View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Aaltodoc Publication Archive

Department of Computer Science and Engineering

Empirical assessment of the adoption, use, and effects of pair programming

Jari Vanhanen

Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 135/2011

Empirical assessment of the adoption, use, and effects of pair programming

Jari Vanhanen

Doctoral dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the School of Science for public examination and debate in Auditorium T2 at the Aalto University School of Science (Espoo, Finland) on the 2nd of December 2011 at noon.

Aalto University
School of Science
Department of Computer Science and Engineering
Software Process Research Group

Supervisor
Professor Casper Lassenius

Preliminary examiners
Professor Laurie Williams
North Carolina State University, USA

Professor Markku Tukiainen
University of Eastern Finland, Finland

Opponents
Professor Emilia Mendes
Zayed University, Dubai

Aalto University publication series
DOCTORAL DISSERTATIONS 135/2011

© Jari Vanhanen

ISBN 978-952-60-4413-2 (pdf)
ISBN 978-952-60-4412-5 (printed)
ISSN-L 1799-4934
ISSN 1799-4942 (pdf)
ISSN 1799-4934 (printed)

Unigrafia Oy
Helsinki 2011

Finland

The dissertation can be read at http://lib.tkk.fi/Diss/

441 697
Printed matter

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Jari Vanhanen

Name of the doctoral dissertation
Empirical assessment of the adoption, use, and effects of pair programming

Publisher School of Science

Unit Department of Computer Science and Engineering

Series Aalto University publication series DOCTORAL DISSERTATIONS 135/2011

Field of research Software engineering

Manuscript submitted 14 June 2011 **Manuscript revised** 27 October 2011

Date of the defence 2 December 2011 **Language** English

☐ **Monograph** ☒ **Article dissertation (summary + original articles)**

Abstract

Developing large software systems requires team work, which in turn calls for lots of communication within the team. However, programming is typically conducted alone by individual software developers. Pair programming, where two persons actively collaborate in the implementation of a single task, is an alternative way of developing software. It has been proposed as a means to increasing software quality, knowledge transfer and learning, among other things.

This research studied the adoption, use, and effects of pair programming through a literature study and three empirical studies. The literature study was a systematic mapping study of the previous pair programming research in the industry. The empirical studies consisted of two long industry case studies and an experiment where project teams consisting of experienced students conducted a moderately large software development project.

The systematic mapping study analyzed the content of 154 papers. It identified industrially relevant aspects of pair programming and organized them as a pair programming framework containing additional and more detailed aspects of pair programming over the previously published frameworks. The framework grouped all the identified aspects under eighteen factors of pair programming, for which their state of research was analyzed. The analysis showed that of many factors, only a few or no studies had been conducted using rigorous research approaches and data collection methods.

The adoption and use of pair programming were analyzed in the two case studies. In the larger, more established organization, there were issues with adoption, related to both infrastructure and organizing of pair programming. A separate pair programming room was a successful solution to the infrastructural issues. However, lack of time for pair programming due to insufficient organizing of its use, remained an issue at the end of the study.

The effects of pair programming on software quality and developers' knowledge were positive in all three empirical studies, but the development effort for individual tasks increased. The increase in effort occurred mainly when using pair programming for simple tasks or during the beginning of a project, when the developers were learning pair programming and getting to know one another.

Keywords pair programming, empirical research, industry

ISBN (printed) 978-952-60-4412-5 **ISBN (pdf)** 978-952-60-4413-2

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Espoo **Location of printing** Helsinki **Year** 2011

Pages 132 **The dissertation can be read at** http://lib.tkk.fi/Diss/

Aalto-yliopisto

Tiivistelmä

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tekijä
Jari Vanhanen

Väitöskirjan nimi
Pariohjelmoinnin käyttöönoton, käytön ja vaikutusten empiirinen arviointi

Julkaisija Perustieteiden korkeakoulu

Yksikkö Tietotekniikan laitos

Sarja Aalto University publication series DOCTORAL DISSERTATIONS 135/2011

Tutkimusala Ohjelmistotuotanto

Käsikirjoituksen pvm 14.06.2011 **Korjatun käsikirjoituksen pvm** 27.10.2011

Väitöspäivä 02.12.2011 **Kieli** Englanti

☐ **Monografia** ☒ **Yhdistelmäväitöskirja (yhteenveto-osa + erillisartikkelit)**

Tiivistelmä

Suurten ohjelmistojärjestelmien kehittäminen on ryhmätyötä, joka vaatii paljon kommunikointia ryhmän sisällä. Ohjelmointityö tehdään kuitenkin tyypillisesti yksittäisten ohjelmistokehittäjien toimesta. Pariohjelmointi, jossa kaksi henkilöä tekevät aktiivista yhteistyötä saman tehtävän parissa, on vaihtoehtoinen tapa kehittää ohjelmistoja. Sen on ehdotettu parantavan muun muassa ohjelmiston laatua, tiedon siirtoa ja oppimista.

Tässä tutkimuksessa tutkittiin pariohjelmoinnin käyttöönottoa, käyttöä ja vaikutuksia. Tutkimus koostui yhdestä kirjallisuustutkimuksesta ja kolmesta empiirisestä tutkimuksesta. Kirjallisuustutkimus oli systemaattinen kartoitustutkimus aiemmista teollisuudessa tehdyistä pariohjelmointitutkimuksista. Empiiriset tutkimukset koostuivat kahdesta pitkästä tapaustutkimuksesta teollisuudessa ja yhdestä kokeesta, jossa kokeneista opiskelijoista koostetut ryhmät tekivät kukin kohtuullisen laajan ohjelmistokehitysprojektin.

Systemaattinen kartoitustutkimus analysoi 154 artikkelin sisällön. Siinä tunnistettiin ohjelmistoteollisuuden näkökulmasta relevantit asiat pariohjelmointiin liittyen ja organisoitiin ne pariohjelmoinnin viitekehykseksi, joka sisältää täydentäviä ja yksityiskohtaisempia asioita pariohjelmoinnista verrattuna aiemmin julkaistuihin viitekehyksiin. Viitekehys ryhmitteli tunnistetut asiat 18 tekijän alle. Kunkin tekijän osalta analysointiin aiemman tutkimuksen tilaa. Moniin tekijöihin liittyen löytyi korkeintaan muutamia tutkimuksia, jotka oli tehty kurinalaisia tutkimuksen lähestysmistapoja ja tiedonkeruumenetelmiä käyttäen.

Pariohjelmoinnin käyttöönottoa ja käyttöä tutkittiin kahdessa tapaustutkimuksessa. Suuremmassa ja vakiintuneemmassa organisaatiossa oli ongelmia sen käyttöönotossa. Ongelmat liittyivät sekä pariohjelmoinnin organisointiin että infrastruktuuriin. Erillinen pariohjelmointihuone ratkaisi infrastruktuuriin liittyvät ongelmat. Pariohjelmoinnin puutteellisesta organisoinnista johtunut ajanpuute sen käyttämiseen jäi kuitenkin avoimeksi ongelmaksi vielä tutkimuksen lopussa.

Pariohjelmoinnin vaikutukset ohjelmistojen laatuun ja kehittäjien tietämykseen olivat positiivisia kaikissa kolmessa empiirisessä tutkimuksessa, mutta yksittäisiin tehtäviin käytetty työmäärä lisääntyi. Työmäärän lisääntyminen esiintyi pääosin silloin, kun pariohjelmointia käytettiin yksinkertaisiin tehtäviin ja projektien alussa, jolloin kehittäjät vasta opettelivat käyttämään pariohjelmointia ja tuntemaan toisensa.

Avainsanat pariohjelmointi, empiirinen tutkimus, teollisuus

ISBN (painettu) 978-952-60-4412-5 **ISBN (pdf)** 978-952-60-4413-2

ISSN-L 1799-4934 **ISSN (painettu)** 1799-4934 **ISSN (pdf)** 1799-4942

Julkaisupaikka Espoo **Painopaikka** Helsinki **Vuosi** 2011

Sivumäärä 132 **Luettavissa verkossa osoitteessa** http://lib.tkk.fi/Diss/

Acknowledgements

This research was done at SoberIT at the Department of Computer Science and Engineering at Aalto University. There are many people I want to thank for their contribution to this research.

First, I would like to thank my supervisor, Professor Casper Lassenius, for his support and guidance in my research work. Over the years also all the other members of the Software Process Research Group have participated as co-authors or internal reviewers of my papers, or by commenting my research in informal discussions. Especially, I want to mention Mika Mäntylä, Kristian Rautiainen, Juha Itkonen, Jarno Vähäniitty and Timo Lehtinen for their contributions.

Timo Rihtniemi, Harri Korpi and Tuomo Kähkönen had a crucial role in making the data collection from the industrial case studies possible. Timo Jalonen and Matti Kokkola made a huge contribution for the infrastructure of the student experiment by developing the core system and use cases, and by lecturing the related course. Mikko Rusama participated also in preparing the infrastructure for the experiment. I also want to thank the software developers in the companies and the students in the experiment who participated in my research as subjects.

It is difficult to recall the starting moment of my pair programming research, but Asko Seeba and Priit Salumaa deserve to be mentioned as the first persons with whom I had several, long discussions on planning a "perfect" pair programming experiment during the XP 2003 conference.

I also wish to thank my preliminary examiners Professor Laurie Williams and Professor Markku Tukiainen for their comments.

This research would not have been possible without the financial support of the Finnish Funding Agency for Technology (Tekes) and the participating companies for the SHAPE and ESPA projects, and the Graduate School for Electronic Business and Software Industry (GEBSI).

I want to thank my wife Heidi for her love and support, my son Aapo for forcing me regularly keep my thoughts away from all other things except playing with him, and my newborn daughter Oona, who has brought me joy in the last weeks of finalizing this thesis.

Espoo, October 2011

Jari Vanhanen

List of publications

This dissertation consists of this summary and the following publications which are referred to in the text by their roman numerals.

I. J. Vanhanen and M.V. Mäntylä, "**A systematic mapping study of empirical studies on the use of pair programming by professional developers**," *IEEE Transactions on Software Engineering*, submitted for review in March 2011, (17 pages + 15 appendix pages).

II. J. Vanhanen, C. Lassenius, and M.V. Mäntylä, "**Issues and tactics when adopting pair programming: A longitudinal case study**, " in *Proceedings of the Second International Conference on Software Engineering Advances* (ICSEA 2007), Cap Esterel, France, August 2007, pp. 70 (7 pages).

III. J. Vanhanen and C. Lassenius, "**Perceived effects of pair programming in an industrial context**," in *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications* (EUROMICRO 2007), Lübeck, Germany, August 2007, pp. 211–218.

IV. J. Vanhanen and H. Korpi, "**Experiences of using pair programming in an agile project**," in *Proceedings of Hawaii International Conference on System Sciences* (HICSS-40), Waikoloa, Hawaii, USA, January 2007, pp. 274b (10 pages).

V. J. Vanhanen and C. Lassenius, "**Effects of pair programming at the development team level: An experiment**," in *Proceedings of International Symposium on Empirical Software Engineering* (ISESE 2005), Noosa, Australia, November 2005, pp. 336–345.

Author's contribution

The author of this dissertation is the primary author of all the included publications. For publications I, II, III and V, he was solely responsible for the research design, data collection, data analysis and writing of the publication. For publication IV, he was partially responsible for the data collection, and solely responsible for the data analysis and writing.

