Chapter 1

Pair Programming

Chih-Song Kuo

1.1 Introduction

Pair programming, as a rule of Extreme Programming, requires two developers work together on the same problem at the same time.

The pattern of pair programming in which two individuals develop one software module together has attracted researcher and practitioner's interest for almost 15 years . People following the pair programming protocol sit in front of one screen, use a set of keyboard and mouse to collaboratively solve a programming task [30]. While one developer is modifying the source code, another is required to perform continuous code-review. This pattern was claimed by it advocators to yield earlier release and higher software quality [30], and was included as one of the rules of Extreme Programming (XP) [27], a popular software development methodology used widely in the software industry.

Advocates claim pair programming to provide higher software quality with the same or minor additional effort.

Despite the advantages proposed for pair programming, many still suspect the overall usefulness and benefit over traditional solo programming. One of the most questioned aspects lies on the feasibility of achieving superlinear speedup as compared to the legacy pattern [30]. From a simple management's viewpoint, there is no reason to pair up developers if they cannot do things twice faster. In addition to productivity, people also doubt if the

improved software quality deserves hiring twice many programmers [30]. Although pair programming is claimed to cost an insignificant 15% more effort yet achieve a higher quality than solitary programming on the same task [30], the paradigm is still questioned if solo programming plus an additional review phase, which might be cheaper and equally effective, will achieve the same goal.

Empirical research on the effectiveness of pair programming in both academic and industrial settings conclude inconsistent findings.

While pair programming is suggested by Extreme Programming, it is unclear to what extent the setting is applied and how effective it is in the software industry. Persuasive empirical evidence are still of relative shortage. Earlier studies were mostly conducted in a course room setting with college students being the subjects [30][20][8][24][25][19]. Despite the difference to the industrial environment, the conclusion of these research were not entirely consistent. Certain weaknesses in the design of experiments, for example, small samples and short programs, also threat the validity of their outcomes. Recently, a few more papers investigating the effect of pair programming in the industrial setting were published [14][21][16][22][17][3]. However, due to their inconsistent findings, it is still hard to tell whether pair programming is really worth adopting. In a word, insufficient evidence from the research community restrains companies transforming their development pattern to pair programming.

A few implications can be inferred from previous studies to help practitioners better use pair programming.

Although it cannot be clearly concluded from previous research whether pair programming reduces the development effort and meanwhile improves the quality of the artifacts, a few additional observations can be found which might help practitioners know when is and how to make pair programming more effective. Issues like under what situation pair programming appears to be advantageous are discussed. Some research begin to understand the impact of pairing individuals with different levels of expertise[17][3]. Studies seeking a combined form of pair programming and solo programming are also published [18][15]. Furthermore, there are evidence showing that pair programming might help teaching activities in Computer Science programs, for example, improving student confi-

dence and course performance [28].

This paper introduces pair programming and explores its effectiveness based on historical research.

This seminar paper is organized as follows. In section 2 a detailed description of pair programming including the origin of the concept as well as the claimed advantages and disadvantages will be given. In section 3 we do a thorough review of previous empirical research mainly on the evidence proving or disproving the effectiveness of pair programming. We investigate a few potential issues of pair programming based on recent publications in section 4. In the last section we conclude the research findings of pair programming.

1.2 The Pattern of Pair Programming

The concept of pair programming can be traced back to 1975.

Although pair programming seems to be a new pattern of coding within about one decade, record shows that people practiced pair programming more than 35 years ago. It was in year 1975, when Fortran was the mainstream language, Jensen experimented pair programming in a team of the United States Air Force and observed 2.27 times speedup and three order-of-magnitude improvement in error rate [14]. In 1991, Flor introduced the idea of collaborating different individuals to overcome complex problems [11]. In 1995, Constantine stated in his book that "Two programmers in tandem is not redundancy; it's a direct route to greater efficiency and better quality" [9]. In the same year, Coplien named what he called "Developing in Pairs" pattern [10]. There, programming in pairs was claimed to be useful for solving big problems than programming individually. That is, pairs are less likely to be sideblinded when they deal with software developing tasks.

Extreme Programming and Laurie William's study promotes the use of pair programming.

