Extreme Programming Refactored: The Case Against XP

MATT STEPHENS AND DOUG ROSENBERG

Apress™

ISBN (pbk): 1-59059-096-1

Printed and bound in the United States of America 12345678910

Technical Reviewer: Alex Chaffee

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Tracy Brown

Copy Editor: Nicole LeClerc

Production Manager: Kari Brooks

Compositor and Proofreader: Kinetic Publishing Services, LLC

Indexer: Ron Strauss

Cover Designer and Artist: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

CHAPTER 6

Pair Programming (Dear Uncle Joe, My Pair Programmer Has Halitosis)

Your Pair Will Hold Your Hand
(Sing to the tune of "I Want to Hold Your Hand" by The Beatles)

When you're coding something
That you don't understand

You don't
Have to worry

Your pair will hold your hand
Your pair will hold your hand
Your pair will hold your hand

And when you're coding you feel happy inside
The joy of coding is just one you can't hide

People say
We need requirements
They always make a fuss
We think
That requirements
Should be in C Plus Plus
Should be in C Plus Plus

And when you're coding you feel happy inside
The joy of coding is just one you can't hide

When you're unit testing
And you find a bug
You don't have to feel bad
Your pair will give you a hug

Your pair will give you a hug
And when you're coding you feel happy inside
The joy of coding is just one you can't hide

"The only constraint that XP puts on you is that any production code has be [sic] *written by a pair.* Your preferences and comfort do not supercede the delivery of quality to the project, or your parcitipation *[sic]* in the team."[1]

—Robert C. Martin

"In reality, it fosters a stinky noisy room and a headache."[2]

—Anonymous XPer

"I think maybe concentration is the enemy. Seriously. If you're working on something that is so complex that you actually need to concentrate, there's too much chance that it's too hard."[3]

—Ron Jeffries

Pair programming is one of the most controversial aspects of XP, and it presents some of the most amusing scenarios, some of which have even appeared in the Dilbert comic strip.[4] In this chapter we examine some of the claims that have been made about this practice, including those from Laurie Williams' book *Pair Programming Illuminated*. We also share with you some eye-opening real-world "eXPeriences" from our correspondents "in the field."

1. Robert C. Martin posting to the newsgroup `comp.object`, subject: "Pair Programming—Yuck!" October 28, 2001.
2. Anonymous XPer, see the "Voice of eXPerience: Tales from the Front Line" sidebar in Chapter 4.
3. Ron Jeffries, posting to the C2 Wiki page "Pair Programming Ergonomics," `http://c2.com/cgi-bin/wiki?PairProgrammingErgonomics`.
4. See `http://www.dilbert.com`.

Pair Programming Basics

The basic idea behind pair programming is that two heads are better than one—that a pair of programmers working together will produce a higher quality product than a single programmer. So, for production code, XP mandates that everyone on the team program in pairs. Pairing up is an effective way of catching errors and identifying ways to further simplify the code. To facilitate collective ownership of the code, XP encourages people to move around and switch partners frequently. It's kind of like square dancing: "Change your partner!" every couple of hours. And, of course, in XP all the pair programming goes on in a single room.

On the surface, pair programming sounds like a great idea. It's been well known since the ancient days of punched cards that having another programmer look at your code can help to catch bugs. In some circumstances it can undoubtedly work tremendously well. Just as obviously, you've got two programmers doing the work that one programmer was previously doing if everybody has to pair program. Some research studies claim there is no significant decrease in productivity (as we discuss later in this chapter).

Problems with pair programming can arise because of several factors, including

- Social dynamics
- Lack of privacy
- Lack of "quiet thinking time"
- Ergonomic issues

Pair programming is necessary in XP because it compensates for a couple of practices that XP shuns: up-front design and permanent design documentation. Coding in pairs can be seen as a compensatory practice that theoretically makes up for the fact that the programmers are (courageously) making up the design as they write the code.

There is certainly no problem with people working in pairs if they want to work that way. So it makes no sense to prohibit pair programming. However, changing it from a voluntary practice to a mandatory one makes quite a difference.

People You'd Least Like to Pair With (No. 48)

Pair programming is definitely not for everybody, as Doug relates here.

Having a number of years of programming experience, I place a pretty high value on peace, quiet, and space to think in. According to a study from IBM's Santa Teresa Laboratory,[5] putting programmers in private offices with doors that closed instead of cubicles resulted in a huge boost to productivity. So the idea of all the programmers in a big, noisy room seems like it would be a huge detriment to productivity. It would figure to drive many people nuts.

I've also worked with some incredibly talented programmers who would have been awful to pair with. One who comes to mind is hypoglycemic and has a tendency to get upset very easily when his blood sugar drops. He's a brilliant programmer who produces prodigious amounts of code that's always efficient and well structured. He and I worked very well together but I would never dream of sharing a desk and a keyboard with him. And I'd resist any process that wouldn't allow me, as a manager, to make use of his skills. So I would never mandate pair programming.

I Can't Code Alone . . . 'Cause I Need My Pair
(Sing to the tune of "Ticket to Ride" by The Beatles)

I guess I'd better go home
It's time to go play, yeah
That pair programmer I had
Called in sick today, yeah

I can't code alone
'Cause I've tried
Gotta have my pair by my side

I can't really write any code
'Cause I need my pair
'Cause I need my pair

When I'm refactoring code
He always sits right, always sits right by me
When we're runnin' our unit tests
He always sits right, always sits right by me

I can't code alone
'Cause I've tried
Gotta have my pair by my side

I can't really write any code
'Cause I need my pair
'Cause I need my pair

5. Gerald M. McCue, "IBM's Santa Teresa Laboratory—Architectural design for program development," *IBM Systems Journal* (`http://www.research.ibm.com/journal/sj/171/ibmsj1701C.pdf`), vol. 17, no. 1, 1978.

There's a Study That Proves My Point!

