Software Process Fusion: Uniting Pair Programming and Solo Programming Processes

Kim Man Lui and Keith C.C. Chan

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

Abstract. The role of pair programming process in software development is controversial. This controversy arises in part from their being presented as alternatives, yet it would be more helpful to see them as complementary software management tools. This paper describes the application of such a complementary model, software process fusion (SPF), in a real-world software management situation in China. Pair and solo programming are adopted at different stages of the process and according to the background of programmers, as appropriate. Unlike the usual practice of eXtreme Programming, in which all production code must written in pairs, all-the-time pair programming, the proposed model encourages programmers to design code patterns of their own in pairs and then to use these patterns to build sub-modules solo. The report finds that the longer team members work alone, the more code patterns they develop for reuse later in pairs.

1 Introduction

The success of a software development relies on not only a development paradigm but also people management. Programmer management remains more of an art form than an engineering principle. Pair programming (PP) is a form of teamwork in which two developers sit together and collaborate on a single computer [1]. One programmer, called the driver, controls the keyboard and implements the program while the other, the observer, continuously examines the work, identifying defects and thinking ahead. From this perspective, we may define a software process as being a pair process if a team is performed by pairs and as a solo process if it is performed by individual developers. It should be noted that pair programming includes not only programming but also design, system analysis, testing, and other typical programmer activities.

The benefits of pair programming processes may include job rotation/succession against personnel turnover, skills transfer for knowledge management, and, as a result of pairs being able to explore a larger number of alternatives than a single person [2], more creative thinking leading to better ways to solve problems. The processes of pair programming also raise issues of staff appraisal, economics, and productivity [3, 4, 5].

Many empirical studies [3, 4, 6, 7, 8] have shown higher that pair programming processes are more “productive” than solo programming process, particularly where they consider novice programmers. One report, however, recounts the development of highly negative attitudes towards management of professional programmers implementing eXtreme Programming in real situations when software managers enforced 100% pairing [9]. Similarly, in our experience in China in which we had inexperienced programmers write production code pairs we found little evidence that programmers were either motivated by the practice or that they were more productive.

In this paper we are interested in the use of a mixed pair programming-solo programming method and its potential effect on the effectiveness and efficiency of the management of developers and, in particular, inexperienced programmers in China. We introduce the concept of Software Process Fusion (SPF) and propose its application with a case study of pair programming process and solo programming process that accounts not only for productivity but also considers staff motivation. It should be noted that although many programmers may from time to time work in pairs either willingly or out of necessity, at the moment we cannot say that this practice is well-defined, especially in terms of its role in the context of general professional practice.

This paper is organized as follows. Section 2 reviews the empirical studies on productivity in pair programming. Section 3 introduces the concept of SPF and a management case study in which inexperienced programmers are set to both pair programming and solo programming activities. Section 4 reports an industrial case of the use of SPF. The final section offers our Conclusion.

2 Pair Programming Process

All-the-time pair programming requires that all production code be written by pairs of programmers. It is the core of extreme programming [1]. In the past, many agile software practices individually proposed or re-introduced have been similar but have been called by different names. However, none of the practices has been alike to pair programming and this has made extreme Programming and pair programming so undividable.

Many software processes are usually regarded as team processes. The basic unit that forms the basis of a software team is an individual programmer, shown in (a) (b).

Fig 1.a. According to the team composition, we can categorize them as solo programming process. Alternatively, when software processes are adopted by a team formed by pairs as illustrated in (a) (b).

Fig 1.b, they are pair programming process. Collaborative Software Process by Williams is such software process [4].

