Available online at www.sciencedirect.com
SCIENCE DIRECT®
Int. J. Human-Computer Studies 64 (2006) 915–925
International Journal of Human-Computer Studies
www.elsevier.com/locate/ijhcs

Pair programming productivity: Novice–novice vs. expert–expert

Kim Man Lui*, Keith C.C. Chan

Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Hong Kong

Received 8 July 2005; received in revised form 21 April 2006; accepted 21 April 2006
Communicated by D. Boehm-Davis
Available online 16 June 2006

Abstract

Agile Software Development methodologies have grown in popularity both among academic researchers and industrial practitioners. Among the various methodologies or practices proposed, pair programming, which is concerned with two programmers collaborating on design, coding and testing, has become a controversial focus of interest. Even though some success stories have been reported with the use of pair-programming in real software development environment, many people remain rather skeptical of the claims on pair-programming productivity. Previous studies in pair programming have only addressed the basic understanding of the productivity of pairs and they have not addressed the variation in productivity between pairs of varying skills and experience, such as between novice–novice and expert–expert. Statistical productivity measurements reported by different researchers also seem to lead to contradictory conclusions. Until now, the literature has not addressed how those results and experiments were related to each other. In this paper, we propose a controlled experiment called repeat-programming which can facilitate the understanding of relationships between human experience and programming productivity. Repeat-programming can be performed when controversial issues in non-traditional programming methodologies and development productivity need to be investigated into. To illustrate how the proposed empirical experiment can put arguable, divisive problems into perspective, we have examined the productivity in pair programming as a case study. With repeat-programming, we are able to (i) better understand why results of previous pair programming control experiments reached different conclusions as to the productivity of pair programming and (ii) most importantly, present a case in which novice–novice pairs against novice solos are much more productive than expert–expert pairs against expert solos.
© 2006 Elsevier Ltd. All rights reserved.

Keywords: Programming model; Pair programming; Programmer productivity

1. Introduction

Studies of the applications of cognitive science and programming in computer science started in the 1970s. This area has focused on empirical studies of observable behavior of people as they program. The results of such studies are useful in understanding natural language texts (Atwood and Ramsey, 1978), in educational models (Traynor and Gibson, 2004), in the development of computer aptitude tests for screening job candidates (Mayer and Stalnaker, 1968) and in the modeling of knowledge (Mckeithen et al., 1981; Soloway and Ehrlich, 1984). Previous approaches to the study of computer programming have formulated it as an intelligence-oriented task undertaken by a single programmer. Rarely it has been considered as a multi-disciplinary topic in group dynamics and human intelligence. Further, understanding a highly intelligent task solved by two people in a collaborative manner may also involve fundamental problems of modern informatics, software engineering, time-to-market product development and psychology. For example, pair programming, which involves two developers collaborating as a single individual on the same programming task, has been shown by two experiments to be productive, and to produce higher quality code than either of them may as single individuals alone (Nosek, 1998; Williams et al., 2000). This can be a case where our knowledge of the software development process can be beneficial to other fields such as cognitive science.

*Corresponding author. Fax: +852 2774 0842.
E-mail addresses: cskmlui@comp.polyu.edu.hk (K.M. Lui), cskcchan@comp.polyu.edu.hk (K.C.C. Chan).

1071-5819/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ijhcs.2006.04.010

Human programming is in fact an interdisciplinary topic that is concerned with the internal information mechanisms on one side and the brain on the other side. To build highly intelligent systems, we are interested in using process engineering and software engineering to explain the mechanisms of memory and learning. Such developments will have significance for the development of the next generation computer (Wang, 2002). This paper aims at exploring the two ways of programming, pair programming and solo programming and at advancing our understanding of the programming productivity.

The idea behind pair programming, also known as Collaborative Programming, is straight-forward. It involves two programmers collaborating side-by-side on the design, coding and testing of a piece of software. One, the Driver, controls the keyboard/mouse and actively implements the program. The other, the Navigator/Observer, continuously observes the work with a view to identifying tactical defects and providing strategic planning.

Pair programming has recently drawn much attention because of the increase in popularity of eXtreme Programming (XP) (Beck, 2000), but it was first reported in the workplace in 1995 (Constantine, 1995) and was discussed as early as 1991 in Distributed Cognition (Hutchins, 1995). The researchers in distributed cognition reported that the verbal and non-verbal correlated behaviors of a pair of programmers can help them to search through larger spaces of alternatives (Flor and Hutchins, 1992). From 1998 to 2001, scholars in computer science conducted a number of controlled experiments to evaluate the effectiveness of pair programming. The settings of these experiments had a number of similarities in that subjects were divided into a pair-programming group and a solo-programming group and that the two groups were then asked to write the same program so that their results could be directly compared. Interestingly, the experiments produced different and apparently incompatible results.

In pair programming, time productivity can be measured as a pair takes z% more effort than an individual. To avoid confusion between elapsed time and total time per programmer in our discussion, the z% is interpreted here as "Relative Effort Afforded by Pairs" (**REAP**) against solo programming. The formula for calculating **REAP** is shown in (1):

$$\mathbf{REAP} = \frac{(\text{Elapsed_time_of_pair}) \times 2 - (\text{Elapsed_time_of_individual})}{\text{Elapsed_time_of_individual}} \times 100\%. \tag{1}$$

If **REAP** is zero, pair programming halves the time required for solo programming. When **REAP** is greater than zero but less than 100%, pairs require more total man-hours but do complete tasks faster. This can be useful when the critical issue is time-to-market. The commanding market share advantage that can accrue to early mover companies can make it worthwhile for them to spend more on short-term development costs.