In 1998, another inspiring empirical study on the use of the pattern in industry came to being [2]. The article describes a successful software project using Extreme Programming at Chrysler, where pair programming was enforced and provided high quality of the resulting artifacts. One of the project member, Don Wells, later published "The

Rules of Extreme Programming" [27] with one of them stating "All production code is pair programmed." [27]. There pair programming is defined as two developers working together at a single computer on the same code. Wells suggests the setting that two programmers sit side by side in front of a screen and slide the keyboard and the mouse back and forth, which later confirmed by Williams by stating the "slide the keyboard/don't move the chairs" rule as illustrated in Figure 1.1 [33]. The one controlling the keyboard and mouse is called the driver, while another one reviewing and thinking is called the navigator. In addition, The rule

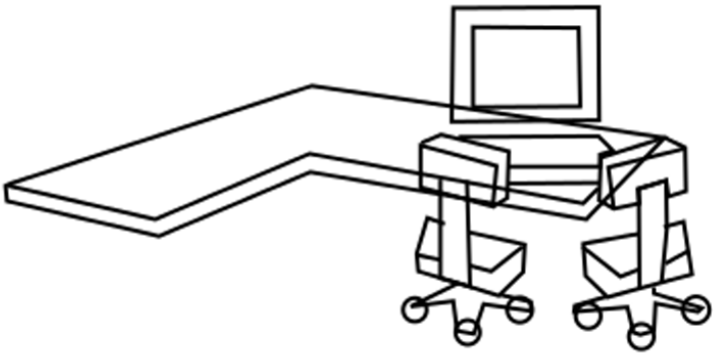

Figure 1.1: Suggested pair programming setting based on [33].

demands an adequate learning period of the pattern in order to get that pattern to work. It is claimed by the rule that pair programming will yield the same productivity while a better quality of code.

Potential pros and cons of pair programming are summarized.

The inclusion of pair programming as a rule of Extreme Programming and Williams's publication puts pair programming onto the stage and thus attracted reasonably many discussions from the industry and the research community. Pandey asserted that the new paradigm overcame four weaknesses of solo programming: steep learning curve, single point of failure, too complex for one, inability to review work [22]. Despite the dispute on speedup and quality improvement, several possible additional advantages and disadvantages are proposed as a result of cer-

tain observations. We summarize these points as follows:

1.2.1 Advantage

Pair programming is claimed to be advantageous in several aspects, notably and arguably on the development speedup and software quality improvement.

(1) Earlier task completion: Pair programming can potentially increase the development speed and thus shorten time-to-market. On the economic's perspective, two developers working in pairs is only superior when the resulting time is less than halved compared to individuall programming. That is, two times speedup. However, with the addition of other advantages, lower rate of speedup might also be acceptable [30]. Another useful situation could be an indivisible task on the critical path of a project. There pair programming is claimed to be a helpful way accelerating the development [32].

(2) Higher software quality: Software quality is expected to increase from pair-programming due to the "two heads are better than one" rule as well as continuous code review. In practice, earlier fault removal is really critical. The longer defects remain in the product, the more costly it is to find and to fix them [30]. Rework after release can cost far more effort and possibly degrade customer's satisfaction. Different observations of software quality improvement can be found in [30].

(3) Smaller variance of effort: Smaller variance of effort means greater predicability of effort, which is considered very beneficial in software project planning. Research [8][20]implied lower standard deviation of the time needed for a pair programmed task.

(4) Smaller variance of quality: Studies [8] reported that the quality of software designs produced by pairs had 70% less standard deviation. This advantage implies better predicability of software reliability.

(5) Cross-learning: Several researchers identified interchange of knowledge in the process of pair programming [30]. It should be noted that usually employee training is quite expensive.

(6) Positive peer pressure: Two developers working together introduce pair pressure [29], which makes peo-

ple concentrated and punctual, and avoid them to get distracted. Research [33] reported that the developers working in pairs are far less likely to waste time reading e-mail, surfing the web, or zoning out the window - because their partner is awaiting continuous contribution and input.

(7) Better code readability: Readability promotes exchangeability of code and potentially lowers program maintenance cost. Related evidence such as improved conformance of coding standards and higher comment ratio can be found in [21][13].

(8) Improved self-satisfaction: Self-satisfaction makes people happy at work and retains the talented staff. The survey done in [30] reported that 90% programmers enjoyed pair programming more than solo programming.

(9) Improved self confidence: Self-confidence improves one's willingness and ability to solve complex tasks. Studies [30] showed that pair programmers were more confident.

(10) Decreased communication cost: Working in pair halves the communication needed for integration and thus reduces additional effort in a large project team [30].

(11) Decreased integration effort: As the project was coarse partitioned, the modules needed to be integrated is halved [30].

(12) Reduce the chance of task misunderstanding: the chance that one person incorrectly interprets a requirement is higher than two [20]. Correct interpretation of the task reduces additional rework.

(13) Early identification of faults: Due to continuous code review, potential software faults can be identified earlier, where as in solo programming, a fault typically exists until an appropriate code review or test is undertaken [30].

1.2.2 Disadvantage

The most criticized point of pair programming is the increased effort of the project.

