Effects of Pair Programming at the Development Team Level: An Experiment

Jari Vanhanen and Casper Lassenius
Helsinki University of Technology, Software Business and Engineering Institute
P.O. BOX 9210, FIN-02015 Finland
firstname.lastname@hut.fi

Abstract

We studied the effects of pair programming in a team context on productivity, defects, design quality, knowledge transfer and enjoyment of work. Randomly formed three pair programming and two solo programming teams performed the same 400-hour fixed-effort project. Pair programming increased the development effort of the first tasks considerably compared to solo programming, but later the differences were small. Due to this learning time the pair programming teams had worse overall project productivity. Task complexity did not affect the effort differences between solo and pair programming. The pair programming teams wrote code with fewer defects, but were less careful in system testing, and therefore delivered systems with more defects. They may have relied too much on the peer review taking place during programming. Knowledge transfer seemed to be higher within the pair programming teams. Finally, we also found weak support for higher enjoyment of work in the pair programming teams.

1. Introduction

Pair programming is an intensive form of collaboration where two programmers design, code and test software together at one computer [1]. It has been reported that pairs produce better designs with fewer defects in the code, in shorter elapsed time and more enjoyably without using significantly more effort than solo programmers [2, 3]. Benefits for teamwork, knowledge transfer and learning have also been proposed [4]. Improved knowledge transfer decreases the risk of having and losing critical persons and facilitates new persons in becoming productive developers. If the increase in the development effort is low or nonexistent, the realization of just some of the proposed benefits would make pair programming affordable from a project's or organization's viewpoint. Two economic models for evaluating the feasibility of pair programming in different development projects have been proposed [5, 6, 7]. They contain variables such as effort difference, quality difference, personnel cost and economic value of shorter time to market. No experiences on using these models have been reported.

The results of previous pair programming experiments contain apparent contradictions [8] and have mostly studied individuals and pairs doing small tasks in isolation from other developers, i.e. in non-team context. Our objective was to execute a well-planned experiment where co-located teams develop a larger piece of software using either pair programming or solo programming, and as a result get more data on the effects of pair programming.

Section 2 discusses earlier experiments. Section 3 presents our hypotheses and the experimental setting. Section 4 presents and discusses the results of our experiment. Section 5 evaluates the experiment. Conclusions are drawn in Section 6.

2. Related work

Several pair programming experiments have been reported in the software engineering literature. Below we summarize the most relevant ones emphasizing those similar to our experiment.

2.1. Experiments in non-team context

Most experiments have studied the effects of pair programming on effort and quality when small, isolated tasks were developed by pairs and individuals working as isolated entities instead of in a team. Effort increases of about 100% have been reported in [9, 10, 11], a 42% increase in [3], and a 60% increase for the first task followed by a 15% increase later in [2].

Improvements in quality based on defect count or test case pass rate have been reported in [2, 3, 9, 12]. A similar level of quality of the work results was ensured only in [10] by measuring quality by the number of resubmissions to acceptance testing until all tests passed. No quality difference between pairs and individuals was found in [10], but the quality metric differed quite a lot from those used by other researchers. These experiments propose that pair programming improves quality but requires some extra effort.

0-7803-9508-5/05/$20.00 (c)2005 IEEE

2.2. Experiments in a team context

Three experiments [2, 13, 14] studied pair programming in a team context on university courses. They are summarized in Table 1 and described below.

Table 1. Experiments done in a team context

	Williams [2]	Ciolkowski [13]	Baheti [14]
Comparison	PP vs. solo	PP vs. unsystematic collaboration	PP vs. solo
N	7 + 3 teams, 4 in a team	3 + 3 teams, 6 in a team	9 + 16 teams, 2-4 in a team
PP training	effective pp was taught	some material was given	?
Sw size	?	~4000LOC	?
Effort	?	~240h	?
Length	4 weeks	5 weeks	5 weeks
Results (PP vs. the other way)			
Effort	-28%	9%	3%
Quality	-2% (passed test cases)	smaller LOC and coupling factor	1% better grade

The experiment which studied isolated pairs vs. individuals in [2] was continued by assigning two pairs or four individuals to four-person teams performing four-week projects [2 p. 77-78, 100-101]. The pairs were formed based on with whom each student wanted to work. The proportion of high, average and low performers classified based on their GPA was similar among pair programmers and solos. All teams continued using the same type of programming as before, i.e. pair programmers were already familiar with pair programming and partly also with each other. The pair programming teams used 28% less effort, but passed 2% less test cases.

Ciolkowski and Schlemmer [13] had six-person student teams spend 13 weeks and about 700 hours of effort for the projects. However, the experiment covered only the programming phases lasting 5 weeks and taking about 240 hours of effort. The researchers were not able to evaluate quality using defect metrics, but analyzed LOC and the coupling factor instead. These metrics showed slightly better values for the pair programming teams. The pair programming teams spent 9% more effort for the programming tasks (including test case writing).

