Are Reviews an Alternative to Pair Programming ?

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

Abstract

From the first presentation of extreme programming on, pair programming has attracted a wide range of programmers to work together in front of one display. The proposed advantages of pair programming are a faster development cycle and code with higher quality. However, the nearly doubled personal cost when compared to single developers seems to outweigh these advantages. Instead of showing the superiority of pair programming, we seek for an alternative. Can a single developer be assisted by an already known technique with which he produces 80% of the quality of pairs with only 20% of the cost? The answer with some restrictions is: yes, he can. Reviews are a reasonable candidate with respect to code quality and cost.

1 Introduction

Pair programming has attracted lots of developers in the last five years. The expected fun when pair programming is one reason for this attraction but the main reasons from a project's management perspective are its promising goals: faster development and higher quality code. But not only project management profits from pair programming: single programmers learn from their partner while pair programming, share ideas, and find solutions which none of them would have found alone. And after all, pair programming promises fun in the sometimes dull development process. A team of developers committing to pair programming shares responsibilities, denies specialization and thus, reduces the risk of a project failure caused by personal change-over. However, the major drawback of pair programming is the almost doubled programming effort assumed. It prevents conservative managers from introducing pair programming.

The economical efficacy of pair programming depends mainly on the increased quality and programming speed of pairs over single developers [7]. Instead of estimating possible values for both parameters and thus, delimiting the potential of pair programming over single developers, this study tries to seek an alternative technique with which a single developer produces code with almost the same quality but with lower cost. Inspections are the first candidate everyone thinks of as an alternative to pair programming. However, they are time consuming and thus very expensive. But reviews, used for the preparation of an inspection meeting, seem to be a reasonable competitor because they lead to higher quality code with only moderate additional cost.

This paper investigates with a controlled experiment if a single programmer equipped with an additional review phase can compete with pair programming in terms of code quality and development cost. The experiment was conducted with CS students at the Universität Karlsruhe during the lectures of the summer semester 2002. It was designed to evaluate the following null-hypotheses.

Reliability Pairs of developers do *not* produce more reliable code than single developers equipped with reviews.

Effort Pairs of developers are *not* more costly in person hours than single developers equipped with reviews.

The results of the evaluation were unexpected because both hypotheses could not be rejected.
The next section discusses related studies concerning the efficacy of developer pairs. Section 3 details the experiment settings before section 4 presents the results. Conclusions are drawn in section 5.

2 Related Work

So far, pair programming has been compared only to single developers.

Williams studied pair programming with 41 undergraduate students [12]. 13 students formed the control group in which all the work was done individually. The remaining 28 students formed the experiment group in which all the work was done in pairs. The study took six weeks during which all the students and pairs completed four assignments. Concerning reliability, the pairs passed more of the automated post-development test-cases. This difference was significant with $p < 0.01$. The data-sample of the pairs also had a smaller variance as opposed to the 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 dropped to 15%.

Nosek studied 15 professional programmers on a database consistency check [9]. None of the subjects had worked on this problem before. The 5 individuals and the 5 pairs got a period of 45 minutes to complete the task. All pairs outperformed the individuals. Although the average time for completion was more than 12 minutes (41%) longer for individuals, the difference was not statistically significant on the 5% level.

Nawrocki studied 21 computer science students [8] on the first four assignments of the *Personal Software Process* programming course [4]. 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 in sharp contrast to the Williams and the Nosek studies. But the variability within the pair programming group was smaller when compared to the other two groups, which led to the conclusion that the pair programming process is more predictable.

Although all of the studies were conducted to evaluate the advantage of pair programming over individuals, the results of the first (and maybe also of the third[1]) study are inconclusive to some extent because they are confounded by a second uncontrolled variable: *personal attitude to testing*. The assignment was considered completed as soon as the participants (pairs and individuals) claimed so. Thus, the produced programs differed not only in the time needed for completion but also in their quality because each subject and pair has a different attitude on when it is beneficial to stop testing and when not. In this case, it is methodologically questionable to compare these program versions. The only way out of this dilemma is to fix either of the two variables: time to completion or code quality. For example, a valid comparison of two development methods should consider only program versions that are developed within a fixed period of time as it was done in the Nosek study, or those that achieve a certain reliability and to force rework if a program does not reach the reliability threshold.