(1) Lower productivity per developer: Pairs typically do not complete the task twice fast. Certain research [20][3] even reported an unchanged development time for paired programmers. The observation implies twice as many effort to be allocated for a project using pair programming.

(2) May easily get exhausted: Pair pressure makes one concentrated on work but also makes one tired soon. Previous research [29][33] indicated that pair programming could be a very intense and mentally exhausting.

(3) Incompatible pair harms: Pairs having incompatible personalities or coding styles can make things even worse and thus slow down the development progress [4].

(4) Scheduling challenge: Pairs must have exactly the same schedule. This includes the same office hours, meeting minutes, which are usually difficult to enforce [4].

(5) Might not work for all characteristics: According to [30], it is said that "Some have trouble working in pairs. Typically those with excess ego or too little ego." In certain cultures, high or low level of ego might be encouraged. Thus pair programming might not work so well in these contexts or need to be adapted.

(6) Slow down the genius: Sometimes the most talented developers refuse to be paired up with juniors as doing so might actually degrade her productivity. According to the experiment conducted in [31], around 5% students, who were typically the top-performing ones, desired to work alone and did not want to be slowed down.

(7) Require a pair jelling period: Whenever two programmers who never worked in pair before are coupled together, reasonable time is proved to be necessary for the individuals to understand each other and to find a good pattern of collaboration. Research [30][24] has indicated a relatively low productivity during this transition period.

(8) Hiring difficulties: IT professionals are constantly in a shortage such that even pair programming works, it might be challenging to hire reasonably many more qualified programmers [21]. In case the company can only add less-experienced people to the team, would pair programming still outperform solo programming?

Concluding the potential pros and cons of pair programming.

In a word, pair programming can bring plenty of benefits to a development team. These typically include but not restrict to progrees acceleration, quality upgrade, higher code readability, and distraction avoidance. Advocates believe that these merits together actually decrease the total effort of a project, and lead to a better software product, a better team. While criticisms are more focused on the increased effort revealed in the empirical research, several further activities that can potentially profit by the additional factors might have been ignored. For example, integration of modules can be very time-consuming because of conflicting assumptions made by different individuals. Pair programming cuts this cost into half. Moreover, by producing higher quality code in the beginning, pair programming reduces later test and rework effort, which could be reasonably high in some cases.

1.3 Empirical Experiments and Studies

The findings on the impact of pair programming to development speedup and software quality improvement are summarized.

Although earlier empirical studies [14][2] accredited pair programming so much, the pattern did not went through smoothly as Extreme Programming did, but instead received noticeable criticism [5]. Among all, the economic feasibility was most frequently questioned. Skeptics doubted about the doubled productivity brought from pair programming and thus consider it as a waste of resources compared to solo programming [30]. As a result, further quantitative evidence is needed to clarify the effectiveness. In the followings, we summarize the experimental results of previous empirical research, with the focus on (1) the speed-up and (2) the software quality improvement, as these two measures are considered the most critical ones, for example, in [21].

Speedup Factor (SF) is defined as the ratio of the time needed by a solitary to a pair.

To make a standardized comparison, certain performance indices must be established. In this paper we define the Speedup Factor (SF) as the ratio of duration needed by a solo programmer over a team of paired programmers to achieve the same functionality. That is,

$$SF = \frac{Time\ needed\ by\ a\ solo\ programmer}{Time\ needed\ by\ a\ team\ of\ paired\ programmers} \tag{1.1}$$

. When SF is 1, the same task takes a pair the same time as an individual programmer, which implies twice as many efforts to be allocated. Having SF of 2, meaning that the time needed by a pair is halved than a solo, is usually regarded as the baseline for adopting pair programming disregarding other advantages such like quality improvement. Note that the comparison between the time spent by the two programming patterns is only valid when the same functionality and the level of quality are fixed. Nevertheless this assumption is not guaranteed in several experiments we investigated. The reader (of this paper) should understand this bias when interpreting the respective numbers.

Software Quality Improvement Factor (SQIF) is defined as the ratio of the quality achieved by a pair to a solitary.

We define the Software Quality Improvement Factor (SQIF) as the ratio of a quality index yielded from pair programming over solo programming. However, due to the diversity of quality measurement methods used in different research, defining SQIF explicitly is rather difficult. In practice, a quality index can be the number of passed test cases, the inverse of defect count or the defect density, etc. The reader should realize the possible influence made by different quality measures to the SQIF index. When SQIF equals 1, pair programming yields the same quality of code to solo programming. Having SQIF greater than 1 implies higher software quality from the use of pair programming.

Empirical studies may be conducted in a university or in the industry.

