A Multiple Case Study on the Impact of Pair Programming on Product Quality

Hanna Hulkko
Elektrobit Ltd.
Tutkijantie 8
FIN-90570 Oulu, Finland
+358 40 344 3440
hanna.hulkko@elektrobit.com

Pekka Abrahamsson
VTT Technical Research Centre of Finland
P.O. Box 1100
FIN-90571 Oulu, Finland
+358 40 541 5929
pekka.abrahamsson@vtt.fi

ABSTRACT

Pair programming is a programming technique in which two programmers use one computer to work together on the same task. There is an ongoing debate over the value of pair programming in software development. The current body of knowledge in this area is scattered and unorganized. Review shows that most of the results have been obtained from experimental studies in university settings. Few, if any, empirical studies exist, where pair programming has been systematically under scrutiny in real software development projects. Thus, its proposed benefits remain currently without solid empirical evidence. This paper reports results from four software development projects where the impact of pair programming on software product quality was studied. Our empirical findings appear to offer contrasting results regarding some of the claimed benefits of pair programming. They indicate that pair programming may not necessarily provide as extensive quality benefits as suggested in literature, and on the other hand, does not result in consistently superior productivity when compared to solo programming**.**

Categories and Subject Descriptors

D.1.0. [**Programming Techniques**]: General.

D.2.8. [**Software Engineering**]: Metrics – *Process metrics, Product metrics.*

General Terms

Measurement, Design, Human Factors.

Keywords

Agile Software Development, Extreme Programming, Empirical Software Engineering, Software Quality, Productivity.

1. INTRODUCTION

Pair programming, by definition, is a programming technique in which two programmers work together at one computer on the same task [1]. The person typing is called a driver, and the other partner is called a navigator. Both partners have their own responsibilities: the driver is in charge of producing the code while the navigator's tasks are more strategic, such as looking for errors, thinking about the overall structure of the code, finding information when necessary, and being an ever-ready brainstorming partner to the driver.

Pair programming is one of the key practices in Extreme Programming (XP) [2]. It was incorporated in XP, because it is argued to increase project members' productivity and satisfaction while improving communication and software quality [2]. Since then, pair programming has become one of the most researched topics in the realm of agile software development techniques [3].

In literature, many benefits of pair programming have been proposed, such as increased productivity, improved code quality, enhanced job satisfaction and confidence, to name a few. On the other hand, pair programming has also received criticism over increasing effort expenditure and overall personnel costs, and bringing out conflicts and personality clashes among developers. However, the scientific empirical evidence behind these claims is currently scattered and unorganized, and thus it is difficult to draw conclusions in one way or the other. In fact, Hanks [4] points out regarding the quality improvement claims that "There does not appear to be any empirical evidence that the programs [produced by pair programming] are better in terms of design, readability, maintainability, or other internal quality attributes." As a consequence, the industry has been rightfully hesitant in adopting the pair programming practice.

The purpose of this paper is twofold. First, it summarizes and organizes the findings from existing pair programming studies in order to systematically review the empirical body of evidence. Second, it provides new scientific evidence by reporting results from a multiple controlled case study [5] on pair programming performed in close-to-industry settings. The focus of this empirical study has been investigating, how pair programming is adopted and used by developers in industrial settings, and determining whether pair programming improves software product quality as claimed by its proponents. The objective of this paper is, therefore, to answer the following three research questions:

1. What is the current state of knowledge on pair programming?
2. How is pair programming used in practical settings?
3. How does pair programming affect software quality?

The remainder of the paper is organized as follows. In the next section, the existing empirical body of evidence is reviewed. Then, the context and findings of the four case studies are presented, after which the results and their implications are discussed. In the end, the paper is concluded with final remarks.

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

2. REVIEW OF THE EMPIRICAL BODY OF EVIDENCE

In this section, the existing empirical evidence on pair programming is summarized and reviewed. First, background information, such as research approaches and focuses, on the studies is provided. Then, the main findings of the existing studies are presented under three categories. Finally, the results of the review are summarized.

2.1 Overview on the reviewed studies

Included in the review, are empirical studies focusing on pair programming, such as case studies, experiments, surveys and experiment reports, published in various scientific forums. Also, studies focusing on XP and thus only partially addressing pair programming, have been included. However, all studies, which have been focused on educational aspects related to pair programming, have not been included in the review because of the industrial emphasis of our study.

The first reported empirical study on pair programming was published in 1998 by Nosek [6], and the most recent studies have been dated only a few months prior to writing this paper (June 2004). Figure 1 illustrates the annual distribution of different types of empirical studies found in literature, which were included in the review. The studies have been organized based on their research approaches to the following categories: case studies (CAS), experiments (EXP), surveys (SUR) and experience reports (REP). A growing research interest towards pair programming can clearly be seen from the figure. However, it should be noted, that while Figure 1 illustrates the trends on the types and amounts of empirical studies, it is not fully comprehensive, i.e. studies focused on using pair programming for educational purposes in university settings have not been thoroughly explored.

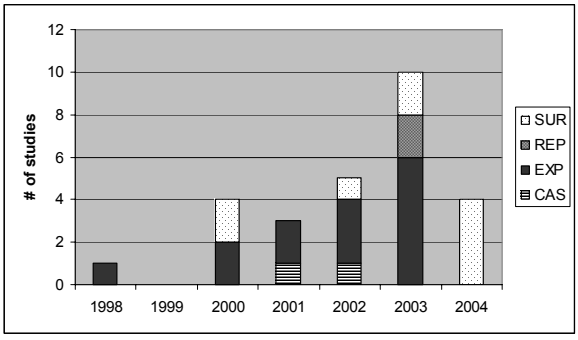

Figure 1. Studies on pair programming.