Baheti [14] studied pair programming with students working in self-selected teams of 2-4 persons. Each team chose whether they were going to use pair programming or solo programming. Each team made one of several available assignments during a 30-day project. The pairs needed 3% more effort for writing the same amount of LOC (14.8 vs. 15.2 LOC/h). The quality of the systems was evaluated by a teaching assistant based on only a 30 minute demo. The pairs received slightly better grades (93.6 vs. 92.4 on scale 0-110).

Compared to the experiments where individuals and pairs worked as isolated entities doing small tasks, the effort increase incurred by pair programming was much smaller or even negative. Comparing quality is hard due to the different metrics used, but it seems that the quality improvements were smaller in the team context. However, it may be that the smaller effort used decreased the quality. None of the results of the team experiments were statistically significant as can be expected with such small numbers of teams.

2.3. Other experiments

Müller [15] made a slightly different experiment comparing pair programming to solo programming combined with code reviews using 27 experienced students as the subjects. His results suggest that reviews produce the same code quality to a slightly lower cost than pair programming. The use of pair programming on introductory programming classes has been studied with hundreds of students at North Carolina State University [16, 17, 18] and University of California Santa Cruz [19, 20]. The results show some improvements in quality and some increase in the total effort.

2.4. Summary of previous work

In most experiments individuals and pairs worked as isolated entities doing small, isolated tasks. Therefore it is hard to generalize the results to industrial settings, where large systems are developed by teams. The experimental designs have been diverse between studies, they have not been described accurately, and some have contained deficiencies making their replication impossible or useless. LOC as a design quality metric or evaluating software quality based on a short demo are not reliable metrics. Comparing different projects such as GUI building or development of an algorithm may cause variation in the results. Allowing the subjects to choose their partners and whether to use pair or solo programming differs from a randomized setting.

Probably due to all these differences and deficiencies the results of previous pair programming studies are quite varying and especially the results about the additional effort required differ considerably. More experiments using improved and accurately reported experimental designs are clearly needed. Next we describe our experiment, in which we tried to address some of the above shortcomings.

3. Research design

The research consisted of an experiment whose design we describe below. Then we present the hypotheses we aimed to study. We considered the guidelines for empirical research by Kitchenham et al. [21] when planning and reporting this experiment.

3.1. Experimental setting

Five teams of four developers did a similar project using the same development process, work practices, tools, and specifications. The experiment had a one-factor randomized design [22], where the factor was the type of programmer collaboration. Three teams used pair programming (PP) and two solo programming (SP). The PP teams had to use pair programming for all development work whereas the SP teams were not allowed to use pair programming for more than occasional collaboration, i.e. not for implementing whole use cases together. Pair programming was taught to the PP teams on a one-hour lecture.

The experiment was conducted as a non-compulsory J2EE course in the spring of 2004. We had twenty participants, all at least 4th year computer science students at Helsinki University of Technology. Their programming experience was 1.5-10 years (avg. 4.7), of which 1-6 years (avg. 2.2) was using Java. Their average grade from previous programming courses was 3.9 on a scale from 1-5 (5=best), and they all considered themselves average or better programmers compared to their fellow students.

We ranked the participants by their programming skills, i.e., the effort spent on two J2EE programming assignments, previous programming experience, average grade from programming courses, and their personal opinion on their skills compared to fellow students. We formed the teams randomly so that all teams had one person from each quartile of the ranking. Finally we randomly selected whether a team should use pair programming or solo programming. The requirement of using pair programming by random participants was mentioned in advance.

The course began with a two-week J2EE training period (about 15h of lectures) followed by a nine-week project. The course was evaluated on a pass/fail scale. Passing required participation in the J2EE training, working 100h for the project, following certain work practices and answering three questionnaires.

The project included developing, testing and delivering a distributed, multi-player casino system using the J2EE technologies as described in a requirements specification containing, e.g., use case descriptions and HTML layouts for the web user interface. The teams were given a 10-page technical specification and a core architecture implementation including examples of suitable J2EE design patterns and a build script. The development tools used were Eclipse 3, J2EE 1.4 SDK, XDoclet, JBOSS with Tomcat, Hypersonic SQL, CVS and Ant.

The project effort was fixed to 400 hours, i.e., 100h per person. Everyone had to spend at least 75% of the effort in co-located team sessions lasting 4-8 hours. The project consisted of a one-week project planning phase followed by two four-week implementation iterations. The teams had to follow work practices such as iteration planning, collective ownership, version control, coding standard, continuous refactoring, unit testing, system testing, time reporting, defect reporting, and documenting. The prioritized project goals were to: 1) follow the defined work practices, 2) minimize the amount of defects, 3) implement as many use cases as possible, and 4) avoid wasting effort on activities that do not directly contribute to the project.