We classify the empirical research investigated by the type of the subjects, which can be either college students or professionals that work in the industry. Part-time students who are simultaneously working as programmers are

Table 1.1: Studies with students being the subjects

	Subject	Task	SF	SQIF	Test
[30]	13 solos 14 pairs	Web unknown size	Start: 1.3x Later: 1.8x	1.2x	No
[20]	11 solos 5 pairs	C/C++ 300 LOC 3 hours	1.0x	1.0x	No
[8]	24 people change roles	2h/task 1 simple 1 complex	Simple: 1.6x Complex: 1.0x	N/A	Yes
[24]	8 solos 6 pairs	JAVA 4000 LOC	UC 1-10: 1.4x UC 1-4: 1.0x UC 5-10: 2.1x	N/A	Yes
[26]	30 solos 30 pairs	C++ 45 hours	Simple: 1.1x Complex: 1.8x	N/A	Yes
[25]	15 people	2 hours	1.3x	2x	Yes

regarded as professionals. We make this distinction due to the acclaimed difference of expertise between the two groups which is sometimes questioned to threat the validity of empirical research gained from a classroom setting. Skeptics sometimes regard students' insufficiency of knowledge and experience as one major cause to the optimistic pair programming experiment results. In practice, using student as subjects is convenient and economical, while implementing a controlled pair programming experiment in industry could be very costly and challenging.

Williams reported a decent speedup and meaningful increase in software quality from the use of pair programming.

We summarized the findings of empirical pair programming research in the academic setting in Table 1.1. The

field "Test" indicates if certain statistical hypothesis testing was performed in order to establish statistical significance of the result. Among those research, the one that attracted software practitioners the most were the experimental findings in University of Utah in 2000 by Williams [30]. There a speedup of 1.8 times after the so-called pair jelling process was reported. Pair Jelling, as defined in [30], means the activities that a pair needs to get used to each other's working habit and make proper transitions. A jelled team was claimed there to yield much higher productivity. In the study, it is observed that 20% more test cases was passed for pair-programmed artifacts. Though the conclusion was rather bright, important parameters of the experiment such as the type, length of the task were unknown, and no statistical testing was performed. These issues somehow degrades the paper's persuasiveness.

Small scale experiments conducted by Nawrocki, Canfora, Padmanabhuni yield rather conflicting results.

Right after Williams, Nawrocki carried out a similar controlled experiment in Poznan University of Technology, Poland [20]. The result was, however, rather pessimistic. There was nearly neither improvement in the development time nor enhancement in the program size and the number of re-submissions to get the program pass an automated test. Nevertheless, a 50% standard deviation was observed for the time needed by pairs compared to solos, which implied better predicability of effort estimation. Canfora [8] had an experiment focusing on productivity, where 1.6 and 1.0 times of improvements were observed for one simple and one complex task, respectively. Here it is rather counterintuitive that pair programming gained almost nothing in the more complex task. But still the study confirmed a much lower variance in the development duration for pairs. Another study [25] of a similar scale in terms of number of subjects and program size instead reported more optimistic numbers, with 1.3 times speedup and halved defect counts. In summary, the above three research leads to conflicting results. Nevertheless, it should be reminded that the programming tasks implemented in [20][8][25] were only about 150 to 400 lines of code or 2 hours of working time. Such size was apparently too small either in the industrial setting or in terms of achieving statistical significance. Limited development time might also restrain pairs

to jell and thus prevented productivity improvements.

Empirical studies with larger program size are discussed.

To overcome the threat from small program size, empirical research that employees larger projects were carried out. In 2005 it is observed from a Java project that pair programming can provide the speedup of 1.4 times overall and 2.1 times in a later period of the project [24], which again suggests the potential influence of pair jelling. Another more recent study [26] involving more subjects and about 50-hour developing time had a bit worse findings, with 1.1x speedup and 1.5x speedup on the simple and complex tasks, respectively. Note that here the effectiveness of pair programming grows as the task difficulties increases.

The observation of pair programming in a the classroom setting was not consistent but rather bright.

In short, although the results reported from the experiments conducted in the classroom setting seem to be conflicting, with [20] being the most exceptional one, the overall trend was toward the positive side. In general, certain improvement in both time and quality was observed. Specifically, the SF index was around 2 in [30][24][26]. If we consider the additional benefit to software reliability and code readability improvement, adopting pair programming could be a good deal.

Industrial studies are more difficult to carry out.

Apart from the evidence found in college, we also summarize the influences of pair programming in the context of professionals in Table 1.2. Unlike students, companies are usually not willing to invest on a large-scale well-controlled experiment where several teams of solo and pair programmers are established to develop the same software product. Some research [21][16] overcame this problem by enforcing very small tasks. Lui and Arisholm conducted larger experiments lasting approximately one day [17][3]. Studies [14] [22][2][23] did not structure an experiment but instead compare different but similar projects using different development patterns. Though the similarity between the compared artifacts is to be questioned, these papers involve formal-executed projects of reasonably big size which went into production.

