Should we use Programmer Pairs or Single Developers for the next Project ?

Matthias M. Müller
Fakultät für Informatik
Universität Karlsruhe
Am Fasanengarten 5, 76 128 Karlsruhe, Germany
muellerm@ipd.uka.de

Abstract

Todays project leaders are unsettled. Should they use programmer pairs or should they rely on single developers for the next project ? So far, empirical software engineering research gives no clear advice. One study reports on a doubled personnel cost when using programmer pairs instead of single developers while another study reports on a substantial increase of development speed and program quality. But, what would be if project leaders add a separate review phase to single programmers' development ? Then things change in terms of development cost and program quality. Two experiments conducted at the Universität Karlsruhe revealed that there is almost no difference in cost if both programmer pairs and single developers with reviews are forced to develop programs with comparable quality.

1. Introduction

Pair programming has become a valid alternative to conventional programming in the last years. Single 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 expected fun when pair programming should not be underestimated. However, programmers felicity is not untroubled. The nearly doubled personnel cost is *the* main disadvantage of pair programming. The monetary impact keeps project leaders from introducing pair programming in their projects. Wasting resources is the ubiquitous argument of the management when facing a project leader who tries to justify the usage of pair programming. Other arguments in favor of pair programming increase the probability that the project leader succeeds in this debate. Pair programming is said to accelerate development speed while at the same time increasing the quality of the developed programs. But in the end, the debate finishes with a draw (which quite naturally implies that the project leader had lost), if the management is aware of the state of the art in empirical software engineering research (ESER).

Current results in ESER depict a highly diversified picture on the advantages and disadvantages of pair programming. There is the study of Williams [13] stating a productivity increase of 42.5 percent[1] when developer pairs are compared to single developers. The pairs also produced 15 percent fewer errors than single programmers. Nosek [9] reports more moderate figures. In his study, the pairs finished 29 percent earlier than single programmers. And finally, Nawrocki [8] found no difference at all in development time. This diversified picture motivated several studies to look at the economic tradeoff when using pair programming [2, 12, 10]. Unfortunately, there is no easy answer, so far.

This study follows another approach than the other empirical studies mentioned so far. Two experiments were conducted to compare programmer pairs to single developers who were assisted by an additional anonymous review phase. The motivation was to find a technique which improves the performance of single developers such that they produce programs with the same quality as programmer pairs do, but with lower cost. The two experiments took place in the summer semester 2002 and 2003 at the Universität Karlsruhe. They are referred to as *Exp02* and *Exp03*, respectively. Participants were 38 computer science students.

This paper reports about the results of these two experiments. If the programmer pairs and the single developers are forced to produce programs with comparable quality, the single developers are 7 percent cheaper than the pairs. However, this difference is too small to be seen in practice. If same quality is not an issue, pairs cost 13 percent more and produce 29 percent more reliable programs than single developers with a separate review phase do. But the second

[1] *50 percent faster* means that the pairs are twice as fast as single developers.

result is confounded by the individual attitude to quality the pairs and single developers had. Thus, this result does not depend on the used programming method alone.
The author published the results of Exp02 in [7] (also referred to as *previous study*). This paper presents the replication of Exp02 and the results of the combined data-set. The replication of Exp02 was necessary because it addresses two weaknesses of Exp02. First, the replication increases the size of the data-samples and therefore the possibility of revealing an effect. And second, the replication removes a threat to internal validity the first experiment suffered from. In Exp02 *each* subject of a pair had to implement a task on its own and had to perform a review on the same task also. Thus, it was possible to get hints from the foreign program which could have been used for subsequent individual development. As this was actually the case for three subjects in Exp02, the according data-points had been deleted from the data-set used throughout this paper. Consequently, the experiment plan for Exp03 was altered such that from each pair, only one subject performed the implementation and the other one did the review. This halved the number of the data-points for the review group for Exp03, however, this change was necessary to obtain a higher internal validity of the experiment. According to the terminology on replications introduced by Basili [1, 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*.
As the design of Exp03 is nearly identical to that of Exp02 their descriptions vary only in small parts. To avoid too much repetition, the author repeated and modified the parts of [7] which are *essential* for understanding the design of the experiments, the results, and the underlying reasoning. For example, the description of Exp02's design was copied and extended (Section 2) to keep track of the changes and to ensure that this paper is understandable without knowledge of the previous work. All other parts that are not related directly to the repetition were removed, e.g. the detailed discussion of related work fell in this category.

2. The Study

Both experiments had a counterbalanced design and were held during the summer lectures 2002 and 2003, respectively, at the Universität Karlsruhe.

