On the nature of pair programming

Stuart Wray

swray@bournemouth.ac.uk

DRAFT

Abstract

In this paper, I describe the practice of pair programming and explain four mechanisms underpinning the practice. By explaining what makes pair programming work, these mechanisms should help software developers and managers to adopt pair programming more successfully. Finally I present an explanation for the unusually vocal resistance to pair programming exhibited by some developers.

Introduction

Usually, people write programs while sitting alone at a desk or a computer screen. Certainly, they *discuss* their programs with others in small groups, drawing diagrams and equations on whiteboards, scraps of paper and napkins. But the actual line-by-line writing of programs has always been seen as a solitary activity. *In extremis* programmers will ask others for help, but it is seen as a point of honour that a professional developer should tackle a job entirely on their own. To admit that one enjoys writing code line-by-line in the close company of another human is at best seen as a wasteful foible, at worst a sign of incompetence.

In the relatively recent past, this conventional wisdom has been challenged by enthusiasts of pair programming. As the name suggests, pair programming is a technique for writing programs line-by-line where two people sit down, literally side-by-side at the same computer. This is an easier technique to demonstrate than to describe, as is clear by some of the garbled accounts and polemics against pair programming which still circulate in the development community. It is *not* a technique where one person programs and the other person watches. The two programmers work very closely together, talking 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.

Some texts, for example Williams & Kessler (2003), Chong et al (2005), describe one person assuming the role of "driver", thinking of tactical issues, while the other acts as "navigator" or "observer", concentrating on more strategic architectural concerns. Perhaps this description appealed to some writers when they were attempting to explain what happens in pair programming, but despite its wide currency this description is far from the truth. My personal experience is confirmed by recent ethnographic analysis of pair programming sessions (Chong 2006). What actually happens is 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. Both programmers work on both levels together.

Surely, one might think, if two people write a program together in a pair, it will take a similar time to one of them working on their own, so their productivity will be halved. If that were so, then pair programming would be quite a hard sell. However, personal experience, anecdotes and research all point to a different conclusion: in fact, two people programming as a pair will produce a *working* program in a much shorter elapsed time than a person programming alone. The only real difference in individual productivity is that the pair programmers' program will contain fewer latent bugs than the program written by an individual programmer. In short: pair programming is actually about as productive as solo programming.

What does it feel like to program in a pair? To understand this, you have to first understand what it is like to fluently program on your own. Programmers work most productively when they are in a state of "flow". This is a "trance-like" state of concentration which Csikszentmihalyi (1990) describes as able to occur when engaging in an activity where one has a chance of success, where one is able to concentrate, where one's goals are clear, and where feedback as to success or failure is immediate. All professional developers are able to describe times where they have programmed in flow for hours, falling out of this state at some point to realise that their coffee is cold, their legs are stiff and their bladder is full. They have worked with intense concentration for hours and yet somehow time seemed not to pass.

Programming is of course just one instance of flow; Csikszentmihalyi describes many others. Athletes experience flow, as do artists and writers. The central role of flow in computer programming was pointed out by DeMarco and Lister (1999). It takes a while to get into flow, perhaps 15 or 30 minutes when everything is going right, but it can be disturbed by interruptions of very short duration. Just a few interruptions in a day can dramatically reduce the time spent in flow, and hence substantially reduce the productivity of programmers. "Don't talk to the programmers!" seems to be the message of DeMarco and Lister, and this has been taken on board by many development teams as the ideal to which they should aspire. (Although if it were taken more seriously by managers, we might see fewer cubicle farms and more offices with doors.) Pair programming seems to be completely at odds with DeMarco and Lister's message. Surely pair programmers will be disturbed by their constant chatter and never get into flow? In

fact it seems to be the opposite.

We are now in a position to describe what it feels like to program in a pair: it feels like being in flow with another person. Because, I believe, it is being in flow with another person. The constant chatter does not disturb, because it is about the subject on which they are both concentrating. The chatter tends to draw their attention back to the subject when it wanders. A pair is also in fact more resistant to outside interruptions than a single programmer: usually only one person attends to the interruption, and afterwards their concentration is steered back to the problem quickly by the other. Perhaps outsiders are also kept at bay by a social pressure: when you see two people in such rapt and intimate conversation, you naturally feel unwilling to butt in.

So, rather than disturbing the flow of programmers, pair programming can actually enhance it. But I believe that this effect is an epiphenomenon, a consequence of the four mechanisms which underpin pair programming rather than a principal cause. Let us therefore examine these four mechanisms in turn.

Mechanism 1: Pair programming chat makes it easier to access exisiting knowledge about a program

Programmers have knowledge about their program in a variety of "cognitive modules", notably in those modules dealing with visual images. In a theory by Carruthers (2002), linguistic abilities of the brain are needed to integrate knowledge from diverse cognitive modules. Interfering with available linguistic processing has been demonstrated to impede such integration (Hermer-Vazquez et al 1999). On the other hand, talking about a particular topic will elicit responses from a variety of cognitive modules and seems likely to promote their integration. This appears to explain the widely known effect sometimes called "Expert Programmer Theory", and hence the effectiveness of pair programming chatter. I will return to Carruthers' ideas and the modular theory of mind after describing "Expert Programmer Theory".

Around about 1980, the group of hackers that I hung out with at university noticed a strange phenomenon, which we subsequently called "Expert Programmer Theory". The same phenomenon has, I know, been observed many times by many groups of software developers, and given many names. Another term I have heard for it is "The Rubber Plant Effect" (Sturdy 2005), a strange name but one whose derivation will become clear in a moment.

The phenomenon was as follows: one of us would be having trouble getting a 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. No, what happened was stranger. Somehow, telling the story to them let us see the solution for ourselves. Often the person whose ear we were bending seemed little wiser about the problem or the solution after the whole exchange.

But it wasn't as if you could just talk to anyone at all about your program: it seemed as if you had to believe that the person you were talking to really could understand what you were telling them, that they were in short an "Expert Programmer". Hence the name, "Expert Programmer Theory". However, it wasn't lost on us that in fact real understanding was not necessary on the part of the listener. It appeared that it was more important to have real belief on the part of the explainer. Thus we joked that, provided that you really believed that you were talking to an Expert Programmer, then a life-size cardboard cut-out would do just as well. Or even a rubber plant! (To continue the same joke, when I became a PhD student I bought a small rubber plant. I used this as a prop while explaining Expert Programmer Theory to other people. The same phenomenon subsequently went by the name "The Rubber Plant Effect" amongst one group of Cambridge software developers.)

