2009 16th Asia-Pacific Software Engineering Conference

Investigating the Effect of Pair Programming and Software Size on Software Quality and Programmer Productivity

Raymund Sison
College of Computer Studies
De La Salle University
Manila, Philippines
e-mail: raymund.sison@delasalle.ph

***Abstract*— Although pair programming has been studied since the late 1990s, it is only recently that the results of earlier studies are being fine-tuned. For example, the results of a 2007 study conducted in Europe suggests that although pair programming might not always increase software quality, it can do so when the software being built is relatively complex and junior developers are on the team. This paper describes two experiments involving the development of small systems (10,000 to less than 100,000 lines of code) and very small programs (less than 10,000 lines of code) by student programmers in an Asian university. The results involving the small systems, which were actually complete systems designed to support the Personal Software Process (PSP), showed that defect densities of systems written by the pair programming teams were significantly lower than those written by the teams that used the traditional approach of individual coding and testing of units, followed by integration testing. On the other hand, results involving the very small (and therefore much less complex) programs did not show any significant differences between the defect densities of the programs written by the pair programmers and of those written by the solo programmers, though they did show significantly greater productivity of the solo programmers when writing simpler code. The combined results of the two experiments suggest that pair programming might increase software quality without decreasing productivity when projects are sufficiently large (or complex) for the programmers working on them.**

Keywords-pair programming, personal software process

I. INTRODUCTION

Pair programming is a collaborative approach to software development that Williams and Kessler [1] define as follows:

"Pair programming is a style of programming in which two programmers work side by side at one computer, continually collaborating on the same design, algorithm, code, or test. One of the pair, called the driver, is typing at the computer or writing down a design. The other partner, called the navigator, has many jobs, one of which is to observe the work of the driver, looking for tactical and strategic defects. Tactical defects are syntax errors, typos, calling the wrong method, and so on. Strategic defects occur when the driver is headed down the wrong path—what is implemented just won't accomplish what needs to be accomplished. The navigator is the strategic, long-range thinker."

Several experiments and case studies conducted in the late 1990s and early 2000s (e.g., [2], [3], [4]) have suggested a number of benefits of pair programming, including higher software quality, greater developer productivity, greater developer satisfaction and confidence, enhanced learning (particularly on the part of the novice developer), and reduced staffing risk (on the part of the software organization). There were, to be sure, dissenting opinions; in the experiment of [5], for example, pair programming did not lead to higher programmer productivity compared to the use of the Personal Software Process (PSP) [6] in the coding of the first four programming exercises of the PSP.

In a recent systematic review [7] of 15 pair programming studies conducted in Europe and North America, all the 15 studies that were considered showed pair programming as positively affecting quality, although the effect sizes were mostly small to medium. Meta-analysis also showed a small positive effect of pair programming on duration, but a medium negative effect on effort.

More recently, experimental results of [8] had the effect of fine-tuning all these earlier statements by showing that software complexity and programmer expertise can moderate the effects of pair programming. Thus, although pair programming might not always increase quality or productivity, their results suggest that pair programming can increase software quality when the software being maintained is relatively complex and junior developers are on the team, and can increase developer productivity when the software being maintained is relatively simple and senior developers are on the team. The authors of the said paper explained the quality improvement as possibly being due to social laboring [9 p.16], in which a group as a whole demonstrates greater productivity than the potential productivity calculated from individual performance baselines. In turn, social laboring, the authors say, is impacted by the complexity of the project and also by programmer expertise, so that junior programmers' finding a project complex can increase social laboring, whereas senior programmers' finding a project too simple can diminish social laboring.

1530-1362/09 $26.00 © 2009 IEEE
DOI 10.1109/APSEC.2009.71

In [8], software complexity was a function of control design. More specifically, software with a delegated control design was viewed as being more complex than software with a centralized control design. The actual tasks that were performed by the programmers in the said study were change tasks, which, in the words of the authors, were small (possibly even very small), typically only involving the addition of a button or choice. Thus, the authors suggested that future experiments include larger systems and more complex tasks.

