In Support of Student Pair-Programming

Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
williams@csc.ncsu.edu

Richard L. Upchurch
Computer and Information Science Department
University of Massachusetts Dartmouth
N. Dartmouth, MA 02747-2300
rupchurch@umassd.edu

"Knowledge is commonly socially constructed, through collaborative efforts toward shared objectives or by dialogues and challenges brought about by differences in persons' perspectives." [1]

ABSTRACT

Industry, particularly those following the eXtreme Programming (XP) methodology [2], has popularized the use of pair-programming. The pair-programming model has also been found to be beneficial for student programmers. Initial quantitative and qualitative results, which will be discussed in this paper, demonstrate that the use of pair-programming in the computer science classroom enhances student learning and satisfaction and reduces the frustration common among students. Additionally, the use of pair-programming relieves the burden on the educators because students no longer view the teaching staff as their sole form of technical information. We explore the nature of pair-programming, then examine the ways such a practice may enhance teaching and learning in computer science education.

1 INTRODUCTION

Industry, particularly those following the eXtreme Programming (XP) methodology [2], has popularized the use of pair-programming, where two programmers develop software side by side at one computer. The two programmers work collaboratively on the same algorithm, design or programming task. One person is the "driver" and has control of the pencil/mouse/keyboard and is developing the design or code. The other person, the "observer," continuously and actively examines the work of the driver – watching for defects, thinking of alternatives, looking up resources, considering strategic implications of the work at hand, and asking questions. The observer identifies tactical and strategic deficiencies in the work.

Computer science educators around the country are expressing interest in applying pair-programming in educational settings. Much of this interest is sparked by anecdotal evidence from industry extolling the benefits of the practice. Yet a growing body of empirical evidence indicates its efficacy as an educational practice [3-6]. We examine the experimental data to understand the benefits of pair-programming in an educational context. This paper presents the results of that investigation.

2 INVESTIGATIVE PATHS

In [7] six aspects software engineering effectiveness for pair-programming are discussed. These investigative aspects can be briefly described:

Economics. As documented in [8], there was a 15% reduction in defect count with a (15%) increase in development time with the second person.

Satisfaction. Students working in pairs found the experience more enjoyable than working alone.

Continuous Reviews. Pair-programming's shoulder-to-shoulder technique served as a continual design and code review, leading to most efficient defect removal rates.

Problem solving. Pair-programmers refer to the team's ability to solve "impossible" problems faster.

Learning. Pair-programmers repeatedly cite how much they learn from each other.

Team Building and Communication. Interview participants describe that people learn to discuss and work together. This improves team communication and effectiveness.

Here we wish to elaborate on each of these investigative paths as they apply in the educational context.

3 ECONOMICS

The affordability of pair-programming is a key issue for industrial managers. If it is much more expensive, managers simply will not permit it. Skeptics assume that incorporating pair-programming will double code development expenses and critical manpower needs.

A controlled experiment to investigate the economics of pair-programming was conducted [8]. Advanced undergraduates in a Software Engineering course participated in the experiment. One third of the class coded class projects by themselves, following the Personal Software Process (PSP) [9] to track and improve their work habits. The rest of the class completed their projects with a collaborative partner (the same partner for the entire course) following a process similar to the PSP, the Collaborative Software Process (CSP) [8]. CSP was developed specifically to leverage the power of two programmers working together.

Students recorded time spent on their assignments via a web-based data recording and information retrieval system. The results are shown below in Figure 1. After an initial adjustment period in the first program (the "jelling" assignment, which took approximately 10 hours), the pairs only spent about 15% more working hours in total - or 42.5% fewer elapsed hours - completing their assignments compared to the individuals. Along with code development costs, however, other expenses,

such as quality assurance and field support costs must also be considered in industry. When these factors are considered, a detailed economic analysis of yields results largely in favor of pair-programming [10].

