Strengthening the Case for Pair-Programming

Laurie Williams	Robert R. Kessler	Ward Cunningham	Ron Jeffries
North Carolina State University	University of Utah	Cunningham & Cunningham, Inc	
william@csc.ncsu.edu	kessler@cs.utah.edu	ward@c2.com	ronjeffries@acm.org

Abstract

Pair programming – two programmers working side-by-side at one computer collaborating on the same design, algorithm, code or test – has been practiced in industry with great success for years. Higher quality products are produced faster. Yet, most who have not tried and tested pair-programming reject the idea immediately as an overtly redundant, wasteful use of programming resources. The purpose of this article is to demonstrate through anecdotal, qualitative and quantitative evidence, that incorporating pair-programming into a software development process will help yield software products of better quality in less time with happier, more confident programmers. Supportive evidence comes from professional programmers and from the results of a structured experiment run with advanced undergraduate students at the University of Utah.

Keywords: pair-programming, collaborative programming, productivity, quality
Genre: Applied Research Results

> *You know what I like about pair-programming? First, it's something that has shown to help produce quality products. But, it's also something that you can easily add to your process that people actually want to do. It's a conceptually small thing to add, as opposed to having an overblown methodology shoved down your throat. And, when times get tough, you wouldn't likely forget to do pair-programming or decide to drop it "just to get done." I just think the idea of working together is a winner.*
>
> -- Chuck Allison, Consulting Editor, C/C++ Users Journal

---

begin sidebar:

A First Pair-Programming Experience

Ron Jeffries

Like most programmers, I had done some pair programming, usually when working on something particularly tricky, or during some difficult debugging sessions. Although Ward Cunningham had recommended full-time pair-programming to me a few times, my first experience with "real" pair-programming came on the C3 project, where I was coach.

I was sitting with one of the least-experienced developers, working on some fairly straightforward task. Frankly, I was thinking to myself that with my great skill in Smalltalk, I would soon be teaching this young programmer how it's really done.

We hadn't been programming more than a few minutes when the youngster asked me why I was doing what I was doing. Sure enough, I was off on a bad track. I went another way. Then the whippersnapper reminded me of the correct method name for whatever I was mistyping at the time. Pretty soon, he was suggesting what I should do next, meanwhile calling out my every formatting error and syntax mistake.

I'm not entirely stupid. I noticed very quickly that this most junior of programmers was actually helping me! Me! Can you believe it? Me! That has been my experience every time thereafter, in pair-programming. Having a partner makes me a better programmer. Ward was right – as usual.

end sidebar

Introduction

Throughout the software industry, there is an infinite myriad of software projects. Each has its own financial and business objectives and a diverse set of individuals that comprise its development team. There, then, must be a diverse set of software processes to match with these varying objectives and individuals. Pair-programming, the software development technique discussed in this article has been shown to improve software quality and reduce time-to-market. Additionally, the student and professional programmers involved in this study consistently found pair-programming more enjoyable.

In 1998, Temple University Professor Nosek reported on his study of 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." Combining their time, the pairs spent 60% more minutes on the task. However because they worked in tandem, they were able to complete the task 40% more quickly and effectively by producing better algorithms and code in less time. The majority of the programmers were initially 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 [1].

This practice of pair-programming – *two* programmers working side-by-side at *one* computer, collaborating on the same design, algorithm, code or test – is not new. In his 1995 book, "Constantine on Peopleware, " Larry Constantine reported observing *Dynamic Duos* at Whitesmiths, Ltd. producing code faster and more bug-free than ever before [2]. That same year, Jim Coplien published the "Developing in Pairs" Organizational Pattern [3]. In 1996, the Extreme Programming (XP) software development methodology started evolving. XP was developed initially by Smalltalk code developer and consultant Kent Beck with authors Ward Cunningham and Ron Jeffries. A significant part of XP is pair-programming and those practicing pair-programming within XP are the largest known group of pair programmers. XP's project success rate is so impressive that it has aroused the curiosity of many highly-respected software engineering researchers and consultants. XP credits much of this success to their use of pair-programming by <u>all</u> their programmers, experts and novices alike. XP advocates pair-programming with such fervor that even prototyping done solo is scrapped and re-written with a partner.

