Experimenting with Industry's "Pair-Programming" Model in the Computer Science Classroom

Laurie A. Williams
North Carolina State University
(919)513-4151
williams@csc.ncsu.edu

Robert R. Kessler
University of Utah
(801)581-4653
kessler@cs.utah.edu

Abstract

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 would working alone. Two courses taught at the University of Utah studied the use of this technique, often called pair-programming or collaborative programming, in the undergraduate computer science classroom. 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 benefit from "pair-learning," which allowed them to learn new languages faster and better than with solitary learning. The workload of the teaching staff is reduced because the students more often look to each other for technical support and advise.

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. With pair-learning, two students work simultaneously at one computer to complete one program. Using 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, the "observer," continuously and actively examines the work of the driver – watching for defects, thinking of alternatives, looking up resources, and considering strategic implications of the work at hand. The observer identifies tactical and strategic deficiencies in the work. Examples of tactical deficiencies are erroneous syntax, misspelling, or smaller logic mistakes. An example of a strategic deficiency is an implementation that does not map to the design. The student pairs apply a positive form of "pair-pressure" on each other, which has proven beneficial to the quality of their work products. Initial experimental results indicate that pair-learning also improves the success and morale of the students.

Initial experimentation with pair-learning also reveals benefits to computer science educators. Students working in pairs are able to answer each others' questions. They no longer look to the teaching staff as their sole source of technical advice; educators are no longer as burdened by an onslaught of questions. Grading can be significantly reduced when two students submit one assignment. The number of cheating cases is reduced because collaboration is legitimized. The classes are calmer; the students are more satisfied and self-sufficient.

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. 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 (Anderson, Beattie, Beck, & al., 1998), 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. The observer programmer detects an amazing number of obvious but unnoticed defects. XP's qualitative results (Wiki, 1999) 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.

Cognitive theory can help explain why pair-programming might be more effective than solo-programming. In 1991 Nick Flor, a masters student of Cognitive Science at U.C. San Diego, reported on distributed cognition in a collaborative programming pair he studied. Flor recorded via video and audiotape the exchanges of two experienced programmers working together on a software maintenance task. In (Flor & Hutchins, 1991), he correlated specific verbal and non-verbal behaviors of the two programmers 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.* (Flor & Hutchins, 1991)

Additionally, in an anonymous survey (Williams, 2000) 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."

Language Learning

But, how about in our classrooms? Can programming students also benefit from collaborative 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., 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 (Constantine, 1995)." Two classes taught at the University of Utah (summer and fall semesters in 1999) set out to study pair-programming in an educational setting. The second class, which will be discussed in section 4, was a formal empirical study, which studied differences between individual and collaborative programmers.

The first 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. 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, 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. 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 eCommerce web site.

Each student was paired with another student to work with for the entire semester. At the start of the class, the students were asked whom they wanted to work with and whom they did not want to work with. Of the ten collaborative pairs, eight pairs were mutually chosen in that each student had asked to work with their partner. The last two pairs were assigned because the students did not express a partner preference. 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 (Williams & Kessler, 2000), 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.

During the class, the students gave feedback on their collaborative experiences in three ways. First, six times throughout the semester, the students completed web-based journal entries in which they had to answer specific question about their collaborative experience. Additionally, three times throughout the semester, the students completed anonymous. 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 throughout sections 5-8.

The Collaborative Software Process^{SM} (CSP^{SM})

The Software Process

The second course at the University of Utah in which pair-programming was actively used was a senior Software Engineering course. The class learned and used the Collaborative Software Process (CSP) (Williams, 2000), which has been developed by the first author as her doctoral dissertation. CSP is based on the Personal Software Process (Humphrey, 1995) (PSP) developed by Watts Humphrey at the Software Engineering Institute. The PSP is a framework that includes defined processes and measurement and analysis techniques to help software 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 is an extension of the PSP, and it relies upon the foundation of the PSP. The CSP adapts the PSP for work by collaborative teams. The activities of the CSP are specifically geared towards leveraging the power of two software engineers working together on one computer. Most of the forms, and templates of the PSP have been changed in the CSP 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 observer programmer.

