The Costs and Benefits of Pair Programming

Alistair Cockburn
Humans and Technology
7691 Dell Rd
Salt Lake City, UT 84121, USA
arc@acm.org 801.947.9277

Laurie Williams
University of Utah Computer Science
50 S. Central Campus #3190
Salt Lake City, UT 84112, USA
lwilliam@cs.utah.edu 435.649.7931

"Only if the various principles - names, definitions, intimations and perceptions - are laboriously tested and rubbed one against the other in a reconciliatory tone, without ill will during the discussion, only then will insight and reason radiate forth in each case, and achieve what is for man the highest possible force..."
-Plato

"Knowledge is commonly socially constructed, through collaborative efforts toward shared objectives or by dialogues and challenges brought about by differences in persons' perspectives." -Salomon [1]

ABSTRACT

Pair or collaborative programming is where two programmers develop software side by side at one computer. Using interviews and controlled experiments, the authors investigated the costs and benefits of pair programming. They found that for a development-time cost of about 15%, pair programming improves design quality, reduces defects, reduces staffing risk, enhances technical skills, improves team communications *and* is considered more enjoyable at statistically significant levels.

Keywords

pair programming, collaborative programming, extreme programming, code reviews, people factors, collaborative programming

1 INTRODUCTION

When pair programming, two programmers work collaboratively on the same algorithm, design or programming task, sitting side by side at one computer. This practice has been nominated several times in the last decades as an improved way of developing software [2] [3].

However, convention speaks against having two people work together to develop code -- having "two do the work of one", as some people see it.

- Managers view programmers as a scarce resource, and are reluctant to "waste" such by doubling the number of people needed to develop a piece of code.
- Programming has traditionally been taught and practiced as a solitary activity.
- Many experienced programmers are very reluctant to program with another person. Some say their code is "personal," or that another person would only slow them down. Others say working with a partner will cause trouble coordinating work times or code versions.

At the same time:

- Several well-respected programmers prefer working in pairs, making it their preferred programming style [2] [3].
- Seasoned pair programmers describe working in pairs as "more than twice as fast" [3].
- Qualitative evidence suggests the resulting design is better, resulting in simpler code, easier to extend.
- Even relative novices contribute to an expert's programming, according to interviews.

This raises some provocative questions. Is pair programming really more effective than solo programming? What are the economics? What about the people factor - enjoyment on the job?

Based on recent interest in pair programming, the authors examined interview and experimental data to understand the costs and benefits of practice. This paper presents the results of that

investigation. Previous publications [4] [5] [6] have demonstrated that pair programming is beneficial. The purpose of this paper is to re-examine these results and to further explain why pair programming is beneficial.

2 A PROJECT EXPERIENCE

The following excerpt comes from an experienced programmer describing his organization's first venture into pair programming. It reveals many features of the pair programming experience, as will be discussed in this paper.

> In early December my team began a high-risk activity: it involved touching just about every file, merging the code together, and trying to keep everything working. Furthermore, one tree involved fairly deep rearchitecture. So the merge would be composed of both utter tedium and massive thought effort.
>
> The staff agreed with my points that pair programming:
>
> - Should significantly reduce the risk of subtle errors that would make debugging excruciating;
>
> - Would give us a much broader code review than we'd ever had; and
>
> - Would provide an opportunity to communicate knowledge between coders.
>
> For the first few weeks, things didn't work out as envisioned. Instead of doing side-by-side pair programming, the first few people did what I called "partner programming": they coded individually for awhile, and then reviewed the changes with their partner before checking-in the modifications. They reported that they were catching errors early. This was encouraging, but I was disappointed that they weren't working together consistently.
>
> But about four months into the merge, I began to notice that things were changing. One pair in particular spent their whole day together, doing honest-to-goodness pair programming, and the other two pairs were getting much closer to that ideal. In discussions it was clear that they knew why the change was happening. It simply worked better!
>
> They discovered that it took longer to work independently and then review the changes, since the review process involved teaching your partner everything you had learned in making the changes. And that took almost as long as making the changes to begin with. By working together they could avoid "doing it twice", the coding went faster due to the two-brain effect, and they were much more confident in the correctness of the results.
>
> When we finally made it to the first checkpoint application, we zoomed through QA with hardly a hitch. Everyone, myself included, was amazed that it didn't take weeks to debug, especially given that one of the trees had recently spent SIX WEEKS in QA hell. It was obvious that the pairs had dramatically reduced the defect rate.
>
> As the merge progressed the pairs worked together even more closely.
>
> ... As each subsystem was complete the pairs would get rearranged based on knowledge of the next task. This slowed things down because the new pairs would have to spend time getting in phase with one another before working effectively. But by August the pairs were fairly well cemented, to the point where they would routinely speak for each other in our twice-weekly team meetings.
>
> Subsequent releases, both internal and external, went smoothly and we rarely hit massive show-stopper bugs. The continual review caught many serious issues midstream, including some major design problems that hadn't been noticed before.
>
> I wasn't involved in a pair until fairly late in the game. Once my partner and I synchronized our brains, it was a great experience.
>
> He was relatively junior, but he asked the right questions and, by struggling for answers, we usually forced ourselves to discover the best solution to each problem.
>
> ... The team members decided to do it for themselves. None of my pontification had as much effect as experience.

