

Strengthening the Case for Pair Programming

The software industry has practiced pair programming—two programmers working side by side at one computer on the same problem—with great success for years. But people who haven't tried it often reject the idea as a waste of resources. The authors demonstrate that using pair programming in the software development process yields better products in less time—and happier, more confident programmers.

Laurie Williams, North Carolina State University

Robert R. Kessler, University of Utah

Ward Cunningham, Cunningham & Cunningham

Ron Jeffries

For years, programmers in industry have claimed that by working collaboratively, they have produced higher-quality software products in shorter amounts of time. But their evidence was anecdotal and subjective: “It works” or “It feels right.” To validate these claims, we have gathered quantitative evidence showing that pair programming—two programmers working side by side at one computer on the same

design, algorithm, code, or test—does indeed improve software quality and reduce time to market. Additionally, student and professional programmers consistently find pair programming more enjoyable than working alone.

Yet most who have not tried and tested pair programming reject the idea as a 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!” But we have found, as Larry Constantine wrote, that “Two programmers in tandem is not redundancy; it’s a direct route to greater efficiency and better quality.”¹

Our supportive evidence comes from professional programmers and from advanced undergraduate students who participated in a structured experiment. The experimental results show that programming pairs develop better code faster with only a minimal increase in prerelease programmer hours. These results apply to all levels of

programming skill from novice to expert.

Earlier Observations

In 1998, Temple University professor John Nosek reported on his study 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 40% faster than the control groups, and produced better algorithms and code.²

Most of the programmers were initially skeptical of the value of collaborating and

You know what I like about pair programming? First, it's something that has been 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*
(in a conversation with Laurie Williams)

thought it would not be an enjoyable process. But, according to Nosek, "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."²

Pair programming is not new. In 1995, Larry Constantine reported observing pairs of programmers producing code faster and freer of bugs than ever before.² The same year, Jim Coplien published what he called the developing-in-pairs organizational pattern.³

In 1996, Smalltalk code developer and consultant Kent Beck, with authors Ward Cunningham and Ron Jeffries, began developing the Extreme Programming software development methodology (see the sidebar). XP is now practiced by programmers worldwide. A significant part of XP is pair programming, and people who practice XP are the largest known group of pair programmers. The XP methodology's success rate is so impressive that it has aroused the curiosity of many software engineering researchers and consultants. The XP founders credit much of this success to the use of pair programming by all XP programmers, experts and novices alike.

The Development Cycle

In pair programming, two programmers jointly produce one artifact (design, algorithm, code). The two programmers are like a unified, intelligent organism working with one mind, responsible for every aspect of this artifact. One partner, the driver, controls the pencil, mouse, or keyboard and writes the code. The other partner continuously and actively observes the driver's work, watching for defects, thinking of alternatives, looking up resources, and considering strategic implications. The partners deliberately switch roles periodically. Both are equal, active participants in the process

at all times and wholly share the ownership of the work product, whether it is a morning's effort or an entire project.

Ideally, pair programmers will always work together. But reality—illness, time conflicts, or even efficiency—dictates that at times the pair must split. Experienced pair programmers prioritize the parts of the development cycle, deciding which are most important to work on together and which can be done separately. They must also decide what to do with the independently developed work when they reunite. When a larger group adopts pair programming as the normal way of working, the long-term continuity of a particular pair becomes less important. By pairing regularly with all members of the group, an individual programmer maintains sufficient general awareness to substitute for a missing partner at a moment's notice.

Unanimously, the pair programmers we have observed agree that pair analysis and pair design are critical to success. That is, two brains are better than one when performing analysis and design. Pairs consider many more possible solutions to a problem and converge more quickly on which solution to implement. According to researchers Nick Flor and Edwin Hutchins, the feedback, debate, and idea exchange between partners significantly decrease the probability of proceeding with a bad design.⁴ Collaborators often can perform tasks that are too challenging for one to do alone. In an anonymous online survey (<http://limes.cs.utah.edu/questionnaire/questionnaire.htm>), one professional programmer reflected, "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 continuously review the design and think

What Is eXtreme Programming?

