Software Quality Management

Chilla Kartheek
Department of Software Engineering
9304163778
BTH, KARLSKRONA
chilla.kartheek87@gmail.com

Motivation For selection: My topic for the assignment 3 is whether using the pair programming in designing and developing the code show impact on the quality of the product. Does the pair programming show impact in reducing the defect density, which in turn improves the quality of the code. Reduce in the defect density by using the pair programming instead of using solo programmer to work on code is the research area where it is keen to know if there is any improvement in the quality, if so, which one is followed to achieve the improvement. Generally Pair programming means one person handles mouse or keyboard to write the code. The other person scrutinizes or verifies the code for any modifications or improvements to reduce the bad smell codes (ineffective Lines of Code).

Selected article1: This paper is selected because it exemplifies the impact of pair programming in four different software development projects on software product quality. The article is interesting because the different projects given in this article the research case study can help to understand whether there is impact in software quality when the pair programming is implemented. This article help to understand whether the pair programming show impact on the attributes like design, readability, understandability and other internal quality attributes. Thus article show results where in 2 case the productivity of pair programming is higher and in 1 case the productivity of solo programming is high. In where the impact of solo programming can and cannot achieve quality and productivity for a product can be understood from this article. These article present the information about the defect density that is appeared in the code when pair programming and solo programming is used. Thus this article can help to study the impact of pair programming on the quality of the product. Thus study provides the information on the existing evidences about the impact on pair programming and the current findings form 4 different software projects with respect to impact of pair programming on software product quality*[1]*.
Selected article 2: This article is an extensive case study on pair programming in software development teams and its benefits. First article helps to find in which case to utilize the pair and in which case to not utilize the pair programming. Thus article gives the glimpse of pair programming and its benefits, which include the quality of the code. Thus article is important because it supports the statement pair programming is a suitable method to ensure the quality of the code as the observer does the first review while the driver writes. This article also supports the statement like the program implemented by pair have same functionality as the solo once but with less number of lines of code (LOC) and the quality is thus higher in this case. Usually the quality of the program is identified by the passed test cases, the pair programing show higher pass percent than solo programing. Thus selecting this article will help to understand the benefits of pair programming over solo programming and its impact over the quality of the code*[2]*.

[1] H. Hulkko and P. Abrahamsson, “A Multiple Case Study on the Impact of Pair Programming on Product Quality,” in *Proceedings of the 27th International Conference on Software Engineering*, New York, NY, USA, 2005, pp. 495–504.
[2] T. Bipp, A. Lepper, and D. Schmedding, “Pair programming in software development teams – An empirical study of its benefits,” *Inf. Softw. Technol.*, vol. 50, no. 3, pp. 231–240, Feb. 2008.

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.

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.

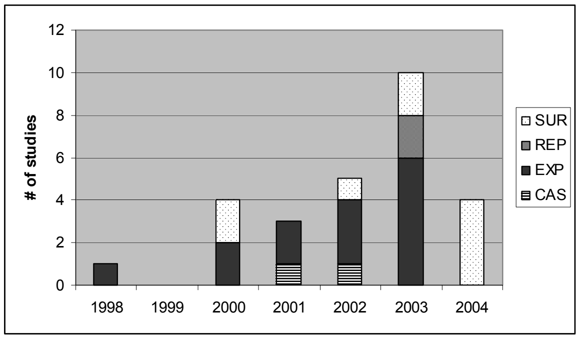

Figure 1. Studies on pair programming.

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.

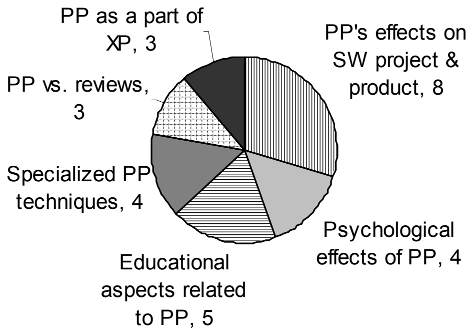

Figure 2. Distribution of empirical studies based on their research focuses.

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

$$PP\% = \frac{E_P}{E_T}, \tag{1}$$

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.

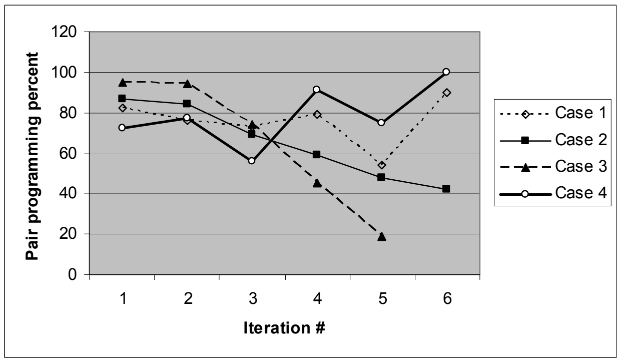

Figure 3. Pair programming percent between iterations in case projects.

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

$$P_N = \frac{C_N^L}{E_N}, \tag{2}$$

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.

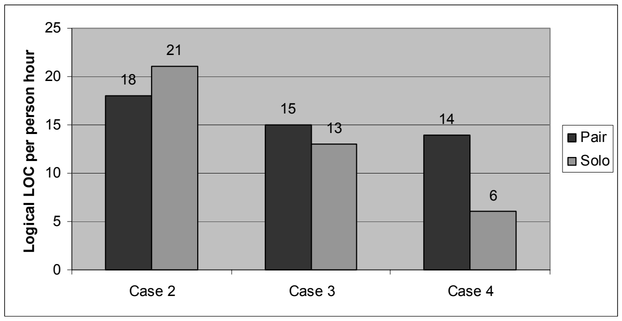

Figure 4. Total productivity of pair and solo programming in the case projects.

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

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

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.

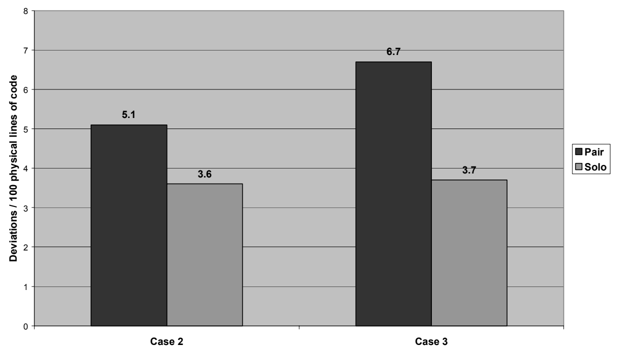

Figure 5. Density of coding standards deviations in case projects 2 and 3.

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

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

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.