This study implements the second possibility to avoid the pitfall of confounding variables. This is done with an additional quality assurance (QA) phase issued at the end of the implementation process. This QA phase ensures a comparable quality of subject's 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 QA phase.

Other studies investigated the advantages of pair programming for educational purposes [11, 6] or evaluated the potential costs and benefits of pair programming [1, 7].

[1] The experimental settings are described too vague in [8] as to get any reliable information on this topic.

3 The Study

The experiment had a counterbalanced design and was held during the summer lectures 2002 at the University of Karlsruhe as a part of an extreme programming course. 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 experiment took place from May to June between the introductory sessions and the project week. Java was the programming language for both the experiment and the lab course. The subjects 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. The students had on average more than 6 years of programming experience and their largest developed programs ranged between 50 and 250,000 LOC with a median of 2000 LOC.[2] One subject had used both reviews and pair programming prior to the experiment, six subjects mentioned little experience only in reviews, and another subject only in pair programming.

3.1 Methods and Tasks

The study compared the following methods:

Pair programming Two persons sit in front of a display, keyboard, and mouse and work together on the same task. Both developer share ideas in order to get a solution to the actual programming task.

Review A single developer implements a solution to a problem and fixes all compilation errors, but he does not execute the program. Then, he hands in his program for review. The task of the unknown reviewer is to find errors according to a checklist. Design flaws, violations of any sort of convention, and suggestions for a better solution are of no concern to the review. The developer gets back the program source together with a short description of the marked errors and after all, starts testing.

Subjects got an introduction in pair programming and reviews. Each course took about 1.5 hours. Pair programming was introduced by extreme programming professionals. Reviews were introduced in the following week by the author. The subjects were forced to use only these two development methods. All the other techniques of extreme programming 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. The polynomial-class contained methods for the basic arithmetic expressions and for I/O.

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 subejcts had to add a method findMoves to the basic class Shuffle-Puzzle. This class initially contained constructors and methods for I/O.

The polynomial-task description contained a hint for a possible numeric solution of the problem. However, the students were not forced to use a special method to get a solution. In fact, they could use the method they thought most suitable for this problem. For most students, the task involved implementing the suggested method as well as thinking about all special cases. The shuffle-puzzle task implies 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] The pretest questionnaire asked for the size of the largest developed program. It seems quite reasonable that the presented figures contain reused code as well. The final decision, if reused code should be counted as well, was left to the subject.

3.2 Procedures

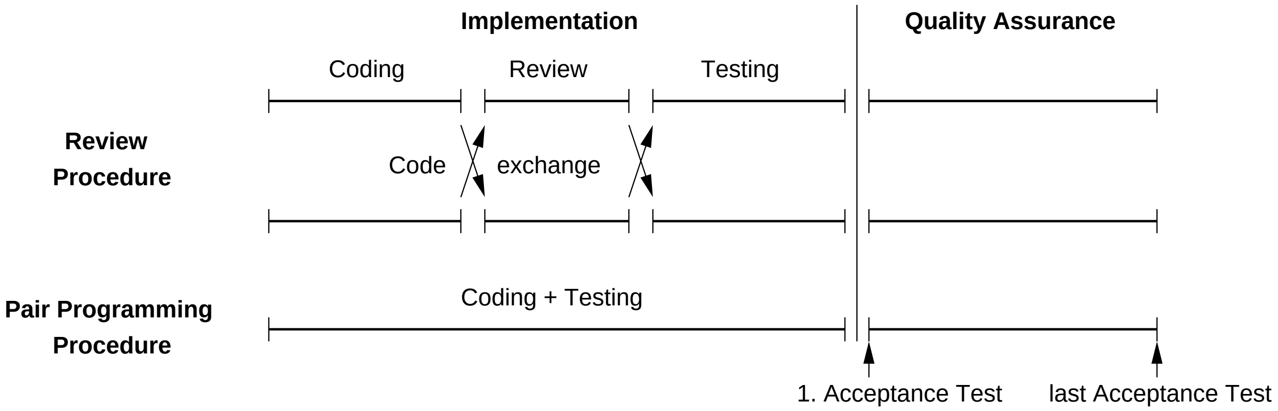

Figure 1. Procedure for the review and pair programming task.

The Review Procedure

Figure 1 outlines the procedure of the review and the pair programming tasks. Both procedures were divided into an *implementation* and a *quality-assurance* (QA) phase. First, the review task procedure is described, see 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 have finished. 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 review process started only after both subjects finished *coding.* The review time was bounded to at least 100 lines of code per hour. After the review was done, 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 finished the task.