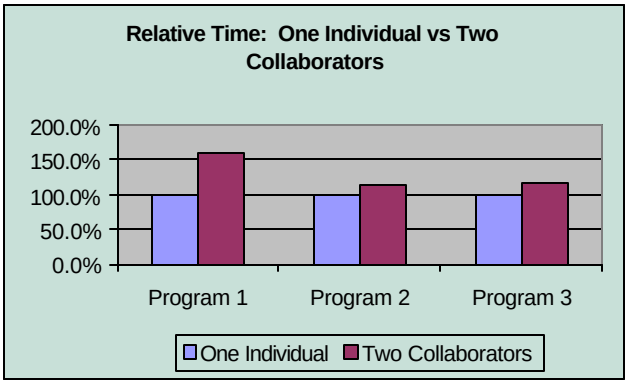

Figure 1: Programmer Time

Significantly, in the experiment, the final code had about 15% fewer defects [8]. (These results are statistically significant with $p < .001$.) Figure 2 shows the post-development test cases the students passed for each program – essentially the percentage of the instructor's test cases passed.

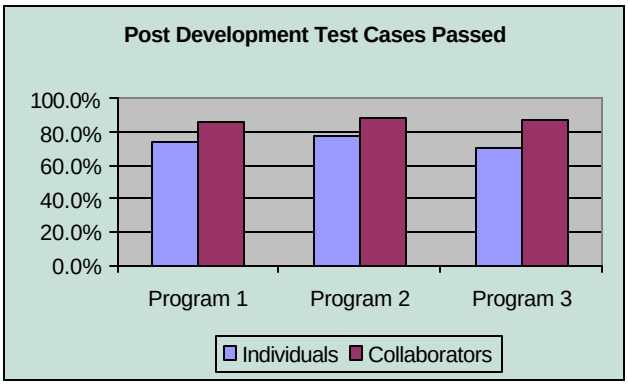

Figure 2: Code Defects

From an industrial perspective, pair-programming can be justified on purely economic grounds, but how are issues of economics addressed in educational settings?

The use of pair-programming may decrease staffing requirements. The quantity of grading is reduced because two students turn in one assignment. There are far less questions from students. When one partner does not know/understand something, the other almost always does. Between the two of them, they can tackle anything, which makes them much less reliant on the teaching staff. Technical email questions are significantly reduced.

4 SATISFACTION

For all the good intentions and diligent work of computer science educators, students find introductory computer science courses very frustrating—so frustrating that typically one-quarter of the students drop out of the classes and many others perform poorly. Incorporation of pair-programming in the classroom may ameliorate these problems. In the classes where pair-programming was used, students were happier and less frustrated with the class. They had the camaraderie of another peer while they completed their assignments. Between the two in the pair, they could figure most everything out. Students were more confident in their work. They felt good that they had a peer helping them to remove and prevent defects. They also felt good that they were better able to come up with more creative, efficient solutions when working with a partner.

It stands to reason that if students are to work hard, they will do so only if they believe that they can and will succeed through their efforts. This has been experimentally demonstrated in a number of other settings. A group who expected to improve with practice showed improvement with practice, whereas the fixed ability group showed a steady decline in performance goals, efficiency of problem solving, and actual performance [11, 12]. In studies of mathematics instruction those practices that improve students' tendencies to persist and feelings of self-efficacy were found to be most effective [11]. Similarly, in writing instruction increasing students' persistence and thereby time-on-task has been found to increase the quality of writing [13]. Hence, we believe pair-programming can demonstrate similar effects through increasing the students' persistence by strengthening their control over the learning context.

5 CONTINUOUS REVIEWS

Inspections were introduced more than twenty years ago as a cost-effective means of detecting and removing defects from software. Results [14] from empirical studies consistently profess the effectiveness of reviews. Even still, most students and professional programmers do not find inspections enjoyable or satisfying. As a result, inspections are often not done unless mandated, and many inspections are held with underprepared inspectors.

The theory on why inspections are effective is based on the knowledge that the earlier a defect is found in a product, the cheaper it is to fix the defect. Many sources, including [16] state that it is ten times more expensive to remove a defect for each additional process step. The continuous reviews of pair-programming perhaps provide the ultimate in defect removal efficiency.

The continuous reviews of pair- programming create a unique educational opportunity, whereby pairs are endlessly learning from each other.

> *"Indeed, review has a unique educational capability: The process of analyzing and critiquing software artifacts produced by others is a potent method for*

learning about languages, design techniques, application domains, and so forth [15]."