Besides their research approaches, the studies can also be grouped based on their research focuses. Figure 2 organizes the reviewed studies based on the areas of their main contributions. It shows that several research aspects have been addressed in the studies with a slight focus on the effects that pair programming has on software project and product, such as schedule and quality. In the following subsections, the review of empirical body of evidence is presented with an emphasis on the findings related to the research questions of this paper.

2.2 Analysis of existing empirical results

Project related findings. A claimed benefit gained from pair programming is shortened time to complete the given task due to e.g. increased problem solving abilities of a pair compared to an individual [7]. Also in larger scale projects, schedule advantages have been credited to pair programming because of decreased communication overhead, as an example. Williams et al. [8] report that in their experiment, pairs completed their assignments 40 – 50% faster than solo developers. Lui and Chan [9] present more moderate results of 5% time savings gained through pair programming. Also, Müller [10] discovered that pair programming halved the time spent on the quality assurance phase of a project. However, Nawrocki and Wojciechowski [11] offer contrasting results and report that in their experiment, there were no significant differences between the development times of groups who were employing XP in pairs, compared to ones employing XP or Personal Software Process [12] individually.

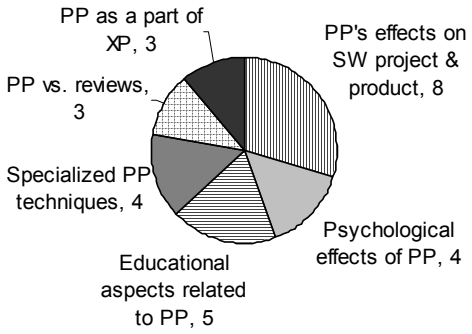

Figure 2. Distribution of empirical studies based on their research focuses.

The effort expenditure and productivity of paired software developers is one of the most studied aspects of pair programming, since one could assume that when two people are doing the same task, the spent effort is doubled. However, Williams [13] found that pairs spent approximately only 15% more effort on a task than solo developers. Other studies offer support as well: in [14], a 10% increase and in [9], a 21% increase in effort expenditure resulting from pair programming was detected. In a recent study Williams et al. [15] explored the impact of pair programming on productivity of new team members who were added to a delayed project, and concluded that pair programming reduces the assimilation and mentoring times and thus improves the productivity of the whole team. All of these results offer empirical evidence suggesting that pairs are more productive than solo developers. The increased productivity is supported also by Jensen [16], who has reported a 127% productivity gain achieved by pair programming. On the other hand, Nawrocki and Wojciechowski found pair programming less productive than XP done by solo developers [11]. Based on the existing evidence, it can be argued that pair programming requires more effort than developing software individually, but that the increase in the effort expenditure is definitely not linear with regards to the number of developers. Furthermore, the effort expenditure increases more after the initial transition to pair programming, but gradually the productivity of the pair rises above the productivity of solo developers. This phenomenon, which is sometimes referred to as pair jelling [8], could be in part used to explain the findings of Nawrocki and Wojciechowski [11] towards inferior productivity of pairs, because the tasks in their experiment were relatively small (i.e., 150 – 400 lines of code) and thus short. Hence, it could be assumed that pair jelling took

place during the experiment and increased the effort expenditure of the pairs.

Product related findings. The main argument for compensating the increased overall project costs due to higher effort expenditure of pair programming is improved quality of the resulting software [1]. Proposed reasons for the quality improvements include the continuous review performed by the navigator, which is claimed to outperform traditional reviews in defect removal speed [1], enhanced programmers' defect prevention skills [13], and pair pressure, which according to Beck [2], encourages better adherence to process conventions like coding standards and use of refactoring, i.e. increases process fidelity. Wood and Kleb [17] offer empirical support for Beck's suggestions. Gallis [18] speculates even further that stricter adherence to coding standards may improve the readability of the code and indicate increased information and knowledge transfer between developers.

In addition to the anecdotal evidence, the quality effects of pair programming have been explored in empirical studies using various, even divergent metrics. Initial findings indicating that pair programming produces shorter code (e.g. [17, 19]) and results in better adherence to coding standards [19] have been made. Shorter code, which conforms to standards can be perceived to improve the maintainability of the software. In addition, shorter code is claimed to indicate better underlying design [14, 19]. Also, decreased defect rates (e.g. [13, 16, 20]) and increased number of test cases passed [8] resulting from pair programming have been reported in empirical studies. In addition, subjective indicators of increased quality, such as improved readability [6, 17] and better grades obtained by the students in educational settings (e.g. [21, 22]) have been reported. Also, anecdotal evidence on the positive impact of pair programming on code quality exists [23, 24].

While most of these findings point to a positive direction, the generalizability and significance of these findings remains questionable. One reason for this is the fact that often the metrics used for describing quality have not been either defined in detail in the studies, or lack the connection to the quality attribute they should be presenting.

Some studies have also attempted to summarize the effects of pair programming and calculate overall cost-benefit ratios for adopting it. This task involves identifying and quantifying the effects of pair programming on the multiple parameters discussed in previous sections. While the results are initial at best, Müller [10] reported an increase of 5% on the total project costs caused by applying pair programming. According to Williams and Kessler [1], pairs have a higher efficiency and overall productivity rate compared to individual developers, and pair programming increases the business value of a project.

Usefulness in different application scenarios. In addition to evaluating, if pair programming is beneficial, it should be also considered, when it is most useful [18]. For example, pair programming may be most beneficial when applied to only certain types of tasks [1]. In the existing empirical studies, some scattered observations about the suitability of pair programming for different types of tasks have been provided.