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

XP consists of a dozen practices appropriate for small to midsize teams developing software with vague or changing requirements. The methodology copes with change by delivering software early and often and by absorbing feedback into the development culture and ultimately into the code.

Several XP practices involve pair programming:

- Developers work on only one requirement at a time, usually the one with the greatest business value as established by the customer. Pairs form to interpret requirements or to place their implementation within the code base.
- Developers create unit tests for the code's expected behavior 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 refactor their code and other's if necessary to achieve this result. A partner who has been tracking the programmer's intention is well equipped to judge the program's expressiveness.
- Developers continuously integrate their work into a single development thread, testing its health by running comprehensive unit tests. With each integration, the pair releases ownership of their work to the whole team. At this point, different pairings can form if another combination of talent is more appropriate for the next piece of work.

To learn more, see Kent Beck's book,¹ or consult the eXtreme Programming Roadmap at xp.c2.com, where a lively community debates each XP practice.

Reference

1. K. Beck, *eXtreme Programming Explained: Embrace Change*, Addison Wesley Longman, Reading, Mass., 2000.

more strategically about its implications. Will it run into a dead end? Is there a better strategy? Thus, pair programming prevents design defects or removes them almost as soon as they hit the paper. A further benefit is the elimination of design tunnel vision—making a design decision and sticking with it no matter what. With partners reviewing and questioning decisions, the exploration of design alternatives increases. To prepare for an effective joint analysis and design session, programmers should individually research the problem they need to solve and perhaps do experimental prototyping.

After developing a quality design, the pair must implement it. Again, one programmer is the driver, who types code into the computer, while the other observes, continuously reviews the code, and considers the implementation's strategic implications. This side-by-side form of code review is an effective and efficient form of defect removal. As Gerald Weinberg observes,

*"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."*⁵

With pair programming, four eyeballs are better than two, and a huge number of defects are prevented right from the start.

Programmers view pair analysis and design as more critical than pair implementation. Pairs report that they sometimes work on implementation individually, usually the rote, routine, or simple coding phases of a project. They find that doing this type of programming individually is more effective. Developers report that having a partner for detail-oriented tasks such as drawing a graphical user interface doesn't help much. Some programmers code average-complexity modules individually if a situation such as a time conflict dictates, but most immediately feel uncomfortable and more error prone. Some even say that any work done individually should be scrapped and redone by the pair. Most programmers perform a thorough review of the individual work before incorporating it in the project. A small minority integrate individual work without review.

The least critical part of the development cycle is pair testing, 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 they discover defects, the partners usually rejoin to find the best solution.

University Experiment

In 1999 at the University of Utah, senior software engineering students participated in a structured experiment. Its purpose was to validate quantitatively the anecdotal and qualitative pair-programming results ob-

Some organizations that use pair programming heavily have achieved superior results.

served in industry. All students attended the same classes, received the same instruction, and participated in class discussions on the pros and cons of pair programming. On the first day of class, 35 of the 41 students (85%) indicated a preference for pair programming. Later, many of the 35 admitted that they were initially reluctant, but curious, about pair programming.

We divided the students into two groups, both composed of the same mix of high, average, and low performers, as determined by their grade point averages. Thirteen students formed the control group, in which all worked individually on all assignments. Twenty-eight students formed the experimental group, in which all worked in two-person, collaborative teams on the same assignments as the individuals. The collaborative pairs also had additional assignments to keep the workload equal between the two groups.

All 28 students in the experimental group had expressed an interest in pair programming. Some students in the control group had wanted to try pair programming. Prior to enrolling in this class, the students had significant coding practice. Most had industry internship experience, and most had written small compilers, operating system kernels, and interpreters in other classes.

