Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 78 (2005) 166–179
The Journal of Systems and Software
www.elsevier.com/locate/jss

Two controlled experiments concerning the comparison of pair programming to peer review

Matthias M. Müller *

Department of Computer Science, Fakultät für Informatik, Universität Karlsruhe, Am Fasanengarten 5, 76131 Karlsruhe, Germany

Received 3 June 2004; received in revised form 23 December 2004; accepted 24 December 2004
Available online 22 January 2005

Abstract

This paper reports on two controlled experiments comparing pair programming with single developers who are assisted by an additional anonymous peer code review phase. The experiments were conducted in the summer semester 2002 and 2003 at the University of Karlsruhe with 38 computer science students. Instead of comparing pair programming to solo programming this study aims at finding a technique by which a single developer produces similar program quality as programmer pairs do but with moderate cost.

The study has one major finding concerning the cost of the two development methods. Single developers are as costly as programmer pairs, if both programmer pairs and single developers with an additional review phase are forced to produce programs of similar level of correctness. In conclusion, programmer pairs and single developers become interchangeable in terms of development cost. As this paper reports on the results of small development tasks the comparison could not take into account long time benefits of either technique.

Keywords: Pair programming; Peer reviews; Empirical software engineering; Controlled experiment

1. Introduction

Pair programming has become widely accepted as an alternative to solo programming in the last years. When working in pairs, individual developers learn from their partners, share ideas, and when they pair off they find solutions which none of them would have found alone. A team of developer pairs shares responsibilities, denies specialization and thus, reduces the risk of a project failure caused by personal change-over. And last but not least, the pleasure of those who like pair programming should not be underestimated. But the advantages of pair programming are bought at the expense of nearly doubled personnel cost.

So far, empirical studies only compared single developers to two person inspections methods and pairs of programmers to single developers. For example Bisant and Lyle (1989) showed that a two person inspection method improves individual programming process and Sauer et al. (2000) state that during inspections expert pairs perform as well as any larger inspection group. In the last years, a lot of studies compared pair programming to solo programming. For a discussion of these studies, see Section 2. Both techniques, two person inspections and pair programming, lead to higher quality programs as compared to programs written by single developers. However, current empirical software engineering research lacks a comparison of the two techniques.

This study is aimed as a step towards filling the gap between pair programming and two person inspections. It is a first step because it compares pair programming

* Tel.: +49 721 608 7333; fax: +49 721 608 7343.
E-mail address: muellerm@ipd.uka.de

0164-1212/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2004.12.019

to an informal code review process as opposed to the formal process of code inspections. The following techniques were compared:

Pair programming: Two persons sit in front of a workstation and work together on the same task. Both developers share ideas in order to obtain a solution to the actual programming task.
Review: A solo developer implements a solution to a problem and fixes all compilation errors. Then, he hands in his program for anonymous code review. After the review, he receives the program source together with a short description of the errors and finally starts testing.

The experiments took place in the summer semester 2002 and 2003 at the University of Karlsruhe. They are referred to as *Exp02* and *Exp03*, respectively. Participants were 38 computer science students.

The result of the experiments is as follows. Programmer pairs are as cheap as single developers if both developer pairs and single programmers are forced to produce programs of similar correctness.

Müller (2003) presents first results of Exp02. This paper extends the results by a thorough analysis of the combined data sets of Exp02 and Exp03. Throughout the paper, the group of the single developers will be called the *review group* as well.

2. Related work

Three studies compared pair programming to solo programming.

Williams et al. (2000) studied pair programming with 41 undergraduate students. The control group consisted of 13 students who performed the work individually. The pair programming group was made up of 28 students. The study lasted 6 weeks. During that time frame, all individuals and pairs completed four assignments. Concerning program correctness, the programs of the pairs passed more of the automated post development tests than the programs of the individuals did. This difference was significant with $p < 0.01$. The data sample of the pairs also showed a smaller variability than the data sample of the individuals. The evaluation of the development cost showed that, after an initial adjustment period in which the pairs spent about 60% more programmer hours on the completion of the tasks, the working overhead of the pairs dropped down to 15%.

Nosek (1998) studied 15 professional programmers, 5 individuals and 15 pairs, on a database consistency check. None of the subjects had worked on this kind of problem before. The time needed to complete the task was limited to 45 min. All pairs outperformed the individuals. Although the average time for completion was more than 12 min longer for individuals (41%), the difference was not statistically significant on the 5% level.

Nawrocki and Wojciechowski (2001) studied 21 computer science students on the first four assignments of the *Personal Software Process* programming course (Humphrey, 1995). The students were divided into three groups: the first group applied the PSP-baseline process (time and defect logging), the second group used XP tailored to single programmers, and the third group used pair programming. Overall, the pair programming group was not faster than the other two groups. This result stands in contrast to the Williams et al. and the Nosek studies. The variability within the pair programming group was smaller than within the other two groups. This observation led to the conclusion that the pair programming process is more predictable.

Tomayko (2002) compared the defect rates of programs of teams following Extreme Programming (XP) (Beck, 1999) and the Team Software Process (TSP) (Humphrey, 1999). The XP team achieved a defect rate more than half as low as the defect rate of the TSP team (9.6–19.7 defects per thousand lines of code). The study compared development processes, thus, the reported effect cannot be subscribed to a single technique alone, for example pair programming or inspections.

Williams and Kessler (2000) and McDowell et al. (2002) investigated the advantages of pair programming for educational purposes. Other studies evaluated the potential costs and benefits of pair programming (Cockburn and Williams, 2000; Müller and Padberg, 2002; Padberg and Müller, 2003).

3. Methodological issue

Although Williams et al. (2000), Nosek (1998), and Nawrocki and Wojciechowski (2001) conducted their studies to evaluate the advantage of pair programming over solo programming, the results of the first (and maybe also of the third [1]) study are inconclusive to some extent. The results depend not only on the development methods but also on programs' *level of correctness* because programs with different number of failures were compared. The programs' level of correctness differ because the assignments were considered completed as soon as the participants (pairs and individuals) claimed so. Thus, the produced programs naturally differ by the time needed for completion *and* by the correctness level because of the subjects' different attitude on when it is beneficial to stop testing and when not. It is methodologically questionable to compare these program versions. The only way out of this dilemma is to keep

[1] The experimental settings are described too vague by Nawrocki and Wojciechowski (2001) as to get any reliable information on this topic.

either of the two variables fixed: *time to completion* or *code correctness*. For example, a valid comparison of two development methods should only consider program versions that were developed within a fixed period of time as it was done in the Nosek study, or those that achieve a certain level of correctness and to force rework if a program is below this level.

As the first approach (limiting the time) faces the problem of incomplete programs, this study implements the second approach by issuing an additional quality assurance phase at the end of the implementation process. This quality assurance phase ensures a comparable correctness of subjects' programs. Thus, the measured programming effort solely depends on the different implementation methods. However, in order to be consistent with previous studies this paper also presents the confounded results before the quality assurance phase.

4. The study

The two experiments had a counterbalanced design and were held at the University of Karlsruhe during the summer lectures 2002 and 2003, respectively. The experiments were part of an Extreme Programming (XP) course (Müller et al., 2004) which took place in the summer semester. The course consisted of four short sessions (introducing pair programming, test-first, refactoring, and the planning game) and a whole week of project work. The experiments were performed from May to June between the introductory sessions and the project week. Java was the programming language for both the experiments and the lab course.

