Pair Programming

Laurie Williams

North Carolina State University

Keywords –pair programming, collaborative programming, software inspection

Abstract
Pair programming is a style of programming in which *two* programmers work side-by-side at *one* computer, continuously collaborating on the same design, algorithm, code, or test. In industry, the practice of pair programming has been shown to improve product quality, improve team spirit, aid in knowledge management, and reduce product risk. In education, pair programming also improves student morale, helps students to be more successful, and improves student retention in an information technology major. This chapter provides an overview and history of pair programming followed by a summary of the use of pair programming in industry and academia. The chapter also provides insight into the principles that make pair programming successful, the economics of pair programming, and the challenges in the adoption of pair programming.

1. INTRODUCTION

> *The human eye has an almost infinite capacity for not seeing what it does not want to see.... Programmers, if left to their own devices, will ignore the most glaring errors in their output—errors that anyone else can see in an instant.* [60]

Pair programming is a style of programming in which *two* programmers work side-by-side at *one* computer, continuously collaborating on the same design, algorithm, code, or test [61]. Pair programming has been practiced sporadically for decades [61]; however, the emergence of agile methodologies and Extreme Programming (XP) [4] has recently popularized the pair programming practice.

With pair programing, one of the pair, called the *driver*, types at the computer or writes down a design. The other partner, called the *navigator*, has many jobs. One of these is to observe the work of the driver—looking for tactical and strategic defects in the driver's work. Some tactical defects might be syntax errors, typos, and calling the wrong method. Strategic defects occur when the driver's implementation or design will fail ultimately to accomplish its goals. The navigator is the strategic, long-range thinker of the programming pair. Because the navigator is not as deeply involved with the design, algorithm, code or test, he or she can have a more objective point of view and can better think strategically about the direction of the work. Both in the pair are constant brainstorming partners. An effective pair will be constantly discussing alternative approaches and solutions to the problem [57, 61]. A sign of a dysfunctional pair is a quiet navigator. Periodically, the driver and the navigator should switch roles. On a software

development team, team members should pair program with a variety of other team members to leverage a variety of expertise.

The name of the technique, pair *programming* can lead people to incorrectly assume that software engineers only pair during code development. However, pairing can occur during all phases of the development process, in pair design, pair debugging, pair testing, and so on. Programmers could pair up at any time during development, in particular when they are working on a complex or unfamiliar problem. Pair programming is also an effective way to distribute domain and system knowledge throughout the team.

The rest of this article proceeds as follows. In Section 2, we provide a brief history of the use of pair programming. In Sections 3 and 5, we provide information on the use of pair programming in industry and in academia, respectively. In Section 4, we provide an overview of the economics of pair programming. In Section 6, we discuss the use of distributed pair programming, the use of pair programming when the programmers are not co-located. In Section 7, we propose some conjectures about the principles that make pair programming successful. We present the challenges in transitioning to pair programming in Sections 8. We conclude with a summary in Section 9.

2. A HISTORY OF THE PAIR PROGRAMMING PRACTICE

People have advocated and practiced pair programming for decades, long before it was ever called pair programming. Fred Brooks, author of *The Mythical Man Month* [8], has communicated : "Fellow graduate student Bill Wright and I first tried pair programming when I was a grad student (1953-1956). We produced 1500 lines of defect-free code; it ran correctly first try." [61] In the early 1980s, Larry Constantine, author of more than 150 technical articles and 16 books, reported observing "Dynamic Duos" at Whitesmiths, Ltd., producing code faster and more bug-free than ever before [14]. He commented that the code benefited from the thinking of two bright minds and the steady dialog between two trusted programmers. He concluded that two programmers in tandem was not redundancy, but rather it was a direct route to greater efficiency and better quality. Based upon research findings of the Pasteur project (a large sociological/anthropological study of 50 highly effective software development organizations) at Bell Labs Research, James Coplien published the "Developing in Pairs" Organizational Pattern [15, 16] in 1995. Coplien identified the forces of this pattern as "people sometimes feel they can solve a problem only if they have help. Some problems are bigger than any one individual," And the solution is to, "pair compatible designers to work together; together they can produce more than the sum of the two individually." The result of applying the pattern is, "a more effective implementation process. A pair of people is less likely to be blindsided than an individual developer."

In 1998, Temple University professor John Nosek was the first to run an empirical study on the efficacy of pair programmers [46]. Nosek reported the results of 15 full-time, experienced programmers working for a maximum of 45 minutes on a challenging problem important to their organization. In their own environments and with their own equipment, five worked individually and 10 worked collaboratively in five pairs. The conditions and materials were the same for both the experimental (team) and control (individual) groups. A two-sided t-test showed that the study provided statistically significant results. Combining their time, the pairs spent 60% more minutes

on the task. Because they worked in tandem, however, they completed the task 20% faster than the control groups, and produced better algorithms and code.

