A Pair Programming Experience

Dr. Randall W. Jensen
Software Technology Support Center

Agile software development methods, including extreme programming, have risen to the forefront of software management and development interest during the last few years. The "Agile Manifesto" published in 2001 created a new wave of interest in the agile philosophy and re-emphasized the importance of people, along with the idea of "pair programming." As defined, pair programming is two programmers working together, side by side, at one computer collaborating on the same analysis, design, implementation, and test. I was introduced to teamwork and pair programming indirectly as an undergraduate electrical engineering student in the 1950s. Later in 1975, I was asked to improve programmer productivity in a large software organization. The undergraduate experience led me to an experiment in pair programming. The very positive results of this experiment are the subject of the case study in this article.

Agile methods and extreme programming have risen to the forefront of software management and development interest during the last few years. Two definitions of *agile* are (1) able to move quickly and easily, and (2) mentally alert. Both definitions rely on the capabilities of the people within the development process.

The "Agile Manifesto" [1] published in *Software Development* in 2001 created a new wave of interest in the agile philosophy and reemphasized the importance of people. One of the points highlighted in the manifesto is, "We value individuals and interactions over processes and tools." That does not mean processes and tools are evil. It implies that individuals and interactions (people) are of higher priority than processes and tools.

Textbooks [2, 3] describe the importance of people in these new software development approaches that have demonstrated improved productivity and product quality. *Extreme programming* (XP) [4] is one member covered by the umbrella of agile methods. *Pair programming* [5] is a major practice [6] of XP. The official definition of pair programming is two programmers working together, side by side, at one computer collaborating on the same analysis, design, implementation, and test. In other words, consider it like two programmers using one pencil.

We have all experienced elements of the pair-programming concept in one way or another during our lives. How many times have you been stuck removing an error from a design or program with no success? When everything else failed, you went to your neighbor programmer, the *casual observer*, to see if you could get some assistance. While explaining the problem, you have a flash of inspiration, and the problem is quickly solved. How much time did you waste before asking a neighbor for insight? Can you relate this to pair programming?

I was introduced to pair programming indirectly as an undergraduate electrical engineering student in the 1950s. The class and laboratory workload were such that any free time during the four-year program was more wishful thinking than reality. Working part time made the program even more daunting. Fortunately, two other electrical engineering students in the same academic program were struggling with different sets of outside commitments. We decided to work together on homework assignments, lab work, and test preparation to lighten the course load.

> *"The second major benefit demonstrated in this experiment – a three order-of-magnitude improvement in error rate – is hard to ignore."*

We successfully maintained this approach through the entire program in spite of having been conditioned throughout our lives to perform solitary work. Our educational system does not condone or encourage teamwork. That education philosophy supports individual student evaluation, but works against learning. The teamwork concept became ingrained in my thinking as well as in my programming and management research activities.

Much later, I was asked to find ways to improve programmer productivity in a large software organization. The undergraduate experience led me to propose an experiment in the application of what we called *two-person programming teams*. The term pair programming had not been coined at that time. The experiment results are the subject of the remainder of this article.

Development Task

Problem

Providing a description of the results achieved through pair programming without knowledge of the project or development task underlying the experience would be meaningless. The software to be developed in this project was a multitasking real-time system executive. The product consisted of six independent components containing a total of approximately 50,000 source lines of code. The product contained no reused or commercial-off-the-shelf components. Fortran was the required software development language. The real-time executive was to be used to support the development of a large, complex software system by the developing organization. The development schedule for the executive was critical and short.

Team Composition

The development team consisted of 10 programmers with a wide range of experience and one manager. I tend to divide managers into two primary groups: Theory X[1] and Theory Y[2] [7, 8]. The manager for this task was experienced and from the Theory Y group.

The 10 programmers assigned to the executive development had prior experience that ran the gamut from an expert system programmer to a couple of fresh, young college graduates. None of these programmers had any experience working in a team environment. As a collection, I would place them as about average for that development organization.

The manager grouped the programmers into five teams according to their experience level. Each team pair was composed of the most experienced and least experienced programmer of the remaining

group. The first team consisted of the expert system programmer and a person who had just returned from a six-year leave of absence. The fifth team consisted of two programmers of near equal capability and experience. These first and fifth programming teams were important in the way they impacted the project. I will address their impacts in the Lessons Learned section of this article.

No special changes from normal were made to the development environment. The facilities were essentially two-person cubicles. The programming pairs were collocated in these cubicles. Each cubicle contained two computer workstations, two desks, and a common worktable. The pair-programming approach dictated that the pair (remember: two programmers, one pencil) use only one development terminal located on the common worktable. The second terminal was to be used for documentation, etc., not related to the team's assigned development.