Table of contents

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Definitions 2
- 1.3 Research goal and questions 3
- 1.4 Structure of the thesis 4

2 Background 5
- 2.1 Local amount of pair programming 5
- 2.2 Targets of pair programming 6
- 2.3 Infrastructure for pair programming 8
- 2.4 Managing pair programming 8
- 2.5 Adoption of pair programming 9
- 2.6 Productivity 10
- 2.7 Software quality 10
- 2.8 Developer's knowledge of work 11

3 Research design 13
- 3.1 Research methodologies 13
  - 3.1.1 Systematic mapping study 13
  - 3.1.2 Case study 15
  - 3.1.3 Experiment 16
- 3.2 Research environment 16
  - 3.2.1 Study A 16
  - 3.2.2 Study B 17
  - 3.2.3 Study C 17
- 3.3 Summary of the empirical studies 17

4 Results 19
- 4.1 Pair programming factors 19
- 4.2 Adoption and use of pair programming 22
  - 4.2.1 Adoption of pair programming in Study A 22
  - 4.2.2 Adoption of pair programming in Study B 24
  - 4.2.3 Use of pair programming 24
- 4.3 Effects of pair programming 25
  - 4.3.1 Productivity 25
  - 4.3.2 Software quality 25
  - 4.3.3 Developer's knowledge of work 26

5 Discussion 28
- 5.1 Pair programming factors 28
- 5.2 Adoption and use of pair programming 29
  - 5.2.1 Adoption of pair programming 29
  - 5.2.2 Use of pair programming 30
- 5.3 Effects of pair programming 31
  - 5.3.1 Productivity 31
  - 5.3.2 Software quality 32
  - 5.3.3 Developer's knowledge of work 32
- 5.4 Evaluation of the research 33
  - 5.4.1 Strengths 33
  - 5.4.2 Limitations 34
- 5.5 Summary of the results 35

6 Conclusions 36
- 6.1 Contributions of the research 36
- 6.2 Future work 36

References 38

Publications 41

1 Introduction

1.1 Motivation

Building large software systems requires the participation of large teams of developers. Even though successful teamwork is likely to require lots of communication between team members, programming is typically an activity that is divided into separate tasks that are mostly implemented alone by individual persons. Pair programming (PP), where two persons actively collaborate in the implementation of a single task, is an alternative way of developing software.

Pair programming became a better known practice when it was introduced as part of the extreme programming (XP) (Beck, 1999) software development methodology in the late 1990s. However, even though the term "pair programming" was not commonly used before the introduction of XP, anecdotal evidence of two programmers working together has been reported all the way back to the 1950s (Williams and Kessler, 2002). Nowadays, according to many surveys, pair programming is used to some degree in many companies within the software industry (see e.g., Salo and Abrahamsson, 2008; Schindler, 2008; Begel and Nagappan, 2008).

Pair programming is used because many benefits are expected to result from using it instead of solo programming. In the context of XP, pair programming is a mandatory practice for all development work because it is expected to encourage communication and increase code quality, and it is required by certain other practices (Beck, 1999). In a book solely devoted to pair programming, Williams and Kessler (2002) describe pair programming as an independent practice that may be incorporated into any software development methodology. They propose many benefits from the use of pair programming, including higher quality without increase in effort, higher morale, trust and teamwork, and better knowledge transfer and enhanced learning.

The empirical evidence on the realization of the proposed effects of pair programming, including both its benefits and costs, is still inconclusive or scarce, depending on the effect. Software quality and development effort are the most frequently studied effects in pair programming experiments, and the results of the main experiments have been analyzed in a meta-analysis (Hannay et al., 2009). However, there is a large variance among the results of the studies included in the meta-analysis, and scarcely studied factors such as task complexity and developer's experience seem to affect the results (Hannay et al., 2009). Identifying any other factors that may affect the effects of pair programming and studying their role are necessary steps for advancing the state of pair programming research.

It is also important to study the adoption of pair programming. Various difficulties related to the adoption have been reported in the literature, as listed in Publication I. In addition, in a previous study, we found that even in organizations that promote agile software development in general, certain agile software development practices, pair programming included, did not just emerge without explicit adoption (Vanhanen et al., 2003).

1.2 Definitions

In this dissertation, we use the definition of pair programming given by Williams and Kessler (2002). Their definition states that *pair programming* is a software development practice where two persons design, code and test software together at one computer, actively communicating with each other. Thus, despite its name, the pair programming practice can also be applied to other software development activities in addition to the coding activity.

The definition above does not mandate using pair programming for all development work, as is the case in the context of extreme programming. We consider also occasional working in pairs as pair programming. Distributed pair programming, where the pair is not physically collocated but may share the same display using network collaboration tools, is not considered pair programming in this dissertation.

We divide the analysis of pair programming into three viewpoints: effects, use and adoption of pair programming. They are defined as follows:

- The *effects of pair programming* cover anything that is affected by the use of pair programming, either positively or negatively, including the developed software and the software development organization along with its people and processes.
- The *use of pair programming* covers anything that may affect the realization of the effects of pair programming such as the way of use, context of use, and amount of use.
- The *adoption of pair programming* considers both the effects and use of pair programming, but in particular considering their role in motivating the adoption of pair programming, or in preventing or supporting the achievement of the desired use of pair programming in an organization, especially when the application of the pair programming practice has just begun.

Pair programming involves numerous aspects that may have relevance for the practitioners and researchers of pair programming. In this dissertation *a factor of pair programming* denotes a group of closely related aspects of pair programming, which may include any effects of pair programming and any aspects that may affect the realization of the effects. Certain factors can be considered both as effects of pair programming and factors that affect the realization of effects of pair programming on some other factors. For example, pair programming may increase developer's knowledge of work, but developer's knowledge of work may also affect, e.g., how strongly quality is affected due to pair programming. The *pair programming framework* denotes the structured presentation of the aspects and factors of pair programming identified in the literature study reported in Publication I.

1.3 Research goal and questions

The research goal is

> *to increase empirical knowledge of the adoption, use, and effects of pair programming*.

We focus on studying pair programming from those viewpoints that are relevant to real-life software development (i.e., when software is developed for real use). We scope out those viewpoints that are relevant only to the application of pair programming for learning programming in university courses, which has been a popular context in previous pair programming studies.

The research questions of this dissertation are described below. They summarize the more detailed research questions presented in Publications I–V. The relationships between the publications and the research questions studied in them are shown in **Table 1**.

Research Question 1: What are the factors of pair programming and how well have they been studied empirically?

We identify any potentially relevant aspects of pair programming and group the related aspects to form a set of factors of pair programming. For each factor, we analyze the previous empirical research through many properties of research, such as research approach, data collection method, discussion type, data type, and author's role in the studies. The overall relevance of each study discussing a factor and the number of these studies are considered when evaluating how advanced the research is per factor.

The research focus on real-life software development scopes out aspects that are relevant only in the educational context. An example of such an aspect is the benefit of pair programming due to halving the number of course assignments and thus saving the course arrangement effort.

We answer Research Question 1 through a systematic mapping study of empirical studies on professional software developers using pair programming published in scientific journals, conferences or workshops. We scope out other literature such as studies of computer science students, non-empirical studies and practitioner literature in order to keep the effort realistic, even though other literature might contribute additional results with regard to the research question.

Research Question 2: What issues and other experiences are confronted in the adoption and use of pair programming in the industry and how can the issues be resolved?

The issues can include anything that prevents reaching the desired use of pair programming in an organization. Experiences can be negative or positive observations that are relevant to the practitioners and researchers of pair programming. Regarding the use of pair programming, we study in

particular aspects related to the targets of pair programming, infrastructure of pair programming, pair formation, and pair programming sessions.

We answer Research Question 2 by conducting two case studies—Study A and Study B—in two organizations. The studies are conducted at a time when the organizations are adopting pair programming.

Research Question 3: Does pair programming affect productivity, software quality, and developer's knowledge of work, and how much, if so?

We study in particular the effects of pair programming on productivity, software quality, and developers' knowledge. Certain other effects such as enjoyment of pair programming, are studied with less emphasis in Publications II–V, and are not included in this summary.

We answer Research Question 3 by conducting an experiment where several teams of experienced students each develop a moderately large software system using either pair programming or solo programming and through the same two case studies as Research Question 2. In the experiment, the results are based on both objective measurements of and the subjects' perceptions of the effects of pair programming. In the case studies, the results are mainly based on the developers' perceptions of the effects of pair programming.

Table 1 Research questions covered in each publication

Publication	Context	RQ 1	RQ 2	RQ 3
I	a literature review of empirical studies	X		
II	a case study in the industry (Study A)		X	
III	a case study in the industry (Study A)		X	X
IV	a case study in the industry (Study B)		X	X
V	an experiment with student projects			X

1.4 Structure of the thesis

Chapter 2 summarizes the previous literature on the areas of pair programming studied in this dissertation. Chapter 3 introduces the utilized research methodologies and the contexts where the empirical research was conducted. Chapter 4 presents the results, and Chapter 5 discusses them along with the evaluation of the research. Chapter 6 summarizes the contributions of the dissertation and proposes future work.

2 Background

This chapter summarizes the most relevant scientific literature related to the research questions covered in this dissertation. The selection of the literature is based on the results of the systematic mapping study reported in Publication I. That study found 154 empirical papers discussing the use of pair programming by professional developers, published before year 2010.

In the systematic mapping study, all the aspects of pair programming discussed in the papers were extracted and organized under eighteen factors of pair programming (see **Table 4**, page 20). Eight of these factors were the main targets of research in Publications II–V. These included *local amount of PP, targets of PP, infrastructure for PP, managing PP, adoption of PP, productivity, software quality, and developer's knowledge*.

In the systematic mapping study the content of each paper was classified according to the overall relevance of research per factor. Based on this classification each section below summarizes the results of the most relevant empirical studies of each of the eight factors.

The systematic mapping study also identified but excluded more than 200 papers that discussed the use of pair programming by students or that were non-empirical papers. Of those papers, the most relevant ones related to the factors discussed below have been included in this chapter. The selection of the most relevant papers was done based on the titles and abstracts of the papers, instead of reading the full content of the papers, as was the case in the systematic mapping study. Regarding the productivity and software quality factors, most of the previous student experiments are included in the meta-analysis by Hannay et al. (2009), whose results are discussed in the corresponding sections below rather than discussing the individual studies included in the meta-analysis.

2.1 Local amount of pair programming

The local amount of pair programming covers the amount of using the pair programming practice in an organization. The perspectives of interest include the realized and desired amounts of pair programming as well as the absolute amount of pair programming and proportion of pair programming of all the development work.

Fronza et al. (2009) studied a team that used customized XP where the use of pair programming was not enforced but the developers we free to implement it. During the year-long measurement period, the mean proportion of pair programming was 34% of the total work-time, varying between 21% and 56% on a monthly basis. Two of the 17 developers joined the team at the start of the study, and they were studied in particular as they represented novices in the team. For them, the mean proportion was 46% of the total work-time, varying between 3% and 100% per month.

Williams et al. (2004) studied a team developing a product at IBM. They analyzed data from two consecutive release projects following the adoption of XP. For the latter release, people were given a choice of pairing, inspect-

ing, or justifying why code was written by one developer working alone. Based on the entries in the source code file headers, the amount of modifications made using pair programming increased from 11% to 48% between the releases. Based on a survey answered by the developers, the time spent using pair programming increased from 32% to 68% between the two releases.