The PSP documented in *A Discipline for Software Engineering* (Humphrey, 1995) is aimed at graduate and senior-level undergraduate university courses. It is also very popular in industry. The *Introduction to the Personal Software Process* (Humphrey, 1997) was written to instruct first-year programming students the basics of the PSP. This second book maintains the disciplined philosophy of the original PSP book, but teaches simpler versions of the activities required to develop software in a disciplined manner. The simpler versions are appropriate for beginning students. CSP is aimed at sophomore level students and above; the complexity of the process is in between Humphrey's two books just discussed. Additionally, more recent object-oriented analysis and design and testing techniques were incorporated into the CSP.

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 because the non-driver is

constantly reviewing the work of the driver. Pair-programming is, perhaps, the ultimate form of "defect prevention" and "efficient defect removal." 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.*

Senior Software Engineering Class Experiment

The senior Software Engineering class was structured as an experimental class. The experiment was designed to validate the effectiveness of CSP and to isolate and study the costs and benefits of collaborative programming. The experimental class consisted of 41 juniors and seniors. The students learned of the experiment during the first class. Generally, the students responded very favorably to being part of an experiment that could drastically change the way software development could be performed in the future.

On the first day of class, the students were asked if they preferred to work collaboratively or individually, whom they wanted to work with, and whom they did not want to work with. The students were also classified as "High" (top 25%), "Average," or "Low" (bottom 25%) academic performers based on their GPA. The GPA was not self-reported; academic records were reviewed. Using this information, the twenty-eight students were then assigned to the collaborative group and thirteen to the individual group. The GPA was used to ensure the groups were academically equivalent. The students were also grouped to ensure there was a sufficient spread of high-high, high-average, high-low, average-average, average-low, and low-low pair grouping. This was done in order to study the possible relationship between previous academic performance and successful collaboration. Of the fourteen collaborative pairs, thirteen pairs were mutually chosen in that each student had asked to work with their partner. The last pair was assigned because the students did not express a partner preference.

Students in the collaborative group completed their assignments in pairs using the CSP. Students in the individual group completed all assignments using a modified version of the PSP. The version of the PSP used by the students was modified from that defined in (Humphrey, 1995) in order to parallel the software development approaches defined in the CSP (i.e. object oriented analysis and design and testing techniques were incorporated). Therefore, the only difference between the individual and the collaborative groups was the use of pair programming. All students received instruction in effective pair-programming and were given a paper (Williams & Kessler, 2000) on strategies for successful collaboration. These helped prepare them for their collaborative experience.

Specific measures were taken to ensure that the pairs worked together consistently each week. One class period each week was allotted for the students to work on their projects. Additionally, the students were required to attend two hours of office hours with their partners each week where they also worked on their projects. It is critical for student pairing success to establish these regular meeting times, lest the students get too involved in other classes and their

jobs and never actually work together. During these regular meeting times, the pairs jelled or bonded and were much more likely to establish additional meeting times to complete their work.

A Windows NT data collection and analysis web application was used to accurately obtain data from and provide feedback to the students, as easily as possible for the students. (Disney, 1998) stresses the importance of such a tool for accurate process data collection.

The outcomes from both the web programming and the senior software engineering class will now be discussed.

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. The students almost flawlessly delivered their products to their (teaching staff) customers on time.

Additionally, the students performed much more consistently and with higher quality in pairs than they did individually – even the less motivated students performed well on the programming projects. The students communicated that, in general, this performance was not due to one person carrying the load of two. 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 "*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 "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.*

As reported in (Williams, Kessler, Cunningham, & Jeffries, 2000), in the senior Software Engineering class experiment, the pairs passed significantly more of the automated post-development test cases run by an impartial teaching assistant (see Table 1 below). On average, students that worked in pairs passed 15% more of the instructor's test cases! This quality difference was statistically significant to $p < .01$.

	Individuals	Collaborative Teams
Program 1	**73.4%**	**86.4%**
Program 2	**78.1%**	**88.6%**
Program 3	**70.4%**	**87.1%**
Program 4	**78.1%**	**94.4%**

Table 1: Percentage of Test Cases Passed on Average

Productivity and Learning

The total hours spent on programs was essentially the same for individuals and collaborating pairs. For example, if one individual spent 10 hours on an assignment, each partner would spend slightly more than 5 hours. (Note: the empirical study showed that on average the pairs spent 15% more programmer-hours than the individuals to complete their projects. However, the median time spent was essentially equal for the two groups and the average difference was not statistically significant.)