4.1. Subjects

Students subscribed voluntarily to the course. They knew from the course announcement that they had to take part in an experiment in order to get their course certification. All subjects were computer science graduate students who were on average in their fourth year of study, see the leftmost plot in Fig. 1. The outlier on the lower part of the scale originates from an exchange student from Norway who reported only the time of study he spent in Karlsruhe. The outlier on the upper part of the scale represents a student who is also a developer in his own small software company.

The other three plots of Fig. 1 show subjects' overall programming experience in years, the Java programming experience in years, and the Java Programming experience in lines of code. An outlier at 270 000 lines of code Java programming experience is omitted for presentation purposes in the rightmost plot. The outlier refers to the previously mentioned student with his own software company. Although, his self reported numbers suggest high programming skills, he and his pair programming partner did not perform best in their group which led to the decision to not remove the data point from the analysis. However, his data as single programmer was removed because of the internal threat which is discussed in Section 4.4.1. As it is typical for student groups, subject's programming experience is wide spread: some participants were almost beginners while others have a programming experience similar to professional software developers.

Prior to the experiment, subjects were introduced to pair programming and reviews. Each course took about 1.5 h. Pair programming was taught by XP professionals with who the lab course was conducted. Reviews were presented by the author. The subjects were forced to use only these two development methods. The other techniques of XP were not part of this study.

4.2. Tasks

Due to the counterbalanced design of the experiments, the subjects solved two different tasks:

Polynomial: Find the zero positions of an arbitrary polynomial of third degree. The subjects had to implement the method `findZeroPosition` of a given class `Polynomial`.
Shuffle-Puzzle: Find the solution of a given shuffle-puzzle within a given number of moves and list the moves, if a solution exists. The subjects had to add a method `findMoves` to the basic class `ShufflePuzzle`.

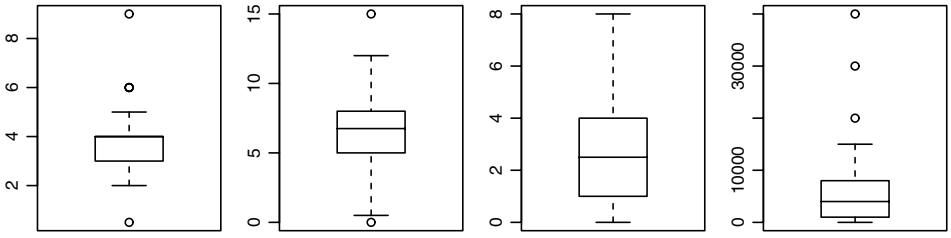

Fig. 1. Students' year of study, overall programming experience in years, Java programming experience in years, and Java programming experience in lines of code.

The classes `Polynomial` and `ShufflePuzzle` already contained constructors and methods for I/O to facilitate implementation and final testing.

The description of the task `Polynomial` contained a hint for a possible numeric solution to the problem. However, the students were not forced to use a special algorithm to solve the problem; they could use any algorithm which they considered suitable for the problem. As a special difficulty, the task required careful handling of the floating point arithmetic. For most students, solving the task involved implementing the suggested method as well as taking care of special cases. Solving the `Shuffle-Puzzle` task requires backtracking which the students knew how to use from their first computer-sciences courses.

4.3. Review technique

The usage of checklist based code reviews raises two questions:

(1) Why were code reviews used instead of design reviews?
(2) Why were checklists preferred to other reading techniques?

To use code reviews and not design reviews was motivated by the so far unstructured pair programming process which does not contain a separate design phase. We did not want to structure the process of the single developers as well. The single developers should use the development style they were accustomed to. Thus, the only phase constant over all development processes is coding. Consequently, we could not issue a design review because we did not know if each single developer performed a design. Another reason for the code review is the constant review performed by the programmer pairs. The motivation for code reviews is whether the effect of the constant code review done by the pair programming partner can be seen also by a distinct review done by a single person.

Why did we use checklists and not any other reading technique for the review? The usage of checklists stemmed mainly from the lack of empirical studies investigating the impact of a reading technique on the defect finding capabilities of code reviews. A lot of studies compared ad hoc, checklist, and scenario based reading techniques for requirements and design documents but the effect of a reading technique on code reviews is not as investigated. Dunsmore et al. (2003) compared the effect of checklists, the use-case reading technique, and the abstraction-driven technique on object-oriented code reviews. In their study checklists perform in most cases as good as the other techniques. Checklists were also effective in the hands of less able subjects. Laitenberger and DeBaud (1997) and Laitenberger et al. (2001) compared perspective based reading with checklists on C code. Their data suggests an advantage of perspective based reading as opposed to Dunsmore et al. (2003). As the effect of scenario based reading techniques on code reviews is yet not clearly understood, we decided to use checklists for the code reviews.

4.4. Experiment plan

This section presents the review and the pair programming procedures. The procedures consist of a *Reading*, an *Implementation*, and a *Quality Assurance* phase, see Fig. 2.

Reading was identical for both procedures. During Reading, subjects acquaint themselves with the description of the programming task. They were allowed but not forced to make any sort of comments or design notes but subjects were not allowed to code. Reading finished when subjects claimed to be done. Thereafter, subjects entered *Implementation*.

The following sections explain the subsequent experiment phases and discuss why the procedures were chosen as presented.

4.4.1. Review procedure

The review procedure is outlined by the upper half of Fig. 2. Although the subjects worked on their own, they were paired off with their pair programming partner, though, anonymously. The solid and the dashed line in

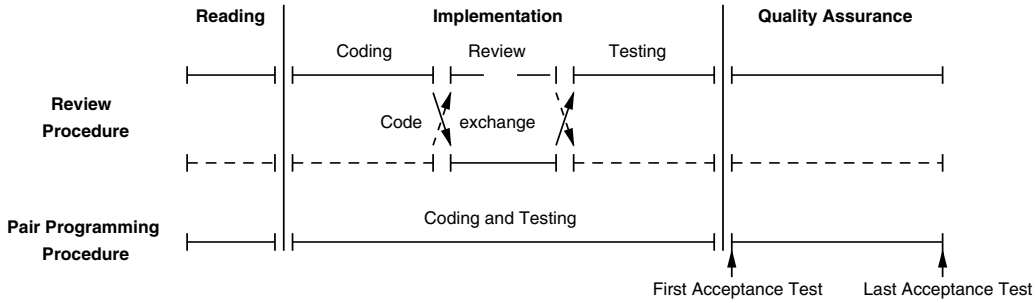

Fig. 2. Procedure for the review and pair programming task.

Fig. 2 represent the work flow of the code for each of the two subjects, respectively.

The *Implementation* phase of the review procedure was split into *Coding*, *Review*, and *Testing*. During *Coding*, subjects had to implement the task until they thought they were done. The subjects could only compile but not execute their programs. This constraint was guaranteed by the experiment environment. After the two subjects finished Coding, they entered *Review*. The program was printed out on paper and handed out to the other subject. As the code review was anonymous, the author did not know who was reviewing his code and the reviewer did not know whose code he was reviewing.

