the electronic process guide. The first author had the overall responsibility of the study, and made the final decisions on form.

P6: **Moe, N. B.**, Dingsøyr, T. and Dybå, T. (2008). Understanding Self-Organizing Teams in Agile Software Development. 19th Australian Conference on Software Engineering, 76-85.
Relevance to this thesis: The aim of the paper was to study autonomy in agile teams. Through investigating a team applying Scrum, we found that autonomy exist on the individual and team level. On the team level the autonomy is both internal (how the team-members coordinate work and make decisions) and external (the influence of management and other individuals outside the team on the team's activities). Understanding the different levels of autonomy is important for understanding how to create the self-managing team. The most important barrier to self-management was highly specialized skills of the developers and the corresponding division of work. This paper contributes to key finding 3, 8, 9 and 10.
My contribution: As the principal author I was in charge of the study design. The second author and I conducted all the interviews and observations. I was also responsible for analysing the qualitative material, by coding and re-coding

it in NVivo. Discussing the results and writing of the paper was done collaboratively.

P7: **Moe, N. B.** and Aurum, A. (2008). Understanding Decision-Making in Agile Software Development: A Case-study. Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference, Parma, Italy, 216-223.
Relevance to this thesis: The paper describes the importance of decision-making in agile software development. Decision-making is important in software development and SPI because it affects how problems are solved. A challenge with introducing agile software development is changing the way decisions are made. We found that a prerequisite for introducing Scrum is the alignment of decisions on all levels in the organization. In addition, specialization can be a barrier for the decision-making process on the operational level because it often results in a decentralized decision-making process. Also we found that people are left out of important decisions. The paper contributes to key finding: 3, 6, 7 and 10.
My contribution: As the principal author I was in charge of the study design, collecting and analyzing the data. The second author was responsible for the literature review on decision-making. Discussing the results of the analysis and writing of the paper was done collaboratively.

P8: **Moe, N. B.**, Dingsøyr, T. and Røyrvik, E. A. (2009). Putting Agile Teamwork to the Test – An Preliminary Instrument for Empirically Assessing and Improving Agile Software Development. 10th International Conference on Agile Processes in Software Engineering and Extreme Porgramming, Sardinia, Italy.
Relevance to this thesis: The interviews in this paper showed the effect of specialization on how work is coordinated in the project. Also, the study shows that when problems are not handled, team-members stop reporting them. This article identifies the key concerns and characteristics of teamwork in change driven development, and presents them along five dimensions that must be addressed when improving teamwork in agile software development. The dimensions are shared leadership, team orientation, redundancy, learning and autonomy. The paper contributes to key finding 8 and 9.
My contribution: As the principal author I was in charge of the study design. I did the interviews in two of the three projects in this article. I also lead the work on creating the team radar tool. Discussing the results and writing of the paper was done collaboratively.

P9: **Moe, N. B.**, Dingsøyr, T. and Kvangardsnes, Ø. (2009). Understanding Shared Leadership in Agile Development: A Case Study. Hawaii International Conference on System Sciences, Hawaii, 1-10.

Relevance to this thesis: This aim of the paper is to understand the concept of shared leadership, because shared leadership is one of the fundaments of change-driven development and self-managing teams. In this paper I participated as a developer in the studied project. From looking from inside, this approach gave a new and better understanding of the improvement work, shared leadership, importance of single- and double loop learning and the problem with developers working on several projects. The paper contributes to key finding 7, 8, and 10.

My contribution: As the principal author I was in charge of the study design. Øyvind Kvangardsnes (master student) and I conducted ethnographic observations and interviews from April 2007 until January 2008. All authors participated in the discussions and writing of the material, however I was responsible for analyzing the qualitative data in Nvivo.

P10: **Moe, N. B.**, Dingsøyr, T. and Dybå, T. (2009). Overcoming Barriers to Self-Management in Software Teams. IEEE Software 26(6): 20-26.

Relevance to this thesis: The aim of the paper is to synthesis the results from five teams doing agile software development in three studies (3, 4 and 5). Self-management emerged as the key higher-order topic. Both team and organizational barriers to self-management were found. Lack of redundancy and conflict between team and individual autonomy were found to be key issues when transforming from traditional command-and-control management to collaborative self-managing teams. The paper contributes to key finding 6, 7, 8 and 9.

My contribution: As the principal author I was in charge of the study design. The second author and I conducted all the interviews and observations in two of the companies. I conducted all the interviews and observations in the last company. I was also responsible for analysing the qualitative material, by coding and re-coding it in NVivo. Discussing the results and writing of the paper was done collaboratively.

P11: Li, J., **Moe, N. B.** and Dybå, T. (2010). Transition from a plan-driven process to Scrum: a longitudinal case study on software quality. Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. Bolzano-Bozen, Italy, ACM: 1-10.

Relevance to this thesis: The aim of the paper was to understand the effect of change driven development on software quality and quality processes through combining qualitative interview and observational data with quantitative defect data. Especially this study contributes to understanding the process of correcting defects and the conflict between the need for short term progress and long term quality and improvement work. From a methodological perspective, the study shows the importance of combining qualitative and quantitative data when understanding how the change-driven perspective affects the quality. The paper became the “best paper award” at ISESE and contributes to key finding 5 and 7.
My contribution: I participated in the planning of the study. I conducted and analysed all the interviews and the observations. Discussing the results of the analysis and writing of the paper was done collaboratively. The first author had the overall responsibility of the study, and made the final decisions on form.

P12: **Moe, N. B.**, Dingsøyr, T. and Dybå, T. (2010). A teamwork model for understanding an agile team: A case study of a Scrum project. Information and Software Technology 52(5): 480-491.
Relevance to this thesis: The self-managing team is responsible for SPI on the project level in change-driven development. This paper report from a part of Study 4. The objective was to provide a better understanding of the nature of self-managing agile teams, as well as the teamwork challenges (on both team and organizational level) that arise when introducing such teams, and finally to relate the findings to current research on teamwork. The paper was first on the list of “the top 25 hottest articles” in Information and Software Technology in April – June 2010, and is still the second hottest article (May 2011). The paper contributes to key finding 6, 7, 8 and 9.
My contribution: As the principal author I was in charge of the study design. The second author and I were both involved in the nine month fieldwork. We interviewed, observed the team, and collected the documentation. I conducted most of the observations. I identified the team-work model used in this article and I was also responsible for analysing the qualitative material, by coding and re-coding it in NVivo. Discussing the results and writing of the paper was done collaboratively.

1.8 Thesis structure

The remainder of this thesis consists of two parts.

PART I – Summary of studies

Chapter	Content
2 – Background	This chapter consists of a short introduction to the background of the research presented in this thesis, which deals with software development and SPI. This is followed by an introduction to SPI and relevant organizational issues, which is important for understanding and analyzing the results. Organizational issues relevant to SPI cover the areas of work coordination, team and self-management, organizational learning, and participation.
3 – Research method and design	This chapter first present the overall research approaches: case studies and action research. The context this research has been conducted in will then be discussed, followed by the description of the methods used in each of the five studies.
4 - Results	The results are organized according to the three research questions. Each question is discussed in a separate section: SPI in plan-driven companies, SPI in change-driven companies, and SPI challenges implementing change-driven development. The key findings presented in this chapter are the results of the synthesis of the contributions made in individual papers using the method described in the method chapter.
5 – Discussion	When answering the research questions, each question will be discussed in terms of SPI, organizational learning, and self-management. Further, the chapter explains the implication for research and practice, limitations and recommendations for future research.
6 – Conclusion	Uses the results and the discussions of the research questions to conclude.

PART II – Included publications

1. Dybå, T., Moe, N. B. and Mikkelsen, E. M. (2004). An Empirical Investigation on Factors Affecting Software Developer Acceptance and Utilization of Electronic Process Guides. Proceedings of the International Software Metrics Symposium (METRICS), Chicago, Illinois, USA, 220–231.
2. Dybå, T., Moe, N. B. and Arisholm, E. (2005). Measuring Software Methodology Usage: Challenges of Conceptualization and Operationalization. Fourth International Symposium on Empirical Software Engineering (ISESE), Noosa Heads, Australia, IEEE Computer Society, 447 - 457.
3. Moe, N. B. and Dybå, T. (2006). The use of an Electronic Process Guide in a medium sized Software Development Company. Software Process Improvement and Practice 11(1), 21-34.

4. Moe, N. B. and Dyba, T. (2006). Improving by involving: a case study in a small software company. EuroSPI 2006, Joensuu, Finland, 158 – 169.
5. Dingsøyr, T. and Moe, N. B. (2008). The Impact of Employee Participation on the Use of an Electronic Process Guide: A Longitudinal Case Study. IEEE Trans. Softw. Eng. 34(2), 212-225.
6. Moe, N. B., Dingsøyr, T. and Dybå, T. (2008). Understanding Self-Organizing Teams in Agile Software Development. 19th Australian Conference on Software Engineering, 76-85.
7. Moe, N. B. and Aurum, A. (2008). Understanding Decision-Making in Agile Software Development: A Case-study. Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference, Parma, Italy, 216-223.
8. Moe, N. B., Dingsøyr, T. and Røyrvik, E. A. (2009). Putting Agile Teamwork to the Test – An Preliminary Instrument for Empirically Assessing and Improving Agile Software Development. 10th International Conference on Agile Processes in Software Engineering and Extreme Programming, Sardinia, Italy, 114-123.
9. Moe, N. B., Dingsøyr, T. and Kvangardsnes, Ø. (2009). Understanding Shared Leadership in Agile Development: A Case Study. Hawaii International Conference on System Sciences, Hawaii, 1-10.
10. Moe, N. B., Dingsøyr, T. and Dybå, T. (2009). Overcoming Barriers to Self-Management in Software Teams. IEEE Software, 26(6), 20-26.
11. Li, J., Moe, N. B. and Dybå, T. (2010). Transition from a plan-driven process to Scrum: a longitudinal case study on software quality. Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. Bolzano-Bozen, Italy, ACM, 1-10.
12. Moe, N. B., Dingsøyr, T. and Dybå, T. (2010). A teamwork model for understanding an agile team: A case study of a Scrum project. Information and Software Technology 52(5), 480-491.

Statement of authorship of joint publications from Erik Arisholm
Statement of authorship of joint publications from Aybuke Aurum
Statement of authorship of joint publications from Torgeir Dingsøyr
Statement of authorship of joint publications from Tore Dybå
Statement of authorship of joint publications from Øyvind Kvangardsnes
Statement of authorship of joint publications from Jingyue Li
Statement of authorship of joint publications from Edda Mikkelsen
Statement of authorship of joint publications from Emil Røyrvik

2 Background

Like manufacturing, software development can change the essence of the product. The goal is to build improved products. However, unlike manufacturing, creating software is development not production. The same objects are not reproduced; each product is different from the last. In addition, software systems are becoming larger and more complex, and software developers are forced to interact with and consider the viewpoints of a wide variety of stakeholders, many of whom have conflicting views on the desirability of the software features and its functionality (Boehm 2006). As a result, problems with software development are common, and SPI is becoming more critical.

This chapter consists of a short introduction to the background of the research presented in this thesis, which deals with software development and SPI. This is followed by an introduction to SPI and relevant organizational issues, which is important for understanding and analyzing the results. Organizational issues relevant to SPI cover the areas of work coordination, team and self-management, organizational learning, and participation.

2.1 Software development

In the history of software development different models and approaches were suggested for coping with the increasing complexity and uncertainty of such development: from the Code-and-fix model in the 1950s via the waterfall model (Royce 1970) to the iterative and incremental spiral model (Boehm 1988). Life cycle models such as the waterfall model or the spiral model, focus on the quality of the software artefacts and the predictability of their processes (Boehm and Turner 2003). These models are also known as plan-driven methods. Furthermore, the software development field has largely had an engineering orientation. This is reflected in the adoption of a large number of engineering tools and techniques, e.g. concurrent engineering, prototyping, and computer-aided software engineering (Nambisan and Wilemon 2000).

Agile software development emerged in the mid 1990s (Dybå and Dingsøyr 2008). In 2001, practitioners who proposed many of the agile development methods wrote the Agile Manifesto. While agile software development represents a major departure from traditional, plan-based approaches to software development, the underlying assumptions of agile software development are not novel in any sense, and can be classified as iterative and incremental. Agile software development is also known as change-driven development.

The concepts of plan-driven and change-driven development will be explained first to help understand software development and SPI.

2.1.1 Plan-driven development

Plan-driven methods are usually guided by a life cycle model such as the waterfall model or the spiral model. The focus is on the quality of the software artefacts and the predictability of their processes (Boehm and Turner 2003). The goal is to minimize change in the course of the project through rigorous upfront requirement gathering, analysis, and design; the intent is to attain higher quality results under a controlled schedule (Vinekar, Slinkman and Nerur 2006).

The life cycle model guiding a plan-driven method describes how software is moved through a series of phases from requirements to finished code. The process descriptions specify the tasks to be performed, the desired outcome of each phase, and assign roles (such as systems analyst or programmer) to individuals who will perform these tasks. At every step there is a concern for completeness of documentation followed by verification (Royce 1970; Boehm and Turner 2003). The early version of the waterfall model was introduced in 1970 by Royce (1970), and it has since evolved into a concept consisting of the sequential phases of requirement analysis, design, and development (Larman and Basili 2003).

The field of software development and SPI has been found to be dominated by the plan-driven model defined by the Software Engineering Institute's, capability maturity model (CMM and CMMI) (Hansen, Rose and Tjornehoj 2004). *Software process maturity* is an important concept in CMM and influences SPI work when this approach is applied:

> Software process maturity is the extent to which a specific process is explicitly defined, managed, measured, controlled, and effective. Maturity implies a potential for growth in capability and indicates both the richness of an organization's software process and the consistency with which it is applied in projects throughout the organization. (Paulk, Weber and Chrissis 1999) page 5.

CMMI can be classified as prescriptive and norm-driven (Hansen, Rose and Tjornehoj 2004). The prescriptive approach is concerned with how the processes should be formulated rather than how they are actually implemented. The norm-driven approach, which is a category of the prescriptive approach, is based on an underlying normative model of software process improvement (Aaen, Arent et al. 2001), and describes the norms for how individuals, teams, and organizations should operate, and the norms for how processes should be standardized and improved (Hansen, Rose and Tjornehoj 2004). The dominance of CMM and CMMI in the field of software development is the main reason why in this thesis plan-driven development is understood as prescriptive and norm-driven. This perspective is further explained in the chapter on process improvement.

Another reason for this view of the plan-driven approach is that two of the companies under study relied on the prescriptive and norm-driven approach. One was inspired by CMM and the other based the process descriptions on an extensive system for quality control in accordance with quality routines of the European Cooperation for Space Standardization.

2.1.2 Change-driven development

Methods for agile software development constitute a set of practices for software development created by experienced practitioners (Ågerfalk and Fitzgerald 2006). Agile methodologies are characterized by short iterative cycles of development driven by product features, periods of reflection and introspection, collaborative decision-making, incorporation of rapid feedback and change, and continuous integration of code changes into the system under development (Cockburn and Highsmith 2001; Nerur, Mahapatra and Mangalaraj 2005). Agile development relies on people and their creativity rather than on processes (Cockburn and Highsmith 2001). As opposed to plan-driven methods, agile methods assume that change during the development process is not only inevitable, but also necessary, and aim at achieving innovation through individual initiative (Venkatesh and Davis 2000; Cockburn and Highsmith 2001). The importance of change is the reason why agile software development is also known as change-driven development.

The term agile software development originated from the Agile Manifesto – a statement that expresses a set of basic principles and rules for (agile) software development:

1) Individuals and interactions over processes and tools,
2) Working software over comprehensive documentation,
3) Customer collaboration over contract negotiation and
4) Responding to change over following a plan.

Furthermore, Boehm and Turner (2005) describe agile methods as actively involving users to establish, prioritize, and verify requirements, and as relying on the team's tacit knowledge as opposed to documentation. A truly agile method must be iterative (take several cycles to complete), incremental (not deliver the entire product at once), self-managing (teams determine the best way to handle work), and emergent (processes, principles, and work structures emerge during the project rather than being predetermined) (Boehm and Turner 2005). Agile software development comprises a number of practices and methods (Abrahamsson, Salo et al. 2002; Cohen, Lindvall and Costa 2004; Erickson, Lyytinen and Siau 2005). Among the best known and most widely adopted agile methods are Extreme Programming (XP) (Beck and Andres 2004) and Scrum (Schwaber and Beedle 2001). The companies studied in this thesis all relied on Scrum.

2.1.3 From plan- to change-driven development

The level of adoption of agile systems development is increasing (Dybå and Dingsøyr 2008). Some proponents of agile development claim universal applicability of agile methods, while others believe that it is only suitable in particular situations. Boehm and Turner (2003) argue that there is a pragmatic need to balance stability and agility. Moreover, some industry surveys seem to indicate that most systems development organizations are trying to use both approaches (Venkatesh and Davis 2000). Boehm and Turner (2003) assert that the choice between plan- and change-driven methods for a given project is largely contingent on five factors:

- The size of the systems development project and team
- The consequences of failure (i.e. criticality)
- The degree of dynamism or volatility of the environment
- The competence of personnel
- Compatibility with the prevailing culture

Boehm and Turner point out that plan-driven development is desirable when the requirements are stable and predictable and when the project is large, critical, and complex. They argue that change-driven development, on the other hand, is suitable when there is a high degree of uncertainty and risk in the project, arising from frequently changing requirements and/or the novelty of technology used. Vinekar et al (2006) argue that the client's culture may be the deciding factor in selecting change- or plan-driven methods for a project. First, the client may be uncomfortable with agile systems development's flexible budgets and schedules, and may prefer an upfront contractual obligation to specific features, deadlines, and costs. Second, using a change-driven approach entails significant responsibility on the client's part. Agile methods require that the client identifies and prioritizes features, and collaborates continuously and actively throughout the development. The client may be unwilling to take on this amount of responsibility. Third, the client's organization may dislike constant interruptions by frequent deliveries of partial implementations for user feedback. Similarly, it may also be possible that the client organization has a highly flexible, adaptive culture, and it is uncomfortable with the upfront, explicit, formal, and detailed specification characterizing traditional plan-driven development. Finally, Vinekar et al (2006) argue that new organizational structures are needed to sustain the opposing cultures so that systems development organizations can realize full benefits of both agile and traditional systems development. Table 4 shows a comparison of traditional and agile development.

Table 4 Main differences between change-driven and plan-driven development (Nerur, Mahapatra and Mangalaraj 2005).

	Traditional development	Agile development
Fundamental assumption	Systems are fully specifiable, predictable, and are built through meticulous and extensive planning.	High quality adaptive software is developed by small teams using the principles of continuous design improvement and testing based on rapid feedback and change.
Management style	Command and control.	Leadership and collaboration.
Knowledge management	Explicit.	Tacit.
Communication	Formal.	Informal.
Development model	Life-cycle model (waterfall, spiral, or some variation).	The evolutionary delivery model.
Desired organizational form/ structure	Mechanistic (bureaucratic with high formalization), aimed at large organizations.	Organic (flexible and participative, encouraging cooperative social action), aimed at small and medium-sized organizations.
Quality control	Extensive planning and strict control. Late, exhaustive testing.	Continuous control of requirements, design, and solutions. Continuous testing.

2.2 Software Process Improvement

In software development there is a long tradition of work on software processes (Conradi and Fuggetta 2002; Mathiassen, Ngwenyama and Aaen 2005). Software process improvement is also one of the most widely used approaches in innovative software organizations (Aaen, Börjesson and Mathiassen 2005). Process improvement is about making things better – as opposed to fire fighting or handling crises. It is a way to look at how software developers can do their work better. If software developers only concentrate on solving a problem or correcting a fault, they risk not finding the underlying causes. In the worst case, their actions can make things worse. In addition to identifying problems, the result of SPI should be to identify underlying causes of the problem, define, implement, and evaluate the results of the actions, and finally to carry out possible changes in the rest of the organization. When engaging in process improvement, the goal is to learn about what happened in a process, and to use that knowledge to improve the process as well as the resulting services and products. The improvement work need to be continuous (Aaen, Arent et al. 2001)

While it is easy to argue the importance of SPI, the literature acknowledges that SPI implementation faces various challenges (Aaen, Börjesson and Mathiassen 2005; Mathiassen, Ngwenyama and Aaen 2005). One challenge is related to the importance of organizational issues in SPI (Børjesson and Mathiassen 2004; Hansen, Rose and Tjornehoj 2004; Dybå 2005; Mathiassen, Ngwenyama and Aaen 2005), which means that the whole organization needs to participate in and support the SPI work - from top management to project teams and developers.

This thesis will examine software process improvement in plan-driven companies introducing change-driven development. Because SPI is about making the development process better, it is important to understand the challenges in software development. After explaining these challenges, the norm-driven approach to SPI will be discussed (Hansen, Rose and Tjornehoj 2004), including how this approach can be implemented and tailored to an organization. The change-driven SPI will then be elaborated on.

2.2.1 Challenges of software development and SPI

All software processes are expected to deliver a quality product on schedule and on budget in order to achieve client satisfaction and thereby to ensure long-term profitability for the software organization. Moreover, these fundamental characteristics are important to both the clients and the software organization, and they are, therefore, important for understanding and definition of SPI success. This is also clear in Krasner’s (1999) model of the challenges in software development projects, which focuses on the dynamic relationships between software processes and three outcome factors: cost, schedule, and quality.

However, delivering a product on time, on schedule, and with the right quality seldom happens. The main reason is that software development experiences problems with breakdowns, coordination, and communication (Walz, Elam and Curtis 1993; Kraut and Streeter 1995; Barthelmess 2003), and the challenges are both on the project/product level and on the organizational level. Kraut and Streeter (1995) argue that the challenges of software development are caused by the following characteristics:

- *Scale*. Most software systems are large, and this often results in specialization and division of labor. This in turn leads to compartmentalization of interdependent actors, which limits people’s opportunities and eagerness to share information. It also limits people's breadth of experience, leading to errors, narrowness, and insufficient opportunity for comparing knowledge.

- *Uncertainty*. Often the system is one-of-a-kind, and its specification changes because of the changes in the external world. These changes are inevitable because it is often only by using the software that the client and the end-user understand its capabilities and limitations. In addition, limited domain knowledge among the developers, division of labor, and the extremely complex process of translating user needs into requirements, increase the uncertainty.
- *Interdependence*. Software requires precise integration of its components. Poor coordination between subgroups producing software modules can lead to failure in integrating the modules to create the final product.
- *Informal communication*. Formal communication is useful for coordinating routine transactions, like written specification documents and tracking program errors. However, formal communication often fails in the face of uncertainty, which typifies much of the software work. Informal, interpersonal communication is the primary way information flows in a development organization, but most attention has been on formalizing communication among specialists. In addition, informal communication is a challenge in larger projects because it is impossible for everyone to talk to everyone else. Informal communication is often too imprecise to work well and it is usually not suitable as a record of the information exchanged.

In addition to facing the challenges described by Kraut and Streeter (1995), software developers often need to consider the viewpoints of a wide variety of stakeholders, many of whom have conflicting views on the desirability of the software features and its functionality. Furthermore, software systems are becoming larger and the development process is often distributed. As a consequence, software development is becoming even more complex. This complexity is the reason why today's software development is often referred to as solving the "wicked problems" (Nerur and Balijepally 2007). Because software development is becoming more complex, so is the improvement work independently of the SPI approach.

2.2.2 Norm-driven approach to SPI

By reviewing 322 SPI papers, Hansen et al (2004) categorized SPI using a simple classification: whether the primary goal is prescriptive (to tell SPI professionals what to do), descriptive (to report actual instances of SPI programs in software organizations), or reflective (theoretically analytical). The field was found to be dominated by the capability maturity model (CMM), and heavily biased towards prescriptive solutions.

Within prescriptive SPI, the norm-driven approach has been dominant (Aaen, Arent et al. 2001). This approach can also be seen as a top-down approach to SPI (Thomas and McGarry 1994)

A major driving force behind the norm-based approach has been the world's largest consumer and producer of software, the U.S. Department of Defense. Faced with increased reliance on software suppliers, in 1984 the Department of Defense established SEI to guide software development organizations toward better practices (Iversen, Mathiassen and Nielsen 2004).

The norm-driven approach focuses on software development processes at the organizational, project, team, and individual level, and is concerned with standardizing and improving these processes (Hansen, Rose and Tjornehoj 2004). Software process norms have emerged from a number of schools, for example CMM and CMMI, Bootstrap, SPICE, and ISO standards. They all prescribe norms for how individuals, teams, and organizations should operate, and for how processes should be standardized and improved. These are also known as best practice and are rooted in the rationalistic paradigm, which promotes a product-line approach to software development using a standardized, controllable, and predictable software engineering processes (Dybå 2000). The main purpose for the SPI initiative in a norm-driven approach is to align the software company with the best practice.

The rationale behind norm-driven approaches to SPI is the convergence hypothesis (Mintzberg 1989) assuming that there exists one best way. The idea is that the quality of a software product is largely governed by the quality of the processes used to create and maintain it (Humphrey, Kitson and Kasse 1989). The main motivation for an organization to apply this approach is to increase the average performance, and at the same time to reduce the variance in performance (increase predictability) (Dybå 2000).

One important assumption of the norm-based approach is that processes can be measured, both as a baseline for improvement and to provide indications for subsequent improvements. The idea behind this approach is that there are well understood software development processes that everyone agrees on, which can be recommended in all situations. Organizational improvement in this context is normally related to a maturity ideal; the mature organization has articulated, standardized, measurable software development processes, and measures them in order to learn how to improve them further. Maturity levels can be measured using various questionnaire-based techniques, and 'immature' organizations should normally follow a prescribed roadmap to achieve the next maturity level. Success is largely defined as progress up the CMM/CMMI levels. However, it is important to realize that compliance does not automatically lead to success (Aaen, Arent et al. 2001). An organization may comply with what is seen as best practice, but still fail to meet its own needs.

2.2.3 Tailoring best practice

Several studies indicate that standardization and the usage of software development methods tend to increase the productivity and quality of software development (Iivari 1996). However, a methodology cannot have an impact unless it is used (Devaraj and Kohli 2003). One challenge when standardizing the development process is the fact that there is often a wide disparity between the official development process and the actual behaviour of developers in practice (Fitzgerald 1997). One reason for this is the belief that there exist software development processes that everyone agrees on which can be recommended in all situations. However, research has shown that the development methods need to be tailored to the actual development context (Fitzgerald, Russo and O'Kane 2003).

A common way of tailoring the best practice methods to the company's needs is to assign the task to a group of expert process engineers as described by Becker-Kornstaedt (2001). These process engineers are in charge of the process improvement planning, execution, and evaluation, and for documenting the newly tailored process. Best practice can be implemented in the organization in the form of an electronic process guide (EPG) (Scott, Carvalho et al. 2002). Kellner et al. (1998) argue that process guides can help users of processes to track their work and implement the processes effectively.

An argument that has been presented against using process guides is that the mechanistic nature of structured development does not fit the complex reality (Ciborra 1993). Parnas and Clements (1986) acknowledged that process descriptions do not represent real-life complexity, but argued that they are useful nonetheless, because the description of an ideal process can help users of the process to bring the real process closer to this ideal. In spite of the problems with process guides, most companies make them available on the company Intranet or on wikis creating electronic process guides.

2.2.4 Change-driven approach to SPI

Change-driven development has a different focus on how work is coordinated and subsequently on how SPI is implemented. Change-driven development focuses on leadership and collaboration, informal communication, and aims at an organic (flexible and participative, encouraging cooperative social action) organizational form (Nerur, Mahapatra and Mangalaraj 2005). The ideologies of agile software development emphasize the need for process adaptation within ongoing projects, and seek to move process control from the organizational level to the practitioners (Lycett, Macredie et al. 2003). Because software developers work in teams, the SPI work in the change-driven approach should focus on improving teamwork.

While SPI in the plan-driven environment focuses on reflection after the projects are finished in what is often known as post-mortem meetings (Dingsøyr 2005), Salo and Abrahamsson (Salo and Abrahamsson 2007) argue that the SPI in the change-driven environment is concerned with constant reflection and therefore continuous improvement. The primary focus is on the immediate use of the experiences of developers in improving the ongoing project. Table 5 shows how Salo and Abrahamsson (ibid) understand the underlying differences between traditional and agile software development and SPI.

Table 5 Underlying differences between traditional and agile software development and SPI (Salo and Abrahamsson 2007)

	Traditional software development and SPI	Agile software development and SPI
Software development process	Universal approach and repeatable solutions to provide predictability and high assurance.	Flexible approach adapted to collective understanding of contextual needs to provide shorter development times, responsiveness to rapid changes, increased client satisfaction, and lower defect rates.
Process control	Control on organizational level.	Self-managing teams.
Primary means of knowledge transfer	Document based knowledge transfer.	Face-to-face communication.
Immediate focus of process improvement	Improvement of organizational software development processes and future projects.	Improvement of daily work practices in the ongoing project.

Aaen et al (Aaen, Börjesson and Mathiassen 2005) describe SPI in the agile mindset as decentralized and bottom-up with an emphasis on project and team level standardization of processes. Key to this approach is the support for adaptive SPI practices. Learning takes place within the project through continuous sense-and-response cycles, which identify current weaknesses, initiate new efforts, and implement their results as the project evolves and delivers its outcomes. The characteristics of the CMM and agile mindset according to Aaen et al (ibid) are shown in Table 6.

Table 6 Characteristics of the CMM and Agile mindset in SPI (Aaen, Börjesson and Mathiassen 2005)

Issue	CMM mindset	Agile mindset
Organization	Centralist, top-down	Decentralized, bottom-up
Coordination	Between SPI projects	Between SPI and practice
Process	Generic	Dedicated
Diffusion	Process push	Practice pull
Learning	Software organization level	Software project level

The agile team is also supposed to be self-managing and empowered, which means from a socio-technical perspective that the team members are responsible for managing, monitoring, and improving their own processes (Trist 1981). Therefore, SPI in change-driven development can be classified as a bottom-up approach.