2.1. Environment

The experiments were part of an extreme programming course which in both cases 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 took place from May to June between the introductory sessions and the project week. The subjects subscribed voluntarily to the course and they knew from the very first course announcement that they had to take part in an experiment in order to get their course credits. All subjects were computer science undergraduate students who were on average in their fourth year of study. Java was the programming language for both the experiment and the lab course.

2.2. Methods and Tasks

Both studies compared the following methods:

Pair programming Two persons sit in front of a workstation and work together on the same task. Both developers share ideas in order to get a solution to the actual programming task.

Review One developer implements a solution to a problem and fixes all compilation errors. Then, he hands in his program for anonymous review. After the review, the developer gets back the program source together with a short description of the marked errors and finally, starts testing.

In both studies, subjects were introduced to pair programming and reviews. Each course took about 1.5 hours. Pair programming was taught by Extreme Programming (XP) professionals. Reviews were presented in the following week by the author. The subjects were forced to use only these two development methods. All the other techniques of XP were not part of this study.

The subjects had two solve two different tasks, each with another method:

Polynomial Find the zero positions of an arbitrary polynomial of third degree. The subjects had to implement the method `findZeroPosition` of a given polynomial-class.

Shuffle-Puzzle Find the solution of a given shuffle-puzzle within a given number of steps and list them if a solution exists. The subjects had to add a method `findMoves` to the basic class `ShufflePuzzle`.

The classes `Polynomial` and `ShufflePuzzle` contained constructors and methods for I/O to facilitate implementation and final testing.
The polynomial-task description contained a hint for a possible numeric solution to the problem. However, the students were not forced to use a special method to solve the problem. In fact, they could use the method they thought most suitable for this particular problem. For most

students, solving the task involved implementing the suggested method as well as thinking about special cases. The shuffle-puzzle task implied solving a backtracking problem to which the students knew the solution from their very first computer-sciences courses. Overall, the students were assumed skilled enough in both areas to overcome any problem-domain specific difficulties.

2.3. Plan

The procedures of the review and the pair programming task is outlined in Figure 1. Both procedures are divided into an *implementation* and a *quality-assurance* (QA) phase. As the experiment plan of Exp03 contains a slight modification of Exp02's plan, the Exp02 plan is described first. A discussion of the internal threat imposed by Exp02's plan follows. Finally, the plan of Exp03 is presented as it deletes the threat Exp02 suffered from.

Review Procedure of Exp02

For the description of the review task procedure we refer you to the upper half of Figure 1. *Implementation* is split into *coding*, *review*, and *testing*. During *coding*, the subjects had to implement the task until they thought they are done. During this part of the procedure, the subjects could only compile but not execute their programs. This constraint was guaranteed by the experiment environment. Thereafter, the program was printed out on paper and handed in for *review* to the subject's pair programming partner. As the review was anonymous, both the author and the reviewer of the code did not know each other.

The task of the unknown 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 process started only after both subjects finished *coding*. The review time was lower-bounded to at least 100 lines of code per hour. After the review had completed, the subjects got back the reviewed 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* (QA) 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% reliability, subjects got 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.

Discussion of Review Procedure

The review procedure did not allow the subjects to execute the code before review. This might not seem intuitive because, normally, the code is executed and tested very carefully before it is reviewed. However, as motivated by Humphrey [5, pp. 267-268], the author chose not to permit the 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 far from being correct. Therefore, it is worth a review. Second, the author of the program does not want the reviewer to find any errors. Thus, he develops his program carefully and does quite naturally a separate code-review of his own. Consequently, the program passes two reviews: the review of the program owner and the anonymous review. This would not have been the case, if the programs had been reviewed after the testing phase.

The lower bound for the review time of 100 lines of code per hour is based on figures shown by Gilb and Graham [4, p. 154]. Gilb and Graham 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 velocity seemed reasonable.

The main incentive of the quality assurance phase was to ensure high and comparable quality 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 quality, 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 reliability. But this variation is expected to be too small to be statistically detectable.

Pair Programming Procedure of Exp02

As opposed to the review procedure, the pair programming procedure was straight forward. During *implementation*, the pairs could 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, as described above, between running 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.

Threats of Exp02's plan

The major threat of Exp02's plan was the possibility that subjects could get hints from the foreign program for their own development. This actually occurred as it was pointed out in Section 3.7 of [7]. Three subjects admitted in the post-test questionnaire that they found suggestions for program improvement.