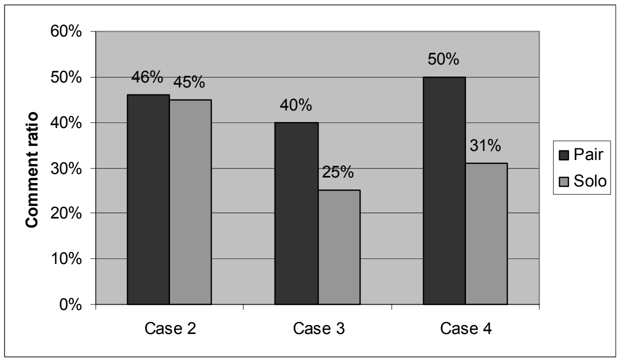

Figure 6. Comment ratios in case projects 2 to 4.

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.

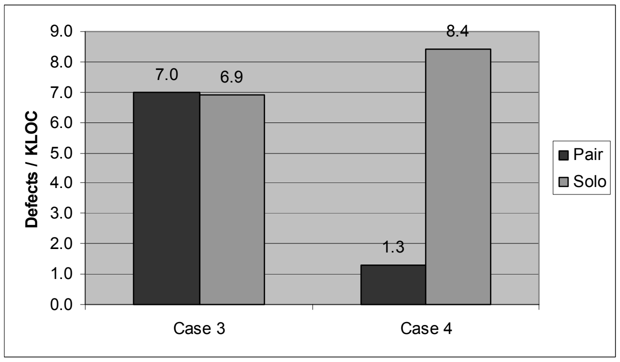

Figure 7. Relative defect densities in case projects 3 and 4.

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.

Available online at www.sciencedirect.com

ScienceDirect

Information and Software Technology 50 (2008) 231–240

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Pair programming in software development teams – An empirical study of its benefits

Tanja Bipp [a,1], Andreas Lepper [b], Doris Schmedding [b,*]

[a] Organizational Psychology, University of Dortmund, Dortmund, Germany
[b] Department of Computer Science, University of Dortmund, Dortmund, Germany

Received 27 June 2006; received in revised form 24 May 2007; accepted 25 May 2007
Available online 16 June 2007

Abstract

We present the results of an extensive and substantial case study on pair programming, which was carried out in courses for software development at the University of Dortmund, Germany. Thirteen software development teams with about 100 students took part in the experiments. The groups were divided into two sets with different working conditions. In one set, the group members worked on their projects in pairs. Even though the paired teams could only use half of the workstations the teams of individual workers could use, the paired teams produced nearly as much code as the teams of individual workers at the same time. In addition, the code produced by the paired teams was easier to read and to understand. This facilitates finding errors and maintenance.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Pair programming; Empirical software engineering; Quality of software

1. Introduction

Pair programming [23] is an important feature of extreme programming [2]. Our own experiences [3] with extreme programming with groups of students showed that students less experienced in software development were overwhelmed by planning and organizing an extreme programming project. In particular, estimating the efforts to realize a feature and distinguishing between essential and nice-to-have features were very difficult for the students. Still, we noticed that the students do benefit from pair programming. With this in mind, we decided to continue using this concept in our courses and to investigate the benefits and drawbacks of this method in more detail.

In the “Software-Praktikum” course (SoPra) at the University of Dortmund, teams of eight students of computer science carry out software development projects. They improve their programming knowledge and study software engineering by applying software engineering methods in projects close to reality.

In accordance with Nosek [20], we define a *paired team* as a team in which each task of software development is done simultaneously at one workstation by two members of the team. There are two roles within a pair [23]: one person is the “driver” who has control of the pencil/mouse/keyboard and is writing the design or code. The other person, the “observer”, continuously and actively examines the work of the driver – watching for errors, thinking of alternatives, looking up resources, and considering strategic implications of the work at hand. The observer identifies tactical and strategic deficiencies in the work. The partners in a pair switch their roles periodically. Additionally, the pairs in the team are put together in new combinations frequently to support the distribution of knowledge in the whole team.

Distribution of knowledge through the whole team can be seen as one advantage of this type of team work. If at least two developers work on every task, a paired team can compensate for the loss of an expert for a specific task

* Corresponding author. Tel.: +49 231 755 2436.
E-mail addresses: Tanja.Bipp@udo.edu (T. Bipp), Andreas.Lepper@udo.edu (A. Lepper), Doris.Schmedding@udo.edu (D. Schmedding).
1 Tel.: +49 231 755 2841.

0950-5849/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2007.05.006

much more easily. The solutions of the pairs should be better than the solutions of individuals because the combined creativity and experience of both are greater. Pair programming can be seen as a suitable method to ensure quality of code since the observer does a first code review while the driver writes. This helps to avoid and detect defects early on.

Especially in work groups of students, the potential advantages of working in pairs can be very useful. Within student groups in SoPra, the programming knowledge is very heterogeneous. One potential cause is the students' experience in programming, which in some cases is based solely on homework in programming courses at the university, whereas other students have held jobs as Java programmers or user interface designers. A software development method based on "learning from each other" is supposed to be particularly helpful, especially in situations such as these, where prior knowledge is not homogeneous.

Working in teams of two instead of utilizing the work force of two single programmers, the postulated low efficiency of pair programming is one of the most prevailing counter arguments for this type of work organization. It is therefore not surprising, that this approach is met with particularly little acceptance in teams working under great time pressure.

When two developers work together on one task all the time, it is generally expected that a team consisting of pairs needs twice as much time as a team of developers working on their own. These additional costs must be compared to the positive effects of pair programming on software quality, on the climate in the work teams, and a higher level of knowledge in all team members. Results of current research studying the costs and benefits of pair programming are presented in Section 2.

To study the advantages and disadvantages of software development by teams of pairs, we did some extensive experiments in the SoPra with a total of 95 students in summer 2004 and in spring 2005. In Section 3, these studies are described in detail. Section 4 discusses our experiences on the acceptance of the pair programming concept and the work results of the groups are compared. In the conclusion, the relevance of the results for the field of software engineering and the limitations of our experiment are discussed.

2. Studies with paired teams

A seminal study on pair programming was carried out by Nosek in 1998 [20]. Subjects of his experiment were 15 full-time system programmers from a program trading firm working at system maintenance. They were asked to write a small script. Ten programmers worked in pairs and five of them worked individually and served as a control group. When comparing the five solutions of the experimental group with the five solutions of the control group, Nosek found that the readability of the code of the pairs and the functionality of their solutions was significantly higher. Additionally, the pairs were more confident in their solutions and enjoyed their collaborative problem-solving process more. In fact, all groups outperformed the individuals. Although the average time for completion was 30.20 min taken by the pairs and 42.60 min taken by the individuals, the prediction that pairs need less time to solve a problem was not statistically supported. Prior knowledge of the system programmers and quality of the solution (functionality and readability) were highly correlated for both the experimental and the control group.

