How Pair Programming Really Works

Stuart Wray, *Royal School of Signals*

Pair programming isn't always successful, and recent studies cast doubt on the "driver-navigator" metaphor. Four mechanisms can improve pair programming performance.

Share your comments at http://computingnow.computer.org/wray.

Pair programming has generated considerable controversy: some developers are enthusiastic about it, almost evangelical; others are dubious, even hostile. However, a large factor in this controversy is that programmers label a wide variety of practices under the "pair programming" umbrella. Thus, before our community can sensibly discuss how pair programming works, we first need to establish exactly what it is.

As a dictionary definition, I'd say that pair programming is a technique in which two people sit down, literally side by side, and write a program at the same computer. When Kent Beck originally coined the term, he described two programmers working at different levels of abstraction.[1] Laurie Williams and Robert Kessler made this idea more concrete, using the metaphor of one programmer being the "driver" and the other the "navigator."[2] In this metaphor, the driver controls the keyboard and focuses on the immediate task of coding, and the navigator acts as a reviewer, observing and thinking about more strategic architectural issues.

My own experience as a developer using pair programming is that it isn't just a technique where one person programs and the other person watches. Both programmers work closely together, chatting the whole time, jotting down reminders of things to do, and pointing out pieces of code on the screen. (One of the clichés of pair programming is that if you're doing it right, your screen should be covered with greasy finger-marks by the end of the day.) Programmers take turns at the keyboard, usually swapping over with a phrase like, "No, let me show you what I mean."

Jan Chong and Tom Hurlbutt confirmed this view of successful pair programming after spending several months on an ethnographic study of professional developers who use pair programming in their daily work.[3] They found that programmers tended to work together on the same facet of a problem almost the whole time and swap between tactical and architectural levels as a pair. Similar ethnographic studies by Sallyann Bryant and her colleagues[4] and Stephan Salinger and his colleagues further confirmed this.[5]

Of course, not all attempts at pair programming have been successful—Matt Stephens and Doug Rosenberg, for example, reported unfavorably on their experiences.[6] However, what they described is a caricature of the driver-navigator metaphor, with one programmer firmly in control and the other sitting quietly, doing little. Such misunderstanding shows that we can't take a claim that developers are pair programming at face value; they might not be doing what experienced and effective pair programmers actually do.

0740-7459/10/$26.00 © 2010 IEEE

This kind of misunderstanding also casts doubt on the many attempts to assess pair programming's effectiveness. (Tore Dybå and his colleagues provide a very nice summary of this experimental work.[7]) If the subjects of these experiments did different things, can we really compare their results? And if they weren't doing what successful pair programmers do in commercial practice, can we apply their findings to commercial development?

In this article, I advance four mechanisms prompted by my own experience of pairing in both agile and non-agile development. These mechanisms explain a large part of what successful pair programmers do. Of course, this is only the beginning: you might have experiences that confirm or contradict my suggestions. What have I missed? I hope you'll contribute to the discussion of these issues on the Web site (http://computingnow.computer.org/wray).

Mechanism 1: Pair Programming Chat

Around 1980, as computer science undergraduate students at the University of Cambridge, my friends and I noticed a strange phenomenon that we called *expert programmer theory*. When one of us had trouble getting our programs to work, we'd describe the nonfunctioning state of our code to each other over coffee. Quite often, we'd realize in a flash what was wrong and how to solve it. These epiphanies were quite independent of the other person having any real understanding of our problems—the listener often seemed little wiser about the subject.

Since then, I've found this phenomenon is well known to professional developers, and sometimes described in textbooks and research papers. For example, Brian Kernighan and Rob Pike recommended explaining problems aloud, even to a stuffed toy,[8] a practice that John Sturdy called the *rubber-plant effect*.[9] Part of pair programming's effectiveness is presumably due to this effect being continually triggered: as one programmer gets stuck, the back-and-forth chat serves to unstick them in the same way as solo programmers talking about their problems out loud. However, this raises the question of whether any type of speaking will help or whether something specific is needed.