Hulkko and Abrahamsson (2005) studied four different projects in close-to-industry settings containing both students and professionals. The projects lasted 5–8 weeks and contained 4–6 developers. The projects used a development method based on XP and SCRUM, and the use of pair programming was encouraged. In all cases, the proportion of pair programming relative to all programming effort was 75–95% during the first two weeks. Thereafter, it started to decrease, in two cases dropping to 40% and 20% respectively, and in the other two cases fluctuating between 60% and 100%.

There are at least 7 more papers studying industrial settings (see Publication I) that report some kind of a figure regarding the realized total proportion of pair programming. The figures vary between about 30% and 70%.

All the above studies report rather high levels of pair programming application. However, it must be noted that the studies are heavily biased toward XP-based contexts. The literature focuses mainly on reporting the realized proportions of pair programming from all work. There are no good studies of the desired amount of pair programming. Neither are there good studies about the possible effect of the amount of pair programming on the strength of the expected effects of pair programming, such as overall project productivity.

2.2 Targets of pair programming

The targets of pair programming cover the activities and situations for which pair programming is used. The perspectives that are of interest here include the suitability and amount of pair programming for the various targets. The targets can be classified by their type (e.g., programming or testing) and by their characteristics (e.g., complex or easy work).

Arisholm et al. (2007) found that task complexity affects the effects of pair programming on effort and quality. They conducted an experiment where 295 professionals performed artificial programming tasks for eight hours. The differences between pair programming and solo programming were studied using two tasks of different complexity and junior-junior, intermediate-intermediate, and senior-senior partner combinations as moderating factors. For the more complex task, pair programming increased the proportion of correct solutions by 48% and effort by 112% when considering all subjects. Pair programming increased correctness considerably for the junior and intermediate pairs, but did not affect it for the senior pairs, even though the effort increased considerably for all types of pairs. For the simpler task, there was no significant difference in correctness between pair programming and solo programming, but the effort was 60%

greater for pair programming when considering all subjects. Pair programming increased correctness slightly for the junior pairs but took more than twice the effort of solo programming. For the other pairs, pair programming decreased correctness slightly for the simpler task, but still increased effort.

The results of the experiment by Lui and Chan (2006) can be analyzed from the viewpoint of task complexity. They compared individuals and pairs who wrote the same program from scratch on four consecutive weekends, each time requiring several hours of effort. The task of writing the same program can be assumed to become easier every time it is performed. The subjects were part-time students who also had full-time programming jobs. The pairs practiced pair programming before the experimental tasks were conducted. The effort increase due to pair programming grew continuously from 29% for the first and most complex task up to 91% for the last and easiest task. The result indicates that pair programming is clearly less suitable for simple tasks than for complex tasks.

Back et al. (2004) report on the distribution of pair programming between various activities in an XP project conducted by hired students. The realized proportion of pair programming was 79% for programming, 77% for refactoring, 27% for debugging.

Schindler (2008) surveyed 42 Austrian organizations. He found that of those organizations that used pair programming, about half used it only for complex tasks and about one tenth only for tutoring.

Bryant et al. (2006) found that task type affects the amount of mutual participation in the pair programming sessions. They transcribed the communication of experienced pair programmers in 36 pair programming sessions in industrial settings; then, they split the communication into subtasks and classified their type. The proportions of sub-tasks on which both partners collaborated verbally varied between 81-95% among the sub-task types. The collaboration by both partners was most typical for understanding the problems or existing code (95% of all sub-tasks), corresponding with a third party (95%), writing new code (95%), and refactoring (94%); it was least typical for configuring the environment (81%) and commenting code (83%).

Dozens of other papers studying industrial settings (see Publication I) report anecdotal comments on suitable targets of pair programming. Use for complex tasks is the most frequently mentioned target, and tutoring new developers another frequently mentioned target.

Based on the fine study by Arisholm et al. (2007), it seems that task complexity is an important context factor of pair programming. The anecdotal evidence from numerous experience reports supports the findings of Arisholm et al. (2007) that pair programming works better for complex tasks. However, other rigorous studies concerning the role of complexity or other characteristics of the targets do not exist.

2.3 Infrastructure for pair programming

The infrastructure of pair programming is constituted by computer hardware and software, furniture, office layout, and noise in the workspace. The points of interest include the questions of how these aspects are organized and how that organization affects the other factors of pair programming.

There are no good studies concerning the infrastructure of pair programming, either treating industry or academic settings, even though there are dozens of papers (see Publication I) which describe shortly the infrastructure used in a certain organization in industrial settings. Sometimes, these studies mention anecdotal negative or positive experiences related to the used infrastructure. Negative experiences include, for example, inconvenient desks, small cubicles, small displays, and noise disturbing other people. Noise is also often mentioned as a positive aspect improving information exchange.

2.4 Managing pair programming

Managing pair programming concerns deciding on the use of pair programming, assigning pair programming tasks, scheduling pair programming, and degree of collaboration when using pair programming for a task. The points of interest include how these aspects are organized and how that affects any other factors of pair programming.

In an experience report, Belshee (2005) reported on an industrial XP team where assumedly all tasks were done using pair programming. He measured the effects of various task ownership, task assignment, and partner rotation frequency alternatives to the velocity of each iteration. Individually owned tasks (i.e., tasks where the owner stayed with a task until it was finished) resulted in lower velocity than team-owned tasks, where neither of the original partners needed to stay with the task. Tasks assigned per iteration resulted in lower velocity than tasks assigned just-in-time. Optimal partner rotation frequency was 90–120 minutes. Lacey (2006) tried to replicate the results using team-owned tasks with 2-hour partner rotation frequency in another industrial XP team, but in that context the velocity dropped considerably and remained low up to the end of a one-month observation period.

In a controlled experiment with a few dozen professional or student subjects, Domino et al. (2007) studied, for instance, the effect of full or partial collaboration in pair programming, or no collaboration at all, on code accuracy and developers' satisfaction with their work method. In partial collaboration, the partners prepared for coding together, but coded alone. The code accuracy was lower for full collaboration than for partial collaboration or no collaboration. Satisfaction was highest for full collaboration, next highest for partial collaboration, and worst for no collaboration.

Begel and Nagappan (2008) queried 487 developers at Microsoft about, for example, the most common problems in pair programming. Two of the top-10 problems were related to managing pair programming. Scheduling

pair programming ranked second and difficulties in finding a partner seventh.

Dozens of other papers from industrial settings (see Publication I) report individual points of information on managing pair programming. They propose, for instance, assigning tasks to pairs in a daily meeting, but without more thorough evaluation on the mentioned aspects.

Managing pair programming is a broad topic that includes many practically relevant aspects whenever pair programming is used. Despite this, good studies of this topic in industrial setting are scarce.

2.5 Adoption of pair programming

The adoption of pair programming analyzes all other aspects of pair programming with reference to how they motivate the decision to start using pair programming, or how they prevent or support the achievement of the desired use of pair programming in an organization. In addition, an interesting point to note is the general difficulty level of adopting pair programming, as for example, compared to other practices.

Schindler (2008) surveyed 42 Austrian software development companies and found that the most typical reasons for not using pair programming were that there was no need (mentioned by 30% of respondents), that it was too expensive (24%), that there was no time (21%), that it halved productivity (15%), and that it was used only with complex code (15%). These reasons indicate that many respondents believed that pair programming increases effort and is not to be used at all or only for complex tasks.

An experience report by Sharifabdi and Grot (2002) presents instructions for project managers who want to adopt pair programming despite team resistance, as was the situation in their XP project, where pair programming was to be used for all work. They propose using pair programming first only for task planning, and once it is shown to work well, also for programming. They propose, for instance, stressing the importance of mutual feedback in pair programming sessions, giving personal feedback if undesired behavior is observed, and coming in as a third wheel in pair programming sessions if needed.

The general difficulty of adopting pair programming was studied in a survey by Misic (2006), where 86 persons from different organizations evaluated the difficulty of adopting XP practices. Pair programming was the second most difficult one to adopt, but the mean difficulty level was not higher than 3.18 on a 5-point scale, where three meant “neutral” and five “very difficult”.

There are dozens of other papers containing some empirical information on the adoption of pair programming (see Publication I). However, they are mostly experience reports that mention individual difficulties in or aids for adopting pair programming. Examples of difficulties and aids are listed in **Table 4**, page 20. There is lack of good studies on, for example, evaluating the effects of the aids in industry. Also, besides the survey by Misic (2006), there are no broad studies about the general difficulty of adopting pair programming in industry.

2.6 Productivity

Productivity covers the development effort, duration and scope-related effects of pair programming. The most interesting point here concerns the effect of pair programming on any of these aspects of productivity.

Productivity is one of the two factors covered in the systematic literature review of the pair programming experiments (Hannay et al., 2009). The review analyzed 11 experiments that compared the required amount of development effort between pair programming and solo programming. Four of the experiments used professionals as subjects. The meta-analysis showed a medium negative overall effect on effort. The result of the meta-analysis is aligned with the broad survey by Begel and Nagappan (2008), where cost efficiency was clearly the most frequently mentioned problem in pair programming.

However, the meta-analysis concluded that inter-study variance was high and proposed focusing on studying the possible moderating factors of the effects of pair programming (Hannay et al., 2009). The experiment by Arisholm et al. (2007), covered in the meta-analysis, found that task complexity and partners' experience level affected the effort differences between pair programming and solo programming. Moderating factors related to managing pair programming were studied by Belshee (2005), as discussed already in section 2.4. Learning-time as a further moderating factor of productivity can be analyzed based on the results of the student experiments by Williams (2000) and by Nawrocki and Wojciechowski (2001). In both of these experiments, the effort increase due to pair programming compared to solo programming decreased as the pairs had gained experience in the use of pair programming and worked together for a longer time.

A few dozen other papers from industrial settings (see Publication I) report anecdotal productivity-related comments. Both positive and negative comments are equally presented in the different papers.

It seems that there is increase in the required amount of development effort due to using pair programming in the context of individual tasks. This is probably the main reason threatening the overall utility of pair programming. If the effort increase cannot be compensated for by the various proposed benefits of pair programming in the long run, the usefulness of pair programming is questionable. However, rigorous studies of industrial settings evaluating these long-term effects do not exist.

2.7 Software quality

Software quality covers all quality-related effects of pair programming, such as defects in code and maintainability of code and design. The most interesting point is the effect of pair programming on any aspect of software quality.

Software quality is the other factor analyzed in the systematic literature review by Hannay et al. (2009). The review analyzed 14 experiments that compared quality between pair programming and solo programming. Four

of them used professionals as subjects. The meta-analysis showed a small positive overall effect on quality. The result of the meta-analysis is aligned with the surveys by Begel and Nagappan (2008) and by Schindler (2008) where quality-related benefits where the most frequently mentioned benefits.

However, as mentioned above, the meta-analysis concluded that inter-study variance was high, and the results by Arisholm et al. (2007) regarding the role of task complexity and partners' experience level also applied to quality. Similarly, a multiple case study by Hulkko and Abrahamsson (2005) reported mixed results, where code written using pair programming had a higher comment ratio but more deviations from coding standards, and the results regarding the defects were inconclusive.

Dozens of other papers from industrial settings (see Publication I) report anecdotal quality-related comments. Most of them mention a positive effect on some quality-related aspect.

