Computing and Informatics, Vol. 29, 2010, 73–91

EXPLORATORY COMPARISON OF EXPERT AND NOVICE PAIR PROGRAMMERS

Andreas Höfer

Universität Karlsruhe (TH)
Fakultät für Informatik
Am Fasanengarten 5
D-76131 Karlsruhe, Germany
e-mail: andreas.hoefer@kit.edu

Revised manuscript received 16 October 2009

Abstract. We conducted a quasi-experiment comparing novice pair programmers to expert pair programmers. The expert pairs wrote tests with a higher instruction, line, and method coverage than the novice pairs and changed the given program skeleton to a larger extent. However, the expert pairs were also slower than the novice pairs. The pairs within both groups switched keyboard and mouse possession frequently. Furthermore, most pairs did not share the input devices equally but rather had one partner who is more active than the other.

Keywords: Pair programming, experts and novices, quasi-experiment

Mathematics Subject Classification 2000: 68N99, 68N19, 68M15

1 INTRODUCTION

Pair programming has been investigated in several studies in recent years. The experience of the subjects with pair programming in these studies varies widely: On the one extreme are novices with no or little pair programming experience who have just been trained in agile programming techniques, on the other extreme are experts with several years of experience with agile software development in industry. It seems rather obvious that expertise has an effect on the pair programming process and therefore on the outcome of a study comparing pair programming to some

other technique. Yet, the nature of the differences between experts and novices has not been investigated so far. Nevertheless, knowing more about these differences is interesting for the training of agile techniques as well as for the assessment of studies on this topic. This study presents an exploratory analysis of the data of nine novice and seven expert pairs, exposing differences between the groups as well as identifying common attributes of their pair programming processes.

2 RELATED WORK

When it comes to research on pair programming, a large part of the studies focus on the effectiveness of pair programming. Research on that topic has produced significant results as summarized in a meta-study by Dybå et al. [10]. They analyzed the results of 15 studies comparing pair and solo programming and conclude that quality and duration favor pair programming while effort favors solo programming. Arisholm et al. [1] conducted a quasi-experiment with 295 professional Java consultants in which they examined the effect of programmer expertise and task complexity on the effectiveness of pair programming compared to solo programming. They measured the duration for task completion, effort and correctness of the solutions. The participants had three different levels of expertise, namely junior, intermediate and senior and worked on maintenance tasks on two functionally equivalent Java applications with differing control style. The authors conclude that pair programming is not beneficial in general because of the observed increase in effort. Nevertheless, the results indicate positive effects of pair programming for inexperienced programmers solving complex tasks: The junior consultants had 149 percent increase in correctness when solving the maintenance tasks on the Java application with the more complex, delegated control style.

Other studies have taken an experimental approach to identify programmer characteristics critical to pair success: Domino et al. [9] examined the importance of the cognitive ability and conflict handling style. In their study, 14 part-time students with industrial programming experience participated. Cognitive ability was measured with the Wonderlic Personal Test (WPT), conflict handling style with the Rahim Organizational Conflict Inventory (ROCI-II). The performance of a pair was neither correlated with its cognitive ability nor its conflict handling style. Chao et al. [6] first surveyed professional programmers to identify the personality traits perceived as important for pair programming. They then conducted an experiment with 58 undergraduate students to identify the crucial personality traits for pair success. The experiment yielded no statistically significant results. Katira et al. [16] examined the compatibility of student pair programmers among 564 freshman, undergraduate, and graduate students. They found a positive correlation between the students' perception of their partners' skill level and the compatibility of the partners. Pairs in the freshman course were more compatible if the partners had different Myers-Briggs personality types. Sfetsos et al. [20] present the results of two experiments comparing the performance of 22 student pairs with different Keirsey

temperaments to 20 student pairs with the same Keirsey temperament. The pairs with different temperaments performed better with respect to the total time needed for task completion and points earned for the tasks. The pairs with different temperaments also communicated more than the pairs with the same temperament.

Furthermore, there are several field studies reporting on data from professional programmers, some of them including video analysis of pair programming sessions. None of these studies were designed to produce statistically significant results, but the observations made are valuable, because they show how pair programmers behave in typical working environments. Bryant [3] presents data from fourteen pair programming sessions in an internet banking company, half of which were videotaped. Initial findings suggest that expert pair programmers interact less than pair programmers with less expertise. Additionally, partners in expert pairs showed consistent behavior no matter which role they played, whereas less experienced pair programmers showed no stable activity pattern and acted differently from one another. Bryant et al. [5] studied 36 pair programming sessions of professional programmers working in their familiar work environment. They classified programmers' verbalizations according to sub-task (e.g. write code, test, debug, etc.). They conclude that pair programming is highly collaborative, although the level of collaboration depends on the sub-task. In a follow-up study Bryant et al. [4] report on data of 24 pair programming sessions. The authors observe that the commonly assumed roles of the navigator acting as a reviewer and working on a higher level of abstraction do not occur. They propose an alternative model for pair interaction in which the roles are rather equal. Chong and Hurlbutt [7] are also skeptical about the existence of the driver and navigator role. They observed two development teams in two companies for four months. They state that the observed behavior of the pair programmers is inconsistent with the common description of the roles driver and navigator. Both programmers in a pair were mostly at the same level of abstraction while discussing; different roles could not be observed.

