‘Talking the talk’: Is intermediate-level conversation the key to the pair programming success story?

S. Freudenberg (née Bryant), P. Romero, B. du Boulay
IDEAS Laboratory, University of Sussex
s.bryant@sussex.ac.uk

Abstract

Pair programming claims to provide benefits over and above those offered by a programmer working alone. In particular, a number of studies have suggested that pair programming improves software quality. The literature speculates that the ‘driver’ (the programmer currently typing in the code) and ‘navigator’ work together in a complimentary manner, and that the nature of these roles may be key in realizing the reported benefits. Here we dispute two of these existing claims: (i) That the navigator providing a ‘continual review’ of the drivers work and highlighting errors (i.e. acting as a reviewer); (ii) That the navigator is focused on a higher level of abstraction that the driver (i.e. acting as a foreman).

Our findings suggest that the key to the success of pair programming does not lie in the differences in behaviour or focus between the driver and navigator. Rather, we suggest an alternative perspective (the “tag team”) and remark upon the proliferation of talk at an intermediate level of detail in pair programmers’ conversations. This leads us to suggest that producing the type of talk necessary to work effectively together may itself be fundamental in realizing the reported improvements in software quality.

1. Introduction

Computer programming is a cognitively taxing task. Not only is it difficult because of a lack of direct manipulation (Blackwell, 2002) and a ‘product that no-one can see’ (Perry, Staudenmayer & Votta, 1994) but due to many other factors. These include complexity, the need for a multi-layered, multi-dimensional model capable of supporting mental simulations and the sheer amount of knowledge required, its suitable organisation and mechanisms for its access.

One possible method of taming the complexity of software development may be to work collaboratively. In fact, one form of collaborative programming has now been formalised as ‘pair programming’, one of the core practices of the Extreme Programming (XP) methodology. In pair programming, “all production code is written with two people working at one machine, with one keyboard and one mouse” (Beck, 2000).

A wide range of studies have considered the benefits of pair programming in terms of its effect on the quality of the resulting software. These studies have taken place in both academic and commercial environments. In the commercial arena two studies are particularly note-worthy: Nosek (1998), who showed that pair teams significantly outperformed individuals on program quality and Jensen (2003), who showed an error rate three orders of magnitude less for a project with pair programming than other similar projects. In an academic environment, the most cited study is probably that described in Williams, Kessler, Cunningham & Jeffries (2000) in which 13 university students worked individually on a project and 28 chose to work in pairs. The findings showed that code produced by the pairs passed more automated tests over four different programming exercises. It is, however, possible that these findings might have been due to learning effects or the fact that the participants were free to choose whether or not to pair. For example, more able students might have been more willing to work in pairs.

Despite these reported benefits, the cognitive aspects of pair programming are seldom investigated and little understood. An ethnographic study by Sharp and Robinson (2003) provides an insightful story of XP in a commercial environment, but does not assess pair programming from a cognitive perspective. In fact there have been a number of calls for further investigation of ‘the nature of the interactions that underpin these results’ (Wiedenbeck, Ramalingam,

Sarasamma & Corritore, 1999) and when and why pair programming is effective (Chong et al., 2005).

The literature suggests two possible methods by which the programming pair may achieve these benefits. We have called these the 'navigator as reviewer' and the 'navigator as foreman'. By 'reviewer' we mean that the navigator reviews the code that the driver is typing in, pointing out any syntax and spelling errors. By 'foreman' we mean that the navigator thinks about the overall structure of the code-base and whether the code is solving the business problem for which it is intended.

Here we use data from four studies of commercial pair programmers to seek evidence of these two realizations of the navigator role. We begin by outlining in more detail what is meant by the 'reviewer' and 'foreman' and by clarifying how these have been related to 'levels of abstraction'. In section 4 we provide the background to our studies and in section 5 we detail the methodology we have used. We then present the results of our analyses, focusing on the 'reviewer' in section 6.6 and the 'foreman' in section 6.7. In the discussion that follows we present an alternative perspective, that the driver and navigator form a kind of cognitive 'tag team'. We also indicate the proliferation of an intermediate level of talk and theorise about how it might be beneficial. We conclude by summarising our theories, discussing its limitations and suggesting future studies in this area.