This phenomenon of seeing the solution to a problem after explaining it to someone who *could* understand it may partially explain what is going on in pair programming. After all, the pair talk to each other all the time about the program they are working on together. Could it be that pair programming is actually "Continuous Expert Programmer Theory"? Perhaps so, but that "explanation" really begs the further question of what is actually happening in Expert Programmer Theory.

Fortunately, there is a very plausible explanation based on Carruthers' (2002) work, but to set the scene we first need to look at the modular theory of mind, and the work of Hermer-Vazquez et al (1999) on linguistic integration of modular knowledge.

There are good reasons from developmental and evolutionary psychology for supposing that the capabilities of our minds are organised into a number of "modules" which are specialised for dealing with particular kinds of information and whose workings are largely isolated from each other. Amongst the possible candidates for such modules are the early processing of visual stimuli, face-recognition, naïve physics, intuitive grasp of small numbers, and so on. Most of these abilities appear to be innate, independent of other abilities and not learned, but rather emerge as a child develops:

> "Very young infants already have a set of expectations concerning the behaviours and movements of physical objects, and their understanding of this form of causality develops very rapidly over the first year or two of life. And folk-psychological concepts and expectations also develop very early and follow a characteristic developmental profile." (Carruthers, 2002)

Of course, language itself is another candidate module, albeit an exceptionally well connected one, since it too is largely innate (Pinker, 1994) and develops in children independently of other modules.

Computational psychologists also argue for modularity, but on the grounds of computational tractability. It is generally accepted that the mind arises as a result of computational processes in the brain, but how are these computations organised? The "connectionist" theory asserted that the brain was a largely amorphous body, engaging in generalised pattern-matching, without local representations of knowledge. However, this theory is at odds with experimental evidence of localised activity in the brain and it is also at a loss to explain the "one-shot" learning of which most animals are capable. From an engineering point of view, it is also hard to see how an information processing system like the brain could function without tight encapsulation of function and limited interconnection between components. This is certainly the lesson which has been learned from artificial intelligence research.

Hermer-Vazquez et al (1999) provide evidence for the surprising role of language in integrating the capability of different modules in humans. Their work follows from earlier experiments on the navigational abilities of rats (Cheng 1986) and young children (Hermer & Spelke 1994, 1996). Their first experiments were on rats who were shown food being buried in one of the four corners of an otherwise featureless rectangular enclosure. The rats were then taken away and disoriented. As expected, when returned to the enclosure they were equally likely to look for the food in the correct corner and in the geometrically indistinguishable opposite corner. However, when an extra cue, such as different lighting, a scent or patterns on the walls, were added to the enclosure to make it non-symmetrical, the rats still looked half the time in the geometrically equivalent opposite corner. When disoriented, they *only* used geometric information to re-orient themselves, even though they were entirely capable of responding to other cues in other circumstances. (This does make some sense: geometry changes slowly in the natural habitat of the rat, and entirely symmetric natural landscapes are very unusual.)

Interestingly, the next experiments on young children looking for a toy in a small rectangular room showed exactly the same effect: the children used only geometry to reorient themselves. They were equally likely to look in the geometrically equivalent corners, even if one wall of the room was a different colour, breaking the symmetry. Adults in the same situation can easily reorient themselves properly and hence select the correct corner. However, the ability to do this in older children was found to correlate strongly with only one thing: the productive use of the terms "left" and "right" in their language. (That is to say that they understood the terms properly and could make use of them in their own speech.) It thus appeared that the ability to combine information on geometry and on colour, presumed to lie in separate cognitive modules, was mediated by language in adult humans.

Hermer-Vazquez et al (1999) describe a beautiful series of experiments which confirms this hypothesis. Adults were set the same task as the children, but were trained to simultaneously perform "verbal shadowing", where they continually repeated back speech that was played to them over loudspeakers. It was hoped that this interfering task would occupy their linguistic abilities, so that they would be unable to integrate the information on geometry and colour. This was exactly what happened: they were no better than rats or small children at identifying the correct corner. To rule out other

explanations, the experimenters performed various other experiments. They trained people to "shadow" rhythms, and established that this was at least as challenging as shadowing speech, then performed the same reorientation test while shadowing rhythms. The subjects were able to identify the correct corner. They also ruled out the possibility that speech shadowing was interfering with the ability to register colour alone. Which left the only explanation: that in adults, when disoriented, we really do rely on our linguistic ability to integrate the outputs of at least these two cognitive modules.

Carruthers suggests a reason why the linguistic mechanisms in the human brain may be uniquely connected to so many other modules. Speech is used as an output method of the brain: we need to be able to say things based on the specialist knowledge of many modules. Speech is also an input mechanism: we use our understanding of the speech of other people to influence many otherwise separate modules. It thus appears that there is a very good reason for a linguistic module to be strongly connected to both the inputs and outputs of many other modules. Which is not to say that the other modules are entirely separate: many point-to-point connections clearly exist, but it seems that language is the only "broadband" connection between modules. (Carruthers makes a good case for there being no other similar connection, based on evolutionary simplicity: given that language does this, how could another redundant mechanism evolve? And if there were another connection, surely the reorientation experiment would not have shown the results it did.)

This is not to say that we integrate the outputs of several modules by consciously talking about them in our heads. Such "inner speech" does of course happen a lot of the time and is probably related, but the mechanisms proposed by Carruthers are principally unconscious. He proposes that the mechanisms which handle the underlying "logical form" of language are redeployed to accomplish the integration between modules. (In much the same way that the parts of the brain used to analyse visual images we see with our eyes are redeployed when we imagine a scene.) The "logical form" (LF) can represent objects with properties derived from several separate modules (for example a corner with a short blue wall on the left) since this is the basis for noun-phrases in speech.

Programmers clearly use visual imagination to help them to design and to debug programs, although the diagrams used by modern programmers often bear little relation to the texts of their programs. From the reorientation experiment it should be clear that there is no guarantee that this visual imagination is confined to one module, in which case integration of several pieces of visual knowledge must be mediated by language. When thinking about a program, this integration of information from visual modules may be the most important, but clearly other modules must be also used by programmers:

- The "theory of mind" module (otherwise known as "mind-reading", "folk-psychology" or "naïve psychology") is what we use to attribute motives to other people, and to predict their actions. Interestingly, this module appears not to have privileged access to our own motivations: we are prone to confabulation when describing why we do things, and are really no better at explaining our own motivations than those of others (Gazzaniga 1998). Programmers often talk about parts

of a system "wanting" to do this or "trying" to do that. If we build systems whose parts act in ways that are in line with our intuitions about people, they will be easier to comprehend, since we will be directly using this module. This perhaps explains the success of object oriented programming, where we imagine that our programs consist of largely independent agents who have particular things that they ask each other to do.

