

How does pair programming work?

Stuart Wray
Royal School of Signals, Blandford, UK

swray@bournemouth.ac.uk

Abstract

Anecdotes and experiments show that pair programming sometimes brings benefits and sometimes doesn't. Recent studies have cast doubt on the "driver-navigator" metaphor, often used to explain pair programming. We need a new theory to explain why pair programming is sometimes useful and sometimes not. This article proposes four distinct mechanisms, by which pair programming may improve the performance of developers, and explains the psychological basis of each mechanism. With this theoretical explanation in hand, we may be better placed to predict when we should use pair programming.

Key words: software psychology, software process, programming teams

Introduction

Pair programming has generated considerable controversy over recent years. Some developers are enthusiastic about it, almost evangelical; other are dubious, even hostile. However, when we read the different accounts of pair programming more closely, it is clear that a large part of this controversy must be due to the fact that different programmers are in fact engaging in a wide variety of practices, and yet calling them all "pair programming". Before we can sensibly ask how pair programming works, we need to establish what pair programming is.

As a "dictionary definition" we might say that pair programming is a technique where two people sit down, literally side-by-side, and write a program together at the same computer. When Beck [1] originally coined the name "pair programming", he described the two programmers as working at different levels of abstraction. This idea was made more concrete by Williams and Kessler [2] who used the metaphor of one programmer being the "driver" while the other was the "navigator". In this metaphor, the driver has control of the keyboard and focuses on the immediate task of coding.

The navigator, on the other hand, is said to act as a reviewer, observing and thinking about more strategic architectural issues. Perhaps this metaphor helped to suggest potential benefits to sceptical managers who might otherwise see one programmer too many sitting at the computer. Unfortunately, some programmers have blindly followed this description and been disappointed with the results [3], without ever learning what experienced and effective pair programmers actually do when they sit together.

My own experience as a developer using pair programming is that it is *not* a technique where one person programs and the other person watches. The two programmers work very closely together, chatting the whole time, jotting down reminders of things to do in the next few minutes, and pointing out pieces of code on the screen. (One of the clichés of pair programming is that if you are doing it right, your screen should be covered with greasy finger-marks at 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” when they have already been taking the initiative for a while and the person with the keyboard is not quite keeping up with their dictation.

This view of successful pair programming is confirmed in recent work by Chong and Hurlbutt [4], who spent several months on an ethnographic study of professional developers using pair programming in their everyday work. She found that the two programmers work together on the same facet of the problem almost the whole time, and they swap between the tactical and architectural levels *as a pair*. Although one or the other of them will be taking the initiative at a particular time, in no sense is there a division of labour between local and architectural design. This has been further confirmed in similar studies by Bryant et al. [5] and Salinger et al. [6].

The “driver-navigator” metaphor is therefore wrong. Attempting to follow it may have deceived some programmers into practices which were far from those used by successful pair programmers, and which were therefore of little benefit. The failure of the “driver-navigator” metaphor even casts doubt on the many attempts to assess the effectiveness of pair programming (summarised very nicely in a paper by Dybå et al. [7]). If the subjects of these experiments were doing different things, can we really compare their results? And if they were not doing what successful pair programmers do in commercial practice, can we really apply their findings back to commercial development?

If we want to assess the true effectiveness of pair programming, we need a new theory which can explain when pair programming will be practically useful and when it will not. Only by controlling for the factors which truly determine success can we establish when pair programming will be useful. This article suggests four psychological mechanisms which may form part of such a theory. Let us now turn to

the first of these proposed mechanisms.

Mechanism 1: Pair programming chat may make it easier to integrate existing knowledge about a program

Around 1980, while I was an undergraduate studying computer science at the University of Cambridge, my friends and I noticed a strange phenomenon, which we called “expert programmer theory”. One of us would be having trouble getting our program to work, and sitting as we did in a communal area with terminals attached to the university mainframe, we would break off and get a drink from the coffee machine. We would describe the current non-functioning state of our code to each other, and quite often we would realise in a flash what was actually going wrong and how to solve it. This moment of epiphany was quite independent of any real understanding of our problem by the person we were talking to: it wasn’t as if they followed our tale closely and then told us the answer. Often the person whose ear we were bending seemed little wiser about the problem or the solution after the whole exchange.