As mentioned above, the emergence of the XP software development methodology in the late 1990s/early 2000s brought the pair programming practice to the forefront. The XP methodology was initially defined [5] as consisting of 12 software development practices; pair programming was one of these. Five years of experience led to a re-definition of XP [4] in which practices were divided 13 primary and 11 corollary practices. Beck [4] explains that the primary practices can be thought of as best practices that are useful independent of the software develeopment methodology being used. The primary practices work independently, but work better together. The corollary practices are difficult or dangerous to implement before implementing a core set of primary practices. Some of the initial 12 XP practices, such as On-site Customer, were moved to corollary practices. Pair programming remained as a primary practice.

As XP emerged, many people were doubtful about pair programming because they believed that the amount of effort spent on a programming task would double if two programmers worked together. This incredulousness motivated an extensive empirical study that was run at the University of Utah in 1999 [62, 68, 69] to isolate and study the costs and benefits of pair programming. Forty-one third- and fourth-year undergraduate students in a Software Engineering class participated in a structured experiment for the duration of a 15-week semester. On the first day of class, the students were asked if they preferred to work in pairs 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 the grade point average (GPA) in their academic records. Twenty-eight students were assigned to the paired group and 13 to the solo group. The GPA was used to ensure the groups were academically equivalent. Of the 14 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.

All students received instruction in effective pair-programming and were given a paper [69] on strategies for successful collaboration to help prepare the students. 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. During these regular meeting times, the pairs jelled or bonded and were much more likely to establish additional meeting times to complete their work.

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 at $p < .01$.

	% test cases pass by solo students	% of test cases passed by paired students
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

The results also indicated that on average the pairs spent 15% more programmer-hours than the individuals to complete their projects. For example, if one individual spent 10 hours on an assignment, each partner would spend slightly more than 5 hours. The median time spent was essentially equal for the two groups and the average difference was not statistically significant.

Between the use of the practice in Extreme Programming and the results of the University of Utah study that indicated the benefits of pair programming without doubling resources, pair programming began to be used more prevalently in industry and education. The next two sections outline the use of pair programming in each of these environments.

3. PAIR PROGRAMMING IN AN INDUSTRIAL SETTING

This section outlines some of the pair programming practices that have been demonstrated in industrial organizations and the results cited for these practices.

3.1 Industry Practices in Pair Programming

Industrial teams have reported their experiences with sustained use of pair programming. Practitioners can often can be hesitant to embark on the use of pair programming (see Section 8 for a discussion of this topic). Developers often require several days to become familiar and comfortable with the dynamics and practices of pair programming when transitioning from solo programming [57]. Often programmers do not work in pairs for the full work day; a suitable length of time for pairs to work together is between 1.5 and 4 hours [58]. Extended pair programming sessions can be difficult for developers because pairing can be mentally exhausting [58, 61] due to the rapid pace a pair can work at and the constant focus on the task at hand.

In industry teams, pair rotation is a common practice to keep the pairs dynamic, rather than having assigned pairing partners for days or weeks at a time. Many teams rotate pairs several times per day [6] or once per day [22]. Frequent pair rotation is important for knowledge transfer among the team [57]. Additionally, pair rotations aids in indoctrinating and training new team members as empirically demonstrated at Menlo Innovations, Microsoft, Motorola, and Silver Platter Software [6, 33, 66]. Deciding who to pair with is usually done casually and without difficulty [11, 57], often done in a short daily meeting [57].

Rotating the roles of driver and navigator is considered important for keeping both software engineers engaged [57, 61]. However, in a four-month ethnographic study [11], Chong and Hurlbutt did not observe driver/navigator roles in the pairs on two professional software development teams practicing pair programming. Rather, when the programmers had equivalent expertise, they engaged jointly in discussion and brainstorm. In this case, the driver mainly served the role of typist. When the programmers had different levels of expertise, the software engineer with more expertise dominated the interaction. Chong and Hurlbutt also noted that possession of the keyboard had a subtle but consistent effect on decision making—the driver as the final decision maker. They also note that the keyboard is often passed back and forth pervasively, and that engineers were most effective when they jointly took on the roles of driver and navigator. Engineers seemed more engaged when they were in posession of the keyboard or perceived keyboard control was imminent. As a result, they advocate the use of dual keyboards

and mice. Analyses performed by Höfer [26] and Freudenberg et al. [21] support the findings of Chong and Hurlbutt.

Development teams at IBM and Guidant were given the choice of using pair programming or inspections/reviews, which increased the use of pair programming among the teams from from 5% to 50% of the time [63] at IBM to essentially all the time [49] at Guidant.

Often software engineers feel pair programming should be saved for specification, design, and more complex programming tasks [17, 27, 36, 57-59]. One large expertiment of 295 consultants demonstrate a quality improvement with pairs on complex tasks but not quality improvement on simpler tasks [1].

Teams have been more successful when having a structured, organized approach to pair programming, such as proclaimed pair programming hours or having pairs assigned daily in a meeting, rather than having its use voluntary and unstructured. In one organization [59], a survey indicated positive feelings toward pair programming and a desire to do more pair prgramming. However, these engineers cited the reason for not pair programming more as difficulties in organization, such as in finding common time, lack of encouragement from team leaders, and not considering pair programming in project planning. Finally, pair programmers have found it beneficial to have larger desks, larger screens, wireless mice and keyboards, and whiteboards on the walls [59].