3 INVESTIGATIVE PATHS

We explore eight paths of software engineering and organizational effectiveness. Surprisingly, all paths point to pair programming. These investigative paths are briefly described:

Economics. A recent controlled experiment [4] found only a small development cost for adding the second person. However, the resulting code also had fewer defects. The defect removal savings should more than offsets the development cost increase.

Satisfaction. People working in pairs found the experience more enjoyable than working alone.

Design quality. The study [4] also found that the pairs produced shorter programs than their solo peers, indicating superior designs. Interviewees

made the same comments.

Continuous Reviews. Pair programming's shoulder-to-shoulder technique serves as a continual design and code review, leading to most efficient defect removal rates.

Problem solving. Interview participants constantly refer to the team's ability to solve "impossible" problems faster.

Learning. Pair programmers repeatedly cite how much they learn from each other.

Team Building and Communication. Interview participants describe that people learn to discuss and work together. This improves team communication and effectiveness.

Staff and Project Management. Since multiple people have familiarity with each piece of code, pair programming reduces staff-loss risk.

In this paper, each of these investigative paths will be further discussed -- reviewing the supporting statistical and interview data and highlighting the costs and the benefits.

4 ECONOMICS

The affordability of pair programming is a key issue. If it is much more expensive, managers simply will not permit it. Skeptics assume that incorporating pair programming will double code development expenses and critical manpower needs. Along with code development costs, however, other expenses, such as quality assurance and field support costs must also be considered. IBM reported spending about $250 million repairing and reinstalling fixes to 30,000 customer-reported problems [7]. That is over $8,000 for each defect!

In 1999, a controlled experiment run by the second author at the University of Utah investigated the economics of pair programming. Advanced undergraduates in a Software Engineering course participated in the experiment. One third of the class coded class projects as they had for years – by themselves. The rest of the class completed their projects with a collaborative partner. The results of how much time the students spent on the assignments are shown below in Figure 1. After the initial adjustment period in the first program (the "jelling" assignment), together the pairs only spent about 15% more time on the program than the individuals [4]. Development costs certainly do not double with pair programming!

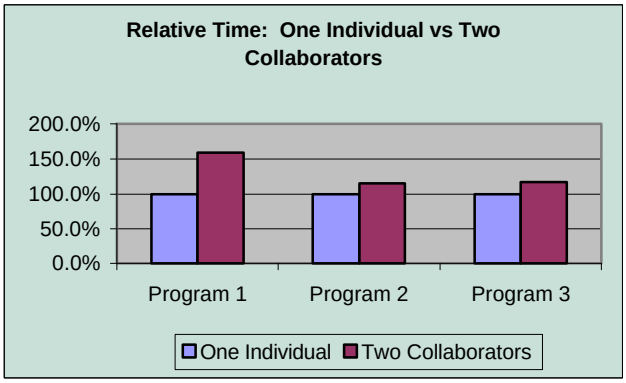

Figure 1: Programmer Time

Significantly, the resulting code has about 15% fewer defects [4]. (These results are statistically significant.) Figure 2 shows the post-development test cases the students passed for each program – essentially the percentage of the instructor's test cases passed.