2.3 SPI and organizational issues

Organizational issues are of great importance in SPI (Børjesson and Mathiassen 2004; Hansen, Rose and Tjornehoj 2004; Dybå 2005; Mathiassen, Ngwenyama and Aaen 2005). In a quantitative survey of 120 software organizations on the key factors of success in SPI, Dybå (2005) found that success depends critically on six organizational factors: business orientation, involved leadership, employee participation, concern for measurement, exploitation of existing knowledge, and exploration of new knowledge. According to Mathiassen et al. (2005) the organization must be able to change four related organizational elements for SPI to have a lasting effect: process, structure, people, and management. In addition, to succeed with SPI work, it is essential to regard the software development organization as a learning organization (Mathiassen, Ngwenyama and Aaen 2005). People are the key ingredient of any well functioning software process (Aaen, Arent et al. 2001); therefore, improving and coordinating work in software teams should be at the heart of software process improvement.

Because of the importance of teams in SPI and software development, a brief introduction to this field will be given. Further, because the complex process of developing software is about coordinating work, SPI is about improving this coordination. Therefore, a model that helps understanding how work is coordinated when developing software will be explored. Finally, SPI is about learning and participation, which will be discussed at the end of this chapter.

2.3.1 Software development teams

Software development depends significantly on people and team performance, as does any process that involves human interaction. A common definition of a team is "a small number of people with complementary skills who are committed to a common purpose, set of performance goals, and approach for which they hold themselves mutually accountable" (Katzenbach and Smith 1993). The topic of teamwork has attracted research from several disciplines (Guzzo and Dickson 1996; Cohen and Bailey 1997; Sapsed, Bessant et al. 2002), and teams developing software have been studied extensively. This section will briefly introduce the field of teamwork in software development.

One major challenge that often makes software teams different from other teams is the fact that software development teams are typically formed anew for each project, depending on project requirements and who is available (Constantine 1993). It is extremely rare for an entire team to move from one project to another. Thus, software development teams seldom develop a history of working as a team over multiple projects. This is one reason why it takes time for software teams to develop shared mental models. In software teams specialization and division of labor is common (Krasner 1999), and because shared mental models are often missing, this can sometimes lead to a divergence in knowledge about the group processes by individual members (Levesque, Wilson and Wholey 2001). Subsequently, division of labor and missing shared mental models are a threat to well functioning software teams, and their ability to improve the development processes.

To increase the commitment and shared mental models among the team members, project goals, system requirements, project plans, project risks, individual responsibilities, and project status must be visible and understood by all parties involved (Jurison 1999). In addition, for achieving cooperation in such teams, face-to-face communication, repeated interactions, monitoring of rule compliance, and sanctions for non-compliance are important (Tenenberg 2008). Job satisfaction is also essential in software teams. Acuna et al. (2009) studied personality factors, cohesion, conflict, task characteristics, and team satisfaction, and found that software developers need to have control over their own work, and over the scheduling and implementation of their own tasks. This is also known as individual autonomy.

Software development teams can be organized in several different ways. Sawyer (2004) argues that there exist three generic archetypes of software development teams: sequence, group, and network.

- The sequence archetype enacts the belief that a good process leads to a good product. Software development is seen as a linear, task-driven, structured effort driven by a known and predefined ordering of the requisite tasks. People's roles are task-specific, discrete, specialized, and identifiable. The control orientation, formalized interactions among team members, and emphasis on automation suggest that there is little need for strong social bonds. Examples of the sequence archetype include software teams following the traditional waterfall model, the CMM, and the SPICE approach.
- The group archetype is based on a set of predefined tasks, which are assigned based on the collective skills and weaknesses of the group members. The tasks are sequential but iterative, and there is explicit attention to process improvement by the members of the group. The examples of group archetype are the iterative and the evolutionary approaches to software development.
- The product is the central focus in the network archetype; production processes are secondary. The development effort takes shape through the network ties developed by the participants. The strength of these ties reflects the frequency and value derived from interaction. In the network archetype, the people's connections and the tasks they perform define the process. One belief underlying the network archetype is that a good product comes from having good people. This people-first approach recognizes that it is difficult (if not impossible) to replace key members of a network because they represent important hubs. Open source software development efforts represent a form of the network group archetype.

Furthermore, Sawyer (2004) argues that a range of hybrid approaches exists, and that these can be decomposed into some combination of the three social structure archetypes. I argue that a team following the plan-driven approach can be seen as the sequence archetype team, and the team following the change-driven approach can be described as the group archetype team.

With the introduction of agile software development, self-managing software teams have become widely accepted. Therefore, to understand fully software teams and change-driven development in particular, it is necessary to understand the concept of self-management. Self-managing software teams are discussed in the next section.

2.3.2 Self-managing teams

Self-managing teams (Hackman 1986) are also known as autonomous or empowered teams (Guzzo and Dickson 1996; Uhl-Bien and Graen 1998; Kirkman and Rosen 1999;

Langfred 2000; Tata and Prasad 2004). Self-managing teams represent a radically new approach to planning and managing software projects. However, the notion of self-management is not new; research in this area has been conducted since Eric Trist and Ken Bamforth's study of self-regulated coal miners in the 1950s (Trist and Bamforth 1951).

In a self-managing team members have responsibility not only for executing the task but also for monitoring, managing, and improving their own performance (Hackman 1986). Furthermore, leadership in such teams should be diffused rather than centralized (Morgan 2006). Shared leadership can be seen as a manifestation of fully developed empowerment of a team (Kirkman and Rosen 1999). When the team and the team leaders share the leadership, it is transferred to the person with the key knowledge, skills, and abilities related to the specific issues facing the team at any given moment (Pearce 2004). While the project manager maintains the leadership for project management duties, the team members lead when they possess the knowledge that needs to be shared during different phases of the project (Hewitt and Walz 2005). Therefore, improvement work in such teams needs to focus on improving both the processes owned by the project manager (project management duties), and the processes owned by the team (decision-making, performing tasks, solving problems, coordinating feedback).

Self-management can also be understood as a strategy for SPI itself, since it can directly influence team effectiveness, improvement work, and innovation. Self-management has also been found to result in more satisfied employees, lower turnover, and lower absenteeism (Cohen and Bailey 1997). Others also claim that self-managing teams are a prerequisite to the success of innovative projects (Takeuchi and Nonaka 1986), especially the innovative software projects (Hoegl and Parboteeah 2006). Moreover, self-management brings decision-making authority to the level of operational problems and uncertainties, thus increasing the speed and accuracy of problem solving (Tata and Prasad 2004), which is essential when developing software. Adaptability is especially important in such teams since operational decisions are made incrementally while important strategic decisions are delayed as much as possible, in order to allow for a more flexible response to last minute feedback from the market place (Takeuchi and Nonaka 1986). Furthermore, having team members cross-trained to do various jobs increases functional redundancy, and thus the flexibility of the team in dealing with personnel shortages.

Although self-management seems to be a sensible strategy for SPI, some studies offer a more mixed assessment of the effect of self-managing teams. One reason is that such teams can be difficult to implement. Effective self-managing units cannot be created simply by exhorting democratic ideals, by tearing down organizational hierarchies, or by instituting one-person-one-vote decision-making processes (Hackman 1986). Research on team performance also indicates that the effects of such teams are highly situation dependent, and depend on factors such as the nature of the workforce and the nature of the organization (Guzzo and Dickson 1996; Cohen and Bailey 1997). Also, self-managing teams risk failure when used in inappropriate situations or without sufficient leadership and support (Hackman 1987).

2.3.3 Coordinating the development process

Independently of the level of self-management and of which team archetype or combination of archetypes is used (sequence, group, or network) (Sawyer 2004), the software development team needs to coordinate their work in an effective manner. The team must coordinate the efforts of those who are part of the process, as well as ensure coordination with suppliers, clients, and other groups both outside and inside the organization. The team has to make sure the work is done and fits together, that there is no duplication, and that components of the work are handed off expeditiously (Kraut and Streeter 1995). Coordination mechanisms are the organizational arrangements, which allow individuals to realize a collective performance (Okhuysen and Bechky 2009). In addition, research on software development teams found that team performance is linked with the effectiveness of teamwork coordination (Kraut and Streeter 1995; Hoegl and Gemuenden 2001). Understanding how work is coordinated is therefore essential to understanding software development and software process improvement.

One of the most common methods to maximize efficiency by coordinating work has been scientific management. Scientific management operated by examining the work which was being performed and decomposing it into its most basic elements, thereby allowing for specialization and the reduction or elimination of waste (Okhuysen and Bechky 2009). Software development has always been influenced by this view (Dybå 2000). However, as the nature of work in software organizations changed due to the shift away from the manufacturing way of thinking, the limitations of this coordination theory have become evident. In the creative work of software design, a single optimal solution may not exist and progress towards completion can be difficult to estimate (Kraut and Streeter 1995). One reason is that interdependencies between different pieces of work may be uncertain or challenging to identify, making it difficult to know who should be involved in work, and whether there is a correct order in which parties should complete their own specialized work (Okhuysen and Bechky 2009). Early research by efficiency experts and organization design theorists rested on the assumption that organizational arrangements can be designed for optimum performance. Recent research is less concerned with optimizing structures for a given environment, assuming that people in organizations must coordinate the work regardless of the organizational design (ibid).

This thesis relies on the framework for coordinating work suggested by Mintzberg (1989). According to Mintzberg, three basic coordinating mechanisms describe the fundamental ways in which organizations can coordinate their work:

1. Mutual adjustment - based on the simple process of informal communication.
2. Direct supervision - one person takes responsibility for the work of others by issuing instructions and monitoring their actions.
3. Standardization - of which there are four types: work processes, output, skills (as well as knowledge), and norms.

The mechanisms may be somewhat substitutable by each other, but all will typically be found in a reasonably developed organization. In the area of software development, all these coordinating mechanisms are important. Different task complexities require different coordination mechanisms (Mintzberg 1989). Mintzberg argues that simple tasks are easily coordinated by mutual adjustment, but when work becomes more complex, direct supervision tends to be added and takes over as the primary means of coordination. When things get even more complicated, standardization of work processes (or outputs) takes over as the primary coordinating mechanism, in combination with the other two. Then, when things become really complex, mutual adjustment tends to become primary again, but in combination with the other coordination mechanisms.

As explained earlier, software organizations often employ experts in multidisciplinary teams which carry out projects in a complex and dynamic environment. According to Mintzberg (1989), such an organization can be classified as innovative, and in such organizations, mutual adjustment should be the most important coordinating mechanism. The managers should avoid rigid control (direct supervision) that impairs creativity and spontaneity (Takeuchi and Nonaka 1986). To innovate means to break away from established patterns, and thus the innovative organization should not rely on any form of standardization for coordination. The paradox is that standardizing work (development) processes has always been important for software organizations (Dybå 2000).

The plan-driven approach is often described as promoting a hierarchy structure involving a command-and-control style of management with clear separation of roles

(Nerur, Mahapatra et al. 2005; Nerur and Balijepally 2007). In this approach, the project manager is responsible for most decisions, and common understanding can be developed when plans are handed over to the team members from the project management. Therefore, in addition to standardization of the development process, direct supervision is important in plan-driven development. The change-driven team relies on shared decision-making, which involves stakeholders with diverse backgrounds and goals. This is more complicated compared to the traditional approach, where the project manager is responsible for most decisions (Nerur, Mahapatra and Mangalaraj 2005). Therefore, change-driven development favors mutual adjustment. Mutual adjustment in its pure form requires everyone to communicate with everyone (Groth 1999). Hence, to employ mutual adjustment as the primary coordinating mechanisms the software team needs to be cohesive and, since our communication abilities are limited, it has to be small.

2.3.4 Organizational learning and SPI

One of the most important driving forces for software process improvement is that the software developers actually learn how to improve their activities (Dybå 2000; van Solingen, Berghout et al. 2000; Børjesson and Mathiassen 2004). SPI can be seen as an organizational change mechanism; therefore, commitment to learning rather than to any SPI model is needed to succeed with SPI. The learning process when conducting SPI demands group learning (van Solingen, Berghout et al. 2000), because software development is a highly collaborative activity carried out within teams, projects, departments, and companies; it always concerns a group of people.

While learning is important in SPI (Dybå 2005), there are several reasons why it is also challenging. First, Aaen et al (2001) claim that the SPI literature is not informed by organizational change and learning theory. Thus, approaches to SPI overlook many issues of organizational learning that affect how experience is perceived, and how change is institutionalized. Second, software development is a highly complex task. When problems become increasingly complex and ill-structured, the need for learning increases, but so does the difficulty in carrying out effective learning (Argyris 1976).

In their theory on learning, Argyris and Schön (1996) distinguish between what they call single and double-loop learning in organizations. Single-loop learning is to change practice as problems arise in order to avoid the same problem in the future. For example, management often engages in single-loop learning by monitoring development costs, software quality, sales, client satisfaction, and other indicators of performance to ensure that the organizational activities remain within established limits, keeping the organization “on course”. In single-loop learning, if outcomes of actions are not met, the actions are changed slightly to achieve the desired results. It is a feedback loop from observed effects to making some changes or refinements that in turn influence the effects, see Figure 2.

Double-loop learning, on the other hand, is when time is taken to understand the factors that influence the effects, and the nature of this influence, called the governing values (Argyris and Schön 1996). It is about using the problems being experienced to understand their underlying causes, and then to take some action to remedy these causes. One example is what happens when a software error is corrected. Correcting the error itself can be seen as single loop learning, but if something is done with whatever caused the error to be introduced, that is considered double-loop learning. The changes based on this type of understanding will be more thorough. Even the act of introducing agile methods to a project team is a change act that requires double-loop learning. One example is the introduction of the self-managing team, which requires that operating norms and rules are allowed to change (double-loop learning) along with transformation in the wider environment (Morgan 2006). When focus is on single-loop learning, norms and values remain unchanged (McAvoy and Butler 2009).

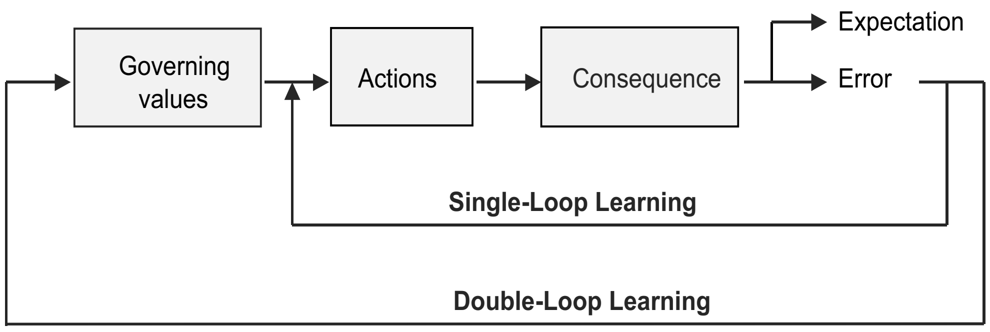

Figure 2 Single and double-loop learning (Argyris and Schön 1996)

Single-loop learning is nevertheless predominant in most organizations (Argyris and Schön 1996). Although some organizations have been successful in institutionalizing systems for double-loop learning, many fail to do so. This failure is especially true in bureaucratized organizations, whose fundamental organizing principles often operate in a way that actually obstructs the learning process (Morgan 2006).

To sum up, in single-loop learning a specific problem is solved, while in double-loop learning, a set of governing variables (goals and constraints) is questioned, which may impact many future problems. Single-loop learning is about asking “are we doing things right?”, while double-loop learning is about asking “are we doing the right things?”.

2.3.5 SPI and participation

Employee participation in organizational development has always been important in the Scandinavian work tradition. The reason is the importance of workplace democracy and the socio-technical tradition in the Scandinavian countries (Emery and Thorsrud 1976; Bjerknes and Bratteteig 1995). Participation is also included as an important element in most works on improvement, from Total Quality Management (Deming 2000) to the knowledge management tradition in Communities of Practice (Wenger 1998). Furthermore, participation is one of the most important foundations of organization development and change, and a critical factor for success in software process improvement (Dybå 2005).

Riordan et al. (2005) use a framework with four attributes to define employee involvement:

- Participative decision-making – employees have control over, or a say in, decisions that affect their work.
- Information sharing – information about the organization, including its plans and goals, is made available to employees.
- Training – employees receive appropriate training, which enables them to acquire the knowledge and develop the skills required for effective performance.
- Performance-based rewards – employees perceive that incentives link their actions to outcomes within the organization.

Several techniques are available for promoting participation. For example, search conferences (Purser and Cabana 1997), survey feedback (Baumgartel 1959), self-managing teams (Hackman 1986; Guzzo and Dickson 1996), and quality circles (Lawler and Mohrman 1987; Guzzo and Dickson 1996) are all predicated on the belief that increased participation will lead to better solutions and enhanced organizational problem solving capability.

This thesis, within the context of software process improvement (SPI), relies on the definition of employee participation as defined by Dybå (2005): *the extent to which employees use their knowledge and experience to decide, act, and take responsibility for SPI.*

3 Research method and design

Five studies in three companies were conducted over six years (Figure 3). Studies 1 and 2 were performed in the plan-driven period, where the SPI focus was on the whole development at the department and organizational level. Studies 3 and 4 started in the plan-driven period and continued into the change-driven period. After Scrum training, the processes in projects in study 3 and 4 were tailored to agile methods, and the projects were classified as change-driven development. Study 5 was conducted fully in the change-driven period. A different SPI focus (the organization versus the team) in the two periods called for a different research approach, from studying how developers were using technology to studying how developers worked together. This resulted in two methodologically different phases of my research, which will be further described in this chapter.

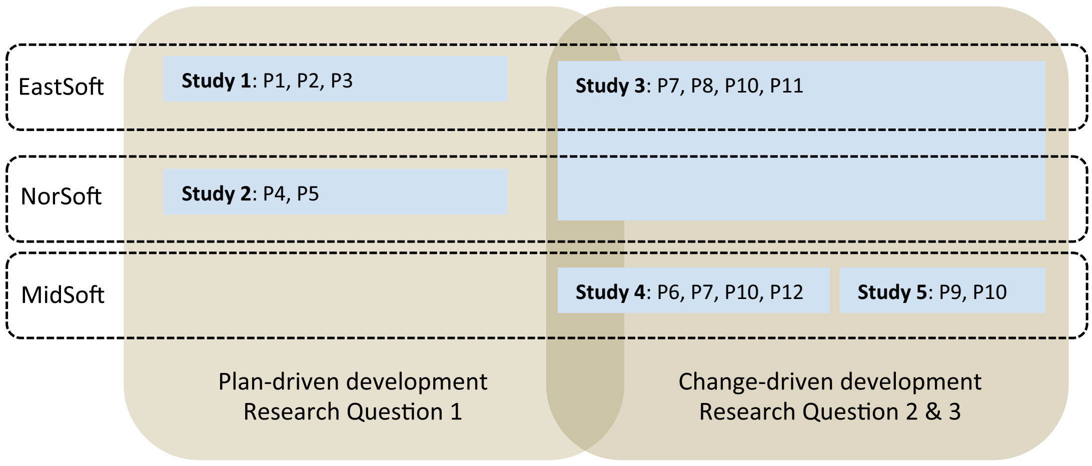

Figure 3 Study design and papers produced in the plan- and change-driven period

Although the research approach has changed during the study, the overall research methodology has always been the same: case study and action research. Case study research was chosen because this methodology is recommended when individual, group, organizational, and social phenomena are investigated (Yin 2002). Furthermore, case studies are found helpful when evaluating the benefits of methods and tools in software companies, because case studies provide a cost-effective way to ensure that process changes produce the desired results (Kitchenham, Pickard and Pfleeger 1995). Finally, case studies make it possible to study a "contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident" (Yin 2002), which can be said to be true for several of the studies presented in this thesis.

The main reasons why action research (Avison, Lau et al. 1999) has been a part of the overall research strategy, is that the research presented in this thesis took place in the context of two larger action research programs, SPIKE and EVISOFT. In these research programs several companies performed improvement work in response to identified problems. Baskerville and Wood-Harper (1998) suggest that action research, as a research method in the study of human methods, is one of the most scientifically legitimate approach available. Action research is also in line with the basic ideas behind Evidence-Based Software Engineering for establishing a fruitful cooperation between research and practice (Dybå, Kitchenham and Jørgensen 2005). Indeed, where a specific new methodology or an improvement to a methodology is being studied, the action research method may be the only relevant research method presently available.

Despite the relevance of action research in the software industry and the fact that action research has been accepted as a valid research method in other applied fields such as organization development and education (Baskerville and Myers 2004), the method is seldom used in the field of software engineering (SE) and information systems (IS). Glass et al (2004) analyzed 1485 papers from the leading journals on Computer Science, SE, and IS in the period from 1995 to 1999. They found that only 0.8% of papers in IS and none of the papers in SE reported using action research as the research method. A review of the software process improvement literature found that most of the work published is *prescriptive*, and that there is a lack of *descriptive* and *reflective* work (Hansen, Rose and Tjornehoj 2004).

The following sections will first present the overall research approaches: case studies and action research. The context this research has been conducted in will be discussed next, followed by the description of the methods used in each of the five studies in the two phases of the research.

3.1 Action research

Action research has been defined as "a post-positivist social scientific research method, ideally suited to the study of technology in its human context" (Baskerville and WoodHarper 1996)(p.235). Baskerwille and Wood-Harper argue that action research is a method that could be identified as a paragon of the post-positivist research methods. It is empirical, yet interpretive. It is experimental, yet multivariate. It is observational, yet interventionist. In addition, the research subjects are often quite willing to carry the costs of being studied, especially since they are allowed to influence the outcomes of the project. To an arch-positivist, it would seem very unscientific. To the post-positivist, it seems ideal (ibid, p.236).

Action research merges research and practice, and its main goal is to achieve change. Together, the researchers and the stakeholders define the problems to be examined, learn about these problems, conduct social research, take actions, and interpret the results of these actions in light of what they have learned (Greenwood and Levin 1998). In other words, I have assisted the companies by not only suggesting and planning the introduction of various SPI and software development techniques and methods, but also in implementing them. One example is that I was involved in Scrum training in all companies and later involved in tailoring and evaluating the development method.

There is a variety of different forms of action research approach. Baskerville and Wood-Harper (1998) identify ten distinct forms of action research in information systems. This thesis uses Davison et al.'s (2004) canonical action research (CAR) method (Figure 4), as it is one of the most widely adopted in the social sciences. The method formalizes an iterative, rigorous, and collaborative research process by describing it in terms of the following five cyclical model phases (originally proposed by Susman and Evered (1978)):

- *Diagnosing* refers to the joint (researcher and practitioner) identification of actual problems and their underlying causes. During this phase, researchers and practitioners jointly formulate a working hypothesis of the research phenomenon to be used in the subsequent phases of the action research cycle.
- *Action planning* is the process of specifying the actions to improve the problem situation.
- *Action taking* refers to the implementation of the intervention specified in the action planning phase.
- *Evaluating* entails the joint assessment of the intervention by practitioners and researchers.
- *Specifying learning* denotes the on-going process of documenting and summarizing the learning outcomes of the action research cycle. These learning outcomes should contribute knowledge to both theory and practice, but they are also recognized as a temporary understanding, which serves as the starting point for a new cycle of inquiry.

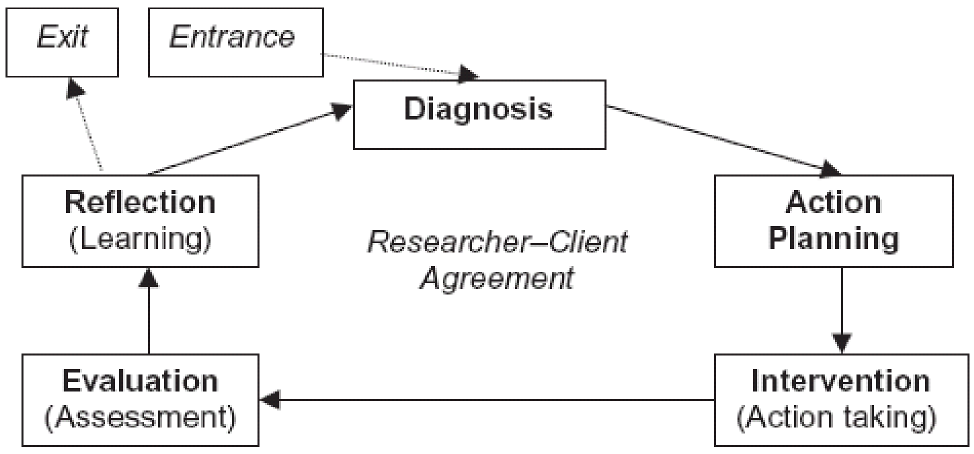

Figure 4 The CAR process (Davison, Martinsons et al. 2004)

Davison et al (2004) developed a set of methodological principles for CAR, and for each principle identified a checklist of specific criteria, which can be used for evaluating a CAR study. The five principles are:

- *The principle of researcher-client agreement*: given the importance of collaboration in action research, this principle seeks to ensure that researchers and practitioners (clients) develop a mutual understanding of and a commitment to the research project, i.e. its scope, focus, and mode of inquiry.
- *The principle of the cyclical process model*: this principle highlights the importance of rigor, as it advocates progressing through all five action research phases in a sequential and systematic manner.
- *The principle of theory*: acknowledging that action research without theory does not constitute research, this principle highlights the importance of using one or more theories not only to guide and focus the research activity, but also to relate the findings to the existing body of knowledge.
- *The principle of change through action*: since the purpose of action research is to change an unsatisfactory situation, this principle stipulates that only interventions appropriate to the problem and the client organization should be designed and implemented.
- *The principle of learning through reflection*: this principle highlights the importance of drawing insights from the research as well as identifying implications for other situations and research contexts.

Action research projects were initiated in each company, hence the overall framework of the research is identified as action research. Together, researchers and the company wrote an improvement plan, which was evaluated and updated three to four times a year. While all research was conducted within action research programs, and studies 2, 3, and 4 were action research studies, in several of the papers they were reported as case studies. One reason was that some of the problems revealed in the diagnosing phase needed a dedicated case study to enable the company and the researchers to understand the underlying cause of the problem. One example was the problem identified early in the change-driven phase, i.e. how to make a team to self-manage. The underlying cause of the problem was not understood until after a longitudinal multiple case study on self-management. Another example was the first study on the EPG. To suggest actions for improvement of the EPG, it was necessary to understand how this tool was actually used and what were the factors affecting its usage level.

3.2 Case study research

Like action research, case study research is concerned with the researcher gaining an in-depth understanding of particular phenomena in real-world settings. This thesis relies on case studies as described by Yin (2002). According to Yin case studies are the preferred research strategy "*...when a «how» or «why» question is being asked about a contemporary set of events over which the investigator has little or no control.*" (ibid p 9). Yin proposes a process consisting of the following phases:

1. Case study design: objectives are defined and the case study is planned.
2. Preparation for data collection: procedures and protocols for data collection are defined.
3. Collecting evidence: execution of data collection on the studied case.
4. Analysis of collected data.
5. Reporting.

Case studies can be based on any mix of quantitative and qualitative evidence, and having multiple sources of evidence is the way to ensure construct validity and to achieve triangulation. By using triangulation, any finding or conclusion in a case study is likely to become more convincing and accurate. According to Yin, there are four types of triangulation: 1) of data sources, 2) among different evaluators (researchers), 3) of perspectives to the same data set (theory triangulation), and 4) of methods.

Yin suggests six major sources of evidence when performing data triangulation. Various sources are highly complementary, and a good case study will therefore use as many of the following sources as possible:

- documentation,
- archival records,
- interviews,

- direct observation,
- participant observation, and
- physical artifacts.

All of the above sources were used in the studies presented here. Examples of documentation are minutes of meetings, project plans, specifications, system documentation, and schedules. Examples of archival records are defects records and usage logs capturing the interactions between the EPG and its users on the intranet. Examples of physical artifacts are the Scrum board and the story cards, which were photographed.

Yin identifies four types of case studies. A case study can be single or multiple. A single or multiple case study can be either holistic (single unit of analysis) or embedded (multiple units of analysis). The evidence from a multiple case study is often considered more compelling and more robust. However, each case in a multiple case study must be carefully selected so that it either a) predicts similar results or (b) predicts contrasting results but for predictable reasons. This thesis relied on both single and multiple case studies.

Considering the importance of the context in both action research and case studies, the next section will describe the research setting of the work presented here.

3.3 Research setting

The three companies were selected for this work, because they all participated in the SPIKE and EVISOFT projects, and all focused on developing process guides and later on implementing change-driven development. Although the companies varied in size, they can all be classified as small or medium-sized. In addition, all companies developed products for clients, which was the reason why all projects studied in the companies combined development activities and client support.

No results are reported from MidSoft implementation of an electronic process guide, because this SPI initiative failed, producing no data to collect and no results to publish. My initial research focus was on understanding process guides in the context of SPI, but the SPI focus changed when all companies decided to introduce Scrum methodology. Research questions 2 and 3 evolved as a result of studying the introduction of Scrum in the three companies.

The following subsections will give a brief description of the three companies involved, followed by the description of the two most important technologies studied: EPG and Scrum. More details on the companies and the technology used can be found in the individual papers.

3.3.1 NorSoft

This company is one of the leading producers of receiving stations for data from meteorological and earth observation satellites. The company works with large development projects, both as a prime contractor and a subcontractor. Clients range from universities to companies such as Lockheed Martin and Alcatel, and government institutions such as the European Space Agency and the Norwegian Meteorological Institute. Most of the software systems developed at NorSoft run on Unix and the remainder - on Linux operating system.

The company has approximately 60 employees. The staff is stable and highly skilled, many with Master's degrees in computer science, mathematics, or physics, and it has what can be described as an engineering culture. Prior to implementing the electronic process guide, the company relied on an extensive system for quality control, which was in accordance with quality routines from the European Cooperation for Space Standardisation and ISO 9001-2000 (ISO 2000). This system was cumbersome to use and did not emphasize aspects such as incremental and component development. As a part of being certified according to ISO 9001-2000, the company decided to develop a process-oriented system for quality control. Two people from the quality department were responsible for coordinating the SPI work in this company. The action research presented in this thesis included working with the company to define the processes for software development and their electronic process guide. Later, the company needed to be able to build stronger teams and to deliver in shorter increments. To this end they decided to implement Scrum.