The complexity of a task is one of the factors affecting the feasibility of using pair programming. Lui and Chan [9] report that pair programming improved productivity most in demanding design tasks. Similar findings have been made also in [8, 25] where pairing was not found useful in simple, rote tasks. Regarding different software development activities, there are studies suggesting that pair programming is not very useful in testing tasks [8], but more beneficial in performing design and code reviews and tasks related to architecture and coding [26].

2.3 Summary

The main arguments put forth in the existing empirical studies are summarized in Table 1. The right-hand column denotes if the findings are addressed through a research question (RQ) in the empirical study presented in the following section.

Table 1. Summary of main findings of existing studies

Existing empirical evidence suggest that …	RQ
Pair programming (PP) shortens development time	no
PP requires more effort (than solo programming)	no
Pairs have higher productivity than solo developers	yes
Pairs produce code with higher quality (e.g. better readability)	yes
Pairs produce code with lower defect rates	yes
Developers enjoy PP more than solo programming	yes
PP is more useful in complex than simple, routine tasks	yes
PP is useful for training a new person	yes
Cost-benefit ratio of PP (quality vs. effort) is unknown	yes

3. EMPIRICAL RESULTS FROM A MULTIPLE CASE STUDY

The empirical evaluation of pair programming through four case studies had two main goals: first, to provide qualitative and quantitative information on how pair programming is used in actual software projects, and second, to explore the impact of pair programming on software quality, especially on the quality characteristics, which literature has suggested being most affected by it, i.e. maintainability [8, 17] and reliability [10]. This section begins with the layout of the research design for the study. Then, the empirical results are presented.

3.1 Research context

Research settings. The research method used in the four case projects is the controlled case study approach [5]. The approach combines aspects of experiments, case studies and action research, and is especially designed for studying agile methodologies. In brief, it involves conducting a project which has a business priority of delivering a functioning end product to a customer, in close-to-industry settings, where measurement data is constantly collected for rapid feedback, process improvement and research purposes. The software development work is performed in controlled settings and involves both student and professional developers.

Data collection and validation. Since all of the case projects had equally important business and research goals, the data collection was designed to be as effective and extensive as possible, but still consume personnel resources minimally. The empirical evidence has been collected from multiple data sources as suggested by Yin [27] in order to obtain multiple measures of the same phenomenon to improve the validity, reliability and credibility of

the research. Table 2 presents a summary of different sources of data used in the case studies for investigating the pair programming practice.

Table 2. Data sources in case projects

Source	Data type	Case #			
			2	3	4
Excel sheets[a] / TaskMaster tool[b]	Effort: task, effort type, hours	X [a]	X [a]	X [b]	X [b]
Developers' notes (diaries)	Effort, personal remarks	X	X	X	X
Pair programming sheets	PP: time, task, pair name, role changes	-	X	X	X [1]
Defect lists	Defects: originating task, when found, severity, etc.	X [2]	X [2]	X	X
Source code (baselines after each iteration and final code)	Code-related data, e.g. solo/PP parts of the code, LOC counts	X [3]	X	X	X
Final interviews	Qualitative data, experiences	X	X	X	X
Observation	Use of PP	-	-	-	X

[1] Collection of pair programming sheets ended after the first 3 weeks

[2] Only date when defect found and fixed is available

[3] Solo and pair programmed parts of the code were not tagged

Case projects. Table 3 provides a summary of the four case projects including their duration in calendar time, team size, total development effort, product type and the developed concept, product size in terms of logical lines of code [12] and the used programming language.

Table 3. Summary of the case projects

	Case 1	Case 2	Case 3	Case 4
Duration	8 weeks	8	8	5
Team size	4 persons	5.5	4	4-6
Total dev. effort	7.5 person months	10	5.5	5.2
Iterations	6	6	6	6[1] (9)
Product type	Intranet application	Mobile application	Mobile application	Mobile application
Product concept	Research data mgmt system	Stock market browser	Production control system	Production control system
Product size (logical LOC)	7700	7100	3800	3700
Language	Java and JSP	Mobile Java	Mobile Java	Symbian C++

[1]Case 4 was relaunched after 4 weeks into the project. The first three iterations are not included in this study.

The development teams of the case projects worked in a shared co-located office space. The team members were different in each case apart from the project manager who managed cases one, two and four. Also, the developers did not have any prior experience from pair programming (except the project manager, who had of course pair programmed when he started in case projects two and four). To familiarize the developers with pair programming, a tutorial on pair programming practices and means of data collection was held in the beginning of each project. Case one involved 5-6th year Master's students. Case two involved research scientists as well, and finally cases three and four were a mix of both practitioners and students as defined by the controlled case study approach [5]. All team members were committed to a 6-hour work days, which were adopted, because the goal was to achieve a sustainable working pace emphasized in agile methods. In addition, it was seen that 6 hours is a maximum amount of time per day that a programmer can effectively focus on development tasks.

The product in cases two, three and four was developed for commercial markets, but in case one, for internal use. The software development method used in all projects was the Mobile-D approach [28], which is based on known agile methods, namely Extreme Programming and Scrum. Note that for the purposes of this study, the development method used is a secondary issue, and only briefly described here to provide background information on the study context. In Mobile-D, the projects are carried out in short (usually 1 to 2 week) iterations. Pair programming is one of Mobile-D's nine principal elements, and thus coding, testing, and refactoring were encouraged to be carried out in pairs in the case projects.

3.2 Results

As discussed earlier, the data sources, collection procedures and tools evolved during the case projects. This is why all metrics have not been calculated for each project. Table 4 presents a summary of which metrics have been calculated for each case project. Each metric is defined in the subsequent section together with the empirical results. The data sources for calculating the metrics were presented previously in Table 2.

