The Effects of "Pair-Pressure" and "Pair-Learning" on Software Engineering Education

Laurie A. Williams
University of Utah
lwilliam@cs.utah.edu

Robert R. Kessler
University of Utah
kessler@cs.utah.edu

Note to Reviewer: A formal experiment of Collaborative vs. Individual Programming is underway during Fall 1999 Semester as the first author's PhD dissertation. Many factors will be examined quantitatively in comparing the two techniques – for example, cycle time, productivity, and quality. The results of this class obviously cannot be reported in this paper. However, they can be shared with the conference audience in March if the paper is selected for presentation.

> "Traditionally, collaboration in the classroom . . . has been taboo, condemned as a form of cheating. Yet what we discover . . . is that collaboration can only make our classrooms happier and more productive [1]."

Introduction

Anecdotal evidence from several sources, primarily in industry, indicates that two programmers working collaboratively on the same design, algorithm, code, or test perform substantially better than the two working alone. In this technique, often called "pair programming" or "collaborative programming," one person is the "driver" and has control of the pencil/mouse/keyboard and is writing the design or code. The other person continuously and actively observes the work of the driver – watching for defects, thinking of alternatives, looking up resources, and considering strategic implications of the work at hand. A course in web programming was taught at the University of Utah in Summer Semester 1999. In this course, the students worked in pairs, continuously collaborating on all programming assignments. Using the technique, the students applied a positive form of "pair-pressure" on each other, which proved beneficial to the quality of their work products. The students' also benefited from "pair-learning," which allowed them to learn new languages faster and better than they had experienced with solitary learning. "Pair-learning" also reduced the workload of the teaching because the students no longer relied primarily on them for technical support and advise.

Pair Programming: Evidence of Success

Anecdotal and initial statistical evidence indicate pair programming is highly beneficial. In *Extreme Programming* (XP), an emerging software development methodology, all production code is written collaboratively with a partner. XP was developed initially by Smalltalk code developer and consultant Kent Beck with colleagues Ward Cunningham and Ron Jeffries. XP's requirements gathering, resource allocation, and design practices are a radical departure from most accepted methodologies. Customer requirements are written as fairly informal "User Story" cards; a rough "effort" estimate is assigned to the cards. The cards are assigned to a programming pair, and coding begins. With no *formal* design procedures or discussions on overall system planning or architecture, the pair determines which code in the code base needs to be added or changed. This practice requires the use of "Collective Code Ownership" whereby any programming pair can modify or add to any code in the code base, regardless of the original programmer. Extensive unit testing is continually performed on this ever-enlarging code base.

The evidence of XP's success is highly anecdotal, but so impressive that it has aroused the curiosity of many highly-respected software engineering researchers and consultants. The largest example of its accomplishment is the sizable Chrysler Comprehensive Compensation system launched in May 1997. After finding significant, initial development problems, Beck and Jeffries restarted this development using XP principles. The payroll system pays some 10,000 monthly-paid employees, has 2,000 classes and 30,000 methods [2], went into production almost on schedule, and is still operational today.

XP attributes great success to their use of pair programming by **all** their programmers, experts and novices alike. XP advocates pair programming with such fervor that even prototyping done solo is scrapped and re-written with a partner. One key element is that while working in pairs a continuous code review is performed, noting that it is amazing how many obvious but unnoticed defects become noticed by another person watching over their shoulder. Results [3] demonstrate that two programmers work together more than twice as fast and think of more than twice as many solutions to a problem as two working alone, while attaining higher defect prevention and defect removal, leading to a higher quality product.

In addition, two other studies support the use of pair programming. Larry Constantine, a programmer, consultant, and magazine columnist reports on observing "Dynamic Duos" during a visit to P. J. Plaugher's software company, Whitesmiths, Ltd., providing anecdotal support for

collaborative programming. He immediately noticed a room full of two programmers working on the same code at one computer. He reports, “Having adopted this approach, they were delivering finished and tested code faster than ever . . . The code that came out the back of the two programmer terminals was nearly 100% bug free . . . it was better code, tighter and more efficient, having benefited from the thinking of two bright minds and the steady dialogue between two trusted terminal-mates . . . Two programmers in tandem is not redundancy; it’s a direct route to greater efficiency and better quality.”[4]