The task of the reviewer was to find errors according to a checklist. Design flaws, violations of any sort of convention, and suggestions for a better solution were of no concern to the review. The review velocity was lower-bounded to at least 100 lines of code per hour. On completion of the review, subjects received the code and entered *Testing*. Now, the subjects were allowed to compile *and* execute their programs appropriately. Subjects left Testing when they claimed to be done.

At this point, they entered the *Quality Assurance* phase where their programs had to pass 95 out of 100 test cases of the acceptance test. If the programs did not reach the required 95% level, subjects received the output of the failed tests and had to fix the errors. The acceptance test and the subsequent rework repeated as long as the program passed less than 95 tests. Otherwise, the subjects concluded their work.

The presented review procedure was applied to Exp02. When following this procedure, three subjects admitted in the post-test questionnaire that they got hints for their own development from the foreign program, see Section 3.7 of Müller (2003). To remove this threat in Exp03, only *one* of the two subjects, as opposed to both in Exp02, prepared the task while the other subject performed the review. The subject who was preparing the task was randomly selected. To obtain the modified experiment plan of Exp03 remove the dashed line in the review part of Fig. 2.

4.4.2. Discussion of review procedure

The review procedure did not allow the subjects to execute the code *before* review took place. This might not seem intuitive because, normally, the code is executed and tested very carefully before it is reviewed. However, as motivated by Humphrey (1995, pp. 267–268), the author chose not to permit execution of the code before the review because of the following two reasons. First, reviewing code that has not been executed changes the attitude of the reviewers. They know that the program was not executed and tested and thus, it is worth a review. Second, the author of the program might not want the reviewer to find any errors. Hence, he develops his program more carefully. Actually, one subject reported that he tried to developed his program more understandable because of the subsequent review.

The reviews were anonymous because the reviewer's attitude towards the program should not have been influenced by his knowledge of who was the author of the program. However, the drawback of an anonymous review is that in the case of problems concerning the review the author cannot ask the reviewer for clarification. This problem is a threat to the internal validity of the experiments which is discussed in more detail in Section 7.1.

The lower bound for the review time of 100 lines of code per hour is based on figures shown by Gilb and Graham (1993, p. 154). They suggest a review speed of one page (non-commentary, 600 words) per hour. Their checking rate is due to cross checking against several documents: rule sets, checklists, role checklists, and source documents. As both tasks and their specifications are rather small, doubling the review speed seemed reasonable.

The main incentive of the quality assurance phase was to ensure high and similar level of correctness of developed programs, such that the programming effort depends only on one independent variable: the method used for implementation (review or pair programming). The individual attitude to testing or program correctness which differs from pair to pair and developer to developer is factored out. Thus, the comparison of both methods bases solely on the effort imposed on a development task and not on subjective decisions. However, the exit criterion of the quality assurance phase still leaves some room for variation in correctness. But this variation is expected to be too small to be statistically detectable.

The acceptance test used during the quality assurance phase consisted of 100 test cases. As 100 tests are rather few, the aim of the test was to provide almost instant feedback on the correctness of the program. If a larger test had been used instead, a subject would have been waiting for hours for the test results in the worst case. Thus, the delay incurred by a large test would have been an unacceptable disruption of subjects' work flow.

The output of the acceptance test was logged by the experiment environment. Although the results from the acceptance tests were available for analysis, the data was not used for evaluation purposes. The results of a separate final test, the so called *large test*, were used instead. The *large test* was executed after the experiment. It is described in Section 4.7.1.

4.4.3. Pair programming procedure

The pair programming procedure was straight forward. During *Implementation*, the pairs were allowed to compile and execute their programs from the very beginning. They worked on the programs until they

claimed to be done. Then, they entered the *Quality Assurance* phase which also iterated between executing the acceptance test and rework. And again, the exit criterion was to pass at least 95 out of the 100 test cases of the acceptance test.

The pair programming procedure was the same in Exp02 and Exp03.

4.4.4. Realization of experiments

The pair programming procedure could be done in one session, while the review procedure involved at least two different sessions: one for coding and another one for review, testing, and quality assurance. For each session and each task, both the pairs and single programmers made an appointment with the experimenter. If the task was not be finished in the first run, a subsequent appointment had to be made.

4.5. Issues on replication

Replication of the experiment was necessary because of two weaknesses of Exp02. First, replication increased the size of the data samples and therefore the possibility of revealing an effect. And second, the replication removes the previous mentioned threat that subjects could obtain hints from the review. According to the terminology on replications introduced by Basili et al. (1999, p. 469), the second experiment is a *replication that does not vary any research hypothesis*. To be more precise, Exp03 is no *strict* replication, but rather a *replication that varies the manner in which the experiment is run*.

4.6. Group selection and size

Due to the counterbalanced design of the experiments, subjects had to solve two task, each with another method. Table 1 shows group characteristics of the four groups. Subjects in *Group 1* solved the first task `Shuffle-Puzzle` with pair programming and the second task `Polynomial` with solo programming. The next three columns of Table 1 list the overall language independent programming experience in years as well as the Java programming experience in years and in lines of code.

Division of subjects into groups was done according to their overall programming experience measured in lines of code, not shown in Table 1. The measure was collected in the pre-test questionnaire where subjects had to mark one of the four categories: *less than* 3000, *less than* 10 000, *less than* 40 000, or *more than* 40 000 *lines of code* programming experience. Subjects' overall programming experience served as blocking factor for the division into experiment groups as we first arranged equally skilled subjects into the same group. The members of each group where then randomly assigned to the four experiment groups which have already been shown in Table 1.

To establish the partner relationship for pair programming within each group, the most skilled subject had to pair off with the lowest skilled subject, the second best skilled subject with the second lowest skilled subject, and so on. The Java specific experience could have been used as blocking level as well, but in the project weeks which took place after the experiments, it turned out that the overall experience represented the individual skill level better than the Java specific experience.

The aim of the division was to even out pairs' general experience level. The general experience level of a pair was obtained by calculating the mean of the individual experience levels of the two members of the pair. The individual experience level ranged between 1 for *less than* 3000 and 4 for *more than* 40 000 *lines of code*. Fig. 3 shows for each group the distribution of the pairs' mean experience level.

Except for group 1 where each pair had an average experience level of 3 (see white bar in the column named 3), groups show some variability in pairs' mean experience level. For group 3 the variability is balanced at 3 while for group 4 and for group 2 the mean experience level is shifted towards the lower and the upper directions of the scale, respectively.

Most of the subjects had no experience in the techniques under study: 7 subjects admitted some experience in reviews; 7 subjects reported to have some experience with pair programming; and one subject had tried both reviews and pair programming prior to the experiment to some extend.