Table 1.2: Studies with professionals being the subjects

	Subject	Task	SF	SQIF	Test
[14]	10 people	Fortran 50K LOC	2.3x	three order	No
[21]	5 solos 5 pairs	DBCC 45 min	1.4x	1.3x	Yes
[16]	5 solos 5 pairs	Algorithm questions Case 1: multiple choice Case 2: problem-solving	Case 1: 1.7x Case 2: 1.2x	Case 2: 1.7x	No
[22]	9 people	8500 LOC	3.2x	1.7x	No
[17]	8 solos 8 pairs	SQL/Web 7-10 h	1.5x	N/A	Yes
[3]	99 solos 98 pairs	JAVA 5-8 h	1.1x	1.1x	Yes

Several studies in the industrial setting using small program sizes are summarized.

As mentioned in the last section, Jensen reported a super-linear speedup and incredible quality enhancement adopting pair programming in 1975 [14]. In 1998 Nosek carried out the first well-know controlled experiment in which the developers were asked to write a database consistency check script [21]. The task was indeed small so that it took the pairs and solos 30 and 43 minutes to finish. Artifacts were evaluated by two judges by the work's READABILITY and FUNCTIONALITY, which somehow outlined the level of quality. Pair-programmed script enjoyed 1.4 times higher READABILITY and 1.1 times higher FUNCTIONALITY. In 2003, Lui [16] concluded that pairs solved algorithmic problems faster and reached much higher correctness. A further study in 2006 conducted by the same

research group required the subjects to complete a FIFO warehouse in a weekend. The experiment was called repeat programming such that the exactly same task was repeated four times by the same group of subjects. There the main goal was to investigate the effectiveness of pair programming to a task with different level of familiarity. It followed that pair programming contributed 1.5 times speedup in the first try but then this number decreased dramatically with almost no speedup in the last iteration. This somehow confirmed the generally believed hypothesis that pair programming tends to be more beneficial to unfamiliar, difficult tasks.

A large-scale empirical research involving about 300 professionals concludes insignificant effect of pair programming.

The largest structured experiment in the industrial-setting might be [3]. There nearly 100 individuals and 100 pairs of IT consultants from several international companies such as Accenture, Oracle in Norway, Sweden, and UK participated in the study. Programmers were classified into three categories: junior, intermediate, senior, according to their expertise in Java, in order to further investigate the effectiveness of pair programming among different groups. The task was to perform changes on a Java software system given a set of requirement. There was, however, no statistical evidence showing any improvement to either development duration or correctness among all participants. Nevertheless, if we concern only subjects of certain level of expertise, then intermediate developers worked 1.4 times faster if they were paired up. Paired junior programmers produced 1.7 times of correctness than those solo juniors. Specifically, if the difficulty of subtasks was considered, then the SQIF was 2.5 times for the juniors. In short, the study implied better efficiency when pairing-up intermediate developers and higher quality when pairing-up junior developers.

A company making medical products shares a positive experience adopting pair programming.

A process shift to a pattern similar to pair programming which they called ”Tightly Coupled Engineering Team (TCET) ” in Guidant Corporation was reported [22]. The task, which consisted of about 8500 LOC, was to deliver an automated testing program for an embedded health-care device. There were ten members working on the project.

Table 1.3: Speedup Factor and Effort Multiplier Conversion

Speedup F.	Effort Mult.		Speedup F.	Effort Mult.
0.6	3.33		1.6	1.25
0.8	2.50		1.8	1.11
1.0	2.00		2.0	1.00
1.2	1.67		2.2	0.91
1.4	1.43		2.4	0.83

Some reasons that motivated the transition to TCET were (1) code reviews were not effective, (2) it took too long to learn and become effective, (3) it was difficult for projects to recover from setbacks of losing people. Nevertheless, TCET was not strictly enforced but instead the project owner allowed the developers to decide when to pair-program. In the end the project was a successful one. A comparison to a previous similar project was then undertaken. Evidence showed no significant speedup but 40% defect density decrease for the TCET project.

A table converting Speedup Factor to Effort Multiplier is provided.

While in Table 1.1 and 1.2 the performance index Speedup Factor is used in order to provide a more intuitive understanding, the management may be more interested in the change of effort which directly relates to the project cost. For this purpose, a mapping between the Speedup Factor index and Effort Multiplier is provided in Table 1.3. Having Effort Multiplier equivalent to one means no change from solo programming to pair programming. An Effort Multiplier greater than one indicates more effort to be allocated to a pair programmed project. For example, if Effort Multiplier equals 1.4, then 1.4 times of effort is needed compared to solo programming.