- The "folk-biology" module is used for attributing properties and behaviours to non-human animals and plants: it is the reason why we can construct taxonomies as easily as we can. When we deal with systems whose parts can be fitted into a tree-structured taxonomy (as will usually be the case with naturally evolved species) we will be able to use our folk-biology intuitions. This is perhaps the reason that single-inheritance in object oriented languages is simple to understand, but multiple-inheritance always causes confusion. Single-inheritance is exactly what the folk-biology module is set up to understand: it produces a tree-like taxonomy similar to one produced by evolution. Multiple-inheritance on the other hand produces composite artefacts, like a horse with wings, that could never arise in nature, and about whose properties we therefore have little intuition.

We are now, at last, in a position to return to and explain our earlier questions about Expert Programmer Theory. Carruthers suggests two crucial ramifications of his ideas, relating to cycles of "logical form" (LF) activity and LF consumers. On the former, Carruthers says:

> "So the suggestion is that language, by virtue of its role in unifying the outputs of conceptual modules, and by virtue of our capacity for auditory imagination, can be used to generate cycles of central-modular activity, hence recruiting the resources of a range of specialised central-modular systems in seeking solutions to problems. This may be one of the main sources of the cognitive flexibility and adaptability which is so distinctive of our species. But how, exactly are the LF questions which are used in such cycles of enquiry to be generated? How does the language system formulate interrogative sentences which are both relevant and fruitful?" (Carruthers, 2002)

Carruthers admits he has no answer to this, and neither do I, in general for people working alone. If you have been working on a problem for a long time and are stuck, this surely indicates that you were not generating the right LF questions. However, an "Expert Programmer" *asking the right questions* can get "interrogative sentences which are both relevant and fruitful" into the head of a programmer who has until now been stuck. These new questions can drag forth the knowledge which was there, but separate in different modules and unable to be integrated.

This brings us on to Carruthers' second ramification, on LF consumers. He suggests that there is a "domain-general reasoning system taking LF sentences as input and generating LF sentences as output". This could be a very restricted ability to rewrite LF sentences, combining information in a very simple way, but well short of a general purpose

inference mechanism. Humans appear to have many domain-specific heuristics, approximating proper Baysean inference (Gigerenzer & Todd, 2000), so:

> "a set of heuristics and implicit procedures (together with the limited inferential powers of the language faculty) could collectively provide a sufficiently good approximations of logical ability." (Carruthers, 2002)

What would it feel like to have the right information about a program pulled out of the specialised modules by an Expert Programmer's question, and then have these "limited inferential powers" draw the now obvious conclusions? I think it would feel very much like that "moment of epiphany" that I described earlier. So this is how I think that Expert Programmer Theory actually works, and how this mechanism underpinning pair programming is effective. Pair programming chat contains questions and unexpected statements which continually provoke the other programmer to extract and integrate knowledge about the program which would otherwise have lain unnoticed in separate modules.

This also explains why the Rubber Plant or "cardboard cut-out programmer" from the jokes about Expert Programmer Theory aren't really used much in practice. It's the questions the Expert produces which are vital to the effect. Only an expert will ask questions that prompt an integration of knowledge and hence the moment of epiphany. Of course if you are an expert yourself, then you can probably imagine the kinds of questions that an expert would ask, and explicitly ask them of yourself, using your rubber plant as a stand-in. If you are sufficiently expert and are not too embarrassed to ask and answer the questions out loud you may get some benefit. But it would be easier to work with a real person!

Implications for software development

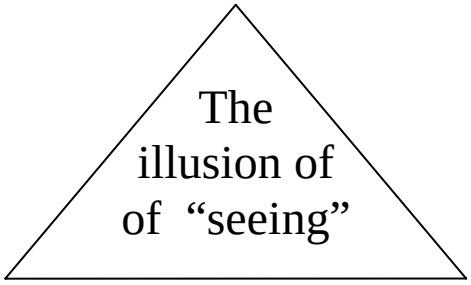
The chat directly related to the code being written is key to integrating the programmers' knowledge of the program's operation, or in other words, continuously triggering the Expert Programmer Effect. Pair programmers have to talk, and they have to talk about the work in hand. If people claim to be pairing, but are not continually chatting about their program, then they are mistaken. They are not pair programming: they are just sitting together in front of a computer.

Mechanism 2: Pair programmers notice more details about their work

In a sense this mechanism is the dual of mechanism one, which hinged on the fact that we might have all the relevant information in our brain, but not be able to integrate it. Mechanism two relates to the phenomenon that if we don't know what is salient, we can fail to get the information into our brain in the first place or to keep it there long enough to be integrated. Because a pair of programmers have some divergence in what they consider salient, they will notice slightly different things, so between them they will notice more than one person working alone. However, over time their views of salience

become more similar as they learn from each other. Mechanism two therefore also explains the phenomenon of "pair fatigue", where a pairing becomes less effective over time, hence the standard practice of rotating pairs at frequent intervals. (My experience has been that pairings of a few hours work well, and anything more than a day is hopeless. Beck (2005) suggests pairings of a couple of hours or less.)

It is a commonplace observation that two pairs of eyes are better than one. Obviously a pair of 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. Work on "change blindness" and "inattentional blindness" illustrates something that stage magicians have known for a long time: if someone doesn't know what to look for, then they can be staring right at it and yet not see it. For example, many people take quite a while to work out that the sign below does not say 'The illusion of "seeing" ' (from O'Regan & Noë, 2001).



What people notice depends crucially on what they expect to see, and what they unconsciously consider salient. Although we experience the visual world as continuously present, in fact we only really notice small parts of what we can see. We do not in fact build up an internal representation of the external visual scene. If we did then we would not suffer from "change blindness". Low-level visual processing of some cues, for example a movement seen out of the corner of our eye, may alert us to a change and grab our attention. But if this low-level process is not triggered (for example because of a change while a scene is briefly obscured, during a saccade, during a blink or because there is too much overall movement in the scene) then the task of spotting a change is very hard indeed.

Following on from several laboratory experiments on change blindness during the 1990s, Simons & Levin (1998) investigated whether this was an effect which occurred only with the kind of 2D images used under laboratory conditions, or whether this effect would also occur in real-world interactions. The experimenters were a pair of young men in their 20s. One of them, carrying a campus map, approached an unsuspecting pedestrian and asked for directions. After a few seconds conversation, they were rudely interrupted by two people carrying a door, who barged between them on the path. As the door passed, the first experimenter grabbed the back of the door and the second experimenter took his place, opening out an identical copy of the map and continuing the conversation.