Based on the meta-analysis of the pair programming experiments (Hannay et al, 2009) and the general trend of the results from the less rigorous studies, it seems that pair programming has a positive effect on quality. However, there may be moderating factors that affect the effects of pair programming on quality.

2.8 Developer's knowledge of work

The developer's knowledge of work covers all work-related aspects of knowledge such as the developed software, problem domain, development tools, and work practices. The most interesting point to observe here is the effect of pair programming on the changes in a developer's knowledge compared to solo programming.

In the survey by Schindler (2008), "knowledge transfer" was the second most commonly mentioned advantage of pair programming just behind "permanent reviews" but before "increased code quality." In the survey by Begel and Nagappan (2008), two of the four most commonly mentioned advantages of pair programming were related to increases in developers' knowledge and the other two to increases in code quality. In the survey by Williams (2004), the respondents estimated that pair programming can more than halve the time lost to assimilate a new employee.

Auvinen et al. (2006) measured the developers' knowledge of the developed software using subjective evaluation and a short quiz before and after piloting pair programming for three months in a team. The knowledge of all developers improved considerably after that period, but the data did not allow evaluating how much of the improvement was due to pair programming.

Dozens of other papers from industrial settings (see Publication I) report anecdotal experiences from individual projects of the effects of pair programming on the developers' knowledge of work. Typically they mention some positive effects on developer's knowledge of the developed software or on general knowledge transfer.

In an experiment by Bellini et al. (2005), dozens of students conducted a small software design task as pairs or individually. The students who worked in pairs improved their knowledge of the design more than those who worked alone.

In addition to the experiment by Bellini et al. (2005), there are no rigorous studies from industry or academia on the effect of pair programming on increasing developers' knowledge. However, the developers' perceptions reported in the studies above all seem to be aligned with the view that pair programming has a positive effect on developers' knowledge of work.

3 Research design

In this chapter, we present the utilized research methodologies on a general level, and their application in our study. In the research environment section, we introduce the two organizations where the case studies were conducted and the context of the student experiment.

3.1 Research methodologies

We used three research methodologies, each of which is described in a separate section below. The systematic mapping study methodology is a type of literature research, and the case study and experiment are methodologies for empirical research.

3.1.1 Systematic mapping study

The purpose of a systematic mapping study is to give an overview of a research area (Petersen et al., 2008). It identifies the quantity and type of research, as well as the results available within the research area; it also often shows yearly publication trends and identifies used publication forums (Petersen et al., 2008).

Systematic mapping studies are similar to systematic literature reviews in the sense that both aim at providing a trustworthy, rigorous and auditable methodology to identify and analyze all available research relevant to a particular research topic (Kitchenham and Charters, 2007). However, systematic mapping studies generally present a larger number of research questions, which also tend to be broader (Kitchenham and Charters, 2007). Systematic mapping studies also cover more studies and present results as summaries of classifications of the included studies instead of synthesizing their results (Kitchenham and Charters, 2007). A systematic mapping study may also go deeper into the papers, as for instance, due to poor abstracts, and thus become more like a systematic literature review (Petersen et al., 2008).

We used the systematic mapping study method in Publication I. Our systematic mapping study was rather deep as we analyzed the full content of the papers. However, it was not a systematic literature review because we did not interpret or synthesize the results presented in the included papers.

The review protocol presented in detail in Publication I followed the guidelines for performing systematic literature reviews by Kitchenham and Charters (2007), with reference to the steps for searching and selecting studies. Comprehensive searches of seven databases, resulting in 1749 hits, were complemented with certain manual searches. All scientific papers containing empirical data on pair programming used by professional developers were included in the study, resulting in a total of 154 papers. No quality criteria were applied when selecting the papers.

The guidelines by Kitchenham and Charters (2007) give very little advice on how to undertake data extraction and analysis for a systematic mapping study in place of a systematic literature review, and their methods needed

to be adapted for our purposes. We created a tentative list of pair programming factors based on a subset of the included papers and previous pair programming frameworks (Gallis et al., 2003; Ally et al., 2005). We modified and refined the list slightly during the data extraction from all the included papers.

During the data extraction, all pair programming-related data in each paper was classified according to the factors of pair programming (**Table 4**, page 20) and numerous predefined categories of the research properties to be analyzed (**Table 2**). Certain research properties, such as "publication forum," are common to all pair programming data in a paper, but certain others, such as "data collection method," are specific to each factor discussed in a paper.

After the classification, we analyzed the distributions of data among the factors and among the various categories of each research property. We also summarized the most relevant studies related to each factor.

Table 2 Main research properties used in the classification of research

Property	Level	Categories
publication forum	paper	• journal • conference/workshop
paper focus	paper	• pair programming (is one of the main focuses) • other
authors' role	paper	• internal (i.e., at least one author worked in the same organization as the studied subjects) • external, includes also visitors who worked at most a month in the studied organization
research approach	paper	• experiment • survey • case study (i.e., an in-depth, possibly multi-method study of one or a few cases) • experience report (i.e., personal experiences from some case(s) without reporting the use of any scientific data collection method)
data collection method	factor	• measurement (i.e., data collection where the error caused by subjectivity is small) • rigorous observation (e.g., audio/video tapes, or someone making rigorous notes on-site) • interview • questionnaire • informal observation (e.g., an author was present, but the use of any data collection method is not reported) • defined, fixed by the authors (e.g., controlled variables in experiments)
data type	factor	• quantitative • qualitative
discussion type	factor	• comparative (i.e., evaluates how this factor was affected by some variation, or how variation in this factor affected some other factor). Comparative claims based on informal observation only are classified as descriptive. • descriptive

3.1.2 Case study

A commonly proposed definition for a case study is that it is an empirical method aimed at investigating contemporary phenomena in their context (Runeson and Höst, 2009). Even though case studies were originally used primarily for exploratory purposes, they can be used for four types or purposes of research: exploratory, descriptive, explanatory and improving (Runeson and Höst, 2009).

We used the case study methodology in two different studies, referred to hereafter as Study A and Study B. By the term "case," we refer to the application of the pair programming practice in the corresponding organization. In both cases, we used triangulation to increase the precision of the research. We used several data collection methods, including questionnaires answered by the developers, interviews and observations of the developers, and the measurement of the work products. We collected data from many subjects, and in the questionnaires we collected both qualitative and quantitative data about the same topics. In Study B, two researchers participated in collecting the data.

In Study A, the main data collection method was a questionnaire containing both open and closed questions targeted at all developers in the case organization. The questionnaire was used four times during a 2-year period, with slightly modified questions. The questionnaire inquired on the developers' perceptions of the effects of pair programming, and the developers' experiences of various aspects of the use of pair programming.

Lots of preparations were done to ask insightful questions in the questionnaires. The preparations included studying literature on pair programming, an interview of a team that had piloted pair programming in the organization before the study, informal discussions during the study with the person responsible for the adoption of pair programming, and three observations of pair programming sessions combined with interviews of the involved pairs.

We also measured defects found in code reviews to evaluate the effects of pair programming on quality, but there were too many factors affecting the defect count and the information could not be used for the intended purpose. Other measurements were not conducted, as the case organization was not willing to add overhead to the developers, and the existing reporting system did not provide relevant data, such as the effort spent using pair programming.

In Study B, the second researcher acted also as a developer in the studied project, which allowed more detailed data collection. He had first-hand information on basically everything that occurred in the project. He ensured that the effort and quality data was collected in a timely way, and he collected the developers' perceptions on various topics by conducting surveys with closed questions at the end of each two-week iteration, and at the end of the project. At the end of the project, the author of this dissertation conducted a team interview complemented with a questionnaire containing closed questions about feelings on and effects of pair programming.

3.1.3 Experiment

In an experiment, the objective is to manipulate one or more factors and control all other factors at a fixed level (Wohlin et al., 2000). Manipulation involves at least two different treatments to compare their effect on the outcome (Wohlin et al., 2000).

We used the experiment research methodology with student subjects on a university course, where adequate control over the factors is easier and cheaper to arrange than in industrial settings. The subjects were assigned randomly to five four-person teams, which were the experimental units of the experiment. We used a randomized one-factor design (Juristo and Moreno, 2001). The treatment used either pair programming or solo programming as the programming method. The treatment was assigned randomly for each team. All the other factors, such as other development practices and tasks to be conducted, were fixed as rigorously as possible.

We analyzed several outcome variables, including effort, software quality, developers' knowledge of the developed system, and developers' enjoyment of the programming method. The data was collected using many different methods. The teams reported manually the effort spent per use-case or other type of task, and defects found in their systems. We tested the final systems and counted the number of successfully implemented use-cases and the defects we found. We measured various source code metrics of the final systems. At the conclusion of the project, the developers answered an online questionnaire covering their understanding of each module in the system and their enjoyment of the used programming method.

3.2 Research environment

We conducted the empirical studies both in a realistic software development context, where professional developers were doing their daily work in industrial settings (Study A and Study B), and in an artificial context, where students conducted course exercises (Study C). The contexts of these studies are described in more detail below.

3.2.1 Study A

Study A took place in a medium-sized software company in a department developing a large, over ten-year-old software product. The adoption, use, and effects of pair programming were studied in the whole department. The data collection period spanned two years.

The department had about 30 developers divided into four independent teams, each having a senior developer acting as a team leader. All teams sat in cubicles in a large open office.

The department had an established development process, which was quite traditional rather than utilizing broadly any agile practices. Before the study, some of the developers had already informally used pair programming to a limited extent, and one team had done a small informal pilot study on pair programming, which produced positive experiences concerning its effects on knowledge transfer and software quality.

3.2.2 Study B

Study B took place in a research department of a large telecommunications company. The adoption, use, and effects of pair programming were studied in a project whose goals were to develop an internal reporting system and to pilot agile practices. The project successfully delivered software featuring 20,000 lines of code. The data collection period spanned the whole 3-month duration of the project, consisting of six delivery iterations.

The project was carried out by a newly hired four-person team. The willingness to use agile practices was ensured when recruiting the developers. All the developers were equal in terms of responsibilities, except that one of them acted also as a team leader who took care of, for instance, arranging the daily meetings. All developers sat in the same open office.

The team decided on which process to use, eventually settling on a collection of practices from several agile methodologies. Three of the developers had not used pair programming before the project and one had used it for about a month.

3.2.3 Study C

Study C took place within the context of a university course that focused on teaching Java 2 Platform, Enterprise Edition (J2EE). After 15 hours of lectures on J2EE, the students applied it in large team projects lasting nine weeks. The projects had identical goals and working methods, with the exception of the use of pair programming in half of the teams, and solo programming in the other teams. The effects of pair programming were studied.

The projects aimed at implementing as many of the specified use-cases as possible with a minimal number of defects, within a budget of 400 hours. All the teams had to work at least 75% of the time together in the same room.

There were five four-person teams. The teams were formed randomly, but in such a way that the members' average skill level was equal between the teams. The students had on average 5 years of programming experience. Three random teams were required to use pair programming for all of their development work, and two teams were not allowed to use pair programming at all. Pair programming was taught in a lecture.

3.3 Summary of the empirical studies

The industry-based organizations studied in the case studies were chosen from among organizations that engaged in research collaboration with our research group. Both of the industry-based organizations took the initiative to adopt pair programming on their own. After that, the author of this dissertation took the role of an objective observer rather than a person who was responsible for the adoption of pair programming. However, in Study A, we gave some instructions for using pair programming based on our experiences, and presented intermediate results of the study, which could have affected the adoption and use of pair programming.