Over the following years, I have found that this phenomenon is well known to all professional developers, and sometimes described in textbooks and research papers (for example [8] and [9]). Going by the name “rubber plant effect” (even talking to a rubber plant may help), and “wooden Indian effect” (they don’t usually talk back), and presumably by many other names, this programmer folk-knowledge helps to remind developers how useful it can be to talk about problems out loud. Part of the effectiveness of pair programming is presumably due to this effect being continually triggered: as one of the pair gets stuck, their back-and-forth chat serves to “unstick” them in just the same way as solo programmers chatting around the coffee machine. But this begs the question of whether any kind of talking out loud will help, or whether something specific is needed.

Research on “self explanation” by Chi and others throws some light on this question. For example, Chi et al. [10] describe a study of student learning in which a control group of students were tested, given a textbook explanation to read and then tested again. Another group was tested in the same way, but they were also strongly encouraged to explain the textbook out loud, and “fill in the gaps” for themselves. The self-explainers learned significantly more than the control group, and the ones who explained the most improved the most. Interestingly, in this experiment the students were prompted for their explanations by the experimenters, they were not 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 aspect of “expert programmer theory” which often seems

to be neglected. Back in the 1980s when we coined that term, we noticed that although real understanding was not necessary on the part of the listener, a belief that the listener really was an expert seemed to significantly improve the outcome (hence our choice of name). And of course, talking to a real expert was most successful of all. We joked that perhaps if you believed strongly enough that you were talking to an expert, it wouldn't really matter who or what you talked to. Maybe a cardboard cut-out would do, or a potted plant. (As part of this joke I actually bought a rubber plant and this subsequently gave rise to the name "rubber plant effect" amongst some programmers in Cambridge.)

But joking apart, why would talking to an expert or believing that you were talking to an expert make any difference, when they didn't need to understand your explanation? The role of prompting questions seems to be the key, as is illustrated by more recent work by Roscoe and Chi [11]. In this study, one student (the "tutor") explained material to another student (the "tutee"). As expected, the tutor actually learnt more than the tutee, but interestingly the questions asked by the tutee made a dramatic difference to quality of the explanations from the tutor. Most questions were "shallow", and could be satisfied by mere repetition of facts, but some questions were "deep", and these often prompted "deep" answers which 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 are talking to an expert, or making believe, will act as a cue and remind the "stuck" programmer of 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 really a good analogy for what happens to the "stuck" programmer? After all, the students in these experiments are given basic science to master, and their explanations help them work out what they don't yet understand. The "stuck" programmer must already have all the facts in their head, but somehow hidden, and yet in a moment of epiphany they are revealed. How is this possible? With a little further exploration we find a plausible answer.

It is 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, face recognition, and so on. Less well known is the role of the language module in integrating the knowledge from other modules. Experiments by Hermer-Vazquez et al. [12] on integrating knowledge about geometry and colour, and by Newton & de Villiers [13] on false-belief reasoning show that adults perform as poorly as young children when their linguistic abilities are occupied with a verbal "shadowing" task. The language module

seems to be crucial to combining knowledge from other modules.

This is not to say that we integrate the outputs of several mental modules by talking to ourselves consciously in our heads. Rather, the suggestion of Carruthers [14] is that since speech is uniquely both an input and output medium for the brain, the language module is the only one with a strong connection to all the other modules. The mechanisms underlying the “logical form” of language may thus be redeployed at a level beneath conscious awareness to achieve integration of information from other modules. The “logical form” must be able to represent objects with properties derived from several modules, since this is the basis for noun-phrases in speech.

As programmers, we clearly use visual imagination to help us design and debug our programs (though the diagrams we use bear little relation to the texts of our programs). This visual information may itself be spread across several mental modules, and further information necessary to understand our programs may be in yet other modules. For example, it seems possible that our understanding of object oriented programs is supported by the “folk psychology” module which supplies intuitions about the actions of other people. (We think of objects as having intentions, wanting to do things, sending each other messages.) So we must often need to integrate information from separate modules when thinking about our programs. Why can we not always integrate it straightaway?