Research on "self-explanation" by Michelene Chi and others throws some light on this question. Chi and her colleagues described a study that tested a control group of students before and after they received a textbook explanation to read.[10] They tested another group in the same way, but encouraged the students to explain the textbook out loud and "fill in the gaps" for themselves. The self-explainers learned significantly more than the control group, and those who explained the most improved the most. The researchers also prompted the students for their explanations; they weren't just left to their own devices. In particular, they were "prompted for further clarification by the experimenter if what they stated was vague."[10]

This brings me back to an often-neglected aspect of the expert programmer theory. When we coined that term, we noticed that although real understanding wasn't necessary on the listener's part, a belief that the listener really was an expert seemed to significantly improve the outcome (hence our choice of name).

But why would believing that you were talking to experts make any difference when they didn't need to understand your explanation? Recent work by Rod Roscoe and Chi showed that prompting questions seems to be the key.[11] In their study, one student (the tutor) explained material to another student (the tutee). As expected, the tutor actually learned more than the tutee, but the questions the tutee asked made a dramatic difference in the quality of the tutor's explanations. Most questions were shallow, and could be satisfied by mere repetition of facts, but some questions were deep and often prompted deep answers that included novel inferences or self-monitoring statements.

So perhaps this is how expert programmer theory really works: an expert is more likely to ask a deep question, which prompts the novel inference from the stuck programmer. It also seems possible that merely thinking that you're talking to an expert—or pretending—will help the stuck programmer produce the sort of deep questions that experts have asked them in the past.

As an explanation for expert programmer theory, this is almost satisfactory, but is student learning a good analogy for what happens to stuck programmers? After all, the students in these experiments had to master basic science, and their explanations helped them work out what they didn't understand. Stuck programmers must already have all the information somehow hidden in their heads and then realize the answer in a moment of epiphany. How's that possible?

It's widely accepted that cognitive abilities are divided into a variety of largely separate *mental modules*, each dealing with a different ability such as intuitive grasp of small numbers, predicting other people's actions, facial recognition, and so on. Less well known is the role of the *language*

An expert is more likely to ask a deep question, which prompts the novel inference from the stuck programmer.

What we notice depends on what we expect to see and what we unconsciously consider salient.

module in integrating other modules' knowledge. Experiments by Linda Hermer-Vazquez and her colleagues on integrating knowledge about geometry and color[12] and by Ashley Newton and Jill de Villiers on false-belief reasoning[13] showed that adults perform as poorly as young children when their linguistic abilities are occupied with a verbal *shadowing* task. The language module seems crucial to combining knowledge from other modules.

This isn't to say that we integrate the outputs of several mental modules by talking to ourselves. Rather, Peter Carruthers suggested that because speech is uniquely both an input and output brain medium, the language module is the only one with a strong connection to all the other modules.[14] The mechanisms underlying the logical form of language might thus be redeployed at a level beneath conscious awareness to integrate information from other modules. The logical form must be able to represent objects with properties derived from several modules because this is the basis for noun-phrases in speech.

As programmers, we clearly use visual imagination to help design and debug our programs (although the diagrams we use bear little relation to our programs' texts). This visual information can be spread across several mental modules, and the other information we require to understand our programs can be in yet other modules. For example, it seems that our understanding of object-oriented (OO) programs is supported by the *folk psychology* module that supplies intuitions about other people's actions. (We think of objects as having intentions, wanting to do things, and sending each other messages.) We therefore need to integrate information from separate modules when thinking about our programs. Why can't we always integrate it straightaway?

Carruthers suggested that we must rely on the language module posing the right question and that the other modules don't usually present information spontaneously. However, when we hear the right question, our brains make the necessary information available, and the language module can then perform rudimentary inference and draw the obvious conclusions. Carruthers suggested that the key is posing a question that's "both relevant and fruitful."[14] The right question draws forth the crucial knowledge, and in a moment of epiphany, the answer becomes obvious

This first mechanism would therefore lead us to predict that programmers who chat about their programs more should be more productive and that those who pose occasional deep questions for each other should be most productive of all.