In this study, we look more closely at how software *size*, as a general indicator of software complexity, might moderate the effect of pair programming on defect density (as a general indicator of software quality) and productivity. There are, of course, other indicators of software quality, but defect density is a basic measure, with which other quality measures such as defect ratios, defect removal leverage, yield (of defect removal phases), and others may be coupled [6].

We conducted two experiments with a total of 48 student programmers who were third-year computer science majors at an Asian university. In the first experiment, we compared 16 pair programmers against 16 solo programmers with respect to defect density (number of defects per thousand lines of code) and productivity (lines of code per person-hour) in their development of the five programming requirements of PSP levels 0, 0.1, 1.0, and 1.1. These programs were very small, having less than 200 lines of code (LOC).

In the second experiment, we compared 7 pair programming teams against 7 "traditional" programming teams with respect again to defect density and productivity in their development of a system to support PSP levels 0 through 1.1. By "traditional" we mean that the individuals in a programming team did their coding and unit testing individually, after with they performed integration and system testing, as opposed to a pair programming team, in which the individuals did the coding and testing collaboratively as described in the quote above from [1]. In contrast to the programs in the first experiment, the systems in the second experiment were much larger (though, at around 10,000 LOC, these systems would still be small following the classification of [10]) and more complex. Through these experiments we would like to see whether there would be any significant differences in the mean software defect density and productivity of pair and non-pair student programmers when developing applications of different sizes (very small and small).

We had earlier conducted a study [11] similar to the second experiment. In that study, there were 6 pair and 6 traditional programming teams that developed a Unified Modeling Language (UML) Tutor. The results of that study showed a significantly lower mean defect density for the pair programming teams compared to the traditional programming teams. However, there were no significant differences in the productivity levels of the experimental and control groups. We therefore also wanted to see whether we could replicate the experiment of [11] and obtain the same results in this study's second experiment.

The rest of this paper describes the design of the experiments and their execution, and then presents and analyzes the results of the experiments, and finally provides concluding remarks and suggestions for further work.

II. Research Design

The subjects of the experiments were 48 third-year computer science undergraduate students who were using pair programming for the first time as part of the requirements of a one-trimester Advanced Software Engineering (ADVANSE) course taught by the author from January to April 2009 at the De La Salle University in Manila. The said course enables students to analyze and in some cases experience the various software methods on Boehm and Turner's [12] spectrum of plan-driven and agile methods, including: the Personal Software Process (PSP), the Team Software Process (TSP), the Capability Maturity Model Integration (CMMI), the Unified Process (UP), eXtreme Programming (XP), and Scrum. The said course also introduces students to important trends in software engineering, such as pattern-based, aspect-oriented, and global software development. The primary prerequisite to ADVANSE is Introduction to Software Engineering (INTROSE), in which students should have learned to perform requirements elicitation, software design, and testing (unit, integration, and system testing). The prerequisite to INTROSE is a sequence of 3 programming courses in C and Java.

On the second session of this ADVANSE course, the students were asked to form, by themselves, groups having 4 members each. The ADVANSE students were in the final months of their year-long capstone project, so they grouped themselves according to their capstone project groupings. Two of the four members of a group would work collaboratively as pair programmers, while the other two would work solo in the first experiment, and as traditional (non-pair) programmers working as a team in the second experiment.

The ADVANSE course had two programming requirements. The first was to do, individually and in pairs, the first five (of the 10) carefully designed programming exercises of the Personal Software Process (PSP) [6], a software development and process improvement method that is on the plan-driven end of Boehm and Turner's [12] agility spectrum. Being on the said end of the spectrum, the PSP is very plan-oriented and documentation-intensive. Through this programming requirement, the subjects were trained to log every defect in every phase of software development (i.e., in analysis, design, coding, and test). In the PSP a defect is something that is wrong with the program, and this could be a misspelling, a punctuation mistake, or an incorrect program statement [6]. The students were also trained, through the PSP, to log the time –

productive as well as interruption time – that they spent in every phase. From the data in their defect and time logs, they would later compute their defect densities (as the number of defects detected during code, compile, and test per thousand LOC) and productivity (as LOC per person-hour), where LOC refers to "new and changed LOC", following [6]. This means that lines of code that were not written by the students were not counted when computing their programs' LOC.

