Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems

School of Information Systems

6-2013

Introducing programmers to pair programming: A controlled experiment

A. S. M. Sajeev

Subhajit DATTA
Singapore Management University, subhajitd@smu.edu.sg

Citation

1

Published in Agile Processes in Software Engineering and Extreme Programming: 14th International Conference, XP 2013, Vienna, Austria, June 3-7: Proceedings, pp. 31-45.
DOI: 10.1007/978-3-642-38314-4_3

Introducing Programmers to Pair Programming: A Controlled Experiment

A.S.M. Sajeev[1] and Subhajit Datta[2]

[1] University of New England, Australia
sajeev@une.edu.au
http://mcs.une.edu.au/~sajeev
[2] IBM Research, Bangalore, India
subhajit.datta@acm.org
http://www.dattas.net

Abstract. Pair programming is a key characteristic of the Extreme Programming (XP) method. Through a controlled experiment we investigate pair programming behaviour of programmers without prior experience in XP. The factors investigated are: (a) characteristics of pair programming that are less favored (b) perceptions of team effectiveness and how they relate to product quality, and (c) whether it is better to train a pair by giving routine tasks first or by giving complex tasks first. Our results show that: (a) the least liked aspects of pair programming were having to share the screen, keyboard and mouse, and having to switch between the roles of driver and navigator (b) programmers solved complex problems more effectively in pairs compared to routine problems, however, perceptions of team effectiveness was higher when solving routine problems than when solving complex problems and (c) programmers who started pair programming with routine tasks and moved on to complex tasks were more effective than those who started with complex ones and moved on to routine ones. We discuss how these results will assist the industry in inducting programmers without prior pair-programming experience into XP process environments.

Keywords: pair programming, empirical software engineering, agile methods, extreme programming, software process, controlled experiment.

1 Introduction

Pair programming is one of the key concepts in agile methods such as XP. It involves two people working as a team to complete a programming task, usually involving, design, coding and testing of the task. One of the pair starts acting as the driver (who will do the keyboard activities) and the other as the navigator (who will watch, analyze, comment and, in general, guide the driver), and the two switch between these roles multiple times for the duration of the task.

Prior research has shown that pair programming when compared to solo programming can produce better outcomes (e.g. [1], [2]), even though others found the argument inconclusive (e.g. [3], [4]). Nevertheless, the popularity of agile methods compared to heavy-weight processes means that increasingly more programmers and software engineers will be required to do pair programming. This includes both professionals with work experience and fresh graduates. The question then is what to expect when engineers without prior background in pair-programming are brought into the XP method and how best to "prepare" them to be effective pair programmers. Addressing such questions will assist the industry to fine tune their induction processes for new pair programmers. The main objective of this paper is to report on an empirical study that advances research in these directions.

We specifically address the following research questions:

- What aspects of pair programming are valued most and least by programmers that are new to pair programming?
- Do new pair programmers work better as a pair in routine problems or in complex problems?
- How best to prepare a pair? Are pairs that start with routine tasks and move on to complex tasks more effective than pairs that start with complex tasks and move on to routine tasks?

In order to address these research questions, we conducted a randomised controlled experiment. Compared to an observational study, a controlled experiment allows us to set controls in order to measure accurately the effect of treatment on the experimental group. It reduces the effect of confounding variables which otherwise could be present in observational studies were the researcher has little control over the events.

There is significant work in the literature on various aspects of pair programming. This involves both experienced pair programmers and those who are new to pair programming. However, as discussed in Section 6, the questions we investigate here have not been explored in the literature with the view of better understanding how programmers without pair-programming experience could be prepared for the method.

The rest of the paper is organized as follows. In the next section, we review the related literature. In Section 3, we describe the research method. Section 4 gives the results of the analysis and in Section 5 we discuss the limitations of the study. Finally, in Section 6 we conclude the paper by discussing the implications of our results for the Information Technology (IT) industry.

2 Literature Review

The literature related to our study can be roughly classified into three themes: those investigating models and frameworks for pair programming, those testing the question "is pair programming better than solo programming", and those investigating the benefits of pair programming.