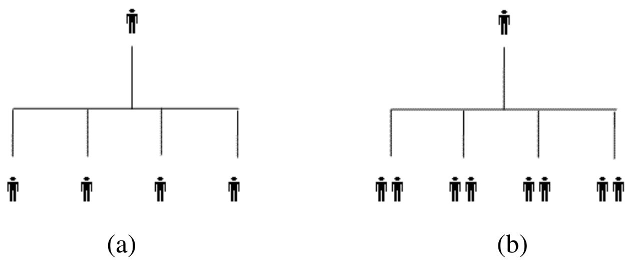

Fig. 1. Teams for solo programming process and pair programming process

This section reviews the literature on pair programming productivity. Two principles are reached in [3, 6, 7, 8]: The first principle is that a pair is much more productive and can work out a better solution than two individuals when the pair is new to design, algorithm, and coding of a program. The second principle is that pair programming can lose its productivity when a pair has prior experience of a task. Section 2.2 discusses an initial study in which pair programming is implemented as a way of assisting newly-hired programmers [10].

2.1 Control Experiments on Pair Programming

Two studies that favor pair programming have found that pair programming can speed up software development and at the same time produce better quality in terms of readability and maintainability than does solo programming. In 1998, Nosek [3] reported that full-time system programmers divided into five pairs and five singles were asked to write a UNIX script that performs a database consistency check (DBCC) in a Sybase database. On average, pairs took 42% longer than individuals on the same task; however, pair programming, in comparison with solo programming, reduced the elapsed time by 29% (i.e. $100\% - \frac{1+42\%}{2}$). In 2000, Williams [4] repeated the experiment in a similar setting using forty-one university students writing a challenging web-based program. The experiment showed that a pair took 15% longer than an individual on the same task; the elapsed time was reduced by 42,5% (i.e. $100\% - \frac{1+15\%}{2}$). In terms of productivity, the extra time was insignificant because pair programming achieves a higher quality.

In 2001, Nawrocki and Wojciechowski [11] reported experimental results unfavorable to pair programming, showing that pair programming consumed twice the time resources of solo programming. Subjects were asked to write four programs for (1) finding the mean and standard deviation of a sample of numerical data, (2) finding the linear regression parameters, (3) counting the number of lines in a program and (4) counting the total program LOC. The pairs took 100% longer than individuals. As the tasks of programming statistical calculations and counting the number of lines in a program were not new to the subjects, these experiments may indicate that pair programming is not as productive as solo programming when subjects are working on familiar tasks.

In 2004 Lui and Chan reported on a series of experiments called Repeat Programming in which pair and solo programmer subjects wrote the same program eight times [6，7] or four times [8]. The purpose of this was to simulate the process in which a novice programmer develops expert familiarity with a task and to measure the change of productivity of pair programming versus solo programming. At the first round, pairs spent 7.5% longer than individuals on the same task, 23% longer on the second round, 40% on the third, deteriorating to 134% on the eighth. These results indicate just how much the relative productivity of pair programming depends on previous programming experience on a particular task. The less experience a pair has, the better it performs relative to the two similarly inexperienced individuals. Lui and Chan [7, 8] conclude that a pair is much more productive and can work out a better solution than two individuals when the pair is new to design, algorithm, and coding of a program. This advantage is lost, however, as subjects gain experience of the task.

2.2 Pair Programming for Newly-Hired Developers

Regarded as a process of learning and practicing, pair programming has a considerable pedigree in the area of learning theory. Active learning has been adopted in colleges [12]. It involves three processes: think-pair-share. Research into the effectiveness of pair learning relative to group learning [13] has shown that the group learning could be more effective than pair learning. Yet pair learning is still widely used in teaching. Research on learning in pair includes English Vocabulary [14], Physics [15], Mathematics [16], and recently, Computer Programming [17, 18]. When student programmers are compared to novice programmers, the success of pair learning formulates our research problem that pair programming can mentor less experienced programmers in industrial software development.

In 2003, a research student, Poff, conducted an empirical study was conducted in which two novice programmers in the company, TCMS, were selected to produce portions of an application for the verification of payload hardware prior to integration into the space shuttle at TCMS in the Kennedy Space Center [10]. The experiment lasted one month and the data collected was compared with historical data at TCMS. Two programmers were told that the experiment was of secondary priority; most important was successful and timely development of the application. The two programmers were then left to decide how often they would actually work together but were asked to work as a pair at least 33% of the time, but if they wished could work as a pair 100% of the time. The result was that the pair worked as a pair 50% of the time.