Experiment plan of Exp03

To remove this threat in Exp03, only *one* partner of a pair (as opposed to both in Exp02) prepared the task while the other

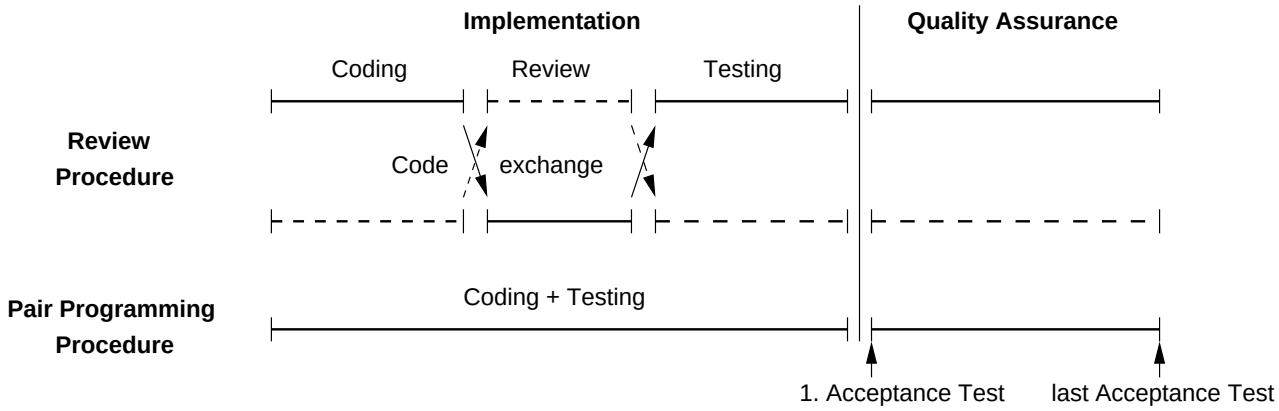

Figure 1. Procedure for the review and pair programming task.

partner performed the review. Remove the dashed line in the review part of Figure 1 to obtain the modified experiment plan of Exp03. Actually, this slight modification removes any effects caused by this threat. The pair programming procedure was not altered in Exp03.

Realisation of both 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 could not be finished in the first run, a subsequent appointment had to be made.

2.4. Selection of Groups

Table 1 shows the groups and the assigned task order for each group. The group numbers of Table 1 are referenced by the Tables 2 and 3.

Table 1. Task order for the groups (PP=pair programming, Re=review, Shu=shuffle-puzzle, Pol=polynomial).

Group	1. Task Method, Problem	2. Task Method, Problem
1	PP, Shu	Re, Pol
2	PP, Pol	Re, Shu
3	Re, Shu	PP, Pol
4	Re, Pol	PP, Shu

Table 2 lists the groups' overall programming and Java experience in years as well as in lines of code. The students had on average more than 6.5 years of programming experience and developed in that time frame on average more than 27,000 lines of code.

Table 2. Mean programming experience for each group.

	Overall Exp.		Java Exp.	
Group	Years	LOC	Years	LOC
	Exp02			
1	7.0	30,000	3.8	45,333
2	6.5	25,000	2.0	3,700
3	6.3	28,833	3.0	11,583
4	5.9	23,500	2.9	11,550
Total Exp02	6.5	27,297	2.9	15,963
	Exp03			
1	7.8	32,500	3.0	3,050
2	7.9	23,250	2.7	5,175
3	4.0	25,000	2.0	2,000
4	5.6	25,000	1.7	5,400
Total Exp03	7.0	27,819	2.3	5,155
Overall	6.7	27,448	2.7	12,834

The division of subjects into groups for both experiments was done according to their overall programming experience, independent of the programming language. The aim was to even out the general experience level. 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 data used for the division was obtained from the pre-test questionnaire the subjects had to fill out prior to the experiment. In both experiments the overall-experience in lines of code was chosen to be the decisive factor. However, the Java-specific experience could have been used also, but in the following project weeks after the experiments, it showed

that the overall experience represented the individual skill-level better than the Java-specific experience. The pre-test questionnaire revealed also, that in Exp02 (Exp03) six (one) subjects had some experience in reviews, two (six) subjects used pair programming, and one (no) subject had used both reviews and pair programming prior to the experiment.

Table 3. Number of available data-points per group.

	Exp02			Exp03		
Group	Size	PP	Re	Size	PP	Re
1	6	3	4	4	2	2
2	4	2	2	4	2	2
3	6	3	5	4	2	2
4	4	2	3	6	3	3
Overall	20	10	14	18	9	9

Table 3 lists the group sizes and the number of data-points available for analysis. In Exp02 three subjects did not finish the review task. They belonged to the groups 2, 3, and 4, respectively. Additionally, two data-points of group 1 and one data-point of group 2 had to be deleted to avoid the previous discussed internal threat. Thus, there were 10 data-points in the pair programming group and 14 data-points in the review group of Exp02. In Exp03 all data-points could be used, leading to 9 data-points for either group. Overall, there were 38 participants yielding 19 pair programming and 23 review data-points.