3.2. Hypotheses

We derived a set of hypotheses to be studied in the context of comparing PP teams to SP teams. The hypotheses were derived mostly based on the literature but also on what we have personally learned from discussions with industrial developers who have used pair programming in their work.

3.2.1. Productivity. We define productivity as the amount of work results divided by the effort spent. Project productivity is the sum of the implemented use cases divided by all effort spent for the project. Use case productivity is the inverse of the development effort (designing, coding, unit testing, bug fixing and documenting the code) of a use case.

H 1.1: The PP teams have lower use case productivity than the SP teams.

Previous research [2, 3, 9, 10, 11] has found an increase of 15-100% in the programming effort when using pair programming without a team context for small, separated tasks.

H 1.2: Higher use-case complexity favors PP teams as measured on use case productivity.

Williams and Kessler propose using pair programming at least for complex tasks [1]. We have heard from many practitioners that pair programming is quite useless for trivial tasks but helpful for complex tasks.

We estimated the use case complexity by the opinions of the solo developers who implemented the use cases. Their opinions reflect the complexity as perceived when using the traditional way of programming.

H 1.3: The PP teams have higher project productivity than the SP teams.

Even though pair programming may cause some

additional effort for a programming task, it may increase the overall project productivity because some proposed benefits of pair programming, such as better knowledge transfer, are likely to become more significant in a project team context.

Previous experiments [2, 13, 14] have found a difference between -28% and 9% in the project effort when comparing pair to solo programming. The worst result (9%) was reported in [13], but only the effort spent for the programming phase was analyzed, which may disregard some of the project level benefits of pair programming.

In our experiment all teams spent the same total effort. The effort data was reported per use case or other tasks such as system testing using a web-based system. The amount of work results was measured as the amount of successfully implemented use cases. All teams implemented the use cases in the same order. We tested the systems after delivery in order to ensure that the use cases were implemented without major defects and to count the number of defects.

The quality of work results must be similar in order for the productivity comparison to make sense. Because a developer makes the final decision of the readiness and quality of a piece of code, we urged them to aim for high and thus hopefully similar quality.

3.2.2. Defects.

H 2.1: After coding and unit testing the PP teams have fewer defects than the SP teams.

H 2.2: After system testing and bug fixing the PP teams have fewer defects than the SP teams.

Several experiments report smaller defect counts for pair programming [2, 3, 9, 12]. In our experiment each team had to report all defects found related to a use case after the unit testing and corresponding fixing of the use case had been performed. The defects in the delivered systems were counted based on a standardized system testing round performed by a researcher.

3.2.3. Design quality.

H 3.1: The PP teams create better software design than the SP teams.

Previous studies propose that pair programming improves design quality [2, 10], but the claim has been mostly based on a smaller value of LOC, which is quite controversial as a metric of design quality, because fewer lines of code is not always better. In [13] the coupling factor metric was additionally analyzed and showed a slightly smaller value for the PP teams' code. We have heard practitioners report that pair programming produces more understandable code.

In our experiment the core architecture, which we gave to the teams largely defined the system level design. Therefore we analyzed design quality on the method level, which should be less affected by the core architecture than the system level design. We used NCLOC per method to characterize the method size, McCabe's cyclomatic complexity [23], i.e., the number of flows through a method to describe the method complexity, and the number of parameters to tell how much information is passed to the method. Reasonably small values for these metrics typically indicate good design.

3.2.4. Knowledge transfer. Both the breadth and depth of the understanding of a system can be analyzed. Depth characterizes how well a person understands a certain module and breadth how many modules a person understands at some depth. The understanding can be analyzed also from the perspective of a module, e.g., how many persons understand a module at some depth.

H 4.1: In the PP teams each developer understands more modules well than in the SP teams.

H 4.2: In the PP teams more developers understand each module well than in the SP teams.

Williams and Kessler propose that pair programmers, especially if the pairs are rotated, know more about the overall system [1], but we have not seen any studies on this.

When pair programming is used, tasks (use cases in our experiment) are allotted to only half the number of worker units (pairs) compared to solo programming. Therefore in a PP team each developer participates in twice as many tasks as in an SP team, if the same tasks are completed in both teams. This distributes a developer's involvement in the development of different modules more broadly in a PP team.

The amount of involvement surely affects the depth of a developer's understanding of a module. The acquired depth of understanding with the same amount of involvement may differ between solo programming and pair programming. Pair programmers may learn from their partner and acquire deeper understanding than when working alone. However, a passive partner may acquire only a shallow understanding.

We asked the developers' involvement in and understanding of the modules after the project using a web questionnaire.

3.2.5. Enjoyment of work.

H 5.1: In the PP teams developers enjoy their work more than in the SP teams.

It has been reported that most developers like pair programming [1]. Some developers have told us that developing critical systems with a pair increases their confidence in the code reducing work-related stress.

We asked the developers' feelings about pair programming after the project using a web questionnaire.

