The Collaborative Nature of Pair Programming

Sallyann Bryant, Pablo Romero, Benedict du Boulay

IDEAS Laboratory, University of Sussex, Falmer, UK
s.Bryant@sussex.ac.uk, pablor@Sussex.ac.uk, b.du-boulay@sussex.ac.uk

Abstract. This paper considers the nature of pair programming. It focuses on using pair programmers' verbalizations as an indicator of collaboration. A review of the literature considers the benefits and costs of co-operative and collaborative verbalization. We then report on a set of four one-week studies of commercial pair programmers. From recordings of their conversations we analyze which generic sub-tasks were discussed and use the contribution of new information as a means of discerning the extent to which each pair collaborated. We also consider whether a particular role is more likely to contribute to a particular sub-task. We conclude that pair programming is highly collaborative in nature, however the level of collaboration varies according to task. We also find that tasks do not seem aligned to particular roles, rather the driver tends to contribute slightly more across almost all tasks.

1. Introduction

Computer programming is known to be a complex skill that is difficult to master. Recently pair programming, formalized as one of the core practices in eXtreme Programming (XP), has been shown to assist in the production of high-quality software (e.g. [1], [2], [3]. [4], [5], [6]). Here we consider co-located pair programming, as 'two people working at one machine, with one keyboard and one mouse' [28] and use the standard terms 'driver' and 'navigator' to indicate who has control of the keyboard (the 'driver'). These existing studies indicate an improved outcome through pair programming (e.g. better quality software, faster production speed, fewer defects and greater enjoyment) and high level reports (e.g. [7]) and ethnographic studies (e.g. [8], [9]) provide useful insights into pair programming in practice. However few, if any, studies have considered in detail the process by which these improved outcomes are achieved. It has been suggested that they may be due to 'pair pressure' [7], where a programmer is more focused and thorough when being watched. Other studies have suggested pairing may be beneficial due to greater enjoyment [4], increased overhearing [8], provision of a better apprenticeship environment [29] and increased knowledge distribution. Pair programming may simply be a way of improving outcome by encouraging programmers to talk to themselves, a phenomena known in other subject areas as self-explanation (e.g. [10]). Here we consider the level of collaboration in pair programming across different types of tasks via a series of on-site studies of experience professional pair programmers 'in the wild' [11]. Via these four, one-week observational studies we gathered, transcribed and analyzed 36 pair programmers' conversations. Here we consider sessions where both programmers have at least six months' commercial pair programming experience, in an attempt to address the following questions:

- Do pair programmers talk to themselves while working on separate sub-tasks?
- To what extent do pair programmers actually 'collaborate' on the same task?
- Are certain types of task more collaborative than others?
- Does a particular role (driver/navigator) contribute more strongly to a particular type of task?

Section 2 provides an overview of perspectives on the effects of verbalization to oneself and others and section 3 considers how to characterize collaboration. We then go on to explain the methodology and background of our studies and in section 4 present the results of an in-depth analysis of 23 hours of pair programmers' dialogue. We conclude by considering what these results tell us about the collaborative nature of pair programming, and discussing further work which we now hope to undertake.

2. Verbalisation

Gathering and analyzing verbalizations from pair programmers seems ideal because, unlike other domains, the pair are already communicating verbally and so do not need to be asked to do so. Hopefully this minimizes the impact of the observation. Here we take verbalisation to mean any talk produced, whether directed at themselves or each other. While extra-pair communication (for example, discussion with a third party) may be an interesting area of study, it has been excluded from this analysis.

Before we can begin to address the questions we have identified, it is necessary to consider how to characterize collaboration. It has been suggested [19] that it is hard to describe the differences between explaining to oneself and explaining interactively, but that collaborative situations may be defined in terms of three factors: interactivity, asynchronicity and negotiability. Similarly it is suggested [20] that co-operative work is accomplished by the division of labour. Here, we will consider a collaborative task one to which both parties are contributing information and a co-operative task one where only one programmer contributes.