2.5. Data

Data collection and analysis was performed identically for both experiments.

Reliability

The reliability of two different versions of the developed programs was measured: the version after the implementation (Imp) and the version after the quality assurance phase. Two tests were created for each task: 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. Each test-case was randomly generated. Each *acceptance*-test consisted of 100 test-cases randomly selected from the large-test. Both acceptance-tests were generated once, before the experiment, and never changed afterwards. The test-cases for the polynomial-task consisted of a list of coefficients (starting from x^0), followed by the number of zero positions, and the zero positions itself. The test-framework read a test-case, initialized the implementation-under-test (IUT), executed it, and compared the results. If a deviation was detected, the test-case counted as failed. The polynomial coefficients for the test-cases were calculated from randomly generated zero positions. The test-cases for the shuffle-puzzle task were structured and executed the same way. The shuffle-puzzles used in the test-cases were randomly created with an upper bound for the number of steps. The number of steps used to create the shuffle-puzzle was later on decreased or increased by one to reduce or enlarge the search space for the IUT.

Apart from the fact that the acceptance-test was used to ensure comparable program quality between the subjects' programs, the usage of the acceptance-test stemmed from another reason also. When the acceptance-test was used during the quality assurance phase, the subjects got almost immediately feedback about the quality of their program. If the large-test had been used instead, the subjects would have been waiting for hours for the test results in the worst case. Thus, the delay incurred by the large-test would have been an unacceptable disruption of subjects' work flow. The results from the acceptance-test were available for analysis, though, the data was not used for evaluation purposes, as the large-test is expected to test a subjects' implementation more thoroughly. Consequently, reliability was evaluated with the large-test only.

The reliability r of a program was measured for two different program versions. The first version was the program right after the implementation phase. The reliability of this program version is denoted with r_{Imp}. The final program represented the second version. It was the version after quality assurance. The reliability of the final program is denoted with r_{Task}.

The reliability of a program version is the fraction of the number of passed tests divided by the number of all tests:

$$r = \frac{|\{\text{passed tests}\}|}{|\{\text{all tests}\}|}$$

The different program versions were gathered non-intrusively by the experimental environment without notice to the subjects. Each time a subject compiled its Java-code, the source-code was archived. At runtime, the experimental environment performed three operations: archiving the actual program version, logging the results, and writing the log to standard output. The subject triggered the data-collection mechanism implicitly by invoking the Java-compiler or the Java virtual-machine.

Cost

To study the cost, we compare the cost c_{phase}^{method} for method $method \in \{Pair, Review\}$ and phase $phase \in \{Imp, QA, Task\}$. The cost is measured in man minutes (mm). The cost of the two procedures consists of the 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}.

$$
\begin{aligned}
c_{Task}^{Pair} &= 2 \cdot (T_{Read} + T_{Imp} + T_{QA}) \\
c_{Task}^{Review} &= T_{Read} + T_{Imp} + T_{Rev} + T_{QA} \\
c_{Imp}^{Pair} &= 2 \cdot (T_{Read} + T_{Imp}) \\
c_{Imp}^{Review} &= T_{Read} + T_{Imp} + T_{Rev} \\
c_{QA}^{Pair} &= 2 \cdot T_{QA} \\
c_{QA}^{Review} &= T_{QA}
\end{aligned}
$$

T_{QA} consists of the rework time only and does not include the execution time of the acceptance-test. The review cost c^{Review} does not account for any additional waiting time, for example, the review synchronisation overhead.
The data were gathered with the *pplog-mode*, a major mode for Emacs which supports logging of work-time and interrupts [11]. Time logging was started and stopped by the experimenter.

2.6. Hypotheses

Both studies aimed to verify the following hypotheses and alternatives. In regard to the quality of the delivered programs, we assume for the null-hypotheses, that the mean reliability of the programs developed by the subjects of the pair programming group is *not* higher than the mean reliability of the programs developed by the subjects of the review group:

$$
\begin{aligned}
\boldsymbol{H}_{\mathbf{0}}^{\mathbf{Rel}} &: \mu_r(\text{Pair Programming}) \leq \mu_r(\text{Review}) \\
\boldsymbol{H}_{\mathbf{Alt}}^{\mathbf{Rel}} &: \mu_r(\text{Pair Programming}) > \mu_r(\text{Review}).
\end{aligned}
$$

With respect to the cost, we investigate the following null-hypotheses, according to which the average effort in man-minutes to complete a programming assignment is *not* higher for the pair programming group than for the review group:

$$
\begin{aligned}
\boldsymbol{H}_{\mathbf{0}}^{\mathbf{Cost}} &: \mu_c(\text{Pair Programming}) \leq \mu_c(\text{Review}) \\
\boldsymbol{H}_{\mathbf{Alt}}^{\mathbf{Cost}} &: \mu_c(\text{Pair Programming}) > \mu_c(\text{Review}).
\end{aligned}
$$

We consider the cost for the whole task, the implementation, and the quality assurance phase.