4. Results and discussion

All five teams finished the projects according to the schedule and spending the required 400 hours. However, one of the PP teams abandoned rigorous pair programming without notice in the middle of the project because they considered it inefficient. Their productivity compared to the other groups did not change noticeably after they started to use pair programming only sporadically, and the team was the least successful team based on the amount of use cases they completed. Their data was removed from the analysis, because they were neither a true PP nor SP team.

4.1. Productivity

Both SP teams finished more use cases than the PP teams as shown in Table 2. The differences in the amounts of implemented use cases are enlarged by the smaller effort required for implementing the latter use cases (see Figure 1). Therefore we estimated the size of the systems by summing up the sizes of implemented use cases. The size of a use case was estimated by the median of the effort different teams spent on implementing it. Based on the system sizes the PP teams had 29% lower project level productivity than the SP teams. This corresponds to 40% ((266-190)/190) higher effort refuting H 1.3.

Table 2. The sizes of the systems (μ=mean)

	PP1	PP2	SP1	SP2	μ_{PP}	μ_{SP}	PP vs. SP
use cases (#)	20	10	25	27	15	26	-42%
system size (sum of use case sizes)	226	154	258	273	190	266	-29%

The reason for the lower productivity can be seen in Figure 1. Both of the PP teams performed very poorly when implementing the first three (PP2) or four (PP1) use cases. Thereafter the differences between the PP and SP teams are minimal. Actually, PP1 spent less effort than either of the SP teams for their last use cases (17-20). Of course, when spending lots of effort on the first use cases, the PP teams may have learned something that helped them with the later ones.

Table 3 shows the efforts for certain subsets of the use cases. For use cases 1-10, which were implemented by all four teams, the PP teams used on average 44% more effort than the SP teams. If we ignore use cases 1-4, where the PP teams performed poorly, the PP teams used 5% less effort than the SP teams.

Table 3. The efforts for subsets of use cases

Use cases	μ_{PP}	μ_{SP}	PP vs. SP
1-10	191h	133h	+44%
1-4	121h	59h	+107%
5-10	70h	74h	-5%

The SP teams spent slightly more effort on general bug fixing, some of which was probably related to certain use cases but is ignored in Table 3. There was almost no difference in the amount of effort spent for non-implementation tasks meaning that the effects of pair programming to the project's productivity were based on the differences in the use case efforts.

A similar inefficient learning time was reported in [2] where the pairs spent 60% more effort for the first and 15% more for the later assignments with the same pair. In [13] the effort increase was 9% in both studied iterations indicating no learning effect, but the reason may be that the developers had worked as a team for several weeks before the observed iterations. In our experiment four use cases were needed before everyone had pair programmed with everyone once, which can explain the poor performance of the PP teams with the first three or four use cases. If we ignore these use cases, pair programming required the same or smaller effort than solo programming refuting H 1.1.

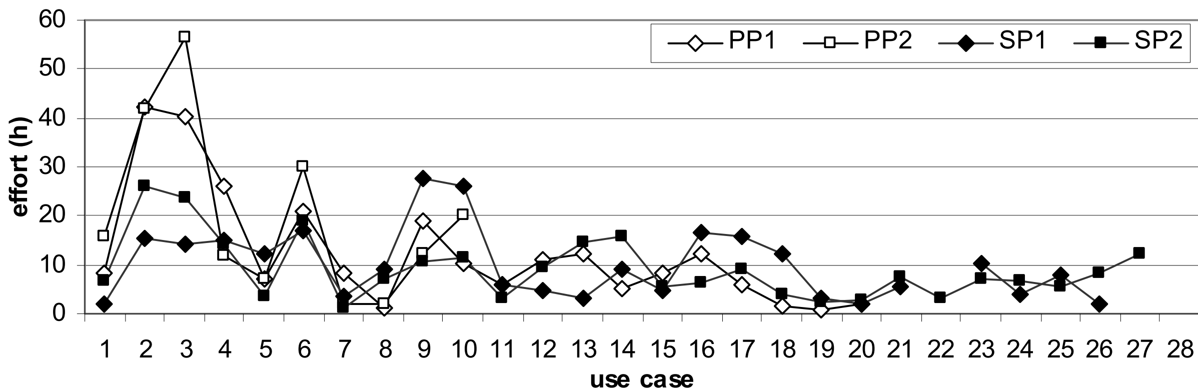

Figure 1. The efforts spent on individual use cases

The diamonds in Figure 2 show the complexity of each use case as evaluated by the solo programmers (1=very easy, 5=very difficult). The squares show the ratio between the efforts used by the SP teams vs. PP teams. For example, 0.5 means that the SP teams used half the effort of the PP teams and 2.0 means doubled effort. There is no Pearson correlation (r=-0.02) between complexity and effort difference refuting H 1.2. This finding contradicts with the results in [1] and the opinion of most pair programmers with whom we have talked. It may be that the feeling of usefulness of pair programming comes from the assumed higher resulting quality, and thus developers' opinions are not solely based upon effort differences. The researchers of the social facilitation theory dealing with the impact of social presence on individual performance have found that social facilitation effects impair performance in case of complex tasks [24]. These studies have focused on studying persons who are not familiar with each other, which was also the case in the beginning of our experiment, when the pairs performed poorly.