Photo from Simons & Levin (1998)

Half of the subjects failed to notice the substitution. Interestingly, these people were all older (35-65 years old) than the experimenters. All the subjects in the same age range of as the experimenters noticed the substitution. Simons & Levin conjectured that this was because the older people considered the experimenters to belong to an “out-group” and having labelled them as belonging to a particular category, then failed to notice a substitution by another person from that same category. To test this hypothesis they then re-ran the experiment, but this time with the experimenters dressed as construction workers (there was at the time a construction site within 50m), and approached only younger pedestrians, who in the previous experiment all detected the substitution.

Photo from Simons & Levin (1998)

This time only one third of the subjects detected the substitution. As one subject confirmed:

> “she quickly categorized the experimenter as a construction worker and did not retain those features that would allow individuation. Even though the experimenter was the center of attention, she did not code the visual details and compare them across views. Instead, she formed a representation of the category, trading the visual details of the scene for a more abstract understanding of its gist or meaning.” (Simons & Levin, 1998)

We observe the related phenomenon of “inattentional blindness” when we are engaged in an activity which requires us to concentrate on certain categories in a scene, then completely fail to notice other things in the scene, even when they are surprising and would normally just “pop-out” into our awareness. For example, Simons & Chabris (1999) describe an experiment where subjects were shown a video in which two teams of three players passed basketballs to each other as they moved about in a lobby. The subjects were asked to keep a silent mental count of passes between team members on either the black or the white team. After watching the video (a little over a minute long), the subjects were asked a series of questions attempting to prompt them to recall the unexpected event which happened part way though: in one version of the video a woman with an umbrella walks across the scene, in the other, a woman in a gorilla-suit. About half the subjects failed to notice. In a further test, another video was used in which the “gorilla” was visible for 9 seconds out of 65 seconds, turned to face the camera and thumped its chest. Half the subjects still failed to notice it.

Photo from Simons & Chabris (1999)

Related work on visual working memory (Olsson & Poom 2005) further highlights the importance of mental categories and expectation on visual perception. Olsoon & Poom asked subjects to state whether the 2D design they were shown was one of a group they had been shown just before. Previous experiments had concluded that visual working memory can store up to four objects. However, when the designs in this experiment were chosen so they were distinctly different but did not belong to different categories, the subjects found the task surprisingly difficult. Their conclusions shed further light on the unnoticed “construction-worker” substitution described previously:

> “our experience of using visual memory in natural settings outside the laboratory typically involves objects that belong to separate categories. Categorical structures kept in long-term memory may be required for the retention of up to four categorically distinct objects in visual working memory, but when the objects belong to the same category, only one object can be retained.” (Olsson & Poom 2005)

All of these experiments may seem of limited relevance to computer programming. But

consider this: a large part of a developer's time is spent reading code, perhaps half of the time according to Lefkowitz (2005). When reading code, the programmer is sometimes looking for known things: where an assignment happens, how a variable is used, and so on. At other times, the programmer is not looking for any one particular thing, but is searching for something more abstract: seeking an explanation for a program's erroneous behaviour, or forming an understanding of how a program will behave when it runs. As shown by the experiments described above, our experience and expectations can sharply limit our ability to properly notice what we are looking at. The same mental processes which cause change blindness and inattentional blindness ensure that we can be looking right at the line of the program which would give us the answer to our current problem, and yet fail to notice that it is salient.

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, and 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.

So a pair working together will make quicker progress because they notice more facets of the program that they are developing. However, as they work together, they will learn from each other and their points of view will converge. As their unconscious categories become more aligned, the things they notice and the things they fail to notice will become more and more similar. Eventually, the benefit that they gain from two pairs of eyes will become negligible.

This prediction is exactly in line with observation: over the course of a few hours, an initially very productive pairing becomes less and less effective. Consequently it is standard practice to rotate pairs every few hours, so that although one member of a pair may continue to work on one piece of code for several days, they are paired with a succession of people who are "fresh" to them. (Being "fresh" to the code itself does not seem to be necessary.) After a longer period working with other people, the freshness seems to be restored and they can, for a short while, work productively as a pair again.

Implications for software development

Programmers who claim to be pair programming but do not swap pairs will be much less

effective. If pair fatigue is caused by convergence of categories, the only solution is to keep swapping pairs and preserve diversity of views. Pairings should preferably only last a few hours, for example a morning or afternoon. It is tempting to continue longer than this, because the programmers feel that they have a lot of program knowledge in their heads and will make good progress. Unfortunately they are wrong. Because of pair fatigue, performance drops off rapidly.

Because of the lack of variety, it is hard to pair program effectively in very a small group: there are not enough alternative pairs to completely avoid pair fatigue. In my experience a group of 6 people is workable, and a group of 4 much less effective. Initial attempts at pair programming in an organisation may involve only a handful of people. For the above reasons this will lead to less variety and therefore more pair fatigue. The observed benefits of pair programming may therefore be less than anticipated in a small initial team or where particular pairs insist on often working together.

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. Pairings between people of similar ability are therefore advisable to obtain maximum benefit from mechanism number two. However, the fact that categories converge during pair programming is proof that some learning is taking place, and as we will see in mechanism number four, this has a longer term benefit. There is a place for expert-novice pairings.

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

Why don't programmers do what they believe to be best practice? Of course, programmers are not unique in this respect. We don't always do in life what we know is best for us: we eat more than we intend, don't go to the gym, don't give up smoking … the list goes on. In the practice of software development, we suffer from a very large problem which is not technical: we have more than enough plausible systematic working practices or "methodologies". If we actually used them more, they might make a substantial difference to the quality of our software. Why do we persist in poor practices when we know they are poor?

The worst style of programming, known as "code-and-fix" is still distressingly popular. Code-and-fix is the style usually practiced by beginning programmers, before they know any better. Programmers try to write a piece of code which they hope will do some particular thing, and then they run it to see what happens. When it appears to work, they go on to add further code (without really trying to systematically test their program and find errors). When their program goes wrong, they change the code in a haphazard way, re-running it until it appears to be working. Certainly, most programming teams would claim to be using some respectable methodology, but if you look at what the individual programmers actually do minute-by-minute, you will find people doing code-and-fix

more often than they would like to admit. Why?

We can view such behaviour from two perspectives: I will give a traditional behavioural explanation and also a more modern explanation based on the neuroscience of addiction. However, both explanations predict the same thing, which is reassuring. Let us take a look at the behavioural explanation first.

Everyone is familiar with Pavlov's experiments on dogs, where the animals were trained to respond in the same way to one external stimulus (hearing a bell ringing) as they did to an another external stimulus (tasting meat). In this example of "classical conditioning" , the dogs learned to salivate when the bell was rung. They did not learn to salivate more in general, but merely in response to the bell being rung.

