# The Effects of Pair-Programming on Performance in an Introductory Programming Course

**Charlie McDowell and Linda Werner**
**Computer Science Department**
**University of California**
**Santa Cruz, CA 95064**
**{charlie,linda}@cs.ucsc.edu**

**Heather Bullock and Julian Fernald**
**Psychology Department**
**University of California**
**Santa Cruz, CA 95064**
**{hbullock,jfernald}@cats.ucsc.edu**

## Abstract

The purpose of this study was to investigate the effects of pair-programming on student performance in an introductory programming class. Data was collected from approximately 600 students who either completed programming assignments with a partner or programmed independently. Students who programmed in pairs produced better programs, completed the course at higher rates, and performed about as well on the final exam as students who programmed independently. Our findings suggest that collaboration is an effective pedagogical tool for teaching introductory programming.

## 1. Introduction

In the academic literature, cooperative or collaborative learning models involve two or more individuals taking turns helping one another learn information [1]. The consensus from numerous field and laboratory investigations is that academic achievement (i.e., performance on a test) is enhanced when an individual learns information with others as opposed to when she or he is alone [2, 3, 4].

Although collaboration has been employed in some software development tasks, computer programming has traditionally been taught and practiced as a solitary activity [5, 6, 7, 8, 9]. Over the last decade, however, a number of advocates of collaborative programming have emerged [10]. In 1991, Flor observed and recorded verbal and non-verbal exchanges between two programmers working collaboratively on a software maintenance task. He [11] found that collaboration allowed each member of the programming dyad to contribute his unique prior experience, task relevant knowledge, and perspective to the problem, resulting in a greater potential for the generation of more diverse plans, and ultimately a greater capacity to solve the problem. His observations both underscore the effectiveness of collaborative programming, and provide evidence for the theory of distributed cognition which asserts that "knowledge is commonly socially constructed, through collaborative efforts toward shared objectives or by dialogues and challenges brought about by differences in persons' perspectives" [12].

In 1995 two additional popular books which discussed collaborative software development practices were published. In "Constantine on Peopleware," Constantine reported observing programming pairs at Whitesmith Ltd. producing code more quickly and with fewer bugs than would be expected of independent programmers [13]. During the same year, Coplien, in "Pattern Languages of Program Design" suggested the "Developing in Pairs Organizational Pattern," which argued that organizations could produce software more efficiently by pairing designers to work collaboratively [14].

In recent years, the growth of extreme programming (XP) has brought considerable attention to collaborative programming. Developed over a fifteen year period by Kent Beck and his colleagues, Ron Jeffries and Ward Cunningham [15], XP is a computer software development approach that credits much of its success to the use of pair-programming by all of their programmers, regardless of experience [16]. The pair-programming dimension of XP requires that teams of two programmers work simultaneously on the same design, algorithm, code, or test [17, 10]. Sitting shoulder to shoulder at one computer, one member of the pair is the "designated driver," and controls the keyboard and mouse while actively creating code. The "non-driver" constantly reviews the keyed data in order to identify tactical and strategic deficiencies, including erroneous syntax and logic, misspelling, and implementations that don't map to the design [10]. After a designated period of time, the partners reverse their roles, or work with other co-workers from the same team on


*SIGCSE'02*, February 27- March 3, 2002, Covington, Kentucky, USA.

another piece of code. Code produced by only one partner is discarded, or reviewed collaboratively before it is integrated.

Anecdotal evidence within industry suggests that the collaborative nature of XP is highly effective. Perhaps the largest and best-known example of successful pair-programming is the Chrysler Comprehensive Compensation system [18]. Plagued by significant development problems, Beck and Jeffries restarted the project using XP programming principles, including the exclusive use of pair-programming. Today, the payroll system pays approximately 10,000 employees and has 2,000 classes and 30,000 methods. The system's success is largely credited to the reduction in defects and improved functionality brought about by pair-programming. Despite the anecdotal evidence, many managers and programmers who have no experience with collaborative programming remain skeptical [5], assuming it will be too costly in terms of scarce programmer hours, or that it will slow programmers down.

In addition to anecdotal evidence, empirical evidence also supports the effectiveness of "pair-programming" or "collaborative learning." Nosek [17] found that students who programmed in pairs outperformed those who worked alone. In a related follow-up study, Nosek randomly assigned 15 full-time experienced programmers to either work as part of a two-member team or to work by themselves on a programming problem for 45 minutes. Final products were assessed in terms of readability (e.g., the degree to which the problem solving strategy could be determined from the subjects' work) and functionality (e.g., the degree to which the strategy accomplishes the objectives stated in the problem description). Teams were found to significantly outperform individual programmers in terms of functionality and readability, to report greater satisfaction with the problem-solving process, and to have greater confidence in their solutions. However, it should be noted that pair-programming was found to take more total programmer time than traditional solo programming, although the elapsed time was less. Pairs required an average of 60 programmer-minutes to complete programming assignments compared to the 42 programmer-minutes used by solo programmers. It should not be concluded, however, that pair-programming requires more time. Nosek did not include time spent debugging in his analysis and this debugging may be expedited in pairs. This point is particularly salient when code quality is considered; Nosek found that code produced by individuals is more error prone than code created by pairs.