The relative productivities of pair and solo programming have been the focus of much research interest. Nosek (1998) reported that **REAP** was 42%. In contrast, Williams et al. (2000) reported that **REAP** was 15% on the same task yet the quality of pair programming was higher. Nawrocki's study (2001) indicated that pair programming could be less efficient as his experiment showed that **REAP** was 100%.

These experiments do appear to indicate that productivity in pair programming can vary significantly but the absence of an underlying principle makes it hard to account for the productivity variations. There is a need to understand pair programming and the previous programming experiments by an easily reproducible experiment. For example, the findings of an experiment should allow us to judge when pair programming should be adopted over solo programming. In this paper, we propose to investigate into the relative productivities of novice–novice and expert–expert pair programming. Nosek (1998) pointed out that two average or less-experienced workers collaborating to perform tasks may take on a much more challenging task than can two individuals. This is the equivalent of saying that pair programming is much more effective and efficient for inexperienced programmers than for expert programmers. By performing an experiment, we would like to investigate into the differences.

The paper is organized as follows. Section 2 clarifies the meaning of productivity in pair programming and introduces the background to the three experimental studies under consideration in this paper, by Nosek in 1998, Williams in 2000 and Nawrocki in 2001. In Section 3, we propose our experiment, "repeat-programming", which describes the behavior of novice and expert programmers. The experiment seeks to keep many variables constant including programming capabilities, fast and slow programmers and problem comprehension. Thus, the outcome of the experiment can clearly illustrate the productivity of pair programming. Section 4 discusses our experimental results as well as those of Nosek, Williams, Nawrocki and generalizes them into two guidelines for the adoption of pair programming. Section 5 suggests guidelines for future work in software development using pair programming. The final section offers our conclusion.

2. Solo programming and pair programming

It is expected that, as a recognized practice in Agile Software Development, pair programming will continue to attract more attention from academic and industrial researchers. Our study here includes three controlled

experiments in pair programming. The experiments have been the center of controversial articles in the Agile Software Development community. We would like to provide the background in details and to show the importance of our case study to the research community.

The three experiments focus on productivity assessment in pair programming in terms of two variables: time and software quality. The experiments have been widely referred to, in different papers, that discuss pair programming education applications (Müller and Tichy, 2001; Gallis et al., 2003; McDowell et al., 2003), personality analysis (Heiberg et al., 2003), an industrial case study of pair programming (Gallis et al., 2002) and distributed pair programming (Baheti et al., 2002). Critics of pair programming also refer to these three experiments (Keefer, 2002; Stephens and Rosenberg, 2003). These three experiments are also referred to in a piece of empirically based research into seeking programming reviews as an alternative to pair programming (Müller, 2003). In these experiments, subjects are randomly divided into pair and solo groups and are asked to write the same program. The productivity of each group is then assessed in terms of time and software quality. The results contradict each other. The discrepancies in their results have not yet been explained in the literature.

Before discussing the three experiments in question, we would like to unambiguously distinguish two concepts insofar as they apply in pair programming: economics and productivity. These concepts are not clearly defined and hence are often confused.

2.1. Productivity and economics

By economical we mean that programs of an expected quality are produced at the lowest cost. Let software quality be an *index* q and costs be an *index* c. Notice that costs are measured here in monetary units and could be defined as a *salary ratio* $(s) \times$ *effort* (e). For single programmers, *effort* = *elapsed time* (t) of solo programming and for pairs, *effort* $= (1 + \mathbf{REAP}) \times$ *elapsed time* (t) of solo programming. The greater the values of q and c, the greater the quality and costs are. Economics in programming is defined by a quality threshold p such that developing programs satisfy two conditions $q \geqslant p$ and $c = \mathrm{Min}(c)$. Considering non-critical modules of an application, we may expect the quality threshold p to be our minimum acceptability level and we are satisfied with software as long as its quality is more than p. The implication of this is that when developing programming modules that do not require sophisticated programming skill, we would employ recent-graduates as developers as long as their *salary ratio* $\times$ *effort* satisfies the condition of minimum cost.

With regard to pair programming, one may want to ask the question "why employ two programmers when the same job can be done by one?" A related question can, in fact, be also asked "why employ experienced programmers when graduates can do the job?" As shown in Williams experiment in Section 2.3, the software quality of programs developed by pairs passed over 80% of the test cases and had an **REAP** of 15%. If the quality threshold p is set at 80%, pair programming is recommended. However, had we expected the software quality to be just 70% of test cases passed, pair programming would not be adopted because it would not be economical.

By productivity we mean achieving the greatest quality within minimum time. The notion of *costs* used in our preceding definition of "economical" is replaced with that of *time* as we fix a *salary ratio* between different levels of programmers. We have $q = \mathrm{Max}(q)$, $c = \mathrm{Min}(c)$ and the salary ratio $\equiv$ constant. Since $c = \text{constant} \times t$, then $q = \mathrm{Max}(q)$, $\mathrm{e} = \mathrm{Min}(e)$. This gives an idea which programming method (pairs or singles) produces better quality programs in less time per person. Unfortunately, tackling two independent constraints without a common relationship is not possible in mathematics. Thus, we have to simplify our definition. Set a reasonably high-quality constant r such that $q \geqslant r$ and $t = \mathrm{Min}(t)$. The differences between *economics* and *productivity* lie in (i) the relationship that r should be much greater than p $(r \gg p)$ and (ii) the different units of *cost*s, in that money is used in economics whereas time is used in productivity.