The experiment compared cycle time, productivity, and quality between the two groups. The students recorded the time they spent on the project in a Web-based tool. An impartial teaching assistant executed automated testing to analyze programming quality.

Pair Jelling

Long conditioned to working alone, many programmers venture into their first pair-programming experience skeptical of benefiting from collaborative work. They wonder how much additional communication will be required; how they will adjust to the other's work habits, programming style, and ego; and whether they will disagree on aspects of the implementation. Indeed, programmers go through an initial adjustment period in the transition from solitary to collaborative programming. (A recent article provides guidelines on making this transition.⁶) Our anecdotal evidence and observations show that in industry this adjustment period varies from a few hours to several days, depending on the individu-

als. In the university experiment, the students generally adjusted after the first assignment, although some reported an even shorter adjustment period.

For the first assignment, the pairs finished in a shorter elapsed time and produced a better product than the individuals. On average, however, they took 60% more programmer hours to complete the assignment. These results are similar to Nosek's results described earlier. After the adjustment period, this 60% decreased dramatically to a minimum of 15%. The end of the second assignment marked an important milestone: all students reported they had overcome the constant urge to grab the mouse or keyboard from their partner's hands.

It doesn't take many clean compiles or declarations of "We just got through our test with no defects!" for the partners to feel as one. Tom DeMarco and Timothy Lister describe this type of union as

a jelled team.... 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 inunjelled 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.⁷

Jon Katzenbach and Douglas Smith explain what happens when a team jells: The members of such a team learn the strengths and weaknesses of each member.⁸ With this knowledge, they can adjust their activities to exploit strengths and avoid weaknesses. As a result, the team's productivity exceeds the sum of its individuals' productivity.

Pair-Programming Results

Some may question the value [of pair programming] 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 expen-

sive 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.²

Some organizations that use pair programming heavily have achieved superior results. The largest example is the Chrysler Comprehensive Compensation System (the C3 project discussed in Ron Jeffries' sidebar). After finding significant initial development problems, Beck and Jeffries restarted the development using XP principles, including the use of pair programming. In the last five months of development, almost the only defects that made it through unit and functional testing were written by solo programmers. The payroll-system software, which handles roughly 10,000 employees, and consists of 2,000 classes and 30,000 methods, went into operation in May 1997, almost on schedule.⁹

In the online survey of professional pair programmers, we found similar experiences. For example, one respondent wrote:

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 university experiment produced quantitative results similar to industry results. The students completed four assignments over a period of six weeks. The pairs always passed more of the automated post-development test cases (see Table 1). Their results were also more consistent, while the individuals varied more about the mean. Some individuals didn't hand in a program or handed it in late; pairs handed in their assignments on time. We attribute the pairs' punctuality to positive pressure the partners exert on each other. They admit to working "harder and smarter" because they don't want to let their partner down. Without this pressure, individuals don't perform as consistently.

In addition to the development-quality

Table 1		
Percentage of Test Cases Passed*		
	Individuals	Pairs
Program 1	73.4	86.4
Program 2	78.1	88.6
Program 3	70.4	87.1
Program 4	78.1	94.4

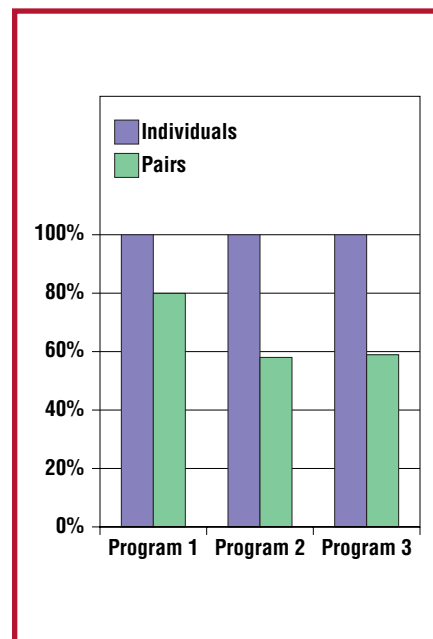
*The difference in quality levels is statistically significant to $p < 0.01$; p is the probability that these results could occur by chance.