Mechanism 2: Pair Programmers Notice More Details

Research on *change blindness* and *inattentional blindness* illustrates something that stage magicians have known for a long time: if we don't know what to look for, we can stare right at it and still miss it. What we notice depends on what we expect to see and what we unconsciously consider salient. So, although successful pair programmers will concentrate mostly on the same things, they might notice different things.

Research on change blindness shows that people are remarkably poor at detecting changes, not only in 2D images under laboratory conditions but in real-life situations such as noticing the substitution of one person with another.[15] It appears that people remember something they saw as belonging to a particular mental category, then fail to notice substitution by another member of that category. A large portion of experts' proficiency is probably in their more detailed and extensive array of mental categories in their particular fields.[16] Research on inattentional blindness has similarly shown that when our attention is focused on a particular task, we can miss something that would otherwise be so obvious that it would just pop out. For example, it might seem unlikely that people would miss a woman in a gorilla suit walking into the shot in a video, but that's what half the subjects did in a study by Daniel Simons and Christopher Chabris.[17] (They'd been instructed to pay close attention to another aspect of the video.)

So, two people programming together won't have the same prior knowledge or categorization: one will presumably spot some things faster and the other different things faster. Where their rate of working is limited by the rate they can find things by just looking, two heads must be better than one. And in fact, one of the earliest observations that people make when they start to pair program is that the person who isn't typing code always picks up typos quicker: "Oh, you've left out the comma here."

Of course, the compiler would pick up such small slips easily, so in this case the early catch isn't very important. However, it's crucial to catch problems early when the slip is more subtle—for example, if the code is syntactically correct but semantically wrong, or where there's a fault in the design itself. Such slips can easily cause hours of problems at a later date. The ability to catch mistakes early in an online code review is only one benefit of two pairs of eyes: perhaps even more important is looking at old code with a fresh eye and

a different set of expectations, reading what it really says, not what it ought to say.

This second mechanism also partially explains the phenomenon of *pair fatigue*, which I've noticed in myself and others. When two programmers pair together, the things they notice and fail to notice become more similar. Eventually, the benefit from two pairs of eyes becomes negligible. Beck suggested that pairs should rotate at frequent intervals, perhaps once or twice a day.[1] Arlo Belshee found that in a *jelled* team, rotating after two hours was optimal.[18] Some pair programmers regard rotation as an optional part of the practice, and on a small team, or with few programmers willing to pair, there might be little alternative. However, pair fatigue means they'll ultimately be much less productive.

On the other hand, because a great deal of expert knowledge is probably in the form of categories in long-term memory, a novice might be unable to distinguish between events experienced at different times. Experts really can see things that novices can't. We could therefore predict that this second mechanism will bring the maximum benefit to novice pairs; indeed, the most extensive experiment with novice pairs and experts found that novice pairs benefited the most.[19]

Mechanism 3: Fighting Poor Practices

As programmers, we don't always use the best practices. An advantage of pair programming is said to be *pair pressure*, the feeling of not wanting to let your partner down.[20] But why is this necessary? Why do we persist in poor programming practices when we know they're poor? Is there something special about programming that makes it more difficult to do the right thing? It appears that there is.

Let's look at a particular example of worst practice: the code-and-fix style of programming most often used by novices (and sadly, often used by more experienced programmers). Programmers write some code that they hope will do a particular thing and then run it to see what happens. If it appears to work, they press on with other code, without systematically searching for flaws. When it fails, which is often the case, they tinker with the code haphazardly until it appears to work. Why is this style of programming so compelling and so easy to discover independently?

