A Preliminary Study on the Impact of a Pair Design Phase on Pair Programming and Solo Programming

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

Abstract

The drawback of pair programming is the nearly doubled personnel cost. The extra cost of pair programming originates from the strict rule of extreme programming where every line of code should be developed by a pair of developers. Is this rule not a waste of resources? Is it not possible to gain a large portion of the benefits of pair programming by only a small fraction of the meeting time of a pair programming session? We conducted a study to answer this question. We split the pair programming process into a pair design and a pair implementation session and started to analyze the impact of the pair design session on pair implementation and solo implementation. Our data shows that after pair design took place the solo implementation phase is 21 percent cheaper than the pair implementation phase.

1. Introduction

Pair programming has become more and more popular in the recent years. Developers who work in pairs in front of one display, keyboard, and mouse learn from their partners and share ideas. Project management appreciates the quantitative benefits of pair programming: faster development and higher quality code. However, the advantages of pair programming are bought at the expense of the nearly doubled personnel cost.
Studies trying to identify the quantitative benefit of pair programming treated the pair programming process as a black box. So far, no study has tried to structure the pair programming process. What would be, if the benefits of pair programming could be obtained by only a small fraction of the meeting time of a pair programming session? We focus on a pair design session in which a mental model of a potential solution to a problem is built. We distinguish between the following techniques:

Pair Design In *Pair Design* two developers work together in order to establish a design of the implementation.

Pair Implementation During *Pair Implementation* two developers work together in front of one display, mouse, and keyboard.

Pair Programming is used when we do not want do emphasize the development phase in which the pairing takes place.

In this paper, we present the results of a study analyzing the impact of a pair design session on the cost of pair implementation and solo implementation. In the solo implementation scenario, developers pair off for the design

and split up for implementation. The study was conducted at the University of Karlsruhe in the summer term 2004. Participants were 18 computer science students.
Our data set reveals the following results:

1. If the level of correctness of the developed programs is of no concern, solo programming is 21 percent cheaper than pair programming.

2. If programs of similar level of correctness have to be developed, solo programming and pair programming lead to the same cost.

The first result is significant at the 5 percent level. Although, the second result claims no difference between the data sets, our data sample suggest the solo programming group to be cheaper than the pair programming group. But due to our small data sample, we had only a chance of 37 percent to reveal an effect.

2. Related Work

Williams et al. [9] studied pair programming with 41 undergraduate students. Concerning program correctness, the programs of the pairs passed more of the automated post-development test-cases ($p < 0.01$). 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 percent more programmer hours on the completion of the tasks, the working overhead dropped to 15 percent.
Nosek [8] compared pair programming to solo programming with 15 professional programmers. The experiment task was to implement a database consistency check. All pairs outperformed the individuals. Although the average time for completion was more than 12 minutes (41 percent) longer for individuals, the difference was not statistically significant on the 5 percent level.
Nawrocki and Wojciechowski [7] studied 21 computer science students on the first four assignments of the *Personal Software Process* [3] programming course. Overall, pair programming was not faster than solo programming in sharp contrast to the Williams el al. and the Nosek studies. But the variability within the pair programming group was smaller when compared to the single programmer group.
Müller [5, 4] conducted two experiments to compare pair programming with solo programming and code reviews. Participants were 38 students. The design of the study forced the subjects to develop programs with a similar number of failures. Both techniques are interchangeable with regards to the development cost.

3. Methodology

Williams et al. [9] and Nawrocki and Wojciechowski [7] conducted their studies to evaluate the advantage of pair programming over solo programming, however, the results of their studies are inconclusive to some extent. The results depend not only on the development methods but also on the number of failures, e.g. Williams et al. report on a difference of observed program failures of 15 percent. The number of failures 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 number of failures because of subjects' different attitude on when it is beneficial to stop testing and when not. It is methodologically questionable to compare these program versions.
This study tries to balance the number of failures by an additional acceptance-test phase at the end of the implementation process. This acceptance-test phase ensures similar number of failures 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 acceptance test phase.

4. The Study

The experiment had an inter-subject design and was held at the University of Karlsruhe during the summer lectures 2004. The study was part of an extreme programming course [6]. The course was composed of four 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 after the pair programming introduction. Java and the Eclipse development environment were used 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.
The two groups were called the *pair programming group* (PPG) and the *solo programmer group* (SPG).