Yet, most who have not tried and tested pair-programming reject the idea immediately as an overtly redundant, wasteful use of programming resources. "Why would I put two people on a job that just one can do? I can't afford to do that!" The purpose of this article is to demonstrate, through anecdotal, qualitative and quantitative evidence, that incorporating pair-programming into a software development process will help yield software products of better quality in less time with happier, more confident programmers. These results apply to all levels of programming skill from novice to expert.

Supportive evidence comes from professional programmers and from advanced undergraduate university students who participated in a structured experiment at the University of Utah in 1999. A purpose of the educational study was to validate quantitatively the anecdotal and qualitative pair-programming results observed in industry. (Details of the research design of this experiment will be outlined in the next section.)

Experimental results show pair-programming pairs develop better quality code faster with only a minimal increase in pre-release programmer hours. “Two programmers in tandem is not redundancy; it’s a direct route to greater efficiency and better quality [2].” This is a mixed result. Later in this paper we will reflect on how improved quality and early completion might be factored into a total cost of software development.

University Experiment

In 1999 at the University of Utah, students in the Senior Software Engineering course participated in a structured experiment. The students were aware of the importance of the experiment, the need to keep accurate information, and that each person (whether in the control or experimental group) was a very important part of the outcome. All students attended the same classes, received the same instruction, and participated in class discussions on the pros and cons of pair programming. When asked the first day of class, 35 of the 41 students (85%) indicated a preference for pair-programming. (Later, many of the 85% admitted that they were initially reluctant, but curious, about pair-programming.)

The students were divided into two groups; both groups were deliberately comprised of the same mix of high, average and low performers. Thirteen students formed the control group in which all the students worked individually on all assignments. Twenty-eight students formed the experimental group in which all worked in two-person collaborative teams; collaboratively, they completed the same assignments as the individuals. (The collaborative pairs also did additional assignments to keep the overall workload the same between the two groups.) All 28 of the students in the experimental group had expressed an interest in pair-programming. Some of the students in the control group had actually wanted to try pair-programming. It is important to note that prior to enrolling in this class, students had significant coding practice. Most students have had industry/internship experience and have written small compilers, operating system kernels, and interpreters in other classes.

Cycle time, productivity and quality results were compared between the two groups. Students recorded information in a web-based tool about the time they spent on the project. Quality was analyzed based on the results of the automated testing executed by an impartial teaching assistant.

Pair-Jelling

In pair-programming, <u>two</u> programmers <u>jointly</u> produce <u>one</u> artifact (design, algorithm, code, etc.). The two programmers are like a coherent, intelligent organism working with one mind, responsible for every aspect of this artifact. One partner 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 driver – watching for defects, thinking of alternatives, looking up resources, and considering strategic implications of the work at hand. The roles of driver and observer are deliberately switched between the pair periodically. Both are equal, active participants in the process at all times and wholly share the ownership of the work products whether they be a morning’s effort or an entire project.

Programmers have long been conditioned to working alone. Many venture into their first pair programming experience skeptical that they would benefit from collaborative work. They wonder about the added communication that will be required, about adjusting to the other’s working habits, programming style, and ego, and about disagreeing on aspects of the implementation. Indeed, there is an initial adjustment period in the transition from solitary to collaborative programming. (For guidelines on making an effective transition from solo to pair-programming, see [4].) In industry, this adjustment period has historically taken hours or days, depending upon the individuals. At the university, the students generally adjusted after the first assignment, though some reported an even shorter adjustment period. For the first assignment, the pairs finished in shorter elapsed time and had better quality, but they took, on average, 60% more

programmer hours to complete the assignment when compared to the individuals. These results are similar to Nosek's initial study of professional programmers discussed above. After the adjustment time, this 60% decreased dramatically to a minimum of 15%. The end of the second assignment marked an important milestone -- all students reported that they had overcome their constant urge to grab the mouse or keyboard from their partner's hands!