3 STUDY

The following sections describe the study which was motivated by the following research hypotheses:

RH_{time} The expert pairs need less time to complete a task than the novice pairs. This research hypothesis is based on the results from a quasi-experiment comparing the test-driven development processes of expert and novice solo programmers [18] where the experts were significantly faster than the novices.

RH_{cov} *The expert pairs achieve a higher test coverage than the novice pairs.* Like the research hypothesis above, this one is based on the findings in [18].

RH_{chg} *The expert pairs change the given program skeleton to a different extent than the novice pairs.* We thought of two opposing effects of the greater experience of the expert pairs: Their experience could make them less reluctant to change the

given code base if they would find something they dislike, which would result in more changes compared to the novice pairs. The alternative is that they might act more pragmatically and use the least effort possible to solve the task, which would result in less changes.

RH_{conf} *The partners in the expert pairs compete less for the input devices than the partners in the novice pairs.* In our extreme programming lab course, we observed that the students were competing for the input devices. Hence, we thought this might be an indicator for an immature pair programming process.

3.1 Participants

The novice group consisted of 18 computer science students from an extreme programming lab course [19] in which they learned the techniques of extreme programming and applied them in a project week. They participated in the quasi-experiment in order to get their course credits. In the mean, they were in their seventh semester, had about five years of programming experience including two years of programming experience in Java. Six members of the novice group reported prior experience with pair programming, three of them in an industrial project. Only one novice had used JUnit before the lab course, none had tried test-driven development before. For the assignment of the pairs the experimenter asked each novice for three favorite partners and then assigned the pairs according to these preferences. Only pair N6 could not be matched based on their preferences.

The group of experts was made up of 14 professional software developers. All experts came from German IT companies, 13 from a company specialized in agile software development and consulting. One expert took part in his spare time and was remunerated by the experimenter, the others participated during normal working hours, so all experts were compensated. All experts have a diploma in computer science or in business informatics. On average, they had 7.5 years of programming experience in industrial projects including on average five years experience with pair programming, about three years experience with test-driven development, five years experience with JUnit, and seven years experience with Java. The expert pairs were formed based on their preferences and time schedule.

3.2 Task

The pairs had to complete the control program of an elevator system written in Java. The system distinguishes between requests and jobs. A request is triggered if an up or down button outside the elevator is pressed. A job is assigned to the elevator after a passenger chooses the destination floor inside the elevator. The elevator system is driven by a discrete clock. For each cycle, the elevator control expects a list of requests and jobs and decides according to the elevator state which actions to perform next. The elevator control is driven by a finite automaton with four states: going-up, going-down, waiting, and open. The task description contained

a state transition diagram explaining the conditions for switching from one state to another and the actions to be performed during a state switch.

To keep the effort manageable, only the open-state of the elevator control had to be implemented. The pairs received a program skeleton which contained the implementation of the other three states. This skeleton comprises ten application and seven test classes with 388 and 602 non-commented lines of code, respectively. The set of unit tests provided with the program skeleton use mock objects [17, 22] to decouple the control of the elevator logic from the logic that administrates the incoming jobs and requests. However, the mock-object implementation in the skeleton does not provide enough functionality to develop the whole elevator control. Other functionality has to be added to the mock object to test all desired features of the elevator control. Thus, the number of lines of test code may be higher than the number of lines of application code. The mock object also contributes to the line count.

3.3 Realization

Implementation took place during a single programming session. All pairs worked on a workplace equipped with two cameras and a computer with screen capture software [21] installed. All novice pairs and one expert pair worked in an office within the computer science department. For the other expert pairs an equivalent workplace was set up in a conference room situated in their company.

There was an implicit time limit due to the cameras' recording capacity of seven hours. Additionally, the task description states that the task can be completed in approximately four to five hours. Each participant recorded interrupts such as going to the bathroom or lunch breaks. The time logs were compared to the video recordings to ensure consistency.

Apart from pair programming, the participants were asked to use test-driven development to solve the programming task. The pairs had to work on the problem until they were convinced they had an error free solution, which would pass an automatic acceptance test, ideally at first attempt. If the acceptance test failed, the pair was asked to correct the errors and to retry as soon as they were sure that the errors were fixed. One pair in the expert group and one pair in the novice group did not pass the acceptance test after more than six hours of work and gave up.

4 DATA ANALYSIS AND RESULTS

Since the samples are small and we do not know the population's distribution, we decided to use the Wilcoxon-Rank-Sum Test [14, p. 106] for hypothesis testing. Almost all hypotheses could be tested with the one-tailed Wilcoxon-Rank-Sum Test because they have an implicit direction. However, RH_{chg} does not specify a direction and was tested with the two-tailed equivalent. The power of the respective one-tailed

t-Tests at a significance level of 5 percent, a large effect size of 0.8[1] and a harmonic mean of 7.88 is 0.446. The power of the two-tailed t-Test at the same significance level, effect size and harmonic mean is only 0.315. The power of the Wilcoxon-Test is in the worst case 13.6 percent smaller than the power of the t-Test [14, pp. 139]. Thus, the probability of detecting an effect is 38.5 percent for the one-tailed Wilcoxon-Rank-Sum Test and 27.2 percent for the two-tailed version. These probabilities are fairly small compared to the suggested value of 80 percent [8, p. 531]. To sum up, if a difference on the 5 percent level can be shown, everything is fine; but the probability that an existing large effect is not revealed is 61.5 percent in the one- and 72.8 percent in the two-tailed case.