The subjects of the experiment were selected by including all students who took an optional software development project course. All participants of the course knew in advance that half of them must use pair programming in their project.

With the students, we could enforce many different subjects to conduct the same tasks, using either pair programming or solo programming, and make objective measurements on the effects. In the industry, this would have been too expensive. Thus, in our industry-based studies, the evaluation of the effects of pair programming is based on the perceptions of the subjects who used both pair programming and solo programming in their work, but never for the same tasks by the same or different subjects.

The attributes of the empirical studies are summarized in **Table 3**. These include attributes related to the research methodologies and research environment.

Table 3 Summary of the empirical studies

Attribute	Study A	Study B	Study C
publications	II, III	IV	V
environment	industry	industry	academia
research methodology	case study	case study	experiment
duration of the study	2 years	3 months	9 weeks
data collection methods	questionnaires, interviews, observations	questionnaires, interviews, measurements, participation	questionnaires, measurements
unit of analysis	a department with about 30 persons in six teams	a four-person project team	five four-person project teams
history of the organization	an established organization developing an old software product	a new team developing new software	new teams developing new software
process	established, traditional development process	new, agile development process	new, rather agile development process
amount of pair programming	little pair programming	lots of pair programming	lots of pair programming / no pair programming

4 Results

This chapter presents the main results related to each of the research questions. The results include the summary of the potentially relevant factors of pair programming organized as a pair programming framework (Research Question 1), empirical results of the adoption and use of pair programming (Research Question 2), and empirical results of the effects of pair programming (Research Question 3). The discussion on the results and the limitations of the studies are presented in the next chapter.

4.1 Pair programming factors

We answered Research Question 1 by conducting a broad systematic mapping study of empirical studies on the use of pair programming by professional software developers. We identified dozens of aspects of pair programming from the 154 papers and organized them into a pair programming framework. The framework contains eighteen factors of pair programming that are further described in more detail through numerous examples of aspects of pair programming, as shown in **Table 4**.

The factors are grouped under six themes. The factors under the theme *preparations for PP* are related to the adoption of pair programming and to the recurrent preparations required when performing pair programming. The *environment* factors involve the software and hardware infrastructure, and the surrounding software development process with all of its practices. The *developer* factors are related to the properties of a developer. The *PP session* factors cover working in pair programming sessions. The *utilization rate* factors are related to the amount of using the pair programming practice, either locally within a single case or generally. The *main effects* contain two typically affected project attributes, productivity and software quality, whereas other affected factors are listed under the other themes.

The factors are non-overlapping with two exceptions. Firstly, adoption of PP overlaps with many other factors because difficulties in, aids for, and reasons for adoption are often related to other factors. Secondly, feelings on PP is actually only a part of feelings on work, but it is analyzed separately due to its importance for pair programming.

We classified the research done on each factor using many properties of research (**Table 2**). Only 18% of the papers were published in journals, and well-known XP/agile conferences were the most common forums. Only 7% of the data on the factors came from experiments, whereas 44% came from the least rigorous research approach used in our classification (i.e., experience reports).

Communication in pair programming sessions was the most thoroughly studied factor. The next most thoroughly studied factors were the commonly proposed effects of pair programming: developer's knowledge of work, productivity and software quality. For many factors, there were no or almost no comparative data, let alone data from reliable data collection methods such as measurement or rigorous observation. Further results of the classification are presented in Publication I.

Table 4 Pair programming framework

Theme	Factor	Examples[a]
Preparations for PP	Adoption of PP	**Difficulty level**: compared to other (XP) practices, length of learning time. **Difficulties**: organizational culture, management resistance, evaluation of personal contribution, lack of partners due to 1) different work schedules, 2) small team, or 3) distributed team. **Aids**: PP guidelines, PP training, PP champion, alternative for reviews, enforcement, limited number of workstations. **Reasons**: many of the expected benefits listed under other factors.
	Managing PP	**Deciding on PP use**: mandatory to use, who decides, when decided. **Assigning PP tasks**: practices such as a pair chooses in daily meeting; task ownership options such as "individual/pair" or "owned by workstation"; task ownership problems such as lack of accountability. **Scheduling PP**: practices such as allocating time for PP, problems such as experts working alone before DLs, common time not found or working away from office, accuracy of estimating PP tasks. **Degree of collaboration**: whole task together (default case), a task is split and both developers work alone for a while, only one person works for a while, synchronization after working alone.
	Pair formation	**Initial pair formation**: organized by managers, self-selected, ad-hoc, based on required skill set. **Partner rotation**: frequency, who continues with an unfinished task.
	Targets of PP	**Activities**: programming (default case), specification, design, refactoring, TDD, debugging. **Situations**: project initiation phase, new developers join the team, evaluating employee candidates. **Characteristics of targets**: task complexity.
Environment	Infrastructure for PP	**Hardware**: big screen, dual keyboard, two workstations, whiteboard, white noise generator. **Software**: large fonts, standardized tools. **Furniture**: shape of desks, whiteboard. **Office layout**: separate PP room, open office, cubicles. **Noise in workspace**: awareness, disturbance.
	Development process	**PP facilitates other practices**: TDD, coding standard, refactoring, collective ownership. **PP replaces other practices**: code review. **PP disturbs other practices**: individual performance evaluation. **Other practices facilitate PP**: such as test-driven approach, collective ownership, planning game. **Discipline within the process**: process conformance, concentration on work.

Table 4 (continued)

Theme	Factor	Examples
Developer	Feelings on PP	**Feelings**: resistance, satisfaction, enjoyment, "general" feelings.
	Feelings on work	**PP affects feelings about work**: team spirit, enjoyment, enthusiasm, exhausting, threatening, peer pressure. **PP is affected by feelings about work**.
	Knowledge of work	**PP affects knowledge of work**: developed software, tools, work practices, or domain, general knowledge of a new developer. **PP is affected by knowledge of work**: PP ability, work experience.
	Characteristics	**Demographics**: nationality. **Psychosocial factors**: personality, self-esteem, communication skills, conflict-handling style.
PP session	Partner combinations	**Combinations**: personality, work expertise, PP experience, age. **Viewpoints**: frequency of combinations.
	Partners' roles	**Characteristics of roles**: one leader, keyboard possession, level of thinking. **Switching the roles**: frequency.
	Communication	**Content**: abstraction level, representations used, value (such as usefulness). **Quantity**: number of utterances. **Issues**: solving disagreements, flow and mental blocks, speed of work such as slow enough for the junior pair or typing speed. **Partners' relationship**: getting to know the partner, courage to criticize the partner's work.
	Breaks	**Types of breaks**: intrusions, distractions and breaks **Viewpoints**: number of breaks, reasons.
Utilization rate	Local amount of PP	**Dimensions**: realized share of development work, realized rate, proposed rate, desired rate.
	Prevalence of PP	**Breadth of use**: worldwide, nationally, embedded software domain, departments of a global company. **Depth of use**: use on an ordinal scale (systematically–never), used vs. not used, using or planning to use.
Main effects	Productivity	**Dimensions**: effort/duration, scope, lines of code.
	Software quality	**Code**: defects, readability, comment ratio. **Design**: understandability, quality. **General**: confidence in results.

[a)] A reference to each example can be found in Publication I.

4.2 Adoption and use of pair programming

The adoption and use of pair programming are somewhat inter-related topics and were both studied in Research Question 2. In this section, we present first the results related to the adoption of pair programming and those aspects of the use of pair programming that were closely related to adoption. After that, we present the results related to other aspects of the use of pair programming.

The adoption and use of pair programming were studied in both case studies (Study A and Study B). Both the issues faced and their solutions when adopting pair programming were studied. In Study B, the adoption of pair programming was straightforward, and therefore most of the identified issues are from Study A.

4.2.1 Adoption of pair programming in Study A

In Study A, the motivation for adopting pair programming was to improve knowledge transfer between the developer's and software quality. Most of the developers had little or no pair programming experience. Pair programming had been used informally by a few developers with promising results. The official use of pair programming was launched by giving a lecture to all developers about pair programming in general and about the guidelines for its use in the organization specifically. One of the team leaders was in charge of the guidelines. The use of pair programming was voluntary, and the goal was to use it in situations when it was expected to be particularly beneficial, such as for knowledge transfer purposes or with difficult tasks.

The amount of pair programming remained at a very low level during the first year after the official adoption. During the second year, the amount increased by about 150%, but was still only about 10% of all development work.

Based on the data from the questionnaires, the developers' attitudes to pair programming were not a reason for the slow adoption. In the beginning of the second year, 60% of the developers desired more pair programming, and nobody wanted less. Despite the increase in the amount of pair programming during the second year, the developers' desire to use it more remained at the same level. The developers' initial feelings on pair programming were mainly on the positive side (median 5.0[1]), even though the responses of a few developers were on the negative side. By the beginning of the second year, all the responses were at or above neutral. By the end of the second year, the median of the developers' feelings of pair programming increased to 6.0 surpassing the median of the feelings of solo programming, which actually decreased from 6.0 to 5.0 by the end of the study.

The developers' perceptions of the effects of pair programming were not the reason for the slow adoption of pair programming, either, given that the

[1] In Study A, the developers' feelings on various topics were inquired about, on a scale of 1–7, where 1=negative, 4=neutral, and 7=positive.

perceptions were generally positive, as will be discussed in Section 4.3. Other possible reasons for the slow adoption were studied in more detail in the third questionnaire in the beginning of the second year. The questions focused on the organizing and infrastructure of pair programming, which had been somewhat problematic areas during the first year. The organizing referred to pair formation, finding common time for pair programming, etc. The infrastructure referred to the physical setting of the company, such as equipment and rooms.

In the beginning of the second year, the developers' feelings about the organizing for pair programming were on the positive side (median 5.0) but a few responses were on the negative side. The issues were mainly related to the resourcing of pair programming, for instance, being too busy to use pair programming relative to the other developers' tasks, difficulties in finding common time, lack of encouragement from team leaders, and lack of considering pair programming in project planning.

Based on the findings, the pair programming guidelines were updated to create a more encouraging and more positive atmosphere for pair programming. To ensure the presence of enough resources for pair programming, its use was to be planned well in advance, including for which tasks it would be used, by whom and for how large a proportion of a task's various activities, such as specification or programming. In addition, the team leaders should encourage using the planned proportion of pair programming for the selected tasks. By the end of the year, no developer had negative feelings about organizing for pair programming anymore, but otherwise the change in the feelings was small and the median remained the same (5.0). The developers' comments in the last questionnaire indicated that the issues had not been completely removed.

In the beginning of the second year, the developers' feelings on the infrastructure of pair programming were only neutral (median 4.0) and many developers had negative feelings. A frequently mentioned problem was that the noise from pair programming disturbs the other developers in the open office. Therefore, pair programming was sometimes done in a meeting room using a developer's laptop. Other problems were inconvenient, cramped desks and small displays.

Based on the findings, a separate pair programming room was adopted for use at the beginning of the second year. It contained a desk with two computers having large displays, a long, straight table, rolling chairs, and a whiteboard. The pair programming room could be reserved in advance. By the end of the second year, 89% of the developers considered the pair programming room as the preferred place for pair programming. According to the developers, the reasons for its popularity were the avoidance of noise and better infrastructure for pair programming. The only benefits of doing pair programming in the open office instead of in the pair programming room were the simplicity of doing ad-hoc pair programming sessions, and being closer to the other developers if help was needed. By the end of the year the median of the developers' feelings about the infrastructure increased from 4.0 to 6.0.