3.3.2 EastSoft

EastSoft has approximately 150 employees in three organizational units. About 80% of the employees have a Master's or a Doctoral degree. The company has very low staff turnover; less than 10% per year in its software division. Most of the people working in the software development department have been trained as engineers (2/3 of the staff) rather than professional software developers (1/3 of the staff) but the proportion of software developers is increasing. The company aims to hire only the best people. Most of the projects relied on the .NET framework in Visual Studio using C#.

EastSoft produces specialized software for the engineering domain. The company sells mass-market software, but also writes client specific software on a contract basis. In addition to Norway, the company conducts software development in its offices in China, Eastern Europe, and UK. All developers in the projects investigated were located at the company's headquarters in Oslo, Norway.

In the plan-driven period, a separate group within the company, called the SPI group, was responsible for building competence in software development processes, methodology, and supporting tools. This group was also responsible for coordinating the use of processes, methodologies, and supporting tools across all projects. This responsibility included the development, support, and deployment of the company's EPG. The EPG was developed in cooperation with the intended users through small workshops and meetings, where the company's best working practice had been mapped. The EPG used fundamental concepts of the Capability Maturity Model (CMM), Microsoft Solution Framework (MSF), and Rational Unified Process (RUP) (Krutchen 2000). The model is meant to be scalable to both smaller and larger assignments. The EPG was mainly supplementary to the company's work procedures, thus, it was of a voluntary nature.

3.3.3 MidSoft

This company was established in 1996. It has three regional divisions and one separate ICT division. The ICT division consists of a consulting department, an IT management department, and a development department. In addition to software development projects for outside clients, the ICT division develops and maintains a series of off-the-shelf software products, which are developed in-house. During the study, the development department had about 16 employees, divided into a Java and a .NET group.

The company develops a software system for archiving, planning, and coordination, with a combination of textual user interfaces and map functionality. The clients are from all over Norway; one important client was the local government of a Norwegian city's. For their planning and coordination system, winter is the low season of use; the high season begins in spring (March/April). The seasonal constraints gave a relatively narrow time frame for introduction of the new version of software.

Like the other two companies, MidSoft planned to implement their own process guide. However, this failed because they could not allocate the needed resources or motivate the organization for such an investment. The main reason was that the company had a lot fewer developers than the other two companies and therefore, they could not afford having personnel working full time on SPI.

3.3.4 Process Guide

EastSoft and NorSoft both implemented an EPG (Scott, Carvalho et al. 2002; Dybå, Moe and Mikkelsen 2004). Another term - system development methodologies - is used in a similar fashion in the field of information systems. For example, Avison and Fitzgerald (1995) defined this as "a collection of procedures, techniques, tools, and documentation aids, which help systems developers in their efforts to implement a new information system". In software engineering there is a long tradition of work on software processes (Conradi and Fuggetta 2002). Many research groups have focused on software process modelling languages and process-centred software engineering environments (Ambriola, Conradi and Fuggetta 1997).

An EPG may be viewed as an online, structured, workflow-oriented (Georgakopoulos and Hornick 1995) reference document for a particular process, which exists to support participants in carrying out the intended process. An EPG may include the following basic elements:

- Activities: descriptions of "how things are done", including an overview of the activities and details regarding each individual activity.
- Artifacts: details of the products created or modified by an activity, as either a final or an intermediate result of the activity or as a temporary result created by one of the steps.
- Roles: details of the roles and actors involved in performing the activities.
- Tools and techniques: details of the tools and techniques used to support or automate an activity.

Meso et al. (2006) distinguish between process guides developed for specific purposes, calling them strong problem-solving approaches, and general purpose process guides, calling them weak problem-solving approaches.

There were some important differences between the EPGs in two companies studied. The EPG at EastSoft was based on ISO standards and thus was of a voluntary nature. This EPG was inspired by CMM and RUP. At NorSoft, the EPG was inspired by quality routines from the European Cooperation for Space Standardization and ISO 9001, and each project had to generate a tailored instance of the process guide. Therefore, the EPG was not of a voluntary nature. In addition, NorSoft included several tools in their EPG. Examples of such tools are an action list tool with automatic e-mail alerts when the due date of an action passed, a tool providing a template for work breakdown structure, and a tool for following up risk and calculating project risk level.

3.3.5 Scrum

The Scrum teams were given significant authority and responsibility for many aspects of their work, such as planning, scheduling, assigning tasks to members, and making decisions. The Scrum master was in charge of solving problems, which could prevent the team from working effectively. He or she did not organize the teams (designers and developers), but let them organize themselves and make decisions concerning their own activities. The prime responsibility of the Scrum master was to remove the impediments to the process, conduct the daily meetings, make decisions in these meetings, and corroborate these decisions with the management (Schwaber and Beedle 2001).

In Scrum, the self-managing team develops software in increments (called sprints); each sprint starts with planning followed by performing tasks, and ends with a review. The team coordinates and makes decisions on a daily basis. Features to be implemented are registered in a product backlog, and a product owner decides which backlog items should be developed in the following sprint. The product backlog defines everything that is needed in the final product based on the current knowledge. It comprises a prioritized and constantly updated list of business and technical requirements for the system being built or enhanced. Backlog items can include features, functions, bug fixes, requested enhancements, and technology updates. Multiple stakeholders, such as clients, project team, marketing and sales, management and support (Abrahamsson, Salo et al. 2002), can participate in the planning phase to identify the product backlog items. Prioritizing the backlog is a complex communication and negotiation process; however, the product owner is the one responsible for the final prioritization. During the planning meeting (usually every second or fourth week) the product owner is responsible for presenting a prioritized product backlog. The highest priority items from the product backlog are then detailed in a sprint backlog by the developers.

While the product backlog is constantly updated, the sprint backlog should not be changed during the sprint, unless critical business requirements suddenly change, or if the team is not able to deliver as planned. The Scrum team members are empowered and expected to make day-to-day decisions within the project. They are also expected always to select the task with the highest priority when starting to work on the items in the sprint backlog.

The next two sections will describe the research methods applied in the plan-driven and change-driven period respectively, including the discussion of Scrum and process guides.

3.4 Research methods applied in the plan-driven period

In the plan-driven period, the focus of the SPI initiatives was on creating a strong development infrastructure for all projects. Hence, the focus was on the organization and the development department level.

Study 1 can be classified as a single case holistic case study as described by Yin (2002), in that the usage of the EPG was studied in one company. To understand the EPG usage, self-reported measures were combined with objective, computer-recorded measures, and measures of EPG template usage. This study relied on documentation, archival records, interviews, and a survey of 97 EPG users.

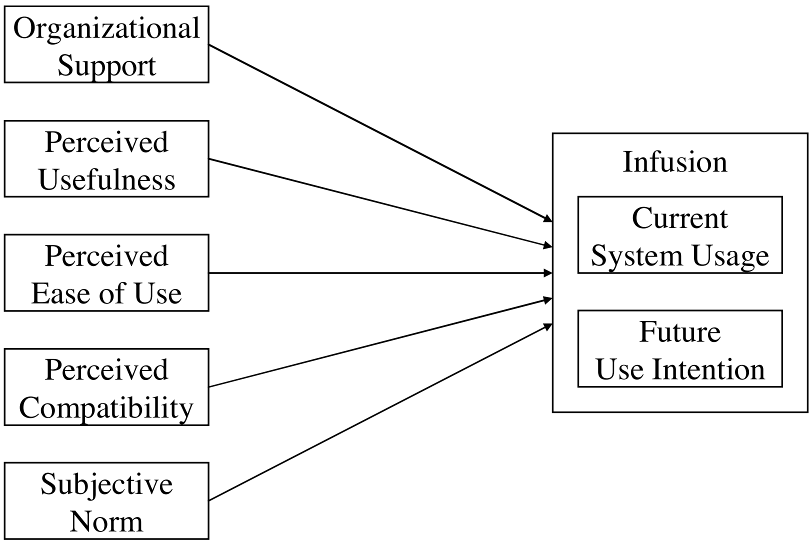

Figure 5 Conceptual model for the survey tested in study 1

The research model empirically tested in the survey is depicted in Figure 5. The model derives its theoretical foundations from combining prior research in technology acceptance (Davis 1989; Venkatesh and Davis 2000) with aspects of innovation diffusion theory (Rogers 1995) and empirically tested research on software developer acceptance of methodologies (Riemenschneider, Hardgrave and Davis 2002). Due to the voluntary nature of using the EPG within the company, voluntariness was not included as a separate construct. In addition, the model was extended to include organizational support.

The objective computer-recorded measures consisted of the total number of server hits on the EPG, and measures of template usage by recording the number of server hits on the template pages. However, simply studying the usage level does not offer an unambiguous interpretation. Is the low usage due to developers knowing what to do without any need for further guidance or is it due to developers believing it is not helpful? Is the high usage due to inexperienced project participants seeking guidance on unfamiliar tasks, or is it due to developers finding it really useful in their daily work? Consequently, there is no general answer to whether low usage is better or worse than high usage with this method of gathering data. Therefore, it was necessary to interview the users and study the documents produced in various projects in more detail. About 1,000 documents in 23 projects were inspected in order to measure the ratio of actual template usage, and 19 users from five different departments were interviewed.

Study 2 was an action research study in a single company. The focus of this longitudinal study was to understand the importance of employee participation in process improvement. An important part of this action research study was to assist the company not only by suggesting and planning the introduction of various participation techniques, but also in applying these techniques. For example the researchers participated in planning the introduction of the EPG, designing and facilitating the process workshop, identifying the processes, and evaluating the usage. Ten users were interviewed in three rounds, and 116 000 look-ups on the EPG over a period of 19 months were analyzed.

3.5 Research methods applied in the change-driven period

In this period the focus changed from studying SPI in the whole development organization to studying SPI in individual projects. The reason was that SPI work in the companies changed from improving the EPG to improving the running projects. To understand SPI in running projects it was necessary to observe the teams and individual developers.

As part of the action research program, I designed a Scrum training program together with the three companies under study. On the first day of the project the participants were introduced to Scrum by a well known and experienced Scrum master. The second day focused on tailoring agile practices to the projects, which were later included in this study. Having the thorough knowledge of the companies and experience with agile development I facilitated this tailoring process. After introducing agile software development in the companies, the teamwork and the processes were regularly evaluated, and new improvement measures were suggested. Five teams were observed (see Table 7) over three years. In addition to the initial training program, the Scrum masters were given extra training and coaching.

Table 7 Teams and data collection sources.

	No. of developers	Agile introduced	Team no	Team size	Project length	No. of interviews	No. of observations
MidSoft	16	At the beginning of the two projects	1	6	11 months	12	75
			2	6	12 months	12	45
NorSoft	60	In the middle of the project	3	7	20 months	13	9
EastSoft	150	In the middle of the two projects	4	8	30 months	11	10
			5	7	30 months	11	10

Study 3, which was an action research study including a longitudinal multiple case holistic study, involved teams 3, 4, and 5, which were studied introducing change-driven development in the middle of the project. In addition to interviews and observations, defect reports were collected from team 4: 449 defects reported during the pre-Scrum phase and 895 defects reported during the Scrum phase. This made it possible to understand how the introduction of the change-driven approach affected the product quality.

Study 4 was an action research study, introducing Scrum in MidSoft. The study included a single case holistic study of a project, which used Scrum from the beginning, focusing on mechanisms that influence teamwork. The study was an interpretative field study (Klein and Myers 1999). The seven principles for conducting such studies, which were proposed by Klein and Myers, were applied in order to determine the main choices related to research method. Table 8 gives an overview of these principles and a description of how they were used in study 4.

Table 8 The use of Klein and Myers' principles in this field research.

The principles for interpretive field research (Klein and Myers 1999)	How each principle was used
1. The fundamental principle of the hermeneutic circle	The understanding of the project was improved by moving back and forth between phases and events. The project had three main phases, each with different characteristics and different events. The data analysis involved multiple researchers having ongoing discussions about the findings.
2. The principle of contextualization	To clarify how situations emerged, the work and organization of the company as well as the context of the project were described.
3. The principle of interaction between researchers and subjects	The researchers' understanding of the project developed through observations, interviews, and discussions with the team participants during coffee and lunch breaks. Project status, progress, and project issues were discussed.
4. The principle of abstraction and generalization	Findings were described and related to the teamwork model proposed by Dickinson and McIntyre (Dickinson and McIntyre 1997).
5. The principle of dialogical reasoning	Dickinson and McIntyre's model was used to identify areas of investigation in the case. The researchers' assumptions were also based on the general knowledge of agile development, SPI, teamwork, and self-management.
6. The principle of multiple interpretations	To collect multiple, and possibly contradictory interpretations of events, data was collected from all participants in the project and from multiple data sources. The case study narrative and findings have been presented to the project participants and led to feedback.
7. The principle of suspicion	By means of analysis, the researchers made themselves aware how roles and personalities affected attitudes to teamwork in order to discover false preconceptions.
	In addition to observations, interviews with different role holders at different levels, and multiple interviews with all team members were conducted. This increased the chance of unveiling possibly incorrect or incomplete meanings.

Study 5 was a single case holistic study motivated by the need for studying agile teams in practice. I conducted ethnographic observations of participants from April 2007 until January 2008. In addition to participant observation (Jorgensen 1989), interviews and documents were used as data sources. Choosing this approach was inspired by Sharp and Robinson (2004). In the beginning of the project I worked extensively with the team on GUI-design, then I was involved in workshops on high-level architectural design, estimation, and participated in planning meetings. From sprint 3 and onwards, the research method changed and the involvement was limited to observing the meetings. During the first research phase no interviews were conducted. The reason was

to create a perception that I was one of the developers. This was a successful approach, judging by discussions with the developers and the problems I was involved in. The observation sessions lasted from ten minutes to eight hours, and the team was visited up to five times a week. As for documents, material from Microsoft Team System was gathered, which gave an overview of items in each sprint, estimates, and burndown charts for the first eight sprints.

3.6 Studies and data analysis

A variety of approaches was used for collecting the data, and the five studies resulted in twelve published papers. This section will give a high level overview of how data was analyzed. Because the qualitative data has been my most important source of evidence in this thesis, this will be the main focus of this section. A more detailed view of all data analysis can be found in the individual papers.

All data from the interviews, minutes, and observations were transcribed and imported into a tool for analysis of qualitative data – NVivo, available from QSR International. All interviews were semi-structured, and all followed a predefined interview guide. The guides can be found in the individual papers.

In the plan-driven phase the interviews were shorter than in the change-driven phase. The reason was that the focus in the first phase was on how developers and managers used and perceived the EPG. Because the EPG was only accessed a few minutes a day at most, the developers did not have much to say about it. In the change-driven phase the interviews focused on teamwork, how the project was running, how decisions were made, what was good and bad, etc. The interviewees subsequently had a lot more to talk about.

All material imported into NVivo has been read and coded several times. Coding was done by assigning interesting expressions of opinions in the text to a specific category with other similar expressions. In this way, concepts were identified and their properties and dimensions were discovered in the data. Events, happenings, objects, and actions/interactions, which were found to be conceptually similar in nature or related in meaning, were grouped under more abstract concepts termed categories. A category represents a phenomenon, that is a problem, an issue, or an event that is defined as being significant to the respondents or to the phenomena observed. An example of coding are the expressions ‘‘you have one place to find information’’ and ‘‘the documentation you need in the project is available’’, which were both coded into category ‘‘information available’’. After the coding, where concepts and categories were created, the connections between categories and their subcategories were identified.

In the plan-driven phase of study 1 the interview data analysis was guided by the principles of grounded theory as described by Strauss and Corbin (1998). A grounded theory is a research method, which seeks to develop a theory grounded in data, which has been systematically gathered and analyzed. As Strauss and Corbin (1998) explained:

> A grounded theory is one that is inductively derived from the study of phenomenon it represents. That is, it is discovered, developed, and provisionally verified through systematic data collection and analysis of data pertaining to that phenomenon. Therefore, data collection, analysis, and theory stand in reciprocal relationship with each other. One does not begin with a theory, then prove it. Rather, one begins with an area of study and what is relevant to that area is allowed to emerge (p. 23).

Inspired by Strauss and Corbin we tried to allow the theory to emerge from the data. Constant comparison is the heart of this process. In this study the interviews were compared to other interviews. By comparing the interviews a theory emerged, and when it began to emerge the data was compared with other existing theory.

Multiple materials describing an event or a phenomenon were collected in the change-driven phase. This enabled a variety of strategies to analyze the material (Langley 1999). First, the project and context were described in a narrative to achieve an understanding of what took place in the project studied. Then analysis was performed across all sources and the results synthesized, as shown in the example from study 4 (Figure 6). Because all team members were interviewed and all teams observed, it was possible to get a deep and thorough understanding of the phenomenon observed.

A variety of techniques and tools were used when analyzing quantitative data. The survey in study 1 was analyzed using Statistical Product and Service Solutions (SPSS). Correlation and multiple regression analyses were used to investigate measurement characteristics and the effects of the independent variables on methodology usage. The EPG look-ups in studies 1 and 2 were imported into Microsoft Excel and analyzed through plots over time for an average number of look-ups per person per month. The defect analysis in study 3 was also performed using Microsoft Excel. The defects of both pre-Scrum and the Scrum phases were classified into different defect *types* using the Orthogonal Defect Classification (ODC) (Chillarege, Bhandari et al. 1992). ODC focuses on tracing each defect back to a specific stage of development.

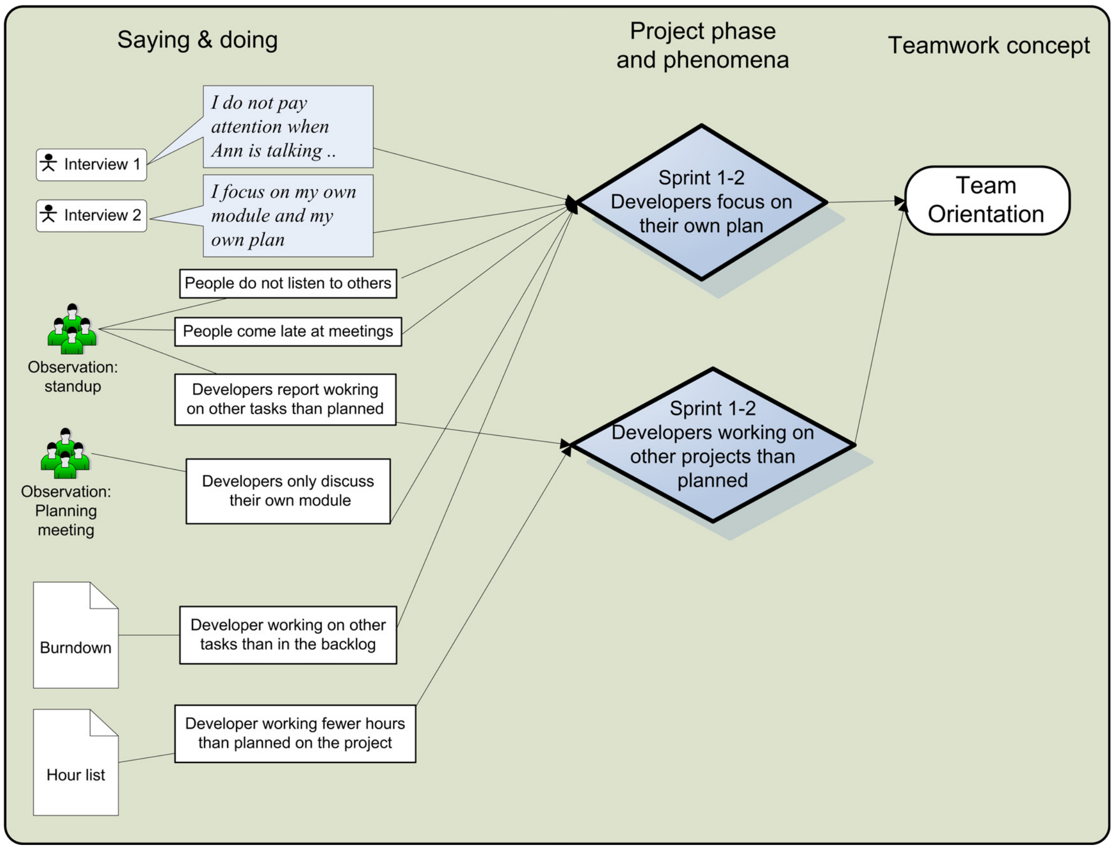
Saying & doing
Project phase and phenomena
Teamwork concept
I do not pay attention when Ann is talking ..
Interview 1
Interview 2
I focus on my own module and my own plan
People do not listen to others
People come late at meetings
Observation: standup
Developers report wokring on other tasks than planned
Observation: Planning meeting
Developers only discuss their own module
Burndown
Developer working on other tasks than in the backlog
Hour list
Developer working fewer hours than planned on the project
Sprint 1-2 Developers focus on their own plan
Sprint 1-2 Developers working on other projects than planned
Team Orientation

Figure 6 Example of the coding process in study 4

4 Results

The results are organized according to the three research questions. Each question is discussed in a separate section: SPI in plan-driven companies (4.1), SPI in change-driven companies (4.2), and SPI challenges implementing change-driven development (4.3). The key findings are the result of the synthesis the contributions made in individual papers using the method described in the previous chapter.

Table 9 shows how the key findings are related to the research questions, and which papers contribute to which key findings. Important quotes from the interviews and observations are highlighted in the text to illustrate the key findings. A description of each key finding starts with presenting the key finding itself, followed by an explanation of how the finding is related to the underlying studies.

Table 9 Relationship of various papers to the key findings and the research questions (RQ)

No	RQ	Key finding	Papers
1	1	Best practice mainly supports project management.	2, 3, 5
2	1	Involvement affects how best practice is adopted.	3, 4, 5
3	1	Individual experts approach is a simple strategy to manage projects.	6, 7
4	1	Post-project reflection is an important learning strategy.	1, 4
5	2	Short iterations make project management easier.	11
6	2	Change-driven development encourages frequent problem reporting.	7, 10, 12
7	2	Long-term quality is in conflict with short-term progress.	7, 9, 10, 11, 12
8	3	Specialization hinders self-management.	6, 8, 9, 10, 12
9	3	Process related problems are difficult to solve.	6, 8, 10, 12
10	3	There are major organizational barriers to self-management.	6, 7, 9, 10

4.1 SPI in plan-driven companies

The process improvement focus in EastSoft and NorSoft was on explicitly defining processes, so called best practices, which could be standardized both within and among the teams. These processes were documented in electronic process guides. MidSoft had a different strategy, where individual experts became responsible for solving client problems directly; developers had full responsibility for a product, from development to client support. Even though the companies relied on different strategies for organizing work, they all used a waterfall approach: a long period of planning and design was followed by implementation, which was then followed by testing. In NorSoft and EastSoft, after a project ended experiences were discussed in what is known as post-project reviews or learning meetings.

The major part of the findings regarding plan-driven companies is related to the concept of best practices.

4.1.1 Key Finding 1: Best practice mainly supports project management

In both EastSoft and NorSoft, the most important SPI initiative was their Electronic Process Guide (EPG), which was based on CMM, RUP, and quality routines from the European Cooperation for Space Standardization. The goal was to describe the companies' best working practice in cooperation with the intended users through small workshops and meetings. Although the goal was to support everyone participating in a project, EPGs ended up mostly supporting project management while the developers received little support. Two reasons for this will be described in the following paragraphs.

First, the process guides were mostly designed to give support for writing documentation and reporting status (paper 3 and 5), both functions being the project managers' responsibility. In addition, the EPG supported mainly project startup and close down, which were also the responsibility of the project management. Other examples of project management support were following up risk and calculating project risk level, progress reporting, writing requirements according to the company standard, documenting use cases, and documenting the minutes of meetings (paper 5). In a study on *Measuring Software Methodology Usage: Challenges of Conceptualization and Operationalization* at EastSoft (paper 2), about 1,000 documents in 23 projects were examined to determine whether the self-reported usage level and usage logs corresponded to the actual usage level. This study confirmed that projects produced deliverables according to the EPG, which demonstrated that processes and checklists supporting management activities were used.

How do developers get support?
"it is important that you know people. You must know whom to talk with if you are going to have a chance of knowing how to solve the tasks. . . . if you are new here, you are not helpless, but it takes a while before you are up and running."
EastSoft (paper 3) p. 27

Second, in a study on the use of an electronic process guide in a medium-sized software development company (paper 3) the developers at

EastSoft described the EPG as not useful because it gave them little or no support during the implementation process. Some developers even claimed that the EPG reduced their productivity. For the developers it was more important to talk to others when performing their tasks (see text box). Although the respondents at NorSoft also claimed that the EPG mostly supported the project managers (paper 5), the developers reported a higher usage than at EastSoft. Studying the usage logs showed that the reason for this was the associated tools, which were part of the EPG. Although mainly supporting project management, NorSoft also included some tools supporting the developers, such as tracking errors, a checklist, and action lists for developers.

4.1.2 Key Finding 2: Involvement affects how best practice is adopted

The management in both EastSoft and NorSoft believed that for the EPG to have an effect, it needed to be adopted by the whole organization. To achieve this, the companies focused on involving the intended users when creating the EPG. Two results confirmed the importance of involvement (paper 5 and 3).

First, the usage level was significantly higher in NorSoft, who involved more users than EastSoft (paper 5). Both companies claimed to involve the intended users. However, in EastSoft only two out of 19 people who were interviewed reported such involvement (paper 3). At NorSoft, nine out of the 31 employees in the primary user group were involved in developing the EPG through process workshops (paper 5). The aim of the process workshops at NorSoft was to make people discuss the current working processes and how they wanted to work in the future. The process workshops, which can be classified as a quality circle, relied on creative group techniques and involved marketing and sales personnel, developers, and project managers. The process workshops are described in detail in papers 4 and 5. One alternative explanation of the higher usage level at NorSoft could be that the EPG was mandatory for all projects there, while it was voluntary at EastSoft. This was the motivation for investigating the effect of participation in NorSoft, which is described next.

Second, the importance of involvement was confirmed when studying the usage level in NorSoft (paper 5). By investigating usage logs and interviews it was found that those involved in the process workshops showed a higher degree of usage, used a larger number of functions, and reported more advantages and disadvantages than nonparticipants. On average, a workshop participant at NorSoft accessed the electronic process guide 65% more often than a nonparticipant did. In addition, the findings suggest that employee participation has long-term positive effects on electronic process guide usage.

4.1.3 Key Finding 3: Individual experts approach is a simple strategy to manage projects

While EastSoft and NorSoft focused on describing and institutionalizing best practice through an EPG, MidSoft relied on a strategy, which gave developers the lifetime responsibility for a product or a part of a product. There was a common understanding among managers and developers that this was the most efficient way of working. While introducing Scrum in MidSoft two reasons were found why this strategy was seen as a simple way to manage projects, and one reason why this strategy was problematic (paper 6 and 12).

First, it allowed many tasks to be completed in parallel. All developers were given responsibility for their own project or module, and there was little dependence on others (paper 6), meaning that there was little need for coordination between developers. This strategy was enabled by the way the company organized development projects and support. Traditionally the company had small projects, mostly with only one or two persons involved. When several developers were allocated to a project, the projects were usually split so each person ended up being responsible for his or her own module. A developer then worked on the module where he or she was seen as a specialist (e.g. map-interfaces or databases).

Second, task responsibilities were clearly defined and understood. Giving the developers' lifetime responsibility also meant that the developers were responsible for client support of the system or module he or she developed (paper 6). The client could usually call the developer directly, and ask him or her to solve problems. The management regarded this approach as a competitive advantage because they were responding quickly to the client requests, faster than their competitors were.

While it was a common understanding that work was performed and coordinated the best way possible, nevertheless developers experienced difficulties. One was that the developers never knew when the client would call, which resulted in work interruptions, therefore progress planning was difficult for development projects (papers 6 and 12).

Even though it was not the primary focus, NorSoft and EastSoft also used specialization as a means to coordinate work efficiently. EastSoft relied on specialization regarding domain models, technology, and roles. As an example, there was often one person responsible for the GUI, one for the business logic, and the chief architect responsible for the architecture. In one project, the chief architect was the one making most decisions on the design and architecture, and he seldom let others participate (paper 7).

4.1.4 Key Finding 4: Post-project reflection is an important learning strategy

Learning from own experience was considered important in all companies. EastSoft and NorSoft focused on learning after the project by organizing post-project reviews or learning meetings. There are three main reasons why this was regarded as an important strategy (papers 1 and 4).

The paper *Improving by involving: a case study in a small software company* (paper 4) describes how project participants in NorSoft accumulated experience and how they reflected on the finished projects to improve their future practice. Answering the following questions facilitated the reflection: What went well? What did not go so well? How to repeat the success? How to improve what did not go so well? NorSoft organized learning meetings in the form of so called post-mortem review. If a project lasted several years, then the post-mortem was sometimes conducted during the project as well. Details on how this was organized can be found in paper 4.

In the same paper it is also described how the company collected experience from the projects to be able to improve the organization as a whole. The tangible outcome of a meeting was a post-mortem report. This report was handed over to the project management forum. This forum, in the form of a learning meeting, then discussed the post-mortem results from several projects, and suggested which improvement actions should be implemented in future projects.

A further motivation for post-project reflection was found in EastSoft (paper 1), where the goal of the post-project reviews was also to evaluate the acceptance and usage of the EPG. The post-project reflection helped the method and tool group, which was also represented in these meetings, to gather lessons learned regarding the deployment of the EPG.

4.2 SPI in change-driven companies

In 2006, all three companies decided to introduce change-driven development. Common for all companies was the need to improve their ability to deliver iteratively and on time, increase software quality, as well as improve teamwork and team communication.

Prior to introducing Scrum, EastSoft was reorganized because of changes in the market. They could no longer afford a separate method and tools group; the group was dissolved, and its members started working in regular software development projects. However, there was still a need for SPI work, and at the same time clients and some developers started talking about using agile software development. It was decided to introduce Scrum. At NorSoft, the motivation for Scrum was the need for a process that better supported short iterations, and the need to strengthen the team. In MidSoft, the size of the projects was growing, and they needed a method for coordinating work in bigger teams. They also needed to deliver more frequently.