The other form of learning explored by behaviourists, called "operatant conditioning", involves learning to perform some action spontaneously, without any external stimulus. 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 spontaneously, and with operant conditioning we supply the animal with a reward after we observe it performing the one we want. (We are said to be "reinforcing" the behaviour.) Perhaps the behaviour we want is very unlikely, but we can still reinforce a behaviour which tends towards what we want, or which is incompatible with behvious we don't want. For example, if you don't want your falcon to land on your head, teach it to land on your hand. As this process of reinforcement continues, the desired behaviour becomes more and more likely to be exhibited 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, this diminution in response happens quite slowly, and the strength of the conditioned behaviour is easily restored by a further reinforcement before it has completely ceased. In fact learning of the behaviour happens quickest and most strongly if the pattern of rewards is somewhat unpredictable, with a so-called "variable ratio" or VR schedule of reinforcement. As explained by Gleitman (2004):

> "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."

So, operant conditioning works just as well on humans as on animals. It is unconscious: we usually do 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! It should by now be clear that we can fall into particular patterns of software development because our interaction with the computer reinforces particular behaviours. With code-and-fix, where we tinker haphazardly with a program, we are effectively putting a coin into the slot-machine every time we alter our code and run it to see what happens. Sometimes we are rewarded by the appearance of it working. Often, our program fails and we are not rewarded. Perhaps we experience a long series of failures. However, the code-and-fix behaviour is very slow to extinguish because it has been very thoroughly learned. Just as with the slot-machines in the casino, this is a VR schedule, which reinforces the behaviour most effectively. And yet as programmers we can remain blissfully ignorant of the roots of our behaviour. In a very real sense, we can become addicted to code-and-fix.

It pays to understand why other methods of development are reinforced more weakly: code-and-fix produces an unpredictable sequence of rewards, because the programmer makes undirected, haphazard changes to the code. In a real sense it *is* random whether the program will work better or worse afterwards. This unpredictable sequence of rewards form a VR schedule for code-and-fix. If a programmer chose another form of development then the pattern of rewards would be much more predictable. They would of course get rewards, so the behaviour would be reinforced, but more weakly than with a VR schedule. Code-and-fix behaviour is therefore is the most likely to occur because it is the most effectively reinforced. Of course we can consciously decide to behave differently, and perhaps we will succeed. We can perhaps, along with other addicts, “just say no”, but along with them it will be hard to keep our resolve.

Behavioural explanations only really help us to understand *what* happens in situations like this. More modern work on learning and addiction starts to explain *why* it happens. Recent work by Pessiglione et al (2006) confirms that the neurotransmitter dopamine has a key function in operant conditioning. Animals unconsciously make predictions all the time about the rewards expected from possible actions. So as to minimise the errors in these predictions, when an animal gets an unexpected reward, a surge of dopamine is released which has the effect of making that action more likely in future. Pessiglinone et al conducted an experiment where the human subjects played a gambling game after being given either a chemical which increased their dopamine, a placebo, or one which reduced their dopamine. Unlike gambling in a casino, the symbols which the subjects chose between had consistent payouts or penalties, but the subjects were not told what these were. Those with more dopamine learned more quickly to unconsciously pick the better choices, presumably because the higher levels of dopamine led to more effective reinforcement of the rewards when they picked winning choices. (It did not appear to have any effect in those instances that they picked losing symbols: learning from losing is controlled by other chemical pathways in the brain, mostly dependent on serotonin.)

Although some addictive drugs directly cause dopamine to be released and hence become addictive because they “feel good” in themselves, it is becoming recognised that many other activities which cause a pleasurable release of dopamine can be just as physically addictive. People can be addicted to gambling, shopping or exercise due to exactly the same brain chemistry as drug addicts. Phillips (2006) describes research in which the

subjects' physiological and EEG response were measured while they were shown a variety of images related to particular addictions. Subjects with drug addictions had similar responses to those addicted to gambling or computer gaming when shown images related to their own addiction. However, people exposed to the same surroundings who remained unaddicted, for example casino staff, did not have these responses when shown images of these surroundings.

Returning finally to the idea that pair programmers are less susceptible to poor practices, it should now be clear that most programmers are in a state similar to recovering gambling addicts. They first learned to program in a code-and-fix style, and although they generally accept that this isn't the most effective way to program, they will always be hooked. Programming in another way will always be a conscious effort, a deliberate avoidance of the behaviour to which they are addicted. Programming is probably worse in this respect than all other engineering activities: because of the absorbingly interactive nature of software development, the raw materials with which we work really do bite back. How can we escape this addiction?

One possible way is to slow the process of development down and make it less interactive. This is perhaps the hope of heavy-weight development processes. For example, the Personal Software Process emphasises book-work and hand-checking of a program away from the computer. Theoreticians have long advocated proving programs correct before being allowed near a real computer. These ideas certainly would counter the addictive influences but removing interactivity and making reward much less frequent. However, it seems foolish to turn our backs on the thousand-fold increase in computer power available to us since the 1980s, 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. While not quite as addictive as code-and-fix, because it is more predictable, test-first-development does produce a steady stream of rewards from successful tests.

Setting two addicts to a task seems to be the key here. If alcoholics had to work behind a bar serving drinks, they would probably be a lot happier to work as a pair, keeping each other on the straight and narrow. It is routine practice in security engineering to use "two-man working" for roles in which people could be tempted or coerced into doing something wrong. 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.

Implications for software development

We need to set up social arrangements which will support us in our attempts to practice those methods of development which we agree are best. Pair programming is one such social arrangement. There must be surely others, for example 12-step programmes for recovery from drug and gambling addiction. However, the prevalence of two-man

working in those jobs where human fallibility is a serious problem should lead us to consider that working in pairs might be the simplest effective solution.

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

How do programmers rank their own expertise and that of their peers, and how do programmers learn to be more expert? Pair programmers are better at judging the expertise of their peers and better at learning from their peers.

It has been acknowledged for decades that there are huge productivity differences between programmers, even between professional programmers in the same development team. The conventional wisdom is that within a development team the range in productivity is often a factor of ten or more (Glass 2003). Even higher ranges are sometimes mentioned. For example, Alan Eustace (a vice-president of Google) claims that the best engineers are worth “300 times or more than the average” (Economist 2006a).

I believe that this wide range of skill is a real phenomenon, though as will become apparent, perhaps overstated. Maybe 300 times is an exaggeration, but ten times is certainly credible. For programmers and managers this raises the fundamental problem of how you recognise an expert. For programmers alone it also raises the further problem of how to become an expert. These are straightforward questions, but as we will see shortly, they are very hard to answer. Failure to answer them may be at the root of many disastrous software failures. Let us first take the question of how to recognise an expert.