One programmer of the pair functioned as the *driver* operating the keyboard and mouse, while the second programmer functioned more as a *navigator* or *co-pilot*. The navigator reviewed, in real time, the information entered by the driver. The roles of the two programmers were not permanent; frequent role changes occurred daily. The navigator was not a passive role at any time.

Results

A Priori

Project individuals could not directly obtain a productivity and error baseline for the project, but data was available from past projects that allowed them to project productivity and error averages for the project. The average productivity and error rates in most organizations with consistent management style and processes are near constant and quite predictable. The baseline productivity was determined to be approximately 77 source lines per person-month. The error rate for the development organization was normal for the aerospace industry. The numerical error rate value is not significant for this presentation, and will remain unknown.

Formal design walkthroughs and software inspections were not scheduled for this project. It would follow a classic waterfall development approach, which is inconsistent with today's agile methods. Formal preliminary and critical design reviews, as well as a final qualification test were planned. Formal review and test documentation were reduced to essential information; that is, all elements necessary to proceed with the development.

Topic	Historical	Pair Results	Gain
Productivity (lines/person-month)	77	175	127 percent
Error Rate			0.001 × normal

Table 1: *Pair Programming Productivity and Error Rate Gains*

A Posteriori

The productivity achieved in the real-time executive development was 175 source lines per person-month as shown in Table 1. We hoped for a productivity gain of anything greater than 0 percent. Any small gain would have compensated for the two programmers loading on each task. The 127 percent gain achieved was phenomenal and a cause for celebration.

The error analysis showed the project had achieved an error rate that was three orders of magnitude less than normal for the organization. Integration of the first two components (approximately 10,000 source lines) was completed with only two coding errors and one design error. The third component was integrated with no errors. The remaining three components had more errors, but the number of errors for these components was significantly less than normal.

The *continuous walkthrough* assumption was demonstrated to be very effective and more than compensated for the lack of formal walkthroughs. The formal preliminary and critical design reviews, as well as a final qualification test, were effective in keeping the five teams coordinated. Few problems were uncovered in the review and test activities.

After the experiment was completed, the development manager presented the very positive results to the organization's management staff. The project managers' reaction to the results was memorable – they claimed that their senior programmers would quit before they would team with another programmer. The use of pair programmers was never implemented in that organization.

Lessons Learned

Several positive and some negative characteristics were observed during the pair-programming experiment. In general, the attributes of the college experience were exhibited here. The positive attributes, not necessarily in any order, are as follows:

- **Brainstorming.** According to the programmers, active real-time collaboration produced higher quality designs than would have been achieved working alone. Little time was lost optimizing code with more than one brain working.
- **Continuous Design Walkthrough.** The design and code were reviewed in real time by both programmers who ultimately produced fewer errors in each team product. Classic walkthroughs and inspections are, whether we like it or not, somewhat adversarial. The continuous walkthroughs within the team were more positive and supportive.
- **Focused Energy.** The individual teams appeared to be more focused in their activities. The highly visible aspect of this attribute was that programmers took fewer breaks for restrooms, coffee, outside discussions, etc.
- **Mentor.** When we started work in this industry, we were usually told about on-the-job training that never materialized. Pair programming, when the two programmers were not of the same experience level, provided a craftsman/apprentice relationship that elevated the junior programmer's skill quickly. Conversely, the craftsman's skill is extended by the apprentice's questions and thinking outside of the box.
- **Motivation.** In general, the programming pairs appeared much more motivated than their single counterparts. The motivation level cannot be solely attributed to the pair concept or the experiment itself. Some of the motivation must be attributed to the project manager. Some must be attributed to rapid progress and the product quality. One of the Theory Y assumptions is that motivation occurs at the social, esteem, and self-actualization levels, as well as physiological and security levels.
- **Problem Isolation.** The time wasted with two pairs of eyes (or brains) was significantly less than the amount of time wasted trying to solve a problem in isolation.

Conversely, the negative observations cannot be ignored. The important observations, not necessarily in order of importance, are as follows:

- **Counter-Productivity.** Pairing programmers of the same experience and capability level is often counter-productive. The most troublesome pairs we dealt with during the experiment were two teams in which both members were near the same capability level. The worst-case team consisted of two *prima donna* programmers. The programming pair theoretically has equal responsibility for the team's efforts and product. We found teams functioned more smoothly, in spite of the members equally being

driver and navigator, if one member was slightly more capable than the other was. I read a statement by a software industry leader that stated hiring software engineers from the top 10 percentile of the top 10 universities would produce the best software development teams. I cannot imagine the stress that many egos can create on one project. Two strong egos of any caliber on a team create chaos until they recognize the power of two minds.

- **Common Area.** Coordination between the five teams would have improved if the teams had been working in a common area. Each team was located in a two-person cubicle, which limited the interaction between the teams. I use the term *war room* (or skunk works) to describe the ideal open environment, which would be a large area with worktables in the center and cubicles around the outside.