The first five programming exercises of the PSP, which the subjects had to do, are [6]:

1A: Write a program to estimate the mean and standard deviation of a sample of n real numbers.

2A: Write a program to count the number of logical lines of a program, omitting comments and blank lines.

3A: Write a program to count the total LOC, the total LOC in each class the program contains, and the number of methods in each class.

4A: Write a program to calculate the linear regression estimating parameters for a set of n programs where the historical LOC and new and changed LOC are available.

5A: Write a program to numerically integrate a function using Simpson's rule and write the function for the normal distribution.

Note that when doing Program 2A, students learn not to count comments and blank lines when computing lines of code.

The second programming requirement of the ADVANSE course was to develop, in teams and in two months, a PSP tool or, more specifically, a system to support the recording of PSP data and the preparation of the various PSP forms and reports for PSP levels 0 through 1.1.

Levels 0 and 0.1 of the PSP focus on the so-called Baseline Personal Process, and it is at these levels that a programmer learns to pay attention to the time that he or she spends in each phase of software development, and the defects that he injects and removes per phase. It is also here (specifically at level 0.1) that the programmer also learns to pay attention to the size of his or her programs (in terms of lines of code or LOC). The number of defects and the development time will later be combined with LOC to form the defect density and productivity metrics, which will then later be used to evaluate the quality and effectiveness of one's personal software process. The programmer also learns at these levels (specifically at level 0.1) to use these time, defect, and size data when examining his or her process for possible improvement.

Levels 1.0 and 1.1 of the PSP focus on Personal Process Management, and includes prescriptions for software size estimation based on linear regression estimates, which, in turn, are based on the historical time and size data that the programmer has learned to collect, and processes for task and schedule planning (and tracking) using the Earned Value approach.

A typical graphical user interface for a PSP system is shown in Figure 1. On the left are tabs for the various PSP forms, reports, and logs that the system should produce for PSP levels 0 through 1.1, namely:

- Project Plan Summary
- Test Report Template
- Process Improvement Proposal

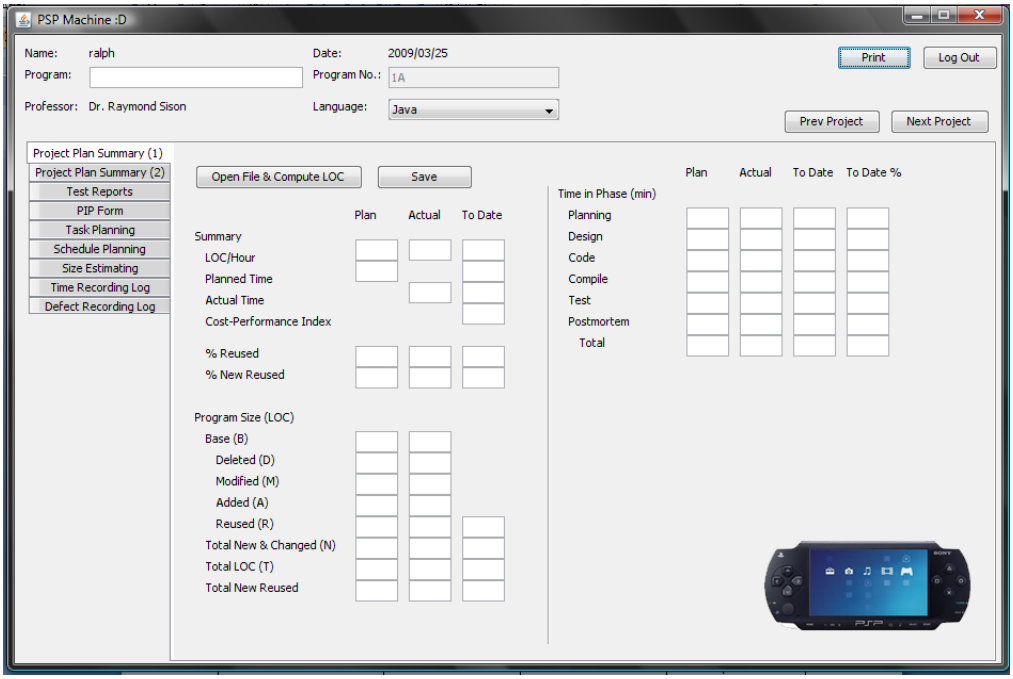

Figure 1. Typical Graphical User Interface of a PSP Support Tool

- Task Planning Template
- Schedule Planning Template
- Size Estimating Template
- Time Recording Log
- Defect Recording Log