This opportunity may prove to be critical for learning. Studies have shown that students who reflect on what they are learning learn better both on declarative and procedural tasks [17], and inducing students to reflect upon their work is effective [18, 19, 20]. Referred to in the literature as self-explanations, pair-programming demands an ongoing dialog between the partners. Hence, the "driver" must reflect on her work as the "observer" insists on darifications and explanations as a solution evolves. It is this reflective act that appears to be essential for developing higher levels of skill [21].

6 PROBLEM SOLVING

"There were times we felt that we would have given up except that we "tag teamed." I'd be on the ropes and I'd describe the problem in such a way that he had a valuable insight. Then he'd fight on as long as he could and stop . . . then I'd have an insight . . . and so on. I suppose others would call it brainstorming, but it feels different to me."

[-D. Wagstaff, software engineer, Salt Lake City]

Pair relaying is our name for the effect Wagstaff describes. Indeed, pairs consistently report that they solve problems faster, and that it is different from improving design quality, or detecting typing errors, or brainstorming. By "problem solving", we refer to when the two are puzzled as to why something doesn't work as expected, or simply can't figure out how to go forward. Pair-programmers describe contributing their knowledge to the best of their abilities. They share their knowledge and energy (and also brainstorming) in turn, chipping steadily away at the problem.

Flor [22] reported on distributed cognition in a collaborative programming pair. Flor recorded via video and audiotape the exchanges of two programmers working together on a software maintenance task. In [22], he correlated specific verbal and non-verbal behaviors of the two under study with known distributed cognition theories. One of these theories is "Searching Through Larger Spaces of Alternatives."

"A system with multiple actors possesses greater potential for the generation of more diverse plans for at least three reasons: (1) the actors bring different prior experiences to the task; (2) they may have different access to task relevant information; (3) they stand in different relationships to the problem by virtue of their functional roles. . . An important consequence of the attempt to share goals and plans is that when they are in conflict, the programmers must overtly negotiate a shared course of action. In doing so, they explore a larger number of alternatives than a single programmer alone might do. This reduces the chances of selecting a bad plan [22]."

A pair programmer's description matches Flor's:

"We often came up with different ideas about how the design should go and the result of arguing over which one was better often led to a truly superior hybrid design."

Problem solving via pair-programming appears equivalent to techniques used effectively by others in helping students develop problem solving skills. Whimbey and Lochhead [23] advocated paired problem solving as a means to focus students on the problem solving process.

7 LEARNING

Knowledge is constantly being passed between partners, from tips on tool usage, to programming language rules, design and programming idioms, and overall design skill. From an educational perspective we describe three distinctive kinds of knowledge: (a) declarative knowledge ("knowledge that"), (b) procedural knowledge ("how to knowledge"), and (c) metacognitive knowledge (self-monitoring, agency, reflection). Declarative knowledge refers to the kind of knowledge typically learned from textbooks--facts and concepts. Procedural knowledge refers to being able to do something, be it writing code, proceeding through analysis and design, using a software process approach, or writing a paper about ethics in the software industry. Metacognitive knowledge refers to a person's skill at planning strategy, monitoring process and progress, changing what one is doing when appropriate, and reflecting on the process so that one can discover ways to improve. The acquisition of metacognitive skills is rarely addressed, either by instruction or by assignment.

Furthermore, research, both within the software domain and in other fields, makes it very clear that each of these ways of knowing must be learned explicitly. For example, studying instructional text is not a sufficient basis for students to solve LISP programs, whereas doing one programming problem improves the probability of doing a second one by 50% [24]; adding an instructional example of how to construct the program produce improvements of over 60% [25].

Pair-programming was used exclusively in a web programming class at the University of Utah, taught by the first author. The class consisted of 20 juniors and seniors, familiar with programming, but not with web programming languages and tools. The majority of the students had only used WYSIWYG web page editors prior to taking the class. During the eleven-week semester, the students learned advanced HTML, JavaScript, VBScript, Active Server Page Scripting, Microsoft Access/SQL and some ActiveX commands. In many cases, they had to intertwine statements from all these languages in one program listing.