It doesn't take many victorious, clean compiles or declarations of "We just got through our test with no defects!" for the teams to celebrate their union – and to feel as one jelled, collaborative team. Tom Demarco shares his inspiring view on this type of union. "A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts. The production of such a team is greater than that of the same people working in unjelled form. Just as important, the enjoyment that people derive from their work is greater than what you'd expect given the nature of the work itself. In some cases, jelled teams working on assignments that others would declare downright dull have a simply marvelous time . . . Once a team begins to jell, the probability of success goes up dramatically. The team can become almost unstoppable, a juggernaut for success [5]."

The authors of *The Wisdom of Teams* [6] offer some explanation for what happens when a team jells. They define high performance as productivity exceeding the sum of the individuals. The members of such team learn the strengths and weaknesses of each member. With this knowledge they can adjust their activities to exploit strengths and avoid weaknesses.

Pair-Development

Ideally, pair-programmers should work together constantly. However, reality dictates that at times the pair must split – for illness, time conflicts, or even efficiency. Experienced pair programmers have prioritized which parts of the development cycle are most important to work together, which can be done separately, and what to do with the independently developed work when reuniting. This information has been derived from personal experience and surveys of professional programmers and students. When a whole group adopts pair-programming as the normal way of working, the long term continuity of any particular pair becomes less important. The ideal becomes *a* pair, not *the* pair, for all development. By pairing regularly with all members of a group, an individual programmer can maintain sufficient general awareness to substitute for a missing partner on a moments notice.

Unanimously, pair-programmers agree that **pair-analysis and pair-design** is critical for their pair success. First, it is important for the pair to collectively agree on the development direction and strategy outlined during these stages. Additionally, it is doubtless true that "two brains are better than one" when performing analysis and design. Together, pairs have been found to consider many more possible solutions to a problem and more quickly converge on which is best to implement. Their constant feedback, debate, and idea exchange significantly decreases the probability of proceeding with a bad design [7]. Perhaps, the collaborators can perform tasks that might be just too challenging for one to do alone. One professional programmer reflects, "It is a powerful technique as there are two brains concentrating on the same problem all the time. It forces one to concentrate fully on the problem at hand."

While one partner is busy typing or writing down the design, the other partner can think more strategically about the implications of the design and can perform a continuous design review -- considering whether the design will run into a dead end or if there is a better strategy. Design defects are prevented or removed almost as soon as they hit the paper. A further benefit is the reduction of "design tunnel vision," which occurs when one makes a design decision and sticks with it no matter what. With the partner reviewing and questioning decisions, the chance of exploring good design alternatives is increased. Importantly, in preparation for a most effective joint analysis and design session, programmers individually read and fully understand the problem they need to solve, think about complex logical problems, and do experimental prototyping.

After developing a quality design, the pair must implement it. Again, with **pair-implementation**, one programmer is the "driver" and types into the computer while the other is actively engaged in observing, performing a continuous code review, and considering the strategic implications of the implementation. This "side by side" form of code review has been found to be a most effective and efficient form of defect removal. "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 [8]." With pair-programming, "four eyeballs are better than two," and a momentous number of defects are prevented, removed right from the start.

Interestingly, programmers view pair-analysis and design as more critical than pair-implementation. Pairs report that they plan to code individually at times. They often deliberately split for the more rote, routine, simple coding of a project. They find performing this type of programming is more effective done individually. It seems that some tasks, such as GUI drawing, are largely detail-oriented in nature. Developers report that having a partner for this work doesn't help much. Additionally, they do allow themselves to code average complexity modules if the situation, such as time conflicts, dictates – though most immediately feel notably uncomfortable and more error prone. Some profess that any work done individually should be scrapped and redone by the pair. Most programmers perform a thorough review of the individual work and incorporate it into the project. A small minority integrates individual work without review.