2.1 Collaboration and Verbalisation

Here we take collaboration to mean both parties contributing new information to a given task. Collaboration is widely documented as being beneficial: Suthers [17] suggests that collaboration increases learning, productivity, time focused on the task, knowledge transfer and motivation and Jeong and Chi [18] show that understanding improves after collaboration - those collaborating on a task learned more than those performing it alone. It could be suggested that collaboration decreases the probability of confirmation bias [11], where we filter information depending on what is expected and therefore are more likely to attend to items confirming our hypotheses (even if incorrect). Similarly, in pair programming literature, Williams et al. [1] suggest that collaborating lowers the likelihood of developing 'tunnel vision'.

2.2 Co-operation and Verbalisation

If pair programmers typically do not collaborate on a task, but are more likely to co-operate (that is, split the task up and work on separate subtasks) verbalisation could still affect performance. There is a body of evidence suggesting that simply talking to oneself helps improve understanding. For example, Chi et al. [10] asked a group of students to self-explain each line of a text about physics and showed that self-explanation resulted in the production of a more correct mental model and a higher gain in understanding. Ainsworth and Loizou [12] suggests that verbalization provides a form of 'computational off-load', perhaps putting part of the problem 'out in the world' rather than requiring it to be kept 'in the head'. Ericsson and Simon [13] state that verbalization provides an intermediate re-coding of information, and that in the process of this recoding, it is necessary to add further information for communication purposes which may itself prove useful. Cox [14] also shows that translation between modalities (in his work from mental to diagrammatical) improves understanding. This might all be easily extrapolated to the domain of computing and suggests that simply talking about a software development issue may assist in its understanding and ultimately its resolution. In fact there are a number of accounts of this effect including talking to a rubber duck [14] or even a poster of your favorite movie star.

Studies considering the effect of requested verbalization have also addressed this issue with somewhat different results. Such studies have questioned the use of eliciting verbal protocol (asking participants to

talk to themselves as a means of gaining insight into mental processes) and considered whether talking aloud may change the manner in which a task is performed. Of particular interest, Ericsson and Polson [15] show that talking aloud has an effect no different from counting out loud while performing a task – it slows participants down but does not affect their performance.

Another group of studies of a phenomenon known as 'verbal overshadowing' suggests that verbalization may sometimes have a negative effect. Schooler et al. [16] show that verbalization may interfere with non-verbal (insight) tasks, because they rely on non-reportable mental processing. An example of these type of insight tasks are those requiring a 'eureka' moment rather than a step-by-step process of deduction.

These three schools of thought may at first seem contradictory, however if we consider task type this suggests a more complementary picture, perhaps where explaining and embellishing help in understanding non-insight problems, 'thinking aloud' has no effect, and trying to talk about an insight problem has a negative impact. This suggests that particular types of software development task may be helped or hindered by verbalization even if just talking to oneself. There may, of course, be other explanations, including the context in which the studies took place and the means by which verbalizations were elicited.

It would appear difficult to distinguish between co-operation and collaboration in pair programming sessions, however this might be achieved by considering whether the two individuals are holding a collaborative conversation or following all the rules involved in having a conversation (turn taking etc) but actually holding two separate self-conversations, or 'interleaved monologues'. The method we have used to ascertain this is to consider not only whether each party is contributing to the conversation, rather whether these contributions are 'on task'. We have particularly looked at instances of new information being added to each task in a pair programming session. This is discussed in further detail in Section 3.