In investigating frameworks, Fronza et al. collected data non-invasively in an industrial development team for 10 months to understand how pair programming helps the integration of novices in a team [5]. Using social network analysis techniques, the authors analyzed developer interactions and proposed a model for novice integration in teams engaged in spontaneous pair programming. Gallis et al., on the other hand, have pointed out the contradiction in the claims around pair programming, which they attribute to the lack of theoretical foundation supporting empirical research [6]. To address this situation, the authors presented a framework for pair programming research by identifying and categorizing important independent, dependent, and context variables, and exploring their relationships. This work was extended by Ally et al.based on a study of pair programming using the Delphi technique [7]. The authors concluded that Gallis et al.'s framework needed to include an additional category of factors relating to organizational matters.

A number of studies compared pair programming with solo programming. Lui and Chan [8] investigated the research question "do pairs outperform individuals in procedural solution tasks?" using Programming Aptitude Test [PAT] rather than traditional programming tasks; the reasoning was that PAT is independent of programming language proficiency and thus language proficiency would not become a confounding variable. They used a measure called REAP (Relative Effort Afforded by Pairs) to compare sole programming productivity with pair-programming productivity. Lui and Chan [8] also introduced the concept of repeat programming in studying pair programming; this is where the pairs repeated the same programming task multiple times. They used the term *novice* to mean that a programmer is doing a repeated task for the first time, and the term *expert* for one who has repeated the same task several times. They concluded that "novice-novice pairs against novice-solos are much more productive in terms of elapsed time and software quality than expert-expert pairs against expert solos". Madeyski investigated how pair programming fares vis-a-vis solo programming for thoroughness and fault detection effectiveness of test suites and did not find support for anecdotal evidence that the former facilitates these activities [4]. Arisholm et al [1] conducted a one-day controlled experiment to test the effectiveness of pair programming with respect to complex tasks. They used junior, intermediate and senior staff from local industries as subjects. They compared pair-programming with individual programming, as well as they studied effectiveness in using pair programming in simpler tasks versus complex tasks. Dybå et al. examined the fundamental assumption behind pair programming – that two heads are better than one – by conducting a meta-analysis of existing studies around pair programming's effects on quality, duration, and effort [2]. They concluded that whether two heads are indeed better than one is a nuanced question, and the answer depends on programming exercise and task complexity. They found empirical evidence that two heads can achieve higher correctness on complex programming tasks and be able to finish simpler tasks earlier. In a subsequent paper the authors extended their results and concluded that, higher quality on complex tasks comes with the price of higher effort and reduced completion time is offset by lower quality [9]. The authors emphasized the need for more attention to moderating factors while exploring the effects of pair programming. Vanhanen and Lessenius reported results from a study of three pair programming and two solo programming teams performing the same 400-hour fixed effort project with a focus on understanding the aspects of productivity, defects, design quality, knowledge transfer, and enjoyment of work [10]. They found that pairs have an initial "learning time" that increases the development effort upfront vis-a-vis solo programming. Although this difference tapers off later in the development cycle, it affects the overall productivity of the pairs. Complexity of tasks was found not to influence effort difference between pair and solo programming. Pair programmers delivered systems with higher number of defects, but had higher knowledge transfer; they also gave weak evidence for higher enjoyment of work. In a subsequent work, the authors studied the perceived effects of pair programming vis-a-vis solo programming in large scale, industrial software development [11]. They surveyed 28 developers and found pair programming's positive effects were maximum for learning, schedule adherence, knowing other developers, and team spirit. Vanhanen and Korpi summarize the experiences of extensive pair programming in an industrial project [12]. They found that frequent rotation between the driver/navigator roles improved knowledge transfer, and the developers perceived that pair programming was better suited for complex tasks rather than easy tasks. Xu and Rajlich conducted a case study with six students in a graduate software engineering course who were assigned to work on incremental changes to an application either individually or in pairs [13]. They found that the paired students delivered their change requests more quickly and with a higher quality. Similarly, Sison reported results from an experiment on the use of pair programming by undergraduate students in a software engineering course at a Philippine university [14]. The author found evidence that defect densities were significantly lower for programs written by pair programming vis-a-vis those written by non pair programming teams.

On the benefits of pair programming, Begel and Nagappan reported results from a longitudinal study of pair programming at Microsoft [15]. They found that pair-programming's biggest benefits were perceived to be fewer bugs, wider understanding of code, and overall higher code quality. Additionally, most of the study's participants were more amenable to a partner with complementary skills, flexibility, and good communication skills. Coman et al. examined the dynamics of the pairing process in a mature agile team of 16 developers in a three months study and found support for the claim that pair programming is useful for training and knowledge transfer [16].