Table 1
Task order, mean programming experience, and available data points for each group (PP = pair programming, Re = review, Shu = shuffle-puzzle, Pol = polynomial)

Group	1. Task	2. Task	Prog.	Java Exp.		Size		Data pts.	
			Years	Years	LOC	Exp02	Exp03	PP	Rev
1	PP.Shu	Re.Pol	7.6	3.4	29 450	6	4	5	6
2	PP.Pol	Re.Shu	6.3	2.1	3438	4	4	4	4
3	Re.Shu	PP.Pol	6.3	2.9	7550	6	4	5	7
4	Re.Pol	PP.Shu	6.5	2.5	9020	4	6	5	6
Overall			6.7	2.7	12 834	20	18	19	23

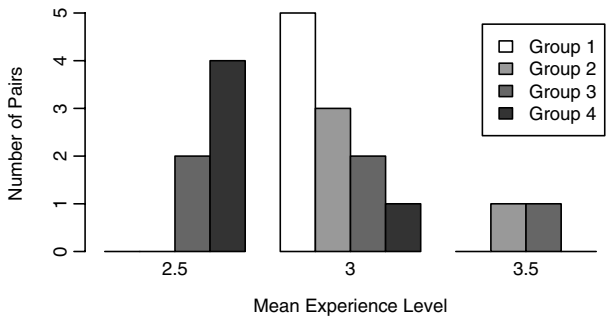

Fig. 3. Group wise distribution of pairs' mean programming experience.

The last four columns of Table 1 list the group sizes and the number of data points. Due to the experiment design, each subject pair is supposed to contribute one pair programming data point as well as two review data points in the case of Exp02 and one review data point in the case of Exp03. However, three subjects did not finish the review task in Exp02. They belonged to the groups 2, 3, and 4, respectively. Additionally, two review data points of group 1 and one review data point of group 2 had to be deleted to avoid the previous discussed internal threat of Exp02. In Exp03 all data points could be used for analysis. Overall, there were 38 participants yielding 19 pair programming and 23 review datapoints.

4.7. Data

4.7.1. Correctness

For each task two tests were created: the *large* test and the *acceptance* test. The *large* test consisted of 700 000 test cases for the polynomial task and of 15 000 test cases for the shuffle-puzzle task.

The test cases for the large polynomial test were constructed by calculating the corresponding coefficients of a polynomial of third degree from randomly generated zero positions. The shuffle-puzzles of the large test were generated by shuffling a solved shuffle-puzzle by a number of moves. For each shuffle-puzzle test, the number of moves was generated randomly with an upper bound of allowed moves.

With the large tests we tried to simulate a usage scenario which lasts over a long time. For the polynomial task, we divided the polynomials into categories of one, two, and three zero positions. For each category, we created 100 000 test cases for integer and for floating point zero positions. A final fourth category contained all three types of polynomials with integer and floating point zero positions mixed. The usage scenario for the shuffle-puzzle test contained puzzles of an arbitrary size between 2×2 and 6×6. The values for the x- and y-dimensions could differ. We then shuffled a puzzle according to a randomly generated number of moves n. The number n ranged between 3 and 14 moves. The test itself contained a description of the puzzle, a number n' which indicated how deep the program should search for a solution, and whether a solution can be found within n' moves. We had three test categories: one where $n' = n$, one where $n' = n - 1$, and one where $n' = n + 1$. Each category contained 5000 test cases.

Each *acceptance* test consisted of 100 test cases which have been randomly selected from the appropriate large test. The test cases were selected once before the experiment and never changed afterwards. The acceptance tests did not only test implementations but also archived submitted Java files. In detail, the acceptance test stored the issued program version, executed the test cases, logged the results, and reported the results to the subject.

The correctness (Correct) of a program was measured for two different program versions: the program version after the implementation phase ($\mathsf{Correct}_{\mathsf{Imp}}$) and the final program version ($\mathsf{Correct}_{\mathsf{Task}}$) on exit of the quality assurance phase. The level of correctness of a program is defined as the fraction of the number of passed tests divided by the number of all tests:

$$\mathsf{Correct} = \frac{|\{\mathsf{passed\ tests}\}|}{|\{\mathsf{all\ tests}\}|}$$

4.7.2. Cost

The cost of both methods ($\mathsf{Cost}^{\mathsf{Pair}}, \mathsf{Cost}^{\mathsf{Rev}}$) is compared for implementation (Imp), quality assurance (QA), and the whole task (Task). The cost is measured in man minutes. The cost is calculated using the following measures: time spent for reading the problem description (T_{Read}), the time spent for implementation (T_{Imp}), the review time (T_{Rev}), and the time spent for quality assurance (T_{QA}).

$$\begin{aligned}
\mathsf{Cost}^{\mathsf{Pair}}_{\mathsf{Imp}} &= 2 \cdot (T_{\mathrm{Read}} + T_{\mathrm{Imp}}) \\
\mathsf{Cost}^{\mathsf{Rev}}_{\mathsf{Imp}} &= T_{\mathrm{Read}} + T_{\mathrm{Imp}} + T_{\mathrm{Rev}} \\
\mathsf{Cost}^{\mathsf{Pair}}_{\mathsf{QA}} &= 2 \cdot T_{\mathrm{QA}} \\
\mathsf{Cost}^{\mathsf{Rev}}_{\mathsf{QA}} &= T_{\mathrm{QA}} \\
\mathsf{Cost}^{\mathsf{Pair}}_{\mathsf{Task}} &= \mathsf{Cost}^{\mathsf{Pair}}_{\mathsf{Imp}} + \mathsf{Cost}^{\mathsf{Pair}}_{\mathsf{QA}} \\
\mathsf{Cost}^{\mathsf{Rev}}_{\mathsf{Task}} &= \mathsf{Cost}^{\mathsf{Rev}}_{\mathsf{Imp}} + \mathsf{Cost}^{\mathsf{Rev}}_{\mathsf{QA}}
\end{aligned}$$

The time for quality assurance consists of the rework time only and does not include the execution time of the acceptance test. The cost of the review does not account for any additional waiting time, for example the review synchronization overhead.

The data was gathered with the *pplog-mode*, a major mode for Emacs which supports logging of work-time and interrupts (PSP Resources Page, 2003). Time logging was started and stopped by the experimenter.

4.8. Hypotheses

The experiments were designed to evaluate whether the average cost μ to complete a programming assignment is the same for pairs and for single developers assisted by a separate peer review phase:

$\mathbf{H}_0 : \mu(\text{Pair Programming}) = \mu(\text{Review})$

$\mathbf{H}_{\mathbf{Alt}} : \mu(\text{Pair Programming}) \neq \mu(\text{Review}).$

We consider the cost for the whole task and the implementation phase.

5. Evaluation plan

This section deals with several questions concerning the statistical evaluation of the data:

- Can the data sets of Exp02 and Exp03 be pooled together although the process of the review group was changed in the repetition?
- Complies the data with the requirements of the analysis of variance?
- What is the expected power of the statistical tests?
- To what extend can we trust the results?

5.1. Pooling of data sets

Data sets of both experiments are characterized by the combinations of the different levels of the *treatment* (pair programming and review) and *block* (shuffle-puzzle and polynomial). The combinations are shown by the two 2×2 matrices on the left hand side of Fig. 4. To combine the appropriate data samples of both data sets, data samples must be taken from the same population. In our case, the review data samples originate from two different processes. Although the process modifications are rather small and possible differences in the data samples might depend on other not controllable factors such as the subjects' educational background we performed for each data set a two-sided Mann–Whitney test on the respective data samples. The Mann–Whitney test showed no difference on the 5% level for the $\mathsf{Cost}_{\mathsf{Task}}$, $\mathsf{Cost}_{\mathsf{Imp}}$, $\mathsf{Correct}_{\mathsf{Task}}$, and $\mathsf{Correct}_{\mathsf{Imp}}$ data sets. The data samples of these data sets were pooled to form one data set for further evaluation.