Williams et al. [22] carried out an experiment in a senior software engineering course at the University of Utah. The students were divided into two groups with nearly equal levels of prior knowledge: 28 students formed the experimental group of pair programmers, 13 students formed the control group of individual workers. Whereas Nosek's experiment was restricted to comparing coding time and quality of code, Williams and her coauthor observed the whole process of software development, analysis, design, implementation, and test. The students completed four assignments over a period of six weeks. The authors compare development time, efficiency, and quality of the results of both groups. The pairs were able to complete their assignments in 40–50% of time spent on the project by the single developers. The quality of the products was measured by counting passed test cases. The four programs of the pairs passed statistically significant ($p < 0.01$) more of the automated post development test cases. Since the programs of the pairs had the same functionality but less lines of code compared to the programs of the teams of individual developers, the authors conclude that the quality of the code of the pairs is higher. The pairs enjoyed their work more because they are more confident in their results. The authors describe their study as quantitative research, although they mostly only show the mean values of their data and not the complete set of raw data with additional descriptive statistics.

McDowell and his coauthors [16,17] received some interesting quantitative results when studying the effects of pair programming in a programming course for beginners with 600 participants at the University of California. They recognized that weaker students in particular benefit from working in pairs. Less pair programmers quitted the course and pair programmers consequently passed their exams more frequently than students of the control group who had to complete their assignments alone. Also, the long term influence of pair programming was impressive as more of the pair programmers took follow-up computer science courses. In comparison to the other studies, the sample of this study is quite large. But it concentrates on programming and does not look at the entire software development process.

Müller [18] and Padberg [19] also compared the work of individuals with the work of pairs instead of building teams of pairs as proposed in extreme programming. They let 38 students at the University of Karlsruhe write two small programs. On average, the students needed 4 h to finish the assignment. Half of the students worked in pairs, the others worked alone. The collected data showed no correlation between the experience level of a pair and the implementation time, but a significant correlation between performance and feel-good factor [19]. The tasks students had to solve were defined in such a way that automated test cases could be used to measure the functionality of programs. High quality of a program was defined by high rate of passed test cases. The average level of correctness of programs developed by pairs was 29% higher than of programs developed by single programmers [18].

All cited authors, besides Williams et al., restrict their studies to programming as one task during software development. Similar to Williams et al., we want to study the entire software development process. In every experiment considered so far, a task was completed by a pair of two developers or by a single developer who worked alone. The tasks were small and the requirements were precisely defined. In these studies, teams of individuals solving one task together are not compared with teams of pairs switching partners and the roles within the pairs. In contrast, we choose a much more realistic experimental setting. Teams of eight students were asked to solve a complex problem. Half of the teams worked in pairs, the others were teams of individual developers.

3. Method

The effects of pair programming were studied in two experimental settings referred to as Study I and Study II, respectively [15]. The main focus laid on testing the postulated effects of the work organization type on performance (paired teams versus teams of individuals). Study I can be seen as a preliminary study, most of the results presented in Section 4 were yielded in Study II.

3.1. Setting/task

The "Software-Praktikum" course is obligatory for all students in the second year of studying computer science at the University of Dortmund. Aims of the course are the improvement of programming skills, to apply software development methods and tools, and to work in a team.

During six weeks, teams of eight students carry out two projects. Each team of students solves the same tasks. During this time, no other classes are scheduled so that the students can work on their projects all day. In Study I, the students realized a card game. The other task was the management of a cocktail bar. In Study II, the tasks were a quiz game and the simulation of the elevators in a multi-story building, respectively.

We use Java as programming language and UML [5] for analysis and design. The software development teams follow a process model [13] based on Unified Process [14] in their projects.

3.2. Subjects

In total, 95 undergraduate computer science students from the University of Dortmund (2nd/3rd year students) participated in our studies, who all had just successfully finished their intermediate examination in software development. All of them received course credits in return for their participation. The students were randomly sampled to the teams of developers (experimental group as well as control group) to control for potential biases, like differences in prior programming experiences.

The present research paper consists of two different samples. Within Study I (2004), 25 students participated and were randomly assigned to one of four groups with six or seven members with different work conditions. Two teams worked within pairs of two whereas the remaining students worked as teams of individuals. Most of the work took place in a laboratory at the University of Dortmund [4], where the groups could be observed and video-taped while working on the assigned projects. Further research questions lead to a second study, which took place in 2005. Seventy participants were split by chance into 9 groups of nearly equal size. Five of nine groups in Study II were asked to work in groups of two. In support of this concept, special work conditions for these students were created. Paired teams were only assigned half the number of workstations of the number of team members. The other teams were supplied with the amount of workstations according to their team size and received no restrictions regarding their work organization.

3.3. Hypothesis

Based on our experiences within Study I, and taking into account the existent literature on the effects of pair programming, we expected several advantages of pair programming concept. First of all, we were interested how the participants in our studies accepted the concept and were able to increase their work performance. Our hypothesis regarding acceptance, quality of products, and distribution of work load are:

- Less experienced students benefit from pair programming because they are better integrated in the software development teams.
- The knowledge about the project and how to do special subtasks is spread through the whole team.
- Code written by a paired team is better, less complex, and easier to read because it is written to be understood by the whole team not only by its single author.

These questions are interesting and new because the before mentioned studies compare only the performance of a pair of two developers with an individual worker, not development teams with complex tasks.

3.4. Measures

To test our hypothesis regarding the influence of work organization on different work performance related variables, we assessed several covariates and performance related variables within the study settings:

- programming skills (programming knowledge prior to SoPra),
- time spent on the task (duration of work periods),
- work behavior,
- general motivation to do work within SoPra, and
- personality factors.

They were either based on observational, objective data or questionnaires. Subjective constructs were assessed by different items within questionnaires before the start of SoPra or while working on the projects. Additionally, participants were asked to fill out a final questionnaire after the completion of the projects to gain information about the work load distribution within work groups. Answers on sole items were only aggregated to a mean scale value, if they reached an acceptable value of reliability (Cronbach's $\alpha > .70$) [9]).

Participants judged their *prior knowledge* and programming skills on the basis of six five-point Likert scaled items (Example: "*I did a lot of programming during my studies.*"[2]). Answers were combined to a mean value (answer range from 1 = little/5 = much knowledge, Cronbach's $\alpha = 0.82$).

To gain information about the *time spent on the task* by different work teams, participants were asked to regularly mark their hours of work on a questionnaire.