In some fields it is straightforward to recognise experts, for example, when individuals compete against each other in games and sports. Expert chess players compete against each other in timed matches, again and again. They are given a numerical ranking which is a very reliable measure of their skill and their likely success against other players in the future. Similarly, players in sports where individuals compete against each other either directly or indirectly can be ranked reliably. We can with some certainty say that one golfer is better than another, and even put a meaningful figure on how much better they are. But this is not true of all sports: for example, players on team games like basketball are much harder to rank. In any team activity, it will be very hard to give the team members accurate rankings, because there are so many confounding factors, and this is true of other team activities. For example, Gladwell (2006) notes:

> “Suppose we wanted to measure something in the real world, like the relative skill of New York City’s heart surgeons. One obvious way would be to compare the mortality rate of the patients on whom they operate --- except that substandard care isn’t necessarily fatal, so a more accurate measure might be how quickly patients get better or how few complications they have after surgery. But recovery time is a function of how a patient is treated in the intensive-care unit, which reflects the capabilities not just of the doctor but also of the nurses in the I.C.U.

> So now we have to adjust for nurse quality in our assessment of surgeon quality. We'd also better adjust for how sick the patients were in the first place, and since well-regarded surgeons often treat the most difficult cases, the best surgeons might well have the poorest patient recovery rates. In order to measure something you thought was fairly straightforward, you have to take into account a series of things that aren't so straightforward." (Gladwell, 2006)

Assessing programming talent is surely no easier than assessing the skill of a heart surgeon, or of players in team sports like basketball. In these cases there are so many variables that "people construct their own arbitrary algorithms --- they assume that every factor is of equal importance, or randomly elevate one or two factors for the sake of simplifying matters" (Gladwell, 2006). For example, basketball pundits seem to use only two factors when ranking players, disregarding many other factors which really do contribute to a team's success or failure. (In software development, "lines of code written per day" often gets elevated above all others, simply because it is easy to measure.) By selecting a few arbitrary factors above all others, this promotes "star players" who demonstrate those particular qualities, but who in fact are mainly good-lookers and not genuinely great performers. Their role in the team's success is often found to be overstated when compared with a more scientific appraisal. As Gladwell speculates:

> "It's hard not to wonder … about the other instances in which we defer to the evaluations of experts. Boards of directors vote to pay C.E.O.s tens of millions of dollars, ostensibly because they believe --- on the basis of what they have learned over the years by watching other C.E.Os --- that they are worth it. We see [star basketball player] Allen Iverson over and over again, charge toward the basket, twisting and turning and writhing through a thicket of arms and legs of much taller and heavier men --- and all we learn is how to appreciate twisting and turning and writhing. We become dance critics, blind to Iverson's dismal shooting percentage and his excessive turnovers, blind to the reality that the Philadelphia 76ers would be better off without him." (Gladwell, 2006)

Rating the ability of programmers is surely an even worse task than rating basketball players. It's never the same game twice and the rules are usually changed half way through. Lines of code written per day is a very poor measure of productivity. Were those lines properly tested? When they need to be changed after a month or a year, how easy will it be for another programmer to understand them? Are those lines even needed --- some of the most productive programming sessions result in fewer and simpler lines of code. How do you measure that?

Perhaps if we observed a programming team minutely and accumulated copious statistics about every aspect of their work and its consequent results, we could work out a more scientific rating of each individual's performance. Perhaps if there were as few professional programming teams as professional basketball teams, and their members were paid as much, then it might be worth someone's while to do this. But we don't live in that world. So what can we do?

Graham (2004) 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:

> "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." (Graham, 2004)

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. (Monogan & Suojanen (2000) has examples of the kinds of problems used in such interviews.) 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 "the same knowledge that underlies the ability to produce correct judgements is also the knowledge that underlies the ability to recognise correct judgement" (Kruger & Dunning, 1999). In a field where expertise is hard to measure this is a serious problem, because as Kruger & 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. This is the only reliable way to rank the expertise of other programmers, and indirectly the only way to rank yourself:

> "But hackers can't watch themselves at work. So if you ask a great hacker how good he is, he's almost certain to reply, I don't know. He's not being modest. He really doesn't know. And none of us know, except about people we have actually worked with." (Graham, 2004)

And now we can return to pair programming. Pair programming is close to an ideal environment for a team to become clearly aware of their own abilities and ranking.

Programmers work closely on problems day after day. They swap pairs frequently, so they can easily compare the strengths and weaknesses of others. They will come to understand their own rank in the team. They will know who is generally more capable and who is weaker, and they will know who is more expert on particular specialist areas. No one can easily pretend expertise when they don't have it. Even if they mistakenly think they have expertise, they are likely to be called out by their peers, who after all will be working on the problem with them.

We should expect that estimates of time and difficulty produced by a pair programming team will be more accurate than a team which uses solo programming. (And this does appear to be the case.) But it's not just a matter of more accurate time estimates: success or failure of a whole project can be in the balance. In a solo programming environment, because it is so hard to correctly rank ability, the self-assured good-looker will sometimes be given more than their fair share of credit for a team's success. They probably believe it themselves. More seriously, they may get less than their fair share of blame for a team's failure, particularly in their own minds. As Kruger & Dunning note:

> "The problem with failure is that it is subject to more attributional ambiguity than success. For success to occur, many things must go right: The person must be skilled, apply effort and perhaps be a bit lucky. For failure to occur, the lack of any one of these components is sufficient. Because of this, even if people receive feedback that points to a lack of skill, they may attribute it to another factor." (Kruger & Dunning, 1999)

So the less-than-expert programmer goes on to another project, perhaps with more responsibility, and the story repeats until eventually in a "blue-sky disaster" their luck runs out. (And takes a whole project with it.) No one really knew how bad they were, because no one ever paired with them. And even afterwards, no one really knows why the project failed.

It is said that pair programming is "not for everyone", perhaps because in pair programming "there is nowhere to hide". People will see you for what you are. However, what you currently are is not what you will always be. A few people will be unhappy to work alongside more expert programmers, and some will even be unwilling to acknowledge their own lack of competence. But for most programmers, a pair programming team can be a wonderful place to learn new skills. People like to learn by working together:

> "most employees value informal training more than formal teaching: in a survey by Delloitte, 67% of respondents said that they learn most when they are working with a colleague, with only 22% saying that they do best when they are conducting their own research, and only 2% happiest with a manual or a textbook." (Economist 2006b)

Programming is still a craft skill, and as such it is best learned by hands-on experience at the side of an expert. Where a pair with different levels of skill work together, they will

have the same sort of relationship as has always existed between a master and an apprentice working side-by-side. This is the traditional, and best, way to learn a craft skill. Where two people of similar level of skill work together as a pair, there will still be some transfer of knowledge between them, since their expertise will not be identical in all areas. In particular, they will tend to transfer recently learned knowledge about other parts of a program that they have worked on separately.