Table 4. Metrics derived from each case project

Metrics used to evaluate pair programming:	Case project			
	1	2	3	4
Usage metrics				
Pair programming effort percent	X	X	X	X
Productivity between iterations: pair and solo	-	X	X	X
Rationale for pair programming	X	X	X	X
Quality metrics				
Density of coding standards deviations: pair and solo	-	X	X	-
Comment ratio: pair and solo	-	X	X	X
Relative defect density: pair and solo	-	-	X	X

3.2.1 Practical use of pair programming

Pair programming effort percent The ratio between effort spent on pair programming activities (i.e. pair coding, pair refactoring, and pair testing) and respective solo activities is calculated for each iteration of the case projects. This metric describes how the

use of pair programming evolves as the project progresses. Pair programming effort percent is

$$PP\% = \frac{E_P}{E_T}, \quad (1)$$

where

E_P is effort spent on pair programming activities during the iteration, and

E_T is the total programming effort spent on the iteration.

Regarding the actual use of pair programming (Figure 3), all case projects had quite similar ratios in their first three iterations, although case three's ratio is a little higher and case four's a bit lower. The differences between the case projects did not become apparent until the fourth iteration, where the percentages scattered more.

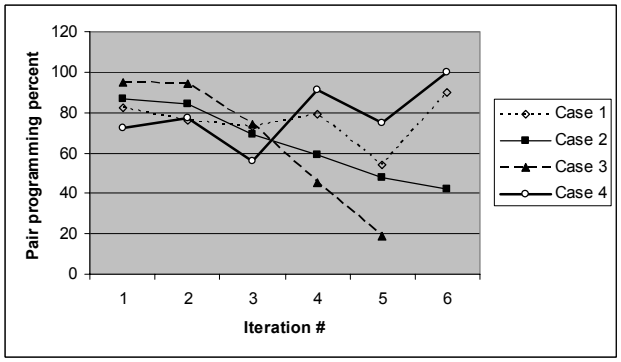

Figure 3. Pair programming percent between iterations in case projects.

As it can be seen from the figure, the pair programming percents of cases two and three decreased steadily after the first two iterations. This decrease was especially evident in case three, in whose 5th iteration over 80 percent of programming was done individually. On the other hand, in cases one and four, the pair programming percentages varied more between the iterations. Interestingly, in both case one and case four, the pair programming percentage increased in iterations four and six, and strongly decreased between them in iteration five. The overall trend in case one was quite steady, unlike in case four, where the trend was ascending despite the fluctuations.

Productivity. The second metric describing the actual use of pair programming in the case projects is productivity, which provides information on how pair programmer's productivity evolves as the project progresses and also allows to compare the productivity of the two different programming styles. In this study, productivity is calculated for both pair and solo programming styles for each iteration as a ratio of produced logical code lines and spent effort. Thus, productivity of programming style N (N is pair or solo) is defined as

$$P_N = \frac{C_N^L}{E_N}, \quad (2)$$

where

C_N^L is the number of logical code lines produced with programming style N in the iteration, and

E_N is effort spent with programming style N in the iteration.

It is acknowledged that measuring productivity is not a straight-forward task, and using lines of code (LOC) counts has its challenges. However, it is the most commonly used means for describing productivity, and thus used also here. The number of code lines for each programming style were obtained by calculating the amount of code lines in the iteration's end baseline made with each programming style (using code tags), and then subtracting the number if previous iteration's code lines produced by the same programming style from it. The productivity metrics are calculated only for the last three case projects, since no code tags were used in the first case project to separate the code produced by different coding styles.

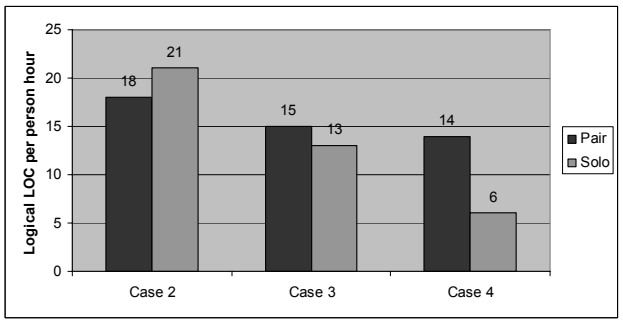

Figure 4. Total productivity of pair and solo programming in the case projects.

Figure 4 shows the total productivity of pair and solo programming in the three case projects. There seems to be no regularity between the productivity of different programming styles: in case two, solo programming has a bit higher productivity than pair programming, in case three the situation is reversed, and in case four, pair programming has substantially higher productivity than solo programming.

Rationale for pair programming. The results concerning with the rationale for pair programming obtained through team interviews are presented in the following. The focus of this qualitative data (i.e., taped, transcribed) is on determining the types of tasks and situations, which developers find especially suitable or unsuitable for pair programming. In addition to the data obtained from the final interviews, entries from TaskMaster effort tracking tool describing reasons for solo programming a specific task were studied.

The interviews aimed at collecting team members' views about the usefulness of pair programming in different application situations and development phases. One developer found pair programming to be suitable for many coding tasks, but not necessarily to e.g. installation tasks. The team members of cases one, three and four found pair programming to be especially useful for novice team members and in the beginning of a project.

> *"In the beginning of a project, pair programming is useful for virtually any task, because it helps everyone to get a clear understanding on the system." [Case three]*

The effect of the complexity of the task on the usefulness of pair programming was also brought up by the developers in the final interviews. The developers felt that pair programming was more useful for demanding and complex tasks than for rote tasks.