A survey of related literature indicated that there have been doubts about the productivity of pair programming (e.g. Keefer, 2002; Müller and Padberg, 2002, Stephens and Rosenberg, 2003). Some of these authors argued against pair programming from the point of view of *economics* and some of them argued from the point of view of *productivity*. Regarding economics, pair programming should be analysed on the basis of (i) productivity and (ii) the salaries of programmers of different skill levels. Therefore, the issue of productivity in pair programming should be answered before the issue of economics. The purpose of the three controlled experiments described in Sections 2.2, 2.3 and 2.4 is to allow pair programming to be analysed in terms of time and software quality.

2.2. Nosek's experiment

In 1998, Nosek reported on his empirical experiment in which fifteen full-time system programmers in five pairs and five singles were asked to write a UNIX script that performed a *d*ata*b*ase *c*onsistency *c*heck (DBCC) in a Sybase database.

In the experiment, the subjects wrote a program that would initiate the command, detect any errors in the log and post a warning email if it found any. Executing the DBCC command returns the status of the database in a log file. For system programmers, these tasks should be straight-forward. The only area in which they lacked experience was DBCC.

The results of this experiment are shown in Table 1. Time was measured objectively. As for software quality, two independent graders evaluated the readability and functionality of the problem solutions and assigned a

Table 1
Results of Nosek's experiment: relative effort overhead for pair programming (**REAP**) is 41.7%

	Singles	Pairs
Elapsed time (min)	42.60	30.20
Readability score (0–2)	1.40	2.00
Functionality score (0–6)	4.20	5.60

readability score between 0 and 2: 0 for an entirely unreadable solution, 2 for an entirely readable solution and 1 for readability in between. Functionality ranged from 0 for a solution not achieving the goal at all, to 6 for achieving the goal entirely. Although the two graders examined the problem solutions with an inter-grader reliability of 90%, human judgment was involved in verifying the software quality.

2.3. Williams' experiment

In 2000, Williams reported a university experiment on pair programming. Forty-one junior and senior university students were assigned in 14 pairs and in 13 singles in a way that there was a sufficient spread of high-high, high-average, high-low, average-average, average-low, and low-low pair grouping based on their GPA. The student subjects were asked to write web scripts that had dynamic contents. They were asked to retrieve and update a Microsoft Access database. The students had approximately 3 years of experience with C++. The applications were similar to those of a typical e-commerce web site (Williams, 2000; Williams et al., 2000).

The students in pairs and in singles completed four assignments over a period of 6 weeks. They recorded the time they spent on the project with a web-based tool. It should be noted that the experiment was not monitored by any person on site and how closely all pairs of the subjects practice pair programming was not known. However, a teaching assistant was involved to execute automated testing to analyse programming quality. The results showed that programmer-pairs passed more of the automated post-development test cases than the singles.

In the first assignment, **REAP** was 60%. After the initial adjustment period, the total programmer hours the pairs spent on the second and third assignments decreased dramatically—**REAP** was 15% on average. Because of data entry problems in the experiment, the completion times for the fourth assignment were not accurate. Thus, the time performance of the fourth assignment was not reported.

Regarding quality, the percentage of pairs passing the test cases was 86.4–94.4% whereas the singles passed 70.4–78.1%, as shown in Table 2.

2.4. Nawrocki's experiment

In 2001, Nawrocki reported 21 fourth-year students were divided into three groups, each group using a different methodology to work on the same assignments: (i) five pairs using XP (called pair-XP in this paper), (ii) five singles using XP but without pair programming (called single-XP in this paper) and (iii) six singles using Personal Software Process (PSP). Nawrocki was primarily interested in two comparisons: (i) single-XP vs. PSP, and (ii) single-XP vs. pair-XP. In this paper, we are interested in the comparison of pairs and singles using the same method. We have selected single-XP vs. pair-XP for our review (for the sake of brevity, in this paper we shall refer to them as pairs and singles).

All the subjects had more than two years of formal study of C and C++. They were asked to write four programs. These were programs for finding the mean and standard deviation of samples of numerical data, finding the linear regression parameters, counting the number of lines in a program and counting the total program LOC.

Their results showed the singles and pairs took around the same amount of time to complete the first three assignments: 2.4, 1.3 and 2.4 h (Nawrocki and Wojciechowski, 2001). This means that **REAP** was 100%. The pairs completed the fourth program in 3.5 h and the singles in 4.3. This indicates that **REAP** was 63%, but Nawrocki remarked that the improvement resulted from the singles misunderstanding the program requirements. The number of re-submissions was counted as the subjects had to rework the program until no errors were discovered during acceptance testing.

As can be seen in Table 3, the number of re-submissions shows that the pairs re-worked slightly less than the singles. Nawrocki concluded that pair programming appeared less efficient than had been reported by Nosek and Williams.

2.5. Summary remarks

There are several differences between three experiments. Both Nosek's and Nawrocki's experiments were conducted in a controlled environment—the subjects worked under direct supervision of the experimenters. In the case of Williams' experiment, the subjects worked at home, so we are not sure whether or not the observer had indeed assist the driver all the time. If they split off at times and worked individually on different parts of the assignment, it would have had quite a strong impact on the completion time. Such an impact has been studied recently (Nawrocki et al., 2005).

Also, the meaning of 'completion time' is not defined the same way in these experiments. In Nosek's experiments there was a variable called FUNCTIONALITY that described the degree to which the strategy accomplished the objectives and that suggested that in that experiment the solutions delivered by the subjects differ in functionality. In Nawrocki's, there were pre-defined acceptance tests that are run automatically and 'completion' meant passing those tests at 100%. In Nawrocki's case, in other words, all the solutions could be regarded as having the

Table 2
Percentage of test cases passed

	Program 1 (%)	Program 2 (%)	Program 3 (%)	Program 4
Singles (test cases passed)	73.4	78.1	70.4	78.1%
Pairs (test cases passed)	86.4	88.6	87.1	94.4%
REAP	60	15	15	N/A

Source: Williams et al. (2000).