3.2 Results of Using Pair Programming in Industry

The teams who have reported sustained use of pair programming have provided insight into the results of the use of the practice. Many teams report improved product quality [17, 30, 36, 57] when using pair programming. Specifically, one large telecommunicaitons company in Finland whose software engineers almost exclusively worked in pairs had only five field failures in one and a half years of production [57]. A different controlled case study in Finland [27] demonstrated equal defect density for paired/solo developers for one project and a six-fold reduction for paired developers on another project.

Teams are motivated to institute pair programming because of the positive effect of improved knowledge sharing [36] including that of the development environment [57, 58]. If only one person understands an area of code, the team can suffer if this person leaves the team or is unavailable due to sickness or vacation. When more than one person is at least casually familiar with each area of the code, risk is reduced for the team.

Teams find code written by a pair to be more understandable [57, 58]. Code is written by a driver to be understandable by the navigator, motivating the driver to be more clear. Alternatively, if the driver does not understand the code, the navigator will stop the driver to ask for clarification which prompts the driver to re-write the code more simply.

An initial decline in productivity is sometimes observed when engineers start to pair program [1, 46]. With sustained use on one development team, pair programming was considered to have a positive effect on productivity when the software engineers worked on complex tasks [57]. They

surmise that other studies that showed a productivity decline [45, 46, 68] were because these studies were on individual tasks or small projects. Another team perceived they had a small decline in productivity [58], as has been observed by earlier studies [45, 46, 68]. A controlled case study of four teams in Finland [27] did not demonstrate any patterns of higher or lower productivity by solo or pair programmers.

Teams report the use of pair programming contributed to their team morale [57, 58]. They also report that its use increases discipline in the use of other prescribed practices, such as test driven development, the use of coding standards, and frequent integration [57, 58].

A structured experiment of eight professional software developers in Thailand was conducted to compare the use of Fagan inspections [19] with pair programming. Product development took 4% more time for the pairs, but the pairs produced a product with 39% fewer defects in user acceptance testing [50].

However, noise from a pair can disturb others who work alone [59]. A room or area set aside for pair programming can help mitigate this problem [59].

4. ECONOMIC STUDIES OF PAIR PROGRAMMING

As discussed in Section 3, studies have found that pair programming can take a bit more total effort but produce a higher quality product. Because the two in a pair work in tandem, the cycle time (or elapsed time) for completing a task is shortened almost in half. The tradeoff between increased speed and quality vs. increased resources has been examined in two economic models [18, 48].

The model developed by Erdogmus [18] is a net present value model to examine the economic feasibility of pair programming. Central to his model is the consideration that value lies in the ability of a pairs to deliver and get paid for a working product, even a partial working product, to customers more rapidly. Additionally, Erdogmus makes the assumptions that each defect that escapes to field costs an organzation 33 person-hours (based upon [28]) and that pairs take 15% more person-hours to complete a function (based upon [68]). He concludes that pair programming is an economically-viable alternative to solo programming.

Padberg and Müller [48] also created a net present value eonomic model. In addition to the factors considered by Erdogmus, these authors also consider market pressure. They also use the assumption that each defect that escapes to field costs an organzation 5-20 person-hours and use a figure of 10 person hours (based upon [28, 29]). They consider a range for what they term pair speed advantage, considering more conservative results of Nosek [46] and more aggressive results of Williams [68]. They conclude that when market pressure is strong adding developers to form pairs can speed up the project and increase its business value despite increased personnel cost.

5. PAIR PROGRAMMING IN AN EDUCATIONAL SETTING

For the most part, students do pair programming in much the same way as do practitioners. In

this section, we discuss some pair programming practices specifically used with students and cite results of using pair programming in educational literature.

5.1 Practices Specific to Education

When using pair programming in an educational setting, the teaching staff determines how to form effective pairs. The teaching staff may allow the students to choose their partners, or the teaching staff may proactively form student pairs that are most likely to work well together. One study indicates that heterogeneous pairs formed of a male and a female had high quality and more creative solutions [41]. A study of 58 undergraduates indicates pairs work best if they rate themselves similarly when asked about their open mindedness and level of responsibility [10]. A large empirical study of more than 1350 students was conducted to examine factors that the teaching staff can use to proactively form pairs that are most likely to be compatible [64]. The study indicated that most often (93% of pairs) students report being compatible with their partners. The results also indicated that the teaching staff can use the following information to form highly-compatible pairs:

- Pair students together who have a similar skill level as measured by computer science and/or total grade point average (qualitative results by Toll et al. [56] also support having students of similar skill level);
- Pair a Myers-Briggs sensor with a Myers-Briggs intuitor; and/or
- Pair students together who have similar work ethic, determined by asking students to provide a number from 1 to 9 where a 1 indicates the student works hard enough to just barely get by and a 9 indicates they work hard enough to get the best possible grade.

Pair rotation is done less frequently than in industry. Most often pairs stay together for the duration of an assignment (generally one to three weeks) [54]. Some educators prefer for pairs to remain consistent for a whole semester [37].