The data samples of the $\mathsf{Cost}_{\mathsf{QA}}$ data set were not pooled because the Re.Pol data samples differ on a 4.6% level. As a consequence, the analysis of $\mathsf{Cost}_{\mathsf{QA}}$ has to be done separately for each experiment.

Analysis of variance requires normally distributed data sets with homogeneous variances. The Shapiro–Wilk test was used to test for normality. Homogeneity was tested with the Bartlett-Test. Except $\mathsf{Cost}_{\mathsf{Task}}$, every data set had a data sample where the normality test showed a difference on the 10% level. As the Bartlett test shows a difference in variances of the $\mathsf{Cost}_{\mathsf{Task}}$ data set on a 20% level, analysis of variance is only applied on the $\mathsf{Cost}_{\mathsf{Task}}$ data set. The remaining three data sets $\mathsf{Cost}_{\mathsf{Imp}}$, $\mathsf{Correct}_{\mathsf{Task}}$, and $\mathsf{Correct}_{\mathsf{Imp}}$ are evaluated using the Mann–Whitney test.

Analysis of variance was performed with the treatment level pair programming and review and the blocking level shuffle-puzzle and polynomial. All other statistical evaluations where done in an experiment wise manner. The Mann–Whitney test was used in these cases.

Significance was set to 5% for all tests.

5.2. Power analysis

The power of the analysis of variance is calculated according to Cohen (1988, p. 364). The power for a large effect $f = 0.4$ (medium effect $f = 0.25$) is 70 (34) percent ($n' = 20$, $u = 1$). From the alternative hypotheses perspective, we detect in seven out of ten replications of both experiments a difference between the review and the pair programming group, if there is any. The combination of both experiments has a good chance of revealing an effect, though, this effect has to be large.

The power of the Mann–Whitney test is calculated using the power of the *t*-test. The power of the two-sided *t*-test is 71%. The power of the *t*-test was calculated with R (Ihaka and Gentleman, 1996) using two samples, the harmonic mean of both group sizes $n = 20.8$, an effect size of 0.8, and a significance level of $\alpha = 0.05$. If we

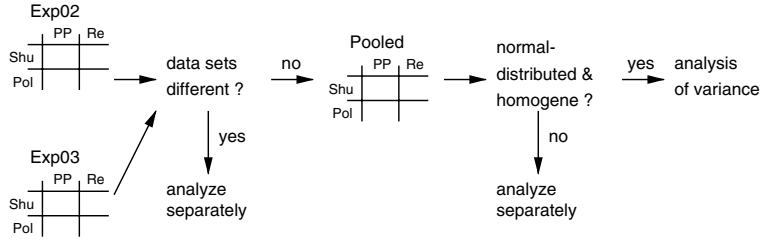

Fig. 4. Overview of evaluation process.

had chosen the interesting effect size to be equal to 0.5, the power of the *t*-test would have been 35%. The power of the Mann–Whitney test is in the worst case 84.6% the power of the *t*-test, see Hollander and Wolfe (1999, p. 139). In any case, if tests do not reveal a significant difference between data sets then there are two possibilities: either the effect is too small to be seen and therefore, our sample size is too small, or there is no effect to detect at all.

Cohen (1988) suggests that a study should have at least a power of 80% in order to have a real chance to reveal an effect. According to his suggestion the power of 71% for the combination of both experiments is rather low. But as the course held in 2002 already exceeded the maximum number of 18 students by 2 we could not afford additional students.

5.3. Trustworthiness of the results

Evaluation plan of the experiments outlined in Section 5.1 is not intuitive and in most cases, the non-parametric Mann–Whitney test is used instead of the parametric *t*-test or analysis of variance. Thus, the statistical evaluation is rather conservative. Conservative means that the Mann–Whitney test may suggest to not reject a hypotheses although a parametric test would indicate a difference on a 5% level. This conservative evaluation has two consequences. First, a difference on a 5% level is not caused by a false use of a test. And second, if a test suggests to not reject a hypotheses, there is still the possibility of difference as the rather low power of the tests suggests.

6. Results

Results are shown with box plots. The boxes within a plot contain 50% of the data points. The lower (upper) border of the box marks the 25% (75%) quantile. The lower (upper) *t*-bar marks the most extreme data point which is no more than 1.5 times the length of the box away from the lower (upper) side of the box. Outliers from the above scheme are visualized with circles. The median is marked with a thin line. The *M* associated with the dashes on each side marks the mean value within a range of one standard deviation on each side. In tables, the abbreviations $\bar{x}$, s, and $\tilde{x}$ are used for the mean, the standard deviation, and the median of the data samples, respectively.

6.1. Correctness

The aim of the quality assurance phase was to ensure high and similar program correctness. This section investigates whether this aim was achieved.

6.1.1. Correctness of final programs

The first three data columns of Table 2 and Fig. 5 present the correctness of the final programs measured with the large test.

First of all, all groups achieve a reasonable level of correctness. The mean values range between 93.3% for PP.Pol and 98.3% for Re.Shu with a standard deviation varying between 0.4 for PP.Shu and 3.2 for PP.Pol. The reason that data points fall below the 95% exit criteria of the quality assurance phase is that the results of the large test and not the acceptance test are shown. The differences in the PP and Re data sets shown in Table 2 are statistical negligible.

6.1.2. Correctness of programs after implementation

The three rightmost columns in Table 2 and Fig. 6 show program correctness after the implementation phase.

The achieved level of correctness is not as high as for the final programs. The mean values for the groups vary between 36.1 for Re.Shu and 59.6 for PP.Shu. The variability in the data sets is higher as well. The standard deviation ranges between 29.0 for Re.Pol and 35.6 for Re.Shu. The average level of correctness for programs developed by pairs is 29% higher than for programs developed by single programmers (54.8% versus 42.4%). Although there is a visible difference in location in the PP and Re data sets, the Mann–Whitney test does not indicate a difference on the 10% level.

6.1.3. Summary correctness

We note two results from the analysis of the correctness of the programs:

(1) The quality assurance phase served its purpose: programs of both groups have high and similar level of correctness.
(2) The pairs developed programs with an observable higher level of correctness.

The second result is statistical insignificant, though, it complies to the results reported by Williams et al. (2000).

Table 2
Correctness of programs

Group	$\text{Correct}_{\text{Task}}$			$\text{Correct}_{\text{Imp}}$		
	$\bar{x}$	s	$\tilde{x}$	$\bar{x}$	s	$\tilde{x}$
PP.Pol	93.3	3.2	93.3	49.4	35.0	50.5
PP.Shu	99.7	0.4	99.9	59.6	30.9	53.1
PP	**96.7**	**3.9**	**99.2**	**54.8**	**32.4**	**50.5**
Re.Pol	96.5	2.1	96.4	48.7	29.0	37.3
Re.Shu	98.3	3.1	99.7	36.1	35.6	29.6
Re	**97.3**	**2.7**	**98.0**	**42.4**	**32.3**	**32.8**

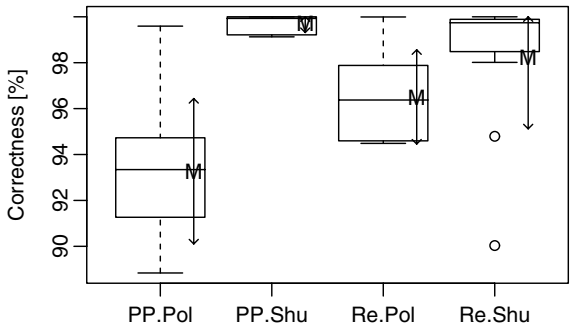

Fig. 5. Correctness level for task $\mathsf{Correct}_{\mathsf{Task}}$.