Table 3
Number of re-submissions on average

	Average number of resubmissions			
	Program 1	Program 2	Program 3	Program 4
Singles	3.4	0.1	1.6	4.6
Pairs	3.6	0.1	1	3.3
REAP	100%	100%	100%	63%

Source: Nawrocki and Wojciechowski (2001).

Table 4
A summary of the past experiments

	Nosek	Williams	Nawrocki
Subject	Full-time programmers	Students	Students
Sample size	15 (5 pairs, 5 singles)	41 (14 pairs, 13 singles)	15 (5 pairs, 5 singles)
start_posstart_posexperiments	Yes	No	Yes
Quality assessment	Rate by two independent graders	Examine the percentage of passing pre-defined test cases	Pass all pre-defined test cases
REAP (see (1))	41.7%	15%	100%[a]
Number of programs written by subjects	1	4	4
Conclusion	PP is productive	PP is productive	PP is unproductive

[a]Note that it holds only for the first 3 programs of Nawrocki's experiment.

same functionality. Thus, the meaning of completion in Nosek's experiment was not the same as in Nawrocki's.

Although these differences mean that the results of these three experiments cannot be directly compared, their results have all shown the productivity effects on pair programming to be variable. In Nosek's and Nawrocki's experiments, the differences in productivity of programmers with different experience levels were not taken into considerations and programmers formed pairs randomly. In the Williams' experiment, pair and solo groups were academically equivalent. Pairs were formed to ensure there was a sufficient spread. The purpose was to establish an even distribution by academic records rather than by randomization. Hence, the three experiments did not set out a formal pre-assessment process to verify whether their programmer subjects had written similar programs before. They also did not take into account whether the subjects were fast or slow programmers. Thus, the first question in studying productivity in pair programming would be whether **REAP** for novice–novice pairs is the same as for expert–expert. This will enhance our understanding of the different conclusions reached in the three previous empirical work of Nosek, Williams and Nawrocki because, if **REAP**s for novice–novice and expert–expert programmer pairs can be significantly different, the sample size in pair programming experiments must be large enough to average such deviation. In this paper, we are concerned with the continuum from novice to expert.

We summarize the characteristics of the three experiments presented in this section in Table 4 below.

3. Repeat-programming

Owing to the differences in programming experience, novice and expert programmers should perform the same programming job very differently in terms of time and software quality. A novice programmer takes longer to complete a new program. Once he has gained experience by working on that and other problems of that type, he can write faster and better. When it comes to programming skill and experience in forming pairs for pair programming,

therefore, we have two extremes: novice–novice and expert–expert. They have the following characteristics:

(1) *Novice–novice*: this is concerned with putting one novice programming with another novice working as a pair in pair programming. Logically, they should complete the whole program faster. We assume that there will be an x% time reduction. Taking only time into account, 50% is the break-even point since there are two people.
(2) *Expert–expert*: This is concerned with pairing one expert programmer with another expert colleague and these experts become experts after they mastered work on that same kind of problem. More precisely, this is an experienced–experienced pair. Hypothetically, as a pair they should work y% faster.

Before we investigated into these two extreme cases, we assumed that the values of x and y can vary independently. Hence, they may not be very meaningful. However, our general hypothesis is that a relationship like the ratio of change of productivity of pair programming along the experience-scale from novice to expert could be less changed and more conservative; otherwise, there is no way that we can tell when we can adopt pair programming so as to be maximally productive or when a pair outperforms two individuals.

Taking the human performance factor into consideration, we attempted to simulate a situation by experiment where novice programmers are *becoming* expert programmers and to assess their productivity in solo programming and pair programming. The experiment is called repeat-programming because in order to emulate a process of passing from the novice to the expert stage it requires programmers to write the same program several times. We observe the behavior and measure the change of productivity. This section will report the details of the experiment with repeat-programming.

3.1. Initial study

Over 2002–2003, we conducted an initial experiment to examine how programming performance varied when measured along an axis in which developers become more familiar with a programming problem. We called the experiment *repeat-programming*. It included three major processes: (i) selecting subjects with "similar-capability" (pre-assessment), (ii) getting subjects familiar with pair programming (pre-experiment) and (iii) having subjects repeatedly write the same program (control experiment).

From among 63 candidates we selected three whose abilities were similar. We did this by pre-assessment. We chose individuals with the *same* programming knowledge. We split these three into an individual and a pair. The pair is required to practise pair programming (i.e. pre-experiment) before they could proceed with the experiments.

The subjects, the individual and the pair, were asked to write an first-in-first-out (FIFO) warehouse application (see Appendix A) in our laboratory. The tasks were standard—they had to create tables in SQL 2000 and code in JSP.

Each subject wrote the same program eight times, each time starting from the beginning. The result is shown in Fig. 1. The development of two curves is more noteworthy and meaningful than the values indicated by the curves. As discussed, programmers with different abilities can produce different sets of results. However, the trend of the curve remains consistent.