2.7. Power Analysis

The power of the one-sided t-test is 81%. From the alternative hypotheses perspective, we have a true chance of 81% to detect a difference between both groups, if any. But since the data-sets are rather small, we use the Wilcoxon-Mann-Whitney-test (up to now referred to as the Wilcoxon-test) and not the t-test. However, the Wilcoxon-test has an asymptotic efficiency of 95% of the t-test, which reduces the actual power down to 77%. The power of the t-test was calculated with R [6] using two samples, the harmonic mean of both group sizes $n = 20.8$, an effect size of 0.8, and a significance level of $\alpha = 0.05$. According to Cohen [3], the combination of both experiments has a quite good chance of revealing an effect, though, this effect has to be large. An effect size of 0.8 is equivalent to the mean difference in height between 13- and 18-year old girls [3, p. 27]. If we had chosen the interesting effect size to be equal to 0.5, which is equivalent to the mean difference in height between 14- and 18-year old girls [3, p. 26], then the power of the Wilcoxon-test would have been 45.2%. In any case, if the Wilcoxon-test does not reveal a significant difference between data-sets then there are two possibilities. Either the effect is too small to see and, therefore, our sample size is still too small or there is no defect to detect.

2.8. Internal threat

Two 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. The other possibility 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 teachers 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 safely because of the strict process definition. The experimentator enforced the process rigorously and left the subjects with no posibility for variation.

2.9. External threat

Several threats may have an impact on the generalisability of the study. One threat concerns both 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 it could be referred to as a professional. But since either group is affected by one of these threats, the effect is considered to be balanced. Another threat originates from the algorithmic structure of the polynomial and shuffle-puzzle task, because both tasks are more complex than every-day development tasks. But the high complexity balances the longer duration of the ordinary development tasks. Consequently, both tasks are not assumed to have any negative impact on the generalisability of the study.

3. Results

We use box plots to show the results of the measurements. The boxes within a plot contain 50% of the data points. The left (right) border of the box marks the 25% (75%) quantile. The left (right) t-bar marks the most extreme data point which is no more than 1.5 times the length of the box away from the left (right) 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. Evaluation bases on the one-sided Wilcoxon-Mann-Whitney two-sample test. Significance is set to a p-level of 0.05. 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. *Exp02* (*Pair02*, *Rev02*), *Exp03* (*Pair03*, *Rev03*), and *Both* (*Pair*, *Rev*) represent the (pair programming, review) data-sets of the first, the second, and the combination of both experiments, respectively. Different figures between this evaluation and the one shown in [7] are caused by the three removed data-points.

3.1. Reliability

Table 4 lists the reliability measures r_{Task} and r_{Imp} of the final and intermediate program versions. For the final programs, the reliability r_{Task} differs only slightly, if at all, and the overall reliability is high. These two observations are caused by the quality assurance phase. However, there is still some variation among the data-sets, see Figure 2. The data-points falling below the 95% exit criteria of the quality assurance phase are explained by the fact that the data-points of the large-test and not the acceptance-test are shown. The figures change for the reliability r_{Imp} of the program versions after the implementation phase, see lower part of Table 4 and Figure 3. On average, the pairs produce 64 (Exp02), 27 (Exp03), and 29 percent (Both) more reliable programs than single developers. Comparing the medians of the data-sets, the programs of the pairs perform 10, 89, and 56 percent better, respectively. However, none of the differences in location is statistically significant on the 5 percent level which is caused by the large variability within the data-sets.

Table 4. Reliability per method (in percent)

exp	PairProg			Review		
	$\bar{x}$	s	$\tilde{x}$	$\bar{x}$	s	$\tilde{x}$
r_{Task}						
Exp02	97	4	99	98	2	98
Exp03	97	4	99	97	3	97
Both	97	4	99	97	3	98
r_{Imp}						
Exp02	54	38	33	38	33	30
Exp03	56	27	70	49	32	37
Both	54	32	50	42	32	32