Further empirical evidence of the effectiveness of pair-programming is provided by an experimental study conducted by Williams and Kessler at the University of Utah [19]. In this study, 41 upper level students enrolled in a course on web design were randomly assigned to complete four programming projects either independently or in pairs. During each programming cycle, the 13 solo programmers completed one program, while the 14 pairs completed two. Across all four cycles, the collaborators had a mean 15% fewer defects in their programs than the individuals. The difference in the rate of defects was statistically significant ($p<.05$) for all but the first cycle. Furthermore, collaborators spent, on average, only 15% more time completing two projects than the solo programmers spent completing one, suggesting that pair-programming is 40-50% faster than programming alone.

In addition to producing more bug free code, pair-programming appears to enhance the programmers' enjoyment and confidence. Students practicing collaborative programming, as well as professional pair programmers were anonymously surveyed. Over 90% reported enjoying their jobs more when working in pairs, and 95% reported feeling more confident in their solutions [20].

## 2. Method

The findings reported in this paper are part of a larger study funded by the National Science Foundation to assess the effectiveness of pair programming on the performance and retention of women in computer science and related fields. The results reported here are based on a small subsection of the data that examined the effects of pair programming on the quality of the programs produced, and on the extent to which new programming skills were acquired. We expected that programmers who worked in pairs would produce better programs than those who worked independently. Furthermore, we did not anticipate that pair-programming would compromise learning to program.

During the 2000-2001 academic year, data was gathered from approximately 600 students enrolled in four sections of an introductory programming course at the University of California – Santa Cruz designed for CS, ISM and CE majors. The results reported in this paper examine data collected from two sections of the course taught by the same instructor. One of the two sections reported here required students to complete programming assignments in pairs ($N=172$), while the other required students to write programs independently($N=141$). The programming assignments, lectures, and quizzes were comparable, and the final exam was identical in both sections. The other two sections were not considered for the study reported here because they were taught by different instructors.

In the pairing section taught in fall 2000, students were required to complete five programming assignments with a partner. On the first day of class, each student made a list of three potential partners and was assigned one partner by the researchers. Pairs were instructed to alternate "driver" and "nondriver" roles from hour to hour on each assignment. The importance of working together was emphasized throughout the quarter and all students completed a variety of measures to assess the amount of time they spent in each role. In the non-pairing section

taught in spring 2001, students were required to independently complete comparable programming assignments.

Scores on programming assignments and scores on the final exam served as the dependent measures. Programming assignments were scored for functionality and readability. The final exam assessed students' knowledge of programming concepts and their ability to write new code.

## 3. Results

### 3.1 Scores on programming assignments

To compare whether programming scores differed as a function of pair-programming experience, analysis of variance (ANOVA) was conducted. Among all students who completed the course, students in the pairing class scored significantly higher on the programming assignments (M=86%) than those in the non-pairing class (M=67%), F(1, 264)=79.24, p<.001. That the difference between the two classes was statistically significant at the .001 level indicates that we could have expected to obtain means this far apart less than 1 time in 1,000 just by chance. In other words, it is highly unlikely that we would have obtained these results if pairing didn't actually influence the quality of the programs. To review means and standard deviations, see the first two lines of Table 1.

**Table 1: Overall program scores**

| | Mean | Med. | Std. dev. |
|---|---|---|---|
| **Pairing (all)** | 86.3% | 88% | 13.9% |
| **Non-pairing (all)** | 67% | 68% | 21.4% |
| **Non-pairing (top half)** | 77.1% | 80% | 19.6% |

There are at least two possible explanations for this difference. First, it may be that pair-programming enhanced the quality of the output resulting in programs that were more functional and readable. A second possibility is that the mean programming score in the pairing class was artificially inflated. Because both members of the pairs earned the same grade on each of the programming assignments, overall scores in the class may have simply reflected the performance of the stronger student in each pair. In the most extreme case, it is possible that each of the pairs in the pairing section consisted of one partner in the top half and one partner in the bottom half of the class, resulting in a mean programming score for the whole class that only represented the performance of the top 50%. If pair-programming did not improve the quality of the programming assignments, then the scores in the pairing class should have been approximately equal to the scores of the strongest 50% of students in the non-pairing class (assuming students in the two classes were similar to begin with). To test this, we performed an ANOVA to compare the programming scores of all students in the pairing class to the students in the top half of the non-pairing class (student ranking was determined by final exam scores). Overall, the scores from the entire pair-programming section (M=86%) were significantly higher than the scores of the top half of the non-pairing class (M=77%), F(1, 210)=14.03, p<.001. Please see bottom line of Table 1 to review means. This suggests that the best 50 programs from a group of 100 students working alone, would not be as good as the programs produced by 50 pairs of students. Thus, it appears that the very process of working collaboratively improves the quality of programs.