4.1. Participants

Eighteen (18) computer science students participated in the study. They had on average 7.5 years programming experience and 3 years programming experience with Java. Although the average is 7.5 years, the actual programming experience is different from student to student. For example, one student already has 16 years programming experience while other students did not program until they started their university studying two years ago. Three subjects tried pair programming prior to the course. Seven subjects reported to have experience with the Eclipse development environment. An introductory course which demonstrated core features of Eclipse was offered to the remaining participants. Pair programming was introduced by extreme programming professionals. The pair programming course took about 1.5 hours.

4.2. Experiment Task

Students had to design and implement a scheduling algorithm and the elevator control of an elevator system. The elevator system resides in a building with multiple floors and multiple elevators. The system distinguishes between three actions:

Request An “up” or “down” button outside the elevators is pressed.

Job The destination floor inside an elevator is chosen.

Lock The door-lock button delays closing of the elevator door while the door is open.

The task of the scheduling algorithm is to assign *Requests* to an elevator. Specification of the scheduler is kept imprecise as the subjects should not rely on a special strategy as long as every elevator is assigned at least one request during simulation. The elevator control consists of a finite automaton with five states. The behavior of the elevator is described by natural language never mentioning the finite automaton explicitly. The tasks involved implementing two methods residing in the elevator system class and the elevator class.

4.3. Experiment Plan

The experiment consisted of three phases: *pre-test*, *design*, and *implementation*. Figure 1 depicts an overview. Purpose of the *pre-test* was to measure programming skills of participants which was used later on to divide the students into the groups.
The two remaining phases are dedicated to the elevator task. All subjects worked in pairs for the *design* phase regardless of the group they belonged to. During the design phase, the pairs had to acquaint themselves with the code base and the task itself by the aid of a template which they had to fill out. At the end of the design, the template was checked for consistency and proper level of detail. Correcting or masking errors was not subject

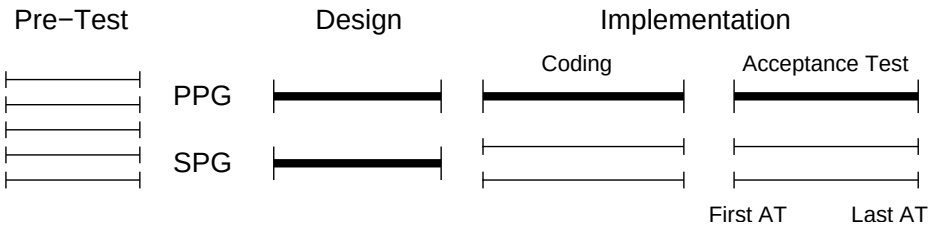


Figure 1. Experiment plan

to the check. The pairs had to rework on the template if inconsistencies or a wrong level of detail were found. Checking of the template was done manually by the experimenter.
For *implementation*, the pairs of SPG were split off. The individuals worked on their own for the remainder of the experiment. The pairs of PPG proceeded unchanged.
Implementation was divided subsequently into *coding* and *acceptance-test* (AT). During *coding*, the subjects had to implement the task until they thought they had finished. At this point, they entered the *acceptance-test* phase where their programs had to pass the acceptance-test. If the programs did not pass, subjects got the output of the failed tests and had to fix the errors. Acceptance-test and rework repeated as long as the program failed the test. Otherwise, subjects finished the task.

4.4. Calculation of Group Sizes

In order to have the best chance to reveal an effect, even with that small number of participants, the number of subjects within each group were selected by the means of power analysis. S_{PPG} and S_{SPG} denote the number of subjects within each group while n_{PPG} and n_{SPG} denote the number of data points for each group. We obtain $n_{PPG} = S_{PPG}/2$ data points from PPG and $n_{SPG} = S_{SPG}$ from SPG. Power analysis requires an equal number of data points per group. But as the number of data points of the two groups are expected to be different, the harmonic mean n is used as a substitute:

$$n = \frac{2 \times n_{PPG} \times n_{SPG}}{n_{PPG} + n_{SPG}}.$$

The power of the t-test (we use the t-test only for power calculation purposes) increases with the size of the harmonic mean all if all other factors (effect size and significance level) are kept constant. The number of data points of both groups must adhere to

$$n_{PPG} \times 2 + n_{SPG} = 18.$$

Table 1 lists the power of the t-test β for different combinations of n_{PPG} and n_{SPG}. The row with the bold figures

