When Does a Pair Outperform Two Individuals?

Kim Man Lui and Keith C.C. Chan

Department of Computing
The Hong Kong Polytechnic University,
Hung Hom, Hong Kong
{cskmlui, cskcchan}@comp.polyu.edu.hk

Abstract. This paper reports experimental measurements of productivity and quality in pair programming. The work complements Laurie Williams' work on collaborative programming, in which Pair Programming and Solo Programming student groups wrote the same programs and then their activities were measured to investigate productivity, quality, etc. In this paper, Pair and Solo industrial programmer groups are requested to complete algorithm-style aptitude tests so as to observe the capability of solving algorithms in singles and in pairs. So doing is independent of the familiarity of a programming language. Besides, we also take another approach to examining pair programming. A single group of industrial programmers carries alternately out Pair Programming and Solo Programming. All these demonstrate that productivity in pair programming hinges upon algorithm design at all levels from understanding problems and implementing solutions. In addition, we reach similar conclusions to Williams. Our findings indicate that simple design, refactoring, and rapid feedback provide an excellent continuous-design environment for higher productivity in pair programming.

1 Introduction

Many veteran software development managers by experience learn the following phenomenon. A newly employed developer takes days to complete a program. If he continues to work on other programs of similar types, the time for algorithm design is substantially reduced even if he writes them from scratch, i.e. without referring his previous source. Two months later the developer will be able to finish a program of that kind in a day, whereas in the past it would have taken him several days. The time reduction is obvious. As time passes, he becomes mastering the technique of coding that problem. However the amount of time that is required to write that program remains more or less constant. The time reduction is much less significant when compared with earlier.

Taking the above phenomenon to an extreme, we conduct an experiment in which a subject is asked to write the same program 8 times. Figure 1 illustrates the result. At the beginning, it takes him around 6 days to complete. Then he can complete it faster because he has been learning about the problem and becoming familiar with converting the problem into a computer algorithm. The more times he writes the same program, the less time he needs to think about the algorithm. Thus eventually, when he starts writing the same program again, his mind already has a full picture of what the algorithm and the structure of the program are. Then he just sits down and codes it without pausing for thought. In Figure 1, the bigger time difference between the first and the eighth times of writing a particular program means that a developer has spent much effort on designing the algorithm and planning the structure of the program. For some other programmers, they might take five days at the first time and eventually take two days at the eighth time. However, the curve remains steadfast in its shape, meaning that the relationship is always held independent of capability of people and complexity of programming.

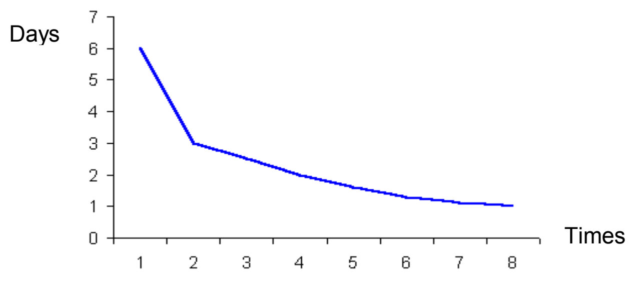

Fig. 1. How much time can be reduced to write a program between the first time and the eighth time?

Let us consider two situations. First, a programmer collaborating with another guy works on coding a brand new problem. Second, a programmer collaborating with another guy but works on a problem that similar ones has been worked on for the eighth time. Which one is more productive? Using pair programming when developers encounter an unfamiliar problem can be much more productive than when they have handled a familiar problem. According to Figure 1, having more people does not help any time improvement when the problem was developed at the eighth time. But there is a room for accelerating the work at the first time. It can then be realized that pair programming will help to speed up programming a problem at the first time more than at the eighth time.

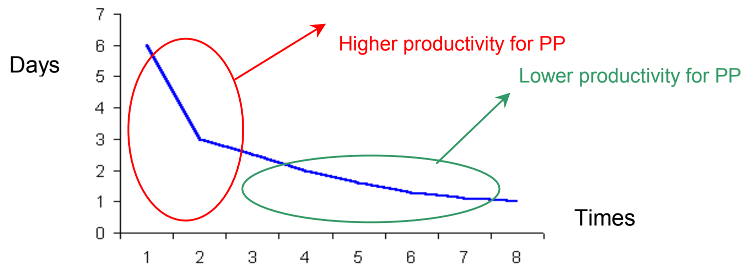

Fig. 2. Productivity for Pair Programming (PP)

Figure 2 shows when pair programming can achieve higher productivity regarding to Figure 1. Based on Figure 1, we can further interpret it as a case in which we achieve *higher* productivity and *better* quality if there is a significant time difference between two consecutive writings for the same problem being programmed.