### 3.2 Pair-programming and final exam scores

In addition to the quality of the programs produced, we also examined the effect of pair-programming on students' conceptual understanding of, and ability to program independently. Final exam scores in the two classes were compared using ANOVA. As indicated in Table 2 the mean exam score in the non-pairing class (M= 75%) was slightly higher than the mean exam score in the pairing class (M= 73%). This small difference, however, was not statistically significant, F(1,264)=.46, p >.05, indicating that the difference between the two classes was not large enough to attribute to anything other than chance. This finding suggests that despite the fact that pair-programming results in improved programs, when used to teach programming it appears not to affect the extent to which students master course material and are able to independently apply their knowledge to new problems.

**Table 2: Final exam score**

| | Mean | Median | Std. dev. |
|---|---|---|---|
| **Pairing** | 72.9% | 79.2% | 21.6% |
| **Non-pairing.** | 74.6% | 78.3% | 18.7% |

One factor that may have contributed to the overall class averages on the final exam is the percentage of students who did not finish the class. As Table 3 indicates the percentage of students who finished the final was dramatically higher in the pairing section (92% vs. 76%).

**Table 3: Retention through final exam**

| | attempted class | took final exam | took final |
|---|---|---|---|
| **Pairing** | 172 | 159 | 92.4% |
| **Non-pairing** | 141 | 107 | 75.9% |
| **Fall 1999** | 168 | 142 | 84.5% |

Any number of factors may have contributed to differential attrition rates. For example, students drop rates may be higher during the spring than fall quarter. It is also possible

that students in the non-pairing class hoped to work in pairs and dropped the spring class in order to take it another quarter in which pairing might be utilized. Pairing may increase the likelihood that students complete introductory programming class. Course completion rates were significantly higher in the pairing class (fall 2000) than in the non-pairing section offered in spring 2001, $\chi^2(1)=16.64$, $p<.001$.

For comparative purposes, we also examined course completion rates in a section of introductory programming offered in fall 1999. This section, which was taught by the same instructor as the other two sections and did not employ pair programming, had a completion rate of 85%. Chi-square analyses revealed that the completion rate in the pair programming section was significantly higher (92%) than the completion rates in the fall 1999 class, $\chi^2(1)=5.25$, $p<.05$. The course completion rates in the two non-pairing sections did not significantly differ from one another, $\chi^2(1)=3.66$, $p>.05$.

Regardless of the reasons, the difference in attrition rates between the fall 2000 pairing class and the spring 2001 non-pairing class may have contributed to the slightly higher final exam average in the non-pairing class. If weaker students in the non-pairing class drop, while their counterparts in the pair-programming class chose to stay, these "weak" students may have pulled down the overall exam performance for the class.

In an attempt to compensate for the significant difference in drop rates, we compared the performance of equal percentages of students from each of the two classes (fall 2000 with pair-programming, and spring 2001 without pair-programming). For the non-pairing class (with higher attrition), we included all students that took the final exam (76% of those that attempted the class). For the pairing class, we included only the "top" 76% of those that attempted the class. The top 76% were selected in two ways: (1) by final exam score and (2) by overall class grade.

**Table 4: Final exam score for equal percentages of students that attempted the class.**

| | Mean | Med. | Std. Dev. |
|---|---|---|---|
| **Pairing (all students that took final)** | 72.9% | 79.2% | 21.6% |
| **Pairing (top 76% by grade)** | 82.5% | 83% | 10.7% |
| **Pairing (top 76% by final)** | 82.7% | 83% | 9.8% |
| **Non-pairing (all students that took final)** | 74.6% | 78.3% | 18.7% |

Not surprisingly, this affected "class performance" on the final as Table 4 indicates. As previously discussed the difference between all students in the non-pairing class that took the final (76% of those who attempted the class) and all students in the pairing class that took the final (92% of those that attempted the class) was not significant. On the other hand, the top 76% of students in the pairing class scored significantly higher on the final ($M$= 83%) than students in the non-pairing class ($M$= 75%) regardless of which method was used to select the top 76% [$F(1,239)=15.44$, $p<.001$ and $F(1, 237)=14.21$, $p<.001$ for top 76% based on final and top 76% based on grade, respectively]. Of course, we recognize that there are many reasons why students drop a course other than poor performance, but the current findings are provocative.