Proponents of pair programming often describe the "body of evidence" that surrounds pair programming's touted benefits. The most commonly cited study[6] was carried out by Laurie Williams, who also happens to be the coauthor of *Pair Programming Illuminated* and a contributor to PairProgramming.com (`http://pairprogramming.com`). The study is also described (and referred to repeatedly) in *Pair Programming Illuminated*.

The basic finding behind the study is that pair programming increases code quality by 15% but takes on average 15% longer. The conclusion then is that in exchange for a small drop in productivity, code quality is increased, which in turn saves time and money in later stages of the project.

The problem with this particular study, apart from the fact that it wasn't an independently run test, is that it was conducted in a university, using students rather than experienced professional programmers. Thus, the study is really only an examination of novice-novice pairing, which is actually the most unlikely and least desirable of combinations (we discuss programmer pairing combinations later in this chapter).

The pair programmers' performance was compared with the performance of some solo programmers who also were students (therefore also novices). This raises the issue that the comparison wasn't a fair one. In a real project, "solo" programmers don't really code alone; they're part of a team. Typically, designs are reviewed by senior engineers who sign off on them, prototypes are written quickly to get the design right before the production version is attempted, team leaders monitor their team closely (particularly the novice programmers), and so on. All these practices help to increase both productivity and code quality, yet none was taken into account.

The result is that Williams' study is academic at best; it isn't a true comparison of pair programming versus software development in the real world. And yet, as we mentioned, it's the most commonly cited "proof" that pair programming works.

In the next "Voice of eXPerience" sidebar we get one programmer's perspective, straight from the source.

---

6. Alistair Cockburn and Laurie Williams, "The Costs and Benefits of Pair Programming," `http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF`, 2002.

Voice of eXPerience: Pair Programming Social Dynamics

by "Rich Camden"

As in other VoXP segments, the reality of what happens to an XP team is often quite different from what XP preaches in theory. For example, XP recommends that the coach also pair programs (i.e., the coach doesn't get "special privileges" above the rest of the team). However, this description from Rich Camden is a good example of what can happen to an XP project when the human factor also plays a part.

The story you're about to read is true. Only the name of the author has been changed to protect the innocent.

Pair programming is the worst of the XP tenets. To recommend pair programming across the board for all developers involved in XP is to not understand the people/personality side of programming. Yes, there are those who will benefit from having another programmer constantly assisting and reviewing their work. But there are many more who will feel uncomfortable and restrained with another programmer watching everything they do.

Unfortunately, this tenet above all others tends to dampen creativity in XP. Many of the best programming gems come after you absorb yourself in a problem and do much deep thinking. This doesn't occur when you have another programmer sitting at the same keyboard.

The strangest part of XP for me is the social dynamics that occur as a result of making many people do things that feel unnatural or make them uncomfortable. In my experience, XP was a mandate from management, not a grassroots developer movement. In a rather short period of time, projects were shifted from comfortable cubicles to small rooms packed with workstations, tables, and people. It looks a lot like a call center, with developers shoulder to shoulder.

I joined my team after the change to XP had occurred. When I first joined the team, I was most interested in pairing and how it worked in real life. I read comments and articles on pairing at PairProgramming.com and at the C2 Wiki pages to prepare myself. I wasn't entirely negative to the idea. Unfortunately, what I read didn't prepare me for what was in store for me.

Joining the Team

On my first day of joining the team, I imagined that an enthusiastic XP developer would volunteer to pair with me and show me the ropes. It seemed that one of the team leads or coaches would be the best person. Instead, I was asked to simply watch the team and get my workstation prepared for development.

As I watched the team, I quickly realized that nobody was enthusiastic about pairing, including the team leads. During the stand-up meeting, developers were asked with whom they were going to pair. Now it seems quite impossible that they would know this until they could surmise who wasn't busy and then ask them to pair. Thus, in most cases the coach would assign pairs. I can't claim to know what people were thinking, but the looks on their faces didn't seem to say, "Oh, boy, I get to pair with ___ today!"

Right of Association

I watched each developer closely to understand the dynamics of the team I was joining. There were two team leads: the coach and the person acting as the customer representative. The coach never paired. This seemed to set the precedent that pairing was indeed undesirable, and if you were lucky enough to be a coach, you didn't have to do it. The second team lead was actively developing but rarely paired either. Again, this seemed to enforce the idea that if you had seniority and were doing important work, you didn't need to pair. This doesn't surprise me at all. Nobody wants someone looking over his shoulder day in and day out. What amazed me, however, was that the same people were actively encouraging the team to pair in classic "do as I say, not as I do" fashion.

That's not to say there was no pairing at all. When assigned a pair, developers would act in good faith and perform their job. The developers that I witnessed pairing normally had a buddy that they tended to prefer to pair with. Again, I wasn't surprised. People associate more with people they work well with.

The ability to choose associates (the right of association) is one of the keys to success. Any successful person uses this fact to her advantage. Forced association only ends in frustration and mediocrity. But even "pairing buddies" weren't actively engaged in pairing as described at PairProgramming.com. What tends to evolve is an arrangement where the two developers agree to work together to beat the system. Essentially, the unspoken agreement goes like this: "I'll work for a while and you daydream, then tomorrow you can type and I'll daydream."

Another interesting anomaly I noticed was that my pairs would always control the keyboard. Never in 5 months did it happen that my pair volunteered to have me type. Of course, if I asserted myself, they would allow me to type, but never did one choose to watch rather than type. The unwritten understanding I gathered was that typing is preferable to watching.

Pairing Fluidity Explained

The partner preferences didn't go unnoticed by the team coaches. As any XP enthusiast will tell you, this is undesirable. A surefire way to solve this would be for each coach to pair with a different person from time to time. Instead, the coaches would break up pairing buddies and assign them using criteria known only to them. This drove me to consider what would cause team members to avoid the fluid pairs approved by XP.