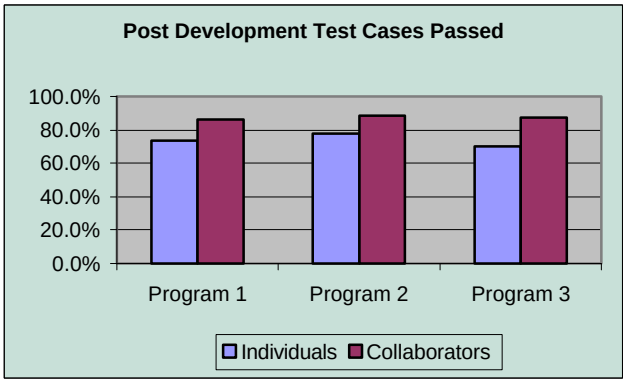

Figure 2: Code Defects

The initial 15% increase in code development expense is recovered in the reduction in defects, as the following example illustrates. Let a program of 50,000 lines of code (LOC) be developed by a group of individual programmers and by a group of collaborative programmers. At a typical rate of 50 LOC/hour, the individuals will develop this code in 1000 hours. It will take the pairs 15% longer or 1150 hours -- a cost of 150 hours. Based on representative statistics reported in [7], programmers inject 100 defects per thousand lines of code. A thorough development process removes approximately

70% of these defects. Therefore, the individuals would be expected to have 1,500 defects remaining in their program; collaborators would have 15% less or 1,275 – 225 less defects.

In some organizations, developers' code is passed to a test or quality assurance department, which finds and fixes many of the remaining defects. Typically, in systems test it takes between four [7] and 16 [8] hours per defect. Using a fairly conservative factor of 10 hours/defect, if testing finds these "extra" 225 defects they will spend 2,250 hours – fifteen times more than the collaborators "extra" 150 hours!

If the program is sent directly to a customer instead of a test department, pair programming is even more favorable. Industry data reports that between 33 and 88 hours are spent on each defect found in the field [7]. Using a fairly conservative factor of 40 hours/defect, if the customer is plagued by these "extra" 225 defects, field support will spend 9,000 hours – sixty times more than the collaborators "extra" time!

Doubtlessly, pair programming can be justified on purely economic grounds. But there are more aspects to consider.

5 SATISFACTION

If pair programming is not satisfying, programmers won't practice it.

Many programmers are initially skeptical, even resistant, to programming with a partner. It takes the conditioned solitary programmer out of their "comfort zone." One programmer wrote,

> "The adjustment period from solo programming to collaborative programming was like eating a hot pepper. The first time you try it, you might not like it because you are not used to it. However, the more you eat it, the more you like it."

In statistically significant results, pair programming teams who had earlier programmed alone reported that they enjoyed pair programming more and that they were more confident in their programs than when they programmed alone (as the defect rates show they are entitled to be). The graph (Figure 3) shows results of anonymous surveys of professional pair programmers and of student pair programmers at the University of Utah. Most of the programmers enjoyed programming collaboratively.

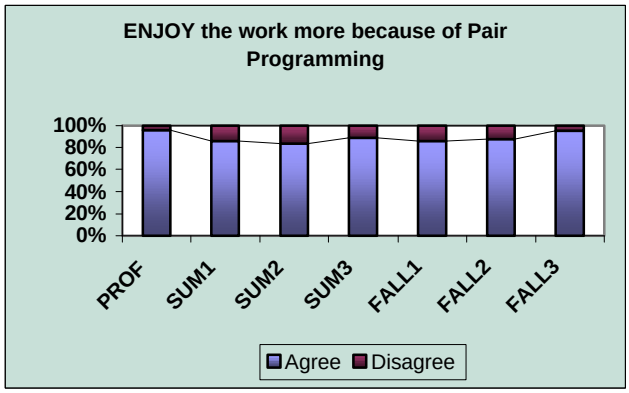

Figure 3: Pair Enjoyment

A programmer comments,

> "It is psychologically soothing to be sure that that no major mistakes had been made . . . I find it reassuring to know that [partner] is constantly reviewing my code while I drive. I can be sure I had done a good job if someone else I trust had been watching and approved."

Another says,

> "It's nice to celebrate with somebody when something works."