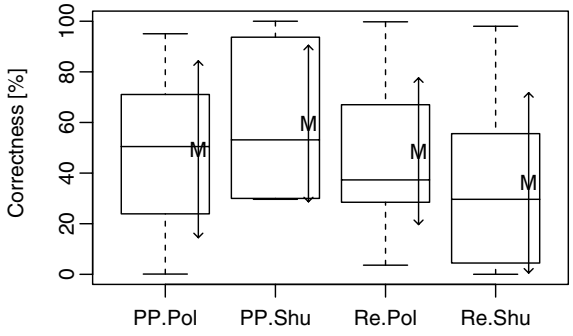

Fig. 6. Correctness level after implementation $\mathsf{Correct}_{\mathsf{Imp}}$.

6.2. Cost

6.2.1. Cost of whole task

The three left most data columns of Table 3 and Fig. 7 show the cost for the whole task.

The pair programming and review data sets lead to two observations. First, the average cost of the two pair programming groups ranges between the cost of the review groups. And second, either group solved the `Polynomial` task quicker than the `Shuffle-Puzzle` task. Combining the two pair and the two review data sets, pairs (mean of 546.6) are on average 7% more expensive than reviews (mean of 511.8). However, analysis of variance does not support this difference ($p = 0.614$).

Table 3
Cost for whole task ($\mathsf{Cost}_{\mathsf{Task}}$) and implementation phase ($\mathsf{Cost}_{\mathsf{Imp}}$)

Group	$\mathsf{Cost}_{\mathsf{Task}}$			$\mathsf{Cost}_{\mathsf{Imp}}$		
	$\bar{x}$	s	$\tilde{x}$	$\bar{x}$	s	$\tilde{x}$
PP.Pol	521.1	164.5	512	434.9	166.7	420
PP.Shu	569.6	213.8	516	444.0	125.8	395
PP	**546.6**	**188.4**	**512**	**439.7**	**142.4**	**400**
Re.Pol	427.9	121.7	413	362.2	111.5	321
Re.Shu	603.4	327.7	529	418.7	175.1	446
Re	**511.8**	**253.5**	**458**	**389.2**	**144.9**	**330**

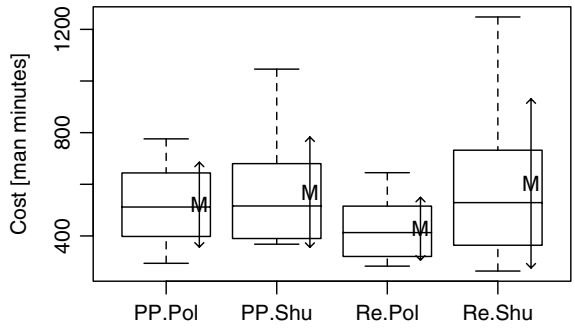

Fig. 7. Cost for whole task ($\mathsf{Cost}_{\mathsf{Task}}$).

6.2.2. Cost for implementation

The three right most data columns of Table 3 and Fig. 8 show the cost of the implementation phase.

On average, the review groups are cheaper than the pair programming groups and again, the `Polynomial` task was solved more quickly by either group. When the pair data sets and the review data sets are combined, the pairs (mean value of 439.7) are on average 13% more expensive than the single developers (mean value of 389.2). However, a p-value of 0.225 of the Mann–Whitney test does not support this difference.

6.2.3. Cost for quality assurance

Table 4 and Fig. 9 show the cost of the quality assurance phase for both experiments. The result of Exp03 is surprising: the review group is cheaper than the pair programming group, although the single developers had to make up for a lower level of correctness after the implementation phase. The data of Exp02 does not show this effect. In this case, the pairs are cheaper than the single developers. However, both differences are insignificant. Further studies are sought to study whether pairs or single developers are more productive during quality assurance.

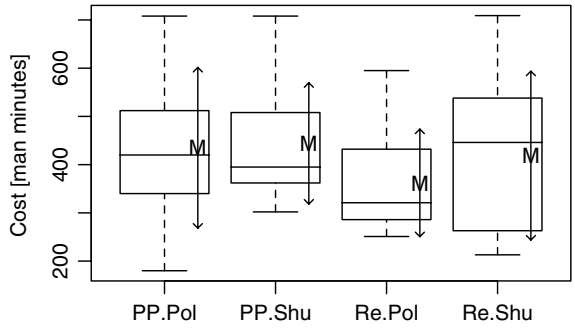

Fig. 8. Cost for implementation ($\mathsf{Cost}_{\mathsf{Imp}}$).

Table 4
Cost for quality assurance ($Cost_{QA}$)

Group	$Cost_{QA}$		
	$\bar{x}$	s	$\tilde{x}$
PP.Exp02	81.0	98.9	31.0
Re.Exp02	122.4	152.7	73.5
PP.Exp03	135.8	112.7	82.0
Re.Exp03	122.9	152.9	43.0

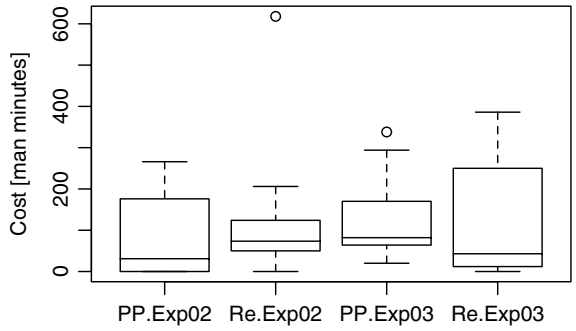

Fig. 9. Cost for quality assurance ($Cost_{QA}$).

6.2.4. Summary cost

Results from the cost analysis:

(1) If a similar level of program correctness is enforced, pairs cost as much as single developers with reviews do.
(2) If a similar level of program correctness is of no concern, programmer pairs tend to develop programs with a higher level of correctness at slightly increased cost as compared to single developers with reviews.

The 13% cost increase of the pairs stated in the second result seems to comply to the 15% working overhead reported by Williams et al. (2000). However, Williams et al. compared pair programming to solo programming while in this study the individual programmers are assisted by a separate review phase.

6.3. Sequence analysis

Sequence analysis aims at revealing effects that have their roots in the consecutive treatment of the two development tasks. The focus lies on the order of the assignments ignoring the specific method and the specific problem. The sequence analysis of Müller (2003) showed a learning effect from the first to the second assignment. Thus, it is essential to repeat the sequence analysis on the combined data sets of Exp02 and Exp03. All data sets of Table 5 show a general tendency, though, the Mann–Whitney test reveals no statistical significance on any data set.