5.2 Results of Using Pair Programming in Education

Studies have shown that pair programming creates an environment conducive to more advanced, active learning and social interaction, leading to students being less frustrated, more confident, and more interested in IT [7, 34, 35, 42, 53]. The benefits to pair programming contrast with the negative aspects of traditional solo programming pedagogies, which can leave students feeling isolated, frustrated, and unsure of their abilities. Pair programming encourages students to interact with peers in their classes and laboratories, thereby creating a more communal and supportive environment. Students of the current Millennial generation place particular value on collaborative environments [47]. Furthermore, the collaboration inherent in pair programming exposes and reinforces students to the collaboration, teamwork, and communication skills required in industry. These benefits appear to help increase retention in computer science, particularly among women [9, 38, 39, 65]. In general, pair programming provides a way to mirror the “laboratory model” that is common practice in the natural sciences such as chemistry or physics.

Students who work in pairs tend to produce projects of higher quality and have higher course passing rates [39, 40, 65, 70] even when students pair program in a distributed manner (see Section 6) [24]. A study of undergraduate students at Pace University found a positive

correlation between out-of-class collaboration and student achievement based on student projects and examination grades [31].

Pair programming also benefits the teaching staff. Less grading is required due to half the number of assignment submissions. A pair of students can oftentimes figure out the low-level technical or procedural questions that typically burden the teaching assistants in the laboratory [23, 67], instructor's office hours and email inbox. Finally, there are fewer "problem students" to deal with because the peer pressure involved in pair programming encourages all students to be active participants in the class. Students become concerned about jeopardizing their partner's grade and work harder on assignments, often getting started earlier than if they worked alone (though not all students report starting earlier [52]).

Alas, there are some costs to implementing pair programming. For students, there are two major costs that persist without apparent recourse. A small segment of students (approximately 5%) will always desire to work alone. Most often, these are the top students who do not want to be "slowed down" by another student and who do not see benefit in teaching others. Another problem for students is the need to coordinate schedules when pair programming is required outside of a classroom or laboratory setting.

6. DISTRIBUTED PAIR PROGRAMMING

Distributed software development is becoming common practice in industry. In education, students may also prefer to work from their dorm rooms or homes, rather than going to the lab to work with their partners. Furthermore, students enrolled in distance education courses may not ever be able to meet each other face-to-face. These distributed workers can practice pair programming through the Internet using a variety of tools. In the simplest of cases, programmers can use VNC[1] or Windows Meeting Space[2] (previously Net Meeting) to share desktops. These tools broadcast the display of the output of any application from a member to all the others, requiring sufficient bandwidth, trust, and security between the parties. Other tools, such as Sangam [25], xpairtise[3], COPPER [43], or Facetop [44] have been designed to only transmits messages that are important for pair programming, such as the latest change made by the driver.

Distributed cognition expert Nick Flor stresses the importance of distributed pair programming systems to support cross-workspace visual, manual, and audio channels [20]. These channels allow pairs to collaborate and provide subtle, yet significant catalysts for on-going knowledge sharing and helping activities. For example, subtle gestures such as a headshake or a mumble can be the catalyst for an exchange between the pair. Transparent images of the partner shown in the screen by Facetop [44] can aid in the transmission of these channels. Additionally, Chong and Hurlbutt [11] discourage tools that have defined driver/navigator roles such as Sangam [25] because they inhibit the behaviors of more effective pair programmers who share the driver/navigator role throughout the session.

Some studies of distributed pair programming have been done with students at both North

[1] http://www.realvnc.com/
[2] http://www.microsoft.com/windows/products/windowsvista/features/details/meetingspace.mspx
[3] http://xpairtise.sourceforge.net/

Carolina State University and the University of North Carolina -- Chapel Hill [2, 3]. These studies indicated that pairing over the Internet shows a great deal of potential when compared with distributed non-paired teams in which programmers work alone, and code is integrated later. In these studies, the students used desktop sharing software, NetMeeting, and Yahoo Messenger/headsets/microphones to communicate.

7. PRINCIPLES OF PAIR PROGRAMMING

Studies have shown that pairing makes programmers work differently. The benefits of pair programming come from six different behaviors that work synergistically.

7.1. Pair Pressure

Pair programmers put a positive form of pressure on each other that functions as a time management strategy. Software engineers say they work harder and smarter on programs because they do not want to let their partner down. They are also less likely to read email, surf the web, or make a phone call. They handle interruptions more quickly so they can return to their primary task they share with their partner [12]. Engineers often pair for a few hours at a time during which they work intensely on their joint task without interruption. As such, the pair can work with a "pair flow" [6] state of mind in which the solution and the problem space are shared between the minds of the participants. The presence of a pairing parter helps an engineer recover the state of a primary task after interruption leading to more rapid interruption recovery [12]. Additionally, solo programmers can use interruptions as means for filling a need for social interaction; this need dimishes with pair programming [12].

Programmers say they work very intensively because they are highly motivated to complete the task at hand during the session. Pairing requires schedule coordination, which imposes explicit deadlines that motivate engineers to work intensively finish their tasks.