3. Study Background and Methodology.

In line with calls for studies of programmers working in an industrial setting [21], the analysis and results presented here are from four, one-week studies of commercial programmers working on on-going tasks in their usual environment. While a variety of levels of experience were studied (see [22] for insights about the differences in behavior between novice and more experienced pairers) this paper only considers programmers who had been commercially pair programming for a minimum of six months. The four studies were from three different industrial sectors and all the studies took place at medium to large scale companies. All of the projects encouraged or expected programmers to work in pairs whenever possible. Across the companies the pairs generally seemed empowered and were considered responsible for completing their tasks as they considered appropriate. The profiles of the session are shown in Table 1:

Table 1. Profile of the companies, projects and sessions studied

	Number of projects considered	**Number of pair programming sessions considered**	**Agile/XP development approach?**
Banking	1	3	Yes
Banking	4	12	Yes
Entertainment	2	10	Yes
Mobile communications	2	11	Yes

The methodology used followed the framework for verbal protocol analysis set down by Chi [24] in which protocols are produced, transcriptions are segmented and coded according to a coding schema, depicted in some manner and patterns are sought and interpreted. A literature review on the use of verbal protocols in software engineering is available [26], which also suggests that the analysis of verbalisation

may be a useful method for use in the study of pair programmers so that 'the cognitive processes underlying productivity and quality gains can be formally mapped rather than speculated about'.

Here each one-hour recording was transcribed and segmented into utterances (an utterance typically being a sentence). A coding schema was produced by reducing the work in each of the session into a tree of numbered subtasks (e.g. see Figure 1). These subtasks were derived from the dialogue by considering what was required in order to complete the task. The derived tasks were at a level of abstraction higher (i.e. less detailed) than writing a line of code but a lower level than the overall task itself. They were typically either:

- Things which needed to be done
- Things which needed to be understood
- Things which needed to be decided
- Things which needed to be 'broadcast' (outside of the pair)

Further division into sub-sub-tasks etc. was common during the process of deriving sub-tasks.

Any utterance in which new information was added was then coded with the number of the subtask the information was contributing to, the contributor (A or B) and their role at that time (navigator or driver - note it was usual for participants to change roles several times during a session). See Table 2 for an example coding (note that line 4 is not coded as it is considered a continuance of line 2).

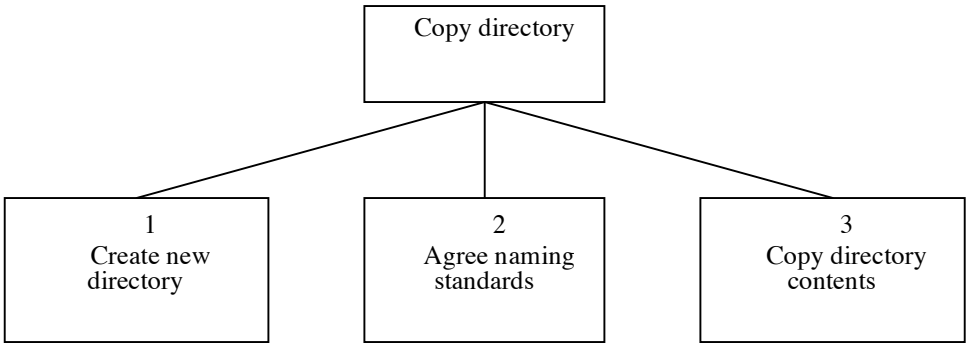

Fig. 1. Example subtask decomposition

Table 2. Example coding of dialogue

No	Participant	Role	Subtask	Generic subtask type	Utterance
1	B	Nav	1	B	So basically we can create a directory…and we can just use…
2	A	Dri	2	A	…We put the date that we are going to put the X in.
3	B	Nav	-		Right
4	A	Dri	-		So when you look at it you know that it was done on this date
5	B	Nav			Good
6	A	Dri	2	A	…Then that's a standard file
7	B	Nav	3	B	I'll just copy it all over, apart from the update.

In order to analyze the extent to which different types of subtask fostered or inhibited collaboration, the subtasks from all sessions were then used to derive a set of generic subtask types (see Table 3). The generic subtasks were then compared with those described in the literature to ensure coverage. A difference with those tasks described in [27] was the lack of a discrete 'design' category. While part of this is covered in 'agree strategy', the lack of a design category is not surprising in an XP environment, where there is no 'up-front' design task, rather design takes place as part of the coding task. The following list shows the derived generic sub-tasks used in the analysis. These cover all the tasks that were identified and therefore categories such as L (Discuss the IDE) were rarely used but are included for completeness. Instances of social chat either within or outside the pair were not considered.