As mentioned before, two pairs did not develop an error free solution. One could argue that the data points of these pairs should be excluded from analysis, because their programs are of inferior quality. Nevertheless, for the evaluations concerning input activity (see Section 4.4) the program quality is of minor importance. Accordingly, the two data points were not removed. Additional p-values, computed excluding the two data points[2], are reported wherever it makes a difference and the two data points are highlighted in all boxplots and tables.

4.1 Time

First of all, we compared the time needed for implementation defined as time span from handing out the task description to the final acceptance test. The initial reading phase, breaks, and the time needed for acceptance tests were excluded afterwards. $\mathrm{RH}_{\mathrm{time}}$ states our initial assumption that the expert pairs need less time than the novice pairs, i.e. $Time_e < Time_n$. Figure 1 depicts the time needed for implementation as boxplots (grey) with the data points (black) as overlay; the empty squares mark the pairs which did not pass the acceptance test. The exact data for these and all other boxplots in this article can be found in Table 1. The boxplots show that there is no support for the initial research hypothesis. Judging by the data rather the opposite seems to be true. Consequently, not the initial research hypothesis but the re-formulated, opposite hypothesis $Time_e > Time_n$ (null-hypothesis: $Time_e \leq Time_n$) was tested. This revealed that the experts were significantly slower than the novices ($p = 0.036$). Omitting the data points from the pairs that did not pass the acceptance test results in an even smaller p-value of 0.015.

4.2 Test Coverage

The test coverage was measured on the final versions of the pairs' programs using EclEmma [11]. The evaluation of test coverage is motivated by $\mathrm{RH}_{\mathrm{cov}}$, which expresses our assumption that the expert pairs write tests with a higher coverage than the novice pairs, i.e. $Cov_e > Cov_n$. The respective null-hypothesis $Cov_e \leq Cov_n$

[1] As defined in [8, p. 26].

[2] With two data points less the power is only 8.4 percent.

Pair	Time Needed for Impl. [min]	Instruction Coverage [%]	Line Coverage [%]	Block Coverage [%]	Method Coverage [%]	Net Test Code Changes [SLOC]	Net Appl. Code Changes [SLOC]	Net Code Changes [SLOC]	Mean Driving Time [min:sec]	Number of Acceptance Tests	Final Acceptance Test Passed?
N1	188	89.0	89.0	87.6	88.5	70	42	112	01:41	1	yes
N2	139	95.1	94.4	90.9	89.9	114	60	174	02:23	1	yes
N3	156	95.8	95.1	90.9	89.2	81	45	126	11:54	1	yes
N4	406	91.1	90.2	82.8	88.9	30	50	80	04:02	3	no
N5	238	96.2	95.7	91.2	89.9	103	26	129	02:19	3	yes
N6	262	95.0	94.6	89.8	89.7	50	38	88	02:37	4	yes
N7	261	96.5	95.9	91.9	90.8	268	79	347	03:46	1	yes
N8	160	85.9	85.5	87.3	87.5	139	44	183	03:07	2	yes
N9	150	95.8	95.3	91.0	90.6	68	41	109	02:02	1	yes
E1	286	97.0	96.1	92.9	90.2	253	61	314	02:37	1	yes
E2	241	97.3	96.3	92.7	90.4	183	37	220	04:51	1	yes
E3	368	96.6	96.2	92.4	90.9	212	71	283	02:16	2	no
E4	247	95.7	95.5	91.3	90.6	87	91	178	02:15	2	yes
E5	219	96.0	95.5	90.9	89.6	185	56	241	06:13	2	yes
E6	305	94.0	93.2	85.8	91.1	158	84	242	04:35	1	yes
E7	297	95.8	95.4	90.5	90.1	131	52	183	05:48	1	yes

Table 1. Raw data set for all boxplots

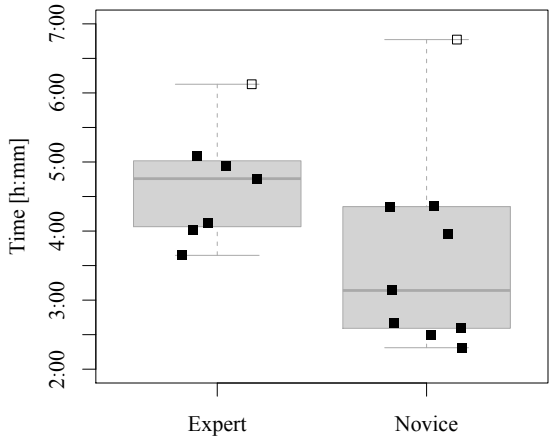

Fig. 1. Time needed for implementation

was tested for instruction, line, block, and method coverage. For instruction, line, and method coverage the null-hypothesis can be rejected on the 5 percent level with p-values of 0.045, 0.022, and 0.025. For block coverage the result is not statistically significant ($p = 0.084$). If we omit the pairs which did not successfully pass the acceptance test we can still observe a trend in the same direction. However, none of the results is statistically significant anymore. The p-values for instruction, line, block, and method coverage are 0.135, 0.068, 0.238, and 0.077, respectively. Figure 2 shows the boxplots for the line and method coverage of the two groups. The dashed line indicates the test coverage of the program skeleton initially handed out to the pairs.