Carruthers suggests that we must rely on the language module posing the right question, and that the other modules do not usually present information spontaneously. However, when the right question is asked, the necessary information is made available, and the language module is then able to perform very rudimentary inference, and draw the now obvious conclusions. Carruthers suggests that the key is posing a question which is “both relevant and fruitful”[14], a question which will elicit the knowledge which is already there, but isolated in different modules and unable to be integrated. The right question draws forth the crucial knowledge, and then in a moment of epiphany the answer to your problem is now obvious.

So, finally, we have a plausible explanation for the power of “deep” questions, which in turn explains “expert programmer theory” and perhaps explains how pair programming chatter can help to solve programming problems. The key may be the occasional “relevant and fruitful” question, a “deep” question which prompts this integration of existing knowledge. This first mechanism would lead us to predict that those programmers who chat about their program more should be more productive, and in particular that those who pose the occasional “deep” question for each other should be most productive of all.

Do all pair programmers chat about their programs like this, posing the occasional “deep” question, and getting each other “unstuck”? It appears that the successful ones

do. However, there are clearly others who do not [3], and their pair programming is less successful. Perhaps they took the “driver-navigator” metaphor too literally, or perhaps they had pair programming forced upon them, and they are just going through the motions. To what extent were the “navigators” in pair programming experiments merely watching quietly as the “driver” programmed, because they thought that was how it was done?

Mechanism 2: Pair programmers may notice more details about their work

Having discarded the “driver-navigator” metaphor, we can see that the pair of programmers will almost all the time be concentrating on the same things. As the saying goes, “two pairs of eyes are better than one”. Obviously two people will tend to notice more things than a single person. However, it is not always understood just how bad we are at noticing things. 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, then we can be staring right at it and yet not see it. What we notice depends crucially on what we expect to see, and what we unconsciously consider salient.

Research on change blindness has shown that people are remarkably poor at detecting changes, not only in 2D images under laboratory conditions, but in real life situations like 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 part of the expertise of experts is probably in their more detailed and extensive array of mental categories in their particular field [16]. Research on inattentional blindness has similarly shown that when our attention is focussed 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 you would miss a woman in a gorilla suit walking into shot in a video, but that is just what half of the subjects did in a study by Simons & Chabris [17]. (They had been instructed to pay close attention to another aspect of the video.)

As shown by the experiments mentioned above, our experience and expectations can sharply limit our ability to properly notice what we are looking at. Two people programming together will not have exactly the same prior knowledge, not exactly the same categorisation, so we can expect that one of them will spot some things faster, the other some different things faster. Where their rate of working is limited by the rate at which 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 is not currently typing code always picks up

typos quicker: “Oh, you have left out the comma here,” they will say to their partner at the keyboard.

Of course, the compiler would pick up such small slips easily, so in this case the early catch is not very important. However, it is very important to catch problems early when the slip is more subtle: for example if the code is syntactically correct but semantically wrong, or where there is a fault in the design itself. Such slips can easily cause hours of problems at a later date. The ability to catch mistakes early, as it were in an “on-line code-review”, is only part of the benefit of two pairs of eyes: perhaps even more important is the ability to look at old code with a fresh eye, and a different set of expectations, reading what it really says, not what we assume it ought to say.

Mechanism two also partially explains the phenomenon of “pair fatigue”, which I have noticed in myself and other pair programmers. As a pair works together, the things they notice and the things they fail to notice become more and more similar. Eventually, the benefit that they gain from two pairs of eyes is negligible. Beck suggested in [1] that pairs should rotate at frequent intervals: perhaps once or twice a day. In [18] he suggests rotating pairs even more frequently: every two hours or so. Beck doesn’t explain precisely why this is a good idea, and some pair programmers regard rotation as an optional part of the practice. (I have spoken to some pair programmers who have worked together for days on end. On a small team, or with few programmers willing to pair, there may be little alternative.) However, mechanism two should lead us to predict that frequent rotation will be beneficial, since it avoids “pair fatigue”.