The author observed that a pair of novice-novice programmers could develop technical and environmental knowledge more quickly. Although the author did not mention a potential application of a mix of pair programming and solo programming, the case illustrates that pair programming and solo programming have been optimized in a reciprocal manner by a pair of newly-hired programmers.

Although Poff did not report the workspace layout, it should safely assume that those two programmers have their own machine so that they can do solo programming. If the two programmers are actually sitting closely and each other machine, they are doing is side-by-side programming proposed by Cockburn [19] in which the developers choose to work in pairs or solo on an ad hoc basis. In fact, two programmers may not have to sit side-by-side as suggested. As far as they are closed enough and can easily see both screens of each other which can be called "pseudo side-by-side programming", it probably achieve the same effect. In some cases, when an effective working rapport has been built between two programmers, they may sit a little far away or opposite to each other as far as they can easy talk and hear each other in a collocated place. It is the people collaboration that makes them productive and a workspace layout is just as a tool that facilities the collaboration.

In the Poff's experiment, whether the pair was practicing side-by-side, pseudo side-by-side programming or talk-and-hear programming, they cannot be considered as a disciplined software practice because there are no clear guidelines when they should pair up and split off. Section 3 will introduce Software Process Fusion (SPF). As an example of SPF, we will describe a combination of pair programming and solo programming.

3 Software Process Fusion (SPF)

The idea of software process fusion was brought from data fusion which is the process of combining two (or more) independent data sets in order to produce information to the user. As two data sets are independent, the challenging is to combine them by formulating common variables in mathematics [20]. In data fusion, one is a recipient set and the other is a donor set. The use of defined common variables allows the recipient set to be enriched with extra information from the donor set. For example in retail, a recipient set can be sales data and a donor set is a marketing survey.

In a similar fashion, A and B are two independent processes that can produce the same work products. One of the processes, say A, is a recipient process and the other, say B, is a donor process. It is possible to use the mechanisms of A and B to define a set of transfer conditions so that, over time, the recipient process can temporarily convert into the donor process for productivity and resource optimization.

In Software Process Fusion, we should start with and end in a recipient process. Although it may appear that two processes alternately change and become an alternating process (see Fig 2), the recipient process and the donor process cannot be mixed up because the transfer conditions are bound to this relationship. A fused process is a recipient process being merged with a donor process. We can draw an analogy between common variables in data fusion that bring two data sets together and transfer conditions in Software Process Fusion (SPF).

In Section 2.2, the real case reported by Poff would be Software Process Fusion as long as transfer conditions could be clearly established. Without those conditions, the alternating process in pair-and-solo programming appears uncontrolled and chaotic. The core of data fusion is to mathematically define common variables between data sets; the challenge in Software Process Fusion is to clearly establish a set of transfer conditions.

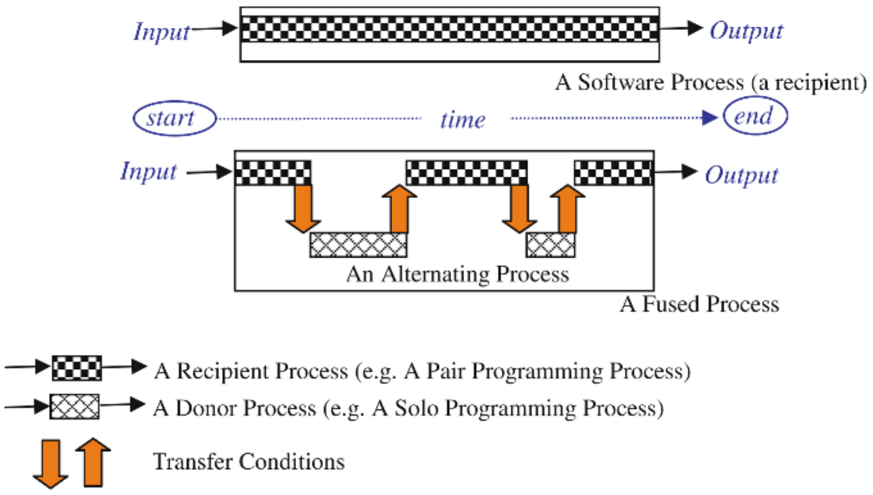
Input
Output
A Software Process (a recipient)
start
time
end
Input
Output
An Alternating Process
A Fused Process
A Recipient Process (e.g. A Pair Programming Process)
A Donor Process (e.g. A Solo Programming Process)
Transfer Conditions