benefits found in our study, we can think of three reasons why pair programming might be more valuable to industry than our results indicate. First, the longer defects remain in the product, the more costly it is to find and fix them. The rework necessary to bring individual programmers' quality levels up to that of collaborative programmers could be very time consuming and expensive. Some of these defects could escape to the customer. Second, unclear designs are difficult to change in later releases, resulting in the long-term cost of poor quality. Third, a side effect of pair programming is a high order of staff cross training, which removes additional long-term costs from development. As Beck writes, "Even if you weren't more productive, you would still want to pair, because the resulting code quality is so much higher."¹⁰

Many people reject the idea of pair programming because they assume that putting two programmers on a job that one can do will increase programmer hours 100%. In our experiment, however, after the initial adjustment period, the total programmer hours the pairs spent on each assignment decreased dramatically (see Figure 1). If we had required the individuals to spend additional time improving their code to the level of the pairs' code, the individuals would have taken even more time than the pairs. By working in tandem, the pairs completed their assignments 40% to 50% faster.

In today's market, getting a quality product out as quickly as possible is a competitive advantage that might even mean survival. Fixing defects after release to customers can cost far more than finding and fixing them during development.

Figure 1. Comparison of pair programmers' and individuals' project completion times. (Data entry problems prevented accurate recording of the completion times for Program 4.)



Unlike many techniques put forth to improve software quality and productivity, pair programming is one that programmers actually enjoy.

The benefits of getting a product out faster, reducing maintenance expenses, and improving customer satisfaction outweigh any programmer-hour increase that pair programmers might incur.

For several reasons, programmer-hours do not double with pair programming as one might expect. First, collaboration improves the problem-solving process. As a result, far less time is spent in the chaotic, time-consuming compile and test phases. Said one of the students in the experiment:

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 nondriver, 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 by pooling their joint knowledge, they can conquer almost any technical task immediately. In teaching the university class that participated in the experiment, the instructor noticed that individual workers asked two or three times more technical questions than did collaborative workers. While waiting for the answers to these questions, these students were generally unproductive. We would expect this situation to translate to a professional environment.

As mentioned earlier, pair programmers put pressure on each other to perform, keeping their partner focused and on task. Programmers have told us 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. They are less likely to spend time talking on the telephone, reading and answering e-mail, or surfing the Web if their partner is

waiting. Because of this peer pressure, pairs usually have a stronger motivation to complete the task during the session; therefore, they work with much greater focus and intensity than individuals.

Pair Satisfaction

Unlike many techniques put forth to improve software quality and productivity, pair programming is one that programmers actually enjoy. Pair programming improves their job satisfaction and overall confidence in their work. In the online survey of professional pair programmers, 96% stated that they enjoyed their job more than when they programmed alone. We surveyed the 41 collaborative programmers in the university experiment six times. Consistently, more than 90% stated that they enjoyed collaborative programming more than solo programming. Additionally, virtually all the surveyed professional programmers stated that they were more confident in their solutions when they pair programmed. Almost 95% of the students echoed this statement.

A natural correlation exists between these two measures of satisfaction. That is, 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 unknowledgeable. They can bounce ideas off a friend. They spend more time doing challenging design and less time doing annoying debugging. They leave each session with an exhilarated, we-nailed-that-one feeling. Indeed, having engaged in pair programming ourselves, we agree completely that it is far more enjoyable than individual programming. We too have experienced the shared euphoria that follows the successful completion of a collaborative task.

Although most programmers enjoy pair programming, sometimes they have trouble working with a particular partner. The difficulty often arises from being paired with someone with excess ego—the my-way-or-the-highway attitude. Or the problem might be too little ego—partners who have trouble asserting themselves and thus contribute little. The majority of participants in our study and in our survey were self-selected pair programmers. Further study is needed to examine the eventual satisfaction of programmers forced to pair program despite their resistance.