The figure shows part of a Project Plan Summary, which summarizes the data found in the time and defect recording logs, among others. As can be seen in the figure, the Project Plan Summary shows the Total New and Changed LOC. As explained earlier, the Total New and Changed LOC, which does not include lines of code not written by the programmer (e.g., GUI code written by GUI builders such as those of Visual Studio), was what the students used when computing defect densities and productivity.

As mentioned in the introduction, two experiments were designed. Experiment 1 was designed to determine whether pair programming would affect defect density and productivity of the student programmers in their writing of the programs required by PSP 1A-5A, which, at 100-200 LOC, would be considered "very small". There were 16 pair and 16 solo programmers. The average cumulative grade point averages (CGPAs) of the pairs and solo programmers up to the trimester prior to their taking ADVANSE were 2.51 and 2.59, respectively, with standard deviations of 0.50 and 0.47, respectively. There were actually more than 48 students in the author's ADVANSE classes during that trimester, but only the 48 under consideration were able to submit complete reports and programs for all five PSP exercises.

Experiment 2 was designed to determine whether pair programming would affect defect density and productivity of the students in their writing of relatively larger and complete systems for supporting PSP levels 0 through 1.1. At 10,000-12,000 LOC, these systems would be considered "small", following [10]. Each group first agreed upon a design for their PSP support system, and then divided itself into two subgroups or pairs, such that the difference in the average CGPAs of the two subgroups was minimized. One pair developed the PSP system using pair programming; the other pair developed the same system using a more "traditional" approach, i.e., with the members coding and unit-testing individually and then integrating their code and performing integration and system testing in the end. Each group was then asked to give its opinion, as a whole group, on whether pair programming, in their experience, was beneficial, and then present quantitative data (in the form of defect densities and LOC/person-hour) and qualitative data that supported the group's opinion.

There were 7 groups that were able to turn in products with complete basic functionality (i.e., all the basic functions of a tool for PSP 0.0 through 1.1 were present and working correctly) and reports with complete and detailed data (in the form of detailed time and defect logs), as verified by the author. These seven groups yielded 7 pair programming teams and 7 traditional programming teams.

The design of Experiment 1 is similar to that of [5], which also compares pair programming against solo programming using the PSP. There are significant differences, however. First, in Experiment 1, both the pair and solo programmers used the PSP, whereas in [5], the pair programmers did not. This design of [5] was because its authors were primarily interested in comparing pair programming against PSP programming, whereas we were primarily interested in comparing pair programming against solo programming, with the PSP context being secondary. Second, our measure of quality was the defect density, while [5] used the number of resubmissions of corrected programs. Third, [5] coupled pair programming with other XP practices notably test-driven development, whereas we focused purely on pair programming.

The design of Experiment 2 differs from all the 17 studies reviewed in [13] in one major respect. All the studies reviewed in the said paper that had experimental and control groups compared the output of pairs against those of individuals. In contrast, we compare in Experiment 2 the output of pairs that used pair programming against those of pairs that used the traditional programming approach, in which members would perform coding and unit testing individually and then integrate their code in the end.

The study reviewed by [13] whose design ours in Experiment 2 would be closest to would be that of [14] in the sense that each experimental-control pair worked on the same programming project. However, those in [14] worked on simple assignments similar in complexity to PSP 1A-5A, whereas the groups in our second experiment worked on a relatively more complex project. Moreover, the members of the control group in [14] actually worked individually, i.e., each individual wrote the complete code for an assignment, whereas the members of the control group in our study worked in pairs, albeit traditionally, i.e., with the members coding individually and then integrating their code in the end. Knowing that "two heads are (generally) better than one," we decided in our study not to compare the program written by a pair versus the program written by a solo student, but the programs written by two pairs: one doing pair programming, the other doing traditional programming.