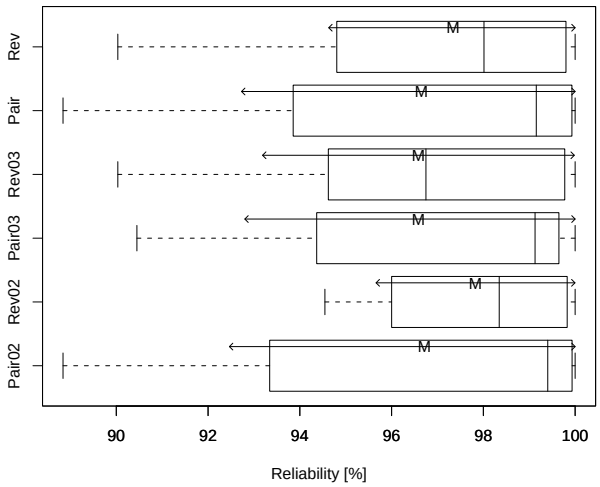

Figure 2. Reliability for Task r_{Task}

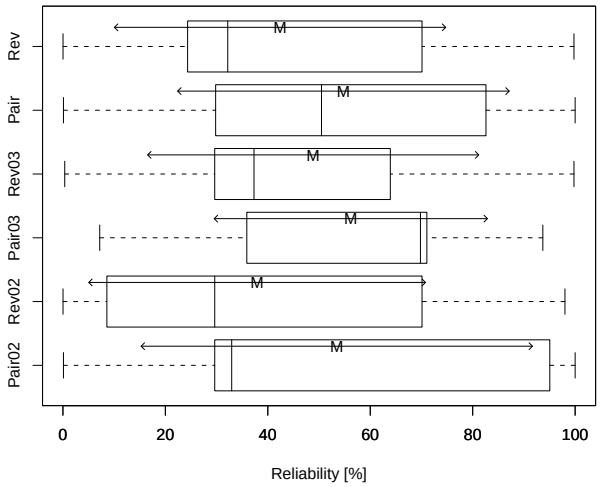

Figure 3. Reliability for Imp r_{Imp}

In summary, if both experiments are combined, the pairs produce 29 percent more reliable programs after the implementation phase than the single developers with reviews. For the final program versions, there is no difference in reliability.

3.2. Cost

Table 5 lists the cost for the whole task c_{Task}, the implementation phase c_{Imp}, and the quality assurance phase c_{QA}. First, the cost for the whole task is examined, see upper-

third of Table 5 and Figure 4. The pairs are on average 1, 10, and 7 percent more expensive than single developers. To be more precise, if the pairs and single programmers are forced to develop programs with equal reliability, there is almost no difference in terms of development effort between both groups. Even, the observed difference is too small as it could be seen in practice.

Table 5. Cost per method (in man minutes)

exp	PairProg			Review		
	$\bar{x}$	s	$\tilde{x}$	$\bar{x}$	s	$\tilde{x}$
			c_{Task}			
Exp02	490	114	471	487	258	452
Exp03	610	238	678	550	257	488
Both	547	188	512	512	253	458
			c_{Imp}			
Exp02	409	118	409	365	141	316
Exp03	474	166	400	427	150	445
Both	440	142	400	389	145	330
			c_{QA}			
Exp02	81	99	31	122	153	74
Exp03	136	113	82	123	153	43
Both	107	106	68	123	149	64

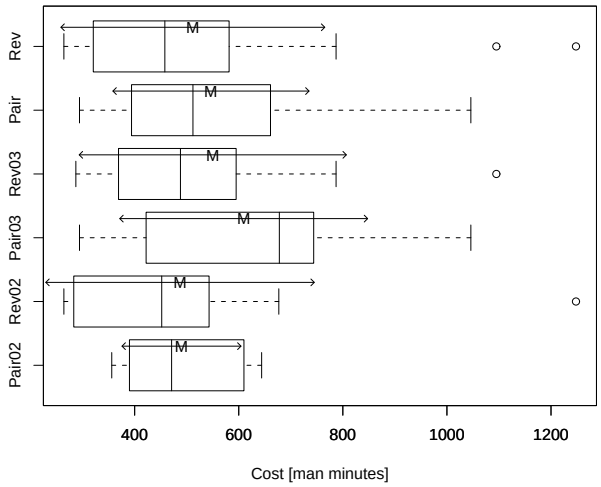

Figure 4. Cost for Task c_{Task}

The difference in cost increases if the cost for the implementation phase c_{Imp} is compared, see middle-third of Table 5 and Figure 5. Now, the pairs are on average 12, 11, and 13 percent more expensive than single developers. The pairs develop 29 percent more reliable programs while bringing about 13 percent more cost than single developers with reviews. But this result is confounded by the attitude to quality or testing each pair or individual developer possesses. Thus, this result is not caused by the different development methods alone.