A few qualitative study in the industrial setting is summarized.

Apart from Table 1.2, a few more experiences using pair programming in the industry are summarized. According to Chrysler's experience [2], the use of pair programming led to fast progress, high quality artifacts, better readability, including consistent naming and similar coding style. It was reported that almost the only defect were written by

individual programmers.Vanhannen [23] summarized the perceived effects in a medium-sized Finnish software product company according to an internal survey. There pair programming was found to yield very positive effects on cross-learning, code-readability, and satisfaction, and small yet clear reduction on defect density. Begel [4] did a survey to Microsoft software developers on the topic of pair programming. The result indicated that 22% of the surveyed had been pair-programmed, but only 3.5% were using the pattern in their current project. Those who have involved in a pair-programming project reported the pattern's top three benefits: fewer bugs, spread code understanding, higher quality code, and top three problems: cost efficiency, scheduling issues, and personality clash.

Evidence supporting the use of pair programming is somehow weaker from the experiments done with professionals.

In summary, compared to the evidence implied from the academic-setting, the findings based on software development professionals is a bit dull. Here the SF index seldom came across 1.5, which seems to be less persuasive for the management to make a transition. Improved quality is only observed in certain context, and might depend on the expertise of developers.

Several potential threats might influence the validity of previous research.

A few additional threats to the validity of previous mentioned studies exist. For example, the subjects of the experiments were mostly new to pair programming, and thus might perform worse because they were not adapted to. Also, the reader can easily observe from Table 1.1 and 1.2 that the duration of several experiments were relatively short so the subjects might not have enough time to learn how to pair programming well, and thus could not come across the improvement ravine as claimed by [12]. In some experiments[30], the pairs were formed by themselves, which in turn potentially caused them to collaborate smoothly. Nonetheless, this might not be the common case in a company, where each one's partner is usually unpredictable.

Pair programming, based on previous empirical studies, does work, but might not be so effective as it was originally claimed.

In conclusion, most empirical studies implied a speedup between 1 and 2. Note that 1.5 times speedup means

roughly 30% additional effort to be allocated for each task. Quality enhancement brought by pair programming is roughly confirmed, while the value of such improvement still needs to be discussed.

1.4 Discussion

A mixture use of pair programming and solo programming is possible.

Having discovered the inconsistency of the effectiveness of pair programming revealed in previous empirical research, we now discuss the situations in which the pattern might be more beneficial than others or vice versa. In fact, according to the survey made in [33], 50% of the programmers claimed working alone 10-50% of the time to be acceptable.

Some tasks could be better done individually.

According to previous surveys, programmers thought that simple, well-defined, rote coding could be more efficiently done by a solitary programmer and then reviewed with a partner instead of working in pairs. Detail-oriented tasks such as drawing the user interface were not recommended for pair programming. Many prefered to do experimental prototyping, tough, deep-concentration problems, and logical thinking alone [33][13]. This somehow implies that working individually could be more suitable for exploring design alternatives, and prototyping [30][13]. Note that doing so does not violate the Rules of Extreme Programming as long as the prototyped code will not be used for production [27]. In fact, exploring new problems individually might result more design choices and avoid some valuable idea to be hindered.

Pair design could be very beneficial.

While the focus of this paper is on pair programming, recently several studies indicated that working in pairs can be beneficial in design activities. Several research [1][6][7] have reported different levels of quality improvement in pair-designed solution, which potentially decreases the probability of proceeding to implementation with a bad design. If an inappropriate design was made in the early stage, it would be painful to revert in a later time step [30].

Al-Kilidar [1] carried out a structured experiment among 150 final-year undergraduates including two design tasks. Design quality was measured according to functionality, usability, portability, and maintainability. 24% quality increase was observed in the first task while indifference of quality was identified for the second task. Canfora carried out an experiment in both academic and industrial setting reported that the solution worked out from a pair-design had a statistically significant higher quality.

The effectiveness of pair programming might depend on the developer's expertise with respect to the problem.

When coding, pairing-up junior developers can significantly reduce defect count, specifically for complex tasks [3]. Since each one's level of expertise differs in each field, this result somehow implies that when the task is too challenging such that one developer is hardly able to take, then pairing-up should be concerned instead. In addition, Hulkko suggested using pair programming when there were many module dependencies in the project [13]. Finally, pair testing, among all, was claimed to be the least critical one, as long as the pair developed the test cases together. [30].

Pair programming should not be used for tutoring.