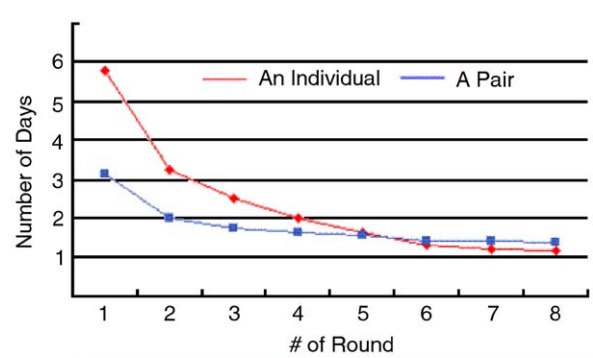
A large sample is needed to confirm the generalization of the result of pair programming vs. solo programming illustrated in Fig. 1. Thus, more empirical studies are required to validate the results of repeat-programming.

3.2. Controlled experiment

In 2005, we conducted "repeat-programming" with a larger sample. The subjects were part-time masters' students who had full-time programming jobs. There were 40 part-time students taking a course on "Agile Software Development and XP" for a double Masters Degree program in Softare Engineering jointly offered by the Hong Kong Polytechnic University, Hong Kong and The Graduate School of the Chinese Academy of Science, Beijing, in 2004/2005.

To minimize disparities in programming abilities, we grouped the forty students so that each group of three subjects had "nearly identical" abilities. We did this by a pre-assessment test, which consisted of fifty multiple choices questions taken from the computer aptitude test proposed by Munzert (1994). Fig. 2 illustrates the assessment results of those forty candidates.

The deviation of their marks was not an issue. We were seeking to form groups of three with each consisting of students with similar capabilities. Each group of three



Individual	5.81	3.25	2.52	2.05	1.63	1.31	1.18	1.17	days
Pair	3.13	2.08	1.76	1.62	1.56	1.43	1.41	1.37	
REAP	7.5	23.1	40	62.5	92.3	119	142.1	134	%

Fig. 1. An initial study on repeated programming: a pair vs. a single.

students was then split into an individual and a pair. Ultimately, we obtained eight groups of three to participate in our experiment.

As the subjects did not have formal experience in pair programming (i.e. continually practicing pair programming for four hours without pairing off), each group of subjects had to practice pair programming before they could proceed with the programming task. They were asked to write Tower of Hanoi as warm-up exercises. The results of this pre-experiment exercise were not recorded. Each group decided on their organization of a pair and a single, which minimized the impact on those who may personally prefer programming alone or may have been biased against pair programming as they could work on solo programming. We also suggested to the pairs that in cases where conflicts arose they should be resolved by the decision falling to the one controlling the keyboard/mouse. Subjects fully understood that dispute and self-assertion would reduce productivity and run counter to the objective of the experiment.

The subjects were asked to write an FIFO warehouse including "in/out operations", "reserved stock", "bin management" and "goods returned" using SQL Server and ASP (or JSP if they did not know ASP). As the subjects were full-time programmers, they were familiar with popular development scripts like SQL and ASP (or JSP). The requirements of the task as used in our initial study are used again except that we simplified the requirements. We did this so that the program could be completed in one weekend (i.e. 2 days) as the subjects had full time jobs. The duration of the experiment was eight weekends, and hence pairs and singles selectively worked on four weekends. The subjects would record the time they spent programming.

A set of 50 test cases were worked out that could be used to assess the software quality. The purpose was to assess software quality of a program through test cases, rather than human judgment. Thus, subjects' programs had to pass 50 test cases. The test cases included application requirements and exception handling. These types of measurement were appropriate because we could objectively measure the quality of the programs written rather than subjectively by relying on human graders and because from a customer perspective, users (i.e. customers) would be more satisfied with software products that had been formally, objectively and extensively tested and they would then by more willing to regard a software product as high quality. Developers and customers, it should be noted, tend to see software quality differently (Friedman and Voas, 1995).

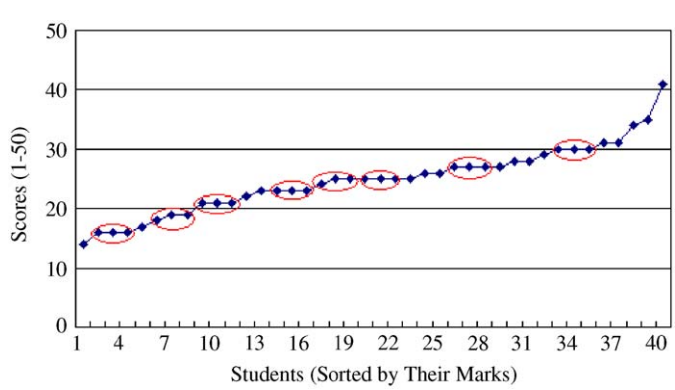


Fig. 2. Pre-assessment to select eight groups of three "similar-capability" programmers.

3.3. Experimental results

On the first round of programming, the individuals completed the program in 637 min on average. Predictably, on the second round, they did it much faster. Clearly, they shortened their learning curve, especially in program design. When we looked closely at each version of their work, we found that, while the overall design was very much the same from version to version, the syntax, the naming standards and ordering of statements, were different in each version.

The pair programmers worked on the same tasks, in the same way, and under the same conditions. In the first round, the pair was much faster than the single, requiring only a little more than 410 to complete the task.

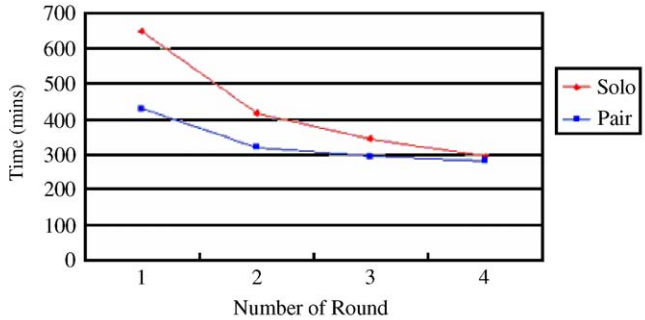
As expected, the finishing time of each unit can vary. If each round is independently considered, the results of each round contradict one another. The effort of the pair was by 29% greater than that of the single on the same task (**REAP** = 29%) while the second, the third and the fourth pairs had **REAP**s of 57%, 69% and 91% (see Fig. 3). Appendix B provides the full results of each.