Implications for software development

If you are a manager and not a programmer, you have to accept that you will not be able to tell who is good and who is bad on your team, except for what your team choose to tell you. Any metric like "lines of code written" is liable to be deceptive and prone to distort the development process. The best choice if you are a manager *and* a programmer is to occasionally pair with all the people on your team. Then you'll know directly for yourself.

Pairs must swap around frequently, not only to avoid pair fatigue, but so that the team can get the measure of each other and help each other to learn. It is not fair, or prudent, to have the least skilled pair always working together, or the most skilled pair.

Explaining hostility to pair programming

You might think that programmers would be eager to embrace a new practice that made their work more productive. How then can we explain the hostility to pair programming which some people show, usually without having tried it or even seen it in action? I have an explanation for this hostility which also has the charming advantage of explaining why I was so enthusiastic about pair programming that I wanted to write this paper.

My explanation for this hostility is based on two observations: firstly an observation that some programmers are much more comfortable with machines than with other people, secondly an observation about how people behave when they are presented with uncomfortable evidence.

First, let us look at why it is that some programmers enjoy working with machines more than people. The nerdy anti-social programmer who forgets to wash and goes train-spotting on the weekend is a stereotype, but there is a grain of truth in it. Such people do exist, we have all worked with them. They represent one extreme in a broad distribution, from the apparently ordinary to those who would definitely be diagnosed as autistic.

Baron-Cohen (2004) conjectured that autistic tendencies could be split into two aspects: how easily an individual understands systems of objects, and how easily they understand the emotions of people. To test this conjecture, two questionnaires for "Systemizing Quotient" (SQ) and "Empathy Quotient" (EQ) were devised. In experiments, there was a strong correlation between the SQ/EQ scores of the subjects and independent diagnoses of Asperger syndrome and other autistic spectrum tendencies (Baron-Cohen et al, 2003).

These experiments also indicated that Asperger syndrome people have a kind of “extreme male brain”. Although there is a large degree of overlap, on average women have an EQ higher than their SQ, on average men have a EQ lower than their SQ and autistic spectrum people (largely men) tend to have an SQ much higher than their EQ.

Obviously, people who find it easier to deal with systems of objects will tend to be drawn to jobs where this is what they do. Baron-Cohen (2004) remarks that amongst the “normal” subjects of his experiments, engineers, scientists and mathematicians tended to have a higher SQ score. It is nowadays accepted that autism has a large heritable component (Silberman, 2001), and this has been offered as an explanation for why Asperger’s syndrome has risen dramatically in some places: the suggestion is that where there is a concentration of technical industry, slightly autistic engineers may meet at work, marry and have slightly autistic children (BBC, 2006).

So, it is only natural to expect that some programmers will be in their jobs because they have a very high SQ and very low EQ, and they find interacting with computers very much less demanding than interacting with people. Then along we come with pair programming and explain that here is a great new way of working which means that they now have to interact with people much more. Obviously they are not going to be enthusiastic. It will seem like very hard work, compared to coding on their own. (Note that this discomfort is unrelated to the effectiveness of pair programming, if they were actually to do it. They may or may not find that it helps them to program better. The discomfort happens even before they try it.)

So that explains why some programmers find the idea of pair programming uncomfortable, and are unwilling to try it. But why would they be hostile to the extent of belittling other people’s efforts to use pair programming? We can explain this curious phenomenon by using Festinger’s theory of cognitive dissonance (Festinger, 1957). (More recent work by evolutionary psychologists on self-deception predicts essentially the same effects.)

When there are inconsistencies between our beliefs, our behaviour and our environment this causes a discomfort, called “cognitive dissonance” and hence a pressure to change something. It might seem that the easiest thing to change would be our beliefs or our behaviour, since the world is “trying to tell us something”. However, people often seek an easier strategy. They can attempt to change their environment, either weakly by merely avoiding distressing information and seeking reassuring information, or strongly by surrounding themselves with other people who share their own erroneous beliefs.

Following the weak strategy, experiments show that we tend to weigh evidence evenly before making a decision, but behave differently afterwards since evidence contrary to our decision causes dissonance. (This explains the purpose of car adverts on the TV: they are not designed to make you buy that brand of car, but to reassure you if you did and to make you uncomfortable if you bought a different brand. This is rather pointless, but all the manufacturers have to play the game, or else their own customers would feel more uncomfortable than the rest and they might buy a different brand next time.) Festinger’s

experiments showed that people tend to avoid noticing the contrary evidence, while focusing on evidence that supports their decision. The more important the decision, the higher the dissonance, so the greater the inclination to ignore bad news.

Festinger (1957) gives a wonderful example of the strong strategy, where people attempt to reduce dissonance by recruiting others into their beliefs. In this example, Mrs Keech, the leader of a UFO cult who thought she was receiving messages from another planet said that the members of the cult would be taken away into the heavens by a flying saucer on a particular date.

> "Festinger's prediction, assuming the momentous event did not occur, was that the followers would attempt to reduce their dissonant state at having their beliefs disconfirmed by attempting to convince others of their beliefs. There is now a vast set of experimental data to support that view, but then it was brand new." (Gazzaniga, 1998)

Of course the flying saucers did not come, but a few hours later a new "message" arrived, telling them to spread the word, an injunction they embraced enthusiastically, despite having previously shunned the media.

> "Mrs. Keech reached for the phone to call the press. She had never done this before, but now she felt that she must, and soon all the members of the group had called various branches of the news media." (Gazzaniga, 1998)

The same phenomenon has happened many times in the history of cults and religions: at the point that you might think they should give up due to contrary evidence, they often gain a further impetus to evangelise. By surrounding themselves with more believers they change their personal environment and reduce the dissonance caused by bad news.

At this point it is fairly easy to see why the programmer who is uncomfortable with the idea of pair programming might feel the need to convince others that it is a bad idea. It is comforting to surround yourself with others who share your beliefs, and an effective way to achieve this is to evangelise.

Of course the cognitive dissonance theory also explains why I am writing this paper. (A good theory should explain both sides of an argument.) I believe that pair programming is effective and worthwhile. I find it uncomfortable for people to express contrary views, so I evangelise about pair programming. I try to encourage other people to pair program; I make sure that my students learning to program do so in pairs; and now I am writing this essay, all to increase the number of people who agree with me and so reduce my own dissonance.