Discussion of Review Procedure

The previously described 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 [4, 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 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 [3, p. 154]. Gilb and Graham suggest a review velocity of one page (noncommentary, 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 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 personal attitude to testing is factored out, see also the discussion in section 2. However, the exit criterion of the QA phase leaves some room for variation in reliability. But this variation is expected to be too small to be statistically detectable.

The Pair Programming Procedure

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.

Realisation

The pair programming procedure could be done in one session, while the review procedure involved at least two different sessions: one session 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.

3.3 Selection of Groups

Table 1. Assignment to tasks and methods, sizes, available data-points, and mean experience for each group (PP=pair programming, Re=review, Shu=shuffle-puzzle, Pol=polynomial).

	1. Task	2. Task				Overall Exp.		Java Exp.	
Group	Method, Problem	Method, Problem	No	PP	Re	Years	LOC	Years	LOC
1	PP, Shu	Re, Pol	6	3	6	7.0	30,333	3.8	45,333
2	PP, Pol	Re, Shu	4	2	3	6.5	25,000	2.0	3,700
3	Re, Shu	PP, Pol	6	3	5	6.3	30,000	3.0	11,583
4	Re, Pol	PP, Shu	4	2	3	5.9	23,500	2.9	11,550
		Overall	20	10	17				

Table 1 lists the tasks and methods assigned to each group as well as the group sizes and the available number of data-points. Since 3 subjects did not finish the review task, 27 out of the 30 expected data-points were available for analysis.

The division of subjects into groups was done according to their overall programming experience, independent of the programming language. The aim was to obtain a general experience-level evenly distributed over the four groups. 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. The four rightmost columns of table 1 list the average overall experience and the Java-specific experience in years and lines of code for each group. The experimenter chose the overall-experience in lines of code to be the decisive factor. However, the Java-specific experience could also have been used, but it showed in the project week after the experiment, that the overall experience represented the individual skill-level better than the Java-specific experience.

3.4 Data

Reliability

The reliability of two different versions of the developed programs was measured: the version after the implementation (Imp) and the version after the QA 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 selected randomly 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), the number of zero positions, and the zero positions itself, see figure 2. The test-framework reads a test-case, initializes the implementation-under-test (IUT), executes it, and compares 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, see figure 3. The shuffle-puzzles for the test-cases were randomly created with an upper bound for the number of moves. The number of moves used to create the shuffle-puzzle was later on decreased or increased by one to reduce or enlarge the search space for the IUT.

The reliability r of a program measured with test $test \in \{Large, Acc\}$ after phase $phase \in \{Imp, QA\}$ is the fraction of the number of passed tests divided by the number of all tests:

$$r_{phase}^{test} = \frac{|\{\text{passed tests}\}|}{|\{\text{all tests}\}|}$$

```
3150 6375 3300 75 2 -42 -1
231875 14175 21 -7 2 -25 53
-680625 43725 1529 11 2 -75 11
```

Figure 2. Input for Polynomial tester.

```
Depth 7 Rows 2 Columns 2
2 3
1 0
Moves DOWN RIGHT UP LEFT
```

Figure 3. Input for Shuffle-Puzzle tester.

The data was gathered non-intrusively by the experiment environment without notice to the subjects. Each time, a subject compiled its Java-code, the source-code was archived. On execution, the experiment environment performed three operations: archiving the actual program version, logging the output, 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 accumulated cost c_{phase}^{method} for method $method \in \{Pair, Review\}$ after phase $phase \in \{Imp, QA\}$ measured in man minutes (mm). The cost of the different procedures consist 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 in the QA phase T_{QA}.

$$c_{QA}^{Pair} = 2 \cdot (T_{Read} + T_{Imp} + T_{QA}) \qquad c_{Imp}^{Pair} = 2 \cdot (T_{Read} + T_{Imp})$$

$$c_{QA}^{Review} = T_{Read} + T_{Imp} + T_{Rev} + T_{QA} \qquad c_{Imp}^{Review} = T_{Read} + T_{Imp} + T_{Rev}$$

T_{QA} consists of the rework time only and does not include the execution time of the acceptance-test.

The cost for the QA phase alone is studied as well. It is the difference between c_{QA} and c_{Imp}.

$$c_{QA-Imp} = c_{QA} - c_{Imp}$$

The review cost does not account for any additional waiting time, for example, the review synchronisation overhead.