Students prefer the 15% overhead

In the study previously discussed, the class was divided into two groups: an "individual" group in which solo programmers did all programming, and a "collaborative" group in which all programming was done in pairs. Each programming assignment cycle, the individuals were given one program to complete while the pairs were given two programs to complete.

After several programming cycles, one pair complained that this arrangement was unfair. They felt they had to work harder than the individuals during each cycle. The instructor suggested that the students split up and work as solo programmers as part of the "individual" group so the would no longer feel they were being unjustly overworked. Both students rejected this offer almost instantaneously. They did not complain about the "unjustness" of the additional workload again.

We feel that this is a strong indicator of the satisfaction of pair programming.

6 DESIGN QUALITY

The following comments came from a team lead who had never heard of pair programming. He described why all of his designer-programmers working together at the same terminal.

> As we proceeded with our work, I started to notice that one team consistently produced designs of distinctly better quality than the others. I asked them about this.
>
> They said that they had taken to working together, on both the design and the programming. They found that their designs and programs were better this way. I agreed with them and made it standard for all teams to work in pairs. The design quality is now better.
>
> [from the interview files of A. Cockburn]

In 1991 Nick Flor, a masters student of Cognitive Science, reported on distributed cognition in a collaborative programming pair he studied. Distributed cognition is a field of cognitive science based on the belief that "Anyone who has closely observed the practices of cognition is struck by the fact that the "mind" rarely works alone. The intelligences revealed through these practices are distributed – across minds, persons, and the symbolic and physical environment [1]."

Flor recorded via video and audiotape the exchanges of two programmers working together on a software maintenance task. In [9], he correlated specific verbal and non-verbal behaviors of the two under study with known distributed cognition theories. One of these theories is "Searching Through Larger Spaces of Alternatives."

> "A system with multiple actors possesses greater potential for the generation of more diverse plans for at least three reasons: (1) the actors bring different prior experiences to the task; (2) they may have different access to task relevant information; (3) they stand in different relationships to the problem by virtue of their functional roles. . . An important consequence of the attempt to share goals and plans is that when they are in conflict, the programmers must overtly negotiate a shared course of action. In doing so, they explore a larger number of alternatives than a single programmer alone might do. This reduces the chances of selecting a bad plan. [9]"

A pair programmer's description matches Flor's:

> We often came up with different ideas about how the design should go and the result of arguing over which one was better often led to a truly superior hybrid design.

In the quantitative study at the University of Utah, the pairs not only completed their programs with superior quality, but they consistently implemented the same functionality as the individuals in fewer lines of code. Details are shown in Figure 4. We believe this is an indication that the pairs had better designs.

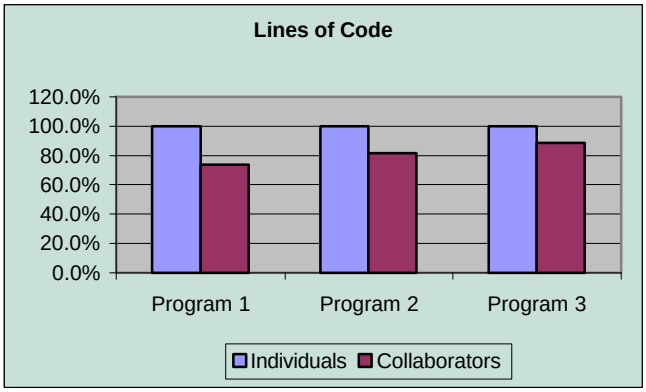

Figure 4

7 CONTINUOUS REVIEWS

Inspections were introduced more than twenty years ago as a cost-effective means of detecting and removing defects from software. Results [10] from empirical studies consistently profess the effectiveness of reviews. Even still, most programmers do not find inspections enjoyable or satisfying. As a result, inspections are often not done if not mandated, and many inspections are held with unprepared inspectors.

> "Despite a consistent stream of positive findings over 20 years, industry adoption of inspection appears to remain quite low, although no definite data exists. For example, an informal USENET survey we conducted found that 80% of 90 respondents practiced inspection irregularly or not at all. [11]"

The theory on why inspections are effective is based on the prominent knowledge that the earlier a defect is found in a product, the cheaper it is to fix the defect. Many sources, including [8] state that it is ten times more expensive to remove a defect for each additional process step.