The key to understanding pairing culture is the XP task card. If you aren't familiar with XP, *task cards* describe small tasks that make up larger project goals driven by user stories. A task card has a completion estimate usually between 2 and 8 hours. Once they've finished a task, developers complete the card by writing down who completed the task and how long it took.

It's easy to see that the cards are basically a fine-grained record of exactly what transpires on a project. It takes software development and turns it into a textbook case of micromanagement techniques. Your objective is then to complete as many cards as possible with your name on them. This trumps the more productive desire to see the project succeed.

Your choices are then to either partner with someone you trust as a good developer or attempt to finish tasks individually. As you've already seen, this natural desire to partner is discouraged in XP. Thus, you're at a disadvantage if you're paired with an undesirable partner.

Who would make an undesirable partner? Someone new to the project would definitely slow you down. Every single module would require some explanation, not to mention a description of the business processes behind the application. This explained why my teammates were so reluctant to show me the ropes when I'd just joined the project. They'd most likely have to spend most of their time explaining the whys and hows of what they were doing, unless we had an unwritten pairing agreement, which only happens after some rapport has been built up over time.

It also explains the lack of fluidity in pairs. There's no advantage or benefit to the individual besides deflecting the constant berating of your coach that you should pair more often (which begs the question, if pairing is so great, why don't they do it too?). It's a classic catch-22. You're damned if you do and damned if you don't.

A Metaphor for Molecular Expansion

XP enthusiasts love metaphors. I find pairing to be a great metaphor for the physical characteristics of air molecules. Boyle's Law states: *If the temperature remains constant, the volume of a given mass of gas is inversely proportional to the absolute pressure.* For most of us, that's pretty clear, but I'll elaborate for the benefit of any XP proponents who might be reading. Boyle's Law tells us that if pressure is reduced, air molecules move apart, occupying more space per molecule.

That's exactly what happens in an XP lab. Let's face it, people enjoy personal space. To deny that is to deny common sense. People can be forced into close contact for periods of time, but when the necessity (pressure) to be in close contact is reduced, they'll spread out. A "we need to pair more" scolding from the coach is the pressure. (Note: Real-life XP consists of daily scolding and self-flagellation in the form of "We aren't doing enough of ___.")

A curious VP stopping by the lab is added pressure. The threat of finding yourself without a job if you don't pair is *tremendous* pressure. It's interesting to see pairs form under pressure. It's even more interesting when people "pretend" to pair. One way developers achieve this is by sitting quite close together. From a distance, it's impossible to tell if they're working together or they're just two people sitting really close together. Either way, it looks ridiculous. Two people hunched over, attempting to make an interface specifically designed for an individual work is a visual oxymoron. Without a doubt, there's going to be an increase in neck and back problems in the developer community.

But what's even more amusing is what happens when the pressure is released. This may be in the form of a coach being in a meeting or perhaps on vacation. It could also be as a result of being very early or late in the day, or on a weekend. When the pressure is reduced, the developers, just like air molecules, adjust to a more natural distance. This natural distance consists of developers working at their own workstations. If fewer people are present than normal, developers will naturally choose a position one or two chairs down from the next person.

Peer Pressure and the Silence Factor

By far the most striking characteristic of developers in the XP lab is their inability to admit (or resistance to admitting) that they hate to pair. If they were to admit that they dislike pairing, they would go through a very unpleasant ostracizing process. The reason is that the worst thing that can happen to an XP developer is for someone to find out that he has a few disagreements with the process. I've never experienced fear like I do now on any project I've ever been on. This is key to understanding the XP world.

In a traditional environment, you would be able to say with impunity, "I hate writing documentation." Contrast this to the XP world, where dissention is fatal. Even though XP proponents make a big deal about the ability to change the process, a few processes are nonnegotiable, pairing being one of them. Quite literally, the words of the prophets are written on the lab walls: "Pair up!" Or more precisely, all production code must be written in pairs. I now know methodology can become dogma to the extent of a religion.

Don't code, don't code, don't code so close to me . . .
Pleeease don't code so close to me . . .

Wishing for the Sound of Silence

Why on earth would someone writing code want a quiet place to think? An experienced consultant expresses it this way:

> *"Pair programming appeals to fresh-faced young programmers who want to bond a little too much, people who spend too much time 'being just mad mate', running around being zany, mistaking this for creativity. The best programmers I know want silence, as they are having 'hard thoughts'. They achieve their magic by 'thinking about the problem', not playing around waiting for the gods of creativity to give them a bolt from the blue, whilst they play Kerplunk and talk about why Postman Pat is so cool."*[7]

Of course, Ron Jeffries' thoughts on the topic are somewhat different and bear repeating:

7. Dino Fancellu, e-mail to author, January 5, 2003.

"I think maybe concentration is the enemy. Seriously. If you're working on something that is so complex that you actually need to concentrate, there's too much chance that it's too hard."[8]

Jeffries' comment reveals a lot about the Extremo mind-set: A roomful of noisy people working on bite-sized chunks of code is preferable to an environment that is more conducive to hard thinking and elegant designs that take all the project's requirements into account. In XP, it appears that everyone on the team must conform to this form of organization. There is no flexibility.

Conversely, other processes besides XP do at least offer some *form of organizational flexibility, as we discuss in Chapter 3.*

Another consultant puts it this way:

"Pair programming will always be a problem in a society of individualists (particularly of the American strain). Most of us are willing to share (i.e., peer review) but we do not want to show our 'stuff' until it is ready. It is just human nature. Dealing with a mentor is a bit different. Most of us are willing to use a mentor for suggestion and help, but we do not even want our mentor to sit on our shoulder all day long. Let the mentor help others, too."[9]

It's a Work of Love, Enforced by Coercive Means

Extremos can take their commitment to pair programming pretty seriously. Consider this from the Wiki Web:

8. Ron Jeffries, posting to the C2 Wiki page "Pair Programming Ergonomics," `http://c2.com/cgi-bin/wiki?PairProgrammingErgonomics`.
9. Gary A. Ham, e-mail to author, March 3, 2003.