Fig. 2. Software Process Fusion

Fig. 2 shows an example of Software Process Fusion. Both a recipient process and a donor process can take and deliver the same input and output. Note that the recipient process is pair programming. In Software Process Fusion, we always start with and end at a recipient process. When the transfer conditions are satisfied during the pair programming process, it stops and initiates a donor process, i.e. the solo programming process. The donor process will not indefinitely take over the control and it will return to the recipient process if the transfer conditions for donors are met. Obviously, defining suitable transfer conditions is the key to the success of Software Process Fusion.

3.1 Transfer Conditions

In Section 2.1, previous studies have shown that it is productive for a pair of programmers to design algorithms, seek design patterns, and code. The productivity of pair programming will fall when the pair works on other modules in which the logic has been similar to the previous modules previously done as a pair [6, 7, 8]. In this case, to optimize resources, the pair should split off and the two individuals should complete those modules solo. Once they have finished, they should pair up again and review the overall task. This process is iterative until they complete their assignment.

Therefore, we define the transfer condition for a recipient process in Fig. 2 to convert into a donor process (i.e. solo programming process) is that a pair of developers has previously completed a similar task and they are individually able to solve the same problem again in the same way. Straightforwardly, the transfer condition for a donor process is for the individuals to pair up again after completing their solo programming tasks.

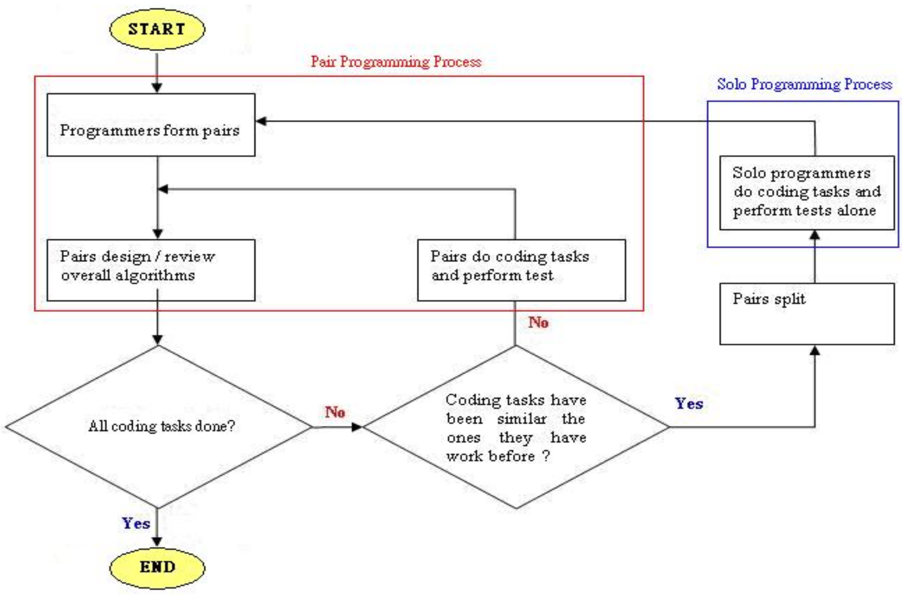
START
Pair Programming Process
Solo Programming Process
Programmers form pairs
Solo programmers do coding tasks and perform tests alone
Pairs design / review overall algorithms
Pairs do coding tasks and perform test
No
Pairs split
All coding tasks done?
No
Coding tasks have been similar the ones they have work before ?
Yes
Yes
END

Fig. 3. Application of SPF for Managing Inexperienced Programmers

The transfer conditions can be regarded as a number of intermediate targets. We would like to reach such intermediate targets as many times as possible and hence the transfer conditions drive that fused process to (1) look for design patterns and implement them once in pairs, (2) reuse the same patterns in solo programming and (3) review overall progress and perform integration testing in pairs.