Traditional behavioral psychology offers a very plausible explanation, although more modern work on the neuroscience of learning and addiction also points in the same direction. One form of learning explored by behavioral psychologists, called *operant conditioning*, involves learning to perform some action spontaneously. This is the way that animals learn to perform tricks in circus acts or domestic dogs are taught obedience. An animal has a variety of behaviors that it engages in occasionally, and with operant conditioning, we can supply the animal with a reward after we observe it doing what we want (which *reinforces* the behavior). As this reinforcement process continues, the desired behavior becomes more likely to happen spontaneously, even when no reward is given.

Of course, if the rewards stop entirely, the behavior diminishes and finally ceases, a process known as *extinction* (which happens quite slowly). If we supply a further reinforcement before the behavior has entirely ceased, we can easily restore it to full strength. In fact, learning happens quickest if the reward pattern is unpredictable, with a so-called variable ratio (VR) schedule of reinforcement. Henry Gleitman and his colleagues explain:

> *In a VR schedule, there is no way for the animal to know which of its responses will bring the next reward. Perhaps one response will do the trick, or perhaps it will take a hundred more. This uncertainty helps explain why VR schedules produce such high levels of responding in humans and other creatures. Although this is easily demonstrated in the laboratory, more persuasive evidence comes from any gambling casino. There, slot machines pay off on a VR schedule, with the 'reinforcement schedule' adjusted so that the 'responses' occur at a very high rate, ensuring that the casino will be lucrative for its owners and not for its patrons.*[21]

This learning is unconscious: we need not realize that it's happening to us, and in the case of the casino, a machine instead of a real person is conditioning the slot-machine patrons. In our habitual patterns of software development, we too can be conditioned by our machines. This is the special property of interactive programming that makes it difficult to do the right thing. With code and fix, we tinker haphazardly with our programs, effectively putting a coin into the slot machine each time we run our code. Slot machines are known as the most addictive form of gambling, and the similarly unpredictable rewards from code-and-fix programming mean that it could be equally addictive.

How can we resist this addiction? Perhaps we can try to "just say no" and choose a differ-

Some programmers are up to 10 times more productive than others.

ent development pattern. Some development processes attempt to remove temptation by being less interactive. Edsger Dijkstra suggested that students shouldn't be allowed near a computer until they'd learned to write programs away from one.[22] Such ideas might have once had merit, but it seems foolish to turn our backs on the orders-of-magnitude increase in computer power available to us.

Pair programmers might be less susceptible to poor practices because they can promise to write code in a particular way and ensure that each other's promises are kept. The prevalence of two-people working in jobs where human fallibility is a serious problem should lead us to seriously consider that pair pressure might be the solution for us, too. However, you can only keep a promise if you made one in the first place. We should therefore expect that to benefit from the third mechanism, programmers must agree in advance how they're going to write and test their code.

Mechanism 4: Sharing and Judging Expertise

Even within a single development team, conventional wisdom says that some programmers are up to 10 times more productive than others.[23] Certainly we see a wide range of expertise, but how confident can we be in saying who contributes most to overall productivity? Assigning credit for success is difficult in team activities because there are so many variables.

In some fields, it's easy to recognize experts because individual contributions are simple to measure. Chess players have numerical rankings; golfers have handicaps. These are good predictors of their likely success against other players. But in team activities, so many factors contribute to success or failure that we simply can't understand the causal relationships without a detailed scientific investigation, so we usually select one or two arbitrary factors to simplify the analysis.[24] In software development, "lines of code written per day" often gets elevated above all others, simply because it's easy to measure. But selecting such arbitrary factors tends to promote "star players" who demonstrate those qualities but don't significantly contribute to the team's success.

Unfortunately, more detailed scientific analysis is seldom practical. So how can we assign individual credit (or blame) for team performance? Paul Graham said that when an expert programmer works alongside another programmer on the same problem, the expert can judge the other programmer's skill. But that's the only way: he or she can't tell just by meeting them. "I can't tell, even now," he wrote. "You also can't tell from their résumés."[25]

This is my experience, too: it isn't enough to talk with someone about programming; you have to work on a problem with them to gauge their expertise. A weak version of this technique is standard practice in programming interviews. After the preliminary discussion centered on the applicant's résumé, the interview proceeds to a series of successively more difficult programming exercises that the applicant has to talk through at a whiteboard. I'm frequently surprised by how a very plausible-sounding candidate, when challenged in this way, completely fails to produce even the most basic evidence of the knowledge that he or she earlier claimed.