The following key findings deal with the characteristics of software process improvement after introducing agile approach. Five teams in the three companies were observed (see Table 10) over three years.

Table 10 Teams introducing agile approach

	Total no. of developers	Agile introduced	Team no	Team size	Project length
MidSoft	16	at the beginning of two projects	1	6	11 months
			2	6	12 months
NorSoft	60	in the middle of the project	3	7	20 months
EastSoft	150	in the middle of two projects	4	8	30 months
			5	7	30 months

4.2.1 Key Finding 5: Short iterations make project management easier

The short iterations characterizing change-driven development made project management easier. This was especially noticeable in team 5 (EastSoft). By following the transition from plan-driven process to Scrum in EastSoft (Paper 11), three main reasons for this finding could be identified.

First, when the team had frequent feedback on the quality it resulted in fewer surprises and better control of the software quality and release date. The reason was that the team was conducting continuous system and acceptance testing, and defect fixing. In the pre-Scrum phase, a long period of planning and designing was followed by a long period of implementation, before testing and debugging commenced (Figure 7 and Figure 8). In the seven-month period of testing and defect fixing, no one actually had the overview of the quality of the code until a few weeks before delivery (see text box). One reason was that correcting one defect

Missing a good overview
"There was an enormous list of defects and errors in the last phase of pre-Scrum phase. It was not easy to have a good overview of this list regarding required work to fix them. Also when correcting one defect, another was found. This resulted in new defects being found late in the process."
EastSoft (paper 11), p. 5

often led to detection of additional defects. In addition, it took the developers a long time to remember the code they had worked on several months earlier. A consequence of missing a good overview of the software quality was that the release date was postponed several times. Furthermore, problems were reported to the whole team in the daily meetings, so team members received frequent feedback on their individual problems, which was seen as a valuable support for fixing defects.

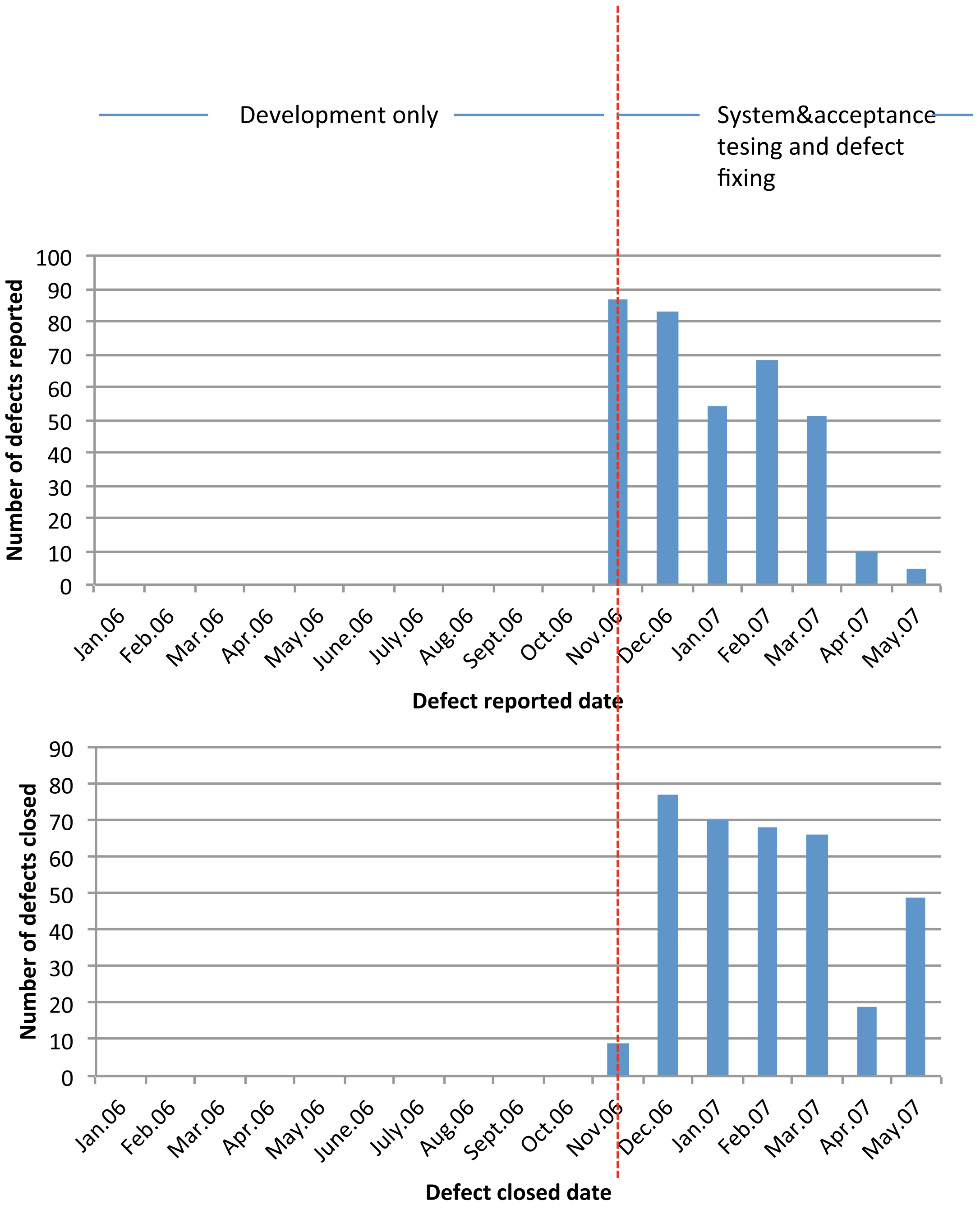

Figure 7 Defects found and closed in the plan-driven phase

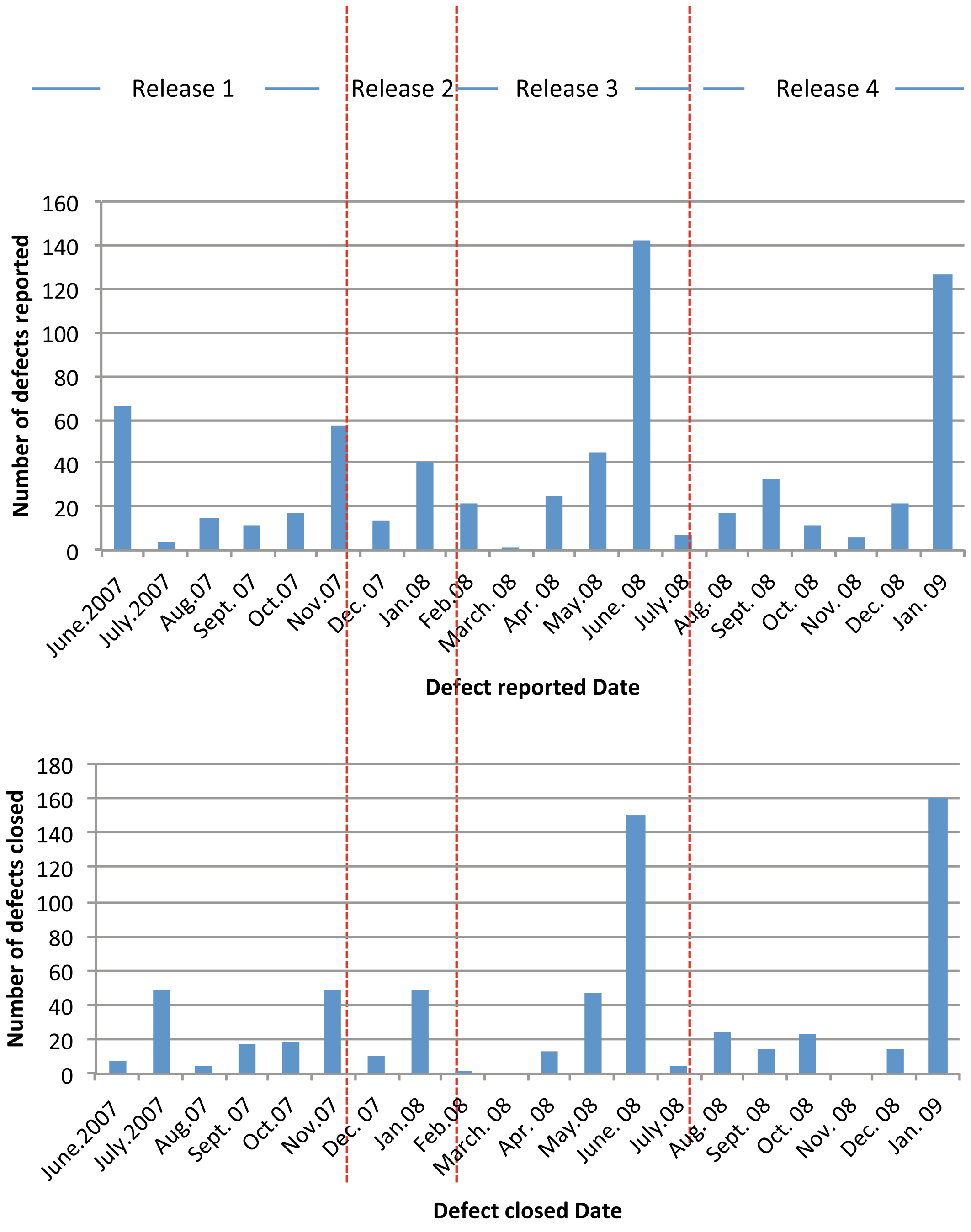

Figure 8 Defects found and closed in the Scrum phase

Second, it became easier to understand and plan the effort needed to solve defects. Defects were now being solved shortly after they were reported, which made it easier to remember the code and understand the cause of the defect.

Third, introducing shorter iterations also made project management simpler by making it easier to protect the team resources (see text box). Without control over the team resources it would be impossible for the team to really commit to what they decided in the planning process.

Shorter iterations make it easier to protect resources
"When other projects ask for resources and they know you are going to deliver in 12 months, it is difficult to deny helping other projects. Now, when using Scrum, we deliver every sprint, and then it is much easier to say no."
EastSoft (paper 11), p. 8-9

While the focus on software quality and knowledge sharing was stronger after introducing Scrum, the new process did not seem to affect the quality of the software produced in each iteration. Hence, the developers did not introduce fewer defects when writing code.

4.2.2 Key Finding 6: Change-driven development encourages frequent problem reporting

Introducing various Scrum meetings created new arenas for the product owner and the team to meet on a regular basis for continuous planning, decision-making, and solving problems. Reporting and solving problems was also given more attention (paper 10).

There were four arenas for frequent problem reporting: the daily meeting, the retrospective, the review meeting, and the planning meeting, each with different types of issues being reported. The daily meeting was found to be an arena for continuously reporting problems and removing impediments to the effective work (paper 7). The agenda of this meeting was to answer the following questions: What have you done since yesterday? What are you going to do today? Did you experience any difficulties? The Scrum master was the one in charge of making sure that the problems reported were taken care of. The frequent team discussions and problem reporting (paper 6) resulted in early identification of problems (see text box).

Problems are reported early
"If the team get stuck, and don't know what to do because of missing or conflicting decisions, this is discovered early. Then we together decide what to do. Before Scrum, developers could use months before we discovered that there were problems we needed to discuss. "
EastSoft (paper 6), p. 220

The retrospective, the second arena for reporting problems, was organized at the end of each iteration. In this meeting, the team focused on what was working well and what needed to be improved, often by comparing the actual situation with what is described by Scrum. Examples of reported problems were defining a stable sprint backlog and finishing it, and problems with the daily meeting (paper 12). Measures were then taken to remedy the issues. By observing these meetings, it became clear that different problems were reported in the retrospective compared to the daily meetings (paper 12). In the daily meetings the focus was mostly on small technical issues (e.g. problems with an error, a library, or component integration), while bigger issues were reported in the retrospective (e.g. problems with the client, the testing framework, and the planning process). The reason was that the daily meetings focused on what the developers were working on at the moment, while retrospective meetings focused on the whole sprint.

Finally, the planning meeting and the review meeting were also arenas for reporting and discussing problems. During the planning meeting held at the beginning of each iteration, problems related to resources, technology, and estimation of tasks were frequently discussed (paper 12). In the review meeting, the team shows the product owner what has been developed during the iteration. The problems discussed in this meeting were related to the product and how the team had handled their tasks (paper 12).

Although problems were reported frequently, it became clear from the interviews and the observations that many problems related to the team process were not reported (paper 12). This is the focus of key finding 9 concerned with why process related problems were difficult to solve.

4.2.3 Key Finding 7: Long-term quality is in conflict with short-term progress

Short iterations in change-driven development are about creating the most business value for the client (immediate value creation). However, this often seemed to be in conflict with the need for long-term quality, which was especially evident when observing the tension between keeping the time schedule and meeting the quality requirements.

While short iterations and frequent testing made it possible to fix defects continually (paper 11), several teams were not strict about the “done” criteria - what does it mean that a component or a feature is finished? (papers 10, 11, and 12). Teams stopped performing thorough testing at the end of iterations in order to be able to deliver all planned features. The main reason for this was that the team felt they needed to show progress, i.e. to deliver what was decided upon in the planning meeting (paper 12). Some Scrum masters and team members even tried to give the impression that the team was better than it actually was. The desire to keep the time schedule in some of the teams hindered the recognition of serious problems with, for instance, a third party component, testing, integration, or performance (papers 9 and 12).

Another challenge was that it seemed difficult to prioritize quality related processes that would improve the quality in the long run (paper 11), when these activities would reduce the pace of producing new features or when the whole team did not agree on the importance of these activities (see text box). Furthermore, it seemed difficult to prioritize architectural work and design. One product owner felt that the team was so busy implementing features that no one was taking care of the biggest concern, which was to build an architecture to last for ten years (paper 7).

Difficult to prioritize long term quality

"Using Scrum is like having a pistol against your neck. It's good and bad. You fix things now and not later. But there are also tasks you should have done like code refactoring. I think we do not use enough time on refactoring, because you need to deliver what you promised. … During our meetings it is difficult to argue for investing resources in doing such tasks. The sprint always seems to be more important."
EastSoft (paper 11), p. 7

All teams discussed the conflicts related to the challenges above. Still, they found it difficult to give priority to quality improvement activities when planning and conducting the sprint. The reason was that keeping the schedule and delivering features according to the plan was seen as more important by the team.

4.3 Key SPI challenges implementing change-driven development

Short development cycles provide continuous and rapid loops of iterative learning, to enhance the processes, and to guide the improvement. The self-managing team was responsible for these improvement initiatives. However, all teams under study still experienced major SPI challenges, especially related to becoming truly self-managing and to handling problems reported during team reflection (team learning).

4.3.1 Key Finding 8: Specialization hinders self-management

The ability of the teams to self-manage was essential for the ability to identify problems, problem solving, and subsequently to determine how SPI was progressing in the companies. However, problems regarding self-management occurred in all teams, and were challenging throughout all projects. Reasons on the team level were the team members not genuinely committing to the team plan, as well as missing shared leadership and shared decision-making. Specialization is identified as the main reason for this, and the results supporting this finding are outlined below.

Because of specialization, it was usually prescribed who should do what in the project (paper 8). Hence, developers mostly worked independently on particular modules according to their specific knowledge (see text box), and they were seldom involved in the work of other developers (paper 10). As a result, shared leadership was difficult. In addition, because developers worked independently (see text box) and had full control over their time schedule and the implementation of their tasks (paper 12), the interaction between the group members did not increase as expected when introducing change-driven development.

No time for collaborating on tasks

"Let the person that knows most about the task solve it! It will take too many resources if several persons are working on the same module, and there is no time for overlapping work in this project. The tasks are delegated and solved the best possible way today. Maybe we can do more overlapping in the next project..."
MidSoft (paper 10), p. 22

Another effect of specialization was lack of team commitment resulting in team members giving higher priority to individual goals, even though the team goals should be the priority in a self-managing team. A number of the respondents explained that because of specialization, they found it difficult to commit to work they were not involved in (paper 12), and consequently it was problematic for them to take part in decisions regarding work of others. This made shared decision-making hard; an individual and decentralized decision-making process resulted in difficulties aligning decisions on the operational level because team members did not know what others were doing (paper 7). In addition, some were unintentionally left out of decision-making processes. Not everyone can be involved in everything, but some team members felt they were excluded from important decisions.

Finally, because of specialization the teams developed unrealistic plans. The planning meeting is where the team is expected to do shared planning and decision-making. However, the meetings often ended up with only a few people talking and the rest listening (paper 7). Some people even fell asleep (paper 9). The poorly managed planning meetings resulted in unrealistic plans with too many tasks. As described earlier, some teams pretended to be more effective than they really were. As a consequence, a new iteration often started by completing what was officially done in the previous iteration. The effect was that the plans became even more unrealistic, and team members focused even more on their own goals and individual plans.

Developers working independently

"When it comes to the daily scrum, I do not pay attention when Ann is talking. For me, what she talks about is a bit far off the topic and I cannot stay focused. She talks about the things she is working on. I guess this situation is not good for the project."
MidSoft (paper 12), p. 486

Because of highly specialized developers and the problems this caused for self-management in the teams, improvement work was challenging and improvement measures were often motivated by individual needs (e.g. solving technical problems, getting new development infrastructure) instead of what the whole team needed.

4.3.2 Key Finding 9: Process related problems are difficult to solve

Through various Scrum meetings, there was an increased focus on reporting problems as described in key finding 6. However, all three companies seemed to have difficulties solving their process related problems. The two main reasons, why process related problems were difficult to solve, were related to difficulties with team reflection in the retrospective meeting and to not reporting process related problems. In Scrum, the retrospective meeting is the most important meeting for discussing and suggesting how to solve process related problems.

All teams had problems making their retrospective meetings work as intended. As an example, team 1 reported the same problems in several consecutive retrospectives (e.g. lack of backup, problems not being reported, and lack of feedback) (paper 12), but no measures were taken to address the *cause* of the problems. When process issues were discussed, teams often ended up talking about the symptoms and not the cause of the problems. In addition, teams usually discussed whether they were doing things right according to the Scrum theory, but they seldom discussed whether they were doing the right things. One example was the conflict between the need for quality and the need for short-term progress. When a team experienced problems with the product quality, the team discussed how to improve the testing process and the testing framework. The real problem however, was that short-term progress was seen as more important by the team than the quality.

For problems to be solved first they have to be identified. However, some process problems were not reported or talked about (paper 12). This became evident when comparing data from observations of daily work and interviews with observations from retrospectives and daily meetings. Team members mostly reported problems related to technology (e.g. development tools, bugs, and integration of third party components). They seldom talked about important process problems such as why the backlog was never completed, why the sprint plan often ended up being unrealistic, why meetings often became unproductive, why some developers were mostly silent in the planning meetings, or why some developers often ended up working on other issues than originally planned.

To understand why problems were not solved it is important to understand why problems were not reported. One reason was that some of the team members perceived the problems as personal and wanted to solve these problems themselves (paper 6) (see text box). Another reason was that some felt that there was too little trust within the team (paper 12) and between the team and the product owner; hence, they did not feel confident reporting problems. Another team experienced relationship problems with the product owner, since he never provided a clear prioritizing of the features for the next iteration. At no point in time however, did the team confront the product owner (paper 7). In team 1 the developers started reporting fewer problems because they did not trust the Scrum master to handle the problems correctly (paper 12). They felt he was overreacting to problems stated in the daily meetings. A third reason for not reporting problems was that when the problems were not handled, the team members stopped reporting them (paper 8). This was seen in team 4, where the team stopped conducting retrospectives for a long period because they felt this type of meeting did not give any value (see text box).

Problems are personal
"When we discover new problems, we feel we own them ourselves, and that we will manage to solve them before the next meeting tomorrow. But this is not the case; it always takes longer time."
MidSoft (paper 6), p. 80

Retrospective do not give value
"the retrospective turned out to be just another nice meeting without really discussing the problems"
EastSoft (paper 8), p. 120

4.3.3 Key Finding 10: There are major organizational barriers to self-management

The implementation of self-managing teams is difficult, if not impossible, if there are critical barriers at the organizational level. Misalignment between team structure and organizational structure can be counterproductive, and attempts to implement self-managing teams can cause frustration for both developers and management. In paper 10 - *Overcoming Barriers to Self-Management in Software Teams* - two important barriers to self-management on the organizational level were identified and discussed.

First, shared resources were a challenge because when developers worked on two or more projects in parallel, and different team goals or needs were in conflict, it threatened at least one of the self-managing teams (paper 9). In addition, some developers had to stop suddenly what they were doing, and support projects they had worked on earlier, without even being formally allocated to such projects. If a developer got involved in a project, he or she was bound to it forever. The developers described this as the "quagmire" (see text box). The reason for this was that parts of the organization expected developers to work even if no resources were provided (see text box). This was a part of the company culture. When team members knew they would

The "quagmire"
[The developer is picking up a coffee at the coffee machine when an internal customer passes]
Internal customer: *Hi, you need to fix the issue I sent you an e-mail about*
Developer [starting to walk back]: *I do not have the time now*
Internal customer [walking after him]: *This is really important and it must be solved now*
Developer [walking faster and obviously stressed]: *I'm working on something very important now, and I do not have the time now*
Internal customer [getting irritated, but stops]: *it need to be solved now*
[The developer gets back to his computer and talks about what just happened]
Developer: *This is the problem with working here. If you ever get involved in developing a system you get stuck in the quagmire. And for every new project you are on, it get worse. We should build a wall of Plexiglas around the developers. When someone want us to do anything they could come over and ring a bell, and leave a note. Later we could look on the notes and decide what to do* [laughing].
MidSoft (paper 6), p. 82

always lose resources during an iteration, it did not make sense for them to commit to the team plan.

Second, a self-managing team needs generalists - members with multiple skills who can perform each other's jobs and substitute as needs arise. However, all companies relied on specialization, and company incentives often supported this culture. An example was found in EastSoft, where one of the most prestigious roles was a chief architect (paper 7). In the project studied, the chief architect participated in important decision meetings with the management; the management trusted him, and he had much influence on future strategy of their products. Becoming a chief architect was seen as positive both from an individual and the company perspective. Because the chief architect was the one solely responsible for the architecture, other team members were rarely involved in the decision-making. In NorSoft, developers were found protecting their knowledge, that is defending their code by not letting others work on it. If the code was important, then the developers became important to the company. Three years before introducing Scrum, NorSoft had to let some developers go, but not any of the "important" specialists (paper 10). Therefore, letting others work on your code was considered a risk that could result in a loss of job. However, being the only one working on important parts of the code was stressful during hectic periods and delayed the team as a whole.

Developers dilemma
"the developers end up in a dilemma. Like today, it's crazy in the other project. The customers are starting to use the system this week, and I'm the only one who can fix problems. Then I just need to help if they are having problems. I do not like to decide which project will not meet its deadlines"
MidSoft (paper 10), p. 24

5 Discussion

The previous chapter described ten key findings from studying SPI in three companies, which transformed from change-driven to plan-driven development. To summarize the results briefly: in the plan-driven period NorSoft and EastSoft focused on reflection after the end of the projects to achieve continuous improvement. Furthermore, all companies provided little support for developers, which resulted in a high degree of freedom for the developers (individual autonomy). The change-driven approach made it easier to manage the project; however, there was a constant conflict between product quality and the need for short-term progress. In addition, problems were reported more frequently, which made change-driven development a strong infrastructure for SPI, but process related problems were difficult to solve. This was caused by problems related to learning and self-management. This section will discuss the results in light of the research questions. When answering the research questions, each question will be discussed in terms of SPI, organizational learning, and self-management.

5.1 Q1: What characterizes SPI in plan-driven companies?

In the plan-driven phase, the employees were involved in identifying company's best practice; however, they were less involved in improving it. The process of identifying best practice can be seen as double-loop learning, while the process of improving it can be seen as single-loop learning. In addition, best practice mainly supported project management activities, giving the developers a high degree of control over their own tasks. These characteristics of plan-driven approach will now be discussed.

5.1.1 Participative bottom-up approach to creating company's best practice

NorSoft and EastSoft relied on the norm-driven approach (Aaen, Arent et al. 2001), which is typically a top-down approach to SPI (Dybå 2000; Aaen, Börjesson and Mathiassen 2005; Salo and Abrahamsson 2007). Surprisingly, it was found that best practice was created through a participatory bottom-up approach to SPI.

Contrary to the top-down character of norm-driven approach, it was found that best practice was based on the local domain and culture indicating a bottom-up approach (Thomas and McGarry 1994). In addition, since process improvement was driven by knowledge of the software developers and their managers, the SPI could be characterized as a participatory bottom-up approach. One explanation for this participatory approach when creating the EPG, is the tradition of relatively high degree of workplace democracy and the influence of the socio-technical tradition in the Scandinavian countries (Emery and Thorsrud 1976; Bjerknes and Bratteteig 1995), which both imply involvement and participation.

Participation was enabled through workshops and meetings to give the users an opportunity to discuss the underlying idea of how software should be developed. NorSoft organized process workshops in the form of a quality circle (Lawler and Mohrman 1987); people discussed problems with the existing process, and identified a new way of working through a brainstorming technique.

5.1.2 Long cycles of single-loop learning and post-project reflection

After the EPG was created, the strategy and the nature of the process improvement work changed from identifying best practice to improving best practice. This can be understood as phases of double-loop and single-loop learning. When creating the EPG, e.g. through the process workshop, existing norms and rules were challenged and changed, and new ways of working were found. This approach to organizational learning is understood as double-loop learning (Argyris and Schön 1996), because the organizations were discussing if they were doing the right things. These findings are in agreement with Morgan (2006), who argues that quality circles offer a perfect illustration of double-loop learning in practice. Descriptions of best practice were improved and changed after discussing project problems in post-project reviews. The question was now: Are we doing things right? From an organizational learning perspective, this SPI strategy can be understood as a feedback loop from observed problems to making changes or refinements, which in turn influence these problems, hence a single-loop learning (Argyris and Schön 1996).

The goal of the SPI work in the plan-driven phase was to avoid repeating problems by reflecting on finished projects. The improvement processes took place outside and were not a part of the project. The underlying assumption of this improvement strategy was that once the process is improved, the next project will use a better process, and therefore, the product being created will improve, or in the least, the risks of conducting new projects will be reduced. This SPI approach was continuous and incremental, and it can be categorized as an evolutionary approach (Aaen, Arent et al. 2001). Although the focus was on continuous improvement through single-loop learning, the speed of learning at the organizational level was slow because reflection was only performed at the end of a project, and process improvement initiatives were not evaluated before being applied in the next project.

Although the project participants were involved in the post-project reflection, it was mainly the quality department or the SPI group that analyzed and suggested improvements to the process descriptions of the company best practice. This way of centralizing SPI in a separate group can be seen as a separation of thinkers (quality department or SPI group) and doers (project participants), which has been the traditional approach to SPI (Aaen, Börjesson and Mathiassen 2005). Such separation can reduce motivation for SPI among the practitioners (ibid) and the level of participation in SPI work.

5.1.3 Project management support focus and high individual autonomy

Plan-driven methods are often characterized by standardization of the work processes (Hansen, Rose and Tjornehoj 2004; Nerur and Balijepally 2007). However, because only part of the work processes were standardized this was found to be only partially true. The goal of the EPG was to support the whole organization in all project activities; however, it ended up supporting mainly project management activities and not development activities. In addition, at MidSoft there was almost no control or process support for the developers solving development problems, because the company relied on specialization, division of work, and personal responsibility.

As a consequence of little or no developer support, developers in all companies ended up relying on the simple process of informal communication as the primary coordination mechanism when solving problems. This is also known as mutual adjustment (Mintzberg 1989). According to Mintzberg (1989), mutual adjustment should be the primary coordination mechanism when solving complex tasks and when engaging in innovative work, as opposed to standardization and direct supervision. Even though this study was not concerned with how the developers used various coordination mechanisms, it can be argued that little focus on standardization and direct supervision enabled mutual adjustment as the primary coordination mechanism among developers. Hence, failing to support development activities resulted unintentionally in the companies supporting innovation and complex problem solving.

Because the developers had few rules and procedural constraints, they had high control over the nature and pace of their work. Using socio-technical theory, this can be understood as developers having task control (Cummings 1978) and high individual autonomy (Van Mierlo, Rutte et al. 2005). Even though this positive effect of high individual autonomy was not the focus of this study, prior research on teams (including software teams) found that high individual autonomy makes individuals motivated and satisfied with their jobs (Langfred 2000), and that employees care more about their work, which in turn leads to greater creativity, more helpful behavior, higher productivity, and higher service quality (Fenton-O'Creevy 1998). Furthermore, motivated software developers often look for better ways to do their job (Yamamura 1999). In addition, Baddoo and Hall (2002) found that autonomy was an important motivator for SPI. So by giving the developers high individual autonomy, the company made it possible for the improvement work to be driven by the developers in the running project. While the literature reports many positive effects of high individual autonomy, the negative effects were observed when the organizations tried to empower their teams by introducing agile software development. The conflict between individual and team autonomy will be further discussed when answering the last research question.

5.2 Q2: What characterizes SPI in change-driven companies?

The motivation for introducing agile software development was that all companies started to question their current project methodology and SPI strategy.

5.2.1 Short cycles of single-loop learning to improve current project

By questioning the plan-driven approach, i.e. asking if we are doing the right things, the organizations once again turned to double-loop learning (Argyris and Schön 1996). Indications of double-loop learning were:

- The SPI group in EastSoft was dissolved because a dedicated group working full time on SPI was seen as too expensive. The members of this group were transferred to ordinary projects, making their SPI competence available for running projects.
- The goal of creating strong empowered teams able to deliver more frequently than before.