Pair-testing is the least critical part of the develop cycle, as long as the pair develops the test cases together. Pairs sometimes split up to run test cases, often side-by-side at two computers. When defects are uncovered, the pairs usually rejoin to collaborate to find the best solution.

Pair-Results

> *"Some may question the value . . . if the collaborators do not perform "twice" as well as individuals, at least in the amount of time spent. For instance, if the collaborators did not perform the task in half the time it takes an individual, it would still be more expensive to employ two programmers. However, there are at least two scenarios where some improved performance over what is expected or possible by a single programmer may be the goal: (1) speeding up development and (2) improving software quality. [1]"*

Organizations that have heavily used pair-programming have yielded superior results. The largest example is the sizable Chrysler Comprehensive Compensation system (the C3 project discussed in Ron Jeffries' story above) launched in May 1997. After finding significant, initial development problems, Beck and Jeffries restarted this development using Extreme Programming principles, including the exclusive use of pair-programming. The payroll system pays some 10,000 monthly-paid employees and has 2,000 classes and 30,000 methods [9], went into production almost on schedule, and is operational today. In the last five months before the first production, almost the only defects that making it through unit and functional testing were written by someone programming alone. Says one pair-programmer in an anonymous survey [10] of professional pair programmers, "I strongly feel pair-programming is the primary reason our team has been successful. It has given us a very high level of code quality (almost to the point of zero defects). The only code we have ever had errors in was code that wasn't pair programmed . . . we should really question a situation where it isn't utilized."

Our experimental class produced quantitative results supporting the pair-programming results in industry. The students completed four assignments over a period of six weeks. Thirteen individuals and fourteen collaborative pairs completed each assignment. The pairs always passed more of the automated post-development test cases run by an impartial teaching assistant (see Table 1 below). (The difference in quality levels is statistically significant to $p < .01$.) Their results were also more consistent, while the individuals varied more about the mean. Individuals intermittently didn't hand in a program or handed it in late; pairs handed in their

assignments on time. This result can be attributed to a positive form of "pair-pressure" the programmers put on each other. The programmers admit to working harder and smarter on programs because they do not want to let their partner down. Individuals do not have this form of pressure and, therefore, do not perform as consistently.

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

Table 1: Percentage of Test Cases Passed

We can think of three reasons why pair-programming might be valued higher in industry than shown in our results. First, if the errors found in the development are judged significant, the developments with more errors will spend more time correcting them. We did not directly measure this cost of rework. Second, there is a long-term cost of quality associated with unclear designs that are more difficult to change in later releases. We will say more about this in the next paragraph. Third, a high-order of staff cross-training is achieved as a side-effect of pair-programming and this removes additional long-term costs from development. "Even if you weren't more productive, you would still want to pair, because the resulting code quality is so much higher [11]."

Many people's gut reaction is to reject the idea of pair-programming because they assume that there will be a 100% programmer-hour increase by putting two programmers on a job that one can do. After the initial adjustment period, discussed above, the total programmer hours spent on each assignment trended downward dramatically. See Figure 1 below. Certainly, if the individuals were required to spend the additional time to bring their code to the quality of the pairs, the individuals would take even more time than the pairs. Because the pairs worked in tandem, they were able to complete their assignments 40-50% more quickly. In today's competitive market -- where getting a quality product out as fast as possible is a competitive advantage or can even mean survival – pair-programming seems the way to go. Statistics show that fixing defects after release to customers can cost hundreds more than finding and fixing them during the development process. The benefits of getting a product out faster, reducing maintenance expenses, and improving customer satisfaction with product quality outweigh the minimal programmer hour increase that was seen.