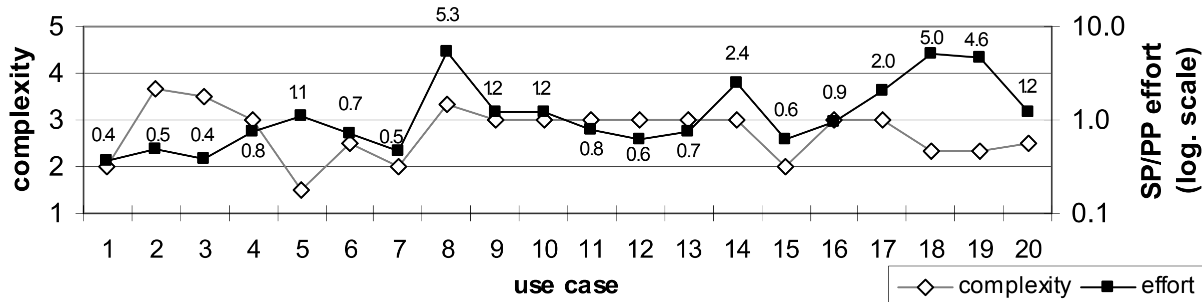

Figure 2. Perceived complexity of a use case vs. effort ratio between the SP and PP teams

4.2. Defects

Pre-delivery defects are defects found by the team during system testing at the end of an iteration or during development if the defect was related to a use case, which the responsible developer/pair considered to be ready. Post-delivery defects are defects found by an external tester after delivery. Defect density per implemented use case and absolute amounts of defects are listed in Table 4. The number of use cases for the PP2 team is larger than in Table 2 because two use cases were not accepted due to major defects.

Table 4. The defect densities of the systems

	PP1	PP2	SP1	SP2	μ_{PP}	μ_{SP}	PP vs. SP
use cases	20	12	25	27			
pre-delivery	0.95 (19)	0.75 (9)	1.60 (40)	0.78 (21)	0.85	1.19	-29%
post-delivery	0.30 (6)	0.33 (4)	0.12 (3)	0.04 (1)	0.32	0.08	303%
sum	1.25 (25)	1.08 (13)	1.72 (43)	0.81 (22)	1.17	1.27	-8%

The SP teams found more pre-delivery defects than the PP teams. The numbers are affected by both the total number of defects in the software and the effectiveness of finding them. The effectiveness may have differed even though the total effort for system testing between the teams was similar. The post-delivery defects were found in standardized testing of all systems. Therefore the sum of both defect types is our best estimate of the total defect count in the code when the responsible developer(s) considered it ready. To conclude pair programmers made 8% less defects to the code during the development supporting H 2.1.

Interestingly, after delivery the PP teams had considerably more defects than the SP teams, refuting H 2.2. The reason was described above, i.e. the SP teams found and fixed more defects before delivery. The percentage difference is huge, because the absolute amounts of post-delivery defects were so small. The small number is partially explained by the focus of the post-delivery system testing on the basic functionality and common error situations instead of more exotic error situations, which the teams themselves tested. Probably the PP teams had a less careful attitude towards system testing due to relying too much on the peer review during pair programming.

4.3. Design quality

We analyzed the source codes using the Metrics plug-in for Eclipse. Table 5 shows the metrics for the core architecture and delivered systems. The NCLOC metric contains only non-blank, non-commented lines inside method bodies. The systems contained additional 4200-5600 lines of comments mostly intended for generating J2EE bean classes automatically by a tool. Generated code is excluded from the metrics, but the code for the core architecture is included, because it cannot be separated from the rest of the code.

All the method level metrics show slightly better average values for the PP teams. However, it seems that the values correlate with NCLOC, which was naturally higher for the SP teams, who implemented more use cases. The core architecture had a good design, but when more code was written on top of it, the less the original code affected the metrics. Therefore it is hard to say whether the smaller values are due to the smaller NCLOC or due to the use of pair programming.

Table 5. The code metrics of the systems

Metric	Core	PP1	PP2	SP1	SP2
System level metrics					
Use cases	0	20	12	25	27
Methods	202	565	525	643	704
NCLOC	1022	4180	2635	4956	5550
Method level metrics					
NCLOC (avg.)	4.82	7.16	4.96	7.61	7.67
McCabe CC (avg.)	1.69	2.25	1.71	2.35	2.36
Parameters (avg.)	0.72	0.72	0.64	0.73	0.80
Proportion of bad methods (%)					
NCLOC > 50	0.0	2.1	0.8	2.3	2.3
McCabe CC > 10	0.5	2.3	1.0	2.5	2.3
Parameters > 5	1.0	1.8	0.8	0.8	0.6

Analyzing the proportion of bad methods should be less dependent on the size of the software. The proportion of very long methods is smaller for the PP teams. The proportion of complex methods is clearly smallest for PP2, but for other teams there are no differences. The proportion of methods with a long parameter list is clearly best for SP2 and worst for PP1.

