But, Isn't That Cheating?

Laurie Williams
University of Utah Computer Science
50 S. Central Campus #3190
Salt Lake City, UT 84112
Phone: (801) 585-3736
Fax: (801) 581-5843
lwilliam@cs.utah.edu

"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]."

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," 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 other – watching for defects, thinking of alternatives, looking up resources, and considering strategic implications of the work. The Extreme Programming (XP) methodology, developed primarily by Smalltalk code developer and consultant Kent Beck, attributes great success to the use of "pair programming." Results [2] demonstrate that the 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.

Two other studies support the use of 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, providing anecdotal support for collaborative programming. He immediately noticed that at each terminal were two programmers working on the same code. 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."[3]

Additionally, an experiment 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. [4].

But, how about in our classrooms? Can university Computer Science 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 [3]." A class I taught at the University of Utah this summer set out to study pair programming in an educational setting.

The class, an Active Server Pages (ASP) web programming class, consisted of 20 juniors and seniors. The students were very familiar with programming, but not with the web programming languages learned and used in the class. Each student was paired with another student to work with for the entire semester. Tests were, however, taken individually. They understood that the idea was not to break the class project into two pieces and integrate later. The idea was 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. Most skeptically, but enthusiastically, embarked on making the transition from solo to collaborative programming. (Two students enrolled although that had an excess of personal commitments, making collaborative programming, which primarily needed to be done in the university computer lab, a large inconvenience to them. These students never open to the benefits of collaborative programming.) The students were surveyed at the end of their class on their view of collaborative programming:

- **84%** of the class agreed with the statement *"I enjoyed doing the assignments more because of pair programming"*
- **84%** of the class agreed with the statement *"I learned ASP faster and better because I was always working with a partner"*
- **95%** of the class agreed with the statement *"I was more confident in our assignments because we pair programmed"*
- They felt they were much more productive when working collaboratively. The main three reasons were:
  - o 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
  - o Having a constant observer watching over their shoulder served as the most efficient of all defect removal methods
  - o When one partner did not know/understand something, the other almost always did. Between the two of them, they could tackle anything.

At a commuter school, such as the University of Utah, one drawback to collaborative programming is the added logistics of getting together with their partner. One student admitted that they really preferred to write programs at home while watching TV. However, the consensus of the class was very, very positive about the technique.

Programmers, however, have generally been conditioned to performing solitary work. 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. The technique deserves further research. However, it has the potential of changing how programming classes are taught in order to benefit the students' learning experience.

Bibliography

1. Bennis, W., Biederman, Patricia Ward, *Organizing Genius: The Secrets of Creative Collaboration.* 1997: Addison-Wesley Publishing Company, Inc.
2. Beck, K., Cunningham, Ward, *Extreme Programming Roadmap*, . 1999, http://c2.com/cgi/wiki?ExtremeProgramming.
3. Constantine, L.L., *Constantine on Peopleware.* Yourdon Press Computing Series, ed. E. Yourdon. 1995, Englewood Cliffs, NJ: Yourdon Press.
4. Nosek, J.T., *The Case for Collaborative Programming*, in *Communications of the ACM.* 1998. p. 105-108.