To check if the groups really worked within their assigned work conditions, participants in Study II were asked about their *work behavior* (Example: *"Most of the time we worked in pairs in our group."*). Four different items were generated (ratings from 1 = totally disagree to 5 = totally agree) and combined into a common value (Cronbach's $\alpha = 0.90$). Additionally, conclusions from Study I are based on behavioral observations. In Study II, the database is supplemented by objective data (number of logged in persons). *Distribution of work* was measured by single items at the end of SoPra ("*The work load was not equally distributed in our group,*" "*I barely know some parts of the project.*"). Both items were rated on a five-point Likert scale, ranging from 1 = totally disagree to 5 = totally agree. Finally, three items for measuring ambiguity about the work schedule were included in the final questionnaire after completing both tasks, from an adapted German version [21] of the job ambiguity questionnaire by Breaugh and Colihan [7]. Three items were rated on a seven-point Likert scale, ranging from 1 = totally disagree to 7 = totally agree (Example: "*I knew precisely, in which chronological order I had to accomplish my work,*" Cronbach's $\alpha = 0.73$).

To account for the *motivation* of group members within SoPra, participants were asked eight different questions to state their motivation for engaging in activities at the beginning of the SoPra (Example: *"I am looking forward to the opportunity to apply in SoPra the knowledge that I acquired,"* answering range from 1 to 5, low values = high willingness to get involved within SoPra). Whereas it was not possible to combine the answers within Study I into a reliable scale, Study II yielded an acceptable reliability value (Cronbach's $\alpha = 0.82$). Additionally, participants responded to a German version of the NEO-FFI [6]. This test measures the "Big Five" *personality factors* (neuroticism, extraversion, openness to experience, agreeableness, and conscientiousness) with 12 items for each construct.

For mainly exploratory reasons, we included at the end of the SoPra nine items linked to the acceptance or evaluation of the pair programming concepts within the setup of Study II ("*For which task of software development is working with a partner useful?*" e.g., program design, product test).

3.5. Methods

Indicators of task performance were obtained by measuring the amount and quality of the programming assignments. We measured the range and the complexity of the developed software by means of software metrics. Based on the metric suits of Chidamber and Kemerer [8], many software development tools offer facilities to analyse an object oriented program. The metric tools determine the depth of inheritance, the coupling between the objects and their cohesion. Cohesion is the degree to which methods within a class are related to one another and work together to provide well-bounded behavior. Additionally, we asked software development experts to inspect the programs of the students and judge the quality.

To compare the results within the paired teams and teams of individuals regarding questionnaire data, we compared values on the individual level (individual workers vs. pair programmers) or on group level (paired teams vs. teams of individuals). Therefore, we conducted several t-tests and ANCOVA's [12] to account for differences in mean values between these groups.

3.6. Manipulation checks

Within the two studies, personality, motivation, time spent on the task, and prior programming knowledge were tested for differential effects between the groups, based on their postulated influence on team performance. Despite the random assignment of students to the different work conditions, three of these covariates showed significant differences in Study I (prior knowledge, personality, and time spent on the task). Therefore, the results of Study I are not

[2] Interested readers may obtain a copy of the complete (German) questionnaire from the authors.

presented in detail within the following sections. Data from this study were mainly used to generate our research hypothesis for testing in Study II. Within this study setup, one group (No. 9) refused to work within the assigned conditions pair programming, so that these seven members are excluded from the dataset for the following analyses. The sample size is therefore reduced to 63.

3.6.1. Prior knowledge

By sampling randomly individual students to different groups of working conditions, we wanted to control for several potential influences on performance. Based upon a wide range of programming experiences (besides university curriculum), prior knowledge of the students was distributed heterogeneously within the groups of both studies. This perception of the SoPra organizer was validated by questionnaire data. The distribution of the questionnaire data regarding prior knowledge of the different development teams in Study II is given in Fig. 1.

The descriptive inspection of self-assessed knowledge scores (see Boxplot in Fig. 1) leads to the assumption that prior knowledge is not equally distributed within the development teams. In some teams, prior knowledge scatters more than in others. However, a statistical test of the average knowledge score in Study II revealed no significant mean difference between the individuals in the experimental and control groups ($M_{\text{pairedTeams}} = 2.58$; $M_{\text{teamsOfIndividuals}} = 2.87$; $t(60) = -1.56$, $p > 0.05$). A distorting influence of different programming experiences on performance between the different development teams can therefore be excluded.

3.6.2. Motivation and personality

Team effectiveness is a broad construct and is therefore influenced by numerous factors. In addition to external environmental determinants (like work conditions), a lot of research has been done on the composition of teams, taking into account group members abilities, attitudes, and motivation. How team members think about their groups and their goals is likely to have an important influence on team performance. From the observational data and personal discussions with the team members in Study I, we know that they were all highly motivated to complete the assignments. In Study II, the mean values of the ratings of motivation within our questionnaire were close to each other in both experimental groups (1.65–2.25). A statistical test for the difference between these groups revealed no significant effect. The members of both teams were comparably motivated to work on the SoPra assignments.

Whereas the comparison of the Big Five personality factors showed no significant difference between the participants in Study II, members of the paired teams in Study scored significantly higher on conscientiousness compared to members of the groups of individuals. Relating personality to individual and team performance, this factor especially shows intercorrelations with task relevant output variables [1]. This finding raised again considerable doubt about the comparability of the two work setting groups in Study I.

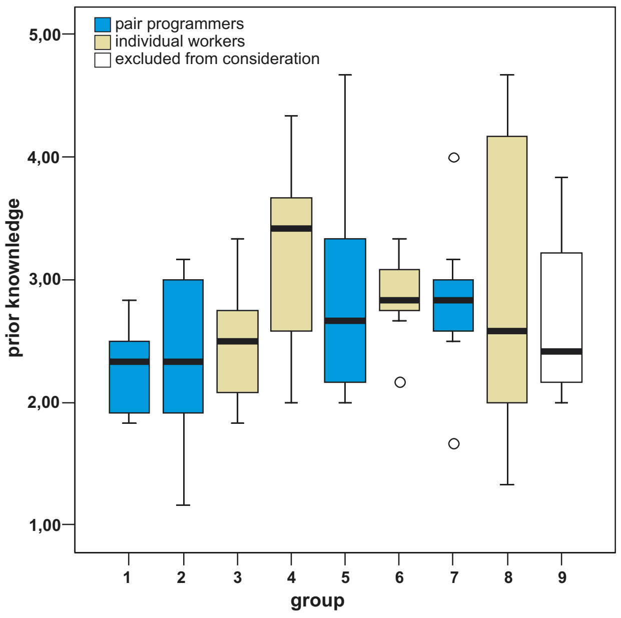

Fig. 1. Prior knowledge within each group in Study II ($N = 72$).

3.6.3. Hours of work

Working hours are usually not specified within SoPra. Instead, every team works the amount of time it needs to fulfill its assignment. Average working time per week varied between 22.7 and 28.8 h between the eight considered groups of Study II. Groups of pairs worked on average 26.0 h a week, whereas the other teams reached a slightly reduced average working time (25.7 h). Therefore, on the mean level the difference between the groups is not statistically significant. On average, all groups worked about the same amount of time per week on their projects. A systematic influence from time spent on the project on performance between the groups can therefore be excluded.