7.2. Pair Negotiation and Brainstorming

The term pair negotiation is used to describe how two pair programmers arrive at the best solution together. When pairing is working at its best, each person brings to the partnership his or her own set of skills, abilities, and outlooks and both partners share the same goal for completing the task. Each person has a suggested alternative for attacking a joint problem, and the partners must negotiate how to jointly approach the problem. In this negotiation, they evaluate more alternatives than either one would have considered alone, whereas, a person working alone tends to pursue the first approach that comes to mind. Together, the partners consider and include each other's suggestions and determine the best plan of attack.

Couched in effective pair programming is the phenomenon known as Beginner's Mind [6] wherein a person that is new to an area with no predisposition of a solution can see more possible solutions.

7.3. Pair Courage

Having a partner is a tremendous courage builder. Gaining affirmation from a partner gives programmers the confidence to do things they might be afraid to do alone When working with

someone else, programmers can piece together enough knowledge to feel confident in what they are doing.

Working with a partner also gives us courage to admit when we do not know something. Developers by themselves tend to be embarrassed when they do not know something and will try to muddle through on their own rather than ask for help from their peers. When two people do not know something, there is a joint realization that it is time to seek help.

7.4. Pair Reviews

Pair programming functions as a form of continuous review and problem identification occurs on a minute-by-minute basis. Syntax or semantic errors and missing assumptions or unconsidered cases in algorithm design that may otherwise go unnoticed can often be observed by an attentive navigator before these problems gestate. This low-level review process complements that pair's strategic brainstorming by avoiding the small, subtle errors that a solo programmer may unknowingly inject and spend considerable time later trying to uncover and fix.

7.5. Pair Debugging

Every person has experienced problems that can be resolved simply through the act of explaining the problems to another person.

> *. . . [an] effective technique is to explain your code to someone else. This will often cause you to explain the bug to yourself. Sometimes it takes no more than a few sentences, followed by an embarrassed "Never mind; I see what's wrong. Sorry to bother you." This works remarkably well; you can even use nonprogrammers as listeners. One university computer center kept a teddy bear near the help desk. Students with mysterious bugs were required to explain them to the teddy bear before they could speak to a human counselor.* [32]

When explaining a problem to a partner, the partner will ask questions and will likely force the programmer to explain his or her potentially-flawed reasoning.

7.6. Pair Learning and Training

Knowledge is constantly being passed between partners, from tool usage tips to programming language rules to design and programming techniques. The partners take turns being the teacher and the student on a minute-by-minute basis. Even unspoken skills and habits cross partners [13]. When pairs rotate to work with different team members, each programmer is then able to share new skills and knowledge with a new partner. As a result, switching pairs often is an effective strategy for spreading knowledge and information around a team [6]. As stated above, pair programming with frequent swapping also aids in indoctrinating and training new team members [6, 33, 66].

8. CHALLENGES

The results above indicate that teams benefit in range of ways when using the pair programming

practice. However, several challenges prevent widespread adoption of the pair programming practice. Four of these challenges are now presented. We also suggest how these challenges can be overcome.

- **Bias/Habit.** Software engineers are conditioned to work alone. As a result, many can be concerned that they will not be able to concentrate when working with others, they may be "wasting their time" with slower programmers, they will feel inadequate compared to their peers, and other concerns. However, of those who try the practice, more than 90% prefer pair programming over solo programming [55, 62]. Because most engineers eventually learn to like the practice, a strategy for overcoming this challenge is to institute pair programming as a non-threatening pilot with a small number of engineers. These engineers are very likely to find the practice preferable and beneficial and will spread the word to their initially-resistant teammates.
- **Economics.** Software organization can be concerned that pair programming will double software development cost as two programmers are working on one task. The results discussed above indicate that pair programming is an economically-beneficial practice. The management should be enlightened with the research and experiential results of the use of pair programming. Again, the practice can be started on a small scale which would cause only a minimal economic risk. The management can gain first-hand understanding that the practice does not cause product lifecycle cost to increase, particularly when the benefits of improved quality are considered.
- **Coordination.** When engineers on a team practice pair programming, the team must decide who works with whom each day. Additionally, the team may need to choose which hours of the day are considered pair programming hours during which no meetings would take place. An ideal time to determine the composition of the pairs is during a daily 10-15 minute Scrum [51] meeting. During the Scrum meeting, engineers talk about their obstacles and the tasks that they will do each day. During this meeting, pairs can be dynamically assigned based upon the tasks of the day.
- **Distributed teams.** Some teams may feel they cannot pair program because their team members are not all physically co-located. As discussed in Section 6, distributed pair programming has been shown to be a viable and beneficial practice.

9. SUMMARY

Industrial teams have realized the following benefits from pair programming.

- Higher product quality
- Can be used as a substitute for code review
- Improved cycle time
- Enhanced learning
- Ease staff training and transition
- Knowledge management
- Reduced product risk
- Enhanced team spirit

Students realize these same benefits, but research results have also shown additional benefits:

- Retention in an information technology field of study
- Reduced frustration in completing course work
- Satisfaction with getting to know their classmates

10. ACKNOWLEDGEMENTS

Some of this material is based upon the work supported by the National Science Foundation under Grants ITWF 00305917 and BPC 0540523. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Thanks for Lucas Layman for reviewing this article.