This paper investigates quality and productivity in Pair Programming. Many experiments were conducted to establish that pair programming improves algorithm design (i.e. semantics and logic of a program) more than use of a computer language (i.e. syntax of a program).

2 Background

A pair spends an insignificant 15% more time than individuals on the same task, yet pair programming achieves a higher quality [1, 2]. These findings were based on an experimental work in which pair-programming and solo-programming student groups were formed and asked to write the same programs so their results could be directly compared.

This paper investigates if there is a repeatable, measurable test that can show substantial differences of "productivity" and "quality" between solo programming and pair programming. Such a test helps to gain insights into pair programming. Thus, we launched a research project aiming at understanding when pair programming is effective. In order to have new contributions and less arguable to our findings, two requirements must be satisfied. First, we tried to make our experiments as closed as an industrial environment. Thus, subjects who participated in this project were full-time industrial programmers. They did programming at least 40 hours a week. Second, we endeavored to make our experiments and results repeatable, so that those who are skeptical of the idea of pair programming can repeat them to judge for themselves.

3 Pair Programming

The fact that instruction in computer programming (i.e. computing algorithms) improves problem solving ability is supported by empirical data and experimental studies [6]. In addition, collaborative learning was found to enhance problem-solving skills [5]. From these, we formulate our hypothesis that people working in pairs can improve problem-solving ability; therefore, pair programming reduces bugs and increases productivity.

We need to establish that people in pairs significantly help solve an algorithm, *the basic element of a computer program*, better and faster. This should not be confused with better knowledge of a particular computer language. Of course, a pair of programmers can help each other to master the computer syntax. However, the key value of talented programmers is their skill in analyzing a problem and devising a computing algorithm for it better and faster. Therefore, we devised our experiments in which subjects in pairs and in solos are asked to solve (1) deduction problems and (2) procedural algorithms, *instead of writing a program*. This way eliminates a factor of how well those subjects are used to particular languages such as Java. In fact, should pair programming be merely beneficial for the use of language commands, two persons could work independently and sit very closely so that they can talk and share their knowledge of language syntax and command.

Deduction problems in connection with programming can be best explained by Game of Live shown in Figure 3, which was introduced by the mathematician John Conway [3]. Solving this deduction problem requires working out an algorithm that describes the problem. Game of Live can easily be programmed into a computer if its algorithm is figured out. The deduction problem can measure a developer's capability of problem reasoning, formulation and representation. Given a problem, we first solve it, then devise an algorithm (a pseudo-program) and lastly code it using a particular language.

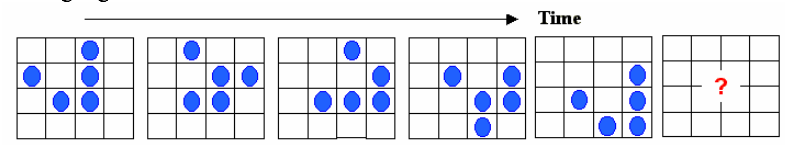

Fig. 3. Does a pair solve a deduction problem like "Game of Live" faster than an individual?

Procedural analysis is an extension of tests for aptitude in logical thinking and progress operations, which were actually designed to access aptitude in areas of competence essential to computer programming. The Procedural analysis measures a developer's capability of coping with complexity. Figure 4 shows an excerpt of a procedural algorithm selected in [4].

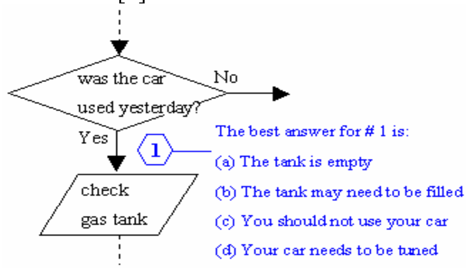
was the car used yesterday?
No
Yes
1
check gas tank
The best answer for # 1 is:
(a) The tank is empty
(b) The tank may need to be filled
(c) You should not use your car
(d) Your car needs to be tuned

Fig. 4. The analysis of the flow chat demands the same type of intellectual activity in programming.

We now describe our experiments. There are two approaches to conducting them. One is to divide subjects into pair and solo groups and compare their results. Another is to ask all the subjects to alternatively work in pairs and then work individually in an alternating style.

3.1 Research Methodology I

Fifteen hands-on programmers from different companies were invited for the experiment at our software laboratory. They had a diversity of backgrounds including database systems, Web applications, etc. Table 1 shows their numbers of years of experience in software development.