Table 1. Power values for different sample sizes of PPG and SPG.

n_{PPG}	n_{SPG}	n	β
7	4	5.09	0.203
6	6	6	0.240
5	**8**	**6.15**	**0.246**
4	10	5.71	0.229

marks the combination of n_{PPG} and n_{SPG} with the highest power. In the beginning, we did not expect the power to be of reasonable height and the total increase in power of 0.6 percent as compared to the row above is only of

Table 2. Group sizes and available data points

group	size	pairs in design	planned data points	available data points
SPG	10	5	5	5
PPG	8	4	8	6

theoretical value. But nevertheless, the application of power calculation for group size estimation in the context of a given and fixed sample size was new to us. Table 2 summarizes group sizes and available data points. We initially planned for 5 data points for PPG and 8 data points for SPG. However, 2 subjects of SPG did not finish coding because of the difficult programming task. The dropouts reduced the number of available data points for SPG by 2.

4.5. Division of Subjects into Groups and Pairs

Division of subjects was done according to their performance in the pre-test. In the pre-test subjects had to write insertion, transformation, and test-methods for a class implementing a binary tree. Figure 2 shows for each subject the length of the pre-test as well as the membership to one of the two groups: P for PPG; S for SPG.

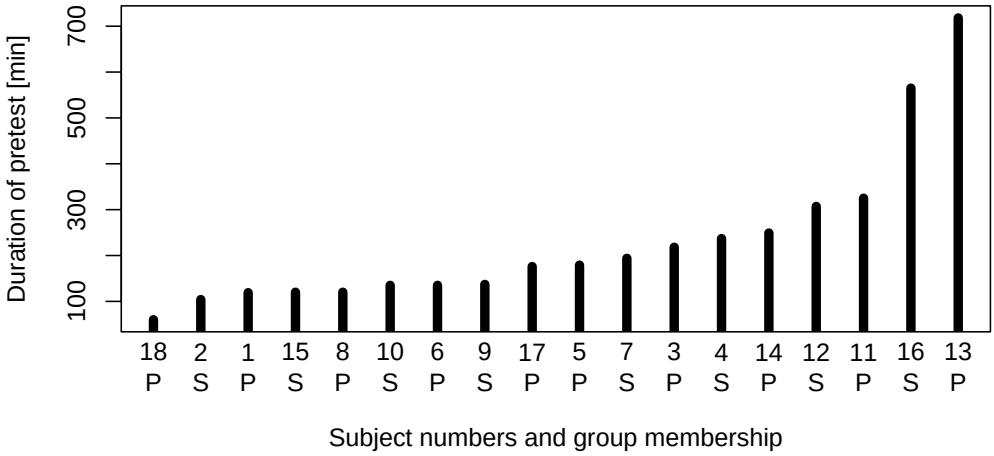


Figure 2. Length of pre-test and group membership for each subject.

There exist two ways to compose pairs for a pair programming experiment. The first way is to pair off friends. These pairs are expected to be most productive as they do not have to become acquainted with their partner. However, this selection method requires that every participant is familiar with somebody. If familiarity is not met, pairs have to be selected randomly in such a way that every pair is made up of two students who do not know each other. We chose the second method because we had some subjects who reported not to know anybody else in the course.

4.6. Acceptance-Test

The acceptance-test is implemented as a web-application. It consists of a simulation environment which applies several scenarios to the implementation under test. For each scenario, the simulation environment records elevators' state transitions and the issued actions. The subjects had to log in, submit their Java-files, and start the test. If a test fails, subjects obtain the corresponding log-file and a message indicating the type of failure and the time step the failure occurred for the first time.
The acceptance-test consists of 50 scenarios which try to provoke distinct kind of failures in the implementation under test. 30 scenarios scrutinize basic elevator functionality by simulating a setting with one elevator. Ten scenarios concentrate on the correct implementation of the door-lock behavior. The remaining scenarios simulate multiple elevators trying to provoke errors in the scheduling algorithm. Finally, the elevator principle is checked for all scenarios.
Another topic concerns the invocation of the acceptance-test. We wanted to ensure that the acceptance-test is executed as seldom as possible. Thus, we decided that only the experimenter should have the right to run an acceptance-test. Therefore, we protected the acceptance-test by a password which was only known by the experimenter. The students in turn had to ask the experimenter to fill in the password for every acceptance-test they wanted to run. With this procedure, we wanted to increase subject's threshold to issue an acceptance-test. But unfortunately, subjects found a way to issue the acceptance-test on their own because of an error in the web-application. But as the acceptance-test for one program under test lasts for about 170 seconds (in the best case), a waiting time of about 3 minutes was large enough for the subjects not to issue a test after every small code change.