Another difference between [14] and the second experiment reported in this study is that the measure of program quality used in [14] was the number of test cases passed, while here it is the total number of defects found in code, compile, and test of systems that were eventually determined to have complete basic functionality.

In this study we investigated the following null hypotheses:

$H0_1$: There is no difference in the defect densities of very small (<10,000 LOC) programs developed by student pair programmers and solo programmers.

$H0_2$: There is no difference in the productivity (LOC/person-hour) of student pair programmers

TABLE I. MEAN, STANDARD DEVIATION (SD), AND P-VALUE OF DEFECT DENSITIES OF PAIR PROGRAMMERS (PP) AND SOLO PROGRAMMERS (SP) DEVELOPING VERY SMALL PROGRAMS

	Mean Defect Density (SD)		Significant?
	PP	SP	
PSP 1A	.048 (.037)	.080 (.049)	No (p=0.121)
PSP 2A	.059 (.030)	.077 (.054)	No (p=0.215)
PSP 3A	.032 (.019)	.057 (.082)	No (p=0.393)
PSP 4A	.030 (.022)	.039 (.039)	No (p=0.589)
PSP 5A	.036 (.039)	.051 (.073)	No (p=0.820)

TABLE II. MEAN, STANDARD DEVIATION (SD), AND P-VALUE OF PRODUCTIVITY (LOC/PERSON-HOUR) OF PAIR PROGRAMMERS (PP) AND SOLO PROGRAMMERS (SP) DEVELOPING VERY SMALL PROGRAMS

	Mean Productivity (SD)		Significant?
	PP	SP	
PSP 1A	42.75 (28.27)	81.25 (62.47)	Yes (p=0.039)
PSP 2A	25.88 (24.38)	51.38 (35.14)	Yes (p=0.042)
PSP 3A	45.38 (50.41)	45.88 (30.79)	No (p=0.352)
PSP 4A	26.44 (20.94)	55.13 (35.95)	Yes (p=0.004)
PSP 5A	25.14 (21.90)	48.06 (41.47)	No (p=0.052)

and solo programmers when developing very small programs.

$H0_3$: There is no difference in the defect densities of small (10,000-100,000 LOC) systems developed by student pair programming and traditional programming teams.

$H0_4$: There is no difference in the productivity (LOC/person-hour) of student pair programming and traditional programming teams when developing small systems.

All statistical analyses were carried out using the Wilcoxon Rank Test because the sample sizes were small and we did not want to make the assumption of a normally distributed population. Nevertheless, similar results (in terms of which null hypotheses were rejected and which not rejected) were obtained when the parametric Student T-test was used. The significance level for rejecting the hypotheses was set at 0.05.

III. RESULTS AND DISCUSSION

As shown in Table I, the mean defect densities of the programs written by the pair programmers were consistently lower than those of the solo programmers. However, the differences were not significant. The non-significant differences in mean defect densities of the experimental and control groups could be due to the very small sizes of the programs that they wrote: In the same way that senior programmers might find small software projects too simple for the need to work collaboratively, junior programmers or, more specifically, third-year student programmers might find very small programs too simple that no social laboring takes place.

Because the p-values of the mean differences were greater than the chosen significance level (0.05), we therefore accept the hypothesis ($H0_1$) that there is generally no difference between the defect densities of the products of the two groups when developing very small programs, specifically, programs whose lines of code are between 100 and 200.

Table II shows mixed results with respect to hypothesis $H0_2$: it seems that when writing PSP exercises 1A, 2A, and 4A, the mean productivity (LOC/person-hour) of solo programmers was significantly higher than (almost double) that of pair programmers, but there were no significant differences in the productivity levels of pair and solo programmers as they worked on PSP exercises 3A and 5A. This could be due to the fact that 1A, 2A, and 4A were simpler to understand and required smaller code than 3A and 5A. (PSP exercise 4A might appear more difficult for some readers to code relative to 3A or even 2A. However, the students had ready access to [6], which already provided the algorithm and formula for computing the α and β linear regression parameters.)