This exponential cost growth is easy to understand. During an inspection, a programmer might say, "The 'if' statement starting on line 450

should also have an 'else' clause." The programmer marks the listing and promptly makes the change at their computer. However, if the software is already in the field, a customer makes an irate call to the software shop, "It's Christmas Eve, and all of my cash registers just crashed. I can't sell anything!"

In the first case, the programmer looks directly at the problem that was just identified to them. In the second case, the field maintenance team must work to translate the symptom (all the cash registered crashed) back to the problem (which exact line(s) of code caused the crash). It is easy to see why the translation of the symptom back to the problem costs exponentially more than direct problem identification. With pair programming, this problem identification occurs on a minute-by-minute basis. These continual reviews not only outperform formal reviews in their defect removal speed, but they also eliminate the programmer's distaste for reviews.

The following, sardonically worded description from a senior (originally skeptical) programmer shows how pairing with even a novice contributed to his programming.

> 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.
>
> [-Ron Jeffries, from [4]]

A final benefit of code reviews is that reviewers learn new coding idioms and language features, as well as more about the system.

> The continuous reviews of collaborative programming create a unique educational capability, whereby the pairs are endlessly learning from each other. "Indeed, review has a unique educational capability: The process of analyzing and critiquing software artifacts produced by others is a potent method for learning about languages, design techniques, application domains, and so forth. [11]."

In keeping with the known characteristics of code reviews, we find practitioners citing:

- Mistakes are found as they are entered, saving even the cost of compilation, and providing the economic benefit of early defect identification and removal.
- Coding standards are followed more accurately with the peer pressure to do so.
- Team members learn to talk together and work together.

8 PROBLEM SOLVING

> There were times we felt that we would have given up except that we "tag teamed." I'd be on the ropes and I'd describe the problem in such a way that he had a valuable insight. Then he'd fight on as long as he could and stop . . . then I'd have an insight . . . and so on. I suppose others would call it brainstorming, but it feels different to me.
>
> [-David Wagstaff, Salt Lake City]

Pair relaying is our name for the effect Wagstaff describes. Indeed, pairs consistently report that they solve problems faster, and that it is different from improving design quality, or detecting typing errors, or brainstorming. By "problem solving", we refer to when the two are puzzled as to why something doesn't work as expected, or simply can't figure out how to go forward.

In interviews and in off-hand remarks, practitioners describe contributing their knowledge to the best of their abilities, in turn. They share their knowledge and energy (and also brainstorming) in turn, chipping steadily away at the problem.

Combining brainstorming and pair relaying is powerful. One seasoned programmer wrote,

> I have found that, after working with a partner, if I go back to working alone, it is like part of my mind is gone. I find myself getting confused about things.

9 LEARNING

Knowledge is constantly being passed between partners, from tool usage tips (even the mouse), to programming language rules, design and programming idioms, and overall design skill.

Learning happens in a very tight apprenticeship

mode. The partners take turns being the teacher and the taught, from moment to moment. Even unspoken skills and habits cross partners.

Line of sight learning in apprenticeships

[12] discusses apprenticeship case studies. These studies range from tailors to flag signalmen in the U.S. Navy to butchers in modern supermarkets.

The book is subtitled "legitimate peripheral participation" to highlight three key aspects of apprenticeship: that the novice actively participates; that the novice have legitimate work to do; and that the novice works on the periphery, steadily moving toward some higher rank. The novice's work is initially simple and non-critical. Later work is more critical.

One of the most distinctive characteristics they note of successful apprenticeship environments is that the novice work in a "line of sight" of the expert, that expertise is transmitted, in part, through the ongoing visual (and auditory) field. They describe successful apprenticeship learning in both tailors and Navy signalmen where “line of sight” is available. The beginner explicitly picks up skills from hearing and/or seeing the expert.

The most interesting of the case studies, for this discussion, is that of the butchers in supermarkets. They did not have line of sight access to their local expert. The beginners were given simple cuts to perform, but did not have a way to learn how to do more difficult cuts, which were being done by the senior butcher in another room. Lave and Wenger present this as a situation in which apprenticeship learning does not happen.