4.2.2 Adoption of pair programming in Study B

In Study B, the motivation for adopting pair programming was related to the goal of experimenting with agile practices in the studied project. At the beginning of the project, the developers' attitudes towards pair programming varied from slightly negative to quite positive.

Pair programming was used a lot straight from the beginning of the project, and its total share of all the programming work during the project was 72%. There was a simple tactic for the adoption: The four person team was given only two high-end workstations to work with along with two low-end ones. All of the developers considered the adoption of pair programming easy, both absolutely and relative to the other practices used in the project, such as writing unit tests or test-driven development.

4.2.3 Use of pair programming

Regarding the targets of pair programming, the developers in both case studies found pair programming most suitable for complex tasks. In Study A, the developers were surveyed on the percentage of pair programming that should be used for the various activities. For the planning and design activities, the average was 70–80%, whereas for coding it was 60% and for testing 50%.

Regarding the infrastructure of pair programming, the default setting was to have one workstation for a pair, in both case studies. However, the introduction of the pair programming room in Study A allowed using two workstations side-by-side. On the other hand, the developers evaluated that they still spent 85% of the time working together at the same workstation, and the other workstation was used, for example, for browsing specifications or code, finding information, testing and debugging.

Regarding the pair formation, the developers in Study A had mixed opinions on the best way to do it. Many developers considered that the developers should have the final decision on the choice of the partner. On the other hand, some developers wanted the team leaders to be more involved in pair formation and supervising the performance of planned pair programming. In Study B, the pairs were formed in a daily meeting, and at first, the pairs remained together the next day if their tasks were unfinished. Later, the pair formation was done by casting a lot in every daily meeting to ensure frequent pair rotation, which was expected to increase knowledge transfer.

Regarding the pair programming sessions, the developers in Study A considered 1.5–4 hours a suitable duration. Shorter sessions were considered inefficient due to the set-up time required, and longer sessions were considered too exhausting. In Study A, only half of the developers switched keyboard possession during a pair programming session. Preferring different development environments was mentioned as a reason for not switching keyboard possession. In Study B, a pair could spend the whole day using pair programming, and keyboard possession was switched 2–3 times per day, typically after having a lunch or some other break.

4.3 Effects of pair programming

Below, we present the main results of our empirical studies related to the effects of pair programming compared to solo programming (Research Question 3). The results cover the effects to productivity, software quality, and developer's knowledge of work. All of these effects were studied both in the experiment (Study C) and in the case studies (Study A and Study B). Additional results related to certain other effects of pair programming, such as enjoyment of pair programming, are presented in Publications II–V.

4.3.1 Productivity

In Study C, the pair programming teams had 29% lower project productivity than the solo programming teams on the average, when considering the amount of the delivered functionality within the fixed project effort. When the effort spent for implementing the individual use-cases was considered, the pair programming teams spent 107% more effort, i.e. over double effort, on the first four use-cases than solo programming teams. However, for the next six use-cases pair programming teams spent 5% less effort than solo programming teams. The effort difference per use-case between the pair programming and solo programming teams was not affected by the perceived complexity of a use-case.

In Study A, the developers' perceptions of the effect of pair programming on the total development effort of individual tasks varied a lot among the developers. The answers were distributed between 2 and 7 on a 7-point scale, and the median was 5.0[2], indicating that pair programming takes somewhat more effort than solo programming.

Also in Study B, the developers' perceptions of the effect of pair programming on the development effort varied among the developers on both sides of neutral, and based on the median, pair programming took somewhat more effort than solo programming. However, the developers chose pair programming among all practices used in the project as the practice that most improved the productivity of the project. The developers commented that for complex tasks, the use of pair programming may even lower the total effort, but for simpler task it takes more effort than solo programming.

4.3.2 Software quality

In Study C, two defect counts were analyzed. The first defect count contained the defects found by the team related to the use-cases that the corresponding developer/pair already considered ready. These defects were found by the team during the development of the further use-cases or in the system testing conducted by the team. The second defect count contained the defects found by the researcher, who conducted system testing after the

[2] In Study A, the developers' perceptions of the various effects of pair programming compared to solo programming were inquired about, on a scale of 1–7, where 1=lower, 4=same, and 7=higher.

team had fixed the defects found by the team members themselves and delivered the system. The sum of both defect counts is an estimate for the number of defects that existed in code that was considered ready by its developer(s). The defect densities (i.e., the defect counts normalized by the number of implemented use-cases per team) were compared between the pair programming and solo programming teams.

The defect density calculated based on the sum of both defect counts was 8% lower for the pair programming teams (i.e., in the pair programming teams, there were fewer defects in code that the corresponding developers considered ready). However, the solo programming teams found more of their defects during further development and system testing, fixed them, and finally delivered systems with a lower defect density. The defect counts after the delivery were generally very low: only 1–6 defects per system in each team.

Code metrics were calculated from the source code of each delivered system. The pair programming teams had slightly better design quality based on the method size and complexity metrics.

In Study B, the developers perceived that pair programming decreased the number of defects in code and increased the understandability of design. The average number of defects found in the production use of the system was less than one defect per thousand lines of code per release, thus supporting the perception of high quality. The developers considered pair programming as the second most important practice after test-driven development for increasing the quality of the system and its design. However, the developers commented that the navigator (i.e., the person without possession of the keyboard) did not spot many defects during the pair programming sessions.

In Study A, the developers perceived that pair programming had a small positive effect on decreasing defects in code, understandability and maintainability of code, and customer satisfaction. The median of the perceptions for each quality aspects was 4.5–5.0 on the 7-point scale, and there were no answers on the negative side for any of the quality aspects.

4.3.3 Developer's knowledge of work

In Study C, all developers evaluated their level of understanding of each of the ten source code packages at the conclusion of the project, on a 5-point scale, where "5 - very much" indicated the deepest level of understanding. In the pair programming teams, the developers understood at least "4 - quite a lot" of 4.5 modules on the average compared to 3.4 modules in the solo programming teams. At the other levels of understanding, there were practically no differences between the pair programming and solo programming teams.

In Study B, each developer evaluated his knowledge of each module after each of the first three iterations on a 5-point scale. The developers' knowledge remained at a high level in all iterations even though the system grew and became more complex. The average of all evaluations per iteration was 3.7–3.9 of the maximum of 5. The differences in the understanding of

each module were rather small among the developers and decreased even more in the third iteration, probably due to the higher frequency of rotating pairs in that iteration.

In Study B, all developers perceived that pair programming increased their knowledge of the system. The developers also ranked pair programming as the most important practice for increasing team communication.

In Study A, all developers perceived that pair programming had a positive effect on learning about the developed system compared to solo programming. The median was 6.0 on the 7-point scale and there were no answers below 5.

5 Discussion

This chapter discusses the main results of each research question in the separate sections. Then the research is evaluated considering both its main strengths and limitations.

5.1 Pair programming factors

Research Question 1 covers the identification of the factors of pair programming, and analysis of the properties of the previous empirical research regarding the identified factors. These points of view are discussed below.

Our pair programming framework (see **Table 4**) includes the content of the existing pair programming frameworks (Gallis et al., 2003; Ally et al., 2005), with some changes in the naming and grouping. For example, the role of pair programming in decreasing software development project risks is not explicitly mentioned in our framework because almost any benefit of pair programming can also be seen as a way to avoid some risk. In our framework, breaks and prevalence are new factors compared to the previous frameworks. Also, the examples in our framework include many additional or more detailed aspects of pair programming over the previous frameworks. However, our framework may still lack some aspects of pair programming relevant to industry, given that in the systematic mapping study we scoped out papers treating the educational context, as well as theoretical papers and non-scientific practitioner literature.

Our analysis of the research reported in the empirical papers showed that there is scarcity of data in the categories that indicate high-relevance research. These categories include experiments as the research approach, data collected using measurements or rigorous observations, and data of comparative type. However, reacting to at least some of these gaps should not be too difficult. We believe that the lack of good data on certain factors may be also due to not considering them in the study designs in addition to the difficulty of studying them (e.g., due to the high costs of conducting experiments in industrial settings). For example, changes in developers' knowledge of work have not been measured even though it would not be very difficult to test the changes before and after using pair programming. Some previous studies' designs, with small improvements in their data collection, could be used to fill some gaps. For example, more information on developer characteristics and previous experience, partner combinations, and characteristics of the tasks could have been collected rather easily in many studies. New studies that would be realistic to conduct could include, for example, experiments on the effects of infrastructure, such as displays of different size, one vs. two displays or keyboards, or different desks.

5.2 Adoption and use of pair programming

5.2.1 Adoption of pair programming

The survey by Schindler (2008) mentions “no time” and costs as the most typical problems preventing the use of pair programming. The former aligns with the resourcing problem in Study A. The latter was not mentioned as a problem in Study A, even though many developers perceived that pair programming takes somewhat more effort per task. Because the developers perceived many benefits from pair programming, the increase in the task-level effort probably was not a significant issue, considering overall productivity.

The problem with noise, as identified in Study A, is not mentioned in the papers discussing the adoption of pair programming identified in Publication I, a study by Fitzgerald et al. (2006) being an exception. However, most of the papers discussing the adoption of pair programming are from the XP context, where a single team shares a room and all developers use lots of pair programming. Thus, in XP, overhearing in the room is considered a valuable communication channel. By contrast, in Study A, the typical way of working was to work alone, and several different teams were working in the same open office. A single team working with inter-related tasks may consider the noise as useful information, but for the other teams this no longer applies. The problems with the noise and cramped desks were solved in a rather straightforward way by adopting the pair programming room, where one pair at a time could work together without disturbing the other developers.

In Study A, the problem with resourcing pair programming (i.e., the lack of time for pair programming due to insufficient organizing of its use) partially remained at the conclusion of the case study. It is a more difficult problem than those related to infrastructure because it has tight dependencies to established processes such as work planning. It also has political aspects, such as whether a developer prioritizes his or her own tasks over helping other developers with their tasks in a situation where a developer’s performance may be evaluated based on her or his own tasks instead of the team’s overall productivity.

It has been proposed (e.g., Cockburn, 2000; Johansen, 2001) that people must try pair programming before they accept it. In Study B, the tactic of adopting pair programming by limiting the number of work stations clearly solved this potential problem by practically enforcing its use. However, in Study A, the problem actually was not with the developers’ attitudes towards pair programming, and similar enforcement could not be used in any case because the goal was not to use pair programming for everything.

In Study A, based on the developers’ evaluations of and comments on the infrastructure and organizing of pair programming, it is likely that the pair programming room was the main reason for the 150% increase in the amount of pair programming. Similarly, it is likely that the main reasons preventing many developers using pair programming to the extent they desired were the remaining problems in resourcing pair programming.

The contexts for Study A and Study B were very different with regard to adoption of pair programming. In a new team with no existing development process and with a mindset of experimenting with new agile practices, the adoption of pair programming was very easy. In an established development organization, implementing any change is likely to prove more challenging, even when the attitudes towards the change are positive. Therefore, carefully considering the potential difficulties in the adoption of pair programming is highly important in such as context.

5.2.2 Use of pair programming

The recommendations from both case studies for using pair programming for complex tasks and design activities are in line with the results from other industry-based studies (Arisholm et al., 2007; Schindler, 2008). However, in Section 5.3.1, we discuss our experiment, where the task complexity did not affect the differences in effort between pair programming and solo programming.