Going now to the larger and considerably more complex PSP system, Table III shows that the mean defect density of the systems written by the pair programmers was 42% lower

TABLE III. MEAN, STANDARD DEVIATION, AND P-VALUE OF DEFECT DENSITIES OF PAIR AND TRADITIONAL PROGRAMMING TEAMS DEVELOPING SMALL SYSTEMS

	Mean (SD)	Significant?
Pair Programming	.0042 (.0031)	Yes (p=0.018)
Traditional Programming	.0073 (.0055)	

TABLE IV. MEAN, STANDARD DEVIATION, AND P-VALUE OF PRODUCTIVITY (LOC/PERSON-HOUR) OF PAIR AND TRADITIONAL PROGRAMMING TEAMS DEVELOPING SMALL SYSTEMS

	Mean (SD)	Significant?
Pair Programming	75.76 (22.82)	No (p=0.310)
Traditional Programming	65.21 (25.25)	

than those of the pairs that performed traditional programming. The table also shows that the p-value for this median difference is 0.018. Because this value is less than the chosen significance level (0.05), we can therefore reject the hypothesis ($H0_3$) that there is no difference between the defect densities of the products of the two groups, and accept the alternative hypothesis that the defect densities of systems developed by pair programming groups are significantly different from (i.e., significantly lower than) those of traditional programming groups.

There was, however, no significant difference between the LOC/person-hour spent by the pair programming group compared to the traditional programming group in their development of small systems (hypothesis $H0_4$), as shown in Table IV.

All seven groups in the second experiment explained the quality result as being due to two heads being better than one, and the navigators' performing static testing while the drivers were writing code. Because the products of the pair programming teams had invariably lower defect densities than those produced by the traditional programming teams, the systems written by the pair programming teams were the ones that were chosen unanimously by all the groups as the superior systems.

The results of Experiment 2 tend to support those of previous studies, including the seminal studies of [2], [3], and [4], and those reviewed by [13]. These results are also consistent with what XP's inventor, Kent Beck, states in [15]: "Even if you weren't more productive, you would still want to pair, because the resulting code quality is so much higher." They also confirm the results of [11], conducted in 2006, whose experimental design is identical to that of Experiment 2 except that the subjects in [11] developed a UML tutor, whose level of complexity would be higher than that of a PSP system.

The results of both experiments also confirm those of [8], the most recent pair programming study, in the sense that quality differences were observed, and no productivity differences were found, in systems that were (relatively) complex and where junior developers were in the pair programming teams. Our notion of system complexity is not exactly the same as that of [8], however. We view a PSP system as complex because of the number and complexity of classes, data structures, and algorithms that they entail; in [8], however, complexity was viewed as the degree of decentralization of the architectural classes. In addition, [8] looked at effort (person-hours) whereas we looked at productivity (LOC/person-hour).

The nonrandom assignment of groups is a possible threat to validity. The assignment of pairs could not be completely randomized because of the need to follow the pair programming guideline of pairing students to maximize the chances that students would work well together [16]. Thus the students were allowed to work with their capstone project groups. Nevertheless, we ensured that within these groups and across all groups, the average CGPAs of those that practiced pair programming and those that did not were almost the same, as reported earlier. This can be viewed as mitigating the possible effects of nonrandom selection by trying to ensure the equivalence of the experimental and control groups at the beginning of the study.

The results of the study can be applied to computer science students in their last year of study and to junior programmers in software development organizations since the projects usually assigned to these groups are on the order of 10,000 to 100,000 lines of code.

IV. CONCLUDING REMARKS

Although pair programming has been studied since the late 1990s, it is only recently that the results of earlier studies are being fine-tuned. In this paper, two experiments were designed to determine the impact of pair programming and program size on defect density and productivity of student programmers. The programmers worked on small

systems (10,000 to less than 100,000 LOC) and very small (less than 10,000 LOC) programs. Results on the small systems, which were actually complete systems designed to support the Personal Software Process (PSP), showed that defect densities of systems written by pair programming teams were significantly lower than those written by teams that used the traditional approach (of individual coding of units followed by individual unit testing and then integration and system testing). This confirms results of an earlier and almost identical study [11] conducted by the author in the same Asian university.

