Pair Programming and the Mysterious Role of the Navigator

Sallyann Bryant, Pablo Romero and Benedict du Boulay
IDEAS laboratory, University of Sussex, UK
s.bryant@sussex.ac.uk

Abstract
Computer programming is generally understood to be highly challenging and since its inception a wide range of approaches, tools and methodologies have been developed to assist in managing its complexity. Relatively recently the potential benefits of collaborative software development have been formalised in the practice of pair programming. Here we attempt to 'unpick' the pair programming process through the analysis of verbalisations from a number of commercial studies. We focus particularly on the roles of the two programmers and what their key characteristics and behaviours might be. In particular, we dispute two existing claims: (i) That the programmer who is not currently typing in code ("the navigator") is constantly reviewing what is typed and highlighting any errors (i.e. acting as a reviewer); (ii) That the navigator focuses on a different level of abstraction as a way of ensuring coverage at all necessary levels (i.e. acting as a foreman). We provide an alternative model for these roles ("the tag team") in which the driver and navigator play much more equal roles. We also suggest that a key factor in the success of pair programming may be the associated increase in talk at an intermediate level of abstraction.

Keywords: Pair programming, verbal protocol analysis, extreme programming.

1. Introduction

Computer programming is widely considered a highly complex task. A number of reasons have been suggested for this. For example, Blackwell (2002) reports that programming is hard because of the lack of direct manipulation and the use of a notation to represent abstraction. Similarly, Pennington, Lee & Rehder (1995) discuss the difficulties involved in maintaining several levels of information at the same time. They claim that the "major problem for programmers is to coordinate fundamentally different problem spaces". Thus highlighting the complexity involved in working in both the highly structured, logical and constrained domain of the computer program and the fundamentally messy real world. While they focus on the problem and design domains, it may be argued that even in the programming domain, the programmer may need to work at a number of different levels of abstraction, from considering the structure of the code of the system as a whole to the syntax of the command that is currently being typed in. In fact, Brooks (1983) describes the programming task as "constructing mappings from a problem domain, possibly through several intermediate domains, into the programming domain". It has also been reported (Petre & Blackwell, 1999) that experienced programmers maintain very complex, "dynamically stoppable" mental models of the programs they are working on which may help to manage this complexity.

Over the last ten years or so Pair Programming has been put forward as an alternative to the traditional model of programming alone. Perhaps most significantly, pair programming has been formalised as one of the primary practices of the Extreme Programming approach ("XP") (see Beck & Andres (2004) and it is within this arena that we will be referring to it. XP is considered an 'agile' approach to software development, stated as valuing:

> Individuals and interactions over processes and tools
> Working software over comprehensive documentation
> Customer collaboration over contract negotiation
> Responding to change over following a plan
>
> (Beck et al., 2006)

Although there is a running debate regarding the appropriateness of remote pair programming (with each programmer in a different physical location and the collaboration taking place virtually) here we will only consider co-located pair programming, described in Beck (2000) as "two people working at one machine, with one keyboard and one mouse". Furthermore we will be referring to the pair according to the two roles they perform. These roles are commonly known as the 'driver', who is the programmer currently in control of the keyboard, and the 'navigator', who contributes to the task verbally and by other more subtle means (see Bryant, Romero & Du Boulay (2006a). It should be noted that whilst these terms have been drawn from the context of transport and applied to software development, this metaphor may not be relied upon at every level. In fact, the role of a more literal navigator may differ considerably between different modes of transport. The sense being captured is that the driver is the programmer at the keyboard focussing on the task of typing in the code, while the navigator is less preoccupied with the immediacy of code production and can concentrate on the overall direction of the development of the program. The purpose of this article is to attempt to provide a deeper understanding of the behaviours associated with these roles, and consider how they might help to tame the complexity of the programming task.