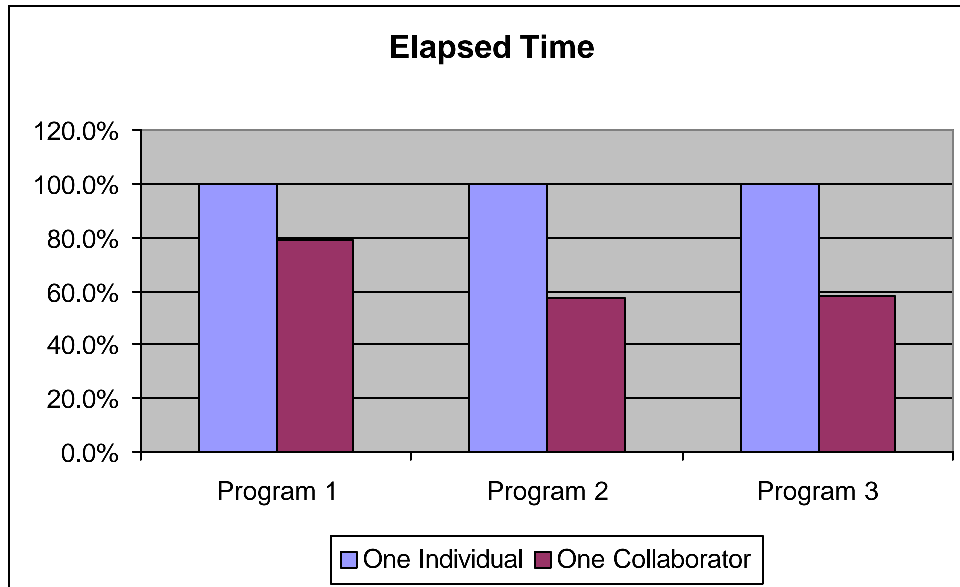

Figure 1

There are many reasons why the programmer hours do not double with pair-programming, as might have been expected. First, as discussed earlier, "two heads are better than one" and "four eyeballs are better than two". Collaboration improves the problem-solving process. Because of this, far less time is spent in the chaotic, time-consuming compile and test phases. Says one of the students,

> *When I worked on the machine as the driver, I concentrated highly on my work. I wanted to show my talent and quality work to my partner. When I was doing it, I felt more confident. In addition, when I had a person observing my work, I felt that I could depend on him, since this person cared about my work and I could trust him. If I made any mistakes, he would notice them, and we could have a high quality product. When I was the non-driver, I proofread everything my partner typed. I felt I had a strong responsibility to prevent any errors in our work. I examined each line of code very carefully, thinking that, if there were any defects in our work, it would be my fault. Preventing defects is the most important contribution to the team, and it put a bit of pressure on me.*

Additionally, pairs find that, pooling their joint knowledge, they can conquer almost any technical task immediately. In teaching the individual/collaborative university class, the first author noticed a two- or three-fold increase in technical questions from individual workers than from collaborative workers. While waiting for the answers to these questions (often done via email), these students are generally unproductive. This situation easily translates to a professional environment.

Pair programmers also put pair-pressure on each other to perform. Pair-pressure is a positive form of peer pressure. Programmers note that even if they come to work after a bad night or are preoccupied with other thoughts – their partner draws their attention to the task at hand. Partners keep each other focused and on-task. Programmers are far less likely to spend time on the telephone, reading and answering emails, or surfing the web because their partner is awaiting their attention. Additionally, pairs usually come to the session with an objective of completing a particular task. They are determined to complete that task during the session and, therefore, work with much more focus and intensity than individuals working alone.

Pair-Satisfaction

As opposed to many techniques and processes professed to improve software quality and productivity, pair-programming is one that programmers actually enjoy. Pair-programming actually improves their job satisfaction and overall confidence. In a survey of professional pair programmers, 96% stated that they enjoyed their job more than when they programmed alone. The 41 collaborative programmers involved with the university experiment were surveyed six different times. Consistently, over 90% of these also stated that they enjoyed collaborative programming more than solo programming. Additionally, virtually every one of the surveyed professional programmers stated that they were more confident in their solutions when they pair programmed. Almost 95% of the students agreed with this statement.