Of course, experts are at an advantage as far as categories are concerned. A great deal of expert knowledge is probably in the form of categories in long-term memory, and without this knowledge a novice may be unable to distinguish between events experienced at different times. Experts really can see things that novices can’t. We would therefore also predict that mechanism two will bring the maximum benefit to pairs of novices, and the most extensive experiment with pairs of novices and experts appears to confirm this [19]. However, when categories converge during pair programming this means that some learning is taking place, and as we will also see in connection with mechanism number four, this should have a longer term benefit.

Mechanism 3: Pair programmers may be better at fighting their addiction to poor practices

As programmers, we don’t always do what we believe to be best practice. An advantage of pair programming is said to be “pair-pressure” [20], the feeling of not wanting to let your partner down. But why should this be necessary? Why do we persist in poor programming practices when we know they are poor? Is there

something special about programming which makes it more difficult to do the right thing? It appears that there is.

Let us look more closely at a particular example of “worst practice”: the “code-and-fix” style of programming most often used by novices. (And sadly, too often used by more experienced programmers.) The programmer writes some code which they hope will do a particular thing, then they run it to see what happens. If it appears to work, they press on with other code, without systematically trying to find flaws. When it fails, which is often the case, they tinker with the code in a haphazard way until it appears to be working. Why is this style of working so compelling, and so easy to discover independently?

A very plausible explanation is to be found in traditional behavioural psychology, though more modern work on the neuroscience of learning and addiction also points in the same direction. One form of learning explored by behavioural psychologists, called “operant conditioning”, involves learning to perform some action spontaneously. This is the way that animals are taught to perform tricks in circus acts or taught to “be obedient” if they are domestic dogs. An animal has a variety of behaviours which it is prone to engage in occasionally, and with operant conditioning we supply the animal with a reward after we observe it doing what we want. (We are said to be “reinforcing” the behaviour.) As this process of reinforcement continues, the desired behaviour becomes more and more likely to happen spontaneously, even when no reward is given afterwards.

Of course, if the rewards stop entirely the behaviour will diminish and finally cease, a process known as “extinction”. However, extinction happens quite slowly. If we supply a further reinforcement before the behaviour has entirely ceased, we can easily restore it to full strength. In fact learning happens quickest if the pattern of rewards is unpredictable, with a so-called “variable ratio” or VR schedule of reinforcement. As explained by Gleitman et al.:

“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 realise that it is happening to us, and in the case of the casino, the slot-machine patrons are being conditioned by a machine, not

even by a real person! In our habitual patterns of software development we too can be conditioned by our machines. This is the special property of interactive programming which makes it difficult to just “do the right thing”. With code-and-fix, we tinker haphazardly with our program, effectively putting a coin into the slot-machine each time we run our code. Slot machines are known to be the most addictive form of gambling, and the similarly unpredictable rewards from code-and-fix programming could mean that in a real sense we too can become addicted to bad programming habits.

How can we resist this addiction? We can perhaps, along with other addicts, “just say no”, and choose a different pattern of development. Perhaps some form of “twelve step programme” is possible? Some development processes attempt to remove temptation by being less interactive. Dijkstra even suggested that students should not be allowed near a computer until they had properly learned to write programs away from one [22]. Such ideas may once have had merit, but it seems foolish to turn our backs on the orders-of-magnitude increase in computer power available to us since then, and which we can exploit if we feel strong enough to engage in more interactive programming practices. (Test-first-development is one such practice, linked with pair programming in the Extreme Programming methodology.)

Pair programmers may be less susceptible to poor practices because they can make promises to write code in a particular way, and sit next to each other to make sure that the promises are kept. The prevalence of “two-man working” in jobs where human fallibility is a serious problem should lead us to seriously consider that “pair-pressure” may be the solution for us too. It you know what you want to do, it’s easier to carry though on the intention if you make a promise to another person and even easier if that person is there with you.

Mechanism 4: Pair programmers may judge expertise better and transfer expertise better

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

In some fields it is easy to recognise experts because individual contributions are easy to measure. Chess players have a numerical ranking; golfers have a handicap. This is a good predictor of their likely success against other players in the future. But in team activities, so many factors contribute to success or failure that we simply cannot understand the causal relationships without a detailed scientific investigation [24]. So

we usually select one or two arbitrary factors, for the sake of simplifying the analysis. In software development, “lines of code written per day” often gets elevated above all others, simply because it is easy to measure. But selecting such arbitrary factors tends to promote “star players” who demonstrate those particular qualities, but who may in reality be merely “good-lookers” and not actually contribute significantly to the team’s success.