4.7. Gathered Data

During the experiment, we measured the length of the phases design $\mathsf{T}_{\mathsf{Design}}$, coding $\mathsf{T}_{\mathsf{Coding}}$, and acceptance-test $\mathsf{T}_{\mathsf{Test}}$. The time for the acceptance-test phase $\mathsf{T}_{\mathsf{Test}}$ contains the execution time of the acceptance-tests. We compared the effort of the acceptance-test phase including test execution time with the effort of the acceptance-test phase without the test execution time. Beside the fact, that the effort was smaller for the second setting, the difference between data sets did not change. Hence, we decided to use the data sets of the acceptance-test phase for analysis which include the test execution time.
The measured periods of time are used to calculate the cost of the phases design, coding, and implementation:

$$\mathsf{Cost}_{\mathsf{Design}}(\mathsf{PPG}) = 2 \times \mathsf{T}_{\mathsf{Design}} \qquad \mathsf{Cost}_{\mathsf{Design}}(\mathsf{SPG}) = 2 \times \mathsf{T}_{\mathsf{Design}}$$
$$\mathsf{Cost}_{\mathsf{Coding}}(\mathsf{PPG}) = 2 \times \mathsf{T}_{\mathsf{Coding}} \qquad \mathsf{Cost}_{\mathsf{Coding}}(\mathsf{SPG}) = \mathsf{T}_{\mathsf{Coding}}$$
$$\mathsf{Cost}_{\mathsf{Imp}}(\mathsf{PPG}) = 2 \times (\mathsf{T}_{\mathsf{Coding}} + \mathsf{T}_{\mathsf{Test}}) \qquad \mathsf{Cost}_{\mathsf{Imp}}(\mathsf{SPG}) = \mathsf{T}_{\mathsf{Coding}} + \mathsf{T}_{\mathsf{Test}}$$

Measures concerning the level of correctness of the programs were gathered after the experiment took place, for example, the number of failures for an acceptance-test is calculated using the logs of the acceptance-test environment.

4.8. Hypotheses

The results of the studies comparing pair programming with solo programming [8, 9, 7] and pair programming with reviews [5, 4] suggest the following null-hypotheses:

$\mathsf{H}_0(\mathsf{Cost}_{\mathsf{Coding}})$: The cost for coding $\mathsf{Cost}_{\mathsf{Coding}}$ is *not* higher for PPG than for SPG.

$\mathsf{H}_0(\mathsf{Cost}_{\mathsf{Imp}})$: The cost for implementation $\mathsf{Cost}_{\mathsf{Imp}}$ is *not* higher for PPG than for SPG.

$\mathsf{H}_0(\mathsf{Failures})$: The number of failures after coding is *not* smaller for the programs of PPG than for the programs developed by SPG.

4.9. Internal Threats

In order to obtain more data points, the pairs of SPG where split off after the design phase and *all* members of SPG had to work on the previous developed design. Thus, during the coding and acceptance-test phases, the solo programmers were not allowed to interact with their pair design partner. However, this limitation does not occur in practice where programmers are used to ask their partner for clarification in the case of problems. Unfortunately, we did not take this situation into account when designing the experiment and thus, we do not know if this problem really occurred.
Choosing the Eclipse development environment for the experiment may have an impact on the outcome of the study because several subjects reported that they had no experience with using Eclipse. But, as these students learned to use Eclipse in the introductory session and the pre-test, the author assumes the impact of Eclipse on the results of the experiment to be negligible.

4.10. External Threats

All participants were students with almost no pair programming experience. The pairs were also made up of students who did not met before: *ad-hoc* pairs. Pairing off with an unknown partner results in a pair which is not as productive as it would be after a phase in which the partners get to know each other. Thus, a replication of the study with well-rehearsed pairs may lead to a reduction of pair programming cost resulting in a smaller difference between both groups. Using ad-hoc pairs is an internal threat, as well.

5. Results

We use box plots to visualize the results of the measurements. The boxes within a plot contain 50 percent of the data points. The lower (upper) border of the box marks the 25 percent (75 percent) 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. Data points are visualized with circles. The median is marked with a thin line. The M associated with the dashes marks the mean value within a range of one standard error on each side. Evaluation bases on the Mann-Whitney two-sample test (p-level of 0.05).