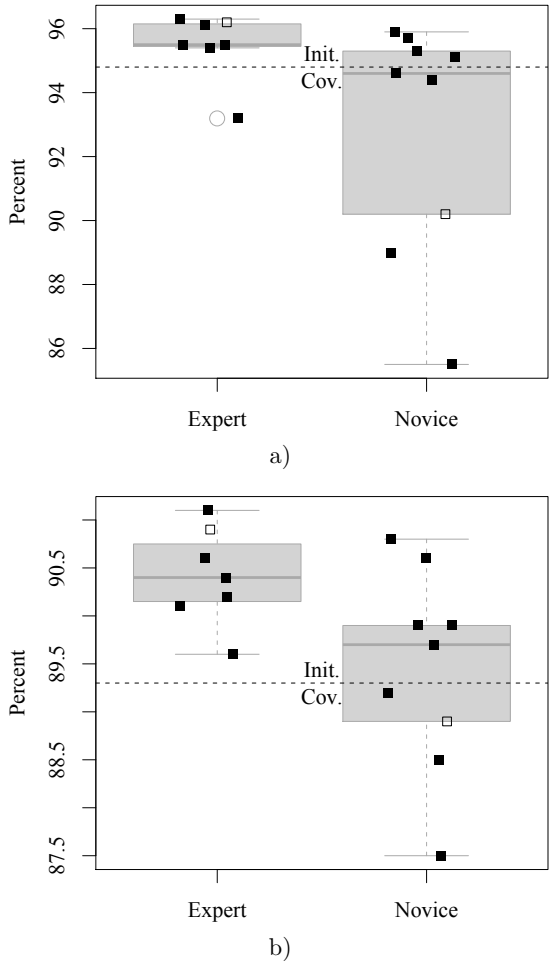

Fig. 2. Test coverage; a) Lines, b) Methods

Looking at the test coverage, it seems that the experts had sacrificed speed for quality. Yet, the costs for the extra quality are high: In the mean, the expert pairs worked more than one hour longer than the novice pairs to achieve a 2.6 percent higher line coverage. Perhaps they also took the acceptance test more seriously than the novices and tested longer before handing in their programs. Figure 3 shows the number of acceptance tests the pairs executed. The numbers of above the bars indicate the absolute number of pairs with the correspondent number of acceptance tests. There is only a slight difference visible between the distributions of the two groups which might be simply due to chance. Consequently, we cannot tell whether the assumption that the expert pairs took the acceptance test more seriously than the novice pairs is true or false.

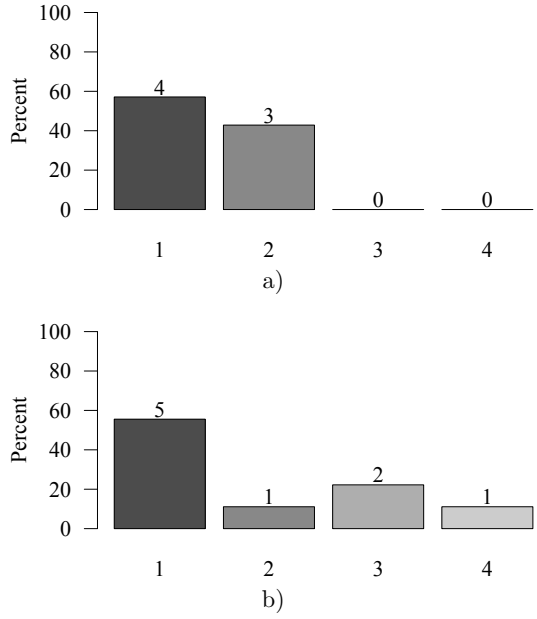

Fig. 3. Number of acceptance tests; a) Expert, b) Novice

4.3 Code Changes

In $\mathrm{RH}_{\mathrm{chg}}$, we formulated our assumption that the expert pairs change the program to a different extent than the novice pairs, i.e. $Chg_e \neq Chg_n$. To examine $\mathrm{RH}_{\mathrm{chg}}$, we counted the number of lines that had changed from the initial to the final version of the pairs' programs. This was done as follows: First, we standardized the formatting of all Java files using Jalopy. Additionally, we removed all empty lines and lines containing import statements from the files. Hence, re-ordering of methods, changes made to white-space characters and import statements, as well as changes

in comments had no effect on the number of changed lines. Finally, we used the Unix diff-command to count the changed lines. Of all pairs, only one pair had created new Java files by introducing two new classes. In this case, we added all lines in the new Java files to the total number of changed lines.

Testing the null-hypothesis $Chg_e = Chg_n$ showed that the difference visible in Figure 4 is statistically significant ($p = 0.017$). This difference remains significant if the data points from the pairs that did not pass the acceptance test are excluded from the analysis ($p = 0.039$).

The fact that the expert pairs changed the code to a larger extent than the novice pairs might explain a vast part of the additional time they consumed. However, we do not know why the expert pairs felt that they needed to change more code than the novice pairs. The only way to answer the question will be to perform manual code reviews and further analysis of the recorded videos.