## 4. Conclusion

It appears plausible that as a result of pair-programming, students that might otherwise have dropped the course, completed the course. It also appears that the programs of even the better students benefited from pair-programming. This is consistent with collaborative learning research, which shows that academic achievement is enhanced when an individual learns information with others.

We remain optimistic that pair-programming can be used effectively in an introductory programming class. The data suggest that students who work in pairs produce better programs. Furthermore, they perform comparably on exams (when not adjusted for varying attrition rates), and possibly significantly better (when adjusted for attrition rates) on a final exam, to students required to program individually.

## Acknowledgments

The authors wish to thank Jennifer Bevan, Tristan Thomte, and Wendy Williams for their assistance with data collection, entry, and management, and Scott Brandt, and Alex Pang for allowing us to collect data for this study from their winter sections of UCSC's introductory programming course. This work was partially funded by a National Science Foundation grant, EIA-0089989. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References

1. E. M. Horn, W. G. Collier, J. A. Oxford, C. F. Bond, and D. F. Dansereu, "Individual Differences in Dyadic Cooperative Learning," *Journal of Educational Psychology*, 90(1), pages 153-160, 1998.
2. A. M. O'Donnell and D. F. Dansereu, "Scripted Cooperation in Student Dyads: A Method for Analyzing and Enhancing Academic Learning and Performance," in R. Hartz-Lazarowitz and N. Miller (Eds.) *Interactions in Cooperative Groups: The Theoretical Anatomy of Group Learning*, pages 120-141, London: Cambridge University Press, 1992.

3. R. E. Slavin" "Research on Cooperative Learning and Achievement: When We Know, What We Need to Know," *Contemporary Educational Psychology*, 21, pages 43-69, 1996.
4. S. Totten, T. Sills, A.. Digby, and P. Russ. *Cooperative Learning*. New York: Garland, 1991.
5. A. Cockburn and L. Williams, "The Costs and Benefits of Pair Programming," in *Extreme Programming Examined,* Addison Wesley-Longman, 2001.
6. M. E. Fagan, "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, 12(7), pages 744-751, July 1986.
7. V. R. Basili, S. Green, O.Laitenburger, F. Lanubile, F. Shull, S. Sorumgard, and M. Zelkowitz, "The Empirical Investigation of Perspective-Based Reading," *Journal of Empirical Software Engineering*, 1(2), pages 133-164, 1996.
8. J. C. Schlimmer, J. B. Fletcher, and L. A. Hermens, "Team-Oriented Software Practicum," *IEEE Transactions on Education,* 37(2), pages 212-220, May 1994.
9. C. Sauer, D. R. Jeffrey, L. Land, and P. Yetton, "The Effectiveness of Software Development Technical Review: A Behaviorally Motivated Program of Research," *IEEE Transactions on Software Engineering*, 26(1), pages 1-14, Jan. 2000.
10. L. A. Williams and R. R. Kessler, "The Effects of 'Pair-Pressure' and 'Pair-Learning' on Software Engineering Education," *Proceedings of Thirteenth Conference on Software Engineering Education and Training*, pages 59-65, March 2000.
11. N. V. Flor and 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.
12. G. Salomon. *Distributed Cognitions: Psychological and Educational Considerations.* Cambridge: Cambridge Press, 1993.
13. L. L. Constantine. *Constantine on Peopleware,* Englewood Cliffs, NJ: Yourdon Press, 1995.
14. J. O. Coplien, "A Development Process Generative Pattern Language," in *Pattern Languages of Program Design*, J. O. Coplien and D. C. Schmidt, Ed. Reading Mass: Addison-Wesley, pages 183-237, 1995.
15. K. Beck. *Extreme Programming Explained: Embrace Change.* Reading, Mass: Addison-Wesley, 2000.
16. L. Williams, R. A. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair-Programming," *IEEE Software*, July/Aug. 2000.
17. J. T. Nosek, "The Case for Collaborative Programming," *Communications of the ACM*, pages 105-108, 1998.
18. A. Anderson, R. Beattie, K. Beck et al., "Chrysler Goes to Extremes," *Distributed* Computing, pages 24-28, Oct.1998.
19. L. Williams and R. R. Kessler, "Experimenting with Industry's 'Pair-Programming' Model in the Computer Science Classroom," *Journal on SW Engineering Education*, Dec. 2000.
20. L. Williams. *Pair Programming Questionnaire*. 2000. Can be found at http://collaboration.csc.ncsu.edu/questionnaire/questionnaire.htm.