The data was gathered with the *pplog-mode*, a major mode for Emacs which supports logging of work-time and interrupts [10]. The *pplog-mode* was initially built for our personal-software-process programming courses, but it is also a comfortable time logging tool for experimentation. The logging was started and stopped by the experimenter. The subjects had to log only interrupts such as going to toilet, taking a smoking-break, or going to lunch.

3.5 Hypotheses

The study 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:

$$\boldsymbol{H}_{\mathbf{0}}^{\textbf{Rel}} : \mu_r(\text{Pair Programming}) \leq \mu_r(\text{Review}) \qquad \boldsymbol{H}_{\textbf{Alt}}^{\textbf{Rel}} : \mu_r(\text{Pair Programming}) > \mu_r(\text{Review}).$$

We consider all four reliability measures defined in section 3.4.
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:

$$\boldsymbol{H}_{\mathbf{0}}^{\textbf{Cost}} : \mu_c(\text{Pair Programming}) \leq \mu_c(\text{Review}) \qquad \boldsymbol{H}_{\textbf{Alt}}^{\textbf{Cost}} : \mu_c(\text{Pair Programming}) > \mu_c(\text{Review}).$$

We consider the accumulated cost after the implementation and the quality assurance phase.

3.6 Power Analysis

The power of the one-sided t-test is 64%. From the alternative hypotheses perspective, we have only a chance of 64% 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 yet another 5% down to 60%. The power of the t-test was calculated with R [5] using two samples, the harmonic mean of both group sizes $n = 13.3$, an effect size of 0.8, and a significance level of $\alpha = 0.05$. According to Cohen [2] this experiment has a poor power of revealing an effect, even if this effect is large. But as the maximum number of students for the extreme programming course, initial planned to be 18, has already been exceeded by two, we could not afford additional students.

3.7 Internal threat

The internal validity of the experiment is threatened by three perils. First, different persons teaching the lectures on pair programming (professionals) and reviews (the author) could cause differences in skill and motivation between 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 safely be ignored because of the strict process definition. The experimentator enforced the process rigorously and left the subjects no posibility for variation.

The third threat originates from the possibility, that the subjects could have obtained information from the review of the other program which they could have used in their own development. To check the existence of this threat, the post-test questionnaire asked whether the subjects got from the foreign program any information or hints for possible errors which they could use to fix or to improve their own program. Actually, 3 subjects affirmed that they learned from the foreign program something they could use in their own development. The first subject reported on tests about null-objects, which were missing in the foreign program and which he inserted in his own program after the review. The second subject (no. 405), changed his comparison of floats with zero ($f == 0$) to an ϵ-range comparison ($|f| < \epsilon$). The third subject saw the inefficiency of his own code, but did not change it.

Finally, the third threat can be ignored except for the second subject (405). The author compared the data of subject 405 with the results of the remaining review-group. The cost of subject 405 ranged very close to the group median and the reliability of its program ranged in the group's middle 50%. As the influence of subject 405 on the whole outcome of the analysis was neglectable, the author decided to use it for evaluation.

3.8 External threat

Several threats may have an impact on the generalisability of the study. First, the subjects did not meet before the pair programming task and second, except for one subject, no one had performed reviews 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. Furthermore, if we compare the development cost of the pairs to the development cost of the individuals, the pairs were 5% more expensive. And, if we ignore the additional review cost of the individuals, the additional pair programming cost increases to 20%. Compared to the pair programming overhead of 15% presented in [12], the pairs performed well and it is not yet clear if they would improve their performance further in successive assignments.

A third 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 assumed not to have any negative impact on the generalisability of the study.

4 Results

We use box plots to show the results of the measurements. The boxes within a plot contain 50% of the data points. The lower (upper) 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 error on each side. Evaluation bases on the Wilcoxon-Mann-Whitney two-sample test (p-level of 0.05).

4.1 Reliability

The two boxplots on left hand side of figure 4 depict the reliability of the final programs r_{QA}. As the exit criterion of the QA phase was to pass at least 95 tests of the acceptance-test, the reliability of all programs lies above this threshold, see left most plot of figure 4. There is almost no difference between both groups concerning the median (pairs 99%, review 100%) and the mean (pairs 98.4%, review 98.8%) of the data-samples. The reliability for the large-test is not as good as compared to the numbers of the acceptance-test (median: pairs 99.4%, review 98%; mean: pairs 96.7%, review 97.4%), see second to left plot of figure 4. But as expected, the differences within both tests, acceptance- and large-test, are far away from being significant.