The infrastructure of pair programming was already discussed under the adoption of pair programming, as it contained issues that were so serious that they hindered using pair programming to the extent desired. An additional result related to the infrastructure was the finding in Study A that despite having two workstations side-by-side available for a pair in the pair programming room, the partners still spent most of the time working together at the same workstation. Thus, the other workstation did not lead to abandoning pair programming, but was utilized sometimes for certain specific tasks such as finding some information related to the task being performed together.

In Study A, the developers had mixed opinions on the responsibilities of forming the pairs regarding whether that was to be accomplished by the developers themselves or by the team leaders. It may be that, for instance, junior developers would appreciate the participation of the team leaders in assigning senior developers as partners for junior developers. In Study B, the self-formation of the pairs among developers was replaced by a random pair formation every morning in order to increase knowledge transfer within the team through the balanced use of different pairs.

In Study B, the developers switched keyboard possession 2–3 times per day, but in Study A, half of the developers did not switch at all during the pair programming sessions. However, the potentially resulting passivity from this did not take place, based on the developers' comments or our observations of the pair programming sessions. The small proportion of pair programming and short pair programming sessions could be an explanation for this.

5.3 Effects of pair programming

5.3.1 Productivity

In Study C, the worse total productivity of the pair programming teams compared to the solo programming teams resulted from the huge effort expended on the first four use-cases by the pair programming teams. This phase can be considered as a necessary learning period, which can be due to, for example, being unfamiliar with the partner or with the pair programming practice. In the student experiments by Williams (2000) and by Nawrocki and Wojciechowski (2001), the increase in effort due to pair programming also decreased in later tasks, but not as much as in Study C. However, in those experiments, the total duration of the conducted tasks was shorter than the duration of implementing the first four use-cases in Study C, which may mean that in those experiments the effort increase could have decreased further if they had lasted longer.

In the long run, in a typical software development organization, the costs of the learning period can usually be neglected. After the learning period, the effort spent for each use-case in Study C was almost equal between the pair programming and solo programming teams. However, it must be noted that the double effort spent with the first use-cases may have decreased the effort spent for the next use-cases, because the pair programming teams may have gained deeper understanding of the core system while spending more time with the first use cases.

In Study A and Study B, the large variation in the developers' perceptions on the effect of pair programming on task efforts suggests that there are some context factors that affect the effect. For example, a senior and a junior developer may perceive opposite effects for the task they are working on together, if they compare the realized effort to what the task would have required from them alone. In addition, at least the type and complexity of a task, and the proportion of pair programming used for a task, may affect the effect on effort.

In Study A and Study B, the developers commented that pair programming is more suitable for complex tasks. However, in Study C, the measured effort differences between pair programming and solo programming did not correlate with the task complexity. One reason for the inconsistent results may be that in Study C, all the use-cases may have been rather complex for the subjects who were just learning the new J2EE technology. On the other hand, in the real projects conducted in Study A and Study B, there were also very simple tasks, at least from the senior developers' point of view, such as implementing cosmetic changes or fixing trivial bugs.

When considering the results of all three studies as a whole, it can be concluded that at the level of individual tasks, pair programming either takes the same or greater effort than solo programming. When working with unfamiliar partners and while learning pair programming, the effort may even double, but after the learning period, the effort increase due to pair programming may decrease considerably. It seems also that the effort

increase from pair programming is lower when it is used for tasks that are perceived complex by the developers conducting the tasks.

Our results are aligned with the meta-analysis that concluded pair programming has a medium effect on increasing effort compared to solo programming (Hannay et al. 2009). In many of the experiments included in the meta-analysis, the duration of using pair programming was at most a day, which means that the effort differences were analyzed at the level of individual tasks and that the learning time may also have affected the effort differences.

5.3.2 Software quality

Considering the defects in code, the results from all our studies point in the direction that pair programming decreases defects compared to solo programming. However, in Study C the benefit of writing better code in pair programming sessions was lost because system testing was performed worse in the pair programming teams. A reason may be that the developers performing system testing in the pair programming teams may have relied too much on the expected positive effect of pair programming on quality.

In Study B, the developers were somewhat uncertain about the mechanism that decreases the number of defects when using pair programming, considering that the partner did not spot many defects during the pair programming sessions. It may be that pair programming prevents the defects before they are even written and therefore the partner no longer needs to point them out. The mechanism could be that the pair brainstorms the design and writes unit tests together and thus has already thought about the solution more thoroughly before writing the actual code.

Considering the design quality, in both case studies the developers perceived that pair programming improves the design compared to solo programming. The source code metrics from Study C also support this, even though the measured differences between pair programming and solo programming were small. The reliability of the source code metrics in Study C is threatened by the potential correlation between the used metrics and software size, as the solo programming teams delivered larger systems with more functionality.

Our results indicating improvements in quality are aligned with the meta-analysis that concluded that pair programming has a small positive effect on quality compared to solo programming (Hannay et al. 2009).

5.3.3 Developer's knowledge of work

In all our studies, the results show that pair programming improved the developers' knowledge of the developed system compared to solo programming. The developers' positive perceptions of the effect of pair programming on increasing knowledge in Study A and Study B were supported by the developers' evaluation of their knowledge of the individual system modules in Study B and Study C.

Based on Study B, more frequent rotation of partners can even out knowledge differences among developers. However, it seems to decrease

productivity in the short term as the developers work more with modules unfamiliar to them and probably spend more time learning new things.

In addition to our studies, there are no rigorous empirical studies on the effects of pair programming on developers' knowledge. However, the previous results of two surveys (Schindler, 2008; Begel and Nagappan, 2008), a student experiment (Bellini et al., 2005), and anecdotal comments from numerous experience reports (see Publication I) are aligned with our result that pair programming has a positive effect on increasing developer's knowledge on various work related topics.

5.4 Evaluation of the research

Below, we discuss the main strengths and limitations regarding the research presented in this dissertation. More detailed discussion on the limitations of each study can be found in the included publications.

5.4.1 Strengths

The systematic mapping study of empirical, industry-based pair programming studies used numerous ways to ensure as high a level of coverage of all scientific papers as possible. These methods included searching seven databases, searching manually certain proceedings missing from the databases, checking the reference lists of the included papers, and applying searches to the full text of the papers whenever possible. The study was also an exceptionally deep mapping study in the sense that it analyzed the full content of all 154 included papers.

The empirical studies included in this dissertation used both the experiment and case study approaches. They both have strengths that justify their parallel use. The experiment allowed the measurement of the effects of manipulating the programming method between pair programming and solo programming in an otherwise similar context. Arranging experiments in industrial settings is very difficult and expensive, but the case studies indeed allowed us to obtain data on pair programming also from industrial settings. In addition, the case studies allowed studying the topic more broadly than measuring a set of outcome variables defined in advance.

The experiment has several distinctive features compared to those conducted before it or even as of today. The subjects were experienced developers, even though they were students. The project was rather extensive instead of being composed of artificial, small tasks worth a few hours of work. We studied project teams working in a realistic, collocated team setting instead of studying isolated developers or pairs. The project teams were formed randomly, while still ensuring their equal average skill level.

Based on our literature review (Publication I), our case studies are among the most relevant studies conducted in industrial settings for many pair programming factors. The case studies covered the topic broadly and provided data from two very different industry contexts regarding the development organization and the used software development process.

The author of this dissertation has no vested interests in pair programming. The possible bias from the attitudes and beliefs of the researcher can

be expected to have been smaller than in software engineering studies where the researchers evaluate their own constructs.

5.4.2 Limitations

Research Question 1, regarding the pair programming factors, was studied in the systematic mapping study. The identified potentially relevant factors of pair programming are limited to those that were discussed in the empirical, scientific papers from contexts where professional developers used pair programming. There may be additional relevant factors that have been discussed in other types of papers. However, based on an unsystematic review of the remaining pair programming literature and on the factors identified in our own empirical studies, the set of identified factors seems to be rather complete.

Research Question 2, regarding the use and adoption of pair programming, was studied in the case studies. In Study B, one of the main goals of the project was to pilot new agile practices, and a new team was hired having this in mind. Therefore, it may not be possible to generalize the results from that study to more established software development contexts. In Study A, there were issues remaining by the conclusion of the study, meaning that additional tactics were still needed for the successful adoption of pair programming.

Research Question 3, regarding the effects of pair programming, was studied both in the case studies and in the experiment. In the case studies, asking the developers about the perceived effects of pair programming was not as reliable as the objective measurement of the effects would have been. However, in the industrial setting, we could not measure most of the potential effects of pair programming directly due to the additional costs involved, and even when we could, we could not separate the effects of pair programming from other affecting factors.

In the experiment, the sample size was only five teams, making the tests of statistical significance irrelevant. This was the price of studying five teams instead of 20 individuals, and of having a realistically sized project instead of small tasks, which could have enticed many more voluntary subjects.

In the experiment, the productivity of the teams correlated with the skill level of their best developer. Therefore, despite balancing the average skill level between the teams, individual differences between the subjects may have affected the differences between the pair programming and solo programming teams.

In the experiment, the pair programming teams were required to use pair programming for all the development work. In the industry, the proportion of pair programming of all the development work is typically lower, and the optimal proportion, considering the cost-benefit ratio, may be anywhere between full use of pair programming and no use at all. We did not have strict control over the students regarding their process conformance including the use of pair programming, and can only trust that they followed the given instructions.

5.5 Summary of the results

The main results and implications related to each research question are summarized in **Table 5**.

Table 5 Main results and their implications

RQ	Result	Implications
1	Eighteen factors and dozens of detailed aspects of PP were identified and organized as the PP framework.	The PP framework acts as a checklist of factors that may need attention when studying or practicing PP.
1	There is general scarcity of good empirical research data of PP from industrial settings. For example, measured or rigorously collected data, or comparative data is scarce, and for half of the factors there was no data in the highest overall relevance category used in our classification.	The results show the main gaps in the current research and allow future research focus on the areas with the most serious gaps.
2	Depending on the context, the adoption of PP may be an easy or challenging endeavor. Issues were identified related to both organizing and infrastructure of PP. The PP room was a working tactic for solving the infrastructure issues, but organizing of PP was a more challenging issue to solve.	Infrastructure and organizing of PP, especially ensuring that developers have enough time to do PP, require careful planning.
3	PP increases software quality and developers' knowledge on work-related topics compared to solo programming.	PP is a suitable practice especially when an organization aims at improving software quality and developers' knowledge level.
3	PP increases the effort spent on individual development tasks compared to solo programming, especially if tasks are simple or when people are learning to do PP or to know each other.	When adopting PP, at least at first, additional effort needs to be invested. The potential productivity gains due to, for instance, increases in software quality and developers' knowledge, do not realize immediately.

6 Conclusions

6.1 Contributions of the research

There are three main contributions in this dissertation. Firstly, we reviewed systematically the previous empirical pair programming research involving professional software developers, in order to identify the potentially relevant factors of pair programming. We also characterized the previous research through many properties of research and identified gaps in the research of many of the identified factors.

Secondly, we studied empirically the adoption and use of pair programming in industrial settings. We reported about both a fluent and a challenging case of adoption. In the challenging case, there were issues related to both organizing of and infrastructure for pair programming. The pair programming room was presented as a successful solution to the infrastructural issues.