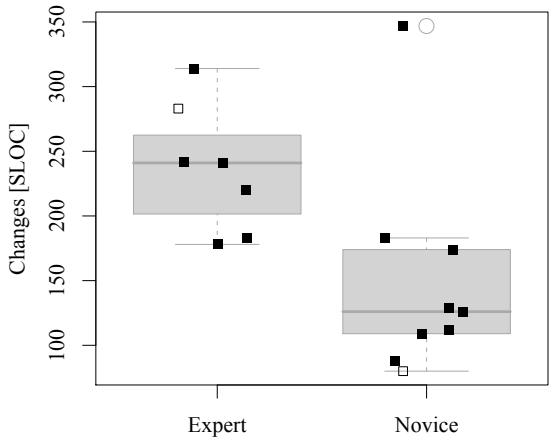

Fig. 4. Changed lines of code

4.4 Measures of Input Activity

Books for extreme programming practitioners mention two different roles when it comes to describing the interaction of the two programming partners and their basic tasks in a pair programming session [2, 15, 23]. Williams and Kessler [24] provide the most commonly used names for these roles: driver and navigator. Even though, the descriptions of the driver and navigator role in these textbooks differ marginally, all agree upon one basic feature of the driver role: The driver is responsible for implementing and therefore uses the keyboard and the mouse. Assuming that this is true, the use of mouse and keyboard by the two partners should make it possible to conclude how long one of the partners stays driver until the two partners switch roles.

4.4.1 Input Device Control and Conflict

We observe the time a programmer touches the keyboard and/or the mouse. Having control of the input devices does not necessarily mean the programmer is really using it to type or browse code. Yet, because the pairs worked on a machine with one keyboard and one mouse possession of keyboard and/or mouse is a hindrance for the other programmer to use them and thus to become the driver. If one partner touches the keyboard while the other partner still has control of it, the time span where both partners have their hands on the keyboard is measured as conflict. Grabbing the mouse while the other partner has control of the keyboard is measured as conflict as well, assuming that the Eclipse IDE [12] (which was used for the task) requires keyboard and mouse for full control over all features.

To obtain the measure of input device control, we transcribed the videos of the programming sessions with separate keyboard and mouse events for each programmer. We used a video transcription tool developed by one of our students especially for the purpose of pair programming video analysis [13].

$\mathrm{RH}_{\mathrm{conf}}$ phrases our initial assumption that the novice pairs spend more time in a conflict state than the expert pairs because they are less experienced in pair programming and do not have a protocol for changing the driver and navigator role; but this assumption could not be confirmed. Only three pairs spent more than one percent of their working time in a conflict state. One of them is in the expert group[3] and two are in the novice group.

4.4.2 Pair Balance

Figure 5 depicts the results from the analysis of input device control. It shows that the majority of the observed pairs did not share keyboard and mouse equally. To make this phenomenon measurable, pair balance b was computed from the input device control as follows:

$$b = \frac{\min(t_1, t_2) + \frac{1}{2}t_c}{\max(t_1, t_2) + \frac{1}{2}t_c} \tag{1}$$

The variables t_1 and t_2 are the times of input device control of the two partners, and t_c the time spent in a conflict state. The values for pair balance may range between zero and one, where one designates ideal balance. A pair balance of less than 0.5 means that the active partner controlled the input devices more than twice as long as the passive partner. Six out of nine novice pairs have a pair balance of less than 0.5; input device control is almost completely balanced in one pair only. In the expert group only one pair has a pair balance of less 0.5, but this pair is the most imbalanced of all. Table 2 shows the exact values for all pairs together with the percentage of conflicts. To check how the participants perceived pair balance, they were asked to rate the statement "Our activity on the keyboard was equal." in

[3] This is the expert pair that did not pass the acceptance test.

the post-test questionnaire[4] on a Likert scale from 1 (totally disagree) to 5 (totally agree). Figure 6 displays the distributions of the replies for both groups. The numbers above the bars indicate the absolute number of replies for the respective level of the Likert scale. The participants' reactions on that statement are not correlated to the corresponding pairs' balance values (tested with Kendall's rank correlation test, $\tau = 0.142$, $p = 0.324$). Their perception seems to differ from reality here.

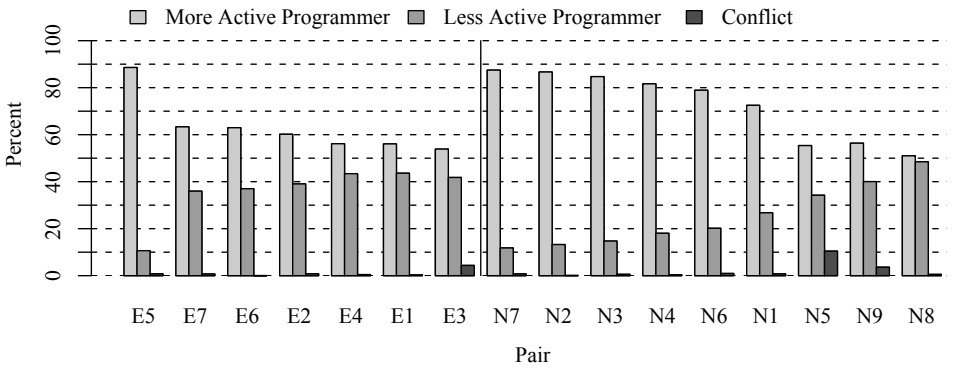

Fig. 5. Input Device Control

4.4.3 Driving Times

Based on the assumption that one programmer remains driver until the other programmer takes control of the keyboard and/or mouse, driving times were computed from the keyboard and mouse transcripts. The driving time is the time span from the point a programmer gains exclusive control over the keyboard and/or the mouse to the point where the other programmer takes over. This time span includes time without activity on the input devices. In case of conflict, the time is added to the driving time of the programmer who had control before the conflict occurred. Further video analysis could help identify the driver during those times; but since at least 90 percent of the working time is free of conflicts the driving times should be precise enough. Figure 7 shows a boxplot of the mean driving times of all pairs[5]. The average driving time of all participants is below four minutes. The pairs switched keyboard and mouse control frequently. At first, the high switching frequency seemed rather odd, but this finding is in line with observations made by Chong and Hurlbutt [7] on a single team of professional programmers working on

[4] Unfortunately, one expert pair had to leave before filling out the post-test questionnaire.