It should be obvious that most project programming environments match the butcher situation, not the tailor or signalmen situation. Indeed, we have found it extremely difficult to set up programming environments in which line-of-sight and line-of-hearing from expert to novice can be accommodated. The novice programmer generally sits in their workspace working on simple code; the expert sits in their own workspace creating complex code and making architectural decisions. Pair programming makes a better apprenticeship situation.

Expert In Earshot

The first author obtained a project management pattern from a set of workshops with 10 senior project managers. The full pattern, listed in the Appendix, includes examples and caveats; it is summarized as:

> Use *Expert In Earshot* when novices are not learning good techniques and habits very well, but you don't want to turn the expert into a full-time teacher. Put the expert or leader in the same workspace as the more novice workers, so that the novices can learn by watching and listening, while the expert does his or her usual work. Novices will pick up expertise and (hopefully good) habits from the expert. (Your expert will be disturbed more often, so you will have to set up ways to create personal quiet time.)

Note the overlap with the apprenticeship studies. It is significant that this pattern had to be accepted by all 10 senior project managers, since they were going to apply it within their company.

Pair programming is an example of both *Expert In Earshot* and of legitimate peripheral participation, with line-of-sight/hearing access. We can expect, therefore, that the learning that occurs in pair programming is more significant than merely learning new tool usage or programming language idioms. This expectation matches off-hand reports from practitioners.

Statistical confirmation

Pair programming was used exclusively in a web programming class at the University of Utah, taught by the second author. The class consisted of 20 juniors and seniors, familiar with programming, but not with web programming languages and tools. The majority of the students had only used WYSIWYG web page editors prior to taking the class. During the eleven-week semester, the students learned advanced HTML, JavaScript, VBScript, Active Server Page Scripting, Microsoft Access/SQL and some ActiveX commands. In many cases, they had to intertwine statements from all these languages in one program listing.

Unusual for such students, they produced their programs with minimal questions of the teaching staff. The students were queried about the reasons for their independence in an anonymous

survey on the last day of class.

- 74% wrote “between my partner and me, we could figure everything out.”
- 84% of the class agreed with the statement “I learned Active Server Pages faster and better because I was always working with a partner.”

We would attribute part of that result to enhanced problem solving in pairs, as described above, and part to enhanced pair learning.

10 TEAM BUILDING&COMMUNICATION

> When I arrived, I saw a disheartening sight: Bill didn't have a team; he had a random collection of six bright, talented individuals who didn't work together. They didn't sit near each other. They didn't even like each other. Here is a scene from a weekly staff meeting
>
> 'Let's talk about pair programming. (benefits of pair programming enumerated) <pause> Therefore, pair programming is mandatory. All production code must be written with a partner present.'
>
> <An awkward silence descends. Furtive eye glances are exchanged.>
>
> 'I don't think that's going to work. What if I need to write code and my partner isn't available?'
>
> 'Then you find someone else. One of our goals is to spread the knowledge around.'
>
> 'What if there's no-one else around?'
>
> 'If I'm available, I'd be glad to work with you. If there really is nobody around, push your keyboard away and wait.'
>
> <Stunned silence >
>
> Some of the first paired sessions went smoothly. Other sessions were awkward. Communication was serial and parsimonious. I handheld these guys by becoming a third wheel. I encouraged the developers to think out loud (what Ward Cunningham calls reflective articulation). This did the trick. They actually began to work together, not just take turns coding.
>
> After about a week, I noticed a remarkable phenomenon. The developers were talking to each other. As people. You really would have to have been there at the beginning to appreciate this. Anyway, I noticed them having real conversations. And laughing. They actually began to enjoy and trust each other.
>
> Within several weeks, they became a real team."
>
> [from the interview files of A. Cockburn]

The Psychology of Computer Programming [13] and Peopleware [14], written 20 and 30 years ago, respectively, still have not been replaced on the topic of teamwork. More recently, eXtreme Programming [2], the Crystal Light methodologies [15] and Adaptive Software Engineering [16] have strengthened attention given to team building and communication. In [17], Cockburn goes farther, and argues that these are first-order project drivers, not side issues.

Pair programming contributes in three ways.