3 Research Method

3.1 Participants

Our participants are 144 students of the two year Master of Technology (MTech) program at the International Institute of Information Technology, Bangalore (IIT-B); they were enrolled in a software engineering course mandatory for all MTech students. The experiment was conducted as part of programming skills assessments in the course for which they received credit. All students had an undergraduate computing degree in science or engineering. Sixty nine percent of the students had no work experience, 7% had less than a year's experience, and the remaining 24% had more than one year experience in the industry with maximum four years; 89% of those with work experience worked as programmers or software developers. Of the 144 subjects who participated in the study, only seven had prior experience with pair programming; these seven subjects participated in the experiment, but were excluded from the analysis of results. (One of them was part of the control and two formed a pair, the remaining four formed pairs with non-experienced subjects; those four non-experienced subjects were also excluded from the analyses in order to avoid their partners' pair-programming experience confounding the results.)

All participants were proficient (through work experience and/or prior coursework) in object oriented analysis, design and programming, and rated their Java programming skills at level 3 or above, on a scale of 0 = novice, 5 = expert. Even then, participants were allowed to complete their assigned programming tasks in either Java or C++. (There was only one submission in C++.)

Thus consistent with our research objectives, our subjects were a collection of fresh graduates and professionals with work experience up to four years, but none with prior experience in pair-programming.

3.2 Method

Our research method was a randomized concurrent controlled experiment spanning three sessions of programming. In a controlled experiment, one group acts as control (in our case, they consisted of solo programmers) and the other acts as the experimental group (in our case, pair programmers). In a randomized experiment, each participant is chosen at random to be in the control group or in the experimental group, and in a concurrent experiment, all groups do the activities (in technical terms, undergo the treatment) at the same time.

The subjects were randomly divided into an experimental group and a control group; this was done by drawing lots from a bowl. The experimental group consisted of 50 pairs of programmers whereas the control group consisted of 44 solo programmers. (As mentioned in the previous section, the pairs and controls that involved people with prior pair-programming experience were excluded from analysis thus resulting in 45 pairs and 43 controls.) Both the experimental group and the control group were further divided into two sub-groups in a two-factorial design; for reasons explained below, one subgroup was named Routine-to-Complex cohort, and the other, Complex-to-Routine cohort (See Figure 1).

3.3 Procedure

The experiment was organized as follows:

1. Approval to conduct the study was obtained by the second author from the institute authorities.

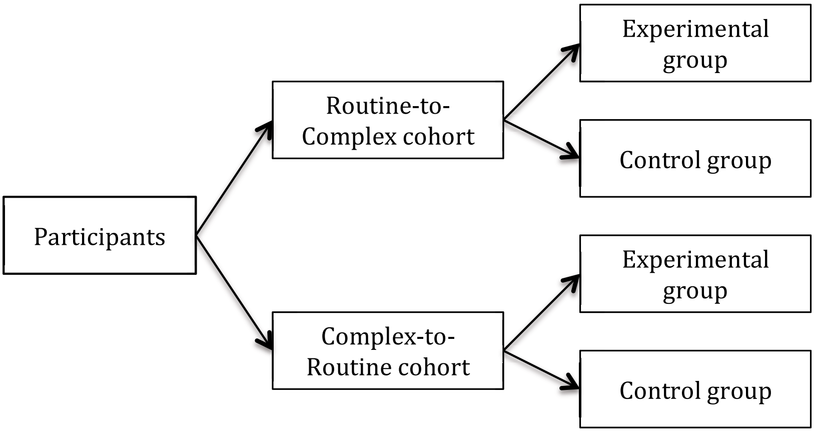


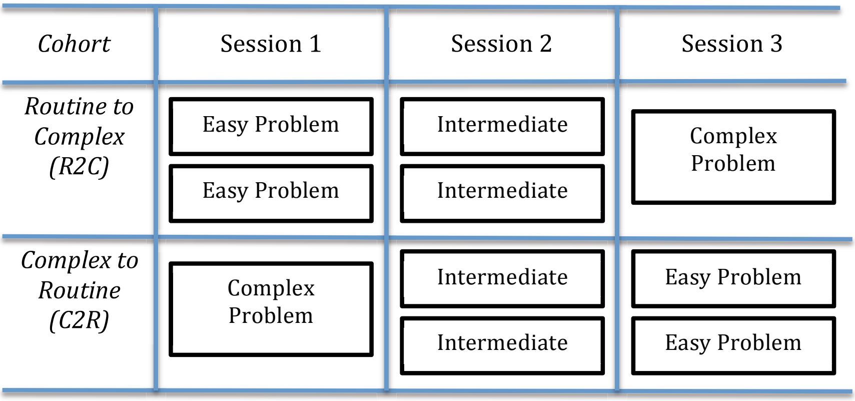
Fig. 1. Formation of experimental and control groups