So far, each subject had to pass a quality control. But what would have happened if the QA-phase had been omitted? The two boxplots on the right hand side of figure 4 show the reliability of the programs after the Implementation phase (r_{Imp}). From a methodological point of view, this comparison is confounded by a second

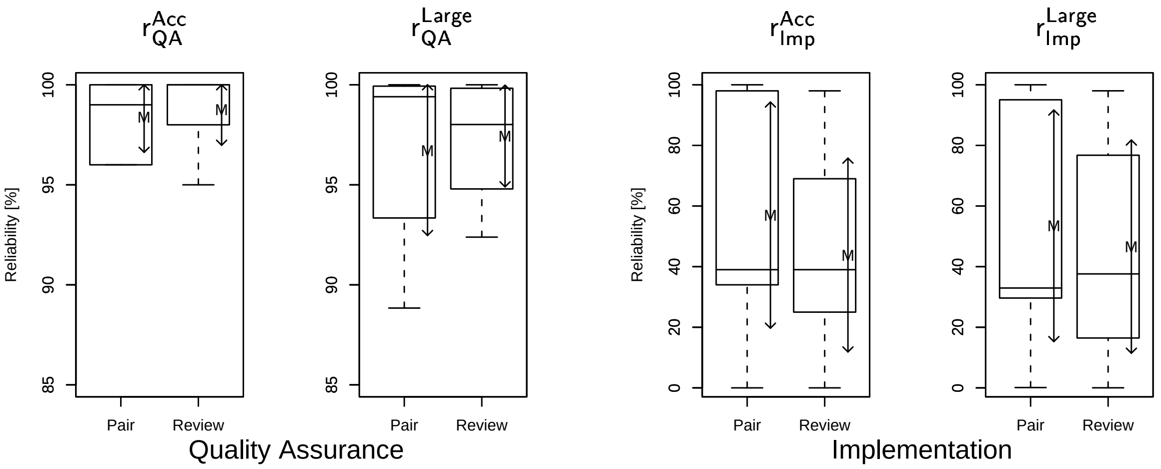

Figure 4. Reliability of final programs after Quality Assurance and after Implementation.

independent variable *personal attitude to testing* as it has already been discussed in section 2. But this comparison is interesting anyway as parts of this study should be comparable to other studies. Both tests (acceptance- and large-test) reveal similar results concerning reliability. The average reliability of the programs developed by the pairs is higher than that of single developers (acceptance-test: pair 57%, review 43.8%; large-test: pair 53.5%, review 46.6%), though the medians differ only slightly (acceptance-test: pair 39%, review 39%; large-test: pair 33%, review 37.6%). The box containing 50% of the pair data-sample is more shifted to the 100% level than the box of the review data-sample but the review data-sample has a smaller variability. Both differences are not statistically significant, however.

Despite the fact that on average the pairs produced programs with a higher reliability before the QA phase, the small measured difference after implementation was not expected. Is this small difference in reliability caused by the reviews? As it turned out from the post-test questionnaire, the subjects got little feedback from the reviews. Thus, the quality of the programs developed by single developers can not be ascribed to the error-finding capabilities of the reviews alone. However, the possible reason for the small difference in reliability could be a psychological effect because the developer's attitude to quality changes if he knows that somebody else is looking at his program. Thus, the subjects were more aware of program flaws and anxious about being blamed on a bad program structure. Finally, this is still an effect caused by the reviews, because if the reviews had been omitted, the subjects would not have changed their programming style and thus, they would have produced programs with lower measurable reliability.

4.2 Cost

Figure 5 depicts the boxplots for the costs of the whole task c_{QA}, for implementation c_{Imp}, and for QA c_{QA-Imp}. There is a small difference in the total task cost between both groups if the medians (pair 471 mm, review 442 mm) and the means (pair 490 mm, review 467 mm) are compared. However, due to the large variance in the data-sets of both groups, this difference is not significant. The variance of the c_{QA}^{Rev} data-set is larger than for the c_{QA}^{Pair} data-set (longer dashes). To say it from the economical point of view: the single programmers are 5% cheaper than the developer pairs if the average cost is compared. If we combine this result with the result of the reliability analysis, we have to say that the single programmers developed programs with comparable quality but with fewer cost as compared to developer pairs. But to be honest, it is not clear if these cost savings can really be observed in practice.