People learn to work together, as illustrated in the above quote. In the University of Utah study, none of the 14 pairs ran into an insurmountable personality clash. In industry, off-hand comments indicate personality clashes occasionally happen, perhaps because there is not sufficient pressure or motivation for the people to learn to work together. Interviews with people who have persevered, however, reveal a pattern similar to, but not as extreme as the one above. Often, by having to work together, people learn to work together.

Learning to work together means that the people on the team will share both problems and solutions more quickly, and be less likely to have hidden agendas from each other. Teamwork is enhanced.

If the pair can work together, then they learn ways to communicate more easily and they communicate more often. This raises the communication bandwidth and frequency within the project, increasing overall information flow within the team. Rotating partners increases the overall information flow farther.

All of these are likely to increase team effectiveness, and, indeed, this is what pair programming teams report. We know of no statistical study of these effects.

11 STAFF & PROJECT MANAGEMENT

Project management benefits from improved staff skills and reduced staff risk.

The company and the project team both benefit from the increased learning. Over the course of the project, the skills of the team members increase, in programming and in software design.

The risk from losing key programmers is reduced, because there are multiple people familiar with each part of the system. This is referred to as the "truck number" in some circles: "How many or few people would have to be hit by a truck (or quit) before the project is incapacitated?" The worst answer is "one." Having knowledge dispersed across the team increases the truck number, and project safety.

12 SUMMARY

The significant benefits of pair programming are that

- many mistakes get caught as they are being typed in rather than in QA test or in the field (continuous code reviews);
- the end defect content is statistically lower (continuous code reviews);
- the designs are better and code length shorter (ongoing brainstorming and pair relaying);
- the team solves problems faster (pair relaying);
- the people learn significantly more, about the system and about software development (line-of-sight learning);
- the project ends up with multiple people understanding each piece of the system;
- the people learn to work together and talk more often together, giving better information flow and team dynamics;
- people enjoy their work more.

The development cost for these benefits is not the 100% that might be expected, but is approximately 15%. This is repaid in shorter and less expensive testing, quality assurance, and field support.

REFERENCES

1. Salomon, G., *Distributed Cognitions: Psychological and educational considerations*. Learning in doing: Social, cognitive, and computational perspectives, ed. R. Pea and J.S. Brown. 1993, Cambridge: Cambridge University Press.
2. Constantine, L.L., *Constantine on Peopleware*. Yourdon Press Computing Series, ed. E. Yourdon. 1995, Englewood Cliffs, NJ: Yourdon Press.
3. Beck, K., *Extreme Programming Explained: Embrace Change*. 2000, Reading, Massachusetts: Addison-Wesley.
4. Williams, L., *et al.*, *Strengthening the Case for Pair-Programming*, in *IEEE Software*. submitted to IEEE Software. Online at http://www.cs.utah.edu/~lwilliam/Papers/ieeeSoftware.PDF
5. Williams, L.A. and R.R. Kessler. *The Collaborative Software Process*. in *International Conference on Software Engineering 2000*. submitted for consideration. Limerick, Ireland. Online at http://www.cs.utah.edu/~lwilliam/Papers/ICSE.pdf
6. Nosek, J.T., *The Case for Collaborative Programming*, in *Communications of the ACM*. 1998. p. 105-108.
7. Humphrey, W.S., *A Discipline for Software Engineering*. SEI Series in Software Engineering, ed. P. Freeman, Musa, John. 1995: Addison Wesley Longman, Inc.
8. Humphrey, W.S., *Introduction to the Personal Software Process*. 1997: Addison-Wesley.
9. Flor, N.V. and E.L. Hutchins. *Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Maintenance*. in *Empirical Studies of Programmers: Fourth Workshop*. 1991: Ablex Publishing Corporation.
10. Fagan, M.E., *Advances in software inspections to reduce errors in program development.* IBM Systems Journal, 1976. **15**: p. 182-211.
11. Johnson, P.M., *Reengineering Inspection: The Future of Formal Technical Review*, in *Communications of the ACM*. 1998. p. 49-52.
12. Lave, J. and E. Wenger, *Situated Learning: Legitimate peripheral participation*. 1991, New York, NY: Cambridge University Press.
13. Weinberg, G.M., *The Psychology of Computer Programming Silver Anniversary Edition*. 1998, New York: Dorset House Publishing.
14. DeMarco, T. and T. Lister, *Peopleware*. 1977, New York: Dorset House Publishers.
15. Cockburn, A., *Crystal "Clear": A human-powered software development methodology for small teams*, Addison-Wesley, 2001, in preparation. Online at http://members.aol.com/humansandt/crystal/clear.
16. Highsmith, J., *Adaptive Software Development*, Dorset House, 1999.
17. Cockburn, A., *Characterizing People as Non-Linear, First-Order Components in Software Development*, in *International Conference on Software Engineering 2000*. submitted for consideration. Limerick, Ireland.. Online as Humans and Technology Technical Report, TR 99.05, http://members.aol.com/humansandt/papers/ nonlinear/nonlinear.htm.