Thirdly, we reported additional empirical evidence on the effects of pair programming both in the industry and in large projects conducted by teams of experienced students, which are contexts where previous high-quality research of pair programming is quite scarce. The effects of pair programming on software quality and developers' knowledge were positive in all three empirical studies, but the development effort required for individual tasks increased. The increase in effort occurred mainly when using pair programming for simple tasks or at the beginning of the project, when the developers were learning pair programming and getting to know each other.

The contributions are valuable to both practitioners and researchers. They help practitioners evaluate whether pair programming could be useful in their context, and to take into account the potentially relevant aspects of pair programming when adopting and using it. The contributions help the research community by providing more empirical evidence on the effects of pair programming. They also help researchers take better into account possibly relevant factors of pair programming and focus on factors that have not been studied adequately. The results of the studies were also directly utilizable in the two companies participating in this study.

6.2 Future work

The literature review reported in Publication I identified many gaps in the previous empirical research of professional developers. It showed that there are still many factors of pair programming that need to be studied better in order to understand their moderating effect on the effects of pair programming. Even though our studies provided more empirical data related to some of these gaps, no single study or even a small set of studies can provide definite answers. Further empirical studies are still needed, especially experiments that consider better the potential moderating factors of the effects. For example, studying professional developers who have at least a few days of pair programming experience and who know their pair

programming partner would provide more practically relevant and possibly also different results with regard to the effects than using subjects with no pair programming experience and who are not familiar with each other.

We also studied the adoption of pair programming in industry-based cases. We identified issues in adoption, and were able to identify partial solutions to them. However, there remained issues related to organizing for pair programming, particularly with regard to ensuring that developers had enough time to do pair programming. More studies on adopting pair programming are needed to provide guidelines for fluent adoption in different circumstances.

References

M. Ally, F. Darroch, and M. Toleman, "A framework for understanding the factors influencing pair programming success," *Proc. 6th Int'l Conf. Extreme Programming and Agile Processes in Software Eng*. (XP 2005), 2005, pp. 82-91.

E. Arisholm, H. Gallis, T. Dybå and D.I.K. Sjoberg, "Evaluating pair programming with respect to system complexity and programmer expertise," *IEEE Trans. Software Eng*., vol. 33, no. 2, 2007, pp. 65-86.

J. Auvinen, R. Back, J. Heidenberg, P. Hirkman, and L. Milovanov, "Software process improvement with agile practices in a large telecom company," *Proc. 7th Int'l Conf. Product-Focused Software Process Improvement* (PROFES), 2006, pp. 79-93.

R.J. Back, P. Hirkman, and L. Milovanov, "Evaluating the XP customer model and design by contract," *Proc. 30th Euromicro Conf.,* 2004, pp. 318-325.

K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

A. Begel and N. Nagappan, "Pair programming: What's in it for me?" *Proc. 2nd Int'l Symposium on Empirical Software Eng. and Measurement* (ESEM '08), 2008, pp. 120-128.

E. Bellini, G. Canfora, F. García, M. Piattini, C.A. Visaggio, "Pair designing as practice for enforcing and diffusing design knowledge", *J. Software Maintenance and Evolution: Research and Practice*, vol.17, no.6, 2005, pp. 401-423.

A. Belshee, "Promiscuous pairing and Beginner's mind: Embrace inexperience," *Proc. AGILE*, 2005, pp. 125-131.

S. Bryant, P. Romero, and B. Du Boulay, "The collaborative nature of pair programming," *Proc. 7th Int'l Conf. Extreme Programming and Agile Processes in Software Eng*. (XP 2006), 2006, pp. 53-64.

A. Cockburn and L. Williams, "The costs and benefits of pair programming," *Proc. 1st Int'l Conf. Extreme Programming and Flexible Processes in Software Eng*. (XP 2000), 2000.

M.A. Domino, R.W. Collins, and A.R. Hevner, "Controlled experimentation on adaptations of pair programming," *Information Technology and Management*, vol. 8, no. 4, 2007, pp. 297-312.

B. Fitzgerald, G. Hartnett, and K. Conboy, "Customising agile methods to software practices at Intel Shannon," *European J. Information Systems*, vol. 15, no. 2, 2006, pp. 200-213.

I. Fronza, A. Sillitti, and G. Succi, "An interpretation of the results of the analysis of pair programming during novices integration in a team," *Proc.*

3rd Int'l Symposium on Empirical Software Eng. and Measurement (ESEM '09), 2009, pp. 225-235.

H. Gallis, E. Arisholm, and T. Dybå, "An initial framework for research on pair programming," *Proc. Int'l Symp. Empirical Software Eng.* (ISESE '03), 2003, pp. 132-142.

J.E. Hannay, T. Dybå, E. Arisholm, and D.I.K. Sjøberg, "The effectiveness of pair programming: A meta-analysis," *Information and Software Technology*, vol. 51, no. 7, 2009, pp. 1110-1122.

H. Hulkko and P. Abrahamsson, "A multiple case study on the impact of pair programming on product quality," *Proc. 27th Int'l Conf. Software Eng.* (ICSE '05), 2005, pp. 495-504.

K. Johansen, R. Stauffer, and D. Turner, "Learning by doing: Why XP doesn't sell," *Proc. 1st XP Universe Conf.*, 2001.

N. Juristo and A.M. Moreno, *Basics of Software Engineering Experimentation*, Kluwer Academic Publishers, 2001.

B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering (version 2.3)," Keele University and University of Durham., Technical Report EBSE-2007-01, 2007.

M. Lacey, "Adventures in promiscuous pairing: Seeking beginner's mind," *Proc. AGILE*, 2006, pp. 263-269.

G. Luck, "Subclassing XP: Breaking its rules the right way," *Proc. Agile Development Conf.* (ADC '04), 2004, pp. 114-119.

K.M. Lui and K.C.C. Chan, "Pair programming productivity: Novice-novice vs. expert-expert," *Int'l J. Human-Computer Studies*, vol. 64, no.9, 2006, pp. 915-925.

V.B. Mišic, "Perceptions of extreme programming: an exploratory study," *SIGSOFT Software Eng. Notes*, vol. 31, no. 2, 2006, pp. 1-8.

J. Nawrocki and A. Wojciechowski, "Experimental Evaluation of pair programming," *Proc. European Software Control and Metrics Conf.* (ESCOM 2001), 2001, pp. 269-276.

K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," *Proc. 12th Int'l Conf. Evaluation and Assessment in Software Eng.* (EASE '08), 2008, pp. 71-80.

P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Eng.*, vol. 14, no. 2, 2009, pp. 131-164.

O. Salo and P. Abrahamsson, "Agile methods in European embedded software development organisations: A survey on the actual use and usefulness of Extreme Programming and Scrum," *IET Software*, vol. 2, no. 1, 2008, pp. 58-64.

C. Schindler, "Agile software development methods and practices in Austrian IT-industry: results of an empirical study," *Proc. Int'l Conf. Computational Intelligence for Modelling, Control and Automation* (CIMCA '08), 2008, pp. 321-326.

K. Sharifabdi and C. Grot, "Team development and pair programming – tasks and challenges of the XP coach," *Proc. 3rd Int'l Conf. Extreme Programming and Agile Processes in Software Eng*. (XP 2002), 2002.

J. Vanhanen and M. Mäntylä, "A systematic mapping study of empirical studies on the use of pair programming by professional developers," *IEEE Trans. Software Eng*., submitted for review.

J. Vanhanen and H. Korpi, "Experiences of using pair programming in an agile project," *Proc. 40th Ann. Hawaii Int'l Conf. System Sciences* (HICSS-40), 2007, pp. 274b.

J. Vanhanen and C. Lassenius, "Perceived effects of pair programming in an industrial context," *Proc. 33rd Euromicro Conf. Software Eng. and Advanced Applications*, 2007, pp. 211-218.

J. Vanhanen, C. Lassenius, and M.V. Mäntylä, "Issues and tactics when adopting pair programming: A longitudinal case study," *Proc. Int'l Conf. Software Eng. Advances* (ICSEA '07), 2007, pp. 70.

J. Vanhanen and C. Lassenius, "Effects of pair programming at the development team level: An experiment," *Proc. Int'l Symp. Empirical Software Eng*. (ISESE '05), 2005, pp. 336-345.

J. Vanhanen, J. Jartti, and T. Kähkönen, "Practical experiences of agility in the telecom industry," *Proc. 4th Int'l Conf. Extreme Programming and Agile Processes in Software Eng*. (XP 2003), 2003, pp. 279-287.

L. Williams, W. Krebs, L. Layman, A.I. Anton, and P. Abrahamsson, "Toward a framework for evaluating extreme programming," *Proc. 8th Int'l Conf. Empirical Assessment in Software Eng*. (EASE '04), 2004, pp. 11-20.

L. Williams, A. Shukla, and A.I. Antón, "An initial exploration of the relationship between pair programming and Brooks' law," *Proc. Agile Software Development Conf.* (ADC '04), 2004, pp. 11-20.

L. Williams and R. Kessler, *Pair Programming Illuminated*, Addison-Wesley, 2002.

L. Williams, *The Collaborative Software Process*, Ph.D. dissertation, University of Utah, 2000.

C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, 2000.

Publications

The appendix contains the five publications included in the dissertation.

I. J. Vanhanen and M.V. Mäntylä, "**A systematic mapping study of empirical studies on the use of pair programming by professional developers**," *IEEE Transactions on Software Engineering*, submitted for review in March 2011, (17 pages + 15 appendix pages).

II. J. Vanhanen, C. Lassenius, and M.V. Mäntylä, "**Issues and tactics when adopting pair programming: A longitudinal case study**, " in *Proceedings of the Second International Conference on Software Engineering Advances* (ICSEA 2007), Cap Esterel, France, August 2007, pp. 70 (7 pages).

III. J. Vanhanen and C. Lassenius, "**Perceived effects of pair programming in an industrial context**," in *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications* (EUROMICRO 2007), Lübeck, Germany, August 2007, pp. 211–218.

IV. J. Vanhanen and H. Korpi, "**Experiences of using pair programming in an agile project**," in *Proceedings of Hawaii International Conference on System Sciences* (HICSS-40), Waikoloa, Hawaii, USA, January 2007, pp. 274b (10 pages).

V. J. Vanhanen and C. Lassenius, "**Effects of pair programming at the development team level: An experiment**," in *Proceedings of International Symposium on Empirical Software Engineering* (ISESE 2005), Noosa, Australia, November 2005, pp. 336–345.

Pair programming, where two persons actively collaborate in the implementation of software development tasks has been proposed as a means to increasing software quality, knowledge transfer and learning, among other things. This research studied the adoption, use, and effects of pair programming through a literature study, two industrial case studies and a student experiment. The effects of pair programming on software quality and developers' knowledge were positive in all three empirical studies, but the development effort for individual tasks increased. The increase in effort occurred mainly when using pair programming for simple tasks or during the beginning of a project, when the developers were learning pair programming and getting to know one another.

ISBN 978-952-60-4412-5
ISBN 978-952-60-4413-2 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934
ISSN 1799-4942 (pdf)

Aalto University
School of Science
Department of Computer Science and Engineering
www.aalto.fi

BUSINESS + ECONOMY

ART + DESIGN + ARCHITECTURE

SCIENCE + TECHNOLOGY

CROSSOVER

DOCTORAL DISSERTATIONS

Aalto-DD 135/2011

Jari Vanhanen

Empirical assessment of the adoption, use, and effects of pair programming

Aalto University