The other two box plots of figure 5 depict the distribution of the cost on the implementation and the QA phase. Single programmers are cheaper during implementation (median: pair 409, review 324; mean: pair 409, review 357) but these savings are lost during QA.

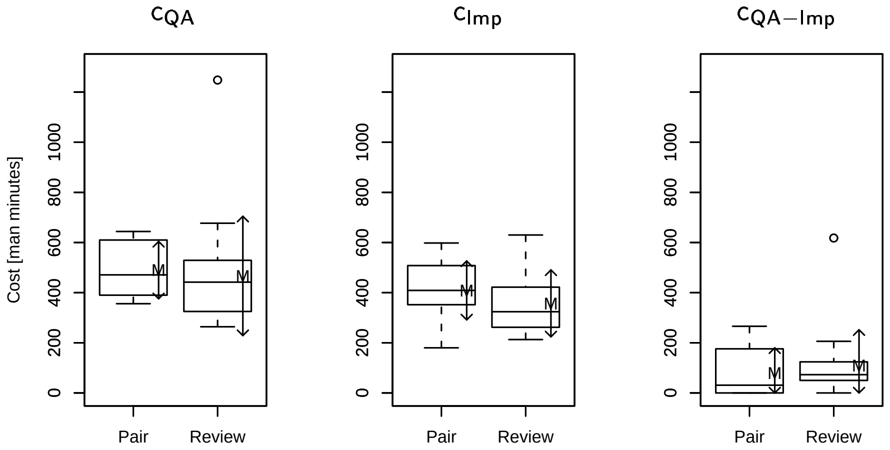

Figure 5. Cost for whole Task c_{QA}, Implementation c_{Imp}, and Quality Assurance c_{QA-Imp}.

4.3 Sequence analysis

Due to the experiment's counterbalanced design, it is mandatory to look at effects that have their roots in the consecutive treatment of two development tasks. We were looking for differences between assignments ignoring the specific method and the specific problem. The focus lies only on the order of the assignments. Table 2 compares the means, standard deviations, and medians of the main metrics used in this study. Statistically significant values are marked.

Table 2. Differences between first and second assignment averaged over groups and tasks.

category	1. assignment			2. assignment			p-level
	mean	std-dev	median	mean	std-dev	median	Wilcoxon
r_{QA}^{Large} [%]	98.4	2.28	99.6	95.8	3.98	96.8	0.235
r_{QA}^{Acc} [%]	99.1	1.37	100.0	98.1	1.93	98.0	0.116
r_{Imp}^{Large} [%]	44.2	34.01	30.0	56.7	37.71	69.7	0.568
r_{Imp}^{Acc} [%]	47.2	33.52	39.0	54.5	36.06	39.0	0.510
c_{QA} [man min]	536	212	512	426	121	398	0.052
c_{Imp} [man min]	442	124	438	331	100	324	**0.018**

The first four rows focus on the reliability of the different program versions. The reliability of the program versions after the QA phase decreases from the first to the second assignment while the reliability of the program versions right after the implementation phase increases. These effects are quite visible from the means and the medians of both assignments. However, none of the effects is statistically significant. In summary, any trends towards a change in reliability can not be seen, as one measure indicates a potential loss and the other a potential improvement in reliability.

In contrast to the divergent picture for reliability, there is a clear trend towards a reduction of the programming effort from the first to the second assignment. Both c_{QA} and c_{Imp} data sets show a decrease of programming effort. While the difference in the c_{QA} data set is not significant, the c_{Imp} data set is an indication towards an improvement of productivity and points out a learning effect from the first to the second assignment.

4.4 Additional Results

The reviews lasted on average 60 minutes or about 15% of the total development time. The average size of reviewed programs was 113 lines of code and the average review speed was 119 lines of code per hour.