After the decision of introducing change-driven development, the focus turned to asking: “Are we doing agile right?”. The main goal of the improvement work was to conduct development according to the processes and roles described by the Scrum methodology, hence single-loop learning (Argyris and Schön 1996)

The change-driven approach was found to provide a strong infrastructure for continuous improvement and single-loop learning, because it encourages frequent problem reporting and fast feedback on improvement measures. The daily meetings made it possible to continuously correct failures and potentially modify the development processes based on actual experience. The outcome was sometimes immediate as it led to an on-the-spot adjustment of actions.

The awareness of a problem is often the first important step towards a solution, and therefore it can be argued that change-driven development is a great facilitator for SPI. This is in agreement with the research of Mathiassen et al (2005), who found that such a strong infrastructure greatly enhanced SPI implementation. The observed effect of short iterations supporting more learning and better facilitating change is also in agreement with the findings of Børjesson and Mathiassen (2004), who studied the Swedish company Ericsson.

By introducing change-driven development, the organization moved from reflection on finished projects to continuous reflection within projects, and therefore changed focus from improving future project to improving current projects. However, while it became less important to put effort into preparation for improving future projects, the developers spread the knowledge from running projects to future projects in which they participated.

It was also found that short iterations made it easier to manage the projects because they gave the team and project managers a continuous and updated overview of the project status and emergent issues, even though some problems were not reported. It was also less complex to correct defects because the time between the introduction and reporting was shortened, making it possible for a developer to still remember the code he or she wrote when the defect was introduced. In the plan-driven period it could be months between introducing and solving a defect.

To summarize, frequent reporting of project and Scrum issues followed by the team suggesting solutions, and a constant overview of the project progress, significantly reduced the time between suggesting and evaluating SPI measures. The result was short cycles of single-loop learning to improve the current project.

5.2.2 Supporting the whole team and not only the project management

After introducing change-driven development, the focus shifted to creating strong self-managing teams and to improving these teams. In the self-managing team (Hackman 1986), team members are responsible not only for executing the task but also for monitoring, managing, and improving their own performance. Involving the whole team in the frequent problem reporting and feedback sessions made this possible. Therefore, it can be argued that there was an explicit attention to process improvement by the whole team and that improving the ongoing project was driven by a participative bottom-up approach. In addition, the SPI work no longer focused only on project management activities and the startup and the closedown phase, but on all project activities.

One example of SPI focus at the team and project level was the frequently reported conflict between long-term quality and short-term progress. This conflict is usually found in all projects because of the dynamic relationships between software processes and the three outcome factors: cost, schedule, and quality (Krasner 1999), however in the change-driven phase this conflict appeared at the end of every iteration. It was found that the need for long-term quality was given a lower priority than the need for short-term progress. This is in agreement with Ramesh et al (2010), who found that agile practices result in non-functional requirements being neglected. Also, the iteration pressure made developers stop following the process implemented (i.e. not proper testing). This is in agreement with Zazworka et al (2010) that received the following answer when asking developers why they did not follow agile practices: ”the implementation of new features to satisfy customer needs had a higher priority than following the steps of the process” p. 8. There seemed to be three main reasons for this. First, the team felt that their primary goal was to deliver according to the release plan and that they always needed to show progress. This could also be one explanation why Scrum masters and team members sometimes tried to give the impression that the team was doing better than it actually was. Second, discussing and changing the release plan was troublesome, because it meant involving top-level management, steering group, and clients. Third, it was observed that the product owner seldom expressed quality as an explicit requirement. Apparently, it was so obvious that often it was not even mentioned.

Although change-driven development enables early identification of problems and a strong SPI focus at the team and project level, it did not improve the quality of the software products in the studied team. Most likely, the productivity of the developers also did not improve, despite the fact that it became easier to correct defects.

Short iterations and frequent feedback enabled SPI support for the whole team. Berente and Lyytinen (2007) argue that iterative development is a more complex concept, and a more complex development process might affect the ability to improve it. The argument supporting the higher complexity of iterative development is that project members need to move back and forth between cognitive or material spaces by constantly refining families of artifacts including conceptual, representational, process instantiations, or methodology. Although this study was not concerned with these particular phenomena, it was found that the iterative approach was challenging mainly because it required the teams to coordinate work more frequently. Frequent face-to-face coordination of work made mutual adjustment even more important as the primary coordinating mechanism within the team, as well as between the team and the product owner. Using the framework for coordinating work by Mintzberg (1989), it can be argued that this made the teams even more capable of solving problems, executing highly complex tasks, and performing innovative work.

5.3 Q3: What are the key SPI challenges when implementing change-driven development?

The self-managing team is the one responsible for SPI on the project level. However, this requires that the team is really able to self-manage, which was found to be a challenge. For a team to self-manage the team autonomy must be strong and the team needs to adopt double-loop learning and learn to learn. In addition, the team must be able to affect managerial decisions, which influence the ability to improve the team's internal processes.

5.3.1 Creating conditions for self-management

Creating team autonomy was a challenge because the high individual autonomy from the plan-driven period persisted, and this often seemed to be in conflict with the need for high team autonomy. High individual autonomy resulted in individual goals becoming more important than team goals. Interaction between group members was difficult and, therefore, threatened collaboration, cooperation, and subsequently the teamwork. The observed effects are in agreement with the findings of Kraut and Streeter in their survey on coordination in software development (Kraut and Streeter 1995). The conflict in combining high team autonomy with individual autonomy was also confirmed by Langfred (2005), who found that when high autonomy exists both at the team and individual level, team performance decreases. One explanation why team members did not reduce their individual autonomy was that it was seen as beneficial by the developers. In other words, team level goals did not immediately become more important than individual goals through implementation of change-driven development. While the organizations seldom debated this problem, they frequently experienced and discussed its symptoms. Examples of symptoms were team members making their own individual plans, not reporting problems, taking decisions without informing others, known as decision-hijacking (Aurum, Wohlin and Porter 2006), and team members taking decisions based on expert power, known as technocracy (Morgan 2006).

Through frequent planning, daily, and retrospective meetings, team level autonomy increased. Team members experienced this as a positive change; however, at the same time they saw it as a more rigid control of each team member compared to the plan-driven period. This is in agreement with Barker (1993), who pointed out that self-managing teams may end up controlling group members more rigidly than with traditional management styles. Even though this particular phenomenon was not investigated in this study, it can be argued that it caused resistance against change because the need for reducing the individual autonomy was not seen as an immediate improvement by the individuals. Resistance against change also made it difficult for the team to improve their development processes. Hence, this was a challenge for implementing SPI in change-driven development.

Teams were also hindered from affecting managerial decisions, which influenced the ability to improve the team's internal processes, and subsequently the ability to self-manage. Management outside the team did not always respect the team's efforts for improvement, which caused the teams to experience symbolic self-management. Symbolic self-management is a well-known obstacle to true self-management (Tata and Prasad 2004). There seem to be two reasons why management outside the team did not respect or support improvement measures suggested by the team.

First, management did not agree with or understand the reason for the problems reported, because management activities and processes changed little by the adoption of the change-driven approach. Introducing change-driven development requires the whole organization to change (Vinekar, Slinkman and Nerur 2006). Examples of the areas with the greatest need for organizational changes were management of resources across teams and handling support. Changing the organizational culture at project level was probably also seen as a threat because it conflicted with existing and established habits of the management. The effect of such threats is confirmed by the argument of van Solingen et al. (2000) explaining why SPI and organizational learning are difficult, and Schneider et al (2002) who found that management might end up blocking emerging change when they do not understand the implication of change.

Second, top management was not involved in the process improvement, although it is a prerequisite for becoming a well-functioning SPI organization (Aaen, Arent et al. 2001; Dybå 2005). Salo and Abrahamsson (2005) found that without support from the organizational level, a majority of improvement measures agreed upon within project teams cannot be implemented. One example of an organizational SPI issue not addressed at the organizational level was the need for building redundancy, to make developers cooperate more, and to make the team flexible and adaptable to changing conditions. From a socio-technical perspective building redundancy is necessary for the team to have boundary control (Cummings 1978), which is essential for creating the self-managing team. However, building redundancy requires additional resources, which should be the responsibility of the organization (Fægri, Dybå and Dingsøyr 2010). However, the top management did not see this as a problem, and hence this did not change much.

5.3.2 Learning to learn

Although the teams frequently reported problems, they experienced difficulties making the necessary changes to solve them. It can be argued that when an organization only suggests improvement measures without being able to implement them, only a potential for improvement exist. One reason for difficulties with implementation was the high individual autonomy, which caused resistance to change. Another reason was that team members either did not manage or were not willing to discuss the underlying cause of problems. Some developers wanted to avoid interpersonal conflict, and some found it more important to conform to other group members, which is an indication of a lack of openness in the team. As a result the teams experienced ineffective decision-making when discussing the need for improvement. This is in agreement with the findings of McAvoy and Butler (2009) on reasons for ineffective decision-making in agile teams. The effect of lack of openness on SPI is also in agreement with van Sollingen et al (2000), who argue that openness and the ability to discuss the underlying problems is one of the most important prerequisites for software process improvement and organizational learning. Because the teams were not able to create a climate for openness and change the way decisions were made, they did not improve the way of reflecting and learning together. In other words, they did not learn to learn. Learning to learn is also known as deutero-learning (Argyris and Schön 1996).

Organizational deutero-learning is critically dependent on individual deutero-learning (Argyris and Schön 1996). This occurs in a team by questioning if we are doing things right (single-loop learning), if we are doing the right things (double-loop learning), and if we make these decisions, when answering “are we doing the right things?” correctly. The teams did mostly single-loop learning by focusing on improving existing agile practices. There were three important reasons for this. First, single-loop learning was the main focus of the plan-driven period. Second, several proponents of agile development claim universal applicability of agile methods, which results in teams focusing on doing things according to the book when focusing on improvement work, and not on questioning if they were doing the right things. Third, some teams tried to give the impression that they were doing better than they actually were. The desire to keep the schedule hindered the recognition of serious problems with the code quality. Impression management (Morgan 2006) is a face-saving process where team members seek to protect themselves from management. This generates shared norms and patterns of group-thinking, which prevent people from addressing key issues.

From an organizational learning perspective, it can be claimed that engaging mostly in single-loop learning was a challenge to SPI, because this stopped the teams from questioning if they were doing the right things. Moreover, after several SPI problems were not solved, team members stopped reporting them, which again affected the ability to improve. It also affected the ability to become self-managing. For a team to become self-managing it needs to change the operating norms and rules within the team, as well as in the wider environment. This is in agreement with the arguments of Morgan (2006), that a team needs to engage in double-loop learning to become self-managing.

5.3.3 Changing the organization

The framework for organizational change proposed by Adler and Shenhar (1990) is useful (Vinekar, Slinkman and Nerur 2006) to explain why it was so difficult for the organizations to change the way of improving and learning to learn after introducing change-driven development (Figure 9). Technological and process changes, like introducing new ways of planning and new ways of coordinating work, were observed in all companies. Such changes occur at the skill and procedure levels, where the magnitude of change is small, the level of learning needed is low, and the time to adjust is short (Adler and Shenhar 1990). However, there was also a need for the organizations

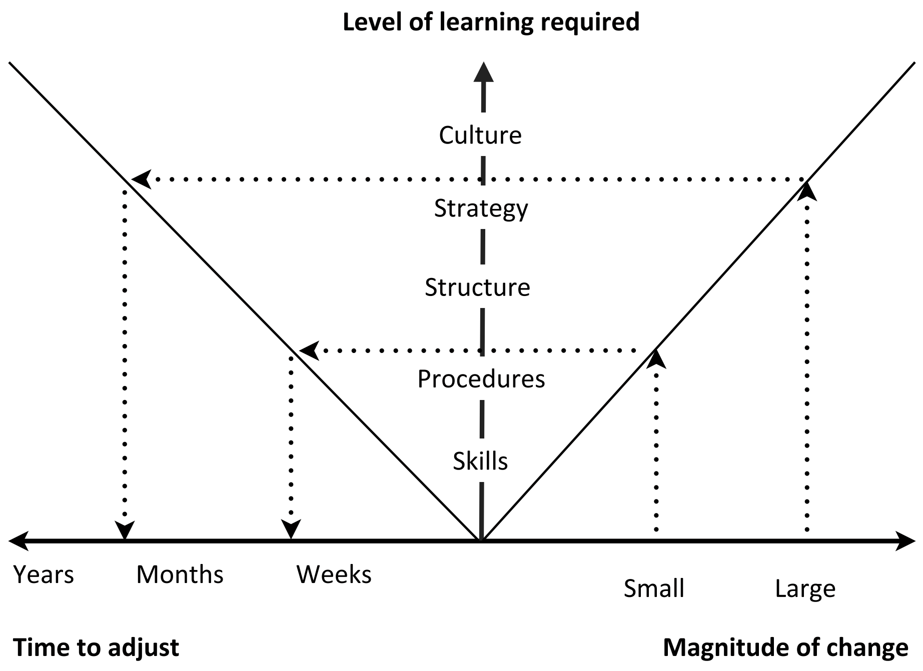

Figure 9 Framework for organizational change (Adler and Shenhar 1990)

to change at the levels of culture, strategy, and structure. The magnitude of such change is relatively large, the level of learning required is high, and the time to adjust is long (Adler and Shenhar 1990). Therefore, it can be argued that introducing and improving change-driven development requires substantial changes in the whole organization, which will take years to implement. This argument is in agreement with Schneider et al (2002) who argue that it takes a long time to create a learning organization.

5.4 Implications for research and practice

This multiple case study of software process improvement in three small and medium-sized companies introducing change-driven development has a number of implications for research and practice.

For research, this thesis shows a clear need for more empirical studies of organizational learning in change-driven development. There are two reasons for this. First, for a change-driven team to improve, it needs to question not only if the team is doing things right (single-loop learning), but also if the team is doing the right things (double-loop learning) and how to develop new structures and strategies to achieve double-loop learning. Second, for an organization to implement change-driven development it needs to change, not only in terms of skills and procedures but also in terms of structure, strategy and culture, which requires a high level of organizational learning. How to do this in practice is not well understood, especially not in the context of change-driven development. In addition, such changes take years, which means that there is a need to do more longitudinal studies on the adoption of agile methods. There is also a need for a better understanding of what a mature agile organization means. Based on the results in this thesis, practicing agile methods for only one or two years is obviously not enough.

Furthermore, this study also shows a need for more empirical studies on self-managing software teams. Self-management is fundamental for change-driven development and for SPI in change-driven development. However, self-management has been largely ignored in research literature on agile methods (Hoda, Noble and Marshall 2011) and it is usually taken for granted. Also self-management is difficult to implement because of the complex nature of software development, which seems to encourage high individual autonomy and specialization. Research needs to explore how to balance individual and team autonomy in a software team, iteration pressure and the need for improvement work, and cross functionality and specialization on an individual level as well as on a team level. Researchers also need to understand the implications for this balancing act on the individual, team and organizational level, because the balancing act is about changing the mindset of individuals, team decision-making and leadership, and informal and formal organizational structures.

As previously pointed out, a main point in the discussion of the results of this research dealt with learning. While the teams were mostly focusing on single-loop learning, it was found that the organization engaged in double-loop learning when creating the EPG and when introducing change-driven development. The companies moved from a period of change to a period of stability, and then back into change (introducing agile methodology), before once again achieving stability (improving agile practices), which can also been described using the punctuated equilibrium model (Lant and Mezias 1992). To succeed with SPI, a balance should be found between optimizing current processes (single-loop learning) and experimenting with new approaches to determine whether they are better than the existing ones (double-loop learning). This is in agreement with van Solingen et al (2000), who argue that there should be a parallel application of optimization of current practices and experimentation with new ones. Hence, the future research should look into the balance between process innovation (Davenport 1993) and process improvement. Also, there is a need for understanding how to develop new structures and strategies to achieve double-loop learning in software organizations.

Learning to learn is difficult; a post-iteration workshop for agile teams, as suggested by Salo and Abrahamsson (2007), could be one framework to apply when the team is having problems learning. This framework relies on an external facilitator helping the team to reflect and to use techniques like the root cause analysis to understand the problems better. All organizations studied in this thesis could have benefitted from improving their retrospectives.

Based on the previous discussion the following recommendations for practice are proposed:

- *Involve the whole organization*. Because of the magnitude of change and learning required to implement change-driven development, and to make SPI effective, the whole organization needs to be involved to be able to succeed.
- *Long-term horizon*. The magnitude of change requires that the organization focuses on both long-term and short-term aspects of SPI.
- *Experiments with new approaches*. There should be a balance between optimizing current processes and experimenting with new approaches.
- *Create a climate for openness*. Openness is important for learning how to reflect and discuss problems together.

- *Balance individual and team autonomy*. Autonomy at the individual level may conflict with autonomy at the group level, counteracting cohesiveness and, indirectly, effectiveness of the team.

5.5 Limitations

The main limitation of this multiple case study is the possibility of bias in data collection and analysis. While the studies in the plan-driven period focused on the organizational level, the focus in the change-driven period was mostly on teams and projects. The reason for this change of focus was the change in the SPI work focus, from the organization to the team. The consequence of this change in focus is that I did not fully investigate SPI on the team and project level in the plan-driven period, and I did not fully investigate SPI on the organizational level in the change-driven period. Subsequently this is reflected in my answer to research question one and two.

The fact that a multiple case holistic design was used makes it possible to reduce some of this bias. The general criticisms of case studies, such as uniqueness and special access to key informants, may also apply to this study. However, the rationale for choosing three companies and several projects in these companies was that they represent a critical case for explaining the challenges, which arise for SPI in plan-driven and change-driven development. The goal was not to provide statistical generalizations about a population on the basis of data collected from a sample of that population.

Another possible limitation is that much of the data collection and analysis was based on semi-structured interviews and participant observation. The consequence of this limitation is that the results are under the influence of my interpretation of the phenomena observed and investigated. The use of multiple data sources made it possible to confirm evidence for episodes and phenomena. The study included observing, talking to, and interviewing team members and managers in all companies and all projects, which made it possible to investigate the phenomena from different viewpoints as they emerged and changed, thus reducing this limitation. Also giving feedback to the observed teams and discussing my interpretation of what was going on helped validating my conclusions.

With action research, the SPI work was also influenced by me participating in planning and conducting the improvement work, as well as in the introduction of the change-driven approach. The consequence of this is that the improvement measures suggested was heavily influenced by what I believed was beneficial for the companies investigated. This approach however provided a unique insight into the problem investigated; also my aim was not to evaluate the success or failure of the SPI work conducted.

5.6 Recommendations for future work

This work is an attempt to understand SPI in change- and plan-driven development. Implementing SPI on all levels of the organization and enabling self-management are essential success factors for SPI in change-driven development; however, none of the organizations investigated was able to establish fully self-managing teams during the period of observation. Accordingly, further work should focus on investigating SPI in a fully self-managing team to determine what characterizes the SPI process and how learning is organized in such a team.

Furthermore, there is a need for exploring how to involve and change the whole organization when adopting change-driven development, and how to make management commit to long-term organizational transformation in a dynamic and turbulent environment. It is therefore important to investigate how the process of organizational change should be conducted in the challenging and complex environment of software development.

Last, there is a need for research on the emerging trend of global sourcing in change-driven development (Smite, Moe and Ågerfalk 2010) and software process improvement. Global sourcing promises organizations the benefits of reaching mobility in resources, obtaining extra knowledge through deploying the most talented people from around the world, accelerating time-to-market, increasing operational efficiency, improving quality, expanding through acquisitions, reaching proximity to market, and many more. However, as the history of previous emerging trends shows, these benefits are neither clear cut nor can their realization be taken for granted. One emerging tendency is that many companies are applying change-driven methods to their distributed projects to meet challenges related to process improvement, communication, and coordination. This will make SPI an even more challenging task because of cultural diversity, time zone differences, and geographical distances. An important area for future research is therefore to understand SPI when applying change-driven methods in global software development.

6 Conclusion

The previous chapter discussed the answers to the three research questions posed in Chapter 1. This final chapter presents the conclusions regarding the overall research problem and questions, and lists the contributions made by this thesis.

Motivated by the importance of SPI in software development, the fundamental differences in SPI between traditional and agile software development, and the lack of research in this area, the overall problem put forward in this thesis was:

> **How does Software Process Improvement work change with the introduction of agile software development in plan-driven companies?**

To narrow the focus of the investigation, the research problem was addressed by studying three research questions. To solve the overall problem formulated in this thesis the answer to each research question will be first presented.

RQ 1: What characterizes SPI in plan-driven companies?

The dominant perspective in NorSoft and EastSoft in the plan-driven period was to identify company's best practice, and to document it in an EPG through a participative bottom-up approach. The goal was to make projects predictable by identifying a common process for developers, managers, sales, and support. Using the concepts of Argyris and Schön (1996), the creation of the EPG can be understood as a double-loop learning activity, while improving it through reflection on finished projects can be understood as single-loop learning. The main focus was on single-loop learning and continuous improvement, however the speed of learning at the organizational level was slow because reflection only took place at the end of a project, and process improvement initiatives were not evaluated before they were applied to the next project. In addition, while the project management activities were well supported, the developers experienced little or no support for their tasks, which made the organization unintentionally support innovation and solving complex problems through mutual adjustment. In addition, all companies relied on specialization and corresponding division of work. As a result, the developers had high individual autonomy, which is an important motivator for SPI.

The main conclusion drawn from the answer to the first research question is that *SPI in plan-driven companies is characterized by a participative bottom-up approach when creating company best practice, project management support focus, high individual autonomy, and long cycles of single-loop learning; the goal of reflection on projects is to improve future projects.*

RQ 2: What characterizes SPI in change-driven companies?

Introducing agile software development can be understood as double-loop learning, while improvement work centered on making agile methods work in the project has been described as single-loop learning. Improvement work also changed focus from improving future projects to improving the ongoing project, which resulted in short feedback loops on improvement measures. Short feedback loops gave the team a good overview of the project progress, which was helpful in managing the project and the SPI work. In addition, SPI changed focus from supporting project management activities to supporting the whole team and all activities. Change-driven development provided a strong infrastructure for SPI because the team reported problems frequently, however it was challenging to solve process related problems.

The main conclusion drawn from the answer to the second research question is that *SPI in change-driven companies is characterized by supporting the whole team, not only project management; practice is improved by short cycles of single-loop learning, and the goal of reflection in projects is to improve the current project.*

RQ 3: What are the key SPI challenges when implementing change-driven development?

Software process improvement in the change-driven period was planned, executed, and evaluated by the empowered self-managing team. The team's ability to implement self-management, i.e. shared leadership, shared decision-making, and high team autonomy, was therefore a key SPI challenge, while specialization was the main obstacle to achieving this. Process problems were identified but often not solved, therefore only the potential for improvement existed. Software process improvement from an organizational learning perspective was particularly challenging because it became evident that the organizations had problems to engage in double-loop learning and to learn how to learn. Introducing change-driven development required a change in skills, procedures, structure, strategy, and culture, which required changes on the individual, project, and organizational level. This is why the transition from plan-driven to change-driven development takes months and years.

The main conclusion drawn from the answer to the third research question is that *SPI challenges while implementing change-driven development are the problems of increasing redundancy to create conditions for the team to self-manage, to learn how to learn, and to perceive the adoption of change-driven development as a large long- term organizational change project.*

Taken together, the answers to these three research questions constitute the main contribution of this thesis. So, returning to the original question on how software process improvement work changes when introducing agile software development in plan-driven companies, the following was found:

- Organizational learning changed from post-project reflection to reflection as part of the project.
- The goal of improvement work changed from improving future projects to improving current projects.
- Improvement work changed from supporting project management to supporting the whole team, and from supporting startup and closedown to supporting all processes in the project.
- The role of participation changed from being a part of creating and improving the best practice descriptions to being a part of improving practice.
- In regards to autonomy, focus changed from individual autonomy to team autonomy and self-management, and as a result, the teams controlled group members more rigidly than in plan-driven development.

While the focus of this thesis was to identify how SPI work changed, I also found that some aspects did not change:

- Single-loop learning was the major focus, and hence the organizations had problems learning to learn.
- The primary coordinating mechanism was mutual adjustment.
- SPI was bottom-up and relied on involvement.
- Developers retained high individual autonomy.

Finally, I found that organizations take self-management for granted. However, a team does not automatically become self-managing by introducing change-driven development, with the main barrier being specialization. Also, creating self-managing software teams is about balancing individual autonomy and team autonomy, iteration pressure and the need for improvement work, and cross functionality and specialization on an individual level as well as on a team level.

References

Aaen, I. (2008). Essence: Facilitating Agile Innovation. Agile Processes in Software Engineering and Extreme Programming. P. Abrahamsson, R. Baskerville, K. Conboyet al, Springer Berlin Heidelberg. 9: 1-10.

Aaen, I., Arent, J., Mathiassen, L. and Ngwenyama, O. (2001). A conceptual map of software process improvement. Scand. J. Inf. Syst. 13: 123-146.

Aaen, I., Börjesson, A. and Mathiassen, L. (2005). Navigating Software Process Improvement Projects. Business Agility and Information Technology Diffusion. R. Baskerville, L. Mathiassen, J. Pries-Heje and J. DeGross, Springer Boston. 180: 53-71.

Abrahamsson, P., Salo, O., Ronkainen, J. and Warsta, J. (2002). Agile software development methods - Review and analysis, VTT Publications.

Acuna, S. T., Gomez, M. and Juristo, N. (2009). How do personality, team processes and task characteristics relate to job satisfaction and software quality? Information and Software Technology 51(3): 627-639.

Adler, P. S. and Shenhar, A. (1990). Adapting your technological base:The organizational challenge. Sloan Management Review (32)1: 25–37.

Ambriola, V., Conradi, R. and Fuggetta, A. (1997). Assessing Process-Centered Software Engineering Environments. ACM Transactions on Software Engineering Methodology 6(3): 283-328.

Argyris, C. (1976). Single-loop and double-loop models in research on decision making. Administrative Science Quarterly 21 (3): 363–375.

Argyris, C. and Schön, D. A. (1996). On Organizational Learning II: Theory, Method and Practise. Reading, MA, Addison Wesley.

Aurum, A., Wohlin, C. and Porter, A. (2006). Aligning Software Project Decisions: A Case Study. International Journal of Software Engineering and Knowledge Engineering 16(6): 795-818.

Avison, D., Lau, F., Myers, M. and Nielsen, P. A. (1999). Action research. Communications of the ACM 42(1): 94-97.

Avison, D. E. and Fitzgerald, G. (1995). Information Systems Development: Methodologies, Techniques and Tools. New York, McGraw-Hill.

Baddoo, N. and Hall, T. (2002). Motivators of Software Process Improvement: an analysis of practitioners' views. Journal of Systems and Software 62(2): 85-96.

Barker, J. R. (1993). Tightening the Iron Cage - Concertive Control in Self-Managing Teams. Administrative Science Quarterly 38(3): 408-437.

Barthelmess, P. (2003). Collaboration and coordination in process-centered software development environments: a review of the literature. Information and Software Technology 45(13): 911-928.

Basili, V. R. (1989). Software development: a paradigm for the future. Proceedings of the 13th International Computer Software and Applications Conference (COMSAC): 471-485.

Baskerville, R. and Myers, M. D. (2004). Special Issue on Action Research in Information Systems: Making is Research Relevant to Practice - Foreword. Mis Quarterly 28(3): 329-335.

Baskerville, R. and Wood-Harper, A. T. (1998). Diversity in information systems action research methods. European Journal of Information Systems 7(2): 90-107.

Baskerville, R. L. and WoodHarper, A. T. (1996). A critical perspective on action research as a method for information systems research. Journal of Information Technology 11(3): 235-246.

Baumgartel, H. (1959). Using employee questionnaire results for improving organizations: The survey "feedback" experiment. Kansas Business Review 12: 2-6.

Beck, K. and Andres, C. (2004). Extreme Programming Explained: Embrace Chage (2nd ed), Addison-Wesley.

Becker-Koernstaedt, U. (2001). Towards Systematic Knowledge Elicitation for Descriptive Software Process Modeling. Proceedings of the Product-focused software process improvement conference (PROFES). F. Bomarius and S. Komi-Sirviö. Berlin Heidelberg, Springer Verlag. 2188: 312 - 325.

Berente, N. and Lyytinen, K. (2007). What is being iterated? Reflections on iteration in information system engineering processes. Conceptual modelling in information systems engineering. A. Sølvberg, J. Krogstie, A. Opdahl and S. Brinkkemper. Berlin ; New York, Springer: 261-278.

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

Boehm, B. (1988). A spiral model of software development and enhancement. Computer 21(5): 61-72.

Boehm, B. (2002). Get Ready for Agile Methods, with Care. Computer 35(1): 64-69.

Boehm, B. (2006). A view of 20th and 21st century software engineering. Proceedings of the 28th international conference on Software engineering. Shanghai, China, ACM: 12-29.

Boehm, B. and Turner, R. (2003). Balancing Agility and Discipline: A Guide for the Perplexed, Addison-Wesley

Boehm, B. and Turner, R. (2005). Management Challenges to Implementing Agile Processes in Traditional Development Organizations. IEEE Software 22(5): 30-39.

Børjesson, A. and Mathiassen, L. (2004). Successful process implementation. IEEE Software 21(4): 36-44.

Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K. and Wong, M.-Y. (1992). Orthogonal Defect Classification-A Concept for In-Process Measurements. IEEE Transactions on Software Engineering 18(11): 943-956.