2. An introductory lecture was given to all participants on pair-programming by the second author (who is an adjunct professor at the Institute) to ensure that all participants understood, in theory, the principles and practices of pair-programming.
3. An experiment is an event that occurs over a pre-defined time period. This means that we need to abstract real-world entities to fit into the framework of the experiment. In our case, we needed to model routine tasks and challenging tasks that pair programmers would encounter in the industry. We abstracted them into a set of programming problems of three levels of complexity (easy, moderately difficult and hard). The problems were selected from end of the chapter exercises of standard Java programming textbooks with appropriate modifications; each problem was annotated with a particular level of difficulty. In addition to that, the second author[1] and each of the four teaching assistants (TA) who assisted him solved the problems independently to confirm their differential level of complexity. For the easy level, we had four exercises that were interchangeable in terms of their difficulty, and similarly for the medium level. For the challenging level, we had two exercises that were interchangeable in terms of their difficulty. To ensure problems at the same level are of similar difficulty, they were selected from the same textbooks and further confirmed by ensuring they take similar amount of time to solve.
4. We prepared a set of test cases (input and expected output) for each problem. The subjects were given, in addition to the problem specification, a subset of the test cases to assist them in deciding when a task is complete. The full set of test cases were used by the researchers to give a quality score to the program produced. The quality score to the solution given was out of 10,

[1] Those interested in replicating the experiment may contact this author for the exercises.

where 0 means does not compile, 5 means means passes 50% of the tests and 10 means passes all tests.

5. The experiment was conducted in three sessions (see Figure 2). The Routine-to-Complex cohort was given the easy problems in the first session, then in Session 2, problems of medium complexity and finally in Session 3, a challenging problem. The Complex-to-Routine cohort solved problems in the other direction as shown in the figure. Since we selected a sufficient number of problems, those given in a session were not reused in another; this was to prevent subsequent sessions being affected by any discussions by the participants outside the experiment of solutions and problems they have worked on in a session.
6. At the end of each session, both a quantitative and qualitative evaluation of the groups were conducted. For quantitative evaluation, as mentioned above, the programs were tested for correctness; this was done by the TAs under the second author's supervision and a score of 0 to 10 is given. For qualitative evaluation, each individual in the experimental groups was given a team-effectiveness questionnaire. The questionnaire measured on a five point Likert scale (where 1 = disagree strongly, 3 = neither agree nor disagree and 5 = agree strongly) each programmer's perceptions on the effectiveness of pair programming. These perceptions were also analysed separately (as described in Section 4.1) to identify how the subjects favoured different pair programming practices.



Cohort	Session 1	Session 2	Session 3
Routine to Complex (R2C)	Easy Problem Easy Problem	Intermediate Intermediate	Complex Problem
Complex to Routine (C2R)	Complex Problem	Intermediate Intermediate	Easy Problem Easy Problem

Fig. 2. Organization of the cohorts

3.4 Statistical Tests

For comparison between mean values of two groups, we used the independent samples t-tests [17]. The significance level, α was set at 0.05. Levene's test was used to check homogeneity of variances. The analysis were conducted using SPSS software package. We did not adjust for the potential Type-1 error increase from multiple tests; Bonferroni's adjustment, for instance, lowers the alpha level for multiple tests; however, some researchers recommend presenting the p-value instead of adjusting the alpha-level and let the readers decide on the results (for example, see [18]). Eta-squared was used to determine the effect size; a value of 0.01 is considered a small effect whereas 0.14 or above is a large effect [19].

4 Results

4.1 Pair Programming Characteristics

Figure 3 shows in descending order how the subjects favored different characteristics of pair programming. The explanation for the features is given in Table 1. The most favored ones were that *pair program allowed good discussions of the problems and solution strategies*, and that *it was helpful in developing teamwork skills.* The least favored characteristics were *the need to regularly swap the roles of driver and navigator* and *the need to share the screen, keyboard and mouse.* Also, the statement: *pair programming makes programming faster* was among the least agreed items.