The differences between the PP and SP teams depend on the metric used and the metrics may be affected by the size of the analyzed system. Thus, we cannot say anything conclusive regarding H 3.1.

4.4. Knowledge transfer

All 16 developers evaluated their involvement in the development of each Java package and their understanding of the internal structure of the packages using the scale shown in Table 6. All systems contained the same packages originating from the core architecture.

The involvement and understanding correlated (Spearman's rho > 0.5, significance level 0.01, 2-tailed) for eight of the ten packages. The number of persons involved at least "quite a lot" in the development of a package was higher in the PP teams for six packages, the same for two, and lower for two packages. In the PP teams, on average 1.3 of 4 developers were involved at least "quite a lot" in the development of each package, compared to 1.1 in the SP teams.

Table 6 shows the number of packages that the developers on the average understood on at least a certain depth. The differences between the PP and SP teams are quite small and depend on the chosen threshold for understanding. The developers in the PP teams understood well, i.e. answered 4 or 5, 32% (4.5 vs. 3.4) more packages. This supports H 4.1, but the situation changes for the other thresholds.

Table 6. The understanding of the packages

Depth of understanding	Number of packages	
	PP (avg.)	SP (avg.)
=very much (5)	0.8	1.0
>=quite a lot (4)	*4.5*	*3.4*
>=some (3)	6.9	7.1
>=little (2)	8.9	8.5
>=none (1)	10	10

Next we analyzed how many developers understood well each individual package. The PP teams had a larger value for seven packages, the same for two and lower for one. In the PP teams, on average 1.8 of 4 developers understood each package well compared to 1.4 in the SP teams. This result supports H 4.2.

According to the Mann-Whitney U-test [25] none of the differences between pair and solo programmers presented below were statistically significant. This was quite natural due to the small sample size.

4.5. Enjoyment of work

The enjoyment of the developers about the way their team did the development work on a scale of "1-It was terrible" to "5-I liked it a lot" are shown in Table 7. The SP2 team had the most satisfied developers. However, seven developers in the PP teams liked the way they worked (answered 4-5) compared to only five in the SP teams. This gives some support for H 5.1.

Table 7. The enjoyment of work

	PP1	PP2	SP1	SP2
Enjoyment	4, 4, 4, 5	2, 4, 4, 5	2, 2, 3, 4	5, 5, 5, 5

Table 8. The feelings about pair programming

Which do you …	PP	SP	Neutral
like more?	3	4	1
consider better for the overall success of this kind of a project?	2	5	1

Two other questions were also asked (Table 8). Three of the eight developers in the PP teams preferred pair programming and four solo programming. However, in this kind of a project only two considered pair programming the more successful choice.

5. Evaluation of the experiment

The strengths of the experiment and the threats to the validity of the results are discussed next. Though we aimed at performing a well-defined experiment as carefully as possible, several threats to both internal and external validity remain.

5.1. Strengths of the experimental design

The experiment was done in a context of a co-located team and a moderately large project compared to the other experiments. The requirement of co-location for all teams is important, because it alone may increase knowledge transfer within a team considerably. It also enforces simultaneous work between the developers thus increasing realism.

The participants represented quite well industrial professionals. Their programming experience was on average 4.7 years and many had experience of professional software development. Also due to the voluntary participation and content of the course, the participants were especially motivated and skilled programmers.

The project topic and technologies enticed quite many students to the rather laborious course. The J2EE training and the core architecture allowed the students to start real work quite soon, and there were almost no problems related to the requirements specification, architecture, or any other materials.

The only comparable experiment whose experimental setting has been orderly planned and reported is [13]. However, it analyzed only the programming phases of the project, not including, e.g., design activities. They also failed to study any defect metrics. In [2] the results of the team experiment were reported very shortly, and in [14] neither a randomized study design was used nor similar projects done by the different teams. All the other previous studies have concentrated on observing pairs as isolated entities.

5.2. Threats to internal validity

The teams had balanced average skill levels, but the project productivity and the experience of the most experienced developer in a team correlated highly. A very skilled developer may have a huge effect in a project that includes learning challenging new technology.

The participants were not fully controlled by the researchers during the project. This may have affected their discipline in following the development practices. It has also been proposed that pair programmers are more disciplined in following the process, which might cause some differences between the PP and SP teams.

Reporting effort for a certain use case was not necessarily uniform. Some consecutive use cases were slightly related to each other and thus some work may have contributed to several use cases. Bug fixing and re-testing effort was also reported in a slightly different way. These issues do not affect the project productivity analysis, but may have introduced a small error for the productivity analysis of individual use cases.