Ciborra, C. (1993). Teams, Markets and Systems: Business Innovation and Information Technology. Cambridge, UK, Cambridge University Press.
Cockburn, A. and Highsmith, J. (2001). Agile software development: The people factor. Computer 34(11): 131-133.
Cohen, D., Lindvall, M. and Costa, P. (2004). An Introduction to Agile Methods. Advances in Computers, Advances in Software Engineering. M. V. Zelkowitz. Amsterdam, Elsevier. 62.
Cohen, S. G. and Bailey, D. E. (1997). What makes teams work: Group effectiveness research from the shop floor to the executive suite. Journal of Management 23(3): 239-290.
Conradi, R. and Fuggetta, A. (2002). Improving Software Process Improvement. IEEE Software 19(4): 92-99.
Constantine, L. L. (1993). Work organization: paradigms for project management and organization. Commun. ACM 36(10): 35-43.
Cruzes, D. S. and Dybå, T. (2010). Synthesizing evidence in software engineering research. Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. Bolzano-Bozen, Italy, ACM: 1-10.
Cummings, T. G. (1978). Self-Regulating Work Groups: A Socio-Technical Synthesis. The Academy of Management Review 3(3): 625-634.
Davenport, T. H. (1993). Process Innovation: Reengineering Work through Information Technology. Boston, Mass., Harvard Business School Press.
Davis, F. D. (1989). Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. Mis Quarterly 13(3): 319-340.
Davison, R., Martinsons, M. G. and Kock, N. (2004). Principles of canonical action research. Information Systems Journal 14(1): 65-86.
Deming, E. W. (2000). Out of the Crisis. Cambridge, Massachusetts, The MIT Press (first published in 1982 by MIT Center for Advanced Educational Services).
Devaraj, S. and Kohli, R. (2003). Performance impacts of information technology: Is actual usage the missing link? Management Science 49(3): 273-289.
Dickinson, T. L. and McIntyre, R. M. (1997). A conceptual framework of teamwork measurement. Team Performance Assessment and Measurement: Theory, Methods, and Applications. M. T. Brannick, E. Salas and C. Prince. NJ, Psychology Press: 19-43.
Dingsoyr, T. and Moe, N. B. (2008). The Impact of Employee Participation on the Use of an Electronic Process Guide: A Longitudinal Case Study. IEEE Transactions on Software Engineering 34(2): 212-225.
Dingsøyr, T. (2005). Postmortem reviews: purpose and approaches in software engineering. Information and Software Technology 47(5): 293-303.
Dixon-Woods, M., Agarwal, S., Jones, D., Young, B. and Sutton, A. (2005). Synthesising qualitative and quantitative evidence: a review of possible methods. J Health Serv Res Policy 10(1): 45-53.
Dybå, T. (2000). Improvisation in Small Software Organizations. IEEE Software 17(5): 82-87.

Dybå, T. (2005). An empirical investigation of the key factors for success in software process improvement. IEEE Transactions on Software Engineering 31(5): 410-424.

Dybå, T. (2005). An Empirical Investigation of the Key Factors for Success in Software Process Improvement. IEEE Transactions on Software Engineering 31(5): 410 - 424.

Dybå, T. and Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. Information and Software Technology 50(9-10): 833-859.

Dybå, T., Kitchenham, B. A. and Jørgensen, M. (2005). Evidence-Based Software Engineering for Practitioners. IEEE Software 22(1): 58-65.

Dybå, T., Moe, N. B. and Arisholm, E. (2005). Measuring Software Methodology Usage: Challenges of Conceptualization and Operationalization. Fourth International Symposium on Empirical Software Engineering (ISESE), Noosa Heads, Australia, IEEE Computer Society: 447-457.

Dybå, T., Moe, N. B. and Mikkelsen, E. M. (2004). An Empirical Investigation on Factors Affecting Software Developer Acceptance and Utilization of Electronic Process Guides. Proceedings of the International Software Metrics Symposium (METRICS), Chicago, Il, USA: 220-231.

Emery, F. and Thorsrud, E. (1976). Democracy at work: the report of the Norwegian industrial democracy program. Leiden, Martinus Nijhoff Social Sciences Division.

Erickson, J., Lyytinen, K. and Siau, K. (2005). Agile Modeling, Agile Software Development, and Extreme Programming: The State of Research. Journal of Database Management 16(4): 88 - 100.

Fayad, M. E., Laitinen, M. and Ward, R. P. (2000). Software engineering in the small. Communications of the ACM 43(3): 115-118.

Fenton-O'Creevy, M. (1998). Employee involvement and the middle manager: evidence from a survey of organizations. Journal of Organizational Behavior 19(1): 67-84.

Fitzgerald, B. (1997). The use of systems development methodologies in practice: a field study. Information Systems Journal 7(3): 201-212.

Fitzgerald, B., Russo, N. L. and O'Kane, T. (2003). Software development method tailoring at Motorola. Communications of the ACM 46(4): 64-70.

Fægri, T. E., Dybå, T. and Dingsøyr, T. (2010). Introducing knowledge redundancy practice in software development: Experiences with job rotation in support work. Information and Software Technology 52(10): 1118-1132.

Georgakopoulos, D. and Hornick, M. (1995). An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. Distributed and Parallel Databases 3(2): 119- 153.

Glass, R. L., Ramesh, V. and Vessey, I. (2004). An analysis of research in computing disciplines. Communications of the ACM 47(6): 89-94.

Greenwood, D. and Levin, M. (1998). Introduction to action research: social research for social change. Thousand Oaks, Ca, Sage.

Groth, L. (1999). Future Organizational Design: The Scope for the IT-based Enterprise. New York, John Wiley & Sons.

Guzzo, R. A. and Dickson, M. W. (1996). Teams in organizations: Recent research on performance and effectiveness. Annual Review of Psychology 47: 307-338.

Hackman, J. R. (1986). The psychology of self-management in organizations. Psychology and work: Productivity, change, and employment. M. S. Pallack and R. O. Perloff. Washington, DC, American Psycological Association.

Hackman, J. R. (1987). The design of Work Teams. Handbook of organizational behavior. J. Lorsch. Englewood Cliffs, N. J. , Prentice-Hall.

Hansen, B., Rose, J. and Tjornehoj, G. (2004). Prescription, description, reflection: the shape of the software process improvement field. International Journal of Information Management 24(6): 457-472.

Hewitt, B. and Walz, D. (2005). Using Shared Leadership to Foster Knowledge Sharing in Information Systems Development Projects. Proceedings of the 38th Hawaii International Conference on System Sciences (HICCS): 1-5.

Hoda, R., Noble, J. and Marshall, S. (2011). Developing a grounded theory to explain the practices of self-organizing Agile teams. Empirical Software Engineering: 1-31.

Hoegl, M. and Gemuenden, H. G. (2001). Teamwork Quality and the Success of Innovative Projects: A Theoretical Concept and Empirical Evidence. Organization science 12(4): 435-449.

Hoegl, M. and Parboteeah, P. (2006). Autonomy and teamwork in innovative projects. Human resource management 45(1): 67.

Humphrey, W. S., Kitson, D. H. and Kasse, T. C. (1989). The state of software engineering practice. Proceedings of the 11th international conference on Software engineering. Pittsburgh, Pennsylvania, United States, ACM: 277-285.

Iivari, J. (1996). Why are CASE tools not used? Communications of the ACM 39(10): 94-103.

ISO (2000). ISO 9001:2000 Quality management systems -- Requirements. ISO.

Iversen, J. H., Mathiassen, L. and Nielsen, P. A. (2004). Managing risk in software process improvement: An action research approach. Mis Quarterly 28(3): 395-433.

Jorgensen, D. L. (1989). Participant Observation: A Methodology for Human Studies. Thousands Oak, California, Sage publications.

Jurison, J. (1999). Software project management: the manager's view. Communications of AIS 2.

Katzenbach, J. R. and Smith, D. K. (1993). The Discipline of Teams. Harvard Business Review 71(2): 111-120.

Kellner, M. I., Becker-Kornstaedt, U., Riddle, W. E., Tomal, J. and Verlag, M. (1998). Process Guides: Effective Guidance for Process Participants. Proceedings of the 5th International Conference on the Software Process: Computer Supported Organizational Work, Lisle, Illinois, USA: pp. 11-25.

Kirkman, B. L. and Rosen, B. (1999). Beyond self-management: Antecedents and consequences of team empowerment. Academy of Management Journal 42(1): 58-74.

Kitchenham, B., Pickard, L. and Pfleeger, S. L. (1995). Case Studies for Method and Tool Evaluation. IEEE Software 12(4): 52-62.

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.

Krasner, H. (1999). The Payoff for Software Process Improvement: What it is and How to Get it,. Elements of Software Process Assessment and Improvement. K. E. Emam and N. H. Madhavji. Los Alamitos, California, IEEE Computer Society Press: 151 - 176.

Kraut, R. E. and Streeter, L. A. (1995). Coordination in software development. Communications of the ACM 38(3): 69-81.

Krutchen, P. (2000). The Rational Unified Process: An Introduction. Massachusetts, USA, Addison-Wesley.

Langfred, C. W. (2000). The paradox of self-management: Individual and group autonomy in work groups. Journal of Organizational Behavior 21(5): 563-585.

Langfred, C. W. (2005). Autonomy and Performance in Teams: The Multilevel Moderating Effect of Task Interdependence. Journal of Management 31(4): 513-529.

Langley, A. (1999). Strategies for Theorizing from Process Data, Academy of Management. 24: 691-710.

Lant, T. K. and Mezias, S. J. (1992). An Organizational Learning Model of Convergence and Reorientation. Organization Science 3(1): 47-71.

Larman, C. and Basili, V. R. (2003). Iterative and incremental development: A brief history. Computer 36(6): 47-56.

Lawler, E. E. and Mohrman, S. A. (1987). Quality Circles - after the Honeymoon. Organizational Dynamics 15(4): 42-54.

Levesque, L. L., Wilson, J. M. and Wholey, D. R. (2001). Cognitive divergence and shared mental models in software development project teams. Journal of Organizational Behavior 22: 135-144.

Li, J., Moe, N. B. and Dybå, T. (2010). Transition from a plan-driven process to Scrum: a longitudinal case study on software quality. Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. Bolzano-Bozen, Italy, ACM: 1-10.

Lycett, M., Macredie, R. D., Patel, C. and Paul, R. J. (2003). Migrating agile methods to standardized development practice. Computer 36(6): 79-85.

Mathiassen, L., Ngwenyama, O. K. and Aaen, I. (2005). Managing change in software process improvement. IEEE Software 22(6): 84-91.

McAvoy, J. and Butler, T. (2009). The role of project management in ineffective decision making within Agile software development projects. European Journal of Information Systems 18(4): 372-383.

Meso, P. and Jain, R. (2006). Agile Software Development: Adaptive Systems Principles and Best Practices. Information Systems Management 23(3): 19-30.

Mintzberg, H. (1989). Mintzberg on Management: Inside Our Strange World of Organizations. New York: Free Press.

Moe, N. B. and Aurum, A. (2008). Understanding Decision-Making in Agile Software Development: A Case-study. Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference, Parma, Italy: 216-223.

Moe, N. B., Dingsøyr, T. and Dybå, T. (2008). Understanding Self-Organizing Teams in Agile Software Development. 19th Australian Conference on Software Engineering: 76-85.

Moe, N. B., Dingsøyr, T. and Dybå, T. (2009). Overcoming Barriers to Self-Management in Software Teams. IEEE Software 26(6): 20-26.

Moe, N. B., Dingsøyr, T. and Dybå, T. (2010). A teamwork model for understanding an agile team: A case study of a Scrum project. Information and Software Technology 52(5): 480-491.

Moe, N. B., Dingsøyr, T. and Kvangardsnes, Ø. (2009). Understanding Shared Leadership in Agile Development: A Case Study. Hawaii International Conference on System Sciences, Hawaii: 1-10.

Moe, N. B., Dingsøyr, T. and Røyrvik, E. A. (2009). Putting Agile Teamwork to the Test – An Preliminary Instrument for Empirically Assessing and Improving Agile Software Development. 10th International Conference on Agile Processes in Software Engineering and Extreme Porgramming, Sardinia, Italy: 114-124.

Moe, N. B. and Dybå, T. (2006). Improving by involving: a case study in a small software company. EuroSPI 2006, Joensuu, Finland: 159-170.

Moe, N. B. and Dybå, T. (2006). The use of an Electronic Process Guide in a medium sized Software Development Company. Software Process Improvement and Practice 11(1): 21-34.

Morgan, G. (2006). Images of Organizations. Thousand Oaks, CA, SAGE publications.

Nambisan, S. and Wilemon, D. (2000). Software development and new product development: Potentials for cross-domain knowledge sharing. IEEE Transactions on Engineering Management 47(2): 211-220.

Nerur, S. and Balijepally, V. (2007). Theoretical reflections on agile development methodologies - The traditional goal of optimization and control is making way for learning and innovation. Communications of the ACM 50(3): 79-83.

Nerur, S., Mahapatra, R. and Mangalaraj, G. (2005). Challenges of migrating to agile methodologies. Communications of the ACM 48(5): 72-78.

Okhuysen, G. A. and Bechky, B. A. (2009). Coordination in Organizations: An Integrative Perspective. Academy of Management Annals 3: 463-502.

Parnas, D. L. and Clements, P. C. (1986). A Rational Design Process: How and Why to Fake It. IEEE Transactions on Software Engineering 12(2): 251 - 257.

Paulk, M. C., Weber, C. V. and Chrissis, M. B. (1999). The Capability Maturity Model for Software. Elements of Software Process Assessment & Improvement. . K. El Emam and N. H. Madhavji. Los Alamitos, California., IEEE Computer Society: 3-22.

Pearce, C. L. (2004). The future of leadership: Combining vertical and shared leadership to transform knowledge work. Academy of Management Executive 18(1): 47-57.

Purser, R. E. and Cabana, S. (1997). Involve employees at every level of strategic planning. Quality Progress 30(5): 66-71.

Pyzdek, T. (1992). To Improve Your Process - Keep It Simple. IEEE Software 9(5): 112-113.

Qumer, A. and Henderson-Sellers, B. (2008). A framework to support the evaluation, adoption and improvement of agile methods in practice. Journal of Systems and Software 81(11): 1899-1919.

Ramesh, B., Cao, L. and Baskerville, R. (2010). Agile requirements engineering practices and challenges: an empirical study. Information Systems Journal 20(5): 449-480.

Riemenschneider, C. K., Hardgrave, B. C. and Davis, F. D. (2002). Explaining Software Developer Acceptance of Methodologies: A Comparison of Five Theoretical Models. IEEE Transactions on Software Engineering 28(12): 1135-1145.

Riordan, C. M., Vandenberg, R. J. and Richardson, H. A. (2005). Employee Involvement Climate and Organizational Effectiveness. Human Resource Management 44(4): 471 - 488.

Rogers, E. M. (1995). Diffusion of Innovations. New York, The Free Press.

Royce, W. W. (1970). Managing the development of large software systems. WESCON, IEEE

Salo, O. and Abrahamsson, P. (2005). Integrating agile software development and software process improvement: a longitudinal case study. International Symposium on Empirical Software Engineering (ISESE). Noosa Heads, Australia, IEEE: 187-196.

Salo, O. and Abrahamsson, P. (2007). An iterative improvement process for agile software development. Software Process: Improvement and Practice 12(1): 81-100.

Sapsed, J., Bessant, J., Partington, D., Tranfield, D. and Young, M. (2002). Teamworking and knowledge management: a review of converging themes. International Journal of Management Reviews 4(1): 71-85.

Sawyer, S. (2004). Software development teams. Commun. ACM 47(12): 95-99.

Schneider, K., von Hunnius, J. P. and Basili, V. R. (2002). Experience in implementing a learning software organization. IEEE Software 19(3): 46-49.

Schwaber, K. and Beedle (2001). Agile Software Development with Scrum, Upper Saddle River: Prentice Hall.

Scott, L., Carvalho, L., Jeffery, R., D'Ambra, J. and Becker-Koernstaedt, U. (2002). Understanding the use of an electronic process guide. Information and Software Technology 44(10): 601 - 616.

Scott, L., Carvalho, L., Jeffery, R., D'Ambra, J. and Becker-Kornstaedt, U. (2002). Understanding the use of an electronic process guide. Information and Software Technology 44(10): 601-616.

SEI (2002). Capability Maturity Model ® Integration (CMMISM), Version 1.1. Carnegie Mellon Software Engineering Institute.

Sharp, H. and Robinson, H. (2004). An ethnographic study of XP practice. Empirical Software Engineering 9(4): 353-375.

Smite, D., Moe, N. B. and Ågerfalk, P. J. (2010). Agility Across Time and Space: Implementing Agile Methods in Global Software Projects. Berlin, Heidelberg, Springer-Verlag.

Strauss, A. and Corbin, J. (1998). Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory. Thousand Oaks, CA, Sage Publications.

Susman, G. and Evered, R. (1978). An assessment of the scientific merits of action research. Administrative Science Quarterly 23(4): 582-603.

Takeuchi, H. and Nonaka, I. (1986). The New New Product Development Game. Harvard Business Review(64): 137-146

Tata, J. and Prasad, S. (2004). Team Self-management, Organizational Structure, and Judgments of Team Effectiveness. Journal of Managerial Issues Vol. 16 (Issue 2): p248 - 265.

Tenenberg, J. (2008). An institutional analysis of software teams. Int. J. Hum.-Comput. Stud. 66(7): 484-494.

Thomas, M. and McGarry, F. (1994). Top-Down vs. Bottom-Up Process Improvement. IEEE Software 11(4): 12-13.

Trist, E. (1981). The evolution of socio-technical systems: a conceptual framework and an action research program. Occasional paper No 2. Toronto, Ontario, Ontario Quality of Working Life Centre.

Trist, E. and Bamforth, K. W. (1951). Some Social and Psychological Consequences of the Longwall Method of Coal-Getting. Human Relations 4(1): 3-38.

Uhl-Bien, M. and Graen, G. B. (1998). Individual self-management: Analysis of professionals' self-managing activities in functional and cross-functional work teams. Academy of Management Journal 41(3): 340-350.

Van Mierlo, H., Rutte, C. G., Kompier, M. A. J. and Doorewaard, H. (2005). Self-managing teamwork and psychological well-being: Review of a multilevel research domain. Group & Organization Management 30(2): 211-235.

van Solingen, R., Berghout, E., Kusters, R. and Trienekens, J. (2000). From process improvement to people improvement: enabling learning in software development. Information and Software Technology 42(14): 965-971.

Venkatesh, V. and Davis, F. D. (2000). A theoretical extension of the Technology Acceptance Model: Four longitudinal field studies. Management Science 46(2): 186-204.

Vinekar, V., Slinkman, C. W. and Nerur, S. (2006). Can Agile and Traditional Systems Development Approaches Coexist? An Ambidextrous View. Information Systems Management 23(3): 31 - 42.

Walz, D. B., Elam, J. J. and Curtis, B. (1993). Inside a software design team: knowledge acquisition, sharing, and integration. Communications of the ACM 36(10): 63-77.

Wenger, E. (1998). Communities of practice: learning, meaning and identity. Cambridge, UK, Cambridge University Press.

Williams, L. and Cockburn, A. (2003). Agile software development: it's about feedback and change. Computer 36(6): 39-43.

Yamamura, G. (1999). Process improvement satisfies employees. IEEE Software 16(5): 83-85.

Yin, R. K. (2002). Case study research: design and methods. Thousand Oaks, Calif., Sage.
Zazworka, N., Stapel, K., Knauss, E., Shull, F., Basili, V. R. and Schneider, K. (2010). Are developers complying with the process: an XP study. Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. Bolzano-Bozen, Italy, ACM: 1-10.
Ågerfalk, P. J. and Fitzgerald, B. (2006). Flexible and distributed software processes: Old petunias in new bowls? Communications of the ACM 49(10): 26-34.

PART II – Included publications

Paper 1

Dybå, T., Moe, N. B. and Mikkelsen, E. M. (2004). An Empirical Investigation on Factors Affecting Software Developer Acceptance and Utilization of Electronic Process Guides. Proceedings of the International Software Metrics Symposium (METRICS), Chicago, Illinois, USA, 220–231.

Paper 2

Dybå, T., Moe, N. B. and Arisholm, E. (2005). Measuring Software Methodology Usage: Challenges of Conceptualization and Operationalization. Fourth International Symposium on Empirical Software Engineering (ISESE), Noosa Heads, Australia, IEEE Computer Society, 447 - 457.

Paper 3

Moe, N. B. and Dybå, T. (2006). The use of an Electronic Process Guide in a medium sized Software Development Company. Software Process Improvement and Practice 11(1): 21-34.

Paper 4

Moe, N. B. and Dyba, T. (2006). Improving by involving: a case study in a small software company. EuroSPI 2006, Joensuu, Finland, 158 – 169.

Paper 5

Dingsøyr, T. and Moe, N. B. (2008). The Impact of Employee Participation on the Use of an Electronic Process Guide: A Longitudinal Case Study. IEEE Trans. Softw. Eng. 34(2): 212-225.

Paper 6

Moe, N. B., Dingsøyr, T. and Dybå, T. (2008). Understanding Self-Organizing Teams in Agile Software Development. 19th Australian Conference on Software Engineering 76-85.

Paper 7

Moe, N. B. and Aurum, A. (2008). Understanding Decision-Making in Agile Software Development: A Case-study. Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference, Parma, Italy, 216-223

Paper 8

Moe, N. B., Dingsøyr, T. and Røyrvik, E. A. (2009). Putting Agile Teamwork to the Test – An Preliminary Instrument for Empirically Assessing and Improving Agile Software Development. 10th International Conference on Agile Processes in Software Engineering and Extreme Programming, Sardinia, Italy, 114-123.

Paper 9

Moe, N. B., Dingsøyr, T. and Kvangardsnes, Ø. (2009). Understanding Shared Leadership in Agile Development: A Case Study. Hawaii International Conference on System Sciences, Hawaii, 1-10

Paper 10

Moe, N. B., Dingsøyr, T. and Dybå, T. (2009). Overcoming Barriers to Self-Management in Software Teams. IEEE Software 26(6): 20-26.

Paper 11

Li, J., Moe, N. B. and Dybå, T. (2010). Transition from a plan-driven process to Scrum: a longitudinal case study on software quality. Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. Bolzano-Bozen, Italy, ACM: 1-10.

Paper 12

Moe, N. B., Dingsøyr, T. and Dybå, T. (2010). A teamwork model for understanding an agile team: A case study of a Scrum project. Information and Software Technology 52(5): 480-491.

Information and Software Technology 52 (2010) 480–491

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A teamwork model for understanding an agile team: A case study of a Scrum project

Nils Brede Moe *, Torgeir Dingsøyr, Tore Dybå

SINTEF, NO-7465 Trondheim, Norway

ARTICLE INFO

Article history:
Received 30 July 2009
Received in revised form 26 October 2009
Accepted 12 November 2009
Available online 20 November 2009

Keywords:
Agile software development
Scrum
Software engineering
Teamwork
Empirical software engineering
Case study

ABSTRACT

Context: Software development depends significantly on team performance, as does any process that involves human interaction.
Objective: Most current development methods argue that teams should self-manage. Our objective is thus to provide a better understanding of the nature of self-managing agile teams, and the teamwork challenges that arise when introducing such teams.
Method: We conducted extensive fieldwork for 9 months in a software development company that introduced Scrum. We focused on the human sensemaking, on how mechanisms of teamwork were understood by the people involved.
Results: We describe a project through Dickinson and McIntyre's teamwork model, focusing on the interrelations between essential teamwork components. Problems with team orientation, team leadership and coordination in addition to highly specialized skills and corresponding division of work were important barriers for achieving team effectiveness.
Conclusion: Transitioning from individual work to self-managing teams requires a reorientation not only by developers but also by management. This transition takes time and resources, but should not be neglected. In addition to Dickinson and McIntyre's teamwork components, we found trust and shared mental models to be of fundamental importance.

1. Introduction

Software development depends significantly on team performance, as does any process that involves human interaction. A common definition of a team is "a small number of people with complementary skills who are committed to a common purpose, set of performance goals, and approach for which they hold themselves mutually accountable" [22].

The traditional perspective on software development is rooted in the rationalistic paradigm, which promotes a plan-driven product-line approach to software development using a standardized, controllable, and predictable software engineering process [15]. Today, this traditional mechanistic worldview is challenged by the agile perspective that accords primacy to uniqueness, ambiguity, complexity, and change, as opposed to prediction, verifiability, and control. The goal of optimization is being replaced by those of flexibility and responsiveness [33].

Setting up a work team is usually motivated by benefits such as increased productivity, innovation, and employee satisfaction. Research on software development teams has found that team performance is linked with the effectiveness of teamwork coordination [19,25]. In the traditional plan-driven approach, work is coordinated in a hierarchy that involves a command-and-control style of management in which there is a clear separation of roles [33,34]. In the agile approach, work is coordinated by the self-managing team, in which the team itself decides how work is coordinated [8].

A team that follows a plan-driven model often consists of independently focused self-managing professionals, and a transition to self-managing teams is one of the biggest challenges when introducing agile (change-driven) development [33]. Neither culture nor mind-sets of people can be changed easily, which makes the move to agile methodologies all the more formidable for many organizations [8]. In addition, it is not sufficient to put individuals together in a group, tag them "self-managing", and expect that they will automatically know how to coordinate and work effectively as an agile team.

Our objective is to provide a better understanding of the nature of self-managing agile teams, which can in turn benefit the effective application of agile methods in software development. To this end, we conducted a longitudinal study that draws on the general literature of teamwork and self-managing teams. Such a study can provide valuable insights for understanding the challenge of introducing the self-managing agile team. We sought to answer the following research question:

How can we explain the teamwork challenges that arise when introducing a self-managing agile team?

* Corresponding author. Tel.: +47 93028687; fax: +47 73592977.
E-mail addresses: nilsm@sintef.no (N.B. Moe), torgeird@sintef.no (T. Dingsøyr), tored@sintef.no (T. Dybå).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.11.004

The remainder of this paper is organized as follows: Section 2 gives an overview of the literature on teamwork and agile software development. Section 3 describes our research question and method in detail. Section 4 presents results from a nine-month fieldwork of teamwork in a Scrum team. Section 5 contains a discussion of the findings. Section 6 concludes and provides suggestions for further work.

2. Background: teamwork and agile software development

In this section, we give a short introduction to the field of teamwork, teamwork in agile development, and the teamwork model that is used as the basis for our work.

2.1. Teamwork

The topic of teamwork has attracted research from several disciplines [10,17,41]. The concept of teamwork carries with it a set of values that encourage listening and responding constructively to views expressed by others, giving others the benefit of the doubt, providing support, and recognizing the interests and achievements of others [22]. Such values are important because they promote individual performance, which boosts team performance, and they help teams to perform well as a group, and good team performance boosts the performance of the organization.

Research on teamwork includes the development of tests to identify personality characteristics, because it has often been argued that good teams need a certain blend of personalities. Examples are the Belbin test [7] and the Myers–Briggs Type indicator. There is also a great deal of research on climate at work group and team level. The most studied model of team climate is that of [48] who suggests that four climate factors (vision, participative safety, task orientation, and support for innovation) are essential for team innovation to occur.

Furthermore, there are studies of teams over time, which indicate that teams go through set phases. The most well-known of these studies are those of Tuckman [46], who identified the phases as forming, storming, norming, and performing. Other studies have focused on the relationships between team members and argue that group cohesiveness is important for team success (cited in [41]). However, the use of teams does not always result in success for the organization [17]. Team performance is complex, and the actual performance of a team depends not only on the competence of the team itself in managing and executing its work, but also on the organizational context provided by management.

Much research has been devoted to what is described as self-managing, autonomous, or empowered teams [17,23,26,45,47]. One of the reasons that the use of self-managing teams has become popular is that some research suggests that their use promotes more satisfied employees, lower turnover, and lower absenteeism [10]. Others also claim that self-managing teams are a prerequisite for the success of innovative projects [20,44].

Although the majority of studies report that using self-managing teams has positive effects, some studies offer a more mixed assessment; such teams can be difficult to implement, and they risk failure when used in inappropriate situations or without sufficient leadership and support [18]. In addition, research on team performance indicates that the effects of autonomous work groups are highly situational dependent and that the effects of autonomous work-group practices depend on such factors as the nature of the workforce and the nature of the organization [10,17]. Further, autonomy on the individual level may conflict with autonomy on the group level. When a team as a whole is given a great deal of autonomy, it does not follow that the individual team members are given high levels of individual autonomy. Barker [5], for example, pointed out that self-managing groups may end up controlling group members more rigidly than they do under traditional management styles, while Markham and Markham [29] suggested that it may be difficult to incorporate both individual autonomy and group autonomy in the same work group. For Individuals to be motivated and satisfied with their jobs they need to have control over their own work and over the scheduling and implementation of their own tasks [1,26].

2.2. Teamwork in agile development: the Scrum team

In a software team, the members are jointly responsible for the end product and must develop shared mental models by negotiating shared understandings about both the teamwork and the task [28]. Project goals, system requirements, project plans, project risks, individual responsibilities, and project status must be visible and understood by all parties involved [21].

Most current development methods have it as a premise that software teams should self-organize or self-manage [36,42]. Scrum, which is a project-management-oriented agile development method, was inspired by a range of fields, such as complexity theory, system dynamics, and Nonaka and Takeuchi's theory of knowledge creation [35], and has adapted aspects of these fields to a setting of software development. Self-management is a defining characteristic in Scrum. Compared with traditional command-and-control oriented management, Scrum represents a radically new approach for planning and managing software projects, because it brings decision-making authority to the level of operational problems and uncertainties.

Rising and Janoff [36] describe Scrum as a development process for small teams, which includes a series of short development phases or iterations ("sprints"). A Scrum team is given significant authority and responsibility for many aspects of their work, such as planning, scheduling, assigning tasks to members, and making decisions: "The team is accorded full authority to do whatever it decides is necessary to achieve the goal" [43].

However, despite the popularity of the method, a systematic review of empirical studies of agile development [16] found only one case study of Scrum in the research literature prior to 2006.

2.3. Dickinson and McIntyre's teamwork model

The issue of what processes and components comprise teamwork and how teamwork contributes to team effectiveness and team performance has been much studied [9,19,26,30,39], but there is no consensus concerning its conceptual structure [38]. Salas et al. [40] identify 136 different models in their literature review and present a representative sample of 11 models and frameworks.

Using recent research and previous reviews, Dickinson and McIntyre [13] identified and defined seven core components of teamwork. Using these components and their relationships as a basis, they proposed the teamwork model that is used in this work. The model consists of a learning loop of the following basic teamwork components: communication, team orientation, team leadership, monitoring, feedback, backup, and coordination (Fig. 1).