We begin in section two by considering the potential benefits of pair programming as suggested in the existing literature. In section three we suggest various ways in which cognitive offload may be realised by the pair and we derive two potential roles that the navigator might play: Reviewer and Foreman. In section four we discuss the study background and the methodology by which we analysed our data in search of evidence for the existence of these possible roles. In section five we present our results in terms of the 'shape' of an average pair programming session and findings regarding the navigator as reviewer and as foreman. In the discussion that follows in section 6 we discuss the implications of our findings regarding the navigator role and we consider the limitations of our studies. In section seven we suggest an alternative view of what it might mean to be 'the navigator' in a programming pair and make some suggestions regarding how this work might inform future studies, before summarising our findings and their implications in section eight.

2. The potential benefits of pair programming

Many potential benefits of pair programming have been reported in the literature. A number of these refer to benefits for the whole programming team, for example, Williams & Kessler (2003) suggest that pairing provides an excellent apprenticeship environment akin to those discussed in Lave & Wenger (1991) and Bryant, Romero & du Boulay (2006a) consider the effect that pair programming has with regards to providing more transparency to the rest of the team. However, here we will limit our consideration to the effect on the pair themselves and the work that they produce.

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. The extent to which these academic studies generalise to a commercial environment, both inter terms of the skills and experience of the participants, the nature of the tasks and the environment in which the studies took place should, of course, be considered.

There are also a number of potential cognitive benefits to pair programming, which may help to understand, at least in part, how the reported gains in quality may be obtained. The presence of a second programmer may help to minimise confirmation bias (Hutchins, 1995). This is a phenomenon whereby an individual is more likely to filter information, focusing on that which confirms their current hypothesis, and discarding potentially useful and important information that does not. In fact, Williams & Kessler (2003) allude to this by claiming that pair programming lessens the likelihood of ‘tunnel vision’. They also mention the possible effect of ‘pair pressure’, a kind of positive peer pressure, which suggests that working under the gaze of a critical colleague may result in positive changes in behaviour that may contribute directly to the improvements in software quality considered one of the main benefits of pair programming.

Another cognitive benefit may simply be that working in a pair encourages a programmer to talk. There is evidence to suggest that this type of verbalisation alone may result in improved understanding. That is, it has been shown that simply by talking effectively to oneself (“self-explanation”), one might achieve a greater level of understanding and create a more correct mental model of the problem (Chi, de Leeuw, Chiu & Lavancher, 1994). This is further explained by Ainsworth & Loizou (2003) who consider

verbalisation a kind of 'cognitive off-load', freeing up working memory. In fact, many programmers anecdotally report occasions when they have had a 'eureka'moment on a difficult problem while explaining it to somebody else. It appears that these benefits are not dependent on the person or thing being spoken to so much as the person talking. For example, there have been reports of the benefits of talking to a cardboard cut-out of your favourite "programming guru" (Portland pattern repository, in Williams and Kessler, 2003).

In this article we will consider in more detail the phenomenon of 'cognitive offload' with relation to the driver and navigator roles. That is, given that producing software is such a cognitively taxing endeavour, it seems likely that at least part of the cognitive benefit of pair programming is obtained through lessening the cognitive load of the lead programmer.

3. Cognitive offload and the driver and navigator roles

It is possible that this cognitive offload between the driver and navigator takes place in a variety of ways.

The pair might split up the task at hand so that each are concerned only with a subset of the complexity involved. This has been considered in some detail in Bryant, Romero & du Boulay (2006b). Here one-hour pair programming sessions from four studies were analysed in order to derive the subtasks that the pair performed within that hour. Transcriptions of the pairs' conversation were then further considered in order to ascertain who added new information to which subtask as a means of detecting whether the pair were allocating time to separate sub-tasks or were, in fact, working together collaboratively on them. The findings showed that pair programming was highly collaborative with both parties contributing information to over 80% of all subtasks. This suggests that if cognitive off-load occurs it is not through the 'divide and conquer' method of sub-task allocation.