[5] Pair N3, represented by the outlier in the novices' boxplot, had a phase of more than 100 minutes where one programmer showed absolutely no activity on the input devices. This biased the mean.

Pair	Balance	Conflict [%]
N1	0.37	0.75
N2	0.15	0.10
N3	0.18	0.57
N4*	0.22	0.33
N5	0.65	10.42
N6	0.26	0.94
N7	0.14	0.75
N8	0.95	0.54
N9	0.72	3.60
E1	0.78	0.33
E2	0.65	0.79
E3*	0.78	4.36
E4	0.77	0.47
E5	0.12	0.78
E6	0.59	0.03
E7	0.57	0.71

*Did not pass acceptance test.

Table 2. Balance and conflict

machines with two keyboards and mice. They state that within this team programming partners switched keyboard control frequently and rapidly. In an exemplary excerpt from a pair programming session in [7], the partners switched three times within two and a half minutes.

5 THREATS TO VALIDITY

Apart from the different expertise in pair programming of the expert and novice pairs other possible explanations for the observed differences in the data set might exist. The novices also have less general programming experience and experience with test-driven development than the experts. Another threat to validity results from the fact that this study is a quasi-experiment and almost all experts came from one company: Thus, the outcome may also be affected by selection bias.

Furthermore, the pairs might not have shown their usual working behavior because of the experimental setting and the cameras. The participants had to rate the statement "I felt disturbed and observed due to the cameras" on a Likert scale from 1 (totally disagree) to 5 (totally agree). Figure 8 displays the distributions of the participants' ratings. In general, the cameras were not perceived as disturbing, although it seems as if they are a bigger source of irritation for the novices than for the experts. Another reason for unusual working behavior might be that the participants were not accustomed to pair programming and therefore could not pair effectively; but we think that this is unlikely because the experts were used to pair and the novices had been trained to pair in the

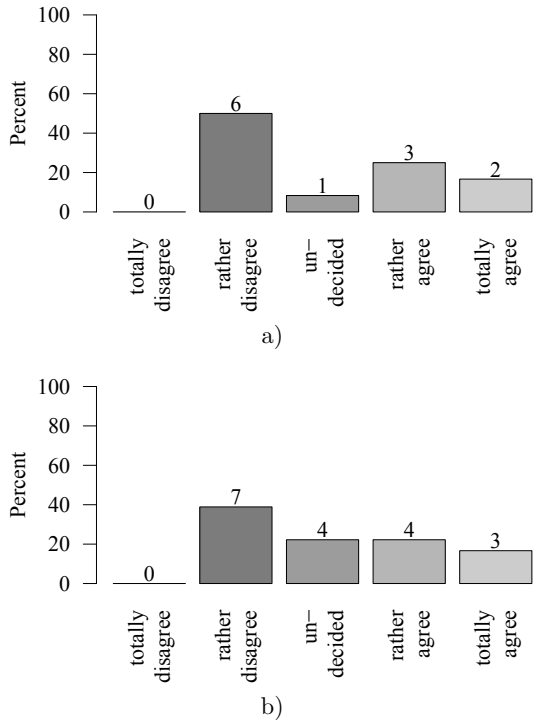

Fig. 6. Replies to "The activity on the keyboard was equal"; a) Expert's replies, b) Novice's replies

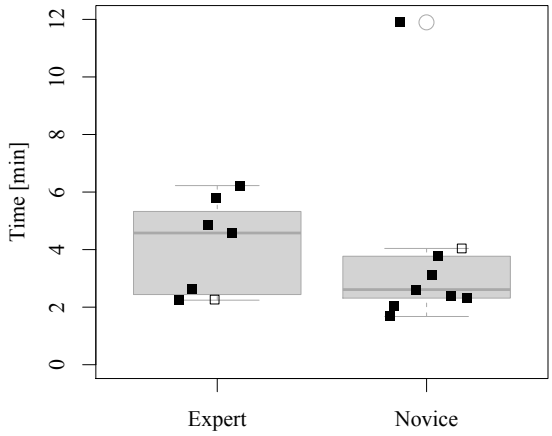

Fig. 7. Mean driving time of the pairs

project week of our extreme programming lab course shortly before the quasi-experiment.

Moreover, the fact that experts were paid for their participation and novices were not might have lead to a bias in motivation. Figure 9 shows the distributions of replies on the statement "I enjoyed programming in the experiment". The experts' distribution of replies seems to be shifted to the right compared to the novices' one which might indicate a higher motivation of the experts. But as the data set is small, this difference is not statistically significant. The participant's motivation might also be influenced by how well the partners got along with each other. Figure 10 summarizes the ratings of the experts and novices of the statement "I would work with my partner again". As before, the experts' distribution appears to be shifted to the right compared to the novices' one. Yet again, this difference is not statistically significant, due to the small size of our data set.