5.1. Cost of Design Phase

Before we analyze the data with respect to the hypotheses, we explore whether the groups have any differences concerning the cost of the pair design phase. The purpose of this comparison is to evaluate whether there is a bias in our data set. As we divided subjects' into groups by the pre-test productivity figures, we expect no difference between the groups. Figure 3 shows the cost for design phase $\mathsf{Cost}_{\mathsf{Design}}$ for both groups. The boxes of SPG and PPG have an almost perfect match. The medians of the groups are 478 man-minutes (mm) for SPG and 490 mm for PPG. The difference between the means (458 mm for SPG vs. 605 mm for PPG) is due to the outlier of 1226 mm in PPG. The mean for PPG without the outlier is 450 mm. The outlier of PPG originates from a pair where one student rated the complexity of the programming task with 2 while the other student rated it as 5 on a five point scale ranging from 1 (very easy) to 5 (very difficult). This pair needed also more time for the implementation than the other pairs, as we will see in 5.2. As a result from the analysis of the design cost, division of subjects into groups led to groups with almost equally skilled pairs.

5.2. Comparison of Cost

The cost of the coding phase and the whole implementation phase is considered. The left plot of Figure 4 shows the cost for the coding phase $\mathsf{Cost}_{\mathsf{Coding}}$. The table on the right of Figure 4 accounts for the p-values of the comparison of SPG with PPG once with and once without the outlier of PPG.

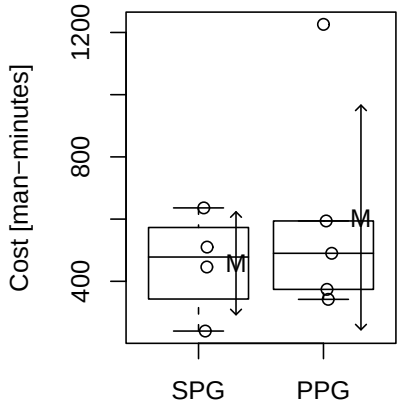
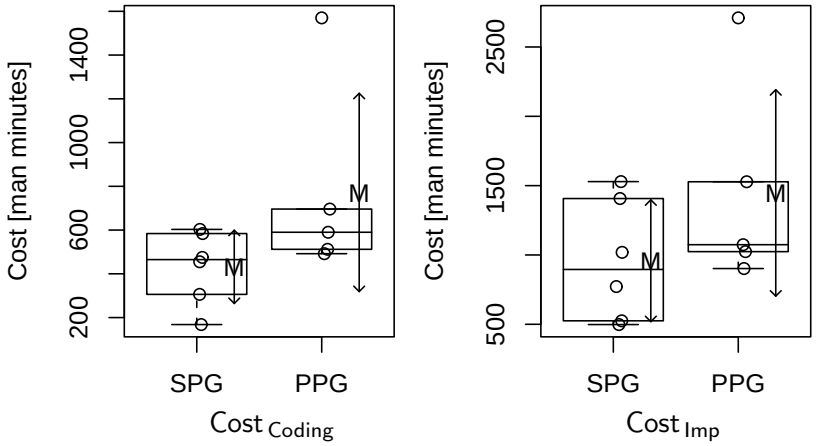


Figure 3. Cost for pair design phase in man-minutes



Mann-Whitney p-values of
the comparison of SPG vs. PPG

data set	with outlier	without outlier
$\mathsf{Cost}_{\mathsf{Coding}}$	0.041	0.086
$\mathsf{Cost}_{\mathsf{Imp}}$	0.123	0.238

Figure 4. Cost for coding and implementation as well as p-values of the Mann-Whitney test.

Half of the single developers in SPG were cheaper than the cheapest pair of PPG. The difference between the data sets is supported by a p-value of 0.041 for the one-sided Mann-Whitney test. The medians of the groups are 465 mm for SPG and 590 mm for PPG (means: 432 mm SPG vs. 772 mm PPG). Comparing the medians, SPG is about 21 percent cheaper. Calculating the on average advantage of SPG compared to PPG is meaningless because the outlier in PPG has too large an impact on the mean of PPG for a comparison to be fair. If we had omitted the outlier, the on average advantage would have been 32 percent. This advantage is not significant on the 5 percent level ($\mathsf{p} = 0.086$) any more. The observed difference is not caused by the different implementation techniques alone but also due to the different number of failures as it is discussed in 5.3.