3.6.4. Check experimental setup/work conditions

Within the first study, participants were observed during their work periods on a regular basis. All groups worked in accordance to the experimental work concept, working on their workstations either individually or in twos. In addition to the programming work, all teams held meetings with all members on a regular basis. Answering questions or helping coworkers was not prohibited within the different experimental setups. Such helping behavior occurred under both sets of working conditions. The sample of Study II consisted of far more persons than Study I, so a constant observation could not be realized for reasons of practicability. Instead, the participants were asked about their working behavior in detail after finishing their projects by questionnaire.

Fig. 2 shows the mean levels of the four items used in Study II (higher values indicating an intensified working in pairs). On average, the members of the paired teams

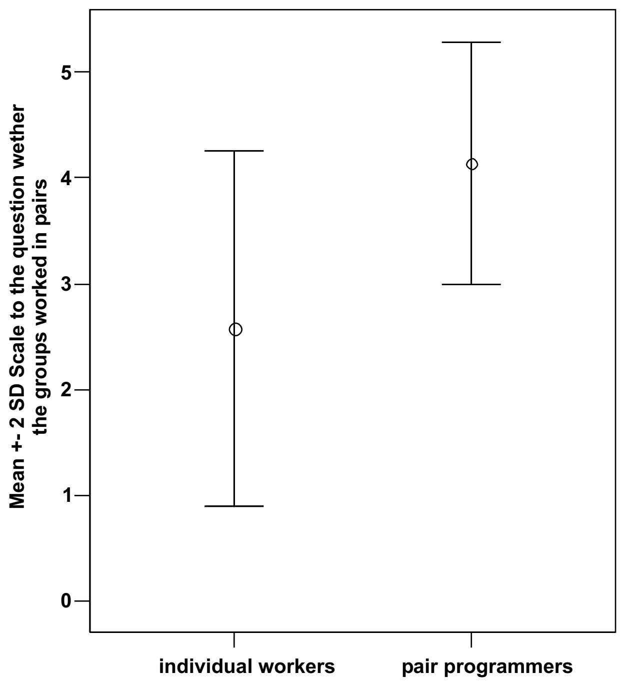

Fig. 2. Fulfilment of paired-team concept in Study II.

reached a value of 4.14, whereas the other groups reached a mean value of 2.58 or the work behavior. The teams of individual workers were not forbidden to work with a partner and did so from time to time. In total, the groups designed to work in pairs realized this working behavior more often than the other groups (significant mean difference), which supports a successful manipulation of work conditions.

Besides the collection of subjective data by questionnaires, the sum of logged in persons within the computer labs was registered by a Unix script in different time periods. Fig. 3 displays the difference between the paired teams

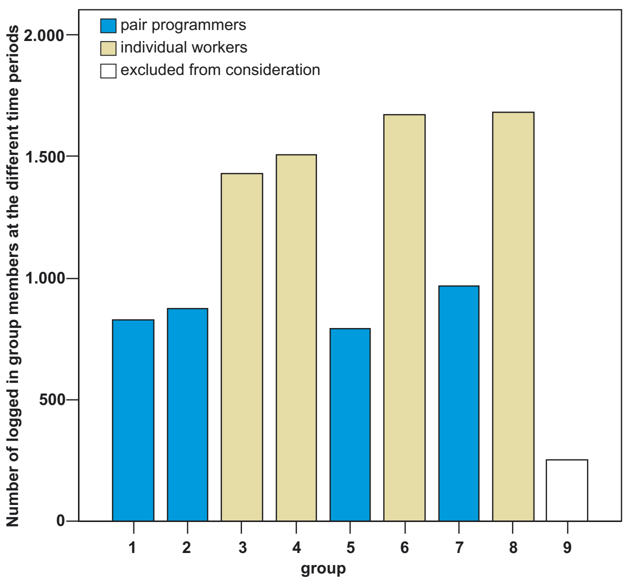

Fig. 3. Number of group members logged in.

(dark shaded) and individual teams (light shaded) clearly. One of the groups (No. 9) refused to participate in the study and worked most of the time on their own laptops without logging in on the provided workstations. Within subsequent analysis, this group is excluded from further consideration.

4. Results

Within the following sections, results regarding the acceptance of the pair programming concept, quality of developed software and further results regarding questionnaire data are presented.

4.1. Acceptance of pair programming

We have to confess that one of our five groups (chosen randomly to work as a paired team) refused to work in this way. Two members in this group (No. 9) expressed great disaffirmation against pair programming already in the first questionnaire before the experiment started. Although the remaining members of this team were absolutely open-minded about pair programming, we could not include this group in our experiment. Pair programming in a team can only be successful if every member of a team accepts this concept.

The remaining participants in the paired teams were asked to report their experiences in the experiment using questionnaires. They judged the role of their partner in the pair in the following way (see Fig. 4): 90% regarded his or her partner as very helpful, 50% felt motivated by their partner, 38% felt hurried by their partner. To feel in a hurry can positively influence performance if it leads to a higher concentration on the task. But this feeling can also influence performance in a negative way, if one feel pressed and controlled. Some participants expressed definitively negative feelings. For example, 5% of the pair-programmers felt hindered or unsettled by the partner.

Nearly all participants in the paired teams agree to the statement "*Working in pairs helps to find errors earlier*." The statement "*I feel uncomfortable while programming because my partner is observing me*" is not well agreed, also the statement that the partner is hindering. Answering the question: "*For which task of software development is working with a partner useful?*," we received the following answers (The students were allowed to give more than one answer.): 83% suggested working with a partner when looking for errors, 79% for program design, 73% for GUI design and two-third for the development of UML diagrams and for complex programming tasks. Most of the nine proposed tasks of the software development process were suggested to be done by pairs. The least accepted was "*every programming task.*" Only 36% would prefer to do every programming task with a partner together. In light of the fact that in complex software development projects there are a lot of simple programming tasks, which

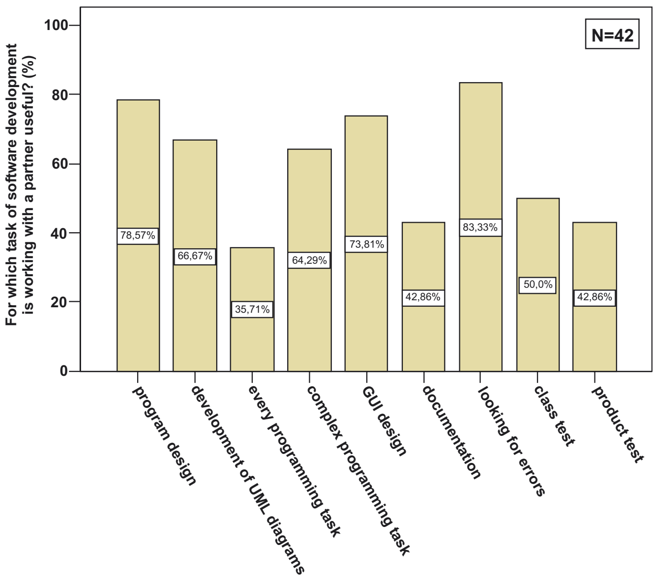

Fig. 4. Answers to the question "For which task is working with a partner useful?".