2. The navigator as 'reviewer' and 'foreman'

In their book on pair programming, Williams and Kessler (2003) refer simultaneously to both the reviewer and foreman when they state that 'The navigator…observe(s) the work of the driver, looking for tactical and strategic defects. Tactical defects are syntax errors, typos, calling the wrong method, and so on. Strategic defects occur when…what is implemented just won't accomplish what needs to be accomplished''.

The 'reviewer' role is also alluded to in Wakes (2002) suggestions that one navigator behaviour is "The partner provid(ing) an ongoing quality boost: review(ing)" and in describing a commercial pair programming 'experiment'. Jensen (2003) also states that "The navigator review(s), in real time, the information entered by the driver".

There are also further occurrences of the 'foreman' role in the literature. Dick & Zarnett (2002) suggest that "The first is responsible for the typing of code (the driver); the second is responsible for strategizing and reviewing the problem currently being worked on (the navigator)". Beck (2000) also says that "While one partner is busy typing, the other partner is thinking at a more strategic level" later describing this further as "One partner….is thinking about the best way to implement this method right here. The other partner is thinking more strategically". Hazaan & Dubinsky (2003) concur that "The one with the keyboard and the mouse thinks about the best way to implement a specific task; the other partner thinks more strategically. As the two individuals in the pair think at different levels of abstraction, the same task is thought about at two different levels of abstraction *at the same time*".

3. Levels of abstraction

These suggestions actually span two different concepts, both of which are present in the wider 'psychology of programming' literature. First, they delineate between two domains, the programming domain and the problem domain; Second, they suggest that the programming domain may then be further defined using the model of a series of 'levels of abstraction'.

The concepts of 'domain' and 'level of abstraction' appear rather interchangeably in the literature. For example, Brooks (1983) suggests the existence of a set of five 'domains' (problem, identifier, algorithmic, programming language and execution) and Pennington (1987) mixes abstraction and domain in her discussion of a detailed domain (of specific programming operations and variables), a program domain (of routines and files) and a real-world domain. Bergantz and Hassel (1991) also discuss programming as requiring hierarchical models of abstract levels of functionality.

Here we refer to the term 'levels of abstraction' to consider both level of granularity within the programming domain and a separation of program domain from problem or 'real world' domain. We have done this in order to create a single scale and because it is clear that having first distinguished between problem and programming domains it is only necessary to further delineate level of granularity in the programming domain in order to investigate the concepts of 'navigator as reviewer' and 'navigator as foreman'. In our scale, the lowest level of abstraction is program syntax and spelling, and the highest is the problem domain.

According to the literature it could be predicted that these foreman and reviewer roles imply working, and therefore verbalising, at different levels of abstraction. For example, when seeking evidence of the 'reviewer' we would expect the navigator to verbalise at a very

granular (or 'low') level of abstraction in discussions about spelling and syntax, and not to simply wait for their turn as driver to make corrections. For the 'foreman' role, we would expect the navigator to work at higher levels of abstraction, discussing the business problem the general layout of the code

4. Study background

In line with calls for studies of programmers working in an industrial setting (**Cite Curtis**), 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 **Cite self**) 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 approach?**
Banking	1	3	Yes
Banking	4	12	Yes
Entertainment	2	10	Yes
Mobile communications	2	11	Yes

5. Methodology

There is a history to the use of verbal protocol analysis for gaining insight into computer programming. In pair programming, this is even more natural, as the pair are already talking about what they are doing. In fact, a literature review on verbal protocols in software engineering is available (Hughes & Parkes, 2003), 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'. 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.

The methodology used for this work followed the framework for verbal protocol analysis set down by Chi (1997) 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. The coding derived is shown in Table 2. It was based on that used by Pennington (1987) to analyse the level of detail of programmers' statements. In addition, and following the work of Good & Brna (2004) regarding a coding scheme for programming summaries, a BRIDGE code was included for use for utterances bridging the real or problem domain and the programming domain. Finally, in order to consider the hypothesis that part of the navigator role is to correct spelling and programming grammar, a code for SYNTAX was added. The coding scheme is intended to be exhaustive, hence the inclusion of a 'VAGUE' category in order that every sentence has a corresponding code.