The plot in the middle of Figure 4 shows the cost of the implementation phase $\mathsf{Cost}_{\mathsf{Imp}}$. The cost difference between both groups is not as large as for the design phase. The smaller difference is also supported by a higher p-value of the Mann-Whitney test of 0.1234 as compared to the p-value (0.041) for the coding phase. This difference is even smaller if the outlier of PPG is omitted from the data set. In this case, the difference is 23.8 percent due to chance. The cost analysis of the implementation phase compares the effort needed to develop programs with similar number of failures. Thus, the observed difference is mainly caused by the different implementation

methods. Although the plot in the middle of Figure 4 shows a trend to favor SPG over PPG data sets are too small for a conclusive result.
Cost analysis lead to the following two results:

1. If level of correctness of developed programs is of no concern, solo programming can be cheaper than pair programming. Comparison of the medians of both groups yields a 21 percent cost reduction of solo programming as compared to pair programming.

2. If programs of similar level of correctness have to be developed, there might be no difference between both groups.

The first result is not only in line with our expectations it complies also to the results of others [8, 9, 7] who compared solo programming with pair programming. The second result may be caused by the small power of the test or because there is no difference. We should repeat this study with a larger data sample.

5.3. Comparison of Failures

Figure 5 shows the number of program failures after coding. The box containing 50 percent of the data points of

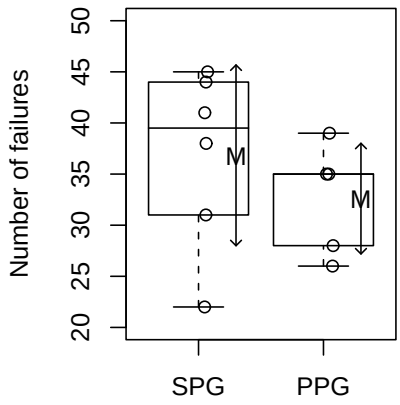


Figure 5. Number of program failures at first acceptance test

PPG is shifted towards the lower end of the SPG box. The medians of both groups are 40 failures per program (fpp) for SPG and 35 fpp for PPG (means: 37 fpp for SPG vs. 33 fpp for PPG). When we take the medians into account the programs of SPG show 14 percent more failures than the programs of PPG. The one-sided Mann-Whitney test reveals a p-value of 0.1571.
Instead on focusing only on the number of failures, we were also interested in the different types of failures. The barplot of Figure 6 shows program failures categorized by failure type. Table 3 describes the used failure codes. As the acceptance test contained 50 different scenarios each program could contribute 50 failures to the data set, in the worst case. The first number of the failure code indicates the severity of the failure in ascending order. While a leading 0 indicates minor failures, failures starting with a 3 indicate an application crash.
Looking at the barplot, the programs of SPG seem to have problems with the door timer. 5 out of 6 programs of SPG close the elevator doors too early, see failure 108. 4 programs of SPG close the elevator doors too late, see failure 109. The programs of PPG seem to have more problems with the driving timer. Either the elevator waited

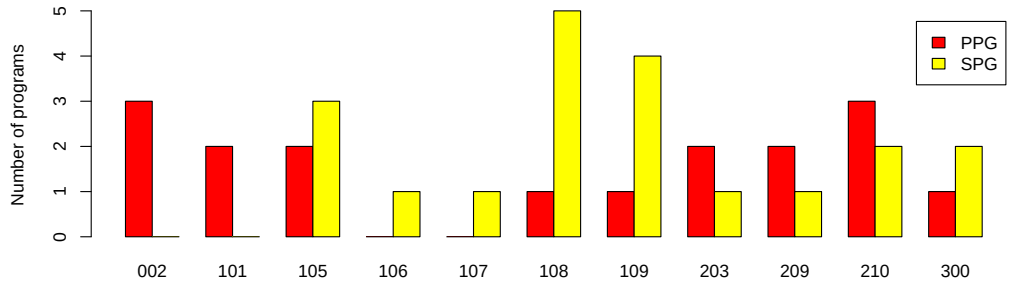


Figure 6. Comparison of type of program failures for PPG and SPG

Table 3. Failure classification

Failure	Description
002	Job finished too late
101	Illegal speed; job finished too early
105	All elevators halted although jobs exist
106	Elevator principle violated
107	Elevator's floor number changed in the direction opposite to the driving direction
108	Elevator too short in state OPEN
109	Elevator too long in state OPEN
203	Illegal floor number; floor number does not exist
209	Job with illegal state
210	Door lock must not be assigned in state LOCKED
300	Corrupt log file

for too long between a floor change-over (failure 002) or floor change-over was too fast (failure 101). 2 programs of SPG and 1 program of PPG crashed.