APPENDIX: THE *"EXPERT IN EARSHOT"* PROJECT MANAGEMENT PATTERN

(slightly abbreviated from http://members.aol.com/humansandt/papers/expertinearshot.htm)

Thumbnail
Novices have a hard time developing good habits on their own, so...
Keep an expert within their hearing distance.

Indications
(1) Novices are not learning good techniques and habits very well.
(2) Working on the same project, your experts have private offices and your novices use a shared workspace.

Counter indications
(1) Regulations prevent putting expert and novices in a shared workspace.
(2) The expert has poor communication skills or work habits you don't want replicated!
(3) The expert spends most of her or his time in activities that would disturb the work of any novices within earshot, such as talking on the phone about other matters.

Forces
(1) You need everyone to get work done, both expert and novices.
(2) You want the novices to learn, and the expert has good habits worth learning.
(3) You can afford for the expert to be disturbed a bit more if the novices will learn some good habits.
But...
(1) You don't want to turn the expert into a full-time teacher.
(2) People hesitate to disturb the boss or expert with a phone call or knock on the door.

Do This
Put the expert or leader in the same workspace as the more novice workers, so that the novices can learn by watching and listening, while the expert does his or her usual work.

Resulting Context
(1) Novices will pick up expertise and (hopefully good) habits from the expert.
(2) Your expert will be disturbed more often, so you or he or she will have to set up ways or conventions to create personal quiet time.
(3) You and the expert have to watch that the novices do not simply delegate their problems to the expert.
(4) You will have more people in the room.

Overdose Effect
(1) Too many questions will lower the expert's productivity too much.
(2) Too many people in the same room will create too many conversations, making it hard to concentrate.

Related Patterns
Training:Day Care says "Your experts are spending all their time mentoring novices, so ... Put one expert in charge of all the novices; let the others develop the system.") [CoSOOP]. This covers the dangers of having the expert try to teach while designing. In *Expert In Earshot*, the expert is not responsible for teaching the novices. The situation is only set up so that the novices can see and hear how the expert works, in accordance with the principles. [15]

Pair Programming is a non-conflicting possible partner pattern to *Expert In Earshot*. It can be used to bring an expert within earshot of one person (the other person in the pair), or of many people - all the rest of the people in the workspace.

Examples
(1) When Thomas J. Watson, Jr., ex-CEO of IBM, went from being an aviator and playboy to a serious businessman, his father, then CEO of IBM, assigned him to sit at the corner of the senior-VP's desk for six months. For those six months, he was to do nothing but watch and listen to how this successful executive ordered his days and handled people. This is an unusual but very clear example of *Expert In Earshot*.

(2) A team leader given four junior designers to design a graphics workstation, was also given a private office. After a few weeks, he felt uncomfortable with the distance to his team, and moved his desk to the floor with the other designers. Although the distractions were great and his main focus was not teaching the other designers, he was able to discuss with them on a timely and casual basis. They became more capable, eventually reducing the time he had to spend with them and giving them skills for their next project.

(3) The lead programmer worked in a room with six novice programmers. He had two bad habits: he scoffed at the idea of doing design in an orderly way, and instead of talking to the other programmers about how to make a good design or program, he would change their code in the middle of the night. They never knew in the morning if their program was the same as when they left it. After several months, the novices both produced bad designs and refused to design carefully. His heroic attitude had become their ideal.
When he left the project, another consultant took his place, also sharing the room. He deliberately discussed designs from his desk, so the others could overhear. After a few months, three of the novices started talking and drawing designs, and soon became skilled in designing as well as programming.