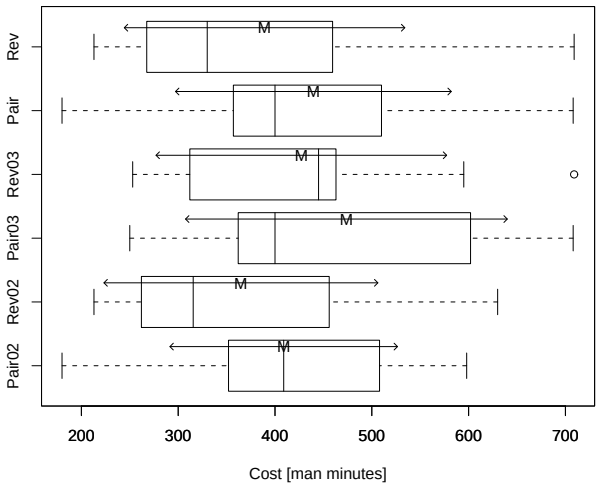

Figure 5. Cost for Imp c_{Imp}

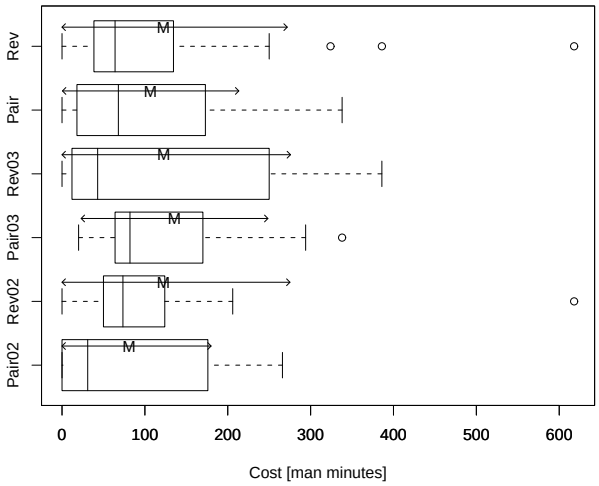

Figure 6. Cost for QA c_{QA}

As single developers are cheaper during implementation, these savings are lost during quality assurance, see lower-third of Table 5 and Figure 6. On average, the pairs are cheaper during quality assurance except for Exp03. This result is not surprising because single developers had to make up for a higher reliability difference than the pairs.
To conclude, if comparable quality is forced, the pairs are a bit more expensive than single developers, though, this difference is too small to be seen in practice.

3.3. 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 the previous study showed a learning effect from the first to the second assignment. Thus, it is demanding to repeat the sequence analysis on the Exp03 and Both data-sets. All data-sets of Table 6 show a general tendency, though, the Wilcoxon-test revealed no statistical significance on any of it.

Table 6. 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	$\tilde{x}$
r_{Task}						
Exp02	98.2	2.4	99.6	96.4	3.7	97.1
Exp03	97.3	3.0	99.0	95.8	4.0	96.7
Both	97.9	2.5	99.4	96.1	3.9	96.9
r_{Imp}						
Exp02	38.9	33.2	29.6	52.2	38.4	52.4
Exp03	48.4	35.0	37.3	56.7	22.0	63.9
Both	42.8	33.6	32.9	54.3	31.1	63.9
c_{Task}						
Exp02	547.0	240.9	512.0	419.1	137.3	398.0
Exp03	627.1	218.0	595.0	532.6	268.0	446.0
Both	579.8	230.0	536.0	470.1	208.8	414.0
c_{Imp}						
Exp02	431.0	134.5	419.0	326.7	106.8	302.0
Exp03	475.9	141.0	463.0	425.1	173.0	386.0
Both	449.4	135.7	450.5	371.0	145.5	348.0

Concerning reliability, there is a somewhat divergent picture. The final reliability r_{Task} decreases while the reliability r_{Imp} of the program versions after the implementation phase increases from the first to the second assignment. This observation is valid for all three data-sets Exp02, Exp03 and Both. Although this effect could be seen also in the previous study, the reason for this counterintuitive behavior is still unclear. However, there is a unique trend to cost reduction as the means and medians of the c_{Task} and c_{Imp} data-sets decrease from the first to the second assignment.

In summary, even though the final reliability decreased during the experiment, the other measures indicate a learning effect from the first to the second assignment.

3.4. Additional Results

The reviews lasted on average 63 minutes or about 12 percent 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.

Table 7 lists the number of acceptance-tests, the developed code size, and the cost for a quality assurance cycle (QAC). A QAC consists of one acceptance-test and its subsequent rework phase, if necessary. The execution time of the acceptance-test is not included. The QAC is obtained by dividing c_{QA} with the number of acceptance-tests. However, in order to perform this division, both data-sets must have a strong (linear) correlation. Correlation analysis for Exp02, Exp03, and Both revealed a correlation coefficient r^2 of 0.78, 0.28, and 0.54, respectively. According to Humphrey [5, p. 513], the calculation of QAC is allowed only for Exp02 and Both as r^2 is larger than 0.5 for these both data-sets. Consequently, the QAC figures for Exp03 have been omitted from Table 7. Anyway, the calculated values for QAC have to be treated with caution.