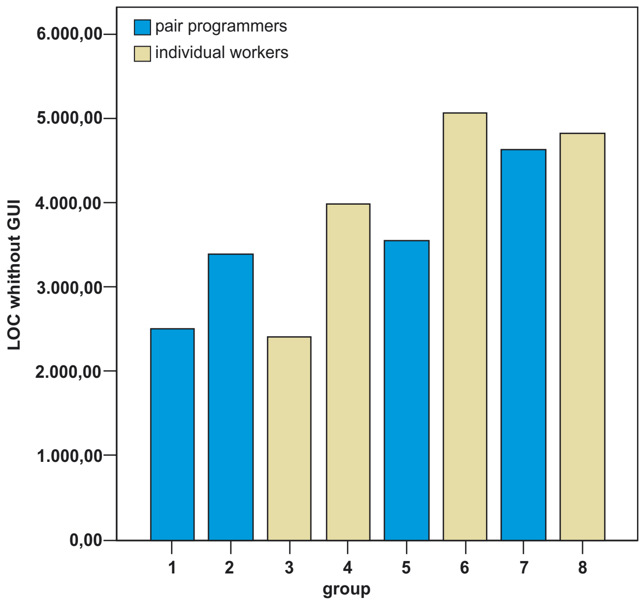

Fig. 5. Range of project for each group.

can easily be done by a single member of the team, this answering quote seems comprehensible.

In conclusion, we can state that the participants who had done their project in a paired team had a lot of positive experiences and suggested it for many tasks of software development. One group did not accept the pair programming concept as two members were against it. This is a risk in any real software development setting.

4.2. Quality of software

Besides the effect of programming in paired teams on quality of the developed software, the influence of the experimental setup on the distribution of work load within the groups was examined.

At the end of the SoPra assignment, a review of the developed programs is done by the organizers of the SoPra together with the participants. All developed programs possessed at least the expected functionality and fulfilled the requirements defined in the description of the problem which should be solved. The produced solutions differed a lot in user interface design and in the offered additional functionality. In both studies, one of the two developed programs was a game. Some of these games simulates team-mates with a strong strategy, one game shows all possible moves to the player, one explains the rules of the game exceptionally colourful, etc. The differences in functionality could not be measured objectively. But the organizers of the lab got the subjective impression that in the second study, the teams of individuals developed programs with a little more functionality compared to the paired teams.

Figs. 5–9 show results of the second study achieved by analyzing the programs of the groups by means of metrics. The metric "lines-of-code" was used to measure the amount of program code. A positive correlation between

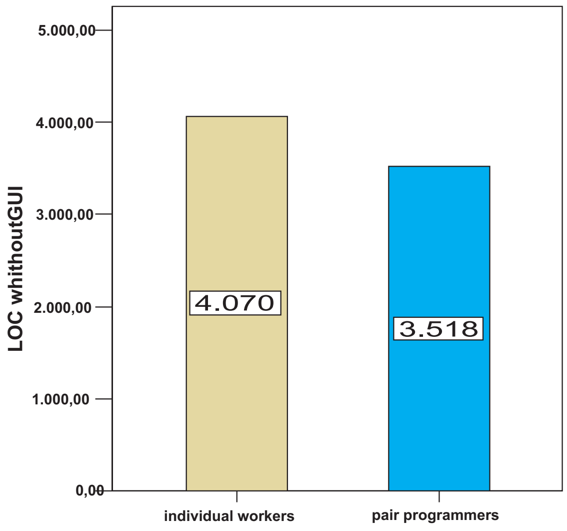

Fig. 6. Average range of projects for paired teams and teams of individual workers, respectively.

LOC and the amount of work hours performed has been proved [10]. The classes of the graphical user interface (which are normally the largest classes in our projects) are developed by means of a graphical user interface editor. Because most lines of code in these classes are generated by this tool, we ignored these classes when counting the lines. Also, we did not include empty lines and lines containing only comments.

One can see in Fig. 5 that the range of the projects scatters between about 2500 lines and about 5000 lines. One paired team and two teams of individual programmers have developed large programs with about 5000 lines of code. Medium scaled projects with about 3500 lines of code were built by two paired teams and one team of individual

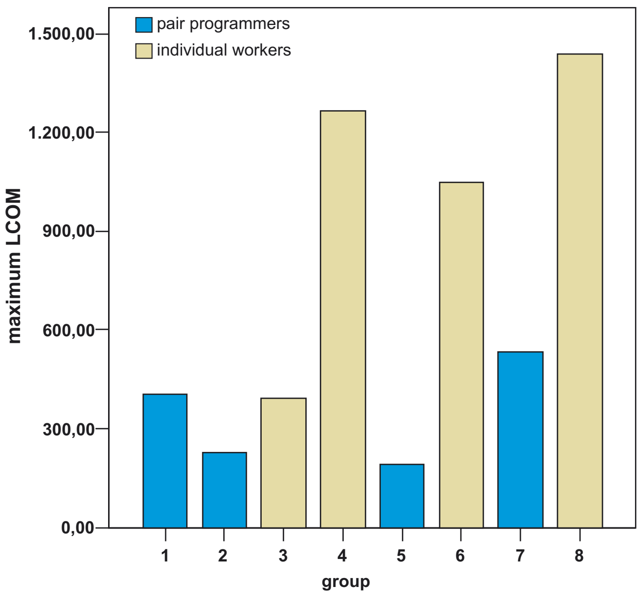

Fig. 7. Program complexity measured by LCOM.

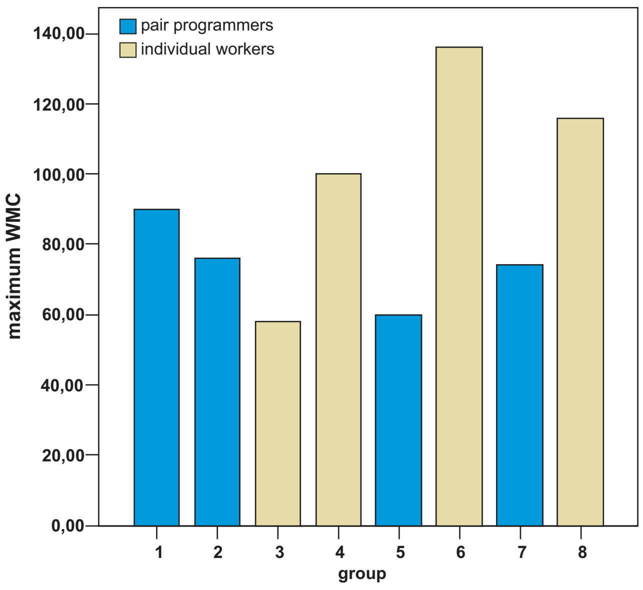

Fig. 9. Program complexity measured by WMC.