REFERENCES

[1] E. Arisholm, H. Gallis, T. Dybå, and D. Sjøberg, "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise," *IEEE Transacations in Software Engineering*, vol. 33, no. 2, pp. 65-86, February 2007.

[2] P. Baheti, E. Gehringer, and D. Stotts, "Exploring the Efficacy of Distributed Pair Programming," in *Extreme Programming/Agile Universe*, Chicago, IL, 2002, pp. 208-220.

[3] P. Baheti, L. Williams, E. Gehringer, and D. Stotts, "Exploring Pair Programming in Distributed Object-Oriented Team Projects," in *OOPSLA Educator's Syposium*, Seattle, WA, 2002.

[4] K. Beck, *Extreme Programming Explained: Embrace Change*, Second ed. Reading, MA: Addison-Wesley, 2005.

[5] K. Beck, *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley, 2000.

[6] A. Belshee, "Promiscuous pairing and beginner's mind: embrace inexperience," in *Agile Conference 2005*, Denver, CO, 2005, pp. 125 - 131

[7] S. B. Berenson, L. Williams, and K. M. Slaten, "Using Pair Programming and Agile Development Methods in a University Software Engineering Course to Develop a Model of Social Interactions," in *Crossing Cultures, Changing Lives Conference*, Oxford, UK, 2005, p. to appear.

[8] F. P. Brooks, *Mythical Man Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1975.

[9] J. Carver, L. Henderson, L. He, J. Hodges, and D. Reese, "Increased Retention of Early Computer Science and Software Engineering Students Using Pair Programming," in *Conference on Software Engineering Education and Training*, Dublin, Ireland, 2007, pp. 115 - 122

[10] J. Chao and G. Atli, " Critical personality traits in successful pair programming," in *Agile 2006*, Minneapolis, MN, 2006, p. electronic proceedings.

[11] J. Chong and T. Hurlbutt, "The Social Dynamics of Pair Programming," in *International Conference on Software Engineering (ICSE) 2007*, Minneapolis, MN, 2007, pp. 354-363

[12] J. Chong and R. Siino, "Interruptions on Software Teams: A Comparison of Paired and Solo Programmers," in *Computer Supported Collaborative Work (CSCW) 2006*, Banff, Alberta, Canada, 2006, pp. 29-38.

[13] A. Cockburn and L. Williams, "The Costs and Benefits of Pair Programming," in *Extreme Programming and Flexible Processes in Software Engineering (XP2000)*, Cagliari, Sardinia, Italy, 2000.

[14] L. L. Constantine, *Constantine on Peopleware*. Englewood Cliffs, NJ: Yourdon Press, 1995.
[15] J. O. Coplien, "A Development Process Generative Pattern Language," in *Pattern Languages of Program Design*, James O. Coplien and Douglas C. Schmidt, Ed. Reading, MA: Addison-Wesley, 1995, pp. 183-237.
[16] J. O. Coplien and N. B. Harrison, *Organizational Patterns of Agile Software Development* Upper Saddle River, NJ: Pearson Prentice Hall, 2005.
[17] T. Dybå, E. Arisholm, D. Sjøberg, J. Hannay, and F. Shull, "Are Two Heads Better Than One? On the Effectiveness of Pair Programming," *IEEE Software*, vol. 24, no. 6, pp. 12-15, November/December 2007.
[18] H. Erdogmus and L. Williams, "The Economics of Software Development by Pair Programmers," *The Engineering Economist*, vol. 48, no. 4, pp. 283-319, 2003.
[19] M. E. Fagan, "Advances in Software Inspection," *IEEE Transactions on Software Engineering*, vol. 12, no. 7, pp. 744-751, July 1986.
[20] N. Flor, "Globally Distributed Software Development and Pair Programming," *Communications of the ACM*, vol. 49, no. 10, pp. 57-58, October 2006.
[21] S. Freudenberg, P. Romero, and B. du Boulay, "'Talking the talk': Is Intermediate-level conversation the key to the pair programming success story?," in *Agile 2007*, Washington, DC, 2007, pp. 84-91.
[22] T. Frever and P. Ingalls, "The pairing session as the atomic unit of work," in *Agile Conference*, Minneapolis, 2006, p. electronic proceedings.
[23] B. Hanks, "Problems Encountered by Novice Pair Programmers," in *International Computing Education Research Workshop*, Atlanta, GA, 2007, pp. 159 - 164.
[24] B. Hanks, "Student Performance in CS1 with Distributed Pair Programming," in *Innovation and Technology in Computer Science Education (ITiCSE) 2005*, Monte de Caparica, Portugal, 2005, pp. 316-320.
[25] C.-w. Ho, S. Raha, E. Gehringer, and L. Williams, "Sangam: A Distributed Pair Programming Plug-in for Eclipse," in *Eclipse Technology Exchange at Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2004.*, Vancouver, BC, 2004.
[26] A. Höfer, "Video Analysis of Pair Programming," in *Workshop on Scrutinizing Agile Practices at the International Conference on Software Engineering*, Leipzig, Germany, 2008, pp. 37-41.
[27] H. Hulkko and P. Abrahamsson, "A Multiple Case Study on the Impact of Pair Programming on Product Quality," in *International Conference on Software Engineering (ICSE) 2005*, St. Louis, Missouri, USA, 2005, pp. 495-504.
[28] W. S. Humphrey, *A Discipline for Software Engineering*. Reading, MA: Addison Wesley, 1995.
[29] W. S. Humphrey, *Managing the Software Process*. Reading, Massachusetts: Addison-Wesley, 1989.
[30] R. Jensen, "A Pair Programming Experience," in *Crosstalk*, March 2003.
[31] A. Joseph and M. Payne, "Group Dynamics and Collaborative Group Performance," in *Thirty-fourth SIGCSE Technical Symposium on Computer Science Education*, Reno, NV, March 2003, pp. 368-371.
[32] B. W. Kernighan and R. Pike, *The Practice of Programming*. Reading, Massachusetts: Addison-Wesley, 1999.