However, a more detailed, more scientific analysis would seldom be practical or economic. So how can we assign individual credit (or blame) for team performance? Graham [25] suggests one way out of this quandary: when an expert programmer works alongside another programmer on the same problem, the expert can tell how good the other programmer is (by “hacker” he means an expert programmer):

“So who are the great hackers? How do you know when you meet one? That turns out to be very hard. Even hackers can’t tell. ... The problem is compounded by the fact that hackers, despite their reputation of social obliviousness, sometimes put a good deal of effort into seeming smart. ... With this amount of noise in the signal, it’s hard to tell good hackers when you meet them. I can’t tell, even now. You also can’t tell from their resumes. It seems like the only way to judge a hacker is to work with him on something.” [25]

This is my experience too. It isn’t enough to talk about programming with someone, you have to work on a problem with them to know how their expertise compares with your own. A weak version of this technique is standard practice in programming interviews. After the preliminary discussion centred around the applicant’s resume, the interview proceeds on to a series of successively more difficult programming exercises which the applicant has to talk through at a whiteboard. I have frequently been surprised how a very plausible sounding candidate, when challenged in this way, completely fails to produce even the most basic evidence of the knowledge that they earlier claimed.

The poor candidates seem blissfully unaware of their own lack of expertise. They are so bad that they don’t realise how bad they are, probably because, in the words of Kruger and Dunning [26], “the same knowledge that underlies the ability to produce correct judgements is also the knowledge that underlies the ability to recognise correct judgement”. In a field where expertise is hard to measure this is a serious problem, because as Kruger and Dunning observe, 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, called the “false consensus effect”, where they believe that their own abilities are typical. This happens for the same reason: it is hard to accurately assess the competence of others,

so the most competent have no reason to believe that they are out of the ordinary — unless they work closely alongside 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 on problems and since they keep swapping pairs, everyone on a team becomes clearly aware of who is most expert at particular things. From this comparison, they also learn how expert they are themselves. We should therefore expect that estimates of time and difficulty produced by a pair programming team will be more accurate than a team which uses solo programming. (From my own experience, this does appear to be the case.)

But more than this, pair programming can also build expertise: with expert-novice pairings the craft skill of programming is transferred in the old fashioned way, with master and apprentice working side by side. The prerequisite for this transfer of knowledge is of course the recognition within a team of who is the master of particular knowledge and who is the apprentice. (And of course an expert at one thing can be a novice at another.)

Conclusions

We started with the observation that the “driver-navigator” metaphor fails to describe what happens in successful pair programming teams. The four mechanisms suggested in this article start to build a theory of pair programming which could explain when the practice is effective and when not. Certainly, practical investigations of pair programming should bear these mechanisms in mind, since if their effects are not controlled for, the conclusions of the experiments may be thrown into doubt.

We can also see that even within the same team, the mechanisms would bring different benefits in different situations. The most critical mechanism is probably the first: if you never chat to your partner about the program then you can hardly claim to be doing pair programming at all. The other mechanisms would vary in their applicability and helpfulness, generally bringing more benefit to pairs of novices than pairs of experts. However, expertise is hard to judge, so the fourth mechanism may be the most crucial to the overall success of projects, as well as to the progression of programmers from novices to experts.