"TheCoach needs to keep individuals on track with respect to the development paradigm: eXtreme Programming. This is a work of love. If you don't respect others, you're not doing it right. TheCoach get [sic] *his respect to show through for people who try hard to do the right thing.*

"TheCoach may need to use coercive means to corral the determined individualist back into the group. Ultimately, TheCoach must ask such an individual to leave, where no supportive contribution to group goals can be made by the individual."[10]

Uh huh. Pair programming isn't everything, it's the only thing. Or to borrow from Diana Ross:

Force—in the naaaaame of love. Pair up or you'll get fired. Think it o-over.

The Adventures of Uncle Joe and Jack the Siberian Code Hound

One day, "Uncle Joe" Steele, the XP coach, and Jack, his Siberian code hound, were walking through the coding room when Jack started barking and growling at Loretta. Poor Loretta was inadvertently programming by herself for a moment because JoJo had gone to the men's room.

"Nyet, Jack!" barked Uncle Joe, to keep Jack from attacking Loretta after she quickly explained that JoJo would be right back. Just then JoJo walked in and Jack (seeing that Loretta was properly paired up) calmed down and began wagging his tail.

"Nice sweatshirt, Uncle Joe," commented JoJo. "I didn't know you went to Georgia Tech."

"*Da!*" replied Uncle Joe. "Whoops—what'd I say? I mean, that's right. Georgia's always on my mind. Kind of like an old sweet song. Smell the code, Jack!"

Jack put his paws up on JoJo's desk and started sniffing at the keyboard. He sniffed and panted intently, then barked three times, put his paws back on the floor, and furiously chased his tail around in a circle.

"See, Loretta," said JoJo. "I told you we need to refactor that state tax computation."

Loretta was not at all pleased (she had already written the tax computation three times and it passed all the unit tests each time), but she didn't want to say anything in front of Uncle Joe and Jack. "I guess you're right," said Loretta. "I'll refactor it again."

10. See `http://c2.com/cgi-bin/wiki?TheCoach`.

"Good work, Jack!" said Uncle Joe with a smile. Jack wagged his tail happily, and the coach and his code hound moved on to the snack area to find a treat for Jack.

As soon as Uncle Joe was safely across the room, JoJo removed his folded-up newspaper from under his chair and resumed working his crossword puzzle, while Loretta tried to figure out how to make the tax computation smell better.

"Hey, Loretta," said JoJo, "what's a five-letter word for 'deception'?"

Loretta paused, thought for a moment, and said, "Do you know what any of the letters are?"

"It starts with 'FR,'" said JoJo, "and it ends with 'UD.'"

Loretta glanced at the copy of *Pair Programming: The SuperEgo and Its Effect on Human Sexuality*, which she kept handy for when JoJo was coding, and said, "It must be Freud. Although I can't quite see where they came up with that definition from."

"Thanks," said JoJo, moving on to his next clue.

Chapter 8 contains more adventures of Uncle Joe and his fearful band.

Voice of eXPerience: Pair Programming Ergonomics

This description was given to us by an anonymous XPer who has some practical reasons for disliking pair programming. (The ergonomic aspect of pair programming, where each pair is "crouched over squinting at a computer screen," can be seen in the photographs of the C3 team at `http://www.xprogramming.com/xpmag/c3space.htm` *and also in* Extreme Programming Installed *on page 78.)*

I've just joined a project using XP. I really wished I had some critical information before jumping in. The pair programming is mind numbing. With this XP stuff, software development is no longer a professional occupation, it's just another type of assembly-line work. We're herded into a small room like telemarketers. (Actually, I bet telemarketers have a better work environment.)

Something that nobody seems to have pointed out is that with pairing, you can't adjust the work chair, monitor, and keyboard to a suitable position. Everybody is crouched over, squinting at a computer screen. The keyboard trays were removed to allow more room for pairing. Another irony is that on my daily walk to the "lab," I pass numerous empty offices and cubes, and several assistants with huge work areas. I'm afraid that in a few years, software development will no longer be considered a "profession."

Productivity: numProgrammers/2 == numProgrammers? Right?

In a white paper on pair programming, Alistair Cockburn and Laurie Williams write:

> *"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."*[11]

This is a common argument against pair programming: Surely with two people sitting at one computer, concentrating on one program, only half as much work would get done. The usual response is that pair programming actually increases productivity in the long run because the code is of a higher quality; in other words, eventually just as much code gets written (or, to be more precise, eventually the same amount of functionality gets implemented—if it's of higher quality, then it will probably involve fewer lines of code). We've found that doing some detailed up-front design before coding allows even solitary programmers to produce very high-quality work!

This is one area that we see a circular argument in XP emerge. In *Extreme Programming Explained*, continuous integration is justified in part by the fact that because the team is programming in pairs, there are half as many streams of changes to integrate[12]—in other words, half as much code is being produced.

Coder's Little Helper
(Sing to the tune of "Mother's Little Helper" by the Rolling Stones)

What a drag it is smelling code

Things are different today
Since my pair has gone away
And the new one comes to work when he is fried

And those onions that he ate
Make me want to leave the state

I go running for the shelter
Of the coder's little helper
And it helps me through the day
Until I can get away

Breath mints pleeeeease
Take four of these
Go to the store
And buy some more

What a drag it is smelling code . . .

---

11. Alistair Cockburn and Laurie Williams, op. cit., p. 3.
12. Kent Beck, *Extreme Programming Explained: Embrace Change* (New York, NY: Addison-Wesley, 2000), p. 68.

Voice of eXPerience: Overuse of Pair Programming

by David Van Der Klauw

I believe that overuse of pair programming is a major flaw in XP.

Before I comment on pair programming (pairing), I want to make the point that it is a complex issue, not a Boolean variable to be decided Yes or No, Right or Wrong.