Finally, the task was used in other studies before so some participants might have known the task. Consequently, we asked the participants if they already knew the task before they started. All participants answered the question with no.

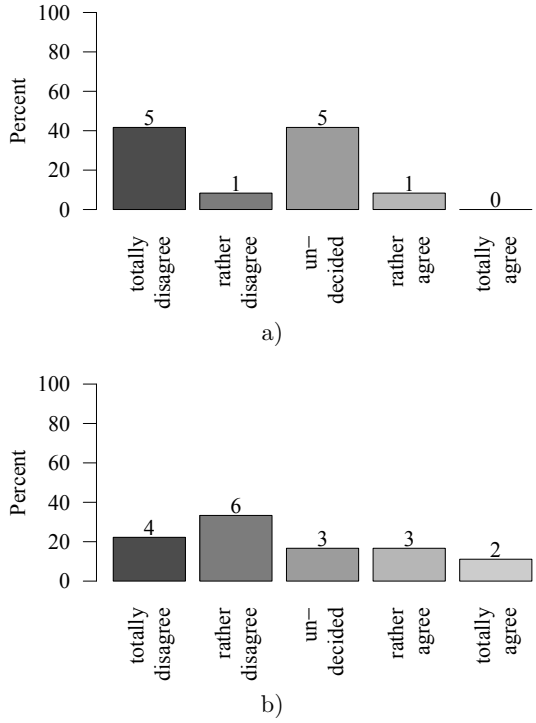

Fig. 8. Replies to "I felt disturbed and observed by the cameras"; a) Expert's replies, b) Novice's replies

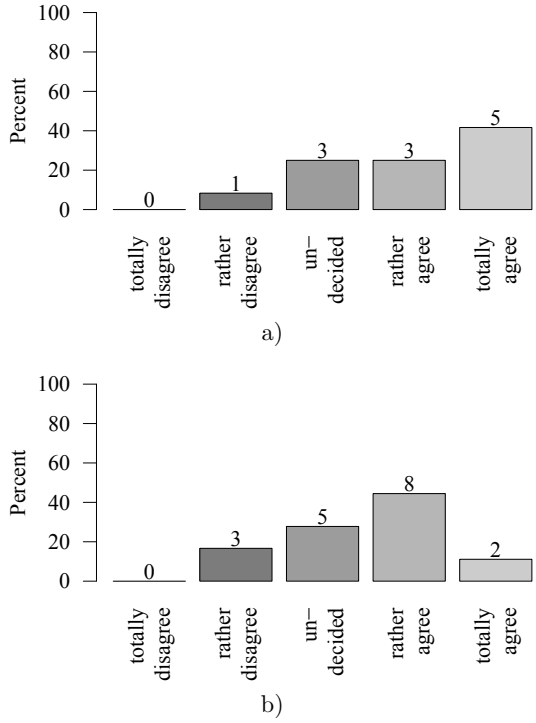

Fig. 9. Replies to "I enjoyed programming in the experiment"; a) Expert's replies, b) Novice's replies

6 CONCLUSIONS AND FUTURE WORK

This article presented an exploratory analysis of a data set of nine novice and seven expert pairs. The expert pairs had changed the code to a larger extent than the novice pairs and they had written better tests in terms of instruction, line and method coverage. In return the expert pairs were significantly slower than the novice pairs. The most important implication of the observed differences is that generalization of studies with novices remains difficult. Also, the direction of the difference is not necessarily the one predicted under the common assumption "experts perform better than novices". In order to determine the reason why the expert pairs were slower than the novice pairs two things have to be done next: First, further analysis of the recorded video could indicate where the experts lost time. Second, we need to check whether the experts adhered more rigidly to the test-driven development process than the novices, which might be time consuming. We will do this with the revised version of our framework for the evaluation of test-driven development initially presented in [18].

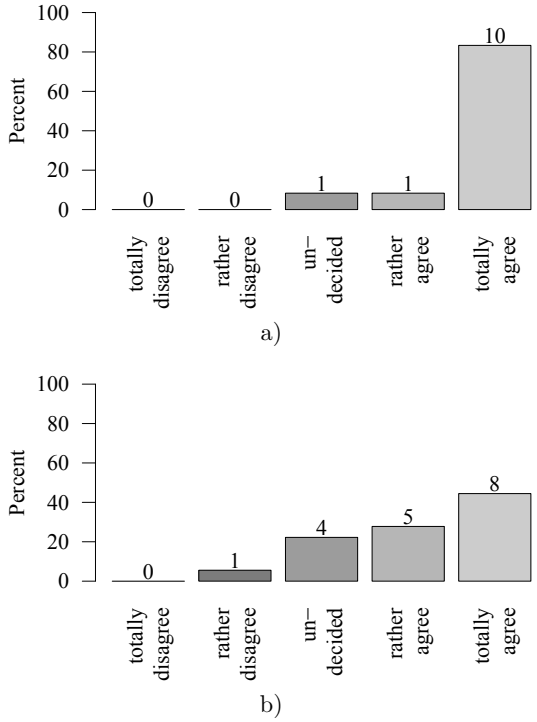

Fig. 10. Replies to "I would work with my partner again"; a) Expert's replies, b) Novice's replies

The analysis of input activity revealed no significant differences between the groups. Nevertheless, it revealed that the roles of driver and navigator change frequently and that a majority of the pairs has one partner dominating input device control. The question what the less active partner did still needs to be answered. Analyzing the existing video material, focusing on the verbalizations of the programming partners, should help answer this question.