Each one-hour recording was transcribed and segmented into utterances (an utterance typically being a sentence). The coding was exclusive, with each utterance having only one code. There were an average of 310 sentences per session and a total of 14,886 sentences were analysed. Four sessions (one randomly chosen from each company) were blind double-coded with an inter-rater reliability of 77%. These four sessions account for 14% of the total number of pages. An example section of coding is shown in Table 3.

Table 2. Scheme for coding utterances by level of abstraction

Code	**Explanation**	**Examples**
SY	Syntax – Spelling or grammar of the program. Spelling is indicated in the transcriptions by single letter capitals. NOT semantics.	S P E L L I N G, dot, F9, 7.
D	Detailed – refers to the operations and variables in the program. A method, attribute or object which may or may not be referred to by name.	This condition, that return value, the list, the counter, what ***this*** returns or gives, getCustomer.
PR	Blocks of the program. Including tests and abstract coding concepts. Also strategy relating to the program and	That loop, truncation, the error handling,

	its structure. General naming standards discussions etc. This could also include cases where the subject of the sentence refers to 'some of them' or 'they all' – i.e. a group of conditions. Anything to do with refactoring. Subsystems or libraries. Directories or paths, even if named.	Oracle, this issue. this part of the program, mock, Mosaic.
BR	The statement bridges or jumps between the real world or problem domain and the programming domain. This may be where a case or condition exists in the code and the real world.	So we need to add a test condition here, to see if the bank account is valid for this kind of transaction.
RW	Real world or problem domain	savings account.
V	Vague, including metacognitive statements and questions about progress or understanding. References to a place on the screen. References to the development environment and/or navigating it's menu structure.	Oh, yeah, I see, that bit at the top.

Table 3. An example section of coding

Partic-ipant	Role (Driver/ Navigator)	Utterance	Code
A	N	If you do a dot dot dot there…umm….and go to…	SY
B	D	You drive…it's easier	V
A	D	It is.	V
A	D	It's just (sub-system name)	PR
B	N	What's (sub-system name) in	PR

6. Results

6.4. The pair programming session

Each pair programming session observed was exactly an hour in length. As the sessions were opportunistically observed, the programmers could equally be just starting, finishing, or indeed in the middle of the task at hand.

We begin by considering the 'shape' of the programming sessions observed. Figure 6.2 shows the average occurrences for utterances at each level of abstraction normalised as a percentage of the total utterances in a sessions, with the maximum and minimum occurrences indicated by 'error bars'. As can be noted, a large number of sentences fell in the 'vague' category. In fact, an average of 57% of the utterances in a session were classed as 'vague'. This is not surprising, as only sentences with a defined level of abstraction would fall outside this category.

There were two main cases where the vague category occurred: First, when utterance did not seem to refer to any level of abstraction, for example questions, such as 'How should we do this?', simple agreements or disagreements ('yes', 'I don't think so') or statements about progress (e.g. 'We've finished that already'). Second, there were some statements where the level of abstraction could not be ascertained simply by reading the transcription. For example, 'that's going to work', which could refer to a line of code, a test, a subsystem, syntax or indeed be a bridging statement between the code and the program

The vague category involves high levels of utterances that, while interesting as a phenomenon, are not relevant to our hypothesis. As such, this category has been removed from further analysis to avoid it having a misleading effect on our results. This categorisation of 'vague' is, in part, due to the post-hoc analysis of the programmer's utterances. However, as the categories used here are particularly stringently defined, it is likely that few, if any, unclassified utterances were of these types.