[33] M. Lacey, "Adventures in Promiscuous Pairing: Seeking Beginner's Mind," in *Agile 2006*, Minneapolis, 2006, pp. 263 - 269.
[34] L. Layman, "Changing Students' Perceptions: An Analysis of the Supplementary Benefits of Collaborative Software Development," in *19th Conference on Software Engineering Education and Training (CSEE&T '06)*, Turtle Bay, Hawaii, 2006, pp. 156-166.
[35] L. Layman, L. Williams, J. Osborne, S. Berenson, K. Slaten, and M. Vouk, "How and Why Collaborative Software Development Impacts the Software Engineering Course," in *Frontiers in Education*, Indianapolis, IN, 2005, pp. T4C 9-14.
[36] G. Luck, "Subclassing XP: breaking its rules the right way," in *Agile Development Conference*, Salt Lake City, UT, 2004, pp. 114 - 119
[37] C. McDowell, L. Werner, H. Bullock, and J. Fernald, "The Effect of Pair Programming on Performance in an Introductory Programming Course," in *ACM Special Interest Group of Computer Science Educators*, Covington, KY, 2002, pp. 38-42.
[38] C. McDowell, L. Werner, H. Bullock, and J. Fernald, "The Impact of Pair Programming on Student Performance, Perception and Persistence," in *International Conference on Software Engineering 2003*, Portland, Oregon, 2003, pp. 602 - 607
[39] C. McDowell, L. Werner, H. Bullock, and J. Fernald, "Pair Programming Improves Stuent Retention, Confidence, and Program Quality," *Communications of the ACM*, vol. 49, no. 8, pp. 90-95, August 2006.
[40] E. Mendes, L. Al-Fakhri, and A. Luxton-Reilly, "A Replicated Experiment of Pair Programming in a 2nd year Software Development and Design Computer Science Course," in *Innovation and Technology in Computer Science Education (ITiCSE) 2006*, Bologna, Italy, 2006, pp. 108-112.
[41] M. Mujeeb-u-Rehman, X. Yang, J. Dong, and M. Abdul Ghafoor, "Heterogeneous and homogenous pairs in pair programming: an empirical analysis," in *Canadian Conference on Electrical and Computer Engineering 2005*, Saskatchewan, Canada, 2005, pp. 1116 - 1119
[42] N. Nagappan, L. Williams, M. Ferzli, K. Yang, E. Wiebe, C. Miller, and S. Balik, "Improving the CS1 Experience with Pair Programming," in *ACM Special Interest Group Computer Science Education (SIGCSE) 2003*, Reno, 2003, pp. 359 - 362.
[43] H. Natsu, J. Favela, A. Morán, D. Decouchant, and A. Martinez-Enriquez, "Distributed Pair Programming on the Web," in *Mexican International Conference on Computer Science (ENC) 2003*, Ciencias de la Computacion, CICESE, Mexico, 2003, pp. 81-88.
[44] K. Navoraphan, E. F. Gehringer, J. Culp, K. Gyllstrom, and D. Stotts, "Next-generation DPP with Sangam and Facetop," in *OOPSLA workshop on eclipse technology eXchange*, Portland, Oregon, 2006, pp. 6-10.
[45] J. Nawrocki and A. Wojciechowski, "Experimental Evaluation of Pair Programming," in *European Software Control and Metrics (ESCOM 2001)*, London, England, 2001.
[46] J. T. Nosek, "The Case for Collaborative Programming," *Communications of the ACM*, vol. 41, no. 3, pp. 105-108, 1998.
[47] D. Oblinger, "Boomers, Gen-Xers, and Millennials: Understanding the New Students," *Educause Review*, vol. 38, no. 4, pp. 37-47, July/August 2003.
[48] F. Padberg and M. Müller, "Analyzing the Cost and Benefit of Pair Programming," in *International Software Metrics Symposium (METRICS) 2003*, Sydney, Australia, 2003, pp. 166 - 177

[49] A. Pandey, C. Miklos, M. Paul, N. Kameli, F. Boudigou, V. Vijay, A. Eapen, I. Sutedjo, and W. Mcdermott, "Application of tightly coupled engineering team for development of test automation software - a real world experience," in *Computer Software and Applications Conference (COMPSAC) 2003*, Dallas, TX, 2003, pp. 56 - 63

[50] M. Phongpaibul and B. Boehm, "An Empirical Comparison Between Pair Development and Software Inspection in Thailand," in *International Symposium on Empirical Software Engineering*, Rio de Janeiro, 2006, pp. 85-94.