All the programs submitted by the subjects had to pass 50 test cases. It was unlikely that subjects would be able to pass all fifty test cases at the first attempt. The subjects needed to test the cases in an iterative manner in order to get through them. This was to keep software quality constant.

Both Figs. 1 and 3 show the time taken by the individual and the pair to write the same program on each occasion. The two curves are similar. The trend is far more noteworthy and meaningful than the values indicated by the curves. Programmers with different abilities can produce different sets of results. However, the trend of the curves basically remains consistent (see Appendix B for the result of each group). The characteristic is conservative and is less dependent on whether they are fast or slow coders, or talented or weak programmers. Nor does it make much difference how long the different pairs took to solve the problem in the first round. As shown in Figs. 1 and 3, it is clear that

$f'(n) < 0$, where $f(n) \equiv (\text{elapsed_time_of_individual} - \text{elapsed_time_of_pair})_{\text{at round } n}$.

This means that the productivity of pair programming diminishes when pairs keep solving the same problem. Although in reality no programmer will write a program twice in exactly the same way, programmers are actually



	1 st Round	2 nd Round	3 rd Round	4 th Round
Solo (mins)	635	407	334	285
Pair (mins)	410	320	281	272
REAP	29 %	57 %	69 %	91 %

Fig. 3. Repeat-programming (8 groups of 3 "similar-capability" members, 1 pair and 1 single).

solving many problems of a similar kind using design patterns from experts or out of their own experience (Gamma et al., 1995).

4. Two issues

Repeat-programming shows that pair programming performs well when a pair encounters challenging programming problems. By "challenging programming problems", we mean problems that require more sophisticated and less straight-forward algorithms to solve. This is rarely related to the programming skills of any particular computer language.

This section discusses some of the implications of our experiment. Particularly, we consider how we can understand the discrepancies of **REAP** in the three controlled experiments and how the difference between **REAP** of novice programmers and **REAP** of expert programmers can be helpful in software development.

4.1. Past experimental results

The three experiments on pair programming reviewed in Section 2 produced inconsistent findings. This indicates that pair programming can only be productive in some situations; otherwise there would not be substantial deviations in their results.

Repeat-programming does not account for those properties which are the focus of Distributed Cognition (i.e. environmental controls, social and cultural factors). Thus, our experiment cannot provide full explanations for the discrepancies between the three studies. In fact, it may be that the variance originates from more than a single factor. Tuckman (1965) develops a model to describe how groups of people change through four stages before becoming maximally effective. Thus, two developers may require time to become an experienced, stable and productive pair. The variance in the results of these three studies demonstrates the fit of Tuckman's model as the subjects in Nosek's and Nawrocki's experiments had no experience of pair programming.

In pair programming, Tuckman's model is often known as pair jelling. Pair jelling is the time it takes for a conditioned solo programmer to learn how to be a pair programmer (Williams et al., 2000). Williams data shows that **REAP** = 60% for the 1st program is followed by a stabilization at about **REAP** = 15% for the 2nd and 3rd programs. Similarly, Nawrocki's students got **REAP** = 100% for the first 3 programs, but had a downward **REAP** = 61% with the 4th program.

Although different **REAP**s in the three earlier studies reported in Section 2 probably have more to do with the kind of problem, their discrepancies are supported by repeat-programming. In the Nosek's and William's experiments, their subjects handled more challenging programming problems than the Nawrocki's subjects, which were asked to find the mean and standard deviation of samples of numerical data and count the number of lines in a program. Thus, according to repeat-programming, **REAP**s should be lower in the Nosek's and Williams' and **REAP**s should be higher in the Nawrocki's study.

What repeat-programming illustrates is how the factor of familiarity affects productivity in pair programming while other factors such as pair jelling, albeit having impact on productivity, are less controlled or are not measured in their experiments. This helps us understand better the discrepancies in the three controlled experiments and hence practitioners can be more confident of the adoption of pair programming.

4.2. Principles for novice–novice pairs vs. expert–expert pairs

To be essentially pragmatic and broadly applicable for pair programming in real development, we must first resolve the question of when a pair outperforms two individuals. Thus, from the interpretation of the people performance along time series in repeat-programming, we establish the first principle:

> *Principle (a):* A pair is much more productive in terms of completion time and can work out a better solution in terms of software quality and maintenance than two individuals when the pair is new to a programming problem and more effort is required to produce the design, algorithm, and coding of that program

This principle suggests that pair programming works well when a pair encounters challenging programming problems. Although few people have defined what the term "challenging programming problems" actually means, for us it simply means solving problems that demand more sophisticated and non-straight-forward algorithms to solve and this is rarely related to programming skills of any

Table 5
A potential paradigm derived from repeat-programming for managing inexperienced programmers using pair programming

Step	Activity	By principle (see Section 4.2)
i	Pair up	
ii	Work on design and algorithm and identify patterns of logic	(a)
iii	Code and test sub-programs in pair programming	(a)
iv	When the pair encounters any sub-program in which same logic has been done in pair before, the pair should split off and two programmers independently code (and test) the sub-program in solo programming	(b)
v	Pair up	
vi	Review and perform integration tests	
vii	Go back to step (ii) until completion of assignments	

particular computer language. The second principle we can establish is as follows:

Principle (b): Pair programming can substantially drop in productivity when a pair has had previous experience of the same task and the pair has not yet forgotten that experience

This principle does not address any change of software quality. It simply states the fact that solo programming is faster than pair programming when programmers are working on solutions they have already met. As long as a pair knows a programming solution well enough, it is effective for the person who controls the keyboard and mouse not to interrupt his writing even though he may make small mistakes such as typos. On the other hand, his partner probably feels less challenged by watching the known solution the guy is writing.