References

1. K. Beck, *Extreme programming explained: embrace change, 1st Ed.*, Addison Wesley, Boston. 2000.

2. L. Williams and R. Kessler, *Pair Programming Illuminated*, Addison Wesley, Boston. 2003.
3. M. Stephens and D. Rosenberg, *Extreme Programming Refactored: The Case Against XP*, Apress, Berkeley, 2003.
4. J. Chong and T. Hurlbutt, "The Social Dynamics of Pair Programming," *Proc. 29th Int. Conf. on Software Engineering*. IEEE Computer Society 2007, pp. 354-363.
5. S. Bryant, P. Romero and B. du Boulay, "Pair programming and the mysterious role of the navigator" *Int. J. Human-Computer Studies*, Vol. 66, No. 7, July 2008, pp519-529.
6. S. Salinger, L. Plonka and L. Prechelt, "A Coding Scheme Development Methodology Using Grounded Theory for Qualitative Analysis of Pair Programming," *Proc. 19th annual workshop of the Psychology of Programming Interest Group*, July 2007, pp. 144-157.
7. T. Dybå, E. Arisholm, D. Sjøberg, J. Hannay and F. Shull, "Are Two Heads Better than One? On the Effectiveness of Pair Programming," *IEEE Software*, Vol. 24 No. 6, Nov/Dec 2007, pp. 12-15.
8. B. Kernighan and R. Pike, *The Practice of Programming*, Addison Wesley, Reading, Mass., 1999.
9. J. Sturdy, "Sidebrain: a sidekick for the programmer's brain," *Proc. 17th annual workshop of the Psychology of Programming Interest Group*, June 2005, pp. 215-226.
10. M. Chi, N. de Leeuw, M. Chiu and C. LaVancher, "Eliciting Self-Explanations Improves Understanding" *Cognitive Science*, Vol. 18, 1994, pp. 439-477.
11. R. Roscoe and M. Chi, "The influence of the tutee in learning by peer tutoring," *Presented at 26th annual conference of the Cognitive Science Society*, August 2004.
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, Aug. 1999, pp. 3-36.

13. A. Newton and J. de Villiers, , “Thinking While Taking: Adults Fail False-Belief Reasoning,” *Psychological Science*, Vol. 18, No. 7, July 2007, pp. 574-579.
14. P. Carruthers, “The cognitive functions of language,” *Behavioral and Brain Sciences*, Vol. 25, No. 6, Dec 2002, pp. 657-726.
15. D. Simons and D. Levin, “Failure to detect changes to people during real-world interaction,” *Psychonomic Bulletin and Review*, Vol. 5, No. 4, Dec. 1998, pp. 644-649
16. N. Charness, P. Feltovich, R. Hoffman and K. Ericsson, *The Cambridge Handbook of Expertise and Expert Performance*, Cambridge University Press, 2006.
17. D.J. Simons and C.F. Chabris, “Gorillas in our midst: sustained inattention blindness for dynamic events,” *Perception*, Vol 28. No. 9, 1999, pp. 1059-1074.
18. K. Beck, *Extreme programming explained: embrace change, 2nd Ed.*, Addison Wesley, Boston. 2005.
19. E. Arishholm, H. Gallis, T. Dybå and D.I.K Sjøberg, “Evaluating Pair Programming with respect to System Complexity and Programmer Expertise,” *IEEE Trans. on Software Engineering*, Vol. 33, No. 2, Feb. 2007, pp. 65-86.
20. L. Williams and R. Kessler, “The effects of “Pair-Pressure” and “Pair-Learning” on Software Engineering Education,” *Conference on Software Engineering Education and Training, 2000*.
21. H. Gleitman, A.J. Fridlund and D. Reisberg, *Psychology*, 6th Ed., W.W. Norton & Co., New York, 2004.
22. E. Dijkstra, “On the cruelty of really reaching computing science,” 1988. Available online at <http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF>, Accessed 17 Jul. 2008.
23. R. Glass, *Facts and Fallacies of Software Engineering*, Addison Wesley, Boston. 2003.
24. M. Gladwell, “Game Theory,” *New Yorker Magazine*, 29 May 2006.

25. P. Graham, "Great Hackers," 2004. Available online at <http://www.paulgraham.com/gh.html>, Accessed 22 Oct. 2007.
26. J. Kruger and D. Dunning, "Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments," *Journal of Personality and Social Psychology*, Vol. 77, No. 6, Dec. 1999, pp. 1121-1134.

About the Author

Stuart Wray is a senior lecturer at the Royal School of Signals in Blandford. His research interests include psychology of programming, computer security and functional programming. He received a BA in computer science in 1981 and a PhD in computer science in 1986, both from the University of Cambridge, U.K. Since then he has worked in research, at ORL and the University of Cambridge Computer Laboratory, and in product development, at Virata, Marconi and BAE Systems. Contact him at swray@bournemouth.ac.uk.