The students felt they were more productive when working collaboratively. There were several reasons observed. First, when they met with their partner they both worked very intently -- 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. We refer to this effect as "pair-presssure." Software Engineering educators that teach software process often struggle with students' resistance to following a defined process. Both in industry and academia, programmers resist following a defined software process, despite statistical evidence that the process yields superior results. Qualitatively, students followed the software process that was taught in the class more faithfully when working with a partner. "Pair-pressure" causes the students to follow the process and to actually practice what the instructor is teaching. Even if they feel like skipping an important step of the process, such as documenting design, either they are embarrassed to admit this to their partner or their partner talks them into completing the step.

Secondly, having a constant observer watching over their shoulder ("pair-reviews") is perhaps a bit unnerving at first. However, the continuous reviews of collaborative programming, in which both partners ceaselessly work to identify and resolve problems, affords both optimum defect removal efficiency **and** the development of defect prevention skills. The learning that transcends in these continual reviews prevents future defects from ever occurring – and defect prevention is more efficient than any form of defect removal. Says Capers Jones, chairman of Software Productivity Research,

> *It is an interesting fact that formal design and code inspections, which are currently the most effective defect removal technique, also have a major role in defect prevention. Programmers and designers who participate in reviews and inspections tend to avoid making the mistakes which were noted during the inspection sessions* (Jones, 1997).

The continuous reviews of collaborative programming create a unique educational experience, whereby the pairs are endlessly learning from each other. "The process of analyzing and

critiquing software artifacts produced by others is a potent method for learning about languages, application domains, and so forth (Johnson, 1998)."

Additionally, collaborative teams consistently report that together they can evolve solutions to unruly or seemingly impossible problems. "Pair-relaying" is a name we give for the effect of having two people working to resolve a problem together. Partners share their knowledge and energy in turn, chipping steadily away at the problem, evolving a solution to the problem. Through this, pairs report that in their problem solving, they do not spend excessive time lost in a particular problem or fix.

Student Morale

The students were extremely positive about their collaborative experience. 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. Ninety-two percent of the students said they were more confident in their projects when working with a partner; 96% of the students said they enjoyed the class work more when working with a partner.

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 (which we call "debug by explaining") 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).

Teaching Staff Workload

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. The quantity of grading is reduced because two students turn in one assignment. 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. Technical email questions were almost non-existent. Lab consultation hours were very calm, even the day the projects were due.

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. If this arrangement is not possible, it would probably be best to make pair-

programming an optional arrangement for completing assignments, whereby students can choose between working alone or with a partner.

- 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. Additionally, time logs of the students should be examined to determine the percentage of time each worked on the project. The web-based tool discussed above allows for easy examination of student time logs.

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

Anderson, A., Beattie, R., Beck, K., et. al. (1998, October 1998). Chrysler Goes to Extremes. *Distributed Computing, October 1998,* 24-28.

Constantine, L. L. (1995). *Constantine on Peopleware*. Englewood Cliffs, NJ: Yourdon Press.

Disney, A. M., Johnson, Philip M. (1998, November 1998). *Investigating Data Quality Problems in the PSP (Experience Paper)*. Paper presented at the Foundations of Software Engineering, Lake Buena Vista, FL.

Flor, N. V., & Hutchins, E. L. (1991). *Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Maintenance*. Paper presented at the Empirical Studies of Programmers: Fourth Workshop.

Humphrey, W. S. (1995). *A Discipline for Software Engineering. Reading, PA*: Addison Wesley Longman, Inc.

Humphrey, W. S. (1997). *Introduction to the Personal Software Process*. Reading, Massachusetts: Addison-Wesley.

Johnson, P. M. (1998, February 1998). Reengineering Inspection: The Future of Formal Technical Review. *Communications of the ACM, 41,* 49-52.

Jones, C. (1997). *Software Quality: Analysis and Guidelines for Success*. Boston, MA: International Thomson Computer Press.

Wiki. (1999, June 29). Programming In Pairs. *Portland Pattern Repository,* Available at: http://c2.com/cgi/wiki?ProgrammingInPairs.

Williams, L., Kessler, R., Cunningham, W., & Jeffries, R. (2000, July/August 2000). Strengthening the Case for Pair-Programming. *IEEE Software*.

Williams, L. A. (2000). *The Collaborative Software Process PhD Dissertation*. University of Utah, Salt Lake City, UT.

Williams, L. A., & Kessler, R. R. (2000, May 2000). All I Ever Needed to Know About Pair Programming I Learned in Kindergarten. *Communications of the ACM, 43, 108-114*.