To combine these two principles, we conclude 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.

5. Future work

Given the limitation that in repeat-programming programmers with "similar-capability" is in fact an ideal case, the experiment requires further tests. It is for this reason that we do not strictly define the term "similar-capability". Indeed, it is conceivable that different programmers might solve different problem at different speeds.

Repeat-programming described here can provide a basis for future work that seek to develop a pair programming framework that can be embedded in a software method for improving overall productivity and optimizing resources. We have concluded in Section 4.2 that, strictly speaking, in pair programming, a pair of less experienced programmers vs. a less experienced programmer will be much more productive than a pair of experienced programmers vs. an experienced programmer alone.

Thus, Table 5 suggests a pair programming framework based on the principles deduced from the results of repeat-programming. Leaving aside human, cultural and workplace environmental factors, the first principle in Section 4.2 would indicate that experienced programmers in pairs perform well in steps (ii), (iii). Step (vi) allows the pairs to review their work products and to integrate and test their sub-programs, particularly work done in solo programming.

6. Conclusion

Previous studies in pair programming have only addressed the basic understanding of the productivity of pairs, rather than the change of productivity between novice–novice pairs and expert–expert pairs. The three controlled experiments, by Nosek, Williams and Nawrocki, all reported statistical productivity measurements, but their conclusions seemed contradictory. Until now, the literature has not addressed how those results and experiments were related to each other. This paper contributes to advancing our understanding of pair programming by connecting their results with ours (i.e. using repeat-programming to demonstrate the change of the productivity along the scale from novice–novice pairs to expert–expert pairs). We find that pair programming effectively helps developers solve unfamiliar programming problems. A further contribution of the work is to provide an innovative and original approach to analysing empirical software engineering experiments and to exploring behavioural relationships between humans (i.e. programmers), work (i.e. programming) and methods (i.e. pair programming).

Acknowledgements

The authors would like to thank John Nosek for the many insights he contributed to this work. They also thank other anonymous reviewers of IJHCS for their thoughtful comments. In particular, we thank one of them who suggested that we use a term to refer to the relative effort of a pair of programmers. We thank Pekka Abrahamsson and Laurie Williams for their comments on the earlier version of this paper. Finally, we also appreciate very much the Associate Editor, Deborah A. Boehm-Davis, for her useful comments and suggestions.

Appendix A

An FIFO warehouse stores inventory in which movement of goods in and out is based on the principle of FIFO. It is a principle where by goods is supposed to be sold in chronological order in which it was received or produced. Thus, items purchased (or manufactured) first and placed in a warehouse are assumed to be sold before items purchased (or manufactured) at a later date. An FIFO warehouse module can be implemented in three sub-modules as follows:

A.1. Basic configuration

A Warehouse Master should have a:
Warehouse code—a unique code for each warehouse
Name—warehouse name
Number of bins—the number of bins available in a warehouse
Status—a warehouse can be set to active or blocked
A Lot Master should have a:
Lot code—a unique code for each lot
Fixed item status—fixed bin for one item only
Multiple item status—allow for any items to be entered into the same bin
An item master should have
An item code—a unique code for each material
Descriptions—item's descriptions
Price—current price of an item
Average unit cost (calculated by system)—determine the value of the inventory by Average Unit cost × total stored quantity.
Production date—item's manufacturing date
Minimum stock—the minimum reorder level
Maximum stock—the maximum storage level
A lot—product link should have
Item code—item code
Batch date (generated by system)—today's date
Lot code—Warehouse's lot code
Quantity in—item quantity received
Quantity out—item quantity issued
Quantity (updated by the system)—item quantity left

A.2. Operations

1 Stock-transfer—The inventory movement from one stock lot to another stock lot
2 Goods-received—The entry of an item into the warehouse if and only if the total quantity of the item is less than its maximum stock level.
3 Goods-issued—the passing out of materials
4 Goods-movement—A report that shows today's items in and out.
5 Replenishment—A report that shows items below the minimum stock level.

A.3. Warehouse in/out logic

1 Item issued is based on the minimum value of the sum of production date and batch date. An example is shown in Table A1
2 The average unit cost for a warehouse item should be calculated whenever there is goods a "Receipt" transaction. Note that the average unit cost remains unchanged for other operations such as stock transfer and goods issue:

$$\text{New_average_unit_cost} = \frac{\text{average_unit_cost} \times \text{total_quantity} + \text{price} \times \text{quantity_of_goods_receipt}}{\text{total_quantity} + \text{quantity_of_goods_receipt}}.$$

3 If the total quantity of a warehouse item is less than the minimum stock level, alert the shortage of quantity in a report.
4 If the total quantity of a warehouse item is more than the maximum stock level, reject any goods receipt for this item.