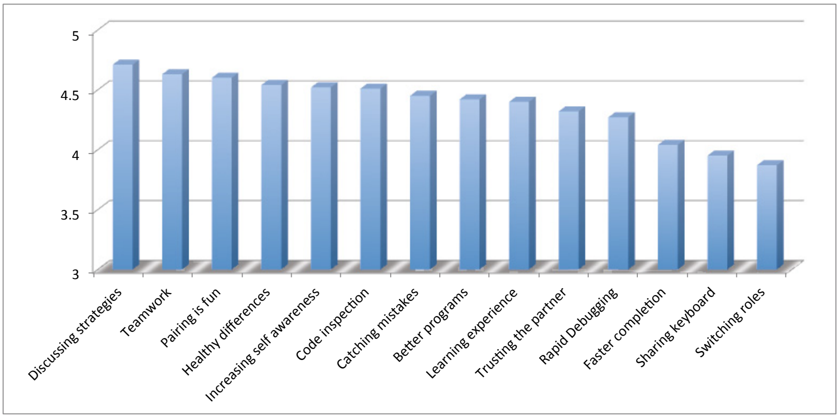


Fig. 3. Mean score on a 5-point Likert scale[3]for different pair programming characteristics (5 is strong agreement and 0 is strong disagreement)

4.2 Routine versus Challenging Development

Routine development is where programmers are familiar with the work and generally know how to proceed to completion, whereas challenging development is where the task that the pair has to address needs lateral thinking and exploring

[3] Strictly speaking, Likert scale is ordinal, however, it is not uncommon for researchers to use it as an interval scale (for example: see [20]).

Table 1. Full form of the programming characteristics in charts

Programming Characteristics	Likert Item
Discussing strategies	Our team was good in discussing the problems and solution strategies
Teamwork	Pair programming is helpful in developing teamwork skills
Pairing is fun	I found pair programming fun
Healthy differences	In our team, we sorted out differences in a healthy manner
Increasing self awareness	Working with a partner makes it easy to understand what I am doing and why I am doing it
Code inspection	Having a partner is beneficial for learning to read another programmer's code
Catching mistakes	My partner and I caught each other's mistakes
Better programs	I believe, pair programming leads to better programs than individual programming
Learning experience	I have learned more working in pairs than when I have worked individually
Trusting the partner	My level of trust in my partner is very high
Rapid Debugging	My team found errors more rapidly than if we were working individually
Faster completion	Without pair programming I would have taken longer to complete the programming task(s)
Sharing keyboard	It was easy to share the keyboard, screen and mouse
Switching roles	In my team, we changed the role of 'driver' and 'navigator' fairly regularly

of different strategies in order to come up with a good solution. Intuitively, one could hypothesize that pair programming is more useful and effective for challenging tasks since it literally doubles the brain power. We tested this hypothesis by taking into consideration only the tasks completed in Session 1. The first session in our experiment is where new teams were formed. Not considering all sessions avoids any confounding influence of growing familiarity of partnerships on team effectiveness.

In Session 1, we had the R2C cohort (see Figure 2) solve easy problems and the C2R cohort solve a challenging problem; both cohorts consisted of an experimental group of pairs and a control group of solo programmers. In the discussion below, the pairs of the Session-1 R2C cohort are referred as the routine-group (that is, the group that solved easy problems) and the pairs of the Session-1 C2R cohort as the complex-group (that is, the group that solved a complex problem).

The questions we investigated are:

a) Is there a significant difference between the mean results of the routine-group and the complex-group?
b) Is the mean result of the routine-group significantly better than the mean result of the corresponding control group?
c) Is the mean result of the complex-group significantly better than the mean result of the corresponding control group?

The effectiveness of a group is measured using its (i) mean test score and (ii) mean team-effectiveness score. The answer to Question (a) could tell us which group has performed better, however, that answer would not be relevant unless that group has also performed better in test results than its corresponding control group; otherwise, we did not have to use pair programming to achieve the better results. Thus, answers to Questions (b) and (c) together with (a) should tell us which kinds of problems, pairing is better suited for.