Table 7. Comparison of number of acceptance-tests, size of written code, and QAC cost.

category	PairProg			Reviews		
	$\bar{x}$	s	$\tilde{x}$	$\bar{x}$	s	$\tilde{x}$
number of acceptance-tests						
Exp02	3.0	2.3	2.5	4.6	5.0	3.0
Exp03	3.7	1.4	4.0	5.3	5.1	4.0
Both	3.3	1.9	3.0	4.9	4.9	3.0
code size in LOC						
Exp02	156	34.7	148	150	31.3	140
Exp03	130	33.6	140	175	67.3	140
Both	144	35.8	144	160	48.8	140
QAC cost in man minutes per cycle						
Exp02	20.0	25.5	8.6	25.6	15.4	24.8
Both	27.6	26.5	20.5	23.6	21.8	20.0

For Exp02, the average QAC cost for pairs is lower than that for single developers. This observation was interpreted in [7] as a more than twofold increase in productivity during quality assurance when switching from single programmers to programmer pairs. However, this figure does not hold anymore if the combined data-set Both is observed.

4. Conclusions

This paper presented two empirical studies concerning the comparison of developer pairs with single programmers. The latter were assisted by a separate review phase. Two effects of the comparison could be observed:

1. Single developers cost 7 percent fewer than developer pairs, if both developer pairs and single programmers are forced to produce programs of equal quality. While the experiment data revealed the 7 percent cost savings, it is questionable if this cost advantage of the single developers can be seen in practice, too.

2. Programmer pairs are 13 percent more expensive while producing 29 percent more reliable programs than single developers with reviews. However, this result is not caused by the different implementation techniques alone. In fact, the individual attitude to quality of the pairs and the single developers is another independent variable.

Even though the reported numbers seem to quantify the difference between pairs and single developers, none of the differences among the data-sets is statistical significant. Consequently, the reported numbers have to be treated cautiously.

Two reasons could explain the absence of any statistical significance. Either there is no effect to detect or the effect is too small to detect. Despite the fact that this study did not detect any effect even though it had a chance of nearly 80 percent to reveal a large one, the author is still convinced that there is a cost difference, though, this difference is too costly to detect.[2]

The main incentive of this study was the comparison of programmer pairs with single developers in terms of personnel cost. The result of this study is clear: if equal quality is an issue, programmer pairs and single developers become interchangeable. This result sounds good news for conservative management seeking an equilibrium in the used development techniques. Arguments against this conclusion are the open questions that still remain. For example, what impact has the information flow during pair programming on the productivity and the skill level of the individual developers ? Or, to what extend does pair programming mitigate staff turn-over, if it does at all ? But these questions address long-term issues of pair programming for which different kind of studies have to be conducted.

5. Acknowledgments

The author would like to thank Marcel Modes for supervising the second experiment and Vlad Olaru for proof reading a previous version of this article.

References

[1] V. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, July/Aug. 1999.

[2] A. Cockburn and L. Williams. The costs and benefi ts of pair programming. In *eXtreme Programming and Flexible Processes in Software Engineering (XP2000)*, Cagliari, Italy, June 2000.

[2] For example, the sample-size required for a one-sided two-sample t-test detecting the effect-size of this study of $(547 - 512)/224.4 \approx 0.16$ with a power of 80 percent is 484 data-points per group. As two developers are needed for one data-point, $484 \cdot 2 \cdot 2 = 1936$ developers have to be engaged.

[3] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Academic Press, 1988.

[4] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.

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

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

[7] M. Müller. Are reviews an alternative to pair Programming ? In *Conference on Empirical Assessment In Software Engineering (EASE)*, pages 3–12, Keele, UK, Apr. 2003.

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

[9] J. Nosek. The case for collaborative programming. *Communications of the ACM*, 41(3):105–108, Mar. 1998.

[10] F. Padberg and M. Müller. Analyzing the cost and benefi t of pair programming. In *International Symposium on Software Metrics (Metrics)*, Sydney, Australia, Sept. 2003.

[11] PSP resources page. http://www.ipd.uka.de/PSP/.

[12] L. Williams and H. Erdogmus. On the economic feasibility of pair programming. In *International Workshop on Economics-Driven Software Engineering Research (EDSER)*, Orlando, Florida, USA, May 2002.

[13] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair-programming. *IEEE Software*, pages 19–25, July/Aug. 2000.