Concerning correctness, there is a divergent picture. While the correctness level of the final programs decreased from the first to the second assignment (mean value of 97.9 versus 96.1), the correctness of the intermediate programs after implementation increased (mean value of 42.8 versus 54.3). Although this effect could be seen also in Exp02 the reason for this counterintuitive behavior is unclear. Concerning development cost, there is a unique trend to cost reduction from the first to the second assignment. The average cost for the whole task and the implementation phase alone decreases.

In summary, even though the final level of correctness decreased from the first to the second assignment, all other measures indicate that subjects improved their programming skill during the experiment.

Table 5
Differences between first and second assignment averaged over groups and tasks

Data set	1. Assignment			2. Assignment		
	$\bar{x}$	s	$\tilde{x}$	$\bar{x}$	s	$\tilde{x}$
$Correct_{Task}$	97.9	2.5	99.4	96.1	3.9	96.9
$Correct_{Imp}$	42.8	33.6	32.9	54.3	31.1	63.9
$Cost_{Task}$	579.8	230.0	536.0	470.1	208.8	414.0
$Cost_{Imp}$	449.4	135.7	450.5	371.0	145.5	348.0

6.4. Additional results

The reviews lasted on average 63 min or about 12% of the development time of the single developers. The average size of reviewed programs was 124 lines of code and the average review speed was 118 lines of code per hour.

Final program sizes and number of acceptance tests are listed by Table 6 and shown by Fig. 10.

The programmer pairs wrote on average smaller programs and required on average less acceptance tests than the single developers. However, the review group has a smaller median for the program size than the pair programming group. The larger mean of the review group is caused by two programs of the Re.Shu group with 292 and 276 lines of code, see left plot of Fig. 10. The reason for the three outliers in the two review groups for the number of required acceptance tests might be the lower inhibition threshold of the single developers. While in a pair situation the two partners must agree on the next acceptance test, a single programmer has

Table 6
Comparison of number of acceptance tests (AT) and final program size in lines of code

Data set	PP			Re		
	$\bar{x}$	s	$\tilde{x}$	$\bar{x}$	s	$\tilde{x}$
Program size	144	35.8	144	160	48.8	140
Number of ATs	3.3	1.9	3.0	4.9	4.9	3.0

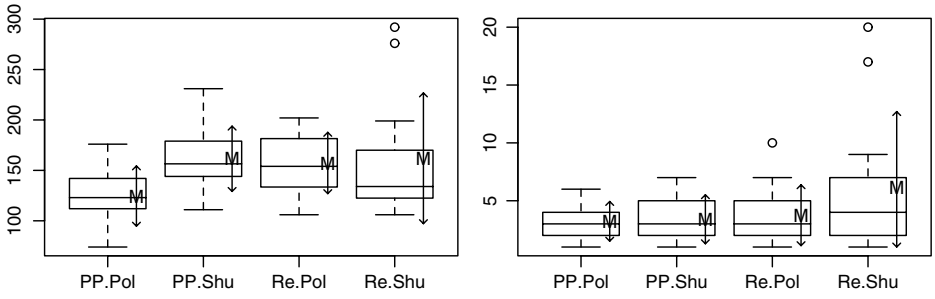

Fig. 10. Program sizes in lines of code (left plot) and number of acceptance tests (right plot).

to salve only his own conscience. Thus, he asks earlier for the next test and has a smaller difference between program versions than the pairs. As a result, the single developer requires more acceptance tests.

In addition to the quantitative data analyzed so far, the remainder of this section evaluates subjects' opinion on pair programming and reviews. The post-test questionnaire asked for subjects' opinion on which method is considered more effective. Subjects had to answer three questions: how effective are reviews; how effective is pair programming; and what is considered more effective reviews or pair programming. Answers were given on a five point ordinary scale. The scale ranged from *not at all* (=1) to *very effective* (=5) for the first two questions. For the last question, the scale ranged from *Review is better* (=1) to *Pair Programming is better* (=5). Fig. 11 shows histograms of the answers. Numbers in the middle of each plot indicate the given number of answers for each category.

The plots on the left and in the middle show that subjects are more committed to efficacy of pair programming than of reviews. The result is supported by the right shift of the answers for the third question, see right plot in Fig. 11. Only one subject believes in the higher efficacy of reviews as compared to pair programming. The tendency of answers in favor of pair programming is quite natural if subjects' preferences are taken into account. As it is pointed out in Section 7.2, all subjects subscribed voluntarily to an extreme programming course where they expect to program in pairs. Thus, their answers might base on their positive attitude towards pair programming and not on its real performance.

7. Threats to validity

7.1. Internal threats

Some perils threat the internal validity of both experiments. First, different persons teaching the lectures on pair programming (professionals) and reviews (the author) could cause differences in skill and motivation among the groups. An alternative approach would have been, that only one person would have taught both courses. However, the author judged the risk of skews in skill and motivation to be higher in the one teacher scenario than in the two teacher scenario. This judgement is due to the fact that subjects could have been biased by the teacher's assumptions if he had taught both topics instead of only one topic. Thus, it was quite reasonable to let each lecture be taught by another teacher.

The second threat concerns the possibility, that a subject did not apply the process it was told to follow. This threat can be ignored because the experimenter attended every programming session and forced the subjects to follow the process.

The anonymous review is another threat to validity because the author of a program could not ask the reviewer for clarification. As a matter of fact, one subject (subject 23) reported that he would have liked to ask the reviewer for clarification because of the meaningless descriptions of the defects. However, further study revealed that subject 23 was the second best subject in the Re.Pol group of Exp03. If the comments in the review of subject's 23 program had been more meaningful, he might have finished earlier. In that case, the

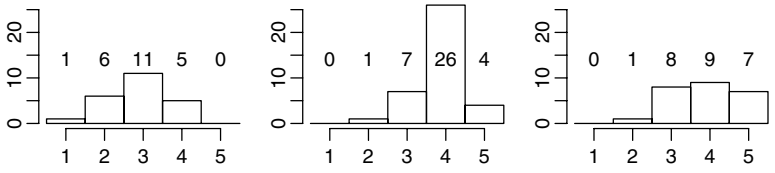

Fig. 11. Answers for ratings (from left to right): how effective are reviews; how effective is pair programming; what is more effective reviews or pair programming.

evaluation would have shown a better result for the review group. Thus, we decided to not remove the data point from the analysis.

7.2. External threats

Several threats may have an impact on the generalizability of the study.

First of all, participants were students who subscribed voluntarily to an extreme programming lab course. Students take part in the course because they want to develop in pairs and probably, they prefer pair programming to reviews, as well. The tendency to favor pair programming over reviews might affect the results in such a way that the performance of pair programming as compared to reviews might be better than it is in practice. To eliminate this threat in further studies a more neutral group of subjects should be used.

Another threat concerns subjects' pair programming and review experience. This threat exists because the subjects did not meet before the pair programming task and because none of the subjects had performed reviews prior to the experiment to that extent that can be expected from a professional. The attempt to even out pairs' general experience level as described in Section 4.6 turned out to be another source of danger. We initially thought that the mean programming skill level of a pair is an indicator for the productivity of a pair. However, studies performed after the experiments show that the programming experience level does not contribute to a pairs productivity as much as we thought, see Müller and Padberg (2004). At our present experience, we should have assigned the pairs randomly.