Acknowledgements

The study and the author were sponsored by the German Research Foundation (DFG), project "Leicht" TI 264/8-3. The author would like to thank Sawsen Arfaoui for her help on the video transcription and the evaluation of the questionnaires.

REFERENCES

[1] Arisholm, E.—Gallis, H.—Dybå, T.—Sjøberg, D. I. K.: Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. IEEE Transactions on Software Engineering, Vol. 22, February 2007, No. 2, pp. 65–86.

[2] Beck, K.: Extreme Programming Explained: Embrace Change. 1st edition. Addison-Wesley, Reading, Massachusetts, USA, 2000.

[3] Bryant, S.: Double Trouble: Mixing Qualitative and Quantitative Methods in the Study of eXtreme Programmers. In: IEEE Symposium on Visual Languages and Human Centric Computing, September 2004, pp. 55–61.

[4] Bryant, S.—Romero, P.—du Boulay, B.: Pair Programming and the Mysterious Role of the Navigator. International Journal of Human-Computer Studies, Vol. 66, 2008, No. 7, pp. 519–529.

[5] Bryant, S.—Romero, P.—du Boulay, B.: The Collaborative Nature of Pair Programming. In: Extreme Programming and Agile Processes in Software Engineering, Vol. 4044/2006 of Springer Lecture Notes in Computer Science, 2006, pp. 53–64.

[6] Chao, J.—Atli, G.: Critical Personality Traits in Successful Pair Programming. In: Proceedings of Agile 2006 Conference, 2006, pp. 65–68.

[7] Chong, J.—Hurlbutt, T.: ICSE '07 The Social Dynamics of Pair Programming. In: Proceedings of the 29th International Conference on Software Engineering, 2007, pp. 354–363.

[8] Cohen, J.: Statistical Power Analysis for the Behavioral Sciences. 2nd edition. Lawrence Erlbaum Associates, 1988.

[9] Domino, M. A.—Collins, R. W.—Hevner, A. R.—Cohen, C. F.: Conflict in Collaborative Software Development. In: SIGMIS CPR '03: Proceedings of the 2003 SIGMIS Conference on Computer Personnel Research, 2003, pp. 44–51.

[10] Dybå, T.—Arisholm, E.—Sjøberg, D. I. K.—Hannay, J. E.—Shull, F.: Are Two Heads Better than One? On the Effectiveness of Pair Programming. IEEE Software, Vol. 24, November/December 2007, No. 6, pp. 12–15.

[11] EclEmma Project web site. Available on: `http://www.eclemma.org`.

[12] Eclipse IDE web site. Available on: `http://www.eclipse.org`.

[13] Höfer, A.: Video Analysis of Pair Programming. In: APSO '08: Proceedings of the 2008 international workshop on scrutinizing agile practices or shoot-out at the agile corral, Leipzig, Germany, 2008, pp. 37–41.

[14] Hollander, M.—Wolfe, D. A.: Nonparametric Statistical Methods. 2nd edition. Wiley Interscience, 1999.

[15] Jeffries, R. E.—Anderson, A.—Hendrickson, C.: Extreme Programming Installed. Addison-Wesley, 2001.

[16] Katira, N.—Williams, L.—Wiebe, E.—Miller, C.—Balik, S.—Gehringer, E.: On Understanding Compatibility of Student Pair Programmers. SIGCSE Bulletin, Vol. 36, 2004, No. 1, pp. 7–11.

[17] Mackinnon, T.—Freeman, S.—Craig, P.: Endo-testing: Unit Testing with Mock Objects. In: Succi, G. and Marchesi, M. (Eds.): Extreme Programming Ex-

amined, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001, pp. 287–301.

[18] Müller, M. M.—Höfer, A.: The Effect of Experience on the Test-Driven Development Process. Empirical Software Engineering, Vol. 12, December 2007, No. 6, pp. 593–615.

[19] Müller, M. M.—Link, J.—Sand, R.—Malpohl, G.: Extreme Programming in Curriculum: Experiences from Academia and Industry. In: Extreme Programming and Agile Processes in Software Engineering. Vol. 3092/2004 of Springer Lecture Notes in Computer Science, June 2004, pp. 294–302.

[20] Sfetsos, P.—Stamelos, I.—Angelis, L.—Deligiannis, I.: Investigating the Impact of Personality Types on Communication and Collaboration-Viability in Pair Programming – An Empirical Study. In: Extreme Programming and Agile Processes in Software Engineering, Vol. 4044/2006 of Springer Lecture Notes in Computer Science, 2006, pp. 43–52.

[21] TechSmith Camtasia Studio web site. Available on: `http://de.techsmith.com/camtasia.asp`.

[22] Thomas, D.—Hunt, A.: Mock Objects. IEEE Software, Vol. 19, May/June 2002, No. 3, pp. 22–24.

[23] Wake, W. C.: Extreme Programming Explored. 1st edition. Addison-Wesley, 2002.

[24] Williams, L.—Kessler, R.: Pair Programming Illuminated. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

Andreas Höfer is currently Ph. D. student at the Department of Computer Science, University Karlsruhe, Germany. He received his diploma in computer science and his M. Sc. degree in computer science and multimedia from the University of Applied Sciences Karlsruhe, Germany. His research interests include the assessment of agile software processes and development methods.