Pair Programming Benefits

PairProgramming, while not a new concept, is the source of some lively discussion. Please carry that on here.

For some qualitative support of the effectiveness of PairProgramming, see AcmOnCollaborativeProgramming.

PairProgramming yields the following benefits, roughly ordered from largest benefit to smallest:

- **Increased discipline.** Pairing partners are more likely to "do the right thing" and are less likely to take long breaks. One unexpected result: TheExtremeProgrammingDiet
- **Better code.** Pairing partners are less likely to go down GopherHoles and BlindAlleys and tend to come up with higher quality designs.
- **Resilient flow.** Pairing leads to a different kind of MentalStateCalledFlow than programming alone, but it does lead to flow. Pairing flow happens more quickly: one programmer asks the other, "What were we working on?" Pairing flow is also more resilient to interruptions: one programmer deals with the interruption while the other keeps working.
- **Improved morale.** Pair programming, done well, is much more enjoyable than programming alone, done well. (On the other hand, pair programming done poorly is much less enjoyable than programming alone done poorly.)
- **Collective code ownership.** When everyone on a project is PairProgramming, and pairs rotate frequently, everybody gains a working knowledge of the entire codebase.
- **Mentoring.** Everyone, even junior programmers, has knowledge that others don't. Pair programming is a painless way of spreading that knowledge.
- **Team cohesion.** People get to know each other more quickly when pair programming. Pair programming may encourage team jelling.
- **Fewer interruptions.** People are more reluctant to interrupt a pair than they are to interrupt someone working alone.
- **?** When you're looking for another job, you'll have references that know your skills well.

An oft-quoted benefit of PairProgramming is that the pairing partner will catch coding mistakes. If you're using TestDrivenDevelopment, this benefit is a RedHerring: the mistakes the pairing partner catches are generally the trivial kind that would quickly be caught by the compiler or tests anyway. If you're not using TestDrivenDevelopment, this benefit may be more of a factor.

Also, LaurieWilliams famously demonstrated improved quality for slightly more effort in her *The Collaborative Software Process* paper (available at http://collaboration.csc.ncsu.edu/laurie/Papers/dissertation.pdf) *[Is this the right one?]*. A result that nobody else has been able to recreate since. Laurie is also the author of a book on the benefits of Pair Programming.....

I've tried PairProgramming only a few times, but those times were far more productive than any I've had by myself. It's great fun, it's more efficient, and it helps me keep the right time and code perspective. I only wish I could do more of it. Since I'm the only programmer where I work, I've had to resort to VirtualPairProgramming -- ShaeErisson

I had a client blather about how they were using XP as their development environment -- yeah, except for all the regular practises like writing tests before code, pair programming, short iterations, et al. Oh, well. One time I had an opportunity to work with another guy who had some "free" time (well, not free to the client, of course). I can't recall exactly what we were working on, but it was complicated enough that I wanted somebody else to backstop me on the design and implementation. In a single two hour session we knocked out a subsystem that would have taken me all day had I been working alone. At the end of it we looked at the results and neither he nor I could tell whose ideas and code were whose. Yeow! It worked pretty durn well **that** time. -- MartySchrader

Moved from PairProgramming: Regarding the benefits of PairProgramming, when i have the opportunity to promote XP practices (usually to mgmt during a presentation on process), i have been presenting the following supporting argument (among others), linked to some research:

1. Projects with high quality (attention to quality) have lower schedule pressure:

- Products with the lowest defect counts have the shortest schedules (Jones 1991, Applied Software Measurement).
- Poor quality is the most common reasons for schedule overruns (Jones 1994, 4000 projects, Assessment and Control of Software Risks).

2. Higher quality comes through culture/values/process, inspection, testing, early feedback, refactoring, and so forth.

3. Regarding inspection and quality and schedule, inspection is a powerful tool:

- Inspections have been found to produce net schedule savings of 10 to 30 percent (Gilb and Graham 1993, Software Inspection).
- Each hour spent on inspections avoided an average of 33 hours of maintenance (Russell 1991, IEEE Software).
- Inspections were up to 20 times more efficient than testing (Russell 1991, IEEE Software).