We selected the Dickinson and McIntyre teamwork model for the following reasons:

1. It includes the most common elements that are considered in most research on teamwork processes [38,39]. In addition, it considers important elements that are required in self-managed teams: team orientation, functional redundancy and backup behavior [32,35], communication, feedback and learning [33], and shared leadership [22]. Further, the model covers important

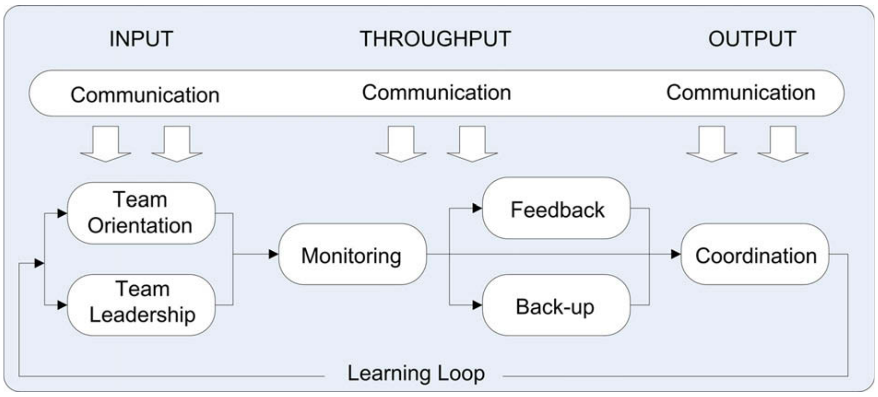

Fig. 1. The Dickinson and McIntyre teamwork model [13].

elements that are found in software teams, such as coordination of work [25].

2. It specifies what teamwork skills should be observed, in that the model is presented with a conceptual framework for developing measures of teamwork performance that can ensure effective individual and team performance [13], pp. 22.
3. It considers the teamwork process as a learning loop in which teams are characterized as adaptable and dynamically changing over time. Continuous self-management requires a capacity for double-loop learning that allows operating norms and rules to change along with transformation in the wider environment [32].

Each component of the model is explained in Table 1. According to Dickinson and McIntyre, team leadership and team orientation are 'input' components of teamwork because at least one of these attitudes is required for an individual to participate in a team task. Team leadership can be shown by several team members that is also a prerequisite for a team's being self-managing. In such teams, team members should share the authority to make decisions, rather than having: (a) a centralized decision structure in which one person (e.g. the team leader) makes all the decisions or (b) a decentralized decision structure in which all team members make decisions regarding their work individually and independently of other team members [20]. So, while the traditional perspective of a single leader suggests that the leadership function is a specialized role that cannot be shared without jeopardizing group effectiveness, when leadership is shared, group effectiveness is achieved by empowering the members of the team to share the tasks and responsibilities of leadership [22].

In the Dickinson and McIntyre model, the components of monitoring, feedback, and backup are the intermediate processes for ensuring effective teamwork. Finally, the 'output' component is coordination because it defines the performance of the team. Communication is a transversal component of particular importance, because it links the other components. To build software effectively, there is a need for tight coordination among the various efforts involved so that the work is completed and fits together [25].

Table 1
The Dickinson and McIntyre teamwork model: definitions of teamwork components.

Team orientation: Refers to the team tasks and the attitudes that team members have towards one another. It reflects an acceptance of team norms, the level of group cohesiveness, and the importance of team membership, e.g.
• assigning high priority to team goals
• participating willingly in all relevant aspects of the team
Team leadership: Involves providing direction, structure, and support for other team members. It does not necessarily refer to a single individual with formal authority over others. Team leadership can be shown by several team members, e.g.
• explaining to other team members exactly what is needed from them during an assignment
• listening to the concerns of other team members
Monitoring: Refers to observing the activities and performance of other team members and recognizing when a team member performs correctly. It implies that team members are individually competent and that they may subsequently provide feedback and backup, e.g.
• being aware of other team members' performance
• recognizing when a team member performs correctly
Feedback: Involves the giving, seeking, and receiving of information among team members. Giving feedback refers to providing information regarding other members' performance. Seeking feedback refers to requesting input or guidance regarding performance and to accepting positive and negative information regarding performance, e.g.
• responding to other members' requests for information about their performance
• accepting time-saving suggestions offered by other team members
Backup: Involves being available to assist other team members. This implies that members have an understanding of other members' tasks. It also implies that team members are willing and able to provide and seek assistance when needed, e.g.
• filling in for another member who is unable to perform the task
• helping another member correct a mistake
Coordination: Refers to team members executing their activities in a timely and integrated manner. It implies that the performance of some team members influences the performance of others. This may involve an exchange of information that subsequently influences another member's performance. Coordination represents the output of the model and reflects the execution of team activities such that members respond as a function of the behavior of others, e.g.
• passing performance-relevant data to other members in an efficient manner
• facilitating the performance of other members' jobs
Communication: Involves the exchange of information between two or more team members in the prescribed manner and using appropriate terminology. Often, the purpose of communication is to clarify or acknowledge the receipt of information, e.g.
• verifying information prior to making a report
• acknowledging and repeating messages to ensure understanding

Table 2
The use of Klein and Myers' principles in this field research.

The principles for interpretive field research [24]	How we used each principle
1. The fundamental principle of the hermeneutic circle	We improved our understanding of the project by moving back and forth between phases and events. The project had three main phases, which had different teamwork characteristics. For each of the phases, we described concrete events. The data analysis involved multiple researchers having ongoing discussions about the findings
2. The principle of contextualization	To clarify for our readers how situations emerged, we describe the work and organization of the company, as well as the context of the project we used to study teamwork
3. The principle of interaction between researchers and subjects	The researchers' understanding of the project developed through observations, interviews and discussions with the team participants in the coffee breaks and during lunch. We discussed project status, progress, and how issues were perceived by team participants
4. The principle of abstraction and generalization	We describe our findings and relate them to the model of Dickinson and McIntyre [13]
5. The principle of dialogical reasoning	We use Dickinson and McIntyre's model to identify areas of investigation in the case. Our assumptions are also based on the general literature of teamwork and self-management Our social background is European
6. The principle of multiple interpretations	To collect multiple, and possibly contradictory interpretations of events we collected data from all participants in the project and from multiple data sources. The case study narrative and findings have been presented to Alpha and led to feedback
7. The principle of suspicion	By means of the analysis, we were sensitive to how roles and personalities affected attitudes to teamwork to discover false preconceptions In addition to observations, we also performed interviews with different roles at different levels, and multiple interviews with all team members. This increased the chance of unveiling possibly incorrect or incomplete meanings

In the rest of the paper, we will explain the challenges that arise when introducing agile methods by appeal to the mechanisms that influence teamwork that are suggested by Dickinson and McIntyre [13].

3. Research method

We designed a single-case holistic study [49] of a project that used Scrum, focusing on mechanisms that influence teamwork. When designing the study, we focused on human sensemaking and on how the mechanisms of teamwork were understood by the people involved. Given that our study was an interpretative field study, we used the seven principles for conducting such studies that were proposed by Klein and Myers [24] in order to determine the main choices that were related to research method. Table 2 gives an overview of these principles and a description of how we used them.

3.1. Study context

The field study was conducted in a company that introduced Scrum in order to improve their ability to deliver iteratively and on time, increase quality, and improve teamwork. The company has three regional divisions with one separate ICT division. The ICT division consists of a consulting department, an IT management department, and a development department. The ICT division develops and maintains a series of off-the-shelf software products that are developed in-house, in addition to software development projects for outside customers. During the study, the development department had 16 employees, divided into a Java and a .Net group.

The goal of the project studied was to develop a plan and coordination system for owners of cables (e.g. electricity, fiber) and pipes (water, sewer). We refer to the project as "Alpha", because this was the first project for which the company used agile methods, in this case Scrum. Alpha produced a combination of textual user interfaces and map functionality. Alpha was to use a commercial package for the map functionality, which was to be customized by a well-known subcontractor located in another city. The subcontractor could only deliver their part of the system 4 weeks before the first deliverable to the customer. This was recognized as a risk, but it was decided that it would be even riskier to develop this component internally. The company would also be responsible for maintenance and support after final installation. Four thousand hours, six developers, one Scrum master, and a product owner were allocated to the project. The product owner was employed by the same company as the developers and acted as a representative for the client, which was the local government of a Norwegian city. Internally, there were plans for reusing deliveries from Alpha and to re-sell the product to other public departments when it was finished. An extra 800 h were allocated to achieve this aim. Before Alpha was begun in May 2006, some initial architectural work was done and some coding activities had started. Alpha used .Net technology and was supposed to last for 10 months.

The developers had usually worked alone on projects divided into modules or on smaller projects, so Alpha was the first experience of working on a larger project for most.

3.2. Data sources and analysis

The two first authors conducted direct observation and collected documents throughout the whole project. In addition, we interviewed the Scrum master, product owner, and developers (Table 3). The interview guide covered the components in the Dickinson and McIntyre model in addition to questions related to Scrum (Appendix A).

Table 3
Data sources.

Source	Comment
Observations and informal dialogues	Participant observation and observation of the daily stand-up, sprint and planning meetings, sprint reviews, and sprint retrospective as well as other meetings. The 60 observations and informal dialogues were documented in field notes, which also include pictures
Interviews	We interviewed the Scrum master and three developers in June 2006, five of the developers in September 2006, and all developers, the Scrum master, and the product owner after the project was completed (March 2007). The 17 interviews were all transcribed
Documents	Product backlog, sprint backlogs, burn-down charts, minutes from review, retrospective and planning meetings

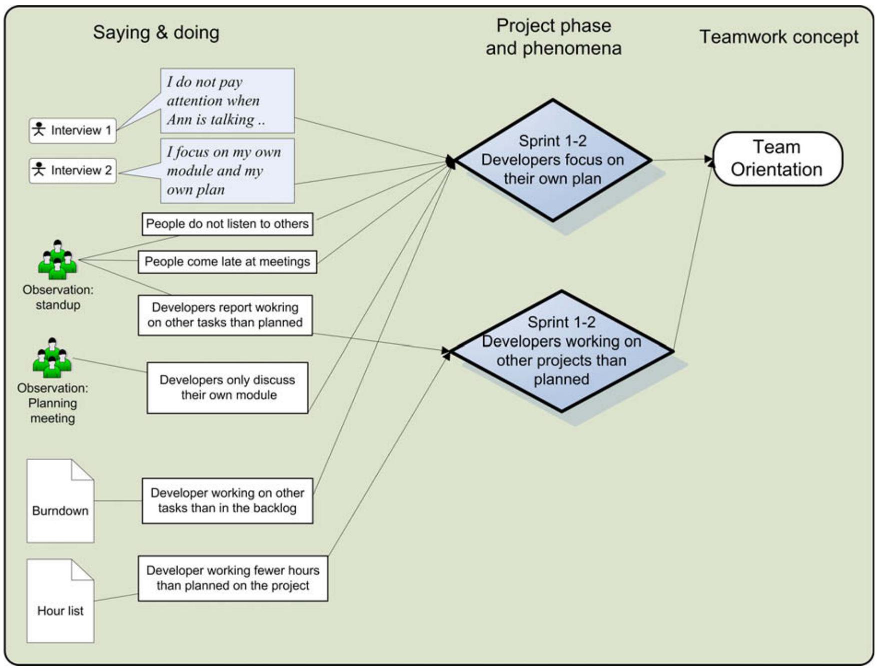
Saying & doing
Project phase and phenomena
Teamwork concept
Interview 1
I do not pay attention when Ann is talking ..
Interview 2
I focus on my own module and my own plan
People do not listen to others
People come late at meetings
Observation: standup
Developers report wokring on other tasks than planned
Observation: Planning meeting
Developers only discuss their own module
Burndown
Developer working on other tasks than in the backlog
Hour list
Developer working fewer hours than planned on the project
Sprint 1-2 Developers focus on their own plan
Sprint 1-2 Developers working on other projects than planned
Team Orientation

Fig. 2. Overview of the coding process. Example material from the concept "team orientation".

We visited the team once or twice a week, conducting a total of 60 observations, each of which lasted from 10 min to 8 h. We observed project meetings and developers working. We often discussed Alpha's status and progress, and how team participants perceived issues during their coffee breaks and lunch. Notes were taken on dialogues, interactions, and activities. The dialogues were transcribed and integrated with notes to produce a detailed record of each session. We also collected Scrum artifacts, such as product backlogs, sprint backlogs, and burn-down charts. All data from the interviews, observations, and documents were imported into a tool for analyzing qualitative data, Nvivo (www.qsrinternational.com). We categorized interesting expressions, observations, and text from documents, using the teamwork concepts proposed by Dickinson and McIntyre as the main categories.

We used a variety of strategies to analyze the material [27]. First, we described the project and context in a narrative to achieve an understanding of what was going on in the project. Then, we described aspects of teamwork using Dickinson and McIntyre's model by pointing to events in three main phases of the project, which had different teamwork characteristics.

In the analysis, we emphasized how events were interpreted by different participants in the project. Material to describe an event was taken across all sources and synthesized, as shown in the example in Fig. 2.

4. Results: teamwork in an agile project

The team that worked on Alpha organized the project according to generally recommended Scrum practices. Plans were made at the beginning of each sprint, after the team had reviewed what was produced in the previous sprint. Features were recorded in the sprint backlog. The team that worked on Alpha held three project retrospectives to identify and discuss problems and opportunities that arose during the development process. Daily meetings were organized throughout the project, though these were less frequent in the last two sprints. These meetings were usually about updating the others on progress, development issues, and the project in general. The daily meetings we observed lasted from 10 to 35 min, but were usually shorter than 15 min. The product owner, who was situated in another city, often participated in these meetings by telephone. He participated because both he and the Scrum master thought that it was important to share information constantly and participate in the decision-making process.

Alpha began in May 2006, with the first installation planned for October and the final installation for November 2006. However, the first installation was not approved until December 2006 and from January 2007, two developers continued working with change requests until the final installation was approved in October 2007. Five of the sprints lasted 1 month, the sprint during summer for two. Fig. 3 shows major events in the project together with a project-participant satisfaction graph. This figure was created by the team in the final project retrospective and was based on a timeline exercise [12]. To create the project-participant satisfaction graph, each team member first drew his own graph for the emotional ups and downs during the project, after which the graphs were merged.

In the initial planning phase, before coding began, several meetings were used to discuss the overall architecture, and decide on the technology and development platform. As can be seen from Fig. 3, the team was frustrated in this period, because of what the team described as "endless discussion without getting anywhere". After Scrum was introduced and code writing began, the team was more satisfied with Alpha. In the first retrospective, the team itself concluded that the team members were taking responsibility, that they were dedicated to the project, and that the team was protected against external issues. Meetings and work were perceived as well-coordinated. During the first retrospective, a developer said:

> Earlier we worked more alone, and when you got a project doomed to failure, you would get a lot of negative response. That was unpleasant. Now we share both the risk and opportunities.

The team was satisfied with their performance in sprints 1–4. However, in sprint 5, problems with integrating a deliverable from the subcontractor emerged, which resulted in the two last sprints being chaotic and the project being delayed. During this period, we saw many empty pizza boxes in the office space, which

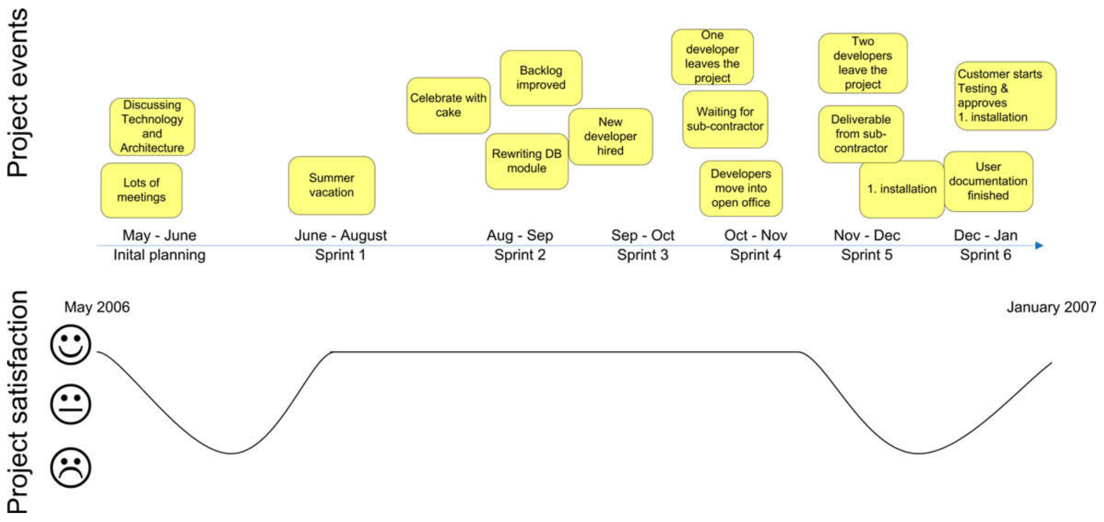

Fig. 3. Main events in the project and project satisfaction.

indicated that the developers were working late. Developers told us they also worked at weekends. The team became less satisfied, as shown in Fig. 3.

However, after the client had approved the first installation, the team became more satisfied. In the last retrospective, the team described the project as a good one, except for the problems related to the deliverable from the subcontractor. Then again, the team saw this as something beyond their control.

Despite the teams' overall satisfaction with the teamwork, throughout the project we observed problems with completing the backlog and following the sprint plan, unproductive meetings, developers often being silent in the planning meetings, and developers often reporting working on issues other than those that it had been initially planned to work on. In addition, the developers received little feedback when talking about what they were doing. In what follows, we will use the data we collected to try to explain some of our observations.

4.1. Introducing Scrum: sprints 1–2

The project leader participated in a Scrum master certification course. The first sprint was initiated with a two-day Scrum course. The first day was spent on introducing Scrum to the whole development department, the second on planning the first sprint for Alpha.

The first sprint completed most of the backlog, and the team was satisfied with the progress. However, in the first retrospective (see Fig. 4), the team reported problems with both defining a stable sprint backlog and finishing it. We observed these problems as well. The team also ended up working on tasks that were not discussed or identified during the sprint planning meeting.

In this company, each team member is usually assigned to work on a specific software module from the beginning to the end. This way of working is known as an isomorphic team structure [21]. The advantages with this structure are that it is organizationally

Fig. 4. From the review and retrospective meeting in sprint 2.

simple, it allows many tasks to be completed in parallel, and task responsibilities can be clearly defined and understood. The Scrum master subscribed to this view [interview]:

> Let the person who knows most about the task perform it! We cannot afford several people doing the same thing in this project. We need to continue working as we have done before.

The team mostly kept this structure after introducing Scrum. A developer said [interview]:

> Because we have to deliver every month, there is never time to swap tasks.

Because of the division of work, the developers typically created their own plan for their own module, often without discussing it with the team. A developer commented [interview]:

> Some are more motivated by the perfect technical solution, than thinking of when things need to be done.

In this phase, one developer even implemented features for future projects, without informing the others (this kind of behavior is often referred to as decision hijacking [3]). This was discussed in a daily stand-up of the second sprint:

> Developer: The customer databases will be used by several applications, so I have implemented support for dealing with various technologies, including Oracle. It took a lot of time.
> Scrum master: Did we not agree on postponing this?
> Developer: We need this later and now it is done.

This illustrates how developers prioritized individual goals over team goals, and subsequently a lack of *team orientation*. As a result of this incident, the Scrum master lost trust in this developer and started to supervise him. Consequently, the developer was not part of the *team leadership* any more, even when discussing modules where he was seen as the expert. We observed that he was sometimes absent from the daily meetings. This is consistent with findings from Bandow such that if team members do not feel that their input is valued, they may be less willing to share information [4].

The Scrum master also observed that the team was not reporting problems. In interviews, we found that the developers thought that the Scrum master was overreacting to problems stated at the daily meetings, which resulted in the team not reporting problems when the Scrum master was present. After the Scrum master confronted the team with this issue, the situation improved. However, for the rest of the project, the Scrum master still felt that problems were reported too late. This was confirmed by our observations of daily stand-ups.

In the second retrospective of Alpha, we found two more reasons for problems not being reported: problems were discovered late and they were seen as personal. One developer said:

> People working alone results in the team not discovering problems, because you do not get feedback on your work.

Because of the isomorphic team structure, the developers perceived new emerging tasks and new problems as personal; as a result, they did not seek assistance when needed. They focused on their own modules, which resulted in problems with *monitoring* each other and subsequently with giving *feedback* and implementing *backup behavior*. In the second retrospective meeting, one developer said:

> When we discover new problems, we feel we own them ourselves, and that we will manage to solve them before the next meeting tomorrow. But this is not the case, it always takes longer.

When individuals are independent and have more control over their schedule and the implementation of their tasks, there is less interaction between the group members [26]. One developer said [Retrospective sprint 2]:

> When it comes to the daily scrum, I do not pay attention when Ann is talking. For me, what she talks about is a bit far off the topic and I cannot stay focused. She talks about the things she is working on. I guess this situation is not good for the project.

In Alpha, this resulted in problems with *communication*, giving *feedback*, and the possibility of *monitoring* teammates. In addition, *team orientation* was hindered, because information sharing and *feedback* was delayed by people not listening. It seemed that the isomorphic team structure resulted in individual goals being seen as more important than team goals.

In this phase, the team spent more than 100 h rewriting a module. The developer responsible for the module said [interview]:

> I was supposed to create a database that every project could use. After I had created it, I explained how it was done during a stand-up, and then I went on vacation. Later, when they started using it, they did not understand how it was supposed to be used, and they decided to rewrite the whole module. The team had probably not understood what I was talking about when I explained the database in the daily stand-up. If I had not gone on vacation they would not have needed to do the rewriting... another problem is the daily meeting. It's only a short debrief, there is never time to discuss what you are working on.

The developer did not verify that the team had understood how he had implemented the module (*communication*), and no one gave *feedback* to the effect that they did not understand how the module was implemented during the stand-up. In addition to missing *monitoring*, the lack of communication and feedback was the reason for the rewriting, and the consequence was reduced progress and team efficiency.

In the second retrospective, the team concluded [retrospective report]:

> The team must work more on the same tasks, and then no one will sit alone. Working alone results in knowledge not being disseminated, and there is no backup. Also, problems are being discovered late and developers not getting feedback on their work.

4.2. Everyday work: sprints 3–4

When the team was getting used to planning and conducting sprints, work was perceived as motivating and developers expressed satisfaction at having something completed early in the project. However, in this phase, many expressed having problems in transferring what was written in the sprint backlog to actual work tasks. Initially, the team viewed this difficulty as being caused by the introduction of "features" rather than the specification of technical requirements. However, the difficulty seemed to run deeper. One developer expressed:

> I have the impression that the sprint backlog has been somewhat distanced from what we have really planned to do.

> Another stated:

> It is really difficult to get answers to questions, because no-one really knows where we are going.

This indicates that the team lacked a clear idea of how to achieve the final result. Applying two different visions for Alpha, one covering the Alpha project and the other future projects, did not help this situation and weakened the *team orientation*. *Team leadership* was already weak and was seldom shown by team members other than the Scrum master and the product owner. The product owner described the challenge with giving clear direction, support, and structure to the developers [interview]:

> You need to give an answer according to what you think, and sometimes I'm not certain I'm giving the right answer.... and you need to give a quick answer; otherwise the developers will start doing something else.

The team discussions, *communication*, and *feedback* improved in this period. One developer said [interview]:

> Using Scrum forces us to work closer with each other, and the result is more communication.
>
> Another said:
>
> The good thing about Scrum is that Scrum reminds us to talk to each other about the project.

However, often, discussions ended without conclusion. One developer said [interview]:

> When we discuss technical issues, it often ends in a kind of "religious" discussion, and then I give up. And then you let people continue to do what they are doing.

This shows a challenge with respect to *team leadership*. The person leading the discussion does not listen to the concerns of other team members.

In this period, we observed the structure of the stand-up proposed by Scrum being followed. However there was little *communication, coordination*, and *feedback* between the developers in these meetings. One developer said after a stand-up:

> The daily meetings are mostly about reporting to the Scrum master. When he is not there, the meetings are better because then we communicate with each other.
>
> Another developer said:
>
> When he is in the meeting we often end up only giving a brief report about status and not the issues we need to talk about.

Without a clear understanding of the system being developed, planning was difficult. In addition, the monthly planning meetings somehow excluded the developers and turned out to constitute *communication* only between the Scrum master and the product owner. During the retrospective, the team identified a need to spend more time planning, but two of the developers whom we observed being silent in the planning meeting thought that they spent too much time on planning. Nevertheless, a lack of thorough discussion was probably one reason for important tasks sometimes not being identified before the end of each sprint. This reduced the validity of the common backlog, did not strengthen the *communication, coordination* of tasks or the possibility of giving *feedback*, and again resulted in the developers focusing more on their own plan, thereby weakening the *team orientation*.

Another reason for the developers performing tasks other than those identified in the planning meeting was the need to adapt to the constantly changing environment. The high complexity of the project and open issues regarding technology, client, and subcontractor resulted in a high level of uncertainty when creating the sprint backlog. We observed the team being sensitive to changes in both the internal and external environment. However, the team did not manage to update the plan to adapt to the changing conditions. Subsequently, it was unclear how much progress had been made, which made it difficult to *monitor* team members' performance.

Despite the lack of *monitoring*, the developers did sometimes look at each other's code. One developer described [interview] the difficulty of giving *feedback* and raised the issue of trust regarding this matter:

> You look at someone's code, and then you think, that was a strange way of doing it. There is no problem getting criticism from people you feel safe with, but when you get feedback from people you do not like, it is different. It is also difficult to give feedback when you are not 100% sure you know that your way of doing it is better.

4.3. Emergency Scrum: sprints 5–6

The major event in this phase was that the deliverable from the subcontractor was delayed, and when it was delivered it was found not to work as intended. The code was unstable and the response time was too long. This came as a surprise to the team. One developer said [interview]:

> This was a shock to us. The end users could not start testing and we had to spend a lot of time trying to fix this. It took almost a month to locate the problems.

Given that the developers were specialized, only two developers worked on this problem, even at weekends. One developer explained [interview]:

> It's chaotic now. We work long hours, but I do not do too much. I have done what I was supposed to, and I cannot help them. I do not know anything about what they are doing, so it does not help if I try assist.

The isomorphic team structure and missing *monitoring* resulted in a lack of *backup behavior*.

The integration problem resulted in a backlog not being finished, but it also became evident that not being strict about the criteria for marking work as having been completed affected this. One developer said [interview]:

> We classified tasks as finished before they were completed and we knew there was still work to be done. It seems that the Scrum master wants to show progress and make us look a little better than we really are. These tasks are then not on the list of the next sprint since they officially are done, but we know there is still some more work needed. Each sprint starts with doing things that we have said were finished and then you know you will not finish the sprint.

The Scrum master and some team members gave the impression that the team was better than they actually were and this is related to a particular challenge to *team leadership*, which is known as "impression management" [32]. Impression management is a barrier for learning and improving work practices.

Before this last period began, developers had given priority to the project for 6 months and had worked more than initially planned. Now, new projects began to start, and because the completion of Alpha was planned to release resources at this time, several developers were supposed to start working full-time on the new projects. Alpha started losing resources, and this became a problem because of poor *backup*. One developer said [interview]:

> When correcting errors in this phase, each person was responsible for correcting the errors he had introduced. This is not how it should have been done.

The Scrum master said [interview]:

> We are having problems in one of the modules, but other developers do not want to fix the problem. They want to wait for the developer who wrote the code.

In this phase, it also became clear that insufficient attention had been paid to long-term planning. One developer said [interview]:

> It turned out that certain parts of the system were simply forgotten. There has been a failure somewhere... the product owner and the client asked for things that no one had thought of and that were not in the backlog.

Two reasons for this omission of certain parts of the system were that the dissemination of information was not *coordinated* among the team-members and that no one had the responsibility for the overall technical solution. Dissemination of information among the team-members became an even bigger challenge at the end. Because the team was losing resources, key personnel were absent from the daily stand-up, which resulted in the rest of the team having problems in *monitoring* the progress of the project and in coming to a common understanding of the changing situation.

After the last planned sprint, only two developers continued to work on the project, in addition to the Scrum master and the product owner. The rest of the team members joined other projects, as had been planned earlier. The two remaining programmers spent 1400 h on finalizing the project (correcting errors and doing more testing). The final testing was done 7 months after the last sprint.

The client was satisfied with the delivered functionality but not with the system performance.

5. Discussion

We have described the introduction of the agile process Scrum in a software development project, using the teamwork model proposed by Dickinson and McIntyre [13], Fig. 1. We now discuss the case in light of our research question: "How can we explain the teamwork challenges that arise when introducing a self-managing agile team?" We found the following:

Dickinson and McIntyre proposed that team leadership and team orientation promote team members' capability to monitor their teammates' performance. This does not seem to be borne out by our case study, in a number of ways. Due to the isomorphic team structure, the developers focused on their own modules and often created their own plan and made their own decisions. In addition, problems were seen as personal. This low team orientation on the part of the developers resulted in them not knowing what the others were doing, and as a result it was difficult to monitor others' performance. The team members seemed to be used to having a very high degree of individual autonomy. This created problems when the team members tried to change their normal way of working to become part of a self-managed team. Our findings confirm previous research by Langfred [26], such that there can be a negative effect on team performance when teams are trying to function as a self-managed team when the team members have high individual autonomy. Team leadership was also not distributed as it should be in a self-managing team [32]. Only a few team-members participated in the decision-making, and the Scrum master focused more on command-and-control than providing direction and support for other team members. The Scrum master even ended up supervising one developer because this developer implemented features for future projects, without informing the others. Because the team-members felt the Scrum master overreacted when they reported problems, they started reporting fewer problems, which again limited the possibility of monitoring each other.