Sadly, these poor candidates seem blissfully unaware of their own lack of expertise. They're so bad that they don't realize how bad they are, probably because, in the words of Justin Kruger and David Dunning, "the same knowledge that underlies the ability to produce correct judgments is also the knowledge that underlies the ability to recognize correct judgment."[26] In a field where expertise is hard to measure, this is a serious problem, because as Kruger and Dunning observed, the less competent are often more confident of their own ability than their more expert peers.

The most competent, on the other hand, suffer from the opposite problem, the *false consensus effect*, in which they believe that their own abilities are typical. This happens for the same reason: it's hard to accurately assess others' competence, so the most competent have no reason to believe that they're extraordinary—unless they work closely with another programmer on the same problem.

Most programmers work on problems on their own, so no one knows how good (or bad) they really are. But with pair programming, people continually work together. Because they keep swapping pairs, everyone on the team learns who's the most expert at particular things. From this comparison, they also realize their own level of expertise. We should therefore expect more accurate estimates of time and difficulty by a pair programming team than from a solo programming team. From my experience, this does appear to be the case.

We're no longer in the first flush of pair programming, yet the gulf between enthusiasts and critics seems as wide as ever. Experimental evidence has been equivocal. How can we advance our understanding? I believe the mechanisms I describe here are

among the fundamental properties shared by all instances of successful pair programming, but other mechanisms are important, too: for example, team jelling appears to have a significant effect. What other mechanisms are significant? Although I believe that pairing works the same in agile and non-agile settings, this has yet to be established. In addition, there might be anti-mechanisms: poor practices that lead to unsuccessful pair programming and that aren't merely the absence of the beneficial mechanisms. To investigate all of these, we could solicit suggestions from working developers; ethnographic researchers could reexamine their records for evidence for or against the mechanisms.

We also need some form of objective checklist to compare results across experiments, so that experimenters can agree how much a particular programming team uses a particular mechanism. With such a checklist, we could then reexamine the experiments that Dybå described[7] and attempt to establish the extent to which the teams used certain mechanisms. However, such post hoc analysis could still give equivocal results. To clearly establish the mechanisms' impact, we must design new experiments that properly control for them.

Perhaps you have some other questions—or even some answers—in mind right now. If so, I invite you to share your comments on the Web site (http://computingnow.computer.org/wray). In any case, I hope that thinking about these mechanisms will help you apply pair programming more effectively.