Table 3. Derived generic sub-tasks

A	**Agree strategy/conventions**	Including approach to take, coding standards and naming conventions
B	**Configure environment**	Setting up paths, directories, loading software etc.
C	**Test**	Writing, running and assessing the success of tests
D	**Comment code**	Writing or modifying comments in the code
E	**Correspond with 3rd party**	Extra-pair communication: person to person, telephone or email
F	**Build, compile, check in/out**	Compiling and building on own or integration machine
G	**Comprehend**	Understanding the problem or existing code
H	**Refactor**	Re-organising the code
I	**Write new code**	Creating completely new code to complete the assigned task
J	**Debug**	Diagnosing, hypothesizing and fixing bugs
K	**Find/check example**	Looking at examples in books, existing code or on-line
L	**Discuss the IDE**	Talking about the development environment

4. Results

The pair programmers studied had all been pairing commercially for at least six months. While the introduction of pair programming was reported as having been accepted very differently (some programmers were initially very reluctant to pair, while others were keen to), all of the pairs observed behaved in a professional manner and were highly focused on the task at hand. The sessions observed showed a surprisingly high amount of verbal interaction. Pair programmers were shown to produce more than 250 verbal interactions per pair programming hour. Generally there were only very brief periods of silence. Even when a pair was awaiting a suite of tests to run, for example, they would often take the opportunity for some social chat.

The analysis performed shows that both partners contributed to more than 93% of subtasks, that is, the programming pair collaborated on 93% of the sub-tasks they performed. Similarly, when considered by role, slightly fewer, but still just more than 93% of subtasks were contributed to by the driver and by the navigator. These results suggest that pair programming sessions are highly collaborative in nature and that the programming pair really are working together on the vast majority of tasks. We will now take a closer look at the types of tasks in which more and less collaboration took place. First, in Figure 2 we consider the number of contributions made for each generic subtask type in order to ascertain which were the most common types of task for the sessions observed.

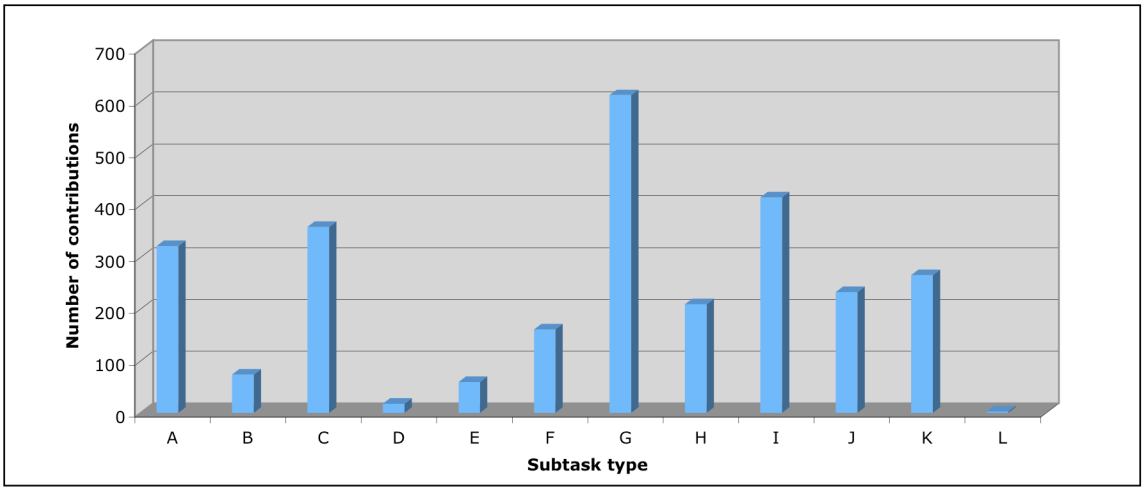

Fig. 2. Distribution of contributions amongst generic sub-tasks

It is interesting to note that the majority of contributions related to comprehension – understanding the problem or existing code. Second most common is writing new code, followed by testing (iwriting and running tests). Least common were discussing the IDE, commenting code (which is in line with the idea of self-commenting code) and corresponding outside the pair. If we normalize our data to ascertain the percentage of tasks of each type that were collaborative both across participants (i.e. both participants contributed to a task) and across role (i.e. both roles contributed) we obtain the percentages outlined in Table 4. Figures in the two columns are often, but not always the same, as a participant may contribute as both driver and navigator when roles changed mid-task.