Pairing is a lot like marriage in the types of questions and answers that arise. Imagine the reaction I would get if I made a few general comments about the benefits of marriage and then insisted that every person should be married and that CompanyXYZ should employ only married people. There would be outrage. Yet this is exactly how the complex issue of pairing is approached by many people.

Learning by Pairing

Remember a lesson at school or university. The teacher gives a general introduction of the topic, works one or two examples on the board, and then gives you some problems to attempt by yourself. You consult the board and/or your textbook, and then attempt the questions. Should you have trouble, the teacher is there to help.

Imagine how silly it would be if instead of all this, the teacher just sat beside you and you both commenced working on the problems together. Whenever your writing slowed, the teacher told you what to write, and when you were confused, the teacher took your book and completed the problem instead of letting you think it through and work it out yourself.

This is exactly what we do with our pairing.

To me, the worst aspect of pairing is that it prevents me from stopping, investigating, and learning at my own pace when the need arises. When I am pairing, I have the responsibility not to waste the time of or bore my partner as I investigate something. As a result, I will often skip the opportunity of learning or investigating. Pairing raises the cost of learning and wastes many opportunities for investigation and improvement of my skills and the product.

Let Them Eat Cake

Is pairing better than working alone?

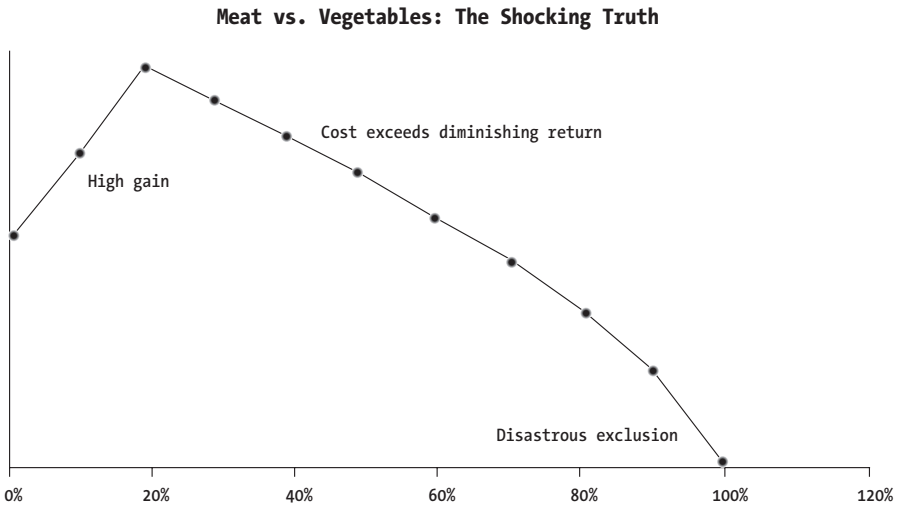
It is better in the same way that cake is better than bread, wine is better than water, and meat is better than vegetables. It is better in the right circumstances, but it is definitely not better for all circumstances.

How Much Meat Do We Need?

I tried to find an analogy to show how much pairing we should do. The best I could find is meat versus vegetables.

Vegetarians eat vegetables and do not eat meat. Without debating that topic, let's agree that while it's possible to get everything your body needs from vegetables, it's difficult. Most vegetarians will freely admit this. Therefore, a lazy vegetarian who neglects to monitor her diet would be better off eating some meat.

I've made a graph showing the health of such a lazy vegetarian as she replaces vegetables with meat.



The "High gain" area is where the meat is providing the vital trace elements, iron and protein that the body was previously deprived of.

The "Cost exceeds diminishing return" area is where the high percentage of meat increases the amount of fat and free radicals, while not providing any further health benefit.

The "Disastrous exclusion area" is where the 100% meat diet causes severe problems due to the total absence of certain vitamins and fiber that come only from vegetables.

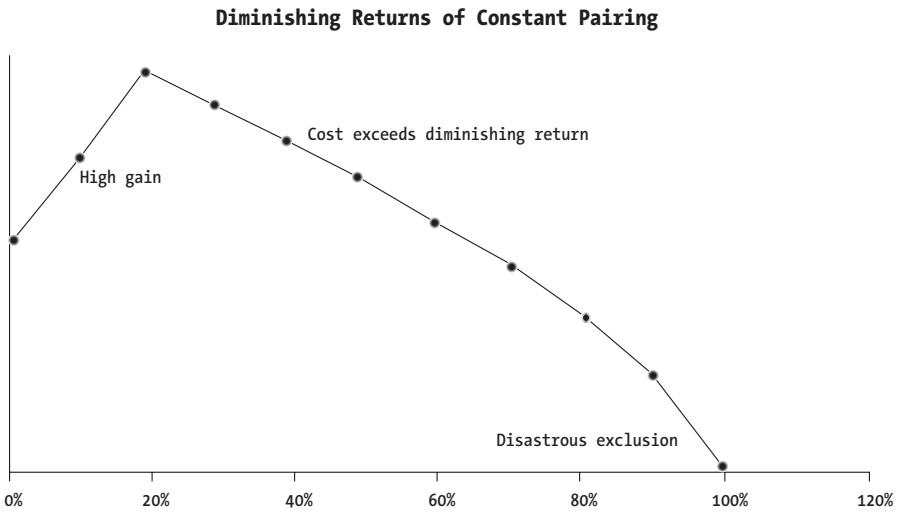
The Pairing Graph

Now I have made a graph showing the productivity of programmers who replace solitary programming with pairing (the figures shown here are purely illustrative).

The "High gain" area is where the programmers pick up those vital tips and knowledge that can only be learned from watching another programmer in action. As solitary programmers, they were deprived of these.

The "Cost exceeds diminishing return" area is where the high percentage of pairing increases the time taken, frustration, and so forth, while not providing any further benefit to the tasks.

The "Disastrous exclusion" area is where the 100% pairing causes severe problems due to the total absence of certain learning and experience that comes from only solitary programming and study.



How Much Pairing?