On the basis of informal observations, the pair programming literature suggests two further methods by which the navigator may assist via cognitive offload. We have called these the 'navigator as reviewer' and the 'navigator as foreman'. As mentioned in section 1, the transport metaphor is only partially appropriate for pair programming. Similarly here we draw metaphors from other domains for convenience, thus it is necessary for us to clarify what is meant by these terms here. By 'reviewer' we mean that the navigator reviews the code that the driver is typing in, pointing out any syntax and spelling errors, similarly to a sub-editor. This might also be likened to a kind of 'back seat' driver. However, unlike a typical reviewer, here the reviewing is done in real time. 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. This is similar to the manner in which a foreman at a building site might concern themselves with the overall aim rather than the laying of each brick. However, here the foreman oversees only one colleague. Retaining with the transport metaphor we might also allude to this role as

a ‘pathfinder’. It should be noted that where these are mentioned to in the literature, there is no suggestion that they are mutually exclusive. In fact, they are more often cited as two of the main properties of the navigator role. In their book on pair programming, Williams and Kessler (2003) refer simultaneously to both of these 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’’. It may be assumed that this viewpoint has arisen from a mixture of the observation of pairs programming and first-hand experience of pair programming as a practice, however this is not explicitly stated.

There is other evidence to suggest the existence of the ‘reviewer’ role. Using a fictional pair programming session as an example, Wake (2002) suggests 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) states that “The navigator review(ed), in real time, the information entered by the driver”. In cognitive terms, this continual ‘reviewer’ role suggests that the navigator catches spelling and syntax mistakes so that the driver does not need to concern themself with reviewing. Essentially, the driver can ‘offload’ the reviewing task onto the navigator.

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)”. In Williams and Kessler (2003), Hayes, a professional programmer suggests that “The tactical programmer is at the keyboard, worrying about the mechanics of getting the code into the machine…..the strategic programmer is thinking about whether the objects are cohesive and loosely coupled, thinking about whether the names are correct, and thinking more about the problem that is being solved.” 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”. In their discussion of Extreme Programming and reflection, 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*”.

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 and domain. We also separate the program domain (concerning digital items and programming concepts) from the problem domain (the physical items, concepts and people that reside outside of the computer). 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.

The existence of these two roles ('reviewer' and 'foreman') form the basis of the work reported here. In particular, according to the literature it could be predicted that these two 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 or the general layout of the code. We also suggest that one of the key aspects of the navigator may be in providing the highly loaded driver with the opportunity to switch to the less challenging navigator role as required.

4. Methodology

In line with calls for studies of programmers working in an industrial setting (Curtis, 1986), the analysis and results presented here are from four, one-week studies of commercial programmers working on on-going tasks in their usual environment. Each study took place at a different company. While a variety of levels of experience were studied (see Bryant, 2004 for insights about the differences in behaviour between novice and more experienced pairers) this paper only considers programmers who had been commercially pair programming for a minimum of six months (Bryant, 2005 provides evidence that this is a useful delineation). As pair composition switched frequently and the organization of pairs and their work was not determined by the studies, some individuals were observed in more than one pair. However, any particular pair was observed working together for at most two one-hour sessions and any individual a maximum of four times. In total 18 different pair combinations were observed and twenty-four hours of pair interactions transcribed and analysed.

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 sessions considered in this article 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	8	Yes
Entertainment	2	6	Yes
Mobile communications	2	7	Yes

There is a history for 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'.

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. This coding was used because it is well established and because we are also specifically concerned with the level of detail that programmers discuss. In addition, and following the work of Good & Brna (2004) regarding a coding scheme for programming summaries, a BRIDGE code was added to Pennington's categories. This code was used for utterances bridging the real or problem domain and the programming domain. This category was added when it became apparent during coding refinement that coding at sentence level meant that an utterence would sometimes usefully refer to both domains. 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. We counted the number of utterences of each type per hour and noted the role of the programmer who produced them. These were the two main variables in our analysis.

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.	Spelling, 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.	That's not actually part of the array that we want. If we have this short description field.
PR	Blocks of the program. Including tests and abstract coding concepts. Also strategy relating to the program and 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.	We did the content test cases in a similar sort of way. I think that before we decommission the database we should take a snapshot of it.
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. How can the feed come in earlier than the trade?
RW	Real world or problem domain	The user might genuinely put his phone down….he wanders away, has a bath, comes back… Just before the time that we set up we call and ask him "can I start now".
V	Vague. Including metacognitive statements and questions about progress or understanding. References to the development environment and/or navigating its menu structure. Utterences where the level of abstraction cannot be ascertained.	I know where you're coming from. Oh, yeah, I see, that bit at the top.