While evidence shows pair programming can be good for cross-training, the rule of Extreme Programming suggests not using pair programming for mentoring. It is said that "A teacher-student relationship feels very different from two people working together as equals even if one has significantly more experience." [27]. That said, cross-learning might better be taken as a side affect of pair programming, but potentially not be the purpose.

Solitary programming plus one additional review phase could result the same effort and reliability as pair programming did.

Since the speedup resulted from pair programming was not completely obvious, an alternative of solitary programming plus one additional review phase is considered to achieve the same quality instead [19]. A controlled experiment was conducted in University of Karlsruhe among eight senior students to compare the difference of time spent and quality between pair programming and individual coding with code review. However result showed

statistical indifference in both total effort and number of passed test cases.

Pair programmers needs more frequent rests. Individual works must be deeply reviewed.

If one decided to adopt pair programming, some hints are summarized as to improve the effectiveness. Since working in pairs yields pair pressure, researchers suggests taking a break periodically for maintaining the stamina for another round of productive pair programming [33]. If not all work are done in pairs, then those artifacts completed individually might better go through a thorough review before they are incorporate [30]. [29] also provided several tips for pair programming to work out. Typically these included but not limited to: do not point out the mistake instantly, change role when one gets tired, find a matched partner, and resolve disagreement wisely.

A possibility of combining solo and pair programming is discussed.

While advocators of solo programming and pair programming seem to compete all the times without clear a result, effort aiming at uniting the two process into one has yet started [18][15]. These papers view solo and pair programming as two complementary processes instead of competing ones. In [18] a new pattern called Software Process Fusion (SPF) was introduced, which promoted alternations between solo and pair programming with several predefined transition conditions. For example, a change to solo programming when the task to be dealt was familiar or trivial. SPF was carried out in two companies' IT department and yielded a very promising result. While the scale of the experimented projects was small, these research point out a new direction toward merging the two processes [15]..

1.5 Conclusion

Pair programming speeds up the development but generally requires more efforts.

Pair programming, being a software development pattern that requires two programmers working on the same problem in front of the same screen using a set of input devices, typically speeds up the development around 1 to 2 times, with the findings being more positive in the academic setting than in the industrial setting. This result in-

dicates a potential increase of effort when choosing the pair programming pattern.

Pair programming yields higher software quality and providesseveral other benefits.

In addition to economic efficiency, several benefits that could be brought from pair programming can be found in recent empirical researches, with improved software quality being the most remarkable one. Apart from that, pair-programmed code typically has better readability. Developers perform cross-learning when they are paired and potentially gain much more self-satisfaction and self-confidence.

Pair programming appears to be effective only in certain contexts. A new form of programming pattern combining individual and pair programming is on demand.

As pair programming does not perform consistently well in all contexts, further investigations on the proper situations to use the pattern are on demand. For example, the impact of different pairing strategies is still not clear. Recent observations implies pair programming to be more useful in designing activities and complex, unfamiliar implementation tasks but maybe not for simple problems and testing activities. Students and junior software engineers seem to benefit the most from pair programming. A combined programming paradigm extracting the merits of solo and pair programming is potentially on demand and to be investigated.

Bibliography

[1] H. Al-Kilidar, P. Parkin, A. Aurum, and R. Jeffery. Evaluation of effects of pair work on quality of designs. In *Proceedings of the 2005 Australian conference on Software Engineering*, ASWEC '05, pages 78–87, Washington, DC, USA, 2005. IEEE Computer Society.

[2] A. Anderson, R. Beattie, K. Beck, D. Bryant, M. DeArment, M. Fowler, Martin amd Fronczak, R. Garzaniti, D. Gore, B. Hacker, C. Hendrickson, R. Jeffries, D. Joppie, D. Kim, P. Kowalsky, D. Mueller, T. Murasky, R. Nutter, A. Pantea, and D. Thomas. Chrysler goes to "extremes". *Distributed Computing*, pages 24–28, Oct. 1998.

[3] E. Arisholm, H. Gallis, T. Dyba, and D. I.K. Sjoberg. Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Trans. Softw. Eng.*, 33(2):65–86, Feb. 2007.

[4] A. Begel and N. Nagappan. Pair programming: what's in it for me? In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '08, pages 120–128, New York, NY, USA, 2008. ACM.

[5] B. Boehm and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2004.

[6] G. Canfora, A. Cimitile, C. Aaron Visaggio, F. Garcia, and M. Piattini. Performances of pair designing on software evolution: a controlled experiment. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '06, pages 197–205, Washington, DC, USA, 2006. IEEE Computer Society.

[7] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. A. Visaggio. Evaluating performances of pair designing in industry. *J. Syst. Softw.*, 80(8):1317–1327, Aug. 2007.