Students' in most cases rather low programming experience is a threat as well. As a possible consequence of the low programming experience, Section 6.3 showed a learning effect from the first to the second assignment. This learning effect might have been avoided, if the experiments had been conducted *after* the programming lab course. However, this procedure would have meant that the experiments fell into the exams preparation phase which might have posed additional stress on the participants.

Usage of checklists for the code review is a threat as well. Checklists are a rather conservative choice. However, as the impact of other reading techniques on the error finding capabilities of code reviews are not yet understood in detail, usage of checklists seemed a reasonable choice. If other techniques are shown to be better than checklists, the presented experiments should be repeated to account for the improvement of the code reviews.

Other problems originate from the algorithmic structure of the polynomial and shuffle-puzzle task. First, both tasks are more complex but require less effort than every-day development tasks. And second, the short duration of the tasks as compared to every-day development tasks might favor solo programming because the strengths of pair programming might pay off only during longer development tasks.

Another peril originates from bundling the experiment with a lab course. The author is aware of the ethical issues that might arise with this approach (e.g. Singer, 2002) but so far, we have gained positive experience with it. Our empirical research group is known among the students for its empirical studies and controlled experiments. Hence, the students know from previous experiment participants what to expect when subscribing for a bundle of lab course and experiment. The students are motivated and most of them are eager to hear the results of the study. And finally, the individual performance of a student in the experiment does not influence the decision whether he participated successfully in the course or not.

8. Conclusions

This paper presented two controlled experiments comparing pair programming to single programmers. The latter were assisted by a separate code review phase. The main contribution concerns the development cost associated with both techniques. Programmer pairs are as cheap as single developers if both developer pairs and single programmers are forced to produce programs of similar correctness. Thus, pair programming and solo programming become interchangeable. This result might imply a first management guideline for those who refrain from using pair programming but who are seeking for an (traditional) alternative. Another result takes programs of different level of correctness into account. In this case, programmer pairs produce programs with less failures at a higher expense as compared to single developers. Although this difference is visible in our data set, the result is not statistically significant. Possible reasons for the absence of statistical significance is that either there is no effect to detect or the effect is too small as that it can be detected with the used sample size.

However, there are open questions. What would have been the outcome of this comparison if professional programmer pairs and experienced reviewers had been used instead of students? The two groups might have performed better in terms of personnel cost. Other questions concern long term issues of the two development techniques. For example, what impact has the information flow during pair programming on the productivity and the skill level of the individual developers? And, is that information flow similar to the information flow of reviews or inspections? These questions can not be answered by one single study. Other studies have to be conducted to address these issues.

Acknowledgments

The author would like to thank Frank Padberg for the discussions and suggestions, Marcel Modes for supervising the second experiment, Vlad Olaru and Guido Malpohl for proof reading, and the anonymous reviewers for their comments on previous versions of this paper.

References

Basili, V., Shull, F., Lanubile, F., 1999. Building knowledge through families of experiments. IEEE Transactions on Software Engineering 25 (4), 456–473, Jul./Aug.

Beck, K., 1999. Extreme Programming Explained. Addison-Wesley.

Bisant, D., Lyle, J., 1989. A two-person inspection method to improve programming productivity. IEEE Transactions on Software Engineering 15 (10), 1294–1304.

Cockburn, A., Williams, L., 2000. The costs and benefits of pair programming. In: eXtreme Programming and Flexible Processes in Software Engineering (XP2000). Cagliari, Italy.

Cohen, J., 1988. Statistical Power Analysis for the Behavioral Sciences. Academic Press.

Dunsmore, A., Roper, M., Wood, M., 2003. The development and evaluation of three diverse techniques for object-oriented code inspection. IEEE Transactions on Software Engineering 29 (8), 677–686.

Gilb, T., Graham, D., 1993. Software Inspection. Addison-Wesley.

Hollander, M., Wolfe, D., 1999. Noparametric Statistical Methods, second ed. John Wiley & Sons.

Humphrey, W., 1995. A Discipline for Software Engineering. Addison-Wesley.

Humphrey, W., 1999. Introduction to the Team Software Process. Addison-Wesley.

Ihaka, R., Gentleman, R., 1996. R: A language for data analysis and graphics. Journal of Computational and Graphical Statistics 5 (3), 299–314.

Laitenberger, O., DeBaud, J., 1997. Perspective-based reading of code documents at Robert Bosch GMBH. Journal of Information and Software Technology 39, 781–791.

Laitenberger, O., Emam, K.E., Harbich, T., 2001. An internally replicated quasi-experimental comparison of checklist and perspective-based reading of code documents. IEEE Transactions on Software Engineering 27 (5), 387–421.

McDowell, C., Werner, L., Bullock, H., Fernald, J., 2002. The effects of pair-programming on performance in an introductory programming course. In: SIGCSE Technical Symposium on Computer Science Education. Cincinnati, Kentucky, USA, pp. 38–42.

Müller, M., 2003. Are reviews an alternative to pair Programming? In: Conference on Empirical Assessment In Software Engineering (EASE). Keele, UK, pp. 3–12.

Müller, M., Link, J., Sand, R., Malpohl, G., 2004. Extreme programming in curriculum: Experiences from academia and industry. In: Conference on Extreme Programming and Agile Processes in Software Engineering (XP2004). Garmisch-Partenkirchen, Germany.

Müller, M., Padberg, F., 2002. Extreme programming from an engineering economics point of view. In: International Workshop on Economics-Driven Software Engineering Research (EDSER). Orlando, Florida, USA.

Müller, M., Padberg, F., 2004. An empirical study about the feelgood factor in pair programming. In: International Symposium on Software Metrics (Metrics). Chicago, Illinois, USA.

Nawrocki, J., Wojciechowski, A., 2001. Experimental evaluation of pair programming. In: European Software Control and Metrics (Escom). London, UK.

Nosek, J., 1998. The case for collaborative programming. Communications of the ACM 41 (3), 105–108.

Padberg, F., Müller, M., 2003. Analyzing the cost and benefit of pair programming. In: International Symposium on Software Metrics (Metrics). Sydney, Australia.

PSP Resources Page, 2003. PSP resources page. Available from: <http://www.ipd.uka.de/PSP/>.

Sauer, C., Jeffrey, R., Land, L., Yetton, P., 2000. The effectiveness of software development technical reviews: A behaviorally motivated program of research. IEEE Transactions on Software Engineering 26 (1), 1–14.

Singer, J., 2002. Ethical issues in empirical studies of software engineering. IEEE Transactions on Software Engineering 28 (12), 1171–1180.

Tomayko, J., 2002. A comparison of pair programming to inspections for software defect reduction. Computer Science Education 12 (3), 213–222.

Williams, L., Kessler, R., 2000. The effects of pair-pressure and pair-learning on software engineering education. In: Conference on Software Engineering Education and Training. Austin, Texas, USA, pp. 59–65.

Williams, L., Kessler, R., Cunningham, W., Jeffries, R., 2000. Strengthening the case for pair-programming. IEEE Software (7/8), 19–25.

Matthias M. Müller received the diploma and PhD degrees in informatics from the University of Karlsruhe, Germany, in 1996 and 2000. In his dissertation, he worked on the topic of optimizing compilers for parallel architectures. In the last four years, he has focused on software process improvement, especially lightweight software processes. In particular, he works on the empirical and economical evaluation of the techniques proposed by extreme programming.