On the other hand, results on the very small programs, which were actually the first five programming requirements of the PSP, did not show significant differences between the defect densities of programs written by pair programmers and of those written by solo programmers, although productivity of the pair programmers were significantly lower than solo programmers when writing the simplest programs.

The combined results of the two experiments tend to confirm the results of [8] that pair programming can increase software quality when the software being built is relatively complex (as indicated in our work by software size) and novice developers are on the project team. The combined results of the two experiments also indicate that pair programming can decrease programmer productivity when writing programs that are relatively simple.

To summarize, the results of the two experiments suggest that pair programming might increase software quality as measured by defect density without decreasing productivity when projects are sufficiently large or complex for the programmers working on them. In other words, pair programming might be used to increase software quality when the size or complexity of the software necessitates collaboration among programmers.

In the context of a software firm using XP for small projects, pair programming might therefore help boost software quality when many in the firm are junior programmers. On the other hand, when most programmers in an XP team are senior, pair programming might not yield the same results as in small projects, but might yield positive quality effects when the team's projects are at least medium in size (100,000 to less than 1,000,000 LOC), though further work is necessary to confirm this.

In the context of introductory programming courses in undergraduate programs in computer science, pair programming might positively impact code quality even when programming exercises are very tiny (less than 100 LOC) because these exercises might still be found to be complex by first-year students, especially, perhaps, by those who are "at risk" (i.e., students who have a high risk of failing), though this last conjecture about "at risk" students needs to be studied further. On the other hand, the use of pair programming for very small projects (such as the PSP exercises) by third or fourth-year computer science students might have no significant positive impact at all on software quality, and might even have negative impact on programmer productivity.

ACKNOWLEDGMENT

The author would like to thank the anonymous reviewers for their comments.

REFERENCES

[1] L. Williams and R. Kessler, Pair Programming Illuminated, Addison Wesley, 2002.
[2] J. Nosek, "The Case for Collaborative Programming," Comm. ACM, vol. 41, no. 3, pp. 105-108, 1998.
[3] A. Cockburn and L. Williams, "The Costs and Benefits of Pair Programming," Proc. XP 2000.
[4] K. Lui and K. Chan, "When Does a Pair Outperform Two Individuals?" Proc. XP 2003.
[5] J. Nawrocki and A. Wojciechowski, "Experimental Evaluation of Pair Programming," Proc. European Software Control and Metrics Conf. (ESCOM) 2001.
[6] W. Humphrey, A Discipline for Software Engineering. Addison Wesley, 1995.
[7] T. Dybå, E. Arisholm, D. Sjoberg, J. Hannay, and F. Shull, "Are Two Heads Better Than One? On the Effectiveness of Pair Programming," IEEE Software, November/December 2007.
[8] E. Arisholm, H. Gallis, T. Dyba, and D. Sjoberg, "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise," IEEE Trans. on Software Engineering, vol. 33, no. 2, 2007.
[9] M. West, Effective Teamwork, 2nd Ed., BPS Blackwell.
[10] C. Jones, "Development Practices for Small Software Applications," Crosstalk: The Journal of Defense Software Engineering, vol. 21, no. 2, 2008.
[11] R. Sison, "Investigating Pair Programming in a Software Engineering Course in an Asian Setting," Proc. APSEC 2008.
[12] B. Boehm and R. Turner, Balancing Agility and Discipline, Addison Wesley, 2004.
[13] E. Mendes, L. Al-Fakhri, and A. Luxton-Reilly, "Investigating Pair-Programming in a 2nd-year Software Development and Design Computer Science Course," Proc. ITiCSE 2005.
[14] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair Programming," IEEE Software, vol. 17, no. 4, 2000.
[15] K. Beck, Extreme Programming Explained: Embrace Change, 2nd Ed., Addison Wesley, 2000.
[16] L. Williams, D. McCrickard, L. Layman, and K. Hussein, "Eleven Guidelines for Implementing Pair Programming in the Classroom," Proc. Agile 2008.