We also studied the frequencies the different kind of failures occur. The barplot of Figure 7 shows the data sets. Two programs of SPG crashed 67 times, see the SPG bars above failure 300 of Figure 6 and 7. 5 programs of SPG lead to 83 failure situations where the elevator door is closed too early. The PPG programs show failure 105 (elevators halted although jobs exist) and 203 (illegal floor number) most often.

To conclude this analysis, solo programmers seemed to have most problems with the timing of the elevator doors. Failures of the programs developed by the programmer pairs seem to be caused by problems concerning the movement of the elevators.

We also studied whether the programs of the solo programmers who paired off for the design phase show the same types of failures. Figure 8 shows the comparison of pair 7 and pair 9. The other two pairs of SPG could not be analyzed because these were the pairs where one subject did not finish coding. The failures of the programs developed by the subjects of pair 9 are rather different. But subjects of pair 7 seem to share a wrong mental model because their programs contribute to all program crashes of SPG.

To sum up, programmer pairs tend to develop programs which show fewer number of failures. Concerning the type of failures shown by our data set, programs developed by programmer pairs might show different kind of failures than the programs developed by the solo programmers. The failures of the SPG programs are formally more severe than the failures of the PPG programs. However, moving an elevator to a non existing floor is harmful, as well. And last, developers' mental model built in the pair design session has a positive as well as a negative impact on the developed program. And again, the results of this analysis have to be seen in the light of a small data sample. Further studies with a larger sample size should yield more conclusive results.

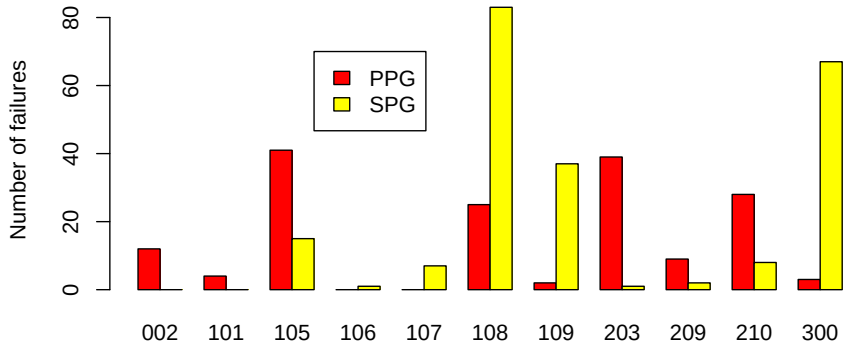


Figure 7. Failure frequency comparison for PPG and SPG

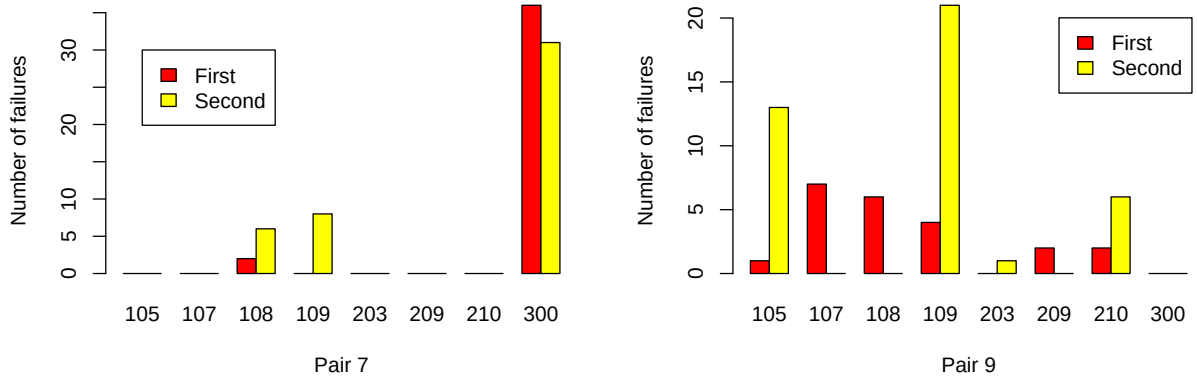


Figure 8. Failure frequency comparison after coding for solo developers who paired off for the design phase