3.2 A Fused Process

A fused process not only is the sum of a recipient process and a donor process, but also includes necessary changes that come along with the transfer condition. This section will present a fused process: combining pair programming with solo programming processes. Fig. 3 illustrates the internal workflow of the fused process. It starts with a recipient process (i.e. a pair programming process) in which pairs work on design and algorithm and identify patterns of logic. Afterward, they code and test sub-programs in pair programming. The pairs then split up and code and test as solo programmers. Once those tasks are complete, they once again pair up, review their work, and perform integration tests.

4 Industrial Case Study

We introduced the work to Huida Technology Ltd in Huizhou, China in 2005. The company had seven technical staff, providing ERP/CRM solutions to their local customers. Two had four years experience and the other five all less than one year. The company would adopt what we proposed in this paper to work on their CRM project for one month. Of the five junior programmers, four paired up and the other worked as usual. Priority was given to the success of the inventory project, so that the company was free to terminate the proposed fusion process at any time. The results, shown in Table 1, were recorded and provided by two supervisory programmers.

The project was not an experimental test. It is a matter of happenstance that there were five staff available for the experiment, so the use of a single programmer was not intended to serve as a control group. Table 1 nonetheless provides for reference a comparison of the pair and solo teams. The programmers appeared to develop more (sub-) modules in terms of stored procedures and GUI. We are particularly interested

Table 1. Experimental Test in 2005

Item	Measurement Description	Huida Programmers		
		Pair	Pair	Single
1	Number of GUI Developed	7	6	2
2	Number of Stored Procedures Written	15	9	5
3.a	Programming Time (%)	49%	41%	40%
3.b	Testing Time (%)	26%	18%	20%
3.c	Debugging Time (%)	25%	31%	40%
4	Fusion Ratio	1.7	1.9	N/A
4.a	Pair Time (%)	37%	34%	N/A
4.b	Solo Time (%)	63%	66%	100%

in a fusion ratio, defined by the total time required for donor processes over the total time required for a recipient process. The fusion ratio was higher than Poff's measurement, which was around 1.0 [10].

The supervisors worked with those five programmers daily and knew them well. Their comments on the process are of interest.

1. They found that they were able to spend less time supervising the pairs as they tended to support and monitor themselves.
2. Coding standards were much better.
3. The fused process encouraged junior programmers to actively seek design patterns for reuse. This has rarely been seen before as the programmers just wanted to complete the program on time, rather than considering software reuse. Hitherto, it was common to see duplications of logic in the junior programmer's code as they had the habit of simply cutting and pasting code.

It has been reported that pair programming comes with pair pressure that a pair does not want to let its partners down and that this leads to pairs budgeting their time more wisely [4]. The supervisors failed to observe any signs of pair pressure; however, it was clear that programmers were glad to move on to solo programming as it demonstrated that they managed to discover reusable patterns of their own. The team had been motivated and influenced by its achievements of pattern discovery. In addition, the time that they split off demonstrated their supervisors that they were making progress.

5 Conclusions

The paper contributes to our understanding of software process fusion. Software methods/processes need not be defined as being in opposition or competition. Rather, they can be seen as complementary. We also presented a case study of the application of SPF in the management of inexperienced programmers in a real industrial project in China. The initial results and comments show that SPF is a promising software management approach.

Researchers on pair programming are divided. Some believe it is more efficient and effective than solo programming whereas others argue it doubles the resources that are consumed in software development. Perhaps, the truth lies between these two views. Software Process Fusion encourages not programming in pairs but working out coding patterns in pairs. Developers can pair up and split off. The proposed fused process has successfully been implemented in a small company in China.