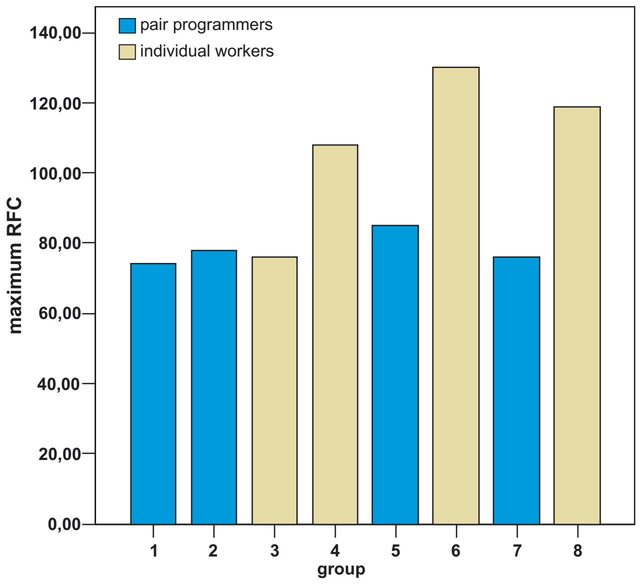

Fig. 8. Program complexity measured by RFC.

developers. Of each work organization team, one belongs to the cluster of small projects.

In Fig. 6, one can see that the paired teams produced only 13.6% less code at half of the workstations than the teams of individuals in comparable working time. In Study I, the paired teams were able to develop larger programs than the two teams of individuals. But we assume that this was a result of the higher prior knowledge of the students in these groups.

Other research studies (see Section 2) used test cases to estimate the quality of the developed software. This was possible because these projects were small and the given problems were well specified. But in this way, you can only measure the functional correctness of a program, other aspects of quality, for example, usability of the user interface or the readability of the code, are not considered.

Since we expect that pair programming positively influences the quality of the written code, we were very interested in studying this quality aspect. Especially, we wanted to analyse the complexity of the program code. Very complex code is not easy to understand, therefore maintenance is difficult. Complex code may still contain errors which had not yet been discovered and which may occur later when the program is already in use. Well written object oriented code is simply structured because using the object oriented concepts inheritance and polymorphism leads to clearly structured design [11].

Analysing the produced software by means of metric tools lead to the result that teams of pairs and teams of individuals do not use inheritance differently. The metrics DIT (Depth of Inheritance Tree) and NOC (Number of Children) do not provide different values for both types of teams. The metric CBO (Coupling Between Objects) showed some differences between the teams but we cannot detect a specific trend for pair programmers. In [15] you can find further details.

But we were able to detect some difference studying the complexity of the programs. LCOM (Lack of Cohesion in Methods) (see Fig. 7), RFC (Response for a Class) (see Fig. 8), and WMC (Weighted Methods per Class) (see Fig. 9) differed obviously for both types of teams. A high LCOM value is an indicator for parts of code in a class which do not really belong to this class. A high RFC value serves as indicator to a strong connection of one class to another class. This has to be avoided because it makes changes of the program difficult. If you change the class with the high RFC value, you probably will also have to change other classes. A high WMC value indicates that a class cannot be tested easily because the number of paths of control flow is very large. The three teams of individual workers, team 4, team 6, and team 8, reached high levels of WMC in contrast to the paired teams (see Fig. 9). Only one team of individual workers (team 3) had a similar non complex program. Their program was also the smallest program.

Although the Figs. 8 and 9 show an obvious difference between the paired teams and the teams of individuals, this is statistically not significant because the sample is too small to draw generally applicable conclusions. But we can state that there is a trend to less complex programs for paired teams.

We measured the code quality not only by means of metrics but we also asked computer science experts, colleagues at our software engineering department, to judge the quality of the programs. Based on "*bad code smells*" defined by Fowler [11], a catalogue of criterions was established to guide the software experts. Especially, understandability and readability achieved by using self-explanatory identifiers and well written comments, for example, were considered because the meaning of names and comments cannot be measured by metrics. Each of the software experts judged two classes of some of the teams not knowing if the team was a paired team or not. The code of the paired teams was rated a little bit better. Its readability and understandability were somewhat higher.

Since the members of a paired team do not only program together with their partner but also carry out every other task together during the software development, it would be interesting to compare not only the developed software of both kinds of teams but also the quality of the documents and diagrams produced in the early phases of software development. But since the judgement on the quality of the software proved to be so complex and difficult, we abandoned the idea of considering UML diagrams and documents. The evaluation of diagrams is expected to be more difficult. We are still working on this topic.

4.3. Further results

We recognized in the laboratory setting of Study I that teams of individual developers were likely to exclude less experienced members of their team. Often less able students were not integrated in the team work and it was nearly impossible for them to offer contributions to the project. The fittest students in the teams of individuals always solved the most important tasks. The less experienced students did some less critical jobs, like writing an user manual. From the viewpoint of the team, this organization of work is efficient and useful. But in a learning environment, this kind of team organization should be avoided in order to give every team member the chance to learn new topics.

To set these observations on a solid empirical basis, we included in Study II several items tapping into this aspect of work or learning behavior. By testing for differences between the two work conditions, significant results were found between participants working in paired teams on two items. Judging the statement "*The work load was not equally distributed in our group,*" the students within the groups of two settings scored significantly lower ($M_{\mathrm{pairedTeams}} = 2.68$) than the members of the other groups ($M_{\mathrm{teamOfIndividuals}} = 3.23$; t-test: $t(59) = -2.19$, $p = 0.03$). Participants in paired settings had been more able to achieve an even distribution of work load between their individual group members.

Results for answers on the statement "*I barely know some parts of the project*" are comparable. Here, the teams of individuals showed a significant higher agreement compared to the paired teams members ($M_{\mathrm{pairedTeams}} = 2.26$; $M_{\mathrm{teamOfIndividuals}} = 3.17$; t-test: $t(59) = -3.35$, $p = 0.01$). The distribution of knowledge about different project aspects was improved by working in paired teams. Support for this effect was also found within questionnaire data of Study I. Independent from prior programming knowledge, members of the paired teams reported a lower uncertainty about how to schedule their work within SoPra. Partners within the pair settings were very clear about the necessary steps in the projects or when to apply which work procedure ($M_{\mathrm{pairedTeams}} = 5.64$). In comparison, members of the individual teams reported significant lower values, even independent from their prior experiences in programming ($M_{\mathrm{teamsOfIndividuals}} = 5.15$; ANCOVA: $F(1, 22) = 6.19$, $p < 0.05$).