"If no one knows how a task should be done, it's useful to do it in pairs to think of different ideas." [Case three]

On the other hand, the some developers felt that tasks, which require a lot of logical thinking, were best when done solo:

"It's difficult for two people to think together, so thinking about a logical task should be done alone." [Case four]

This can, at least partially, result from the noise in the collocated project room, like a developer working in the case four expressed:

"If a task is difficult and complex, and I have to focus on it and think a lot, the noise in the war room is disturbing, so I put on headphones and do the task solo. I could do the task with a pair, if there was a possibility to be undisturbed." [Case four]

Furthermore, the team in case four felt that pair programming was beneficial when writing code, which had many dependencies with other parts of the software. On the other hand, according to a team member in case two, pair helped in simple tasks to find mistakes, to which the coder himself had become "blind" to. This was also supported by findings from case four interviews. Other situations where having a pair was perceived useful was related to naming conventions:

"Pair helped in naming issues, which I find to be the most difficult when coding." [Case four]

3.2.2 Quality

Density of coding standard deviations. The first metric used to describe the quality effects of pair programming is related to adherence to coding standards. In accordance to agile philosophy, the project team was responsible for defining the coding standards in the beginning of each project, and the code has been compared against these same standards when deriving this metric. The density of coding standard deviations is measured through the number of found deviations from the coding standards with respect to the amount of code made with each programming style (all physical lines). Thus, density of coding standard deviations for programming style *N* (*N* is either pair or solo) per hundred lines of code is

$$S_N = \frac{F_N}{C_N^A} \times 100, \quad (3)$$

where

F_N is the number of failures to adhere to coding standards (i.e. deviations) made with programming style *N*, and

C_N^A is the number of all physical code lines produced with programming style *N*.

A smaller density indicates better adherence to coding standards, which further indicates better readability and maintainability of the code [29, 30]. Density of coding standard deviations metric has been derived only from case project two and three. The reason for this is that in case one, there were no explicitly defined coding standards to compare the code against, and in case four, the code was written in Symbian C++ and the available tool (i.e. CheckStyle [31]) could only be used for analyzing Java programs.

In case two, a total number of 597 deviations from coding standards were found from the final source code. 431 of these were made using pair programming, and 166 with solo programming. In case three, there were a total number of 354 deviations from coding standards, of which 302 were made using pair programming and 52 solo programming. Results show that the most common type of deviations was related to method comments. Another common source of deviations was variable naming. The distribution of deviations between different types is quite similar with both pair and solo programming in both projects. However, in solo programming, fewer deviations were concerned with variable naming than in pair programming.

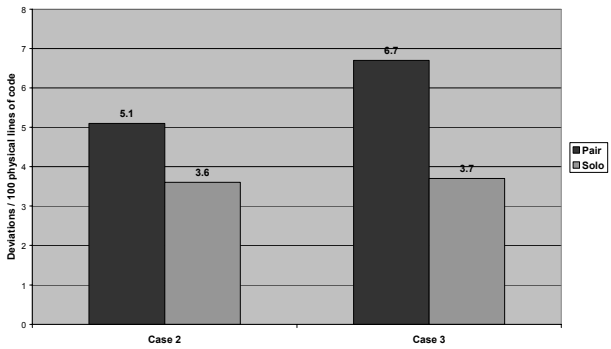

Figure 5. Density of coding standards deviations in case projects 2 and 3.

Figure 5 shows the density of deviations from coding standards in cases two and per 100 physical lines of code. It can be seen, that in both projects, the deviation density was much higher for pair programming than for solo programming.

Comment ratio. Another quality metric used in this study is comment ratio, which is calculated as the ratio of comment lines and total (i.e. physical) lines of code [32]. Thus, comment ratio for programming style *N* is

$$R_N = \frac{C_N^A - C_N^L}{C_N^A} = 1 - \frac{C_N^L}{C_N^A}, \quad (4)$$

where

C_N^L is the number of logical code lines produced with programming style *N*, and

C_N^A is number of all physical code lines produced with programming style *N*.

The higher the ratio is, the more readable and maintainable the code can be perceived to be [32]. Figure 6 illustrates the comment ratios for pair and solo programming in three of the case projects. In every case, the comment ratio for pair programming is higher than for solo programming. In case two, the comment ratios are almost equal, but in both cases three and four, the comment ratio for pair programming is approximately 60% higher than for solo programming.

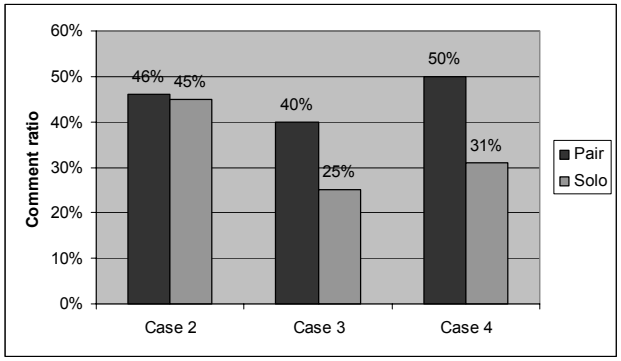

Figure 6. Comment ratios in case projects 2 to 4.

Relative defect density. Since the mode of work (i.e. solo or pair) was not predetermined in the case projects, most of the defined programming tasks have not been programmed with a single style, but rather, a mixture of pair and solo programming has been used. Thus, it has not been possible to trace the origins of the found defects to a single programming style, but only to the originating task, whose pair and solo programming effort ratios are known based on the hour entries in the TaskMaster tool. This is why it is not feasible to investigate traditional defect metrics such as defect density, i.e. the amount of total defect divided by the total number of logical code lines [12, p. 83], as such. Instead, an applied metric which considers also the portion of pair or solo programming done in the task where the defect has originated from has to be used.