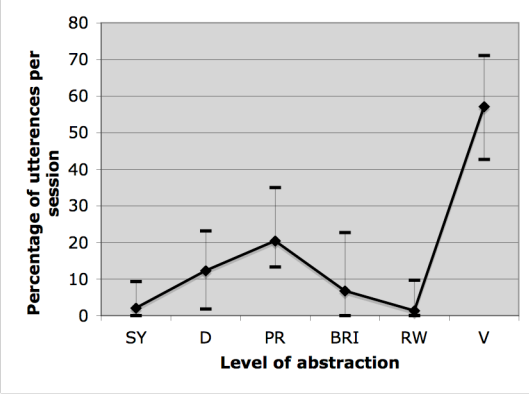

Figure 6.1 Normalised average utterances of each level of abstraction as a percentage of total utterances in a session (error bars showing highest and lowest values).

After removing the 'vague' category, the data presented as normally distributed. A repeated measures ANOVA analysis, with level of abstraction as a between subjects variable showed a main effect for level abstraction, indicating that the mean occurrences of utterances at each level of abstraction differed significantly from each other ($f(1,42)=110.05$, $p < .01$). That is, there is a significant difference between the

average number of utterances at, for example, syntax (SY) level and the number of those at real world (RW) level. Sessions also tended to have fewer utterances at the extreme levels of abstraction (real world and syntax level) and more in the intermediate levels. Planned comparisons in the form of a T-test indicated that there was a significantly higher level of utterances at 'PR' level ($t = 2.71$, $p < 0.01$) than at other levels.

6.5. Level of abstraction and role

All utterances in every session were coded by role according to level of abstraction. Note that these results were not the same as those by participant, as within a session a participant would often change role several times. In order to ascertain whether there were significant differences in the levels of abstraction of the utterances of each role a repeated measures ANOVA was performed with levels of abstraction as a within-subject variable and role as between-subjects. The data was normally distributed. This ANOVA indicated a lack of interaction effects between level of abstraction and role. In other words, the navigators observed did not significantly talk more or less at any level of abstraction than the drivers.

6.6. The navigator as 'reviewer'

As mentioned in Section 2.4.3, there have been suggestions that part of the navigator role might include continually reviewing the work of the driver, pointing out spelling and syntax errors (e.g. Jensen, 2003; Williams and Kessler, 2000). In order to investigate this we must first consider how often these types of utterances occur. The average number of syntax and spelling ('SY') level utterances per session was 14 (of an average total of 620). This amounts to only two percent of the total utterances.

Over all sessions the driver accounted for 47% of SY level utterances and the navigator accounted for 53%). Note that we have not coded which of these SY utterances are corrections and that they could possibly contain a mixture of talking aloud and correcting. It is also likely that the driver would review and correct their own work without saying anything. However, occurrences of SY level utterances were so rare that this is unlikely to affect our findings.

It would seem from our findings, in particular the lack of interaction effects between level of abstraction and role reported in section 6.5, that contrary to what has previously been reported (e.g. Jensen, 2003; Williams & Kessler, 2000) the role of the navigator is not defined by their correcting syntax and grammar significantly more than the driver. In fact, utterances at this level were scarce in the pair programming sessions observed. On the infrequent occasions in which they did occur, they were relatively evenly distributed between driver and navigator roles, with the driver accounting for 47% of 'SY' utterances and the navigator 53% and no significant difference.

It is, of course, entirely possible that a small increase in quality is gained as although the driver more swiftly notices errors while typing, the navigator picks up those which have gone unnoticed and might otherwise have remained undetected. Nevertheless the notable scarcity of utterances of this level suggests that the key to understanding the role of driver and navigator lies elsewhere.

6.7. The navigator as 'foreman'

As discussed in section 2.4.3, clues from the literature also suggest that the driver and navigator might more thoroughly cover the problem space by working at different levels of abstraction. The suggestion is that the driver is working mainly at the lower levels, typing in code and doing other tactical work while the navigator is working more strategically at the higher levels of abstraction, sitting back and considering how the system fits together as a whole and relates to the business domain. Rather like the foreman at a building site might concern himself with how the whole building is fitting together, rather than how each brick is laid. Figure 6.2 depicts how this theory might look in terms of the levels of abstraction we are considering. Note that here we are considering only the 'navigator as foreman'. However, were we also considering the role of 'navigator as reviewer', the level of utterances at SY levels would be reversed for the driver and navigator roles.