5. Conclusion and limitations

Why do eight developers at four workstations not need twice as much work time as eight developers at eight workstations to solve the same problem? We conclude from our inquiries and our observations that teams working in pairs with changing partners benefit a lot from this type of work organization because they gain more knowledge of all parts of the project. The paired teams use their work time more efficiently because they concentrate much more on their task. If someone needs help, the partner is always nearby to answer questions. During the development of complex software much time is spent in finding errors. Testing and bug fixing in particular is done much easier by two developers than by one.

All pair programmers assess working together with a partner as very positive. Pair programmers take advantage of the higher quality of their code which is less complex, better to read, and easier to understand. This supports finding errors faster.

Most of the pair programmers underline the benefit of pair programming in the questionnaires, while only few of the students who worked in a paired team gave negative comments. One SoPra team was excluded from the experiment because two members in this team refused to work with a partner. If working as a paired team is not accepted by all team members, it cannot be realized.

We cannot yet answer the exciting question of whether doing every task of software development together with a partner leads to more knowledge about software development. The self-evaluation on the newly acquired knowledge by the participants does not show a statistically significant difference between students who worked with a partner and students who worked alone. All participants of both groups declare to have learned much or very much. However, the pair programmers in our study stated that they gained more knowledge on the entire project they have done together with their team mates. Therefore, it can be concluded that pair programmers have learned more about software development.

On the one hand, the loss of efficiency resulting from pair programming is very small and this is the only disadvantage we have seen. On the other hand, the quality of the developed code is higher and the integration of less experienced team members is easier. Therefore, we emphatically recommend working in pairs for teams of students.

The experiments at the university offered the opportunity to compare two different kinds of teamwork for software development under controlled conditions with a number of teams who did the same projects at the same time. Nevertheless, our study setup and conclusions from it have to keep in mind several limitations. Although we conducted our studies within a lab setup, we were not able to control all aspects of work conditions within the six weeks lasting course. Violations to our intended work settings (e.g., like working in teams for teams of paired workers or working by two within teams of individuals) can therefore be critical for making causal interpretations of our results. We did not see clear signs of diffusion or imitation of the paired team concept within the control groups, but all students saw each other on a regular basis during the study, so that an exchange of ideas and experiences probably took place. To gain clearer results, future studies could, for example, realize a diversified experimental setup (e.g., with waiting control groups) to account for these kinds of problems. Furthermore, we analysed data mainly on the basis of individual statements on the group level. To gain clearer results, future research should concentrate on the possibility doing hierarchical analyses for testing for effects, not only taking into account results at the group but also individual level.

A next step should be to test whether our results can be transferred to industrial software development settings. Limitations to expand our results can obviously be seen within our experimental lab settings, the use of a student sample, a small number of groups and the used tasks which all limit the external validity of our results. Although the integration of the study within a mandatory course within the study program offered us the opportunity to test our hypotheses on a heterogeneous sample (not only volunteers), the experiences show that some people refused to cooperate within the lab setup or the pair programming concept. To specify the causes for this behavior and to replicate our results with different and larger samples and varying operationalization methods of core constructs should be the next steps within our line of research to generate reliable results.

References

[1] M.R. Barrick, G.L. Stewart, M.J. Neubert, M.K. Mount, Relating member ability and personality to work-team processes and team effectiveness, Journal of Applied Psychology 83 (3) (1998) 377–391.

[2] K. Beck, Extreme Programming Explained: Embrace Chance, Addison Wesley, 2000.

[3] I. Beckmann, D. Schmedding, Experimente mit XP in der Lehre, GI Jahrestagung 2 (2004) 122–126 (in German).

[4] T. Bipp, J.Hüvelmeyer, Das INWIDA Labor, Available from: <http://www.inwida.uni-dortmund.de> (in German).

[5] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language – User Guide, Addison Wesley, Reading, MA, 1999.

[6] P. Borkenau, F. Ostendorf, NEO-Fünf-Faktoren Inventar, Handanweisung, Hogrefe Verlag für Psychologie, 1993 (in German).

[7] J.A. Breaugh, J.P. Colihan, Measuring facets of job ambiguity: construct validity evidence, Journal of Applied Psychology 79 (1994) 191–202.

[8] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.

[9] L.J. Cronbach, Coefficient alpha and the internal structure of tests, Psychometrika 16 (1951) 297–334.

[10] N.E. Fenton, S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, International Thomson Computer Press, 1996.

[11] M. Fowler, Refactoring – Improving the Design of Existing Code, Addison Wesley, 2000.

[12] W. Hays, Statistics, Harcourt Brace, Orlando, FL, 1993.

[13] C. Kopka, D. Schmedding, J. Schröder, Der Unified Process im Grundstudium – Didaktische Konzeption, von Lernmodulen und Erfahrungen, DeLFI 2004, pp. 127–138 (in German).

[14] P. Kruchten, The Rational Unified Process: An Introduction, Addison Wesley, 1999.

[15] A. Lepper, Eine empirische Studie über Paararbeit in der Softwaretechnik, Master thesis at the department of computer science at the University of Dortmund, 2005 (in German).

[16] C. McDowell, L. Werner, H. Bulock, J. Fernald, The effects of Pair-programming on performance in an introductory programming course.ACM SIGCSE Bulletin, in: Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education, vol. 34, No. 1 (2002).

[17] C. McDowell, L. Werner, H. Bulock, J. Fernald, The impact of pair programming on student performance, perception and persistence, in: Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, 2003, 602–607.

[18] M.M. Müller, Two controlled experiments concerning the comparison of pair programming to peer review, The Journal of Systems and Software 78 (2005) 166–179.

[19] M.M. Müller, F. Padberg, An empirical study about the feelgood factor in pair programming, In International Symposium on Software Metrics, Chicago, September 2004.

[20] J.T. Nosek, The case for collaborative programming, Communications of the ACM 41 (3) (1998).

[21] K.-H. Schmidt, S. Hollmann, Eine deutschsprachige Skala zur Messung verschiedener Ambiguitätsfacetten bei der Arbeit, Diagnostica, 44, 1998, pp. 21–29 (in German).

[22] L. Williams, R.R. Kessler, W. Cunningham, R. Jeffries, Strengthening the Case for Pair-Programming, IEEE Software, July/August 2000.

[23] L. Williams, R.R. Kessler, Pair Programming Illuminated, Addison Wesley, 2003.