Relative defect density is a metric which can be applied when the programming style of the defect is not known exactly, but only estimated using the relative amount of effort spent with different programming styles in the task where the found defect has been made. It is calculated as suggested by [12, p. 83], but instead of using the absolute number of defects made with each programming style, every defect is multiplied with the effort percent of the programming style in question of the originating task. For example, if a defect is made in a task, of whose effort 70% has been spent on pair programming, the coefficient for the defect is 0.70 when calculating relative defect density for pair programming, and 0.3 (1−0.7) when calculating relative defect density for solo programming. Also, instead of normalizing (i.e. dividing) the number of found defects with the total finished logical lines of code (LOC), total finished logical LOC made with the programming style in question is used. The obtained number is then multiplied with 1000 in order to obtain the relative number of defects per thousand logical lines of code (KLOC). Thus, relative defect density for programming style *N* is

$$D_N = \frac{\sum_i NN\%_i}{C_N^L} \times 1000, \qquad (5)$$

where

i is the index variable denoting each found defect,

$NN\%_i$ is the relative amount of effort spent with programming style *N* in the task where the defect *i* has been made, and

C_N^L is the number of logical code lines produced with programming style *N*.

Relative defect density is calculated from the final source code for both pair and solo programming. The smaller the relative defect density for a programming style is, the more mature and reliable code it can be perceived to produce. Relative defect density metric is derived from the two last case projects, because the defect lists from the first two case projects do not contain detailed enough information to base the metric on (i.e. the originating task for the found defects have not been defined).

Figure 7 shows the relative defect densities of pair and solo programming in cases three and four. In case three, approximately three times more defects were found than in case four, and thus the overall defect density of case three's source code is significantly higher than case four, due to the fact that the projects' source codes are almost of the same size. In case three, the relative defect densities of pair and solo programming are almost equal, but in case four, the relative defect density of solo programming is over six times higher than of pair programming. Based on this, it seems that the main difference of the overall defect densities of the two case projects results from the very low relative defect density of pair programming in case four.

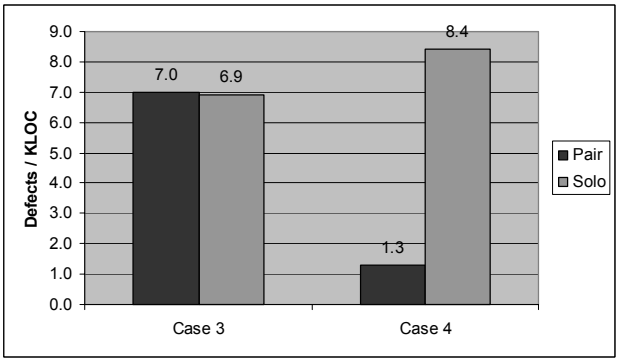

Figure 7. Relative defect densities in case projects 3 and 4.

4. DISCUSSION

Pair programming effort percent. In all case projects, the pair programming effort percent was at a high level during the first iteration, although case one was the only project, where the use of pair programming was mandated in the beginning. The high initial adoption of pair programming is in line with the project members' comments, according to which they found pair programming to be especially useful in the beginning of a project to gain understanding on the whole system and to increase confidence. This may also explain why case four had the lowest pair programming percentage in the first iteration, because its first iteration was actually the fourth iteration, since the project was started over after the first three iterations. The fact that pair programming was found more useful in the beginning of the projects is also supported by the steady decreasing of the pair programming percents towards the final iteration in cases one, two and three, although the dramatic decrease of pair programming percent in iteration five of case three can be affected by the uneven number of team members present (one team member was absent for the whole iteration). The increased pair programming

percent of the final (correction) iterations can be explained by the fact that the team members perceived pair programming most useful in non-rote tasks requiring problem solving, which is also the case in correction iteration, where the defects found in the system test phase are corrected. An exception to the other case projects in terms of the development of the pair programming percent is case four, where the pair programming percent trend was continuously rising throughout the project except for the temporary decreases in iterations three and five. This can be explained by investigating the number of development tool licenses available: in iterations one and two there were four licenses, in iteration three there were three licenses, and in the rest of the iterations (four, five and six), there were only two licenses available. Thus, in iteration three there was an uneven number of development tools available, and as a result, one team member worked alone with one tool, and therefore the pair programming percent temporarily decreased in that iteration. In the following iterations, the number of licenses was reduced to two, and thus the team had to work in pairs most of the time. As a summary, based on the quantitative and qualitative data of the case projects, it can be said that the relative amount of effort spent on pair programming is at its highest in the beginning of the project and in the defect correction (performed after system test) phase of the project.

Productivity. The results of the empirical analysis revealed that the productivity of pair programming compared to solo programming varied a lot between the case projects. Thus, based on the empirical data, no indications towards the superior productivity of one of the programming styles could be detected. This is in contrast with the findings of the existing empirical studies. In addition, although the productivity rates varied in the case projects, the differences between them seem to result mostly from the productivity of solo programming rather than the productivity of pair programming, which remained at constant level in all projects. Interpretation of productivity figures is not a straightforward task, however. Individual productivity rates of the case projects can result from many different reasons which are not related to pair programming, e.g. the high overall productivity in case two results at least partially from reused code lines (i.e., 550 physical and 300 logical lines), which have been included in the code line counts and thus affect the productivity calculations.

However, the productivity metrics do enable to analyze how the productivity of pair and solo programming evolved in the successive iterations of the case projects. According to the existing empirical evidence [8], the productivity of pair programmers is at its lowest in the beginning of a project due to pair jelling, and increases as the project progresses. However, based on the case studies conducted within this research endeavor, no regularity in the development of the productivity rates could be detected between the projects.