I recommend that the pairing level be adjusted to its optimum. How so? Well, I don't have a crystal ball to tell me the perfect levels, so I suggest a method that can be used.

Regularly ask the programmers if they want more or less pairing. Adjust the level a little bit at a time by majority vote, until half want more and half want less. Once this point is reached, consider whether the pairing level can be further adjusted on an individual person and/or task basis.

Two Minds Are Better Than One

Two minds are better than one. But then again, a chain is only as strong as its weakest link, a train is only as fast as its slowest carriage, and too many cooks spoil the broth. Which adage applies to pairing?

No one will argue that in many situations two minds are better than one. Therefore, all CompanyXYZ programming should be done in pairs if we make a few further assumptions:

- Putting two people in front of one computer automatically makes their minds work together effectively.
- CompanyXYZ is happy to pay twice as much to get the better result.
- Programmers enjoy pairing and won't leave CompanyXYZ for an alternative arrangement.

Is the Cost Justified?

When the value of pairing is questioned, its supporters will often point out one case in which a partner was helpful. They use this as proof that all coding should be done in pairs.

This is a classic case of the one-size-fits-all flaw. I've paired now for more than 6 months. I've had times when my partner taught me stuff, helped increase my speed, and found bugs I missed, and I've had a few good chats and great fun on some days. That's not good enough.

It's necessary to do a proper cost-benefit analysis of the practice of pairing. For something to be justified, it isn't good enough to find one benefit. Benefits must exceed costs and there must be no better alternative.

Don't ask

- Can a partner ever be of any value?

Do ask

- Is a partner of sufficient value to justify the cost?
- Does the partner cause additional problems?

To pay off, pair programming must deliver the average task in half the time it would take a solitary programmer to deliver the task to the same quality. My experience indicates that this order of speed increase isn't occurring. In fact, I think many tasks take much longer.

XP proponents will come back with two arguments:

- Sometimes the quality of work of a pair of programmers is higher than is achievable by a single programmer.
- Occasionally, a pair of programmers will rapidly solve a problem that a solitary programmer would be stuck on.

Rather than dispute these claims, I'll concentrate on the impact of the far more frequent occurrence where the pair doesn't get the task done faster and, in effect, takes twice the hours required for the job.

When you invest money, a sobering fact is that to break even after a loss, the percentage gain you need is greater than the percentage loss taken. For example, a 10% loss needs an 11% gain to take you back to even, a 20% loss needs a 25% gain, and a 50% loss needs a 100% gain. It turns out that these mathematics apply to XP's pairing with equally alarming results.

Consider the case where a pair is normally taking 1 hour to do tasks that take a single programmer 2 hours. In this case, a pair is as effective as a single programmer.

Now consider a situation in which the pair fails to work faster on just one task and instead takes the full 2 hours. In this case, to catch up the pair would have to work at twice their normal rate for the next two tasks. That's right, after taking as long as a single programmer on just one task, the pair would need to work at four times the speed of the single programmer for the next two tasks just to catch up.

You do the math if you don't believe me.

I believe that in the real world, a pair will often take as long as a single programmer to do a task. They may occasionally work faster, the quality of their work may even be higher, and they may avoid getting stuck occasionally. But on balance, the frequent slowness will overwhelm these benefits, as the mathematics shows.

Three's a Crowd

If two minds are better than one, are three minds better than two? If it makes sense to place two people on one computer, then why not turn up the volume, take it to an extreme, and place three or four people on one computer?

Don't laugh this off—it's a very important question. The defense of pairing relies on certain hard-to-prove claims about better quality, knowledge sharing, and collaboration. All of these claims support tripling and quadrupling in addition to pairing.

When I asked *[our XP coach]*, he told me that any more than two people results in slower communication and decision making because there's more than one channel of communication. I find this answer to be unconvincing. Surely, the fastest communication and decision making occurs within the brain of a solitary programmer.

When it comes to programming, I think two is a crowd.

Laborer or Artist?

In my analysis of pairing, I thought about all kinds of work where people worked as a pair. Police officers and airline pilots pair for safety and backup. Laborers pair for greater lifting power.

People in creative professions don't pair. Remember, too many cooks spoil the broth. Can you imagine two painters creating a masterpiece by taking turns with the brush?

Most professions don't pair simply because the cost isn't justified. Bus drivers, taxi drivers, bank tellers, judges, teachers, and most workers in other professions do the job with the smallest unit that can do the job: one person.

Is a programming job more like the jobs that justify pairing or is it more like the jobs that don't justify pairing? I think programming is more like the jobs that don't justify pairing or aren't suited to pairing.

Who Likes Pairing?

Forced 100% pairing is like a forced marriage. It might work out, but it would surely be better if it was voluntary and stood on its own merits.

Ideally, the forced pair would be ideal at pairing and would work ideally together. In the real world, however, things are a bit different. In the next few paragraphs, I make a few generalizations that I've placed in italics. These aren't absolute truths, but they're generally true.

When pairing a bossy and quiet person, the bossy person will tend to dominate the quiet person. It's likely the quiet person will dislike pairing, whereas the bossy person will probably like the extra power he has. *Bossy people like pairing, quiet people do not.*

When pairing an expert with a novice, the novice will tend to slow down the expert and the expert will tend to teach the novice. The novice will probably enjoy this arrangement, while the expert will probably not. *Novices like pairing, experts do not.*

When pairing a high achiever with a low achiever, the high achiever will get less done and will no longer be able to take credit for the high achievement. The

low achiever, however, will get more done and will be able to take partial credit for the work, as well as avoid the criticism and responsibility for her previous poor results. *Low achievers like pairing, high achievers do not.*

Although my statements are only generalizations, if they're correct they tend to suggest that if 100% forced pairing was brought into a company, *quiet, high-achieving experts would dislike pairing and leave the company, while bossy, low-achieving novices would like pairing and stay.*

I didn't write this section to insult any particular person or to infer characteristics to someone who likes pairing. I wrote it simply to provoke thought.