Unusual for such students, they produced their programs with minimal questions of the teaching staff. The

collaborating students were queried about the reasons for their independence in an anonymous survey on the last day of class.

- 74% wrote "between my partner and me, we could figure everything out."
- 84% of the class agreed with the statement "I learned Active Server Pages faster and better because I was always working with a partner."

We would attribute part of this result to enhanced problem solving in pairs, as described above, and part to enhanced pair learning. Pair-programming may contextualize the learning activity in a manner that allows the students to focus on the different knowledge types, and provide the feedback necessary to increase their ability to develop monitoring mechanisms for their own learning activities.

8 TEAM BUILDING & COMMUNICATION

Learning to work together means that the people on the team will share both problems and solutions more quickly, and be less likely to have hidden agendas from each other. Teamwork is enhanced.

If the pair can work together, then they learn ways to communicate more easily and they communicate more often. This raises the communication bandwidth and frequency within the project, increasing overall information flow within the team. Rotating partners increases the overall information flow farther.

9 FOR EDUCATORS

There are several benefits for the educator who incorporates pair-programming into their classroom. The number of cheating cases teachers need to deal with is reduced. We believe that pair-programming cuts down on cheating because pair-pressure causes the students to start working on projects earlier and to budget their time more wisely. Additionally, the students have a peer to turn to for help, and therefore, do not feel as helpless.

Naturally, though, pair-programming requires the teaching staff to deal with obvious workload imbalances between the partners that they would not have to deal with if each worked individually. Normal two-person team projects are divided into "my" part and "your" part. However, with collaborative programming, the entire project is "ours." Because of this, we experienced far less partner problems than have been observed in other classes in which students worked in traditional two-person teams, though these situations did arise. We have always given the same grade to both students in the pair. However, students were given formal communication mechanisms to report on the contribution of their partner and to self-assess their own contribution. It appears that students tolerated periods of lower contribution by their partner in times of excessive workload in exchange for similar treatment in their own time of need. However, students report working side by side with equal contribution over 80% of the time.

The following are some guidelines for educators who are embarking on making the transition to pair-programming in their classroom. These guidelines are based on experiences with doing the same:

- It is very important to provide the students some class/lab time to work with their partner. During this time, the pair "bonds" and will plan their next meeting. Requiring students to work together without "forcing" them to start working together can easily lead to failure. During the required class/lab time, the teaching staff can ensure the two are working together at one computer and that the roles of driver and observer are rotated .
- Students need to be given a formal mechanism for reporting on the contributions of their partner and to provide a self-assessment of their own contribution.

10 SUMMARY AND FUTURE WORK

Our experiments and experiences with pair-programming in the computer science classroom have been favorable. Ultimately, students are able to complete programming assignments faster and with higher quality. Students communicate with each other more, appear to learn faster, and are happier and less frustrated. The temptation to cheat is greatly reduced. Additionally, the workload of the educators is reduced.

Based on the above discussion, we believe more computer science educators will attempt to embrace pair-programming as part of their undergraduate program. We hope to coordinate, through an NSF grant that has been proposed, the results of these classes to make a more confident, widespread recommendation about the use of pair-programming in the classroom. Specifically, we want to measure the short-term and long-term success of the student, retention rates particularly among minority groups, and student satisfaction.

Much of what we stated above relates to claims regarding learning and pair-programming. Moving pair-programming to the mainstream of computer science education requires more than our conjectures. We believe the cognition underlying pair-programming help develop the skills needed in the acquisition of expertise [26]. We need an understanding of the types of learning and the nature of that learning. We need empirical studies that help illuminate the realities of activity as an educational practice.

REFERENCES

[1] G. Salomon, *Distributed Cognitions: Psychological and Educational Considerations.* Cambridge: Cambridge University Press, 1993.

[2] K. Beck, *Extreme Programming Explained:*

Embrace Change. Reading, Massachusetts: Addison-Wesley, 2000.

[3] J. T. Nosek, "The Case for Collaborative Programming," in *Communications of the ACM*, vol. March 1998, 1998, pp. 105-108.

[4] L. A. Williams & R. R. Kessler, "The Effects of "Pair-Pressure" and "Pair-Learning" on Software Engineering Education," presented at Conference on Software Engineering Education and Training, Austin, TX, 2000.

[5] L. Williams, R. Kessler, W. Cunningham, & R. Jeffries, "Strengthening the Case for Pair-Programming," in *IEEE Software*, vol. 17, 2000.

[6] L. Williams & R. Kessler, "Experimenting with Industry's "Pair-Programming" Model in the Computer Science Classroom," *Journal of Computer Science Education*, vol. December 2000, 2000.

[7] A. Cockburn & L. Williams, "The Costs and Benefits of Pair Programming," presented at eXtreme Programming and Flexible Processes in Software Engineering -- XP2000, Cagliari, Sardinia, Italy, 2000.

[8] L. A. Williams, "The Collaborative Software Process PhD Dissertation," in *Department of Computer Science*. Salt Lake City, UT: University of Utah, 2000.

[9] W. S. Humphrey, *A Discipline for Software Engineering*. Reading, Massachusetts: Addison Wesley Longman, Inc, 1995.

[10] L. Williams & H. Erdogmus, "An Economic Analysis of Collaborative Programming," submitted to Metrics 2000, London, England, 2001.

[11] A. Bandura, *Self-efficacy: The Exercise of Control*. New York: Freeman, 1997.

[12] R. Wood & A. Bandura, "Social Cognitive Theory of Organizational Mangement Special Issue: Theory development Forum.," *Academy of Management Review*, vol. 14, pp. 361-384, 1989.

[13] J. R. Hayes & J. G. Nash, "On the Nature of Planning in Writing," in *The Science of Writing: Theories, Methods, Individual Differences, and Applications.* Mhway, NJ: Lawrence Erlbaum, 1996, pp. 29-55.

[14] M. E. Fagan, "Advances in software inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, pp. 182-211, 1976.

[15] P. M. Johnson, "Reengineering Inspection: The Future of Formal Technical Review," in *Communications of the ACM*, vol. 41, 1998, pp. 49-52.

[16] W. S. Humphrey, *Introduction to the Personal Software Process*. Reading, Massachusetts: Addison-Wesley, 1997.

[17] P. Pirolli & M. Recker, "Learning Strategies and Transfer in the Domain of Programming," *Cognition and instruction*, vol. 12, 1994.

[18] B. Berardi-Colletta, L. S. Buyer, R. L. Dominowski, and E. R. Rellinger, "Metacognition and Problem Solving: A Process-Oriented Approach," *Journal of Experimental Psychology: Learning, Memory, and Cognition*, vol. 21, pp. 205-221, 1995.

[19] M. Chi, N. de Leeuw, M. Chiu, & C. Lavancher, "Eliciting Self-Explanations Improves Understanding," *Cognitive Science*, vol. 18, pp. 439-477, 1994.

[20] R. L. Upchurch & J. E. Sims-Knight, "In Support of Student Process Improvement," Proceedings of CSEE&T'98, February 22-25, 1998, Atlanta, Georgia. Los Alamitos: IEEE Computer Society Press. p. 114-123

[21] R. L. Upchurch & J. E. Sims-Knight, "Integrating Software Process in Computer Science Curriculum," Proceedings of the Frontiers in Education Conference, Pittsburgh, PA, November 5-8, 1997.

[22] N. V. Flor & E. L. Hutchins, "Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Maintenance," presented at Empirical Studies of Programmers: Fourth Workshop, 1991.

[23] A. Whimbey & J. Lochhead, *Problem Solving and Comprehension*. Philadelphia: Franklin Institute Press, 1980.

[24] J. R. Anderson, F. Conrad, & A. Corbett, "Skill Acqisition and the Lisp Tutor," *Cognitive Science*, vol. 13, pp. 467-505, 1989.

[25] P. Pirolli, "Effects of Examples and Their Explanation in a Lesson on Recursion: A Production System Analysis," *Cognition and Instruction*, vol. 8, pp. 207-259, 1991.

[26] J. E. Sims-Knight & R. L. Upchurch, "The Acquisition of Expertise in Software Engineering Education.," Proceedings of Frontiers in Education, November 4-7, 1998, Tempe, AZ.