A natural correlation exists between these two measures of overall satisfaction. The pairs enjoy their work more because they are more confident in their work. Someone is there to help them if they are confused or unknowing. They have a friend to talk to and to bounce ideas off of. They spend more time doing challenging design and less time doing annoying debugging. Positive feelings about the collaborative problem-solving process improve their overall performance. They leave each collaborative session with an exhilarated, "we nailed that one" feeling. The authors have pair-programmed and agree completely that pair-programming is far more enjoyable than individual programming. There is a shared euphoria that is gained from the successful completion of a pair-programmed task.

Evidently, most programmers enjoy pair-programming. At time, however, programmers are matched with someone they have trouble working with. Most often, the difficult working arrangement is due to being paired with someone with excess ego (who has a "my way or the highway attitude) or too little ego (who contributes little to the pair because he or she has trouble asserting themselves). It must also be noted that the majority of those involved in the study and of those that agreed to do complete the survey are self-selected pair-programmers. Further study is needed to examine the eventual satisfaction of programmers who are forced to pair-program despite their adamant resistance.

Future Work

Large group projects and code integration often brings its share of difficulties. Fred Brooks, in his 1975 landmark book "The Mythical Man-Month" [12] states Brook's Law:

Adding manpower to a late software project makes it later.

The logic behind this law focuses on intercommunication effort. "In tasks requiring communication among the subtasks, the effort of communication must be added to the amount of work to be done . . . The added burden of communication is made up of two parts, training and intercommunication . . . If each part of the [n] task[s] must be separately coordinated with each other part, the [intercommunication] effort increases as $n(n+1)/2$ [12]." Integrating the partitioned tasks of programmers requires this extra effort of intercommunication. Through pair programming, the number of separate tasks to be integrated can be halved and thus we anticipate that the teams of pair-programmers should fair much better. We would like to run another university study to analyze the effect of pair-programming on larger groups.

Finally, we would like to see the same experiments applied to industry. It would be beneficial to run such an experiment in an industrial setting – perhaps with part of a larger development team. Anyone interested in running such an experiment should contact the first author.

Summary

For years, pair-programmers in industry have claimed that they have produced higher quality software products in a shorter amount of time. However, their results were anecdotal and qualitative – "it works" and "it feels right." We have performed research at the University of Utah to validate, quantitatively, the industry claims. Indeed, the research reveals that through pair-programming, software products can be produced in less time, with higher quality. As an added benefit, virtually every programmer involved in the study or surveyed in industry has stated that they enjoy doing their work more and feel more confident in their work when they work with a partner. In many high-pressure, tight-schedule situations, individual programmers may tend to revert to undisciplined practices. Partners, however, put positive pair-pressure on each other and each are less likely to deviate from established practices – "chances are that even if you feel like blowing off one of these practices, your partner won't. [11]" The net result is a winner -- the production of higher quality products faster.

Bibliography

[1] J. T. Nosek, "The Case for Collaborative Programming," in *Communications of the ACM*, vol. 41 no. 3, 1998, pp. 105-108.

[2] L. L. Constantine, *Constantine on Peopleware*. Englewood Cliffs, NJ: Yourdon Press, 1995.

[3] 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.

[4] 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, no. 5, 2000.

[5] T. DeMarco and T. Lister, *Peopleware*. New York: Dorset House Publishers, 1977.

[6] J. R. Katzenbach and D. K. Smith, *The Wisdom of Teams: Creating the High-Performance Organization*: Harper Business, 1994.

[7] 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.

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

[9] A. Anderson, Beattie, Ralph, Beck, Kent et al., "Chrysler Goes to "Extremes"," in *Distributed Computing*, vol. October 1998, 1998, pp. 24-28.

[10] L. Williams, "Pair Programming Questionnaire," http://limes.cs.utah.edu/questionnaire/questionnaire.htm.