4. PairProgramming, in part, is a way to do effective, real-time, responsive inspection, which supports higher quality, which supports reduced schedule pressure.

-- CraigLarman

JoeDavison kindly added to and incited me to improve my model for DayCare to pick up additional factors. One is the CornFieldEffect, which says experts work better together than separately (just as corn gives greater yield when grown in proximity rather than isolation). Another two are the mentoring effects - better design and faster training, which apply when the people have different amounts of expertise. To me, the additional factors for DayCare provide a bit of a model to explain some of the added productivity in PairProgramming. -- AlistairCockburn

One of the other benefits of PairProgramming is its effect on code/design/produced object quality. It is amazing how many obvious but unnoticed defects become noticed by another person watching over your shoulder. In addition, the intimacy of this method makes it possible for defects to be exposed in a non-judgemental manner.

The first time I heard of this paradigm being used in a formal, organizational sense was in DeMarco & Lister's PeopleWare, where they mention it as a benefit and give the example of Whitesmith's (an old compiler company founded by PjPlauger) as a place where it was practiced. *[I believe this is actually in ConstantineOnPeopleware; see http://www.cs.utah.edu/~bwilliam/proposal.htm for a reference. -- PaulChisholm]* Anything earlier? -- FrankAdrian [hounds are coupled together for hunting (and a pack of N hounds will be referred to as "N/2 couple"), so if you're willing to accept non-humans, the practice is at least centuries, and probably millennia, old. A fox or hare that's backtracked and

crossed a stream is probably just as annoying as a bug that doesn't manifest until the stack's unwound and execution is deep in some other module...]

Noel Morris and I used this approach in 1965, writing programs for CTSS. It is mentioned in my note, "The Evolution of My Thinking," http://www.best.com/~thvv/evolution.html. [404] -- TomVanVleck

One of the rules of the ChryslerComprehensiveCompensation team is that all production code be written with a partner. As a testimonial, in the last six months before launching, the only code that caused problems was code written solo. -- KentBeck

{RonJeffries writes this about part of ExtremeSoftware as practiced at Chrysler ... http://www.xprogramming.com/Practices/PracPairs.html)

Copied this from comp.software.extreme-programming:

> Management problems on a software development group scale superlinearly with the size of the group. Two or three people don't need any management; they can just sit next to each other and talk. Somewhere about four or five, it starts to get helpful to have a boss who plans things. But it isn't hard to manage a group of five or six people, lots of people know how to do it. In fact, it isn't that hard up to about 10, but then it starts getting hard. There aren't that many people who can manage a group of 15 effectively.
>
> Pair programming has the effect of cutting your group size in half, but making each developer twice as good. Instead of 16 lines of development, you have 8, but each is going twice as fast. This makes integration and planning easier, and reduces the chances of conflicts between programmers.
>
> This is an interesting idea, but I don't know whether it holds up in practice. When proponents of pair programming say what they like about it, they never mention this.
>
> - RalphJohnson

As I see it, there are a few sides to the Mentoring benefit at the top of this page. These may be worth spelling out in the hope that they can be used to persuade traditional project managers to trial pair programming:

- There are many types of knowledge that can be shared - e.g. technical, application domain, design and application source code knowledge.
- More widely distributed knowledge enables more flexibility in assigning people to tasks. It decreases the likelihood that one person will be on the critical path while many people with less essential skills are assigned to peripheral tasks. This means the team can go faster.
- Knowledge can be flowing both ways in a single pairing session. For instance, one person may have some technical skills to share while the other may have more domain knowledge or experience with the source code.
- Distributed knowledge reduces the damage if a critical member of the team leaves or is hit by the proverbial bus.
- Some tasks require experience in a wide variety of specialist technologies. A pair in which each member has experience in a required technology, but in which neither has all the required skills will be able to complete the task in MUCH less time than either could alone.

-- BruceCropley

See: PairProgramming

CategoryPairProgramming

Last edit January 31, 2006, See github about remodeling.