Table 3. Comparison of number of acceptance-tests, size of written code, and QA cycle cost.

category	Pairs			Reviews			p-level
	mean	std-dev	median	mean	std-dev	median	Wilcoxon
NoAccTests	3.0	2.7	2.5	4.2	4.6	3.0	0.426
CodeSize [LOC]	156.2	34.74	148	151.5	33.29	141	0.669
$\frac{c_{QA-Imp}}{NoAccTests}$ [man min]	20.0	25.47	8.6	24.2	16.15	24.7	0.247

Table 3 lists the medians and means for other metrics of this study: number of acceptance-tests, the developed code size, and the correction cost for a QA cycle. A QA cycle consists of one acceptance-test and its subsequent rework phase, if necessary. However, the QA cycle cost consists only of the time needed for rework. The execution time of the acceptance-test is not included. It is reasonable to study the correction cost for a QA cycle because both data-sets (c_{QA-Imp} and $NoAccTests$) are correlated with $r^2 = 0.78$. According to [4], this is a strong correlation and adequate for planning purposes because $0.7 \leq r^2 < 0.9$.

None of the presented metrics of table 3 shows a significant difference. However, it is remarkable that the average cost for a QA cycle is lower for developer pairs than for the individuals despite the doubled personal cost for the pairs. Thus, the pairs were more than two times faster during QA.

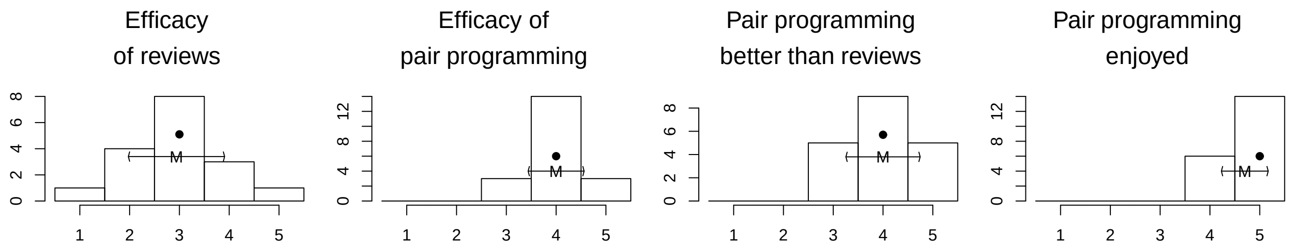

Figure 6. Results from post test questionnaire.

Figure 6 shows some results of the post-test questionnaire. One task of the questionnaire was to classify the overall efficacy of reviews and pair programming, see the two left most histograms of the figure. A scale ranging from 1 (*very bad*) to 5 (*very good*) was provided for each classification. It turned out that the subjects were more convinced about the efficacy of pair programming (median: 4, see thick dot) as compared to the efficacy of reviews (median: 3). This result is consistent with the outcome of the ranking of pair programming as compared to reviews, second to right histogram. Another part of the questionnaire evaluated the fun experienced with pair programming, right most histogram. This issue was worth an evaluation because the fun is regarded as one advantage of pair programming over individual development. And as expected, 15 subjects (75%) enjoyed the pair programming task very much and the other 5 subjects had also fun trying it.

5 Conclusions

This paper compared developer pairs to single programmers assisted by a separate review phase. The comparison considered the issues of more expensive developer pairs and more reliable code produced by pairs as compared to single developers. The results suggest to negatively answering both issues.

- A pair of developers does not produce more reliable code than a single developer whose code was reviewed.
- Although pairs of developers tend to cost more than single developers equipped with reviews, the increased cost is too small to be seen in practice.

To answer the question about a method that produces 80% of the quality of developer pairs with only 20% of the cost: reviews produce the same quality with nearly the same cost.

Another outcome of this study is the lower defect fix cost for the pairs during quality assurance as compared to individual developers, because developer pairs seem to be more than twice as fast during quality assurance than individual developers. However, it is yet unclear if this effect can only be seen for pairs fixing their own code, or if it is inherent to pair programming and thus, independent on the author of the code.

This study is far from being complete. Power analysis showed a high likelihood of committing a type two error. The chance of not showing any difference between both methodologies, despite the fact that there could be one, is too high. Repetition of this experiment is mandatory in order to enlarge the sample size and thus, to reduce the possibility of a missed chance. A repetition with experienced reviewers and pair programmers would also eliminate the discussed aspects of generalisability. Another reason for not finding a noticeable difference between both methods could be the length of the programming tasks because both tasks could have been too short to show the superiority of pair programming.

References

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

[2] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Academic Press, 1977.

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

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

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

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

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

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

[11] L. Williams and R. Kessler. The effects of pair-pressure and pair-learning on software engineering education. In *Conference on Software Engineering Education and Training*, pages 59–65, Austin, Texas, USA, Mar. 2000.

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