[51] K. Schwaber and M. Beedle, *Agile Software Development with SCRUM*. Upper Saddle River, NJ: Prentice-Hall, 2002.

[52] B. Simon and B. Hanks, "First Year Students' Impressions of Pair Programming in CS1," in *International Computing Education Research Workshop*, Atlanta, GA, 2007, pp. 73-86.

[53] K. M. Slaten, M. Droujkova, S. Berenson, L. Williams, and L. Layman, "Undergraduate Student Perceptions of Pair Programming and Agile Software Methodologies: Verifying a Model of Social Interaction," in *Agile 2005*, Denver, CO, 2005, pp. 323-330.

[54] H. Srikanth, L. Williams, E. Wiebe, C. Miller, and S. Balik, "On Pair Rotation in the Computer Science Course," in *Conference on Software Engineering Education and Training*, Norfolk, VA, 2004, pp. 144 - 149.

[55] G. Succi, M. Marchesi, W. Pedrycz, and L. Williams, "Preliminary analysis of the effects of pair programming on job satisfaction," in *Fourth International Conference on eXtreme Programming and Agile Processes in Software engineering (XP2002)*, Sardinia, Italy, 2002, pp. 212-215.

[56] T. Van Toll III, R. Lee, and T. Ahlswede, "Evaluating the Usefulness of Pair Programming in a Classroom Setting," in *International Conference on Computer and Information Science (ICIS) 2007*, Melbourne, Qld. , 2007, pp. 302 - 308

[57] J. Vanhanen and H. Korpi, "Experiences of Using Pair Programming in an Agile Project," in *40th Annual Hawaii International Conference on System Sciences (HICSS) 2007* Hawaii, 2007, pp. 274b - 274b

[58] J. Vanhanen and C. Lassenius, "Perceived Effects of Pair Programming in an Industrial Context," in *33rd EUROMICRO Conference on Software Engineering and Advanced Applications, 2007.* , Lubeck 2007, pp. 211 - 218

[59] J. Vanhanen, C. Lassenius, and M. Mäntylä, "Issues and Tactics when Adopting Pair Programming: A Longitudinal Case Study," in *International Conference on Software Engineering Advances (ICSEA) 2007*, Cap Esterel, French Riviera, France, 2007, p. 70.

[60] G. M. Weinberg, *The Psychology of Computer Programming Silver Anniversary Edition*. New York: Dorset House Publishing, 1998.

[61] L. Williams and R. Kessler, *Pair Programming Illuminated*. Reading, Massachusetts: Addison Wesley, 2003.

[62] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair-Programming," *IEEE Software*, vol. 17, no. 4, pp. 19-25, July/August 2000 2000.

[63] L. Williams, W. Krebs, L. Layman, A. Antón, and P. Abrahamsson, "Toward a Framework for Evaluating Extreme Programming," in *Empirical Assessment in Software Eng. (EASE) 2004*, Edinburgh, Scot., 2004, pp. 11-20.

[64] L. Williams, L. Layman, J. Osborne, and N. Katira, "Examining the Compatibility of Student Pair Programmers," in *Agile 2006*, Minneapolis, MN, 2006, pp. 411-420.

[65] L. Williams, C. McDowell, N. Nagappan, J. Fernald, and L. Werner, "Building Pair

Programming Knowledge Through a Family of Experiments," in *International Symposium on Empirical Software Engineering (ISESE) 2003*, Rome, Italy, 2003, pp. 143-152.

[66] L. Williams, A. Shukla, and A. Antón, "An Initial Exploration of the Relationship Between Pair Programming and Brook's Law," in *Agile Development Conference 2004*, Salt Lake City, 2004, pp. 11-20.

[67] L. Williams, E. Wiebe, K. Yang, M. Ferzli, and C. Miller, "In Support of Pair Programming in the Introductory Computer Science Course," *Computer Science Education*, vol. 12, no. 3, pp. 197-212, 2002.

[68] L. A. Williams, "The Collaborative Software Process," in *Department of Computer Science* Salt Lake City, UT: University of Utah, 2000.

[69] L. A. Williams and R. R. Kessler, "All I Ever Needed to Know About Pair Programming I Learned in Kindergarten," in *Communications of the ACM*. vol. 43, 2000, pp. 108-114.

[70] S. Xu and V. Rajlich, "Pair Programming in Graduate Software Engineering Course Projects," in *Frontiers in Education*, Indianapolis, 2005, pp. F1G-7-F1G12.

[TABLES AND ILLUSTRATIONS ATTACHED AS SEPARATE FILES]