The productivity of the PP teams probably changed during the project in a different way than that of the SP teams due to inefficient learning time and better knowledge transfer in the PP teams. This may have caused inaccuracy to the analysis of the correlation between use case complexity and the pair programming efficiency.

Acceptance testing was done by a researcher, who knew how many defects the teams themselves had found and whether a system was done by a PP or SP team. However, possible bias in testing was minimized because the same test cases were run for all systems as equally as possible, the only difference being caused by the random outputs of the games.

Evaluating design quality with code metrics may have been unreliable, especially because the compared systems contained different amounts of functionality. There is no common understanding of which code metrics best reflect good design, and code metrics may be awkwardly affected by the size of the software. The core architecture also certainly affected the metrics.

The questionnaire about the involvement in and understanding of the packages was made in the end of the projects using a web form listing all the packages. The students were asked to answer the inquiry carefully, but we do not know how careful they were and, e.g., if they checked the code when answering. The scale may also have been interpreted in a different way, e.g., the respondents may have compared themselves to the other team members. It would have been more reliable to arrange an objective test of understanding and cross-check the involvement from the time reporting data.

5.3. Threats to external validity

The requirement to use pair programming for all development work was not the most natural and possibly also not the most useful choice. The optimal amount of pair programming may be anywhere between using it for all development tasks by everyone and not using it at all. Pair programming was taught to the students but we did not observe how actively they changed roles and communicated during the pair programming sessions. For example, Dick and Zarnett report about a case where switching roles did not work despite of frequent intervention by the team coach [26].

In our experiment all teams were having a kind of a meeting whenever they had a team development session and there were not any external co-workers

around disturbing them. In a typical industrial setting pair programming may decrease the amount of interruptions by other people compared to when a person is working alone, but this effect probably did not show up in our experiment

We do not know how well the team members knew each other and what kind of personalities they were. Both of these variables may have affected the success of the teams. Potential participants knew before entering the course that half of the participants must use pair programming, which may have kept away people who are strongly against pair programming.

The results are not statistically significant due to the low number of teams and high variations in the response variables within the SP and PP teams

6. Conclusions

This work studied the effects of pair programming on development effort, software quality, knowledge transfer and enjoyment of work. The results certainly shed some more light on the topic, even though this experiment, like all the previous ones, contained several deficiencies such as the small sample size. Hopefully this work invites others to execute even better pair programming experiments.

The PP teams had 29% lower project productivity than the SP teams. However, the reason was the considerably larger effort they spent for the first three or four use cases. The inefficiency was probably caused by the learning time involved in getting familiar with new people and with the pair programming practice. Later in the projects the PP teams spent 5% less effort than the SP teams for implementing the use cases. If the inefficient learning time is not taken into account, the productivity of the PP teams seems to be equal to that of the SP teams. In a typical software development organization the learning time can usually be neglected because most people already know each other and at least after the first pair programming project are familiar with pair programming. Even if there were still some learning time involved, the cost of a day or two per developer for learning is insignificant. The claim that pair programming is most useful with complex tasks was not supported by this experiment, at least from the perspective of the required effort.

The code written by pair programmers contained 8% less defects per use case when the responsible developers considered the code ready. However, the SP teams were much more successful in finding and fixing the defects, and in the end of the project they delivered systems with a lower number of defects per use case. This indicates that pair programmers write code with fewer defects, but this benefit may be lost unless careful system testing is performed.

The PP teams had slightly better design quality based on the method size and complexity metrics. However, the reason may be the potential correlation between software size and these code metrics. The PP teams delivered systems with less functionality, and therefore these metrics may show better values for them.

In the PP teams developers generally had high involvement in more packages than in the SP teams. Probably related to this, there were generally more developers (1.8 vs. 1.4) in the PP teams with good understanding of each package, and each developer understood more packages (4.5 vs. 3.4) well. This indicates better knowledge transfer within the PP teams.

Even though about half of the developers in the PP teams enjoyed solo programming more than pair programming (and about half vice versa) most developers still liked working in the PP teams. Thus developers' feelings toward pair programming should not hinder its deployment. However, the team which was removed from the analysis abandoned the use of pair programming against the rules of the experiment, which suggests that they were strongly against pair programming after a couple of weeks of experimenting with it. The reason may be that pair programming really was an unsuitable practice for this team, but another reason, backed up by the fact that their productivity did not improve later, may be that the frustration about their slow progress led them to consider pair programming as a new practice as the main cause for their problems.

It seems that the use of pair programming leads to fewer defects in code after coding and better knowledge transfer within the development team without requiring additional effort if the learning time can be avoided. These benefits are likely to decrease the further development costs of the system and increase an organization's productivity due to improved competence of the developers.