References

1. K. Beck, *Extreme Programming Explained: Embrace Change*, 1st ed., Addison-Wesley, 2000.
2. L. Williams and R. Kessler, *Pair Programming Illuminated*, Addison-Wesley, 2003.
3. J. Chong and T. Hurlbutt, "The Social Dynamics of Pair Programming," *Proc. 29th Int'l Conf. Software Eng.* (ICSE 07), IEEE CS Press, 2007, pp. 354–363.
4. S. Bryant, P. Romero, and B. du Boulay, "Pair Programming and the Mysterious Role of the Navigator," *Int'l J. Human-Computer Studies*, vol. 66, no. 7, 2008, pp. 519–529.
5. S. Salinger, L. Plonka, and L. Prechelt, "A Coding Scheme Development Methodology Using Grounded Theory for Qualitative Analysis of Pair Programming," *Proc. 19th Ann. Workshop Psychology of Programming Interest Group* (PPIG 07), Psychology of Programming Interest Group, 2007, pp. 144–157; www.ppig.org/papers/19th-Salinger.pdf.
6. M. Stephens and D. Rosenberg, *Extreme Programming Refactored: The Case against XP*, Apress, 2003.
7. T. Dybå et al., "Are Two Heads Better than One? On the Effectiveness of Pair Programming," *IEEE Software*, vol. 24, no. 6, 2007, pp. 12–15.
8. B. Kernighan and R. Pike, *The Practice of Programming*, Addison-Wesley, 1999.
9. J. Sturdy, "Sidebrain: A Sidekick for the Programmer's Brain," *Proc. 17th Ann. Workshop Psychology of Programming Interest Group* (PPIG 07), Psychology of Programming Interest Group, 2005, pp. 215–226; www.ppig.org/papers/17th-sturdy.pdf.
10. M. Chi et al., "Eliciting Self-Explanations Improves Understanding" *Cognitive Science*, vol. 18, no. 3, 1994, pp. 439–477.
11. R. Roscoe and M. Chi, "The Influence of the Tutee in Learning by Peer Tutoring," Presented at 26th Ann. Conf. Cognitive Science Soc., 2004; www.cogsci.northwestern.edu/cogsci2004/papers/paper278.pdf.
12. L. Hermer-Vazquez, E.S. Spelke, and A.S. Katsnelson, "Sources of Flexibility in Human Cognition: Dual-Task Studies of Space and Language," *Cognitive Psychology*, vol. 39, no. 1, 1999, pp. 3–36.
13. A. Newton and J. de Villiers, "Thinking While Talking: Adults Fail False-Belief Reasoning," *Psychological Science*, vol. 18, no. 7, 2007, pp. 574–579.
14. P. Carruthers, "The Cognitive Functions of Language," *Behavioral and Brain Sciences*, vol. 25, no. 6, 2002, pp. 657–726.
15. D. Simons and D. Levin, "Failure to Detect Changes to People During Real-World Interaction," *Psychonomic Bull. and Rev.*, vol. 5, no. 4, 1998, pp. 644–649.
16. N. Charness et al., *The Cambridge Handbook of Expertise and Expert Performance*, Cambridge Univ. Press, 2006.
17. D.J. Simons and C.F. Chabris, "Gorillas in Our Midst: Sustained Inattentional Blindness for Dynamic Events," *Perception*, vol. 28, no. 9, 1999, pp. 1059–1074.
18. A. Belshee, "Promiscuous Pairing and Beginner's Mind: Embrace Inexperience," *Proc. Agile Development Conf.* (AGILE 05), IEEE CS Press, 2005, pp. 125–131.
19. E. Arishholm et al., "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise," *IEEE Trans. Software Eng.*, vol. 33, no. 2, 2007, pp. 65–86.
20. L. Williams and R. Kessler, "The Effects of 'Pair-Pressure' and 'Pair-Learning' on Software Engineering Education," *Proc. 13th Ann. Conf. Software Eng. Education and Training* (CSEE&T 00), IEEE CS Press, 2000, pp. 59–65.
21. H. Gleitman, A.J. Fridlund, and D. Reisberg, *Psychology*, 6th ed., W.W. Norton & Co., 2004.
22. E. Dijkstra, "On the Cruelty of Really Reaching Computing Science," 1988; www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF.
23. R. Glass, *Facts and Fallacies of Software Engineering*, Addison-Wesley, 2003.
24. M. Gladwell, "Game Theory," *The New Yorker*, 29 May 2006; www.newyorker.com/archive/2006/05/29/060529crbo_books1.
25. P. Graham, "Great Hackers," 2004; www.paulgraham.com/gh.html.
26. J. Kruger and D. Dunning, "Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments," *J. Personality and Social Psychology*, vol. 77, no. 6, 1999, pp. 1121–1134.

About the Author

Stuart Wray is a senior lecturer at the Royal School of Signals. His research interests include the psychology of programming, computer security, and functional programming. Wray has a PhD in computer science from the University of Cambridge. Since then, he's worked in research at the Olivetti Research Laboratory and the University of Cambridge Computer Laboratory, and in product development at Virata, Marconi, and BAE Systems. Contact him at swray@bournemouth.ac.uk.