The Dickinson and McIntyre model suggests that performance monitoring drives both the content of feedback and timely backup behavior. Due to the fact that the team members did not monitor each other much, there was little feedback and almost no backup, which become evident when the team started to lose resources at the end of the project. The team members did provide some positive feedback, but several found it difficult to both give and accept negative feedback. Our findings also confirm the results of previous research by Levesque et al. [28], such that when the roles that group members play become increasingly specialized and as a result reduce team redundancy and backup, there is a corresponding decline in the amount of time that team members spend working with or communicating with each other. This is also consonant with Marks et al. [30], such that if effective backup is to be provided, teammates need to be informed of each others' work in order to identify what type of assistance is required at a particular time. Marks et al. [30] identify three ways of providing such backup: (1) providing a teammate with verbal feedback or coaching, (2) physically assisting a teammate in carrying out a task, or (3) completing a task for a teammate when it is observed that the workload is too much for him. These means seems to be missing at Alpha.

Dickinson and McIntyre argue that, when all the aforementioned teamwork competences occur in unison, they serve synergistically as a platform for team coordination. The Alpha team had problems with all the teamwork competences and as a consequence they had problems coordinating the teamwork. Important tasks were even forgotten. Marks et al. [30] also argue that when teams have communication problems they are likely to experience problems with coordinating their work.

According to the Dickinson and McIntyre model, the feedback resulting from team coordination should serve as input back into the team processes. The team identified early on [second retrospective] the need for developers to start working on the same tasks, the lack of backup, problems not being reported, and lack of feedback. The researchers observed, and the team members thought, that the teamwork improved during the course of the project. However, it seemed to be difficult to change the teamwork, because changing meant changing not only the developers' way of working but also organizational structures. The team worked better together in the later phases, but did not improve the team orientation and team leadership in such a way that monitoring was improved. One reason is the observation of what Morgan [32] defines as "impression management", when the team gave the impression to be better than they actually were. Impression management is a barrier to learning [32]. Continuous self-management requires a capacity for learning that allows operating norms and rules to change in response to changes in the wider environment [32].

In the Dickinson and McIntyre model, communication acts as the glue that links together all other teamwork processes. In Scrum, the daily stand-up is the most important mechanism for achieving such communication. Everyone should communicate with everyone else. However, because of problems with team leadership and a lack of monitoring, these meetings were mostly used by the Scrum master for getting an overview of what was going on in the project. Developers were reporting to the Scrum master and not talking to each other. Communication improved when the Scrum master was absent. As a result of the highly specialized skills and corresponding division of work, there was less interaction and communication between the group members. However, this improved in the last phase of the project because by that time, the developers had become accustomed to talking to each other at the daily stand-up.

5.1. Implications for theory

The Dickinson and McIntyre model explains most of our observations. However, it does not model any of the critical antecedents and outcomes of the team process. In the case of Alpha, it was obvious that the team had problems becoming a well-functioning team from the beginning, and that this was one reason for the team having problems in self-managing.

In addition, Dickinson and McIntyre do not describe certain important components, such as trust and shared mental models. We observed that the team had not developed trust at the group level. A lack of trust among the Scrum master and the members of the team was an important reason for why problems were not reported and why a team member was given instructions on what to do. Our findings are consonant with those of Salas et al. [39]: without sufficient trust, team members will expend time and energy protecting, checking, and inspecting each other as opposed to collaborating to provide value-added ideas. It is evident that trust is a prerequisite for shared leadership, feedback, and communication. Our finding regarding the lack of trust also confirms previous research on trust [4], such that team members may not be willing to share information if they fear being perceived as incompetent.

The team lacked a shared mental model on what the outcome of the project should be. Working cooperatively requires the team to have shared mental models [39]. Our results are also consonant with Salas et al.'s [39] findings that without a shared understanding, the individual members may be headed toward different goals, which in turn will lead to ineffective/lack of feedback or assistance. Shared mental models are also a prerequisite for communication, monitoring, and team orientation. In addition, our finding confirms the results of previous research on shared mental models in software development teams, such that not all teams develop increasingly shared mental models over time [28].

Having problems with trust and developing shared mental models could also be a reason why the team did not manage to change the team process more than we observed. In addition, the previous ways of working in the company hindered effective teamwork, and in this setting the team did not succeed in improving their teamwork skills significantly during the project.

For theory, this study shows that:

- There is a vast literature on teamwork that is very relevant for agile development and that deserves more attention.
- Dickinson and McIntyre's model [13] should be extended to include trust and shared mental models.

5.2. Implication for practice

Agile software development emphasizes that teams should be self-managed. However, Scrum and agile methods offer no advice on how shared leadership should be implemented. A practical implication of Langfred's [26] findings is that, if an organization believes in letting teams be more self-managing, great care must be taken in the implementation. This is especially important when the team members have high individual autonomy.

The Alpha project was the first big project for most developers. Even though they had worked together for years, they should probably have spent more time together focusing on improving teamwork in the initial phase of the project. The successful teams that Katzenbach et al. [22] observed all gave themselves the time to learn to be a team. If developers who work together have problems becoming a team, they will also have problems becoming a self-managing team.

What people should do to provide backup is not specified clearly in Scrum. In the literature on self-managed teams, backup behavior has been identified as an important prerequisite for self-management [32,35]. In our study, highly specialized skills and a corresponding division of work was the most important barrier to achieving backup and then self-management.

Scrum is not very specific on how to establish monitoring in development teams, although this is implicitly a prerequisite for feedback, coordination, and backup. Combining Scrum with, for example, the practice of pair programming in XP [6] would improve monitoring, feedback, and backup.

We believe that our study has the following main implications for practice:

- An isomorphic project structure will hinder teamwork because the division of work will make it more difficult for developers to develop shared mental models, trust each other, communicate, coordinate work, and provide backup. One way of handling this is to organize cross-training and appreciate generalist to build redundancy in the organization [31].
- Self-management should be enabled when starting to use agile methods such as Scrum, and be aware that high individual autonomy may results in problems creating a self-managing team.
- The way in which agile practices are taken up is dependent on the companies' former development process. Changes take time and resources, and for the company in this study, previous practices were sustained throughout their first agile project.
- The development process should be adjusted for enabling efficient work, by making room for reflection and learning. However, achieving learning in software processes is not trivial.

5.3. Limitations

The main limitations of our study are the single-case design and the possibility of bias in data collection and analysis. The fact that we used a single-case holistic design makes us more vulnerable to bias and eliminates the possibility of direct replication or the analysis of contrasting situations. Therefore, the general criticisms about single-case studies, such as uniqueness and special access to key informants, may also apply to our study. However, our rationale for choosing Alpha as our case was that it represents a critical case for explaining the challenges for teamwork that arise when introducing self-managing agile teams. We used Alpha to determine whether we could confirm, challenge, or extend Dickinson and McIntyre's [13] teamwork model. Our goal was not to provide statistical generalizations about a population on the basis of data collected from a sample of that population. On the contrary, our mode of generalization is analytical, i.e., we used a previously developed theory as a template with which we compared the empirical results of the case study, which is similar to Yin's [49] concept of Level Two inference.

Another possible limitation is that we based much of our data collection and analysis on semi-structured interviews [14]. The use of multiple data sources made it possible to find evidence for episodes and phenomena from more than one data source; we also observed, talked to, and interviewed the team members over a period of 9 months, which made it possible to study the phenomena from different viewpoints as they emerged and changed.

Could it be that we as researchers influenced the teamwork characteristics by our presence in the project? Our presence and questions might have made the team members more aware of teamwork characteristics, but we do not think their behavior was influenced by our presence. The everyday demands of the projects were high, and we did not observe changes in behavior that seemed to relate to our interview or observation phases.

5.4. Future work

The results of this study point out a number of directions for future research. Firstly, our study highlights several challenges that must be met when self-managing teams are introduced into agile development. Accordingly, further work should focus on identifying and addressing other problems that may arise when introducing agile development.

Secondly, the extended teamwork model should be used for studying mature agile development teams, in order to get a better understanding of the main challenges in such teams. Also, teams using shorter sprints (e.g. 2–3 weeks) should be studied, since this will give the team more frequent feedback, which affect team learning and the other elements in the Dickinson and McIntyre teamwork model.

Thirdly, our study tries to answer "How can we explain the teamwork challenges that arise when introducing a self-managing agile team?" through Dickinson and McIntyre's teamwork model. However, there are other relevant streams of research to address the adoption of methods and technology, e.g. the diffusion of innovation literature [37]. Other models that attempt to explain the relationship between user perceptions, attitudes and use intentions include the technology acceptance model (TAM) [11], and the theory of planned behavior [2].

6. Conclusion

We have conducted a nine-month field study of professional developers in a Scrum team. We found that the model of Dickinson and McIntyre [13], together with trust and shared mental models, explain our findings. In addition to these teamwork components, highly specialized skills and a corresponding division of work was the most important barrier for achieving effective teamwork. We have also seen that Scrum has several mechanisms in place for supporting the recommendations of the framework, but that many of these mechanisms are not easy to implement in practice.

Transitioning from individual work to self-managing teams requires a reorientation not only by developers but also by management. Making such changes takes time and resources, but it is a prerequisite for the success of any kind of agile method based on self-management.

Acknowledgement

This work was supported by the EVISOFT project, which is partially funded by the Research Council of Norway under Grant 174390/I40.

Appendix A

A.1. Interview guide

The respondent was informed of the nature of the study and how long the interview will take. The respondent was told why it is important to tape the interview and that only the researchers would have access to the transcript. The respondent was finally asked if he/she would agree to the interview being taped.

Questions for warm-up:

- What are you working on now?
- What is the status of the project?

Main body of the interview:

- How is work coordinated in the project?
  - How was it done in earlier projects?
- How are problems that emerge in the project solved?
  - How was it done in earlier projects?
- Do you have an overview of what others are doing?
  - How was it done in earlier projects?
- How easy is it to carry on work that was begun by others?
  - How was it done in earlier projects?
- How do you discover changes in the project?
  - How was it done in earlier projects?
- How do you deal with changes in the project?
  - How was it done in earlier projects?
- Does the team have a common project goal?
  - Did earlier projects have a common project goal?
- Does everyone know the expected outcome of the project?
  - How was it done in earlier projects?
- Do team members give each other feedback in the project?
  - How was it done in earlier projects?
- Do team members share relevant project information with each other?
  - How was it done in earlier projects?
- How is the team communication?
  - How was it done in earlier projects?
- How is the team performance?
  - How was it done in earlier projects?
- How do you think Scrum is working in the project?
  - What is working?
  - What is not working?
- Is there anything else you would like to add that you think is interesting in this context, but not covered by the questions asked?

References

[1] S.T. Acuna, M. Gomez, N. Juristo, How do personality team processes and task characteristics relate to job satisfaction and software quality?, Information and Software Technology 51 (3) (2009) 627–639
[2] I. Ajzen, The theory of planned behavior, Organizational Behavior and Human Decision Processes 50 (2) (1991) 179–211.
[3] A. Aurum, C. Wohlin, A. Porter, Aligning software project decisions: a case study, International Journal of Software Engineering and Knowledge Engineering 16 (6) (2006) 795–818.
[4] D. Bandow, Time to create sound teamwork, The Journal for Quality and Participation 24 (2) (2001) 41–47.
[5] J.R. Barker, Tightening the iron cage – concertive control in self-managing teams, Administrative Science Quarterly 38 (3) (1993) 408–437.
[6] K. Beck, C. Anders, Extreme Programming Explained: Embrace Change, second ed., Addison-Wesley, 2004.
[7] R.M. Belbin, Team Roles at Work, Butterworth-Heinemann, Boston, MA, 1993.
[8] B.W. Boehm, R. Turner, Balancing Agility and Discipline: a Guide for the Perplexed, Addison-Wesley, 2003.
[9] C.S. Burke, K.C. Stagl, C. Klein, G.F. Goodwin, E. Salas, S.A. Halpin, What type of leadership behaviors are functional in teams? A meta-analysis, Leadership Quarterly 17 (3) (2006) 288–307.
[10] S.G. Cohen, D.E. Bailey, What makes teams work: group effectiveness research from the shop floor to the executive suite, Journal of Management 23 (3) (1997) 239–290.
[11] F.D. Davis, Perceived usefulness, perceived ease of use, and user acceptance of information technology, MIS Quarterly 13 (3) (1989) 319–340.
[12] E. Derby, D. Larsen, Agile retrospectives: making good teams great, Pragmatic Bookshelf, 2006.
[13] T.L. Dickinson, R.M. McIntyre, A conceptual framework of teamwork measurement, in: M.T. Brannick, E. Salas, C. Prince (Eds.), Team Performance Assessment and Measurement: Theory, Methods, and Applications, Psychology Press, NJ, 1997, pp. 19–43.
[14] T. Diefenbach, Are case studies more than sophisticated storytelling? Methodological problems of qualitative empirical research mainly based on semi-structured interviews, Quality and Quantity 43 (6) (2009) 875–894.
[15] T. Dybå, Improvisation in small software organizations, IEEE Software 17 (5) (2000) 82–87.
[16] T. Dybå, T. Dingsøyr, Empirical studies of agile software development: a systematic review, Information and Software Technology 50 (9–10) (2008) 833–859.

[17] R.A. Guzzo, M.W. Dickson, Teams in organizations: recent research on performance and effectiveness, Annual Review of Psychology 47 (1996) 307–338.
[18] J.R. Hackman, The design of work teams, in: J. Lorsch (Ed.), Handbook of Organizational Behavior, Prentice-Hall, Englewood Cliffs, NJ, 1987.
[19] M. Hoegl, H.G. Gemuenden, Teamwork quality and the success of innovative projects: a theoretical concept and empirical evidence, Organization Science 12 (4) (2001) 435–449.
[20] M. Hoegl, K.P. Parboteeah, Autonomy and teamwork in innovative projects, Human Resource Management 45 (1) (2006) 67–79.
[21] J. Jurison, Software project management: the manager's view, Communications of AIS 2 (1999).
[22] J.R. Katzenbach, D.K. Smith, The discipline of teams, Harvard Business Review 71 (2) (1993) 111–120.
[23] B.L. Kirkman, B. Rosen, Beyond self-management: antecedents and consequences of team empowerment, Academy of Management Journal 42 (1) (1999) 58–74.
[24] H.K. Klein, M.D. Myers, A set of principles for conducting and evaluating interpretive field studies in information systems, MIS quarterly 23 (1) (1999) 67–93.
[25] R.E. Kraut, L.A. Streeter, Coordination in software development, Communications of the ACM 38 (3) (1995) 69–81.
[26] C.W. Langfred, The paradox of self-management: individual and group autonomy in work groups, Journal of Organizational Behavior 21 (5) (2000) 563–585.
[27] A. Langley, Strategies for theorizing from process data, Academy of Management (1999) 691–710.
[28] L.L. Levesque, J.M. Wilson, D.R. Wholey, Cognitive divergence and shared mental models in software development project teams, Journal of Organizational Behavior 22 (2001) 135–144.
[29] S.E. Markham, I.S. Markham, Self-management and self-leadership reexamined: a levels-of-analysis perspective, The Leadership Quarterly 6 (3) (1995) 343–359.
[30] M.A. Marks, J.E. Mathieu, S.J. Zaccaro, A temporally based framework and taxonomy of team processes, Academy of Management Review 26 (3) (2001) 356–376.
[31] N.B. Moe, T. Dingsøyr, T. Dybå, Overcoming barriers to self-management in software teams, Software, IEEE 26 (6) (2009) 20–26.
[32] G. Morgan, Images of Organizations, SAGE publications, Thousand Oaks, CA, 2006.
[33] S. Nerur, V. Balijepally, Theoretical reflections on agile development methodologies – the traditional goal of optimization and control is making way for learning and innovation, Communications of the ACM 50 (3) (2007) 79–83.
[34] S. Nerur, R. Mahapatra, G. Mangalaraj, Challenges of migrating to agile methodologies, Communications of the ACM 48 (5) (2005) 72–78.
[35] I. Nonaka, H. Takeuchi, The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation, vol. 12, Oxford University Press, New York, 1995.
[36] L. Rising, N.S. Janoff, The Scrum software development process for small teams, IEEE Software 17 (4) (2000) 26–32.
[37] E.M. Rogers, Diffusion of Innovations, fourth ed., The Free Press, New York, 1995.
[38] V. Rousseau, C. Aube, A. Savoie, Teamwork behaviors – a review and an integration of frameworks, Small Group Research 37 (5) (2006) 540–570.
[39] E. Salas, D.E. Sims, C.S. Burke, Is there a "big five" in teamwork?, Small Group Research 36 (5) (2005) 555–599
[40] E. Salas, K.C. Stagl, C.S. Burke, G.F. Goodwin, Fostering Team Effectiveness in Organizations: Toward an Integrative Theoretical Framework. in: 52nd Nebraska Symposium on Motivation, Lincoln, NE, 2007.
[41] J. Sapsed, J. Bessant, D. Partington, D. Tranfield, M. Young, Teamworking and knowledge management: a review of converging themes, International Journal of Management Reviews 4 (1) (2002) 71–85.
[42] B. Schatz, I. Abdelshafi, Primavera gets agile: a successful transition to agile development, IEEE Software 22 (3) (2005) 36–42.
[43] K. Schwaber, Beedle, Agile Software Development with Scrum, Prentice Hall, Upper Saddle River, 2001.
[44] H. Takeuchi, I. Nonaka, The new product development game, Harvard Business Review (64) (1986) 137–146.
[45] J. Tata, S. Prasad, Team self-management, organizational structure, and judgments of team effectiveness, Journal of Managerial Issues 16 (2) (2004) 248–265.
[46] B.B.W. Tuckman, Developmental sequence in small groups, Psychological Bulletin 63 (1965) 384–399.
[47] M. Uhl-Bien, G.B. Graen, Individual self-management: analysis of professionals' self-managing activities in functional and cross-functional work teams, Academy of Management Journal 41 (3) (1998) 340–350.
[48] M.A. West, The social psychology of innovation in groups, in: M.A. West, J.L. Farr (Eds.), Innovation and Creativity at Work: Psychological and Organizational Strategies, Wiley, Chichester, 1990, pp. 309–333.
[49] R.K. Yin, Case Study Research: Design and Methods, vol. xiv, Sage, Thousand Oaks, CA, 2009. p. 219 s.

Statement of authorship of joint publications

Statement of authorship of joint publications from Erik Arisholm
Statement of authorship of joint publications from Aybuke Aurum
Statement of authorship of joint publications from Torgeir Dingsøyr
Statement of authorship of joint publications from Tore Dybå
Statement of authorship of joint publications from Øyvind Kvangardsnes
Statement of authorship of joint publications from Jingyue Li
Statement of authorship of joint publications from Edda Mikkelsen
Statement of authorship of joint publications from Emil Røyrvik

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Dybå, T., Moe, N. B. and Arisholm, E. (2005). Measuring Software Methodology Usage: Challenges of Conceptualization and Operationalization. Fourth International Symposium on Empirical Software Engineering (ISESE), Noosa Heads, Australia, IEEE Computer Society, 447 - 457.

Candidate's described contribution to:
Tore Dybå was responsible for the study design. I was responsible collecting the data. Together with Tore Dybå, I inspected and analyze about 1,000 documents in 23 projects in order to measure the ratio of actual template usage. I was also responsible for analyzing the number of server hits on the EPG template pages. Tore Dybå and Erik Arisholm were responsible for the rest of the analysis. Discussing the results of the analysis and writing of the paper was done collaboratively.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to this paper is correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Oslo, 9.5.2011

[illegible]
Erik Arisholm

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Moe, N. B. and Aurum, A. (2008). Understanding Decision-Making in Agile Software Development: A Case-study. Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference, Parma, Italy, 216-223

Candidate's described contribution to:
As the principal author I was in charge of the study design, collecting and analyzing the data. The second author was responsible for the literature review on decision-making. Discussing the results of the analysis and writing of the paper was done collaboratively.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to this paper is correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Sydney, 10.5.2011

[illegible]

Aybuke Aurum

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Moe, N. B., Dingsøyr, T. and Kvangardsnes, Ø. (2009). Understanding Shared Leadership in Agile Development: A Case Study. Hawaii International Conference on System Sciences, Hawaii, 1-10

Candidate's described contribution to:
As the principal author I was in charge of the study design. Øyvind Kvangardsnes (master student) and I conducted ethnographic observations and interviews from April 2007 until January 2008. All authors participated in the discussions and writing of the material, however I was responsible for analyzing the qualitative data in Nvivo.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to this paper is correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Trondheim, 10.5.2011

[illegible]

Torgeir Dingsøyr

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Moe, N. B., Dingsøyr, T. and Røyrvik, E. A. (2009). Putting Agile Teamwork to the Test – An Preliminary Instrument for Empirically Assessing and Improving Agile Software Development. 10th International Conference on Agile Processes in Software Engineering and Extreme Porgramming, Sardinia, Italy, 114 -123

Candidate's described contribution to:
As the principal author I was in charge of the study design. I did the interviews in two of the three projects in this article. I also lead the work on creating the team radar tool. Discussing the results and writing of the paper was done collaboratively.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to this paper is correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Trondheim, 10.5.2011

Torgeir Dingsøyr

Torgeir Dingsøyr

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Moe, N. B., Dingsøyr, T. and Dybå, T. (2008). Understanding Self-Organizing Teams in Agile Software Development. 19th Australian Conference on Software Engineering.

Candidate's described contribution to:
As the principal author I was in charge of the study design. The second author and I conducted all the interviews and observations. I was also responsible for analysing the qualitative material, by coding and re-coding it in NVivo. Discussing the results and writing of the paper was done collaboratively.

Moe, N. B., Dingsøyr, T. and Dybå, T. (2009). Overcoming Barriers to Self-Management in Software Teams. Software, IEEE 26(6): 20-26.

Candidate's described contribution to:
As the principal author I was in charge of the study design. The second author and I conducted all the interviews and observations in two of the companies. I conducted all the interviews and observations in the last company. I was also responsible for analysing the qualitative material, by coding and re-coding it in NVivo. Discussing the results and writing of the paper was done collaboratively.

Moe, N. B., Dingsøyr, T. and Dybå, T. (2010). A teamwork model for understanding an agile team: A case study of a Scrum project. Information and Software Technology 52(5): 480-491.

Candidate's described contribution to:
As the principal author I was in charge of the study design. The second author and I were both involved in the nine-month fieldwork. We interviewed, observed the team, and collected the documentation. I conducted most of the observations. I identified the team-work model used in this article and I was also responsible for analysing the qualitative material, by coding and re-coding it in NVivo. Discussing the results and writing of the paper was done collaboratively.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to these papers are correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Trondheim, 10.5.2011

Torgeir Dingsøyr
Torgeir Dingsøyr

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Dingsøyr, T. and Moe, N. B. (2008). The Impact of Employee Participation on the Use of an Electronic Process Guide: A Longitudinal Case Study. IEEE Trans. Softw. Eng. 34(2): 212-225.

Candidate's described contribution to:
I participated in the whole process, from planning of the study to analysis, and reporting. We interviewed half of the developers each and I was the one responsible for collecting and analyzing the access logs on the electronic process guide. The first author had the overall responsibility of the study, and made the final decisions on form.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to this paper is correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Trond~~heim~~, 10.5.2011

[illegible]
Torgeir Dingsøyr

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Moe, N. B., Dingsøyr, T. and Dybå, T. (2008). Understanding Self-Organizing Teams in Agile Software Development. 19th Australian Conference on Software Engineering.

Candidate's described contribution to:
As the principal author I was in charge of the study design. The second author and I conducted all the interviews and observations. I was also responsible for analysing the qualitative material, by coding and re-coding it in NVivo. Discussing the results and writing of the paper was done collaboratively.

Moe, N. B., Dingsøyr, T. and Dybå, T. (2009). Overcoming Barriers to Self-Management in Software Teams. Software, IEEE 26(6): 20-26.

Candidate's described contribution to:
As the principal author I was in charge of the study design. The second author and I conducted all the interviews and observations in two of the companies. I conducted all the interviews and observations in the last company. I was also responsible for analysing the qualitative material, by coding and re-coding it in NVivo. Discussing the results and writing of the paper was done collaboratively.

Moe, N. B., Dingsøyr, T. and Dybå, T. (2010). A teamwork model for understanding an agile team: A case study of a Scrum project. Information and Software Technology 52(5): 480-491.

Candidate's described contribution to:
As the principal author I was in charge of the study design. The second author and I were both involved in the nine month fieldwork. We interviewed, observed the team, and collected the documentation. I conducted most of the observations. I identified the team-work model used in this article and I was also responsible for analysing the qualitative material, by coding and re-coding it in NVivo. Discussing the results and writing of the paper was done collaboratively.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to these papers are correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Trondheim, 10.5.2011

[illegible]

Tore Dybå

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Li, J., Moe, N. B. and Dybå, T. (2010). Transition from a plan-driven process to Scrum: a longitudinal case study on software quality. Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. Bolzano-Bozen, Italy, ACM: 1-10.

Candidate's described contribution to:
I participated in the planning of the study. I conducted and analysed all the interviews and the observations. Discussing the results of the analysis and writing of the paper was done collaboratively. The first author had the overall responsibility of the study, and made the final decisions on form.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to this paper is correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Trondheim, 10.5.2011

Tore Dybå

Tore Dybå

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Moe, N. B. and Dybå, T. (2006). The use of an Electronic Process Guide in a medium sized Software Development Company. Software Process Improvement and Practice 11(1): 21-34.

Candidate's described contribution to:
As the principal author I was in charge of the study design. I conducted all the interviews, which was the data source for this article. I was also responsible for analyzing the interview material, by coding and re-coding it in NVivo. Discussing the results and writing of the paper was done collaboratively.

Moe, N. B. and Dyba, T. (2006). Improving by involving: a case study in a small software company. EuroSPI 2006, Joensuu, Finland, 158 – 169.

Candidate's described contribution to:
As the principal author I was in charge of the study design. I was the only author partaking in introducing and assisting the company in four of the five participative techniques introduced: search conferences, autonomous work groups, quality circles, and learning meetings. Tore Dybå was the one responsible for the last technique (survey feedback). The literature review on participation, discussing the results of the analysis, and writing of the paper was done collaboratively.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to these papers are correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Trondheim, 10.5.2011

Tore Dybå

Tore Dybå

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Dybå, T., Moe, N. B. and Arisholm, E. (2005). Measuring Software Methodology Usage: Challenges of Conceptualization and Operationalization. Fourth International Symposium on Empirical Software Engineering (ISESE), Noosa Heads, Australia, IEEE Computer Society, 447 - 457.

Candidate's described contribution to:
Tore Dybå was responsible for the study design. I was responsible collecting the data. Together with Tore Dybå, I inspected and analyze about 1,000 documents in 23 projects in order to measure the ratio of actual template usage. I was also responsible for analyzing the number of server hits on the EPG template pages. Tore Dybå and Erik Arisholm were responsible for the rest of the analysis. Discussing the results of the analysis and writing of the paper was done collaboratively.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to this paper is correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Trondheim, 10.5.2011

Tore Dybå

Tore Dybå

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Dybå, T., Moe, N. B. and Mikkelsen, E. M. (2004). "An Empirical Investigation on Factors Affecting Software Developer Acceptance and Utilization of Electronic Process Guides". Proceedings of the International Software Metrics Symposium (METRICS), Chicago, Illinois, USA, 220–231.

Candidate's described contribution to:
Tore Dybå was the one responsible for the study design and analysis. I was responsible for the data collection, and Edda Mikkelsen helped me distributing the 120 questionnaires. 97 usable responses were received after using a lot of effort chasing the missing respondents, resulting in a good overall response rate of 81%. I was also in charge of the literature review on process guides. The discussions and writing was done collaboratively.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to this paper is correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Trondheim, 9.5.2011

[illegible]

Tore Dybå

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Moe, N. B., Dingsøyr, T. and Kvangardsnes, Ø. (2009). Understanding Shared Leadership in Agile Development: A Case Study. Hawaii International Conference on System Sciences, Hawaii, 1-10

Candidate's described contribution to:
As the principal author I was in charge of the study design. Øyvind Kvangardsnes (master student) and I conducted ethnographic observations and interviews from April 2007 until January 2008. All authors participated in the discussions and writing of the material, however I was responsible for analyzing the qualitative data in Nvivo.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to this paper is correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Oslo, 10.5.2011

[illegible]

Øyvind Kvangardsnes

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Li, J., Moe, N. B. and Dybå, T. (2010). Transition from a plan-driven process to Scrum: a longitudinal case study on software quality. Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. Bolzano-Bozen, Italy, ACM: 1-10.

Candidate's described contribution to:
I participated in the planning of the study. I conducted and analysed all the interviews and the observations. Discussing the results of the analysis and writing of the paper was done collaboratively. The first author had the overall responsibility of the study, and made the final decisions on form.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to this paper is correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Oslo, 10.5.2011

[illegible]

Jingyue Li

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Dybå, T., Moe, N. B. and Mikkelsen, E. M. (2004). "An Empirical Investigation on Factors Affecting Software Developer Acceptance and Utilization of Electronic Process Guides". Proceedings of the International Software Metrics Symposium (METRICS), Chicago, Illinois, USA, 220–231.

Candidate's described contribution to:
Tore Dybå was the one responsible for the study design and analysis. I was responsible for the data collection, and Edda Mikkelsen helped me distributing the 120 questionnaires. 97 usable responses were received after using a lot of effort chasing the missing respondents, resulting in a good overall response rate of 81%. I was also in charge of the literature review on process guides. The discussions and writing was done collaboratively.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to this paper is correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Oslo, 9.5.2011

Edda M. Mikkelsen

Edda Mikkelsen

NTNU – Trondheim
Norwegian University of
Science and Technology

To the evaluation committee

Co-authorship regarded publication included in Nils Brede Moe's dr.philos thesis

Moe, N. B., Dingsøyr, T. and Røyrvik, E. A. (2009). Putting Agile Teamwork to the Test – An Preliminary Instrument for Empirically Assessing and Improving Agile Software Development. 10th International Conference on Agile Processes in Software Engineering and Extreme Porgramming, Sardinia, Italy, 114 -123

Candidate's described contribution to:
As the principal author I was in charge of the study design. I did the interviews in two of the three projects in this article. I also lead the work on creating the team radar tool. Discussing the results and writing of the paper was done collaboratively.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to this paper is correctly identified above, and I consent to Nils Brede Moe including it in his dr.philos dissertation.

Trondheim, 10.5.2011

[illegible]
Emil Røyrvik