With respect to mean test scores, there is significant difference between the routine-group and the complex-group with (as should be expected) the routine-group scoring much higher mean test scores ($M = 7.06, SD = 2.5$) than the complex-group ($M = 1.32, SD = 1.32$), $p < 0.0001$, $\eta^2 = 0.678$ There is also significant difference in perceived team effectiveness, with the routine group perceiving higher effectiveness ($M = 4.33, SD = 0.39$) than the complex group ($M = 3.99, SD = 0.5$), $p = 0.001$. The practical significance is also large ($\eta^2 = 0.126$). It may be the case that, when the teams are able to get better test scores (which is not unexpected with routine problems) they feel that their team is more effective. When the routine pairs were compared with their control group in test scores, however, there was no significant difference ($M = 7.06, SD = 2.5$ versus $M = 6.19, SD = 3.46; p = 0.306$). On the other hand, when the complex pairs were compared with their control group, there was significant difference albeit with a moderate effect size ($M = 1.32, SD = 1.32$ versus $M = 0.86, SD = 0.36; p = 0.025; \eta^2 = 0.07$). This is a very interesting result. *Pair programmers achieved significantly better results than their corresponding solo programmers in solving complex problems, however, they perceived pair programming to be less effective. On the other hand, pairs solving routine problems did not fare any better in test scores than solo programmers doing the same activity, even then, the routine pairs perceived their teams to be more effective than their complex counterparts.*

4.3 Training Pairs

The main challenges in the industry generally comes not from addressing routine tasks, but from having to solve complex tasks. Therefore, we tested what is a better approach in preparing new pairs to solve complex tasks. We explore the question of whether it is better to start new pairs with routine tasks and then move on to complex tasks, or start them with complex tasks. The reasoning behind the former approach is that with easy tasks there is likely to be less tension between the pairs and it gives them time to know each other better before moving on to complex ones. On the other hand, a possible reasoning behind starting pairs with complex tasks first could be that such a problem would force the pairs to put their heads together and work, whereas if a team is started with easy tasks, it might be a hindrance for the individuals to engage in teamwork thinking that “I could do this myself; having somebody sitting next to me is a distraction”.

As explained in Section 3 and Figure 2, we had two cohorts of programming pairs, one cohort (the R2C cohort) started with easy problems and moved on to intermediate and complex problems whereas the other cohort (the C2R cohort) started with a complex problem and moved on to intermediate and easy problems. In other words, both cohorts did exactly the same number of easy, intermediate and complex problems, but in two different directions. For each cohort, as given in Figure 1 there was an experimental group and a control group.

As in the previous section, we tested whether there is a statistically significant difference in the test scores and team effectiveness scores between the R2C and C2R experimental groups in solving complex problems. We then compared each cohort independently with its control to test whether the effect is in fact from pair-programming.

Both in terms of test scores and team effectiveness scores, the cohort that started with easy problems (R2C) performed better in complex problem solving than the cohort that started straightaway with a complex problem (C2R) as shown in Table 2.

Table 2. Difference in scores between R2C and C2R cohorts

	Cohort	*Mean*	*Std. Dev.*	*p*	η^2
Test scores	Routine-to-Complex	3.89	2.01	< 0.0001	0.361
	Complex-to-Routine	1.36	1.35		
Team effectiveness	Routine-to-Complex	4.34	0.48	0.001	0.122
	Complex-to-Routine	3.97	0.51		

The difference in test scores between experimental and control groups in both R2C (p=0.03) and C2R (p=0.044) were significant indicating that pair programmers produced better outcomes than their solo counterparts in solving complex problems irrespective of whether they started with a complex problem or an easy problem.

5 Limitations

As in any empirical research there are several limitations to our study. It would have been ideal to conduct this experiment in an industrial setting with a mixture of fresh and experienced employees as participants. However, we believe, our choice of subjects does not invalidate the findings because even though the subjects come from a software engineering Masters degree program, their background makes them good proxies for fresh and experienced industry employees. Besides, when Höst et al. [21] compared the use of students with industry professionals, they found no significant differences in tasks involving software engineering judgment; they concluded that students can be used instead of professional software engineers if they are senior masters students rather than undergraduates.

Another limitation of the experiment is that we used graded Java exercises as proxies for routine, intermediate and complex industry tasks. The pair-programming tasks used in an industry generally will be part of a larger software product, whereas, the tasks we gave were standalone Java exercises. Since our aim is to identify pair programming behavior which is unlikely to be different whether the task is a standalone program or a well defined part of a larger software system, this abstraction is unlikely to have affected the external validity of the results.

Another threat to external validity is that we tested pair programming in isolation, whereas, in the industry, pair programming is likely to be only one part of the implementation of the Extreme Programming process. Any influence of other features of XP on pair programming is not included in our experiment.