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

[12] F. P. J. Brooks, *The Mythical Man-Month*: Addison-Wesley Publishing Company, 1975.

-- begin sidebar

Extreme Programming (XP) is a method for software development that favors informal and immediate communication over the detailed and specific work products required by any number of traditional design methods. Pair programming fits well within XP for a number of reasons ranging from quality and productivity to vocabulary development and cross training. XP so heavily relies on pair programming that it insists all production code be written by pairs.

Extreme Programming is a system of a dozen practices well suited for small to mid sized teams developing software with vague or changing requirements. XP copes with change by delivering software early and often and by absorbing feedback from deliveries into the development culture and ultimately into the code. The following paragraphs highlight how pair programming interacts with a sampling of other XP practices.

- Developers work on only one requirement at a time, most likely that with the greatest business value as established by the customer. Pairs form to better interpret requirements or to better place their implementation within the codebase.
- Developers create unit tests for the expected behavior of their code and then write the simplest, most straightforward implementations that pass their tests. Pairs help each other maintain the discipline of writing tests first and the complementary, though quite distinct discipline of writing simple solutions.
- Developers expect their intentions to show clearly in the code they write and will refactor their code and other's if necessary to achieve this result. A partner who has been tracking the intention of a programmer is well suited to judge the expressiveness of the program as written.
- Developers continuously integrate their work into a single development thread who's health they test by running comprehensive unit tests. With each integration the pair releases ownership of their work to the whole team. This makes a good point for different pairings to form should another combination of talent be more appropriate for the next piece of work.

To learn more about Extreme Programming see *Extreme Programming Explained: Embrace Change*, by Kent Beck, ISBN 0201616416, or consult the Extreme Programming Roadmap at the xp.c2.com website where a lively community debates each XP practice.

Corresponding author:
Laurie Williams
University of Utah Computer Science
50 S. Central Campus #3190
Salt Lake City, UT 84112
(801) 585-3736
Fax (435) 649 2771
lwilliam@cs.utah.edu

Biographical sketches of authors ➔ **see revision in bio of Laurie Williams**

Laurie Williams is a faculty member at North Carolina State. She received a BS in Industrial Engineering from Lehigh University, an MBA from Duke University, and a PhD in Computer Science from the University of Utah. Laurie worked at IBM in Research Triangle Park, NC for nine years in engineering and software development positions. Her research interests are in software engineering, software process, collaborative programming and eCommerce.

Robert R. Kessler has been on the faculty of the University of Utah since 1983 and is currently a professor and chairman of the Department of Computer Science. In the early 90's, he founded the Center for Software Science, a state of Utah Center of Excellence. He has also founded several startup companies and is currently involved with an Internet startup company, emWare, as a member of the board. He has served as member-at-large and Vice Chairman for Conferences of ACM SIGPLAN. He recently completed a seven year assignment as co-editor-in-chief of the International Journal of Lisp and Symbolic Computation. His current research interests are in agents, software engineering, distributed systems, and visual programming.

Ward Cunningham is a founder of Cunningham & Cunningham, Inc. Ward has served as a Principal in the IBM Consulting Group and as Director of R&D at Wyatt Software. Ward is well known for his contributions to the developing practice of object-oriented programming. Ward is active with the Hillside Group and has served as program chair of the Pattern Languages of Programs conference which it sponsors. Ward created the CRC design method which helps teams find core objects for their programs. Ward has written for PLoP, JOOP and OOPSLA on Patterns, Objects, CRC and related topics.

Ron Jeffries has been developing software since 1961, when he accidentally got a summer job at Strategic Air Command HQ, and they accidentally gave him a FORTRAN manual. He and his teams have built operating systems, language compilers, relational and set-theoretic database systems, manufacturing control, and applications software, producing about a half-billion dollars in revenue, and he wonders why he didn't get any of it. For the past few years he has been learning, applying, and teaching the Extreme Programming discipline.