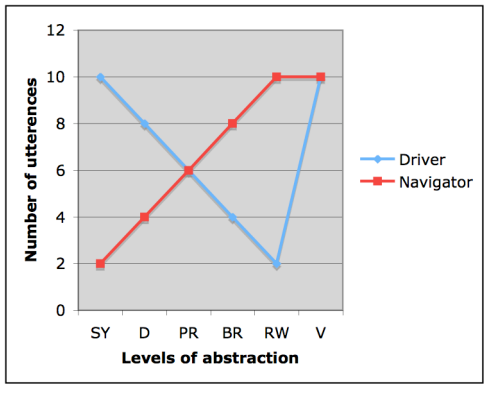

Figure 6.2 Chart showing theoretical levels for utterances by the driver and navigator were they to work at different levels of abstraction.

Rather than the expected chart in Figure 6.2, Figure 6.3 shows the actual average number of utterances of each level per session for each role, making it clear that in the sessions observed the driver and navigator tended to generally talk at the same levels of abstraction.

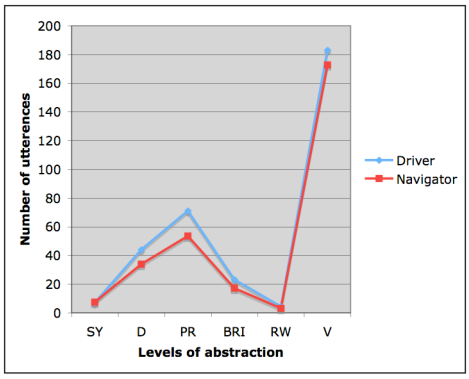

Figure 6.3 Chart showing actual levels for utterances by the driver and navigator.

Also in contradiction to what has previously been suggested (e.g. Dick & Zarnett, 2002; Hazaan & Dubinsky, 2003), the pair programmers in the sessions observed did not show the navigator working at a generally higher level of abstraction than the driver in their discussions. In fact, rather than working at a higher level of abstraction, the pattern of abstraction levels of navigator's utterances are very similar to those of the driver and do not differ significantly.

7. Discussion

7.8. The 'tag team'.

Our findings show that, rather than working at different levels of abstraction, the driver and navigator tend to talk in terms of the same levels of abstraction. In addition, not only do driver and navigator change role regularly, these role changes appear to be very fluid. These findings imply that the navigator continually maintains a firm grasp of what is happening during the session at a number of levels of abstraction

This leads us to suggest that rather than the driver and navigator roles being defined by segmenting the problem space according to level of abstraction, they are more simply defined by the additional physical and cognitive load of typing borne by the driver. In fact, we suggest that the driver and navigator form a kind of 'cognitive tag team', working together, in synchrony, at the problem at hand and then switching role to alleviate the additional cognitive load of typing and providing a running commentary, both of which fall on the driver.

7.9. Intermediate level talk

One interesting finding from studying the levels of abstraction of pair programmers' talk was the significant proliferation of talk at an intermediate level. By an intermediate level, we mean conversation related to 'chunks of code' or 'areas of a program'. For, example mentions of 'the error handling'. In fact, a repeated measures ANOVA analysis with level of abstraction as a between-subjects variable showed main effects for level of abstraction, and planned comparisons in the form of a T-test indicated that there was a significantly higher level of utterances at the intermediate level than the other four levels defined ($t=2.71$, $p<0.01$). We will now consider four theories regarding the benefits of this level of talk and discuss how conversations at this level may be encouraged through the use of eXtreme Programming.

Priming the navigator It is possible that utterances at this level of abstraction help to keep the navigator up to speed with progress. This might occur in order that the navigator is able to 'take over' from the driver on an ad-hoc basis. However, this is unlikely to be the sole reason for utterances at this level, as it has been demonstrated elsewhere that the navigator contributes new information to almost every task the pair performs (**Cite XP2006**).