We did not measure time of completion of tasks therefore were unable to measure success in terms of how fast the different cohorts completed their tasks.

We did not use the same tasks for more than one session; this was deliberately done to avoid the participants discussing the solution with other groups outside of the experiment. Instead, we chose several tasks of similar level of difficulty for different sessions. A threat to internal validity occurs if the level of difficulty varied; as explained in Section 3, we reduced this threat by independently solving each problem prior to the experiment and assessing their difficulty.

6 Discussions and Conclusion

Our study has similarities and differences with prior research discussed in Section 2. Our finding of the appropriateness of using new pairs for complex tasks concur with results of [1] and [2] which found experienced pair programmers also giving better outcomes in the case of complex tasks. Our primary focus, however, was not testing whether pair programming is "better" than solo programming; instead we used solo programmers as controls in identifying significant differences among cohorts of pair programmers. Additionally, we measured pair programming success both in terms of test results and perceived team effectiveness of pairs and compared the two measures, whereas prior literature largely measures effectiveness in terms of time and test results. Further, whereas prior work such as [15] investigated benefits of pair programming in general, we looked at the degree of acceptance of different features of pair programming with the view of identifying the ones that worries programmers new to the method.

As agile methods such as XP get increasingly adopted in the industry, software engineers and programmers who have not experienced the concept of pair-programming will need to be inducted into it. Our results, as discussed below, provide a number of guidelines for the software industry to make the induction smoother.

6.1 Pair-Programming Practices

While programmers appreciate the general benefits of pair programming such as the ability to discuss problem-solving strategies with the teammate, developing teamwork skills and learning to sort out differences in a healthy manner, it is the manual aspects of pair programming that needs attention. Aspects such as the need to work with a shared screen, keyboard and mouse and the need to switch roles between driver and navigator were comparatively less liked by our participants who were new to pair programming.

There are two ways to address these issues. One is education and practice: that is, while introducing pair programming to new programmers, it is not enough to tell them how it works, but also there should be sufficient training sessions for them to practice and become comfortable with its routine aspects such as sharing of the screen. Another way which industrial engineers and researchers could investigate is the possibility of having two screens duplicating the same information, and perhaps also having two keyboards and mice (where the input devices get locked and unlocked with a simple click as the pairs change roles). Further research is needed to see whether such technical solutions will help or hinder pair-programming outcomes.

6.2 Pair Programming Effectiveness

We found a dichotomy between team effectiveness and product quality. Perception of team effectiveness increased as pairs achieved success irrespective of whether the same success could be achieved through solo programming. Thus the pairs who were solving easy tasks found their teams to be more effective than the pairs who were solving complex problems. However, for managers, pair-programming can be considered effective, not just when it achieves better test results, but when it achieves better results *than what a solo programmer could achieve*. In our results, the pairs solving complex problems were getting significantly higher test scores than the solo programmers who were solving the same complex tasks, whereas, there was no statistically significant difference in the test scores between pairs and solo programmers who were solving easy tasks.

This demonstrates that, irrespective of how new pair programmers feel about their team effectiveness, pair programming results in better product quality when they are used for complex tasks. Thus new pairs attempting complex tasks may not feel that their team is working as effectively as it should, however, they are likely to produce significantly better results than solo programmers attempting the same tasks. On the other hand, for easy tasks, pair-programmers may feel that their team is working well, however, their performance in terms of test results is not significantly better than solo programmers, and therefore, it may not be worthwhile employing pairs in easy or routine tasks.

6.3 Training Pair Programmers

In the previous subsection, we mentioned that employing pairs in routine or easy tasks may not be worthwhile. However, there is one important reason to employ pairs in easy tasks, and that is to get them prepared for complex tasks. Whether new pairs programmed easy tasks first before moving on to complex tasks, or tackled complex tasks first, they performed significantly better in solving complex tasks than their solo counterparts. However, more important is the finding that the pairs who started with easy tasks first got better test and team-effectiveness outcomes when solving complex tasks, than the pairs which started with a complex task straightaway. The possible reason is that the former cohort got to work on their team-building skills while on easier tasks and therefore were better prepared to tackle the complex tasks as a team. Thus, when programmers are inducted into pair programming, it is likely to pay off if pairs are started with easy problems before moving on to complex ones. Throwing pairs in at the deep end thinking along the lines that a big challenge will encourage them to work together would not be an effective strategy.