[8] G. Canfora, A. Cimitile, and C. A. Visaggio. Empirical study on the productivity of the pair programming. In *Proceedings of the 6th international conference on Extreme Programming and Agile Processes in Software Engineering*, XP'05, pages 92–99, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] L. Constantine. *Constantine on Peopleware*. Yourdon Press Computing Series. Yourdon Press, 1995.

[10] J. Coplien. *A generative Development-Process Pattern Language*. Cambridge University Press, 1995.

[11] N. V. Flor and E. L. Hutchins. *Analyzing distributed cognition in software teams: a case study of team programming during perfective software maintenance*, pages 36–64. 1991.

[12] M. Fowler. Pairprogrammingmisconceptions, 2006.

[13] H. Hulkko and P. Abrahamsson. A multiple case study on the impact of pair programming on product quality. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 495–504, New York, NY, USA, 2005. ACM.

[14] R. W. Jensen. A pair programming experience. *The Journal of Defense Software Engineering*, Mar. 2003.

[15] K. Lui and K. Chan. Software process fusion by combining pair and solo programming. *Software, IET*, 2(4):379 –390, aug. 2008.

[16] K. M. Lui and K. C. C. Chan. When does a pair outperform two individuals? In *Proceedings of the 4th international conference on Extreme programming and agile processes in software engineering*, XP'03, pages 225–233, Berlin, Heidelberg, 2003. Springer-Verlag.

[17] K. M. Lui and K. C. C. Chan. Pair programming productivity: Novice-novice vs. expert-expert. *Int. J. Hum.-Comput. Stud.*, 64(9):915–925, Sept. 2006.

[18] K. M. Lui and K. C. C. Chan. Software process fusion: uniting pair programming and solo programming processes. In *Proceedings of the 2006 international conference on Software Process Simulation and Modeling*, SPW/ProSim'06, pages 115–123, Berlin, Heidelberg, 2006. Springer-Verlag.

[19] M. M. Müller. Are reviews an alternative to pair programming? *Empirical Softw. Engg.*, 9(4):335–351, Dec. 2004.

[20] J. Nawrocki and A. Wojciechowski. Experimental evaluation of pair programming. In *Proceedings of the 12th European Software Control and Metrics Conference*, Apr. 2001.

[21] J. T. Nosek. The case for collaborative programming. *Commun. ACM*, 41(3):105–108, Mar. 1998.

[22] A. Pandey, N. Kameli, A. Eapen, C. Miklos, F. Boudigou, I. Sutedjo, M. Paul, V. Vijay, and W. Mcdermott. Application of tightly coupled engineering team for development of test automation software - a real world experience. *Computer Software and Applications Conference, Annual International*, 0:56, 2003.

[23] J. Vanhanen and P. Abrahamsson. Perceived effects of pair programming in an industrial context. In *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, EUROMICRO '07, pages 211–218, Washington, DC, USA, 2007. IEEE Computer Society.

[24] J. Vanhanen and C. Lassenius. Effects of pair programming at the development team level: an experiment. In *ISESE*, pages 336–345. IEEE, 2005.

[25] M. Y. S. M. Venkata Vinod Kumar Padmanabhuni, Hari Praveen Tadiparthi. Effective pair programming practice- an experimental study. *Journal of Emerging Trends in Computing and Information Sciences*, 3(4):471–479, Apr. 2012.

[26] V.Venkatesan and A.Sankar. Adoption of pair programming in the academic environment with different

degree of complexity in students perspective an empirical study. *International Journal of Engineering, Science and Technology*, 2(9), 2010.

[27] D. Wells. The rules of extreme programming, 1999.

[28] L. Williams. *Pair Programming*. Oreilly Series. O'Reilly Media, Incorporated, 2010.

[29] L. Williams and R. Kessler. *Pair Programming Illuminated*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[30] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair programming. *IEEE Softw.*, 17(4):19–25, July 2000.

[31] L. Williams, D. S. McCrickard, L. Layman, and K. Hussein. Eleven guidelines for implementing pair programming in the classroom. In *Proceedings of the Agile 2008*, AGILE '08, pages 445–452, Washington, DC, USA, 2008. IEEE Computer Society.

[32] L. Williams, A. Shukla, and A. I. Anton. An initial exploration of the relationship between pair programming and brooks' law. In *Proceedings of the Agile Development Conference*, ADC '04, pages 11–20, Washington, DC, USA, 2004. IEEE Computer Society.

[33] L. A. Williams and R. R. Kessler. All i really need to know about pair programming ilearned in kindergarten. *Commun. ACM*, 43(5):108–114, May 2000.

Typeset August 7, 2012