Stick to the Task

XP claims that when programming alone, programmers tend to take experimental detours rather than sticking strictly to the task. XP claims this is a serious problem and "solves" it by pair programming, with strict estimation and tracking of time taken.

I have something very important to say about this. I believe that the experimental detours are good. They are vital learning that most programmers need. Think about it: Why do so many programmers do it? It's because it comes naturally. Do programmers naturally do a stupid thing, or do they naturally do a sensible thing?

When is the best time to investigate something that you don't understand fully? When you come across a task that might benefit from it, when the task is right there fresh in your mind.

Under XP, you should make a note of the possible spike, complete your task, and then discuss the possible spike with your team, write it up on the board as a spike, and do it. If something good comes from it, you discuss it with the team and customer, and then write a test for it and include it in the production code (working as a pair, of course). Is it any wonder that these investigations never get done under XP?

What happens if you just have a nagging thought that a really useful function or object lies just around the corner? How do you explain that to the team and get permission to do it?

Now that I've suggested that experimental detours are good, I know what the XP proponents will argue. They will ask how you stop a programmer from spending all of her time on useless detours. Self-discipline and common sense are the answers.

If you've just spent half a day on an unfruitful detour, then it makes sense to stick strictly to your tasks for the next day or two so that you have something to show for your effort.

An unproductive programmer, whether prone to detours or otherwise, will be obvious from the lack of output in the long run. Similarly, the high productivity of a creative, detour-taking programmer will also be obvious in the long run, even if the odd day is wasted on a detour.

Pair Programming Illuminated

The book *Pair Programming Illuminated* (we refer to it as *PPI* for the rest of this section) by Laurie Williams and Robert Kessler, not surprisingly given its title, pitches the case for pair programming. However, it also discusses quite openly some of the problems typically encountered by pair programming teams. We focus on some of those problems in this section and examine how they may affect an XP project as a whole.

Problems with Pairing Different Categories of Programmer

PPI divides programmers into different categories and then discusses the effects of the various combinations thereof. The programmer categories are novice, average, expert, introvert, and extrovert. The pairing combinations discussed in *PPI*, with a chapter dedicated to each, are as follows:

- Expert-expert
- Expert-average
- Expert-novice
- Novice-novice
- Extrovert-extrovert
- Extrovert-introvert
- Introvert-introvert

Because pairs are meant to rotate frequently, these various combinations will resurface often in a team of mixed abilities. Thus, in small teams (which is likely, given an XP project), it would be difficult to keep "problem pairs" apart.

"Go Make Me a Cup of Tea" Syndrome

What happens if you pair up a newbie programmer with an expert? This is described in *PPI* as "expert-novice pairing." The intention of such a pairing would be to "get the easier job done well, while training a novice programmer."

The challenge of such a pairing is primarily that the expert must take on a tutoring role and must maintain extreme patience throughout. If the expert

coder slips, then the result is a "watch while I type" session (also known as "go make me a cup of tea while I finish this program" syndrome[13]), in which the novice remains passive throughout and the expert is effectively solo-coding.

Despite this, there are distinct advantages to expert-novice pairing. In fact, it's probably the one pairing combination that's worth mandating, as long as the novice is willing and able to learn and the expert is prepared to give up a portion of her day to teach rather than code in full-flow. This combination is certainly better than novice-novice pairing, which even Ron Jeffries thinks is a bad idea.[14]

Laurel and Hardy Take Up Pair Programming

The intent of a novice-novice pairing combination is described in *PPI* as follows:

> *"To produce production code in a relatively noncomplex area of the project, giving valuable experience to both programmers in the process."*[15]

If you're considering such a pairing, it's important to ask yourself which part of your project is unimportant enough that you can afford to unleash two complete novices, unsupervised, on it.

"Unsupervised" is actually the key. Two novices, unsupervised, would likely produce code that isn't exactly production quality. Luckily, *PPI* has the answer:

> *"There must be a coach, instructor, or mentor available to answer questions and also to help guide the pair. . . . We feel very strongly about the need for a coach. If you are unwilling to assign the mentoring task to some expert, then you need to understand the limitations of the asset being produced by the pair."*[16]

In XP, this responsibility would fall into the lap of the person (or people) performing the coach role.

As with the other pairing combinations, pairs rotate so frequently that in a team of mixed abilities, the novice-novice pairing could happen quite often. Therefore, novice-novice pairing isn't something that can easily be controlled:

---

13. Matt Stephens and Doug Rosenberg, *Extreme Programming Refactored: The Case Against XP* (Berkeley, CA: Apress, 2003), Chapter 6, "Pair Programming."
14. Laurie Williams and Robert Kessler, *Pair Programming Illuminated* (New York, NY: Addison-Wesley, 2002), p. 120.
15. Ibid., p. 118.
16. Ibid., p. 119.

It just happens, almost by accident, several times a week. The coach must be fully aware of the fact that two novices are currently pairing at any time, and the coach must be available to guide them and correct their mistakes. In practice, to combat the proverbial blind leading the blind, there's a risk that the coach may become fully occupied with mentoring one particular pair anytime two novices pair up.

Carrying Your Pair

Similar but less extreme problems occur with expert-average pairing. *PPI* describes three situations where the authors feel that expert-average pairing is a problem. The first is that the average programmer truly is average (i.e., the average programmer is likely to stay that way and will never really progress). The second is when the average programmer doesn't interact enough with the expert. The third is when the average programmer doesn't seem to "get it" and keeps asking the same question over and over:

> *"This can leave the expert frustrated and can reduce the ability of the pair to complete the task."*[17]

And the Winner Is . . .

Aside from the longer-term learning benefits, it seems that the most beneficial form of pairing is with two programmers of roughly the same ability. It's more likely that the pair will be on the same wavelength and will spend less time disagreeing over things that probably don't matter that much.