In conclusion, successful induction of programmers into pair-programming depends on us understanding how programmers without prior experience would respond in such situations. This paper contributes to that effort by addressing a number of research questions on achieving effectiveness.

Acknowledgments. We thank the teaching assistants who helped us with the conducting of the experiment and the anonymous referees for their valuable reviews.

References

1. Arisholm, E., Gallis, H., Dybå, T., Sjøberg, D.: Evaluating pair programming with respect to system complexity and programmer expertise. IEEE Transactions on Software Engineering 33(2), 65–86 (2007)
2. Dybå, T., Arisholm, E., Sjøberg, D., Hannay, J., Shull, F.: Are two heads better than one? on the effectiveness of pair programming. IEEE Software 24(6), 12–15 (2007)
3. Hulkko, H., Abrahamsson, P.: A multiple case study on the impact of pair programming on product quality. In: Proceedings of the 27th International Conference on Software Engineering, pp. 495–504. ACM (2005)
4. Madeyski, L.: On the effects of pair programming on thoroughness and fault-finding effectiveness of unit tests. In: Münch, J., Abrahamsson, P. (eds.) PROFES 2007. LNCS, vol. 4589, pp. 207–221. Springer, Heidelberg (2007)
5. Fronza, I., Sillitti, A., Succi, G.: An interpretation of the results of the analysis of pair programming during novices integration in a team. In: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 225–235. IEEE Computer Society (2009)
6. Gallis, H., Arisholm, E., Dyba, T.: An initial framework for research on pair programming. In: Proceedings of International Symposium on Empirical Software Engineering, pp. 132–142. IEEE (2003)
7. Ally, M., Darroch, F., Toleman, M.: A framework for understanding the factors influencing pair programming success. In: Baumeister, H., Marchesi, M., Holcombe, M. (eds.) XP 2005. LNCS, vol. 3556, pp. 82–91. Springer, Heidelberg (2005)
8. Lui, K., Chan, K.: Pair programming productivity: Novice–novice vs. expert–expert. International Journal of Human-Computer Studies 64(9), 915–925 (2006)
9. Hannay, J., Dybå, T., Arisholm, E., Sjøberg, D.: The effectiveness of pair programming: A meta-analysis. Information and Software Technology 51(7), 1110–1122 (2009)

10. Vanhanen, J., Lassenius, C.: Effects of pair programming at the development team level: an experiment. In: International Symposium on Empirical Software Engineering, 10 pages. IEEE (2005)
11. Vanhanen, J., Lassenius, C.: Perceived effects of pair programming in an industrial context. In: 33rd EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 211–218. IEEE (2007)
12. Vanhanen, J., Korpi, H.: Experiences of using pair programming in an agile project. In: 40th Annual Hawaii International Conference on System Sciences, pp. 274b–274b. IEEE (2007)
13. Xu, S., Rajlich, V.: Pair programming in graduate software engineering course projects. In: Proceedings of the 35th Annual Conference on Frontiers in Education, pp. F1G–F1G. IEEE (2005)
14. Sison, R.: Investigating pair programming in a software engineering course in an asian setting. In: Proceedings of the 15th Asia-Pacific Software Engineering Conference, pp. 325–331. IEEE (2008)
15. Begel, A., Nagappan, N.: Pair programming: what's in it for me? In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 120–128. ACM (2008)
16. Coman, I.D., Sillitti, A., Succi, G.: Investigating the usefulness of pair-programming in a mature agile team. In: Abrahamsson, P., Baskerville, R., Conboy, K., Fitzgerald, B., Morgan, L., Wang, X. (eds.) XP 2008. LNBIP, vol. 9, pp. 127–136. Springer, Heidelberg (2008)
17. Moore, D.S., McCabe, G.P.: Introduction to the Practice of Statistics. W. H. Freeman & Co., New York (2006)
18. Perneger, T.: What's wrong with bonferroni adjustments. BMJ (British Medical Journal) 316(7139), 1236–1238 (1998)
19. Cohen, J.: Statistical power analysis for the behavioral sciences. Lawrence Erlbaum (1988)
20. Haag, S., Raja, M., Schkade, L.: Quality function deployment usage in software development. Communications of the ACM 39(1), 41–49 (1996)
21. Höst, M., Regnell, B., Wohlin, C.: Using students as subjects: a comparative study of students and professionals in lead-time impact assessment. Empirical Software Engineering 5(3), 201–214 (2000)