5.4. Sample Size Calculation

Another incentive of this preliminary study was to estimate the expected effect size for the main study which is planned to be conducted with professional engineers. The effect size is used to calculate the sample sizes of PPG and SPG of the main study. Sample size calculation bases on the parameters power β, significance level α, and effect size *ES*. We assume standard values for the power and the significance level, i.e. $\beta = 0.8$ and $\alpha = 0.05$. In order to estimate the number of professionals needed for the study, the effect size has to be calculated from our data set using the equation

$$\mathsf{ES} = \frac{\overline{\mathsf{Cost}_{\mathsf{Imp}}(\mathsf{PPG})} - \overline{\mathsf{Cost}_{\mathsf{Imp}}(\mathsf{SPG})}}{\mathsf{s}} = \frac{1132 - 958}{374} = 0.47$$

where $\overline{\mathsf{Cost}_{\mathsf{Imp}}}$ denotes the average cost of the implementation phase and s is the standard deviation of the pooled data sets. The values used in the calculation represent the PPG data set without the outlier. An effect size of 0.47 is almost as large as a medium-sized effect (0.5) of Cohen's [1, p. 26] effect size measures. Using power tables of the one-sided two-sample t-test we obtain 57 data points per group. If the Mann-Whitney test instead of the t-test is used for evaluation, group sizes have to be adjusted to account for the 15.4 percent weaker asymptotic relative efficiency of the Mann-Whitney test as compared to the t-test [2, pp. 139]. Thus, the previous calculated sample

size has to be increased by 15.4 percent which leads to a group size of 66 data points. Finally, if a study should have a chance of 80 percent to detect with the t-test a cost difference between PPG and SPG for the implementation phase on a 5 percent level, the groups should be made up of 114 participants for PPG and 57 subjects for SPG.

6. Conclusions

We presented the results of a study which analyzed the impact of pair design on the cost of pair implementation and solo implementation. The study was conducted at the University of Karlsruhe with 18 computer science students. Our data set suggests the following preliminary results:

1. Concerning the cost for developing programs with different level of correctness, solo programming is 21 percent cheaper than pair programming. This difference is significant at the 5 percent level.

2. Solo programming is as expensive as pair programming if programs of similar level of correctness have to be developed

Our small data set suggest no difference for the second result. To increase the likelihood of revealing an effect to 80 percent the study should be repeated using 57 data points per group.
We also analyzed the type of program failures of the developed programs. Solo programmers seem to have different problems than developer pairs. Naturally, this result is preliminary due to the small data set and it should be based on an analysis of program faults rather than on the observed program failures.
Concerning development cost, our data suggests a pair design phase to be a potential alternative to the expensive pair programming process. However, a pair design phase does not solve every problem as even two developers are not immune to a wrong mental model. But the probability of building a wrong solution might be much lower for a developer pair than for a single programmer.

Acknowledgments

The author would like to thank Henning Brechmacher for supervising the students during the experiment and Guido Malpohl for proofreading a previous version of this article.

References

[1] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Academic Press, 1988.
[2] M. Hollander and D. Wolfe. *Noparametric Statistical Methods*. John Wiley & Sons, 2nd edition, 1999.
[3] W. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.
[4] M. Müller. Two controlled experiments concerning the comparison of pair programming to peer review. *Journal of Systems and Software (JSS)*, 05. To appear.
[5] M. Müller. Are reviews an alternative to pair Programming ? *Journal on Empirical Software Engineering*, 9(4):335–351, 2004.
[6] M. Müller, J. Link, R. Sand, and G. Mahlpohl. Extreme programming in curriculum: Experiences from academia and industry. In *International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2004)*, number 3092 in Lecture Notes in Computer Science, pages 294–302, Garmisch-Partenkirchen, Germany, June 2004. Springer.
[7] J. Nawrocki and A. Wojciechowski. Experimental evaluation of pair programming. In *European Software Control and Metrics (Escom)*, London, UK, 2001.
[8] J. Nosek. The case for collaborative programming. *Communications of the ACM*, 41(3):105–108, Mar. 1998.
[9] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair-programming. *IEEE Software*, pages 19–25, July/Aug. 2000.