Table 4. Percentage of tasks of each generic type that were collaborative across participants and roles

Subtask type	Percentage of tasks collaborative across participants	Percentage of tasks collaborative across roles
A - Agree strategy	91.93	91.61
B – Configure environment	81.08	81.08
C – Test	91.92	92.20
D – Comment code	83.33	83.33
E – Correspond	95	93.33
F – Build,compile,check in/out	90.68	90.68
G – Comprehension	95.11	94.94
H – Refactor	94.29	95.24
I – Write new code	94.95	94.71
J – Debug	93.56	93.56
K – Find/check example	92.48	92.48
L – Discuss the IDE	100	100

Table 4 shows that both partners contributed to almost all tasks. Only configuring the environment and commenting code had a level of collaboration below 90% and even these were over 80%, although they were rarely performed. Thus the benefits attributed to pair programming may well be due to the collaborative manner in which tasks are performed. However, in order to further understand the nature and extent of this collaboration we should consider each subtask type. In other words, since we have ascertained that both parties contribute something to almost every task, we should now consider the proportion of contributions made by each participant and each role. If we first consider the level of

collaboration between participants we find the averages shown in Table 5, along with the maximum and minimum number of contributions for each subtask type. These are then expressed as percentages of the total contributions in Figure 3:

Table 5. Most and least collaboration by participant for each generic subtask type

Subtask type	Contributions by most active participant				Contributions by least active participant			
	Average	***Highest***	***Lowest***	***Standard Deviation***	***Average***	***Highest***	***Lowest***	***Standard Deviation***
A **Agree strategy**	**3**	*13*	*0*	*2.6*	**1.4**	*8*	*0*	*1.6*
B **Configure environment**	**3**	*10*	*0*	*3.0*	**0.8**	*7*	*0*	*1.7*
C **Test**	**3.7**	*17*	*0*	*3.2*	**1.5**	*15*	*0*	*2.3*
D **Comment code**	**2.2**	*5*	*1*	*1.5*	**0.8**	*3*	*0*	*1.2*
E **Correspond**	**4.8**	*14*	*0*	*5.2*	**1.9**	*7*	*0*	*2.3*
F **Build, compile, check in/out**	**3.2**	*10*	*0*	*2.5*	**1.7**	*7*	*0*	*2.2*
G **Comprehend**	**5.2**	*32*	*0*	*5.7*	**2.0**	*12*	*0*	*2.6*
H **Refactor**	**4.1**	*11*	*1*	*2.6*	**2.2**	*9*	*0*	*2.4*
I **Write new code**	**3.9**	*14*	*0*	*3.0*	**1.7**	*8*	*0*	*1.7*
J **Debug**	**3.8**	*17*	*0*	*3.5*	**1.6**	*8*	*0*	*1.9*
K **Find/check example**	**4.0**	*19*	*1*	*3.3*	**1.5**	*10*	*0*	*2.1*
L **Discuss IDE**	**2**	*2*	*2*	*0*	**1.0**	*1*	*1*	*0*