Years of programming experience	1	2	3	4	
Number of Subjects	4	9	1	1	Total=15

Table 1. Background of Subjects

Key process areas of CMM and practices of eXtreme Programming (XP) were introduced to the subjects for six hours and five hours, respectively. It should be noted that CMM was introduced to allow the subjects to comprehend XP from a CMM perspective. After introduction to CMM and XP, they were organized in three XP teams (of six, five and four people) and each team was asked to write an inventory system in 17 hours. A business analyst well-versed in operations of sales and distribution was invited to act as an on-site customer.

The purpose of tutorial and workshop is to provide enough hand-on practice to the subjects so that they would do our experiment in a manner of pair programming. Although our control experiment could be performed on subjects without receiving XP training, we believe that XP training is necessary because the subjects learn how to collaborate with a teammate. Measurements for two programmers, never working in pair before or , do not truly reflect the productivity of pair programming.

Now the experiment started. The subjects were grouped into ten units: five pairs and five individuals. The units were asked to complete fifteen multiple choices of procedural algorithm that we extracted from [4]. At the first completion of those questions, we told each subject unit about how many answers were correct. However, we gave no hint on which question were correctly answered. This kind of feedback is shown by "●" in Figure 5.

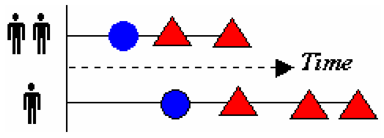

Fig. 5. Visualization of the process of experiment I (Note: where ● and ▲ are two types of checkpoint.)

After the subjects received the feedback, they had to continue to spot wrong ones and figure the answers out. At the second completion, subject units submitted *only* those answers that they previously identified as incorrect. Then we provided feedback on whether all those answers were correct or not, and whether all the problems have been solved, shown in Table 2. The feedback is indicated by "▲" in Figure 5. The process was repeated until the subjects could solve all the questions. The completion time for each repetition was recorded. The feedback mechanism was designed to emulate error messages given by a computer and to minimize the chance of guessing the answer from the feedback

Try	Submitted Answer	Feedback	Solved
1	**1** A **2** A **3** C **4** D **5** C	3 correct	No
2	**1** B **3** B	Incorrect	No
3	**1** C **4** A	Incorrect	No

Table 2: Feedback mechanism

It should be noted that the whole experiment was actually organized as a competition. Subjects were told that they should use less number of submission and less time to *win* it. For those solo units, their personal pride would drive them to challenge pair units. In this sense, solo units in fact were in competition with pair units, more than an experiment. Psychologically, solo units had a strong desire to win whereas pair units simply did not want to lose.

Result 1

Productivity in pair programming can be studied by comparing the results of pair groups and solo groups. At the first attempt, pair groups finished all the algorithms in 13.3 minutes on average, whereas solo groups required 22 minutes. A pair spends 20.9% more time than do individuals on the same task. This was close to what Williams reported of 15%.

So far we have not considered quality. Thus, it is important to compare the time required to correctly tackle all questions, which is the sum of each time of duration. The total time for the pair group was 18.2 minutes and for solo group was 38 minutes.

In consideration of the same quality (i.e. all algorithms being solved), a pair spent 4.2% less time than did individuals on the same tasks. Pair programming is strong on the design of programming algorithms. A pair outperforms two individuals for the design-driven activity.

3.2 Research Methodology II

The objective of the second experiment was to observe the behavior and performance of the same group of people working both in singles and in pairs. Thus, subjects were not divided into pair and solo groups. They work in both groups in shift (see Figure 7). This configuration minimizes the influence of differences of people intelligence and experience.

We prepared two sets of deduction problems, denoted “A” and “B”, respectively. Unlike the algorithm problem (Figure 4), *none* of the deduction problems were multiple choices (Figure 3). First, subjects were formed into pairs and attempted to complete the “A” set of problems. After completing the exercises, then the pairs were separated into individuals and each of them complete the “B” set of problems. Then individuals were formed back to pairs to correct any mistakes in their answers to the set “A” problems. The process was repeated until all problems were correctly solved. The feedback mechanism was the same as in Research Methodology I. Figure 7 depicts the process of this research methodology.

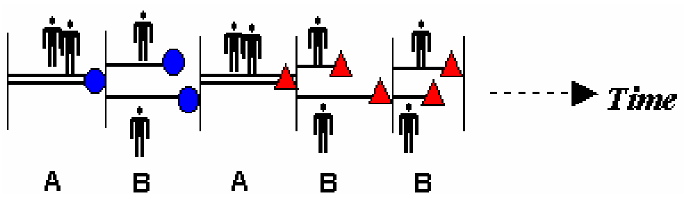

Fig. 7. Visualization of the process of experiment II (Note: where ● and ▲ are two types of checkpoint.)