Some additional characteristics of the successful experiment are noteworthy. First, one of the manager's principle responsibilities was to buffer the teams from outside interference. The manager listed other important responsibilities that included referee (in the case of the prima donnas), arbitrator, coordinator, planner, cheerleader, and supplier of popcorn and other junk food.

Second, project managers must be supportive of the pair programming process. A classic (Theory X) manager observed a programming pair working on a design over a period of time. This manager suggested to their supervisor that one of the two programmers be laid off because only one was doing anything constructive. (The driver always gets the credit.) When the supervisor heard the suggestion, he replied that these programmers were the most productive people in the organization. The manager then asked that the programmers keep their office door closed so others would not get the same idea.

Summary

Most managers who have not experienced pair programming reject the idea without trial for one of two reasons. First, the concept appears redundant and wasteful of computing resources. Why would I want to use two programmers to do the work that one can do? How can I justify a 100 percent increase in person-hours to use this development approach? The project cannot afford to waste limited resources.

The second reason is the assumption that programmers prefer to work in isolation. Programmers, like most other people, have been trained to work alone. Yet according to the 1984 Coding War Games sponsored by the Atlantic Systems Guild, only one-third of a programmer's time is spent in isolation; two-thirds of the time is spent communicating with team members. Managers wonder about the necessary adjustments to another's work habits and programming style. They also worry about ego issues and disagreements about the product's implementation.

This experiment demonstrated strongly that programmers can work together effectively and efficiently to produce a quality product of which both programmers can be proud. Prior programming experience is not an issue. There are initial situations, especially with a team of equal experience and ego, where disagreements arise over who will be the driver. Those situations are generally transient. The benefits listed in the results section overwhelmed any personality issues that arose.

The second major benefit demonstrated in this experiment – a three order-of-magnitude improvement in error rate – is hard to ignore. Repairing defects after developments is much more expensive than uncovering and fixing the defects where they occur. The benefits of developing and delivering a stable product faster, reducing maintenance costs, and gaining customer satisfaction certainly minimize the risk of using pair-programming teams.◆

References

1. The Agile Alliance. "The Agile Manifesto." Software Development 9.8 (Aug. 2001).
2. DeMarco, Tom, and T. Lister. Peopleware. New York: Dorset House Publishers, 1977.
3. Weinberg, G. M. The Psychology of Computer Programming Silver Anniversary Edition. New York: Dorset House Publishers, 1998.
4. Beck, Kent. Extreme Programming Explained: Embracing Change. Reading, MA: Addison-Wesley, 2000.
5. Williams, L., R. R. Kessler, W. Cunningham, and R. Jeffries. "Strengthening the Case for Pair Programming." IEEE Software 17.4 (July/Aug. 2000): 19-25.
6. Beck, Kent. "Embracing Change with Extreme Programming." Computer Oct. 1999: 71.
7. Hersey, P., and K. H. Blanchard. Management of Organizational Behavior, Utilizing Human Resources. Englewood Cliffs, NJ: Prentice-Hall, 1977.
8. McGregor, D. The Human Side of Enterprise. New York: McGraw-Hill, 1960.

Note

1. Theory X assumes the following: (1) Work is inherently distasteful to most people. (2) Most people are not ambitious, have little desire for responsibility, and prefer to be directed. (3) Most people have little capacity for creativity in solving organizational problems. (4) Most people must be closely controlled and often coerced to achieve organizational objectives.
2. Theory Y assumes the following: (1) Work is as natural as play, if conditions are favorable. (2) Self-control is often indispensable in achieving organization goals. (3) The capacity for creativity in solving organizational problems is widely distributed in the population. (4) People can be self-directed and creative at work if properly motivated.

About the Author

Randall W. Jensen, Ph.D., is a consultant for the Software Technology Support Center, Hill Air Force Base, with more than 40 years of practical experience as a computer professional in hardware and software development. He developed the model that underlies the Sage and the GAI SEER-SEM software cost and schedule estimating systems. Jensen received the International Society of Parametric Analysts Freiman Award for Outstanding Contributions to Parametric Estimating in 1984. He has published several computer-related texts, including "Software Engineering," and numerous software and hardware analysis papers. He is currently preparing "Extreme Software Estimating" for Prentice-Hall, Inc. Dr. Jensen has a bachelor's of science degree in electrical engineering, a master's of science degree in electrical engineering, and a doctorate in electrical engineering from Utah State University.

Software Technology Support Center
7278 4th St.
Bldg. 100 G58
Hill AFB, UT 84056
Phone: (801) 775-5733
Fax: (801) 777-8069
E-mail: randall.jensen@hill.af.mil