Rationale for pair programming. As a summary, the developers found air programming most suitable for following application scenarios: a) learning in the beginning of a project, b) solving problems and thinking of ways to do complex tasks and c) finding little mistakes from simple code. Similar findings have also been reported in literature [1, 8, 26]. The a) scenario conforms to claims that pair programming increases confidence and is an effective means for learning and training. The two latter application scenarios highlight the two levels of product related benefits gained from pair programming: ones related to long-range "strategic" issues, e.g. improved designs, and the ones related to the short-range, immediate issues, e.g. code with less minor defects such as syntax errors and typos. However, the developers did not agree about the usefulness of pair programming in scenarios b) and c); some considered thinking in pairs to be difficult, and others preferred to do simple tasks on their own. There are also existing studies indicating that pair programming is inefficient for performing simple, routine-like tasks (e.g. [25]). One common problem with adopting pair programming identified in the literature is scheduling difficulties, i.e. finding common time for the developers to work in pairs [24, 33, 34]. Some evidence, namely from case project four, pointed to this direction as well.

Density of coding standard deviations. The distribution of coding standard deviations of different types was very similar for pair and solo programming in both case projects where the metric was derived. The primary focus of the evaluation of the effect that pair programming has on adherence to coding standards was to determine, if pair programmers actually produce code which adheres better to coding standards, as suggested in the literature [1, 19]. Yet, the comparison of the calculated densities of coding standard deviations for pair and solo programming demonstrated, that in both case projects from which the metric was calculated, pair programming resulted in distinctly higher deviation density than solo programming (approx. 40 % higher in case two, and 80 % higher in case three). Thus, the claims towards higher coding standard adherence presented in literature are not supported based on the case studies. Clearly, more cases would be needed to obtain more comprehensive data to validate the findings in this regard. Nonetheless, the metrics of the two cases were convergent, and thus initial conclusions can be drawn based on them.

Comment ratio. The comment ratios calculated from three case projects are consistently higher for pair programming than for solo programming. This is contrary to the findings of the study by Ciolkowski et al. [14] in which pair programming resulted in slightly lower comment ratio than solo programming. On the other hand, the findings of our study support the arguments made in the literature towards pair programmed code being more readable [6, 17], and of better overall quality [1, 23, 24]. However, it should be noted, that the comment ratio measures only the quantity of the comments, and not their quality.

Relative defect density. The examination of the relative defect densities of pair and solo programmed code of the two case projects revealed no pattern towards lower defect density achieved with one of the programming styles.

In literature, pair programming has been reported to reduce the amount of defects and thus lower the defect density [10, 11, 13, 20]. This is supported in one of the cases. Yet, the findings of case project three do not indicate that the code produced by pair programmers would have relatively less defects than solo programmed code. Thus, our findings are conflicting in this regard.

Table 5 summarizes the empirical findings of this study. The findings are contrasted with the existing empirical body of evidence reviewed in section 3.

Table 5. Summary of the empirical results

Metric	Existing empirical body of evidence	Findings of the present study
PP effort percent	No evidence	New: Effort spent on PP is highest in the beginning of a project and in the final iteration
Productivity	Pairs are more productive than solos; PP productivity gradually increases	Not supported: neither programming style has consistently higher productivity
Rationale for PP	PP is most useful for complex tasks and training/ learning	Supported: PP is most useful for learning, and complex tasks
Density of coding standard deviations	PP produces code, which has higher adherence to coding standards	Contradicted: PP results in lower adherence to coding standards
Comment ratio	Pairs produce more readable code; Pair code has lower comment ratio (one study)	Supported: Code produced by pair programming has higher comment ratio than solo code
Relative defect density	Pairs produce code with fewer defects	Not supported: Conflicting results

5. CONCLUSIONS

There are two main contributions in this research: the performed summary and review of the existing empirical knowledge on pair programming, and the presented new empirical results. The empirical findings related to the practical use of pair programming provide concrete information, which can be utilized in industry in, for example, effort estimations and focusing pair programming efforts to certain kinds of activities, tasks, or project phases. Furthermore, the presented findings related to the quality effects of pair programming provide actual, quantitative information on the effects of pair programming to explicitly defined quality metrics instead of anecdotal evidence or ambiguous metrics. Equally importantly, the findings of the research can be utilized by academia in cost-benefit analysis of pair programming as empirically obtained and validated parameters for different existing calculation models.

The study at hand suffers from not having calculated all metrics from all of the four case studies, but has taken this into account in the discussion section when interpreting the results. To our surprise, some of the results obtained in this study offer contrasting results to the existing empirical body of evidence: our empirical data indicates, that pair programming does not provide as extensive quality benefits as suggested in the literature, and on the other hand, does not result in consistently superior productivity when compared to solo programming. Yet, these results are far from being conclusive in scientific sense, and therefore, further studies on the subject are needed.

In future research efforts, analysis of the metrics proposed in this study, could be extended to a more detailed level. For example, a means of tracing defects back to either pair or solo programming would be valuable, because without this, only relative defect density can be studied instead of absolute defect density. Also, analysis could be extended to consider not only the number of found defects, but also their severity. Additionally, the analysis of the comment ratio and adherence to coding standards could be partially merged to consider not only the quantity, but also the quality of the comments in the source code.

6. ACKNOWLEDGEMENTS

This work has been performed within AGILE-ITEA project (http://www.agile-itea.org/). We would like to thank Professors Juha Röning and Veikko Seppänen from University of Oulu for their valuable comments. Also, the developers and customers of the case projects should be acknowledged for their inputs. We would also like to thank the three anonymous reviewers of this paper for their valuable improvement suggestions.