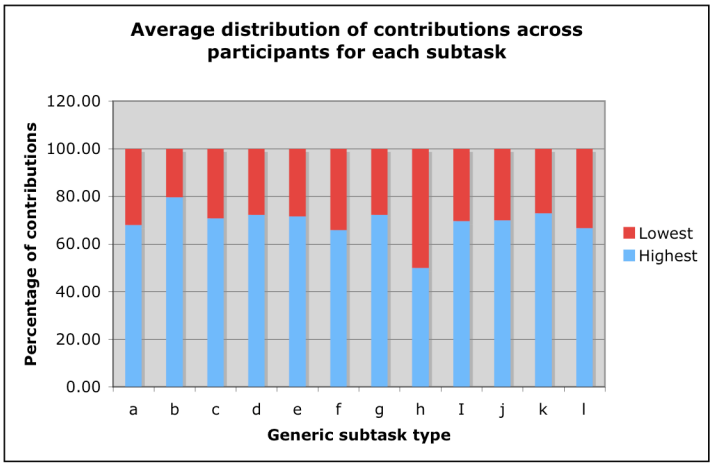

Fig. 3. Average distribution of contributions for generic subtask by participant

Interestingly, the task for which contributions are least evenly distributed (averaging nearly 80:20 between participants) is agreeing strategy. It seems that this is the task on which one person is more likely to take the lead, contrary to suggestions that pair programming lessens the chance of tunnel vision [7]. However, the activity most evenly distributed is Refactoring. This is unsurprising, given the high cognitive load associated with considering both the current and potential future organization of code. Table 6 and Figure 4 below consider the same issues according to role.

Table 6. Most and least collaboration by role for each generic subtask type

Subtask type		Contributions by driver				Contributions by navigator			
		Average	***Highest***	***Lowest***	***Standard Deviation***	**Average**	***Highest***	***Lowest***	***Standard Deviation***
A	**Agree strategy**	**2.4**	*13*	*0*	*2.3*	**2.0**	*13.0*	*0*	*2.3*
B	**Configure environment**	**2.6**	*10.0*	*0*	*3.0*	**1.0**	*8.0*	*0*	*2.0*
C	**Test**	**3.3**	*20.0*	*0*	*3.4*	**1.9**	*12.0*	*0*	*2.5*
D	**Comment code**	**1.8**	*4.0*	*0*	*1.3*	**1.2**	*4.0*	*0*	*1.5*
E	**Correspond**	**4.2**	*13.0*	*0*	*5.3*	**2.4**	*7.0*	*0*	*2.2*
F	**Build,compile, check in/out**	**2.8**	*10.0*	*0*	*2.7*	**2.0**	*7.0*	*0*	*2.2*
G	**Comprehend**	**4.8**	*32.0*	*0*	*5.8*	**2.4**	*12.0*	*0*	*2.9*
H	**Refactor**	**3.6**	*11.0*	*0*	*2.8*	**2.7**	*9.0*	*0*	*2.4*
I	**Write new code**	**3.1**	*10.0*	*0*	*2.5*	**2.5**	*14.0*	*0*	*2.8*
J	**Debug**	**3.1**	*12.0*	*0*	*3.1*	**2.3**	*13.0*	*0*	*2.6*
K	**Find/check example**	**3.2**	*19.0*	*0*	*3.4*	**2.3**	*10.0*	*0*	*2.4*
L	**Discuss IDE**	**1.0**	*1.0*	*1.0*	*0*	**2.0**	*2.0*	*2.0*	*0*

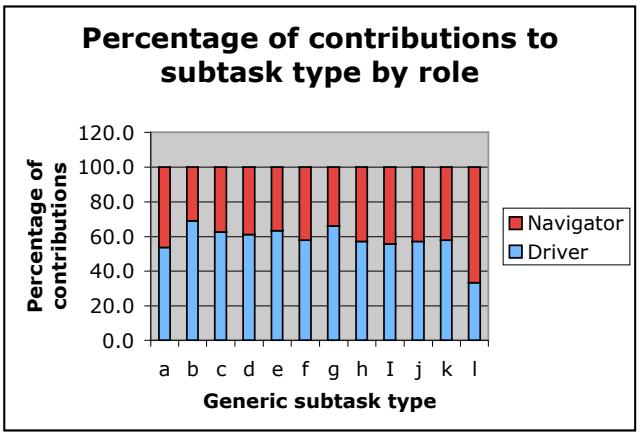

Fig. 4. Percentage each role contributed to each generic subtask type

As illustrated above, contributions were well distributed across roles with the driver contributing slightly more than the navigator across all but one subtask type, 'Discussing the IDE', which happened rarely. This suggests that the driver and navigator roles are less 'tuned to different tasks' but more a convenience in terms of who types. Considering the additional cognitive load of typing, it is surprising that drivers contributed more, however it could be that they were simply commentating on what they were doing.