Result 2
At the first attempt, on average, the groups in pair spent on 65 minutes for the set "A" whereas they in singles completed another the set "B" in 75 minutes. Thus, the pairs spent much more time to complete the deduction problems than did individuals. However, the pairs achieved 85% of correctness and the individuals only reached 51%. Thus the pairs could solve all the deduction problems at the second attempt, whereas the individuals needed more attempts and time to carry out spotting and reworking. It was doubtless that a pair has an increased mental capability since it can search through larger spaces of alternatives [1, 2].

Pairs spent 5.3% less time than individuals on solving their deduction problems. The result also shows that better quality is obtained with pair programming.

4 What Is The Worst Case In Pair Programming

The intuition judgment of many people would be that pair programming spent about 100% more time on the program than the individuals [1,2]. According to the previous, when *ignoring* the quality, pairs did spend much more time. Therefore, we are interested in discovering what the worst situation in pair programming might be. Suppose that coding a program takes an experienced programmer 10 hours and an inexperienced programmer 20 hours. How long would it take to write the same program if both worked collaboratively?

Subjects were asked to write a Transact SQL program. The fast, the middle and the slow programmers are labeled *A*, *B*, and *C*, respectively, in Figure 8. Note that some developers were not familiar with Transact SQL, so wrote more slowly than the others. There was a three-hour gap between the fast and the slow. Then *A* and *C* formed a pair. The pair and *B* were requested to finish another program using Transact SQL. Comparing *B*'s performance with the pair, we conclude that the worst case in pair programming is around the same productivity as the smarter guy of the pair working alone. However, his less-experienced partner can significantly improve his skill at programming.

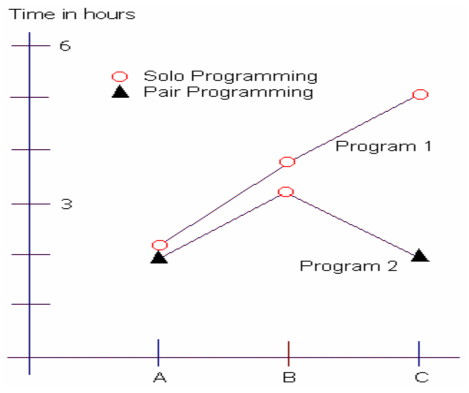

Fig. 8. The productivity of a pair to write a program in the worst situation is the same of the more talented or experienced person of that pair to do it. However, his partner can learn it from him and improve his skill in programming.

4 Conclusions

This paper provides several experimental results to establish a statement that a pair outperforms individuals in working on computer algorithms in terms of quality and productivity. Pair programming excels in procedural problems and deduction questions, which are key elements in programming algorithms. We can conclude that pair programming achieves higher productivity when a pair writes a more challenging program that demands more time spent on design. The finding explains that it is effective to write a program in pair for rapid changing requirements because it demands that programmers concentrate on changing (or continuous) design. In addition, this paper confirms Willam's result.

Our primary contribution, however, is to show when a pair outperforms two individuals. A problem that is new to developers can make pair programming higher productivity because pairs excel to design algorithms faster and better through learning the problem together and exploring larger space of alternatives. pair learning is prolific. In XP, small design, refactoring, rapid feedback altogether forming a continuous design environment for software development also cause pair programming higher productivity because developers are heavily involved in working on algorithms and structures of programs.

Finally, the two methods presented in Section 2 can easily be repeated by those readers who would like to explore more about pair programming in particular and eXtreme programming in general.

References

1. Williams, L. *The Collaborative Software Process*, Ph.D. dissertation, University of Utah, (2000)
2. Williams, L. Pair Programming: Why Have Two Do the Work of One? *Extreme Programming Perspective*, Edited by Marchesi, M, Succi G, Wells, D and Williams, L p.p. 23-33, Addison Wesley, (2002)
3. Kennedy J and Eberhart R, *Swarm Intelligence*, Morgan Kaufmann Publishers, p.p. 17-19, (2001)
4. Munzert, A. Part IV: Computer I.Q. – Program Procedure *Test Your IQ, third Edition,* p.p. 112-117, Random House, (1994)
5. Gokhale, A. Collaborative Learning Enhances Critical Thinking, *Journal of Technology Education* Volume 7, Number 1 Fall (1995). On-line at http://scholar.lib.vt.edu/ejournals/JTE/
6. VanLengen, C and Maddux, C. Does Instruction in Computer Programming Improve Problem Solving Ability? *Journal of IS Education* 12, (1990). On-line at http://gise.org/JISE/