Appendix B

Twenty-four subjects forming eight groups of "similar-capability" pairs and singles wrote the same FIFO program fourth times. The group is listed in the order in Fig. 2 (Table A2):

Table A1

Production Date (YYYYMMDD)	Batch Date (YYYYMMDD)	Sum	Order of goods issued
20020103	20030120	40050223	2
20020204	20030120	40050324	3
N/A	20030210	20030210	1

Table A2

	1st round		2nd round		3rd round		4th round	
	Solo	Pair	Solo	Pair	Solo	Pair	Solo	Pair
Group A	580	440	490	430	470	360	420	410
Group B	1190	760	790	500	520	420	320	350
Group C	1060	610	590	510	630	440	450	415
Group D	560	340	310	280	250	240	220	260
Group E	450	360	330	240	190	230	240	190
Group F	520	310	240	230	220	180	190	160
Group G	490	260	240	190	170	190	230	180
Group H	240	200	260	180	220	190	210	230

References

Atwood, M.E., Ramsey, H.R., 1978. Cognitive structure in the comprehension and memory of computer programs: an investigation of computer program debugging. US Army Research Institute for the Behavioral and Social Sciences, Technical Report (TR-78-A21), Alexandra, VA.

Baheti, P., Gehringer, E., Stotts, D., 2002. Exploring the efficacy of distributed pair programming. In: Proceedings of Extreme Programming and Agile Methods—XP/Agile Universe, USA, August, pp. 208–220.

Beck, K., 2000. eXtreme Programming Explained: Embrace Change. Addison-Wesley, Boston, MA.

Constantine, L.L., 1995. Constantine on Peopleware. Yourdon Press, Englewood Cliffs, NJ.

Flor, N.V., Hutchins, E., 1992. Analyzing distributed cognition in software teams: a case study of team programming during adaptive software maintenance. In: Koenemann-Belliveau, J., Moher, T., Robertson, S. (Eds.), Proceedings of Empirical Studies of Programmers: Fourth Workshop. Ablex, Norwood, NJ.

Friedman, M., Voas, J., 1995. Software Assessment: Reliability, Safety, and Testability. Wiley, New York.

Gallis, H., Arisholm, E., Dybå, T., 2002. A transition from partner programming to pair programming—an industrial case study. In: Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, USA, available at http://www.simula.no/publication_one.php?publication_id=511.

Gallis, H., Arisholm, E., Dybå, T., 2003. An initial framework for research on pair programming. In: Proceedings of International Symposium on Empirical Software Engineering, Italy, pp. 132–142.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA.

Heiberg, S., Puus, U., Salumaa, P., Seeba, A., 2003. Pair programming effect on developers productivity. In: Proceedings of Extreme Programming and Agile Processes in Software Engineering, Italy, May, pp. 215–224.

Hutchins, E., 1995. Cognition in the Wild. MIT Press, Cambridge, MA.

Keefer, G., 2002. Extreme programming considered harmful for reliable software. In: Proceedings of the 6th Conference on Quality Engineering in Software Technology, Germany, December, pp. 129–141.

Mayer, D.B., Stalnaker, A.W., 1968. Selection and evaluation of computer personnel—the research history of SIG/CPR. In: Proceedings of the 23rd ACM National Conference, pp. 657–670.

McDowell, C., Werner, L., Bullock, H., Fernald, J., 2003. The impact of pair programming on student performance, perception, and persistence. In: Proceedings of the 25th International Conference on Software Engineering, USA, May, pp. 602–607.

Mckeithen, K.B., Reitman, J.S., Reuter, H.H., Hirtle, S.C., 1981. Knowledge organization and skill differences in computer programmers. Cognitive Psychology 13, 307–325.

Müller, M.M., 2003. Are reviews an alternative to pair programming? In: Proceedings of Empirical Assessment In Software Engineering, UK, April, pp. 3–12.

Müller, M.M., Padberg, F., 2002. Extreme programming from an engineering economics viewpoint. In: Proceedings of the Fourth International Workshop on Economics-Driven Software Engineering Research, Orlando, Florida, May, pp. 57–60.

Müller, M.M., Tichy, W.F., 2001. Case study: extreme programming in a university environment. In: Proceedings of the 23rd International Conference on Software Engineering, Toronto, pp. 537–544.

Munzert, A., 1994. Part IV: computer I.Q.—program procedure, Test Your IQ (third edn). Random House, pp. 112–117.

Nawrocki, J., Wojciechowski, A., 2001. Experimental evaluation of pair programming. In: Proceedings of the 12th European Software Control and Metrics Conference, London, April, pp. 269–276.

Nawrocki, J., Jasiñski, M., Olek, L., Lange, B., 2005. Pair programming vs. side-by-side programming. In: Proceedings of EuroSPI, Budapest, November, pp. 28–38.

Nosek, J.T., 1998. The case for collaborative programming. Communications of the ACM 41 (3), 105–108.

Soloway, E., Ehrlich, K., 1984. Empirical studies of programming knowledge. IEEE Transactions on Software Engineering 10 (5), 595–609.

Stephens, M., Rosenberg, D., 2003. Extreme Programming Refactored: the Case against XP. Apress, Berkeley, CA.

Traynor, D., Gibson, P., 2004. Towards the development of a cognitive model of programming: a software engineering approach. In: Proceedings of the 16th Workshop of Psychology of Programming Interest Group, Ireland, available at www.cs.may.ie/~dtraynor/papers/PPIGarticle.pdf

Tuckman, B.W., 1965. Developmental sequences in small groups. Psychological Bulletin 63, 384–399.

Wang, Y., 2002. On cognitive informatics. In: Proceedings of the 1st IEEE International Conference on Cognitive Informatics, pp. 34–42.

Williams, L., 2000. The Collaborative Software Process. Ph.D. Dissertation, University of Utah.

Williams, L., Kessler, R., Cunningham, W., Jeffries, R., 2000. Strengthening the case for pair programming. IEEE Software 17 (4), 19–25.