My First Pair-Programming Experience

by 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 a few times, my first experience with “real” pair programming came on the C3 [Chrysler Comprehensive Compensation] 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! Since then, that's been my experience every time in pair programming. Having a partner makes me a better programmer. Ward was right—as usual.

Large-group software projects are prone to peculiar difficulties. As the well-known Brooks's law states, “Adding manpower to a late software project makes it later.”¹¹ 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 $n(n + 1)/2$.*¹¹

Integrating the partitioned tasks of programmers requires this extra intercommunication effort. Pair programming can halve the number of separate tasks to be integrated, and thus we anticipate that large groups consisting of pair-programming teams 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 in an industrial setting—perhaps with part of a larger development team. Anyone interested in running such an experiment should contact Laurie Williams (william@csc.ncsu.edu). ☛

- shop, Ablex Publishing, New York, 1991, pp. 36–64.
5. G.M. Weinberg, *The Psychology of Computer Programming Silver Anniversary Edition*, Dorset House, New York, 1998.
6. L.A. Williams and R.R. Kessler, “All I Ever Needed to Know about Pair Programming I Learned in Kindergarten,” *Comm. ACM*, Vol. 43, No. 5, 2000, pp. 108–114.
7. T. DeMarco and T. Lister, *Peopleware*, Dorset House, New York, 1977.
8. J.R. Katzenbach and D.K. Smith, *The Wisdom of Teams: Creating the High-Performance Organization*, Harper Business, New York, 1994.
9. A. Anderson et al., “Chrysler Goes to ‘Extremes,’” *Distributed Computing*, Oct. 1998, pp. 24–28.
10. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison Wesley Longman, Reading, Mass., 2000.
11. F.P.J. Brooks, *The Mythical Man-Month*, Addison-Wesley, Reading, Mass., 1975.

About the Authors



Laurie Williams is a faculty member at North Carolina State University. Her research interests are software engineering, software process, collaborative programming, and e-commerce. 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. Contact her at williams@csc.ncsu.edu.

Robert R. Kessler is a professor and chairman of the Department of Computer Science of the University of Utah. In the early 1990s, he founded the Center for Software Science, a Utah Center of Excellence. He recently completed seven years as coeditor in chief of the *International Journal of Lisp and Symbolic Computation*. His research interests include agents, software engineering, distributed systems, and visual programming. Contact him at kessler@cs.utah.edu.



Ward Cunningham is a founder and owner of Cunningham & Cunningham, a consulting firm. Among his contributions to the developing practice of object-oriented programming, he created the CRC design method, which helps teams find core objects for their programs. He is active in the Hillside Group and has served as program chair of its Pattern Languages of Programs Conference. Contact him at ward@c2.com.

Ron Jeffries is an independent consultant. He has been developing software since 1961, when he accidentally got a summer job at the Strategic Air Command headquarters, where they accidentally gave him a Fortran manual. He and his teams have built operating systems, language compilers, relational and set-theoretic database systems, and 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. Contact him at ronjeffries@acm.org.

References

1. L.L. Constantine, *Constantine on Peopleware*, Yourdon Press, Englewood Cliffs, N.J., 1995.
2. J.T. Nosek, “The Case for Collaborative Programming,” *Comm. ACM*, Vol. 41, No. 3, 1998, pp. 105–108.
3. J.O. Coplien, “A Development Process Generative Pattern Language,” *Pattern Languages of Program Design*, J.O. Coplien and D.C. Schmidt, eds., Addison-Wesley, Reading, Mass., 1995, pp. 183–237.
4. N.V. Flor and E.L. Hutchins, “Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming during Perfective Software Maintenance,” *Proc. Empirical Studies of Programmers: Fourth Work-*