Table 3. An example section of coding

Participant	Role (Driver/ Navigator)	Utterance	Coding
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

5. Results

As each of the perceived roles that we are testing for can be said to relate to levels of abstraction, we begin by considering the ratio of utterances at different levels of abstraction across all the pair programming sessions observed. We then consider our hypotheses regarding levels of abstraction and first the perceived navigator role of 'reviewer' and then that of 'foreman'.

5.1 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.

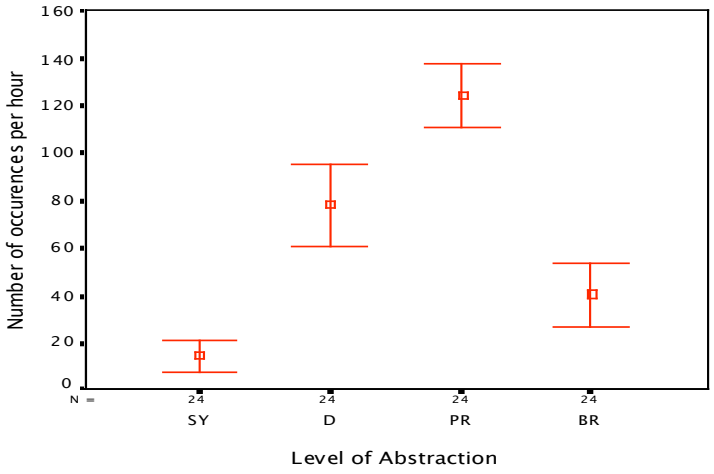
Each one-hour recording was transcribed and segmented into utterances (an utterance typically being a sentence). A sentence was delineated by grammatical flow. That is to say, following a pause in conversation if a participant continued a turn and retained grammatical integrity, it was considered to be part of the same sentence. The coding was exclusive, with each utterance having only one code. There were an average of 620 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%. A Kappa test resulted in a coefficient of K=.64. These four sessions account for 14% of the total number of pages. 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. An example section of coding is shown in Table 3.

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 clearly defined level of abstraction would fall within the utterances that could be coded. There were two main cases where the vague category occurred: First, an utterance may be unrelated to any level of abstraction, for example metacognitive

utterances (e.g. “I’m completely lost”), social interactions (e.g. “ Are you going to the Christmas do”) or bumbling (e.g. ‘Err’). 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. A sample of four sessions was picked at random and the vague utterances were coded according to these two sub-categories to ascertain the frequency with which each occurred. This sample indicated that utterances unrelated to level of abstraction accounted for 78% of ‘vague’ utterances. The remaining 22% of vague utterances referred to an item whose level of abstraction could not be ascertained. Both types of ‘vague’ level utterances have been discounted from the statistical analysis presented here.

Here we are considering the ‘shape’ of the programming sessions observed. First, we discounted the ‘vague’ category to avoid it producing noise in the analysis. Using the Kolmogorov-Smirnov test of distribution, the middle three levels of abstraction were normally distributed (‘D’, ‘PR’ and ‘BR’) however, the more rarely occurring ‘SY’ and ‘RW’ levels were not. This is not surprising due to the scarcity with which they occurred and the different contexts of the sessions which were observed. We then applied square roots to transform the data. There are two potential issues with this approach. First, it is not possible to take the square root of a negative number. Also, this approach has a different effect on numbers between zero and one (which it magnifies) and those greater than one (which it diminishes). Here there were no negative numbers and only very rarely did a category occur zero times. After this transformation all levels except the rarely used ‘RW’ level were considered normally distributed. It is on these transformed values that a repeated-measures ANOVA was performed, treating the levels of abstraction as within-subjects factors, and role as between-subjects. This ANOVA showed that the mean occurrences of the various levels of abstraction differ significantly from each other ($n=24$, $f=157.69$, $p < 0.05$). It also indicated a lack of main effects for role and no interaction effects between level of abstraction and role. Figure 1 shows an average for the number of relevant utterances of each type across all sessions.

Figure 1. Average number of utterances of each level of abstraction in the sessions considered (with error bars showing standard deviation).