The two views above (by participant and by role) indicate that the programming pair really are working together on each subtask, rather than each considering a different part of the problem and then pooling results to cover the whole task. However, when one considers more closely the level of collaboration on different types of task, it becomes clear that some lend themselves more to collaboration than others. Similarly, a particular role does not appear to dominate a particular type of task.

5. Conclusion

This report highlights pair programming as highly collaborative, with both partners contributing information to almost every sub-task, irrelevant of role. This contrasts with suggestions that the benefits of pair programming may come from encouraging verbalization, facilitating overhearing or peer pressure from being watched. The profile of the pair programming sessions showed an overall pattern with most time spent on comprehension (understanding existing code and/or the nature of the problem), followed by writing new code and then testing and least time discussing the IDE and commenting code.

While generally very high (over 80%), the level of collaboration varied according to task. Refactoring and writing new code showed the highest level of collaboration and therefore one might suggest that the challenging nature of these tasks made pairing on them most valuable. When the number of contributions per participant was considered, one person was more likely to lead on (i.e. contribute most new information to) agreeing strategy. This is a surprising and interesting phenomena that requires further investigation, as agreeing how to tackle a problem could be considered a highly complex task which one would imagine would benefit greatly from input from both parties.

The studies performed showed very evenly distributed contributions across role, with the driver contributing only slightly more than the navigator. This negates claims that the driver and navigator roles may be oriented toward different types of task, but further investigation is required if we are to fully understand whether a task benefits from the driver and navigator focusing on different aspects (e.g. working at different levels of abstraction).

It should be recognized that the companies studied were an opportunistic sample rather than chosen for being particularly representative of the pair programming community. In addition, while verbalisation occurs naturally in pair programming and the programmer is already being observed by his/her partner, one should nevertheless consider the possible effect of being observed by an experimenter. Finally, it should be noted that the coding of verbalizations as contributing to particular sub-tasks was only undertaken by one person and not blind double coded for accuracy due to resource constraints.

Although the studies report highly positively on the overall collaborative nature of pair programming, they also raise a number of further questions:

- Can software development tasks be designed to foster collaboration?
- Do the driver and navigator contribute at different levels of abstraction?
- What is the power balance in a pair – does one partner or role tend to lead decision making?
- Is collaboration the key to a ‘successful’ pair programming session?
- Is novice pair programming similarly collaborative in nature, and if not, can this be encouraged.

There is still much to learn about the nature of pair programming, particularly if we are to successfully foster collaborative software development in the workplace and teach it in the classroom in order to reap the many benefits it has been shown to have.

Acknowledgements

This work was undertaken as part of DPhil research funded by the EPSRC. The authors would like to thank the participating companies: BBC iDTV project, BNP Paribas, EGG and LogicaCMG.