7. REFERENCES

[1] L. Williams and R. Kessler, *Pair Programming Illuminated*: Addison-Wesley, 2003.

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

[3] P. Abrahamsson, J. Warsta, M. T. Siponen, and J. Ronkainen, "New Directions on Agile Methods: A Comparative Analysis," International Conference on Software Engineering, 2003.

[4] B. F. Hanks, "Tool Support for Distributed Pair Programming," Workshop on Distributed Pair Programming. Extreme Programming and Agile Methods - XP/Agile Universe, 2002.

[5] O. Salo and P. Abrahamsson, "Empirical Evaluation of Agile Software Development: A Controlled Case Study Approach," 5th International Conference on Product Focused Software Process Improvement, Japan, 2004.

[6] J. T. Nosek, "The Case for Collaborative Programming," *Communications of the ACM*, vol. 41, pp. 105 - 108, 1998.

[7] L. A. Williams, *The Collaborative Software Process. PhD Dissertation,* University of Utah, 2000.

[8] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair Programming," *IEEE Software*, vol. 17, pp. 19-25, 2000.

[9] K. M. Lui and K. C. C. Chan, "When Does a Pair Outperform Two Individuals?" XP2003, Italy, 2003.

[10] M. M. Müller, "Are Reviews an Alternative to Pair Programming?" 7th International Conference on Empirical Assessment in Software Engineering, UK, 2003.

[11] J. Nawrocki and A. Wojciechowski, "Experimental Evaluation of Pair Programming," 12th European Software Control and Metrics Conference, UK, 2001.

[12] W. S. Humphrey, *A Discipline for Software Engineering*: Addison-Wesley, 1995.

[13] L. Williams, "Integrating Pair Programming into a Software Development Process," 14th Conference on Software Engineering Education and Training, USA, 2001.

[14] M. Ciolkowski and M. Schlemmer, “Experiences with a Case Study on Pair Programming,” First International Workshop on Empirical Studies in Software Engineering, Finland, 2002.

[15] L. Williams, A. Shukla, and A. I. Antón, “An Initial Exploration of the Relationship Between Pair Programming and Brooks' Law,” Agile Development Conference, 2004.

[16] R. W. Jensen, “A Pair Programming Experience,” *CrossTalk, The Journal of Defense Software Engineering*, vol. 16, pp. 22 - 24, 2003.

[17] W. A. Wood and W. L. Kleb, “Exploring XP for Scientific Research,” *IEEE Software*, vol. 20, pp. 30 - 36, 2003.

[18] H. Gallis, E. Arisholm, and T. Dybå, “An Initial Framework for Research on Pair Programming,” ISESE, Italy, 2003.

[19] A. Cockburn and L. Williams, “The Costs and Benefits of Pair Programming,” 1st International Conference on Extreme Programming and Flexible Processes in Software Engineering, Italy, 2000.

[20] J. E. Tomayko, “A Comparison of Pair Programming to Inspections for Software Defect Reduction,” *Journal of Computer Science Education*, vol. 12, pp. 213 - 222, 2002.

[21] C. McDowell, L. Werner, H. E. Bullock, and F. J., “The Impact of Pair Programming on Student Performance, Perception and Persistence,” 25th International Conference on Software Engineering, USA, 2003.

[22] L. Williams, C. McDowell, N. Nagappan, J. Fernald, and L. Werner, “Building Pair Programming Knowledge through a Family of Experiments,” International Symposium on Empirical Software Engineering, Italy, 2003.

[23] K. Nilsson, “A Summary from a Pair Programming Survey - Increasing Quality with Pair Programming,” 2003.

[24] T. H. DeClue, “Pair Programming and Pair Trading: Effect on Learning and Motivation in a CS2 Course,” *Journal of Computing in Small Colleges*, vol. 18, pp. 49 - 56, 2003.

[25] M. M. Müller and W. F. Tichy, “Case Study: Extreme Programming in a University Environment,” International Conference on Software Engineering, 2001.

[26] L. Williams and R. Kessler, “The Effects of “Pair-Pressure” and “Pair-Learning” on Software Engineering Education,” 13th Conference on Software Engineering Education and Training, USA, 2000.

[27] R. K. Yin, *Case Study Research - Design and Methods. 3rd ed.*, SAGE Publications, 2003.

[28] P. Abrahamsson, A. Hanhineva, H. Hulkko, T. Ihme, J. Jäälinoja, M. Korkala, J. Koskela, P. Kyllönen, and O. Salo, “Mobile-D: An Agile Approach for Mobile Application Development,” OOPSLA'04, Canada, 2004.

[29] M. Elish and J. Offutt, “The Adherence of Open Source Java Programmers to Standard Coding Practices,” The 6th IASTED International Conference on Software Engineering and Applications, USA, 2002.

[30] X. Fang, “Using a Coding Standard to Improve Program Quality,” 2nd Asia-Pacific Conference on Quality Software, Hong Kong, 2001.

[31] Checkstyle tool. URL: http://checkstyle.sourceforge.net/.

[32] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, “An Integrated Measure of Software Maintainability,” Annual Reliability and Maintainability Symposium, 2002.

[33] J. Kivi, D. Haydon, J. Hayes, R. Schneider, and G. Succi, “Extreme Programming: a University Team Design Experience,” Canadian Conference on Electrical and Computer Engineering, Canada, 2000.

[34] T. VanDeGrift, “Coupling Pair Programming and Writing: Learning about Students' Perceptions and Processes,” ACM SIGCSE, 2004.