In the future we will package the materials of the experiment and publish them on the web in order to provide help for those interested in replicating the experiment. With only minor modifications the package can be used for studying other development practices, such as test driven development. We are planning improving the experiment and replicating it with a greater number of students.

Our research on pair programming will continue by performing case studies at companies using pair programming. In companies it is very challenging to arrange even quasi-experiments, but on the other hand case studies can give valuable qualitative information on, e.g., how pair programming should be practiced in industry.

References

[1] L. Williams and R. Kessler, Pair Programming Illuminated, Addison-Wesley, Boston, 2002.

[2] L. Williams, "The Collaborative Software Process", Ph.D. dissertation, University of Utah, 2000.

[3] J. Nosek, "The Case for Collaborative Programming", *Communications of the ACM*, 41(3), 1998, pp. 105-108.

[4] A. Cockburn and L. Williams, "The Costs and Benefits of Pair Programming", In *Extreme programming examined*, Addison-Wesley, Boston, 2001, pp. 223-243.

[5] L. Williams and H. Erdogmus, "On the Economic Feasibility of Pair Programming", In *International Workshop on Economics-Driven Software Engineering*, 2002.

[6] F. Padberg and M.M. Müller, "Analyzing the Cost and Benefit of Pair Programming", In *Proceedings of the Software Metrics Symposium*, 2003, pp. 166-177.

[7] F. Padberg and M.M. Müller, "Modeling the Impact of a Learning Phase on the Business Value of a Pair Programming Project", In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, 2004, pp. 142-149.

[8] H. Gallis, E. Arisholm and T. Dybå, "An Initial Framework for Research on Pair Programming", In *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, 2003, pp. 132-142.

[9] E. Arisholm, "Design of a controlled experiment on pair programming", *ISERN 2002 Annual Meeting*, [online] 2002, http://fc-md.umd.edu/projects/Agile/ISERN/Arisholm.ppt (Accessed: 21 April 2005).

[10] J. Nawrocki and A. Wojciechowski, "Experimental Evaluation of Pair Programming", In *Proceedings of the 12th European Software Control and Metrics Conference*, 2001, pp. 269-276.

[11] M. Rostaher and M. Hericko, "Tracking Test First Pair Programming – An Experiment", In *Extreme Programming and Agile Methods - XP/Agile Universe 2002,* 2002, pp. 174-184.

[12] J. Wilson, N. Hoskin and J. Nosek, "The Benefits of Collaboration for Student Programmers", In *Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education*, 1993, pp. 160-164.

[13] M. Ciolkowski and M. Schlemmer, "Experiences with a Case Study on Pair Programming", In *Workshop on Empirical Studies in Software Engineering*, 2002.

[14] P. Baheti, E. Gehringer and D. Stotts, "Exploring the Efficacy of Distributed Pair Programming", In *Extreme Programming and Agile Methods - XP/Agile Universe 2002*, 2002, pp. 208-220.

[15] M.M. Müller, "Are Reviews an Alternative to Pair Programming?", *Empirical Software Engineering*, 9(4), 2004, pp. 335-351.

[16] C. McDowell, H. Bullock, J. Fernald and L. Werner, "The Effects of Pair-Programming on Performance in an Introductory Programming Course", *ACM SIGCSE Bulletin*, 34(1), 2002, pp. 38-42.

[17] C. McDowell, B. Hanks and L. Werner, "Experimenting with Pair Programming in the Classroom", In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, 2003, pp. 60-64.

[18] C. McDowell, L. Werner, H. Bullock and J. Fernald, "The Impact of Pair Programming on Student Performance, Perception, and Persistence", In *Proceedings of the 25th International Conference on Software Engineering*, 2003, pp. 602-607.

[19] N. Nagappan, L. Williams, M. Ferzli, E. Wiebe, K. Yang, C. Miller and S. Balik, "Improving the CS1 Experience with Pair Programming", *ACM SIGCSE Bulletin*, 35(1), 2003, pp. 359-362.

[20] B. Hanks, C. McDowell, D. Draper and M. Krnjajic, "Program quality with pair programming in CS1", In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education,* 2004, pp. 176-180.

[21] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering", *IEEE Transactions on Software Engineering*, 28(8), 2002, pp. 721-734.

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

[23] T. McCabe, "A Software Complexity Measure", *IEEE Transactions on Software Engineering,* 2(4), 1976, pp. 308-320.

[24] J. Aiello and E. Douthitt, "Social Facilitation from Triplett to Electronic Performance Monitoring", *Group Dynamics: Theory, Research and Practice*, 5(3), 2001, pp. 163-180.

[25] S. Siegel, Nonparametric statistics for the behavioral sciences, McGraw-Hill Kogakusha, 1956.

[26] A.J. Dick and B. Zarnett, "Paired Programming & Personality Traits", In *Proceedings of Extreme Programming and Agile Processes in Software Engineering*, 2002, pp. 82-85.