References

1. Williams, L. et al., *Strengthening the case for pair programming*, IEEE software, 2000. 17(4): p19-25.

2. Jensen, R, *A pair programming experience*. The journal of defensive software engineering, 2003. 16(3): p.22-24.
3. Nosek, J.T, *The case for collaborative programming*. Communications of the ACM, 1998. 41(3): p.105-108.
4. Cockburn, A. and Williams, L, *The costs and benefits of pair programming*, in *Extreme Programming Examined,* G. Succi and M. Marchesi (Eds). 2001, Addison Wesley.
5. Tessem, B., *Experiences in learning XP practices: A qualitative study*. In Fourth International Conference on Extreme Programming and Agile Processes in Software Engineering, 2003.
6. Lui, K. and K. Chan. *When does a pair outperform two individuals?* In Fourth International Conference on Extreme Programming and Agile Processes in Software Engineering, 2003.
7. Williams, L. and R. Kessler*, Pair Programming Illuminated*. 2003, Boston: Addison Wesley.
8. Sharp, H. and H. Robinson. *An ethnography of XP practices*. In Fifteenth annual psychology of programming interest group workshop, 2003.
9. Bryant, S., P. Romero and B. du-Boulay, *Pair programming and the re-appropriation of individual tools for collaborative software development*, In press.
10. Chi, M., N. de Leeuw, M. Chiu and C. Lavancher, *Eliciting self-explanations improves understanding*. Cognitive Science, 1994. 18: p439-477.
11. Hutchins, E., *Cognition in the wild*. 1995, Cambridge, MA: The MIT press.
12. Ainworth, S. and A. T. Loizou, *The effects of self-explaining when learning with text or diagrams*. Cognitive Science, 2003. 27: p.669-681.
13. Ericsson, K. and H. Simon, *Verbal reports as data*. Psychological review, 1980. 87(3): p.215-251.
14. Cox, R., *Representation construction, externalized cognition and individual differences*. Learning and instruction, 1999. 9: p.343-363.
15. Ericcson, K. and P. Polson, *A cognitive analysis of exceptional memory for restaurant orders*, in The nature of Expertise, M. Chi, R. Glaser and M. Farr (eds). 1988, Lawrence Erlbaum Associates: Hillsdale, USA.
16. Schooler, J.A., S. Ohlsson and K. Brooks, *Thoughts beyond words: When language overshadows insight*. Journal of experimental psychology: General, 1993. 122(2): p166-183.
17. Suthers, D. *Towards a systematic study of representational guidance for collaborative learning discourse*. Journal of Universal Computer Science, 2001. 7(3).
18. Jeong, H. and M. Chi. *Does collaborative learning lead to the construction of common knowledge?* Twenty-second annual conference of the cognitive science society. 2000: Erlbaum, Hillsdale, USA.
19. Dillenbourg, P., *What do you mean by collaborative learning*? In Collaborative learning: Cognitive and computational approachs, D. Dillenbourg, Editor. 1999. Elsevier: London, UK. P1-9.
20. Roschelle, J. and S. D. Teasley, *The construction of shared knowledge in collaborative problem solving*, in Computer Supported Collaborative Learning, C. E. O'Malley, Editor. 1995. Springer-Verlag: Heidelberg. O, 69-97.
21. Curtis, B., *By the way, did anyone study any real programmers?* Empirical studies of programmers, E. Soloway and S. Iyengar (eds). 1986. P.256-261.
22. Bryant, S. *Double Trouble: Mixing quantitative and qualitative methods in the study of extreme programmers*. Visual languages and human centric computing. 2004. IEEE Computer Society.
23. Bryant, S., Romero, P. and du-Boulay, B*, Pair Programming and the re-appropriation of individual tools for collaborative software development* (in press).
24. Chi, M., *Quantifying qualitative analyses of verbal data: A practical guide*. The journal of the learning sciences, 1997. 6(3): p.271-315.
25. Dick, A. and B. Zarnett. *Paired programming and personality traits* in Third International Conference on Extreme Programming and Agile Processes in Software Engineering, 2002.
26. Hughes, J. and Parkes, S., *Trends in the use of verbal protocol analysis in software engineering research*. Behaviour and Information Technology, 2003, 22(2): p127-140.
27. Pennington, N., *Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs*, Cognitive Psychology, 1987, 19: p295-341.
28. Beck, K., *Extreme Programming Explained: Embrace Change*, 2000. Addison Wesley.
29. Johnston, A. and Johnson, C.S. *Extreme Programming: A more musical approach to software development*. Proceedings of the 4th International conference in XP and Agile Processes in Software Engineering, 2003. Goos, G., Hartmanis, J. and van Leeuwen, J. (eds): p325-327.