Generally sessions also tended to have fewer utterances at the extreme levels of abstraction (real world and syntax level) and more in the intermediate levels. In particular, when compared with the next most frequently occurring level ('D'), a paired t-test showed a significantly higher number of utterances at 'PR' (chunks of program) level ($t(47) = -8.1$, $p < 0.001$). That is, in the sessions studied both programmers were more likely to talk about 'blocks of code' in an abstract way.

5.2 The navigator as 'reviewer'

As mentioned in section 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. It therefore seems unlikely that utterances at this level of abstraction played an important part in the sessions that were observed.

Similarly, the repeated measures ANOVA described in section 5.1 indicated a lack of significant interaction effects between level of abstraction and role. Thus there is no evidence to suggest that the navigator talks significantly more or less at any level (including SY) than the driver. Figure 2 shows SY level utterances by the driver and navigator for each session as a normalised percentage of the total SY level utterances for each session. Over all sessions the driver accounted for 47% of SY level utterances and the navigator accounted for 53%. Note that session 17 had no SY level utterances but is included in Figure 2 for completeness.

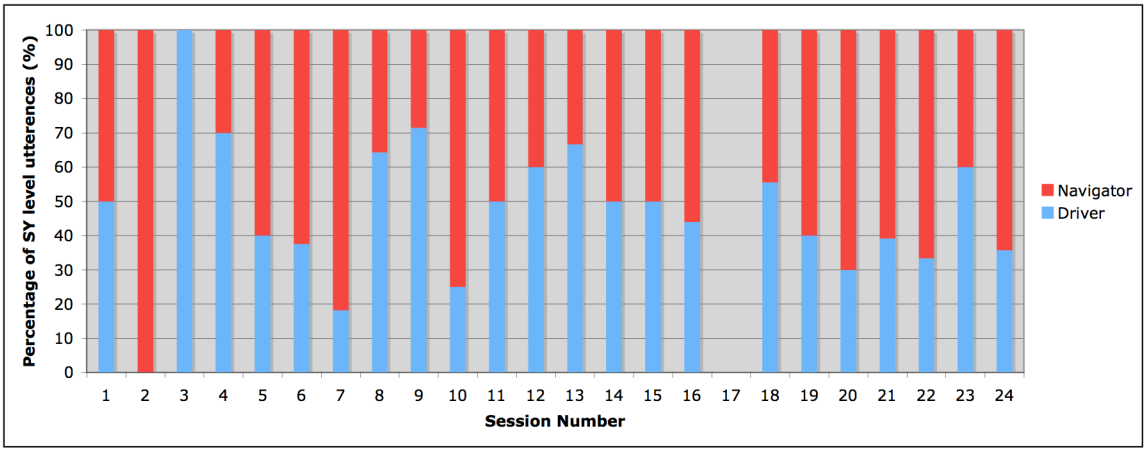


Figure 2. SY level utterances by role for each session

This suggests that helping to ensure software quality by reviewing the driver's code in terms of syntax and spelling is not key to the role of the navigator. In fact, the driver and navigator appear to perform this task almost equally.

5.3 The navigator as 'foreman'

As discussed in section 3, clues from the literature 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 works mainly at the lower levels, typing in code and doing other tactical work while the navigator works 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. This is rather like the foreman at a building site, who might concern himself with how the whole building is fitting together, rather than how each brick is laid. Figure 3 depicts how this theory might look in terms of the levels of abstraction we are considering. Note that here we are considering only the levels of abstraction, were we also considering the role of 'navigator as reviewer' the lines would be reversed for the SY level utterances.

The lack of interaction effects in the ANOVA described in section 5.1 indicate that there is no evidence to suggest particular associations between roles and levels of abstraction. In other words, this lack of interaction effects implies that the navigator does not verbalise, nor we assume work at, a higher level of abstraction than the driver. Rather than the expected chart in Figure 3, Figure 4 shows the actual average number of utterances of each level per session for each role, along with the lack of main effects for role, this makes it clear that in the sessions observed the driver and navigator tended to generally talk at the same kinds of levels of abstraction with the same sorts of frequency.