Unfortunately, when you consider that 50% of all programmers are below average, it becomes obvious that mixed-ability pairing is likely to be the norm. This highlights the problem that teams of mixed abilities are almost unavoidable. Pair programming makes the issue unavoidable by forcing these people to code together on the same program. In a non–pair-programming project, the problem is handled effectively through other more natural practices, such as team leading, code and design reviews, *occasional* (voluntary) pair programming, mentoring, design documents, and so on.

With almost all of the problems described in this section, it's up to the coach to catch and deal with them as promptly as possible. This places a lot of responsibility on the coach (almost as much as the on-site customer!).

---

17. Ibid., p. 108.

Design Documents Reduce Reliance on Pair Programming

Design documents provide a record of design decisions. This makes them particularly helpful for novice programmers to explore the thinking behind the design, as described by the more experienced senior programmers.

If the team is becoming lost in a sea of changed minds and refactorings, the design document often helps to remind the team members of why they originally decided to do something in a particular way. There's usually a pretty good reason.

We discuss the role of documentation in software projects (and how it can lessen the need for pair programming) in Chapter 7.

And More Problems

Chapter 7 of *PPI* (titled "Problems, Problems") discusses several problems with pair programming. We briefly discuss some of these problems here. Although the authors of *PPI* do offer some practical advice to overcome or help prevent these problems, the proposed solutions either result in high maintenance or rely idealistically on the programmers being constantly aware of all the problems (with advice such as "Just proceed a bit more cautiously").

One problem is that of rushing. Because pairs rotate often, they might rush to finish a task before it's time to separate. The advice given in Chapter 7 of *PPI* is as follows:

> *"If a task must roll over to another pairing session, the task must roll over to another pairing session! Slow down, and do it right together."*[18]

The coach would need to be particularly vigilant to spot this problem recurring, because pairs rotate so often. If the problem happens a lot, it may be because the tasks are too big (another direct consequence—evidence of the circle of snakes unraveling. To counter this, the team may need to spend more time planning or designing, or change its process for estimating stories or tasks).

Another problem, which we suspect would particularly manifest in teams that publicly laud themselves as "the best team on the face of the Earth," is that of overconfidence:

18. Ibid., p. 61.

> *"There may be a feeling that a pair can do no wrong. If you're working together, you might convince yourself that whatever you do together must be right. Remain cautious and careful!"*[19]

The problem of overconfidence would need to be watched for carefully by the coach, who should be aware of this type of problem. She would then need to be able to watch out for the telltale signs and be prepared to act on them when she catches pairs reassuring each other into writing bad code. "Well, I suppose it will do for now—we can refactor it later!" is the typical start of a slippery slope.

Another problem is that it's human nature for people to want to be in control, at least of their immediate surroundings:

> *"New folks should specifically be paired with mentoring types, lest they feel unwelcome or frustrated in the hands of a partner who wants to make only personal progress. This mentor must also give up control and allow the less skilled team member to drive most of the time. When the mentor is directing most of the activity, it's better for the trainee to be typing and not just listening. The student might not be assertive enough to ask for the keyboard."*[20]

This is, of course, an idealistic approach. As we discovered in Rich Camden's "Voice of eXPerience" account earlier, being in control of the keyboard is the preferred option for most people. Rich wasn't offered the keyboard once in 5 months (that's not to say that he didn't get to type, but no one actually *offered* to relinquish control). If the other person doesn't speak up, he's not going to be offered the keyboard. As the previous quote suggests, this is particularly a problem with inexperienced programmers being allocated an experienced partner. Everybody likes to be the driver, to be in control.

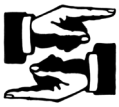
Watch Out, There's a Snake Under the Desk!

The problems we just described must all be watched for and quickly fixed before they lead to other problems. This is a lot of problems associated with one XP practice, all waiting to slip and catch the unwary coach, who must be especially vigilant.

Pair programming should be a beneficial practice, but its problems are much more acute because (as we discussed in Chapter 3) so much else in XP relies so heavily on its correct and consistent execution throughout the project.

---

19. Ibid., p. 60.
20. Ibid., p. 61.



We examine why pair programming's problems are particularly dangerous in XP in Chapter 3.

A Pair of Fangs

The pair programming "snake" isn't just about whether the team stops pair programming. It's also about

- Preventing individuals from "hogging" the keyboard
- Mitigating problems when ill-matched programmers repeatedly pair up
- Tackling conflicting social dynamics that might impair your pair programming experience
- Keeping tasks short enough to complete in one session to prevent rushing
- Identifying and somehow dealing with overconfidence of "experienced" pairs

And it's about the coach remaining super-vigilant throughout the project. If any one of these risks manifests, the circle of snakes is yet again in danger of unraveling.

And, of course, if one programmer gets a cold, everybody gets a cold!

More about pair programming and the common cold in the "Camp Regretestskiy" sidebar in Chapter 8.

Pair Programming Defanged

The beneficial aspects of pair programming (improved code-level design, knowledge sharing via improved communication, sense of team spirit, and so forth) can be achieved without the negative aspects (high-maintenance practice, potential for incompatible pairing that surfaces every time the pairs rotate, "overkill" for simple tasks, and so forth) by not mandating its use, but instead encouraging that programmers simply pair up for complex tasks.

This can be achieved by following a process that doesn't rely heavily on pair programming. By spending more time on up-front design—and by making this a team process (which XP does to an extent through the use of collaborative design sessions)—most of the key design decisions will have been made by the time the team begins writing production code.

Written documentation (possibly in the form of a project Wiki) also lessens the need for pair programming and colocated teams.

We cover XP's approach to documentation in Chapter 7.

Summary

Pair programming, like many of the other XP practices, appears different in theory and practice. Pair programming as a voluntary activity should be encouraged, when appropriate.

It's important, however, to not lose sight of the fact that programmers need peace, quiet, and space in which to think and concentrate, despite Ron Jeffries' claim to the contrary. The social aspects of pair programming can be very difficult. And, when you add it all up, you're still taking two programmers to produce what a single programmer would under normal circumstances.