References

1. Beck, K.: Extreme Programming Explained: Embraced Change (2nd Edition), Addison-Wesley, Boston, MA (2005)
2. Flor, N. and Hutchins, E.: Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Maintenance, In J. Koenemann-Belliveau, T. Moher and S. Robertson (Eds.), Empirical Studies of Programmers: Fourth Workshop, Norwood, NJ: Ablex (1991)

3. Nosek, J.T.: The Case for Collaborative Programming, Communications of the ACM, March (1998) 105-108
4. Williams, L.: The Collaborative Software Process, Ph.D. Dissertation, University of Utah, (2000)
5. Miller, M.M. and Padberg, F.: Extreme Programming from an Engineering Economics Viewpoint, In Proceedings of the Fourth International Workshop on Economics-Driven Software Engineering Research (2002)
6. Lui, K.M. and Chan, K.C.C.: When Does a Pair Outperform Two Individuals. In Proceedings of Extreme Programming and Agile Processes in Software Engineering, Italy (2003) 215-224
7. Lui, K.M. and Chan, K.C.C.: A Cognitive Model for Solo Programming and Pair Programming. In Proceedings of the Third IEEE International Conference on Cognitive Informatics, Canada (2004) 94-102
8. Lui, K.M. and Chan, K.C.C.: Productivity of Pair Programming: Novice-Novice and Expert-Expert, Tentatively Accepted by International Journal of Human Computer Studies (2006)
9. Stephens, M. and Rosenberg, D.: Extreme Programming Refactored: The Case Against XP, Apress (2003)
10. 10 Poff, M. A.: Pair Programming to Facilitate the Training of Newly-Hired Programmers, M.Sc. Thesis, Florida Institute of Technology (2003), Available online at http://www.cs.fit.edu/~tr/tr2003.html
11. Nawrocki, J. and Wojciechowski, A.: Experimental Evaluation of Pair Programming", Proceedings of the 12th European Software Control and Metrics Conference, England (2001) 269-276
12. Bonwell, C.C. and Eison, J. A.: Active Learning: Creating Excitement in the Classroom. ASHE-ERIC Higher Education Report. Washington, D.C. (1991)
13. Roth,V., Goldstein,E. and Marcus,G. : Peer Lead Team Learning A Handbook for Team Leaders. Upper Saddle River, NJ: Prentice- Hall, Inc. (2001)
14. Jones, M.S., Levin, M. E., Levin, J. R. and Beitzel, B. D.: Can Vocabulary-Learning Strategies and Pair-Learning Formats Be Profitably Combined? Journal of Educational Psychology, Vol. 92, No. 2 (2000) 256-262
15. Warnakulasooriya, R. and Pritchard, D.: Learning and Problem-Solving Transfer between Physics Problems using Web-based Homework Tutor. In Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications, Chesapeake, VA (2005) 2976-2983
16. Keeler, C.M. and Steinhorst, R.K.: Using Small Groups to Promote Active Learning in the Introductory Statistics Course, Journal of Statistical Education, [Online journal] (1995) available at
http://www.amstat.org/publications/jse/v3n2/keeler.html
17. McDowell, C., Hanks, B. and Werner, L.: Experimenting with Pair Programming in the Classroom, In Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education, Thessaloniki, Greece (2003)
18. McDowell, C., Werner, L., Bullock, H. and Fernald, J.: The Impact of Pair Programming on Student Performance, Perception, and Persistence, In Proceedings of the 25th International Conference on Software Engineering (2003) 602 – 607
19. Cockburn, A.: Crystal Clear**:** a human-powered methodology for small teams, Boston: Addison-Wesley (2005)
20. van der Putten, P., Kok, J.N. and Gupta, A.: Why the Information Explosion Can Be Bad for Data Mining, and How Data Fusion Provides a Way Out. In Proceedings of Proceedings of the Second SIAM International Conference on Data Mining (2002)