Providing a missing link Another possibility is that the 'PR' level of talk provides a missing level of abstraction not readily available. Typically the lowest levels of abstraction are clearly displayed on the screen and the highest level is written on a story card. Perhaps intermediate level talk helps to fill an abstraction gap. It is possible that this gap occurs because of the lack of over-arching design diagrams in the XP methodology.

Assisting the driver Another suggestion is that intermediate level talk may help the driver to manage all the levels of abstraction at which he/she is working. In particular, it is possible that 'PR' level utterances provide a form of 'cognitive glue' to help relate available information at other available levels of abstraction to each other.

Increasing peripheral awareness It could be suggested that intermediate level talk renders the work of the pair more understandable. In doing so, it may

provide more opportunity for selective overhearing for those outside the pair, therefore maximizing peripheral awareness (**Cite**).

It is possible that the eXtreme Programming methodology creates an environment that fosters this by enforcing a maximum task size, discouraging the use of diagrammatic representations and encouraging verbal communication. It is also feasible that it may be the additional monitoring or some other facet of pair programming which assists in the production of higher quality software, either as well as or instead of this intermediate level of verbalisation.

8. Study limitations

The studies discussed in this paper have a number of limitations. First, the sample of companies and projects was opportunistic. Second, the data collected was limited to audio recordings. Third, the subsequent analysis therefore focuses on the pairs 'talk' without considering the other ways in which they communicate (for example, where their attention was on the screen, how they manipulated the IDE or when they used particular facial expressions or gestures).

Somewhat unusually, role was considered as a between rather than within-subjects variable. This was due to the manner in which the data was initially coded and precluded observations about how a particular individual behaved when in the driver or navigator role.

There may also be other levels of abstraction outside of those used in this analysis. Indeed there may even be different perspectives along which levels of abstraction could be plotted which might highlight role differences more centrally or more convincingly.

Finally, while double-blind tests of the refined coding schema yielded an inter-rater reliability of 77%, a Kappa test resulted in a coefficient of K=.64. Generally a coding scheme is considered robust with a Kappa coefficient of K=0.7 or above. In this case, disagreements in the coding were largely due to the second coder lacking the contextual understanding and specific programming language knowledge required. In test sessions all disagreements were resolved through further explanation on the part of the primary coder. The overall coding should hopefully retain accuracy as it was the primary coder, with the required contextual and programming knowledge, who performed it.

9. Conclusion

Although literature on pair programming consistently refers to the roles of driver and navigator, little is known about the mechanisms by which they are realised. In this chapter we have considered the levels of abstraction at which drivers and navigators talk to gain insights into the meaning of their roles. In particular we have used verbal protocol analysis to consider two main issues: Does the navigator act as a kind of 'reviewer' by catching syntax and spelling errors? Do the driver and navigator work at different levels of abstraction as a way of taming the complexity of each particular sub-task on which they work?

Our findings have been contrary to suggestions in the literature: First, utterances regarding syntax and spelling are rare, and when they do occur are not predominantly made by either the navigator or driver. Second, the driver and navigator do not work at significantly different levels of abstraction but rather remain in step through the problem working together. Most discussions take place at 'abstract chunk of code level'.

We have suggested that the driver and navigator form a cognitive tag team, where they work collaboratively on each sub-task and the navigator is at the ready to relieve the driver of the additional loads of typing and commentating. We also posit that 'PR' level utterances, referring to the code in an abstract way, may assist in taming the complexity of working at many levels of abstraction at once by providing the 'glue' that holds these levels together and which might otherwise have been missing as it is not readily available representationally to the pair.

10. References

(**NEED TO DO **)

List and number all bibliographical references in 9-point Times, single-spaced, at the end of your paper. When referenced in the text, enclose the citation number in square brackets, for example [1]. Where appropriate, include the name(s) of editors of referenced books.

[1] A.B. Smith, C.D. Jones, and E.F. Roberts, "Article Title", *Journal*, Publisher, Location, Date, pp. 1-10.

[2] Jones, C.D., A.B. Smith, and E.F. Roberts, *Book Title*, Publisher, Location, Date.