Conclusions

By examining more closely the four mechanisms which underpin pair programming, we

can see that it is not a single take-it-or-leave-it practice with a single benefit. Even within the same team, the mechanisms 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 will be vary in their applicability and helpfulness. If you never swap pairs, you will gain little benefit from mechanism two, and after a while you will stop benefiting from mechanism four. If you don't agree on a development methodology, you can gain little from mechanism three. However, even if you only benefit from one mechanism, that is still a benefit. Recognising the other mechanisms brings the potential for further benefits by changing how you work together. You could do that today.

References

Baron-Cohen, S. et al (2003). *The systemizing quotient: an investigation of adults with Asperger syndrome or high-functioning autism, and normal sex differences*. S. Baron-Cohen, J. Richeler, D. Bisarya, N. Gurunathan & S. Wheelwright. Phil. Trans. R. Soc. Lond. B. (DOI 10.1098/rstb.2002.1206) Royal Society: London. (2003).

Baron-Cohen, S. (2004). *The Essential Difference: Men, Women and the Extreme Male Brain.* Penguin: London. 2004.

BBC (2006). *Scientific brain linked to autism*. BBC, London. Available online at: http://news.bbc.co.uk/1/hi/health/4661402.stm Accessed 31 January 2006.

Beck, K. (2005). *Extreme Programming Explained. 2nd Ed*. Kent Beck with Cynthia Andres. Addison-Wesley: Boston, MA. 2005

Carruthers, P. (2002). *The cognitive functions of language*. Peter Caruthers. Behavioural and Brain Sciences, 25, pXXX (2002).

Cheng, K. (1986). *A purely geometric module in the rat's spatial representation*. K. Cheng. Cognition, 23:149-178 (2002).

Chong et al (2005). *Pair programming: When and Why it Works.* Jan Chong, Pobert Plummer, Larry Leifer, Scott R. Klemmer, Ozgur Eris & George Toye. Proc PPIG17, June 2005.

Chong, J. (2006). *Pair Progamming Re-Design.* 20 March 2006. Available online at http://www.researchchannel.org/prog/displayevent.aspx?rID=4813&fID=345 Accessed 4 December 2006.

Csikszentmihalyi, M. (1990). *The Psychology of Optimal Experience.* Csikszentmihalyi, Mihaly. Harper and Row: New York.

DeMarco, T. & Lister, T. (1999). *Peopleware. 2nd Ed.* Tom DeMarco & Timothy Lister.

Dorset House: New York. 1999.

Economist (2006a). *The revenge of the bell curve.* Economist, 7 October 2006.

Economist (2006b). *Masters of the universe.* Economist, 7 October 2006.

Festinger, L. (1957). *A theory of cognitive dissonance*. Row, Peterson: Evanston IL. 1957

Gazzanagia, M. (1998). *The Mind's Past.* Michael S. Gazzaniga. University of California Press. 1998.

Gizerenzer, G. & Todd, P. (2000). *Simple heuristics that make us smart.* Gerd Gizerenzer, Peter Todd & ABC Research Group. Oxford University Press. 2000.

Gladwell, M. (2006). *Game Theory*. Malcolm Gladwell, New Yorker Magazine, 29 May 2006. Available online at http://www.gladwell.com/2006/2006_05_29_a_game.html Accessed 4 December 2006.

Glass, R. (2003). *Facts and Fallacies of Software Engineering.* Robert L. Glass. Addison-Wesley: Boston, MA. 2003.

Gleitan, H. et al (2004). *Psychology. 6th Ed*. Henry Gleitman, Alan J. Fridlund & Daniel Reisberg. W.W.Norton & Co: New York. 2004.

Graham, P. (2004). *Great Hackers.* Paul Graham. July 2004. [online] Available online at http://store.yahoo.com/paulgraham/gh.html Accessed 4 December 2006.

Hermer L. & Spelke, E. (1994) *A geometric process for spatial reorientation in young children*. Nature, 370:57-56 (1994).

Hermer L. & Spelke, E. (1996) *Modularity and development: the case of spatial reorientation*. Cognition 61:195-232 (1996).

Hermer-Vazquez at al (1999). *Sources of Flexibility in Human Cognition: Dual-Task Studies of Space and Language*. Linda Hermer-Vazquez, Elizabeth S. Spelke & All S. Katsnelson. Cognitive Psychology, 39, 3-36 (1999).

Kruger, J., & Dunning, D. (1999). *Unskilled and Unaware of It: How Difficulties in Recognising One's Own Incompetence Lead to Inflated Self-Assessments.* Justin Kruger & David Dunning, Journal of Personality and Social Psychology, Vol 77 No 6 (1999).

Lefkowitz, R. (2005). *The Semasiology of Open Source (Part 2).* Talk at O'Reilly Open Source Convention held in Portland, Oregon August 1-5, 2005. Available online at http://www.itconversations.com/shows/detail662.html Accessed 15 Nov 2006.

Monogan & Suojanen (2000). *Programming interviews exposed: secrets to landing your*

next job. John Mongan & Noah Suojanen. Wiley: New York. 2000.

Olsson, H. & Poom. L. (2005). *Visual memory needs categories.* Henrick Olsson & Leo Poom. Proceedings of the National Academy of Sciences. 102(24) 8776-8780 (2005).

O'Regan, J. & Noë, A. (2001). *A sensorimotor account of vision and visual consciousness.* Behavioural and Brain Sciences 24, 939-1031 (2001).

Pessiglione, M. et al (2006). *Dopamine-dependent prediction errors underpin reward-seeking behaviour in humans.* Mathias Pessiglione, Ben Seymour, Guillaume Flandin, Raymond J. Dolan & Chris D. Frith. Nature 442, 1042-1045 (31 Aug 2006).

Phillips, H. (2006). *Just can't get enough.* Helen Phillips. New Scientist. 26 August 2006.

Pinker, S. (1994). *The language instinct.* Stephen Pinker. Penguin: London. 1994.

Silberman, S. (2001). *The Geek Syndrome*. S. Silberman. Wired, Vol. 9 No 12, December 2001.

Simons, D. & Chabris, C. (1999). *Gorillas in our midst: sustained inattentional blindness for dynamic events.* Daniel J. Simons & Christopher F. Chabris. Perception 28, 1059-1074 (1999)

Simons, D & Levin, D. (1999). *Failure to detect changes to people during a real-world interaction.* Daniel J. Simons & Daniel T. Levin. Pyschonomic Bulletin & Review 5(4), 644-649 (1999).

Sturdy, J., 2005. E-mail from John Sturdy, 2005.

Willliams & Kessler (2003). *Pair Programming Illuminated.* L. Williams & R. Kessler. Addison Wesley: Boston, MA. 2003.