An experiment by Temple University Professor Nosek studied 15 full-time, experienced programmers working for 45 minutes on a challenging problem, important to their organization, in their own environment, and with their own equipment. Five worked individually, ten worked collaboratively in five pairs. Conditions and materials used were the same for both the experimental (team) and control (individual) groups. This study provided statistically significant results, using a two-sided t-test. “To the surprise of the managers and participants, all the teams outperformed the individual programmers, enjoyed the problem-solving process more, and had greater confidence in their solutions.” The groups completed the task 40% more quickly and effectively by producing better algorithms and code in less time. The majority of the programmers were skeptical of the value of collaboration in working on the same problem and thought it would not be an enjoyable process. However, results show collaboration improved both their performance and their enjoyment of the problem solving process [5].

Additionally, in an anonymous survey [6] of professional pair programmers, 100% agreed that they had more confidence in their solution when pair programming than when they program alone. Likewise, 96% agreed that they enjoy their job more when programming in pairs. Says one survey respondent: “I strongly feel pair programming is the primary reason our team has been successful. It has given us a very high level of code quality (almost to the point of zero defects). The only code we have ever had errors in was code that wasn’t pair programmed . . . we should really question a situation where it isn’t utilized.”

Pair-Learning in the Classroom

But, how about in our classrooms? Can programming students also benefit from collaborative programming? Larry Constantine, who’s observation of P. J. Plaugher’s software company were reported above, noted that “. . . for language learning, there seems to be an optimum number of

students per terminal. It's not one . . . one student working alone generally learns the language significantly more slowly than when paired up with a partner [4]." A class taught at the University of Utah this summer set out to study pair programming in an educational setting.

The class, **Collaborative Development of Active Server Pages**, consisted of 20 juniors and seniors. The students were very familiar with programming, but not with the Active Server Pages (ASP) web programming languages learned and used in 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, the students would need to intertwine statements from all these languages in one program listing – some of the content running on the browser and some running on the NT server, adding to the overall complexity of the program. The majority of the students had only used WYSIWYG web page editors prior to taking the class. Upon course completion, the students were all writing web scripts that had significant dynamic content that accessed and updated a Microsoft Access database – applications similar (though smaller) to what you would find on a typical e-commerce web site.

Each student was paired with another student to work with for the entire semester. Tests were, however, taken individually. They understood that they were not to break the class project into two pieces and integrate later. Instead they were to work together (almost) all the time on one product. These requirements were stated in the course announcement and were re-stated at the start of the class. The students received instruction in effective pair-programming and read a paper[7], prepared by the authors, which helped prepare them for their collaborative experience. Most skeptically, but enthusiastically, embarked on making the transition from solo to collaborative programming.

The class developed their web sites using a disciplined software development process, the **Collaborative Software Process** (CSP), which has been developed by the first author and is based on the **Personal Software Process**[8] (PSP) developed by Watts Humphrey at the Software Engineering Institute. The PSP defines a framework that includes defined processes and measurement and analysis techniques to help engineers understand their own skills and improve personal performance. Each process has a set of scripts giving specific steps to follow and a set of templates or forms to fill out to ensure completeness and to collect data for measurement-based feedback. This measurement-based feedback allows the programmers to measure their work, analyze their problem areas, and set and make goals. For example,

programmers record information about all the defects that they remove from their programs. They can use summarized feedback on their defect removal to become more aware of the types of defects they make to prevent repeating these kinds of defects. Additionally, they can examine trends in their defects per thousand lines of code (KLOC).

The CSP adapts the PSP for work by collaborative teams. Most of the data input and measurement-based feedback forms have been changed to capture and report on the effectiveness of individual vs. collaborative work within the team. Additionally, many of the scripts have been modified to direct the roles of the "driver" programmer and the "non-driver" programmer.

The PSP has several strong underlying philosophies shared with the CSP. One is that the longer a software defect remains in a product, the more costly it is to detect and to remove the defect. Therefore, thorough design and code reviews are performed for most efficient defect removal. Another philosophy is that defect prevention is more efficient than defect removal. Working in pairs, collaborative programmers perform a continuous code review. This is, perhaps, the ultimate implementation of PSP's "defect prevention" and "efficient defect removal" philosophies. One student in the class commented,

> *"When I worked on the machine as the driver, I concentrated highly on my work. I wanted to show my talent and quality work to my partner. When I was doing it, I felt more confident. In addition, when I had a person observing my work, I felt that I could depend on him, since this person cared about my work and I could trust him. If I made any mistakes, he would notice them, and we could have a high quality product. When I was the non-driver, I proofread everything my partner typed. I felt I had a strong responsibility to prevent any errors in our work. I examined each line of code very carefully, thinking that, if there were any defects in our work, it would be my fault. Preventing defects is the most important contribution to the team, and it put a bit of pressure on me."*

During the class, the students gave feedback on their collaborative experiences and using the CSP in three ways. First, six times throughout the semester, the students completed web-based journal entries. In these journals, the students answered specific questions given by the instructor (and were also given the freedom to write on whatever was on their mind). Some example questions are listed below:

1) It has been said among teachers, “You do not know it unless you can teach it.” Do you find any value to yourself in explaining your work to your partner.
2) Do you feel like you have learned anything about Active Server Pages programming just by reading your partner’s code?
3) What was the biggest hurdle you have had to overcome as a collaborative programmer?
4) What kinds of things does the non-driver do as he/she observes?
5) Which development phases have you tried to work together the most?
6) If you work separately, what do you do with the separate work when you get back together?
7) Which development phases have you found it is OK to work separately at times on?
8) What do you think is the biggest advantage of collaborative programming?
9) What do you think is the biggest problem with collaborative programming?

Additionally, three times throughout the semester, the students completed anonymous surveys on their collaborative experience. Lastly, as part of the final exam, the students wrote a letter objectively giving advise to future collaborative programmers. Highlights of all these forms of qualitative feedback on collaborative programming are reported below.

Pair-Pressure on Quality

Consider quality as a multi-dimensional measure of delivering a software product to a customer where: 1) the product is what they want, 2) they get it when they want it, 3) the product is defect-free. Collaborative programming and the effects of “pair-pressure” seemed to have a positive effect on each of these. First, the students noted that the “two heads are better than one” principle assisted them in translating customer requirements into product designs that would delight the customer. These students then flawlessly delivered their products to their (teaching staff) customers on time. Each of the ten collaborative groups turned in eight projects – all 80 were on time. Additionally, all projects were of very high quality. The average grade on all 80 assignments was 98%!

This same group of students did not perform so flawlessly on their individual work. The students had one in-class midterm exam, one take-home final exam, and one paper evaluating the collaborative process. The average on these items was 78.1% with a standard deviation of 20.91. Again, the average on the collaborative aspects of the class was 97.9% with a standard deviation of 6.74. The students performed much more consistently and with higher quality in pairs than they did individually – even the lazier students performed well on the programming

projects. Through the almost weekly journal entries, the students communicated that this performance was not due to one person carrying the load of two – except on one of the 80 assignments. The students were queried about the reasons for these performance differences in an anonymous survey on the last day of class. There were two overwhelming responses: 74% noted that "*between my partner and I, we could figure everything out*;" 63% noted that "*it was the pair-pressure – I could not let my partner down*." (Of those that did not indicate that pair-pressure was a cause, essentially all noted that "I always do well on my programming assignments.") Overall, **95%** of the class agreed with the statement *"I was more confident in our assignments because we pair programmed."* One student noted,

> *"One day, after I did a lot of testing on our project, I was pretty sure that the project was high quality. I then gave it to my partner. I did not expect he would find any errors. Guess what? He found an error in just two minutes! Oh dear, why didn't I notice that? We all know, two heads are better than one. Pair programming enabled our project to have higher defect prevention and defect removal. As a result, we got a higher quality product."*

The students also felt they were much more productive when working collaboratively. There were two main reasons observed. First, when they met with their partner they both worked very intensively -- each kept the other on task (no reading emails or surfing the web) and was highly motivated to complete the task at hand during the session. (Contrast that with the productivity and quality expected from this student who admitted in his collaborative evaluation paper, "When I worked on assignments individually, I could watch TV while I worked on it.") Secondly, having a constant observer watching over their shoulder served as an extremely efficient defect removal method – though perhaps a bit unnerving at first.

Pair-Pressure on Students

The students were extremely positive about their collaborative experience. On an anonymous survey, **84%** of the class agreed with the statement *"I enjoyed doing the assignments more because of pair programming."* Additionally, **84%** of the class agreed with the statement *"I learned Active Server Pages faster and better because I was always working with a partner."* Specifically, they noted that they were surprised how much it helped them to understand things when they had to explain it to another and when they read their partner's code. As one student said,

> *"When I explained an idea to my partner, I concentrated on what I was saying, and carefully made things clear and logical because I did not want to confuse my*

partner and I wanted him to understand what I was talking about. It helped me better understand the problem I was addressing. It also helped me discover some mistakes I had made but did not notice before I talked with my partner."

Together, defect removal was much more efficient, which significantly reduced the frustration level of debugging they had been accustomed to. One student noted in his final collaborative paper:

"*One problem with single programming is that you can forget what you are doing and easily get wrapped in a few lines of code, losing the big picture. Your partner is able to constantly review what you do, making sure that it is in line with the product design. He/she can also make sure that you are not making the problem too difficult. Many times, these two items alone can waste a lot of time. When it comes down to it, wouldn't you rather just get the job done correctly and quickly? Collaborative programming will help you do just that."*

The collaboration made them confident in their work – giving them a "We nailed that one!" feeling. This sentiment made them feel more positive about the class overall.

The students adjusted their collaborative habits throughout the semester – as they got to know their partner better and as they realized which parts of the development process benefited more from collaboration than others. By the end of the semester, all realized it was essential to do design collaboratively. Many groups also consistently performed complex coding and design/code reviews collaboratively. Some migrated toward doing simple/rote coding and testing separately (though perhaps side-by-side on two computers).

Pair-Pressure on the Teaching Staff

Collaboration also makes the instructor feel more positive about the class. Their students are happier and the assignments are handed in on-time and are of higher quality. There is one additional very positive effect for the teaching staff -- less questions! When one partner did not know/understand something, the other almost always did. Between the two of them, they could tackle anything, which made them much less reliant on the teaching staff. Email questions were almost non-existent. Lab consultation hours were very calm, even the day the projects were due.

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, there were

far less partner problems than have been observed in other classes in which students worked in traditional two-person teams.

Summary

Programmers have generally been conditioned to performing solitary work, rooted at an educational system of individual evaluation. Making the transition to pair programming involves breaking down some personal barriers beginning with the understanding that *talking is not cheating*. First, the programmers must understand that the benefits of intercommunication outweigh their common (perhaps innate) preference for working alone and undisturbed. Secondly, they must confidently share their work, accepting instruction and suggestions for improvement in order to advance their own skills and the product at hand. They must display humility in understanding that they are not infallible and that their partner has the ability to make improvements in what they do. Lastly, a pair programmer must accept ownership of their partner's work and, therefore, be willing to constructively express criticism and suggested improvements.

The transition to pair programming takes the conditioned solitary programmer out of their "comfort zone." The use of the technique may also take the instructor out of their "comfort zone" because the need to deal with additional issues such as one partner ending up with all the work, how to distribute grades, etc. However, pair programming has the potential of changing how programming classes are taught in order to benefit the students' learning experience and the quality of the software products these students produce.

Bibliography

1. Bennis, W., Biederman, Patricia Ward, *Organizing Genius: The Secrets of Creative Collaboration*. 1997: Addison-Wesley Publishing Company, Inc.
2. Anderson, A., Beattie, Ralph, Beck, Kent et al., *Chrysler Goes to "Extremes"*, in *Distributed Computing*. 1998. p. 24-28.
3. Wiki, *Programming In Pairs*, in *Portland Pattern Repository*. 1999. http://c2.com/cgi/wiki?ProgrammingInPairs.
4. Constantine, L.L., *Constantine on Peopleware*. Yourdon Press Computing Series, ed. E. Yourdon. 1995, Englewood Cliffs, NJ: Yourdon Press.
5. Nosek, J.T., *The Case for Collaborative Programming*, in *Communications of the ACM*. 1998. p. 105-108.
6. Williams, L., *Pair Programming Questionnaire*, 1999, http:\\limes.cs.utah.edu/questionnaire/questionnaire.htm.

7. Williams, L., Kessler, R. *All I Ever Needed to Know About Pair Programming I Learned in Kindergarten*, in *Communications of the ACM*. (to appear).
8. Humphrey, W.S., *A Discipline for Software Engineering*. SEI Series in Software Engineering, ed. P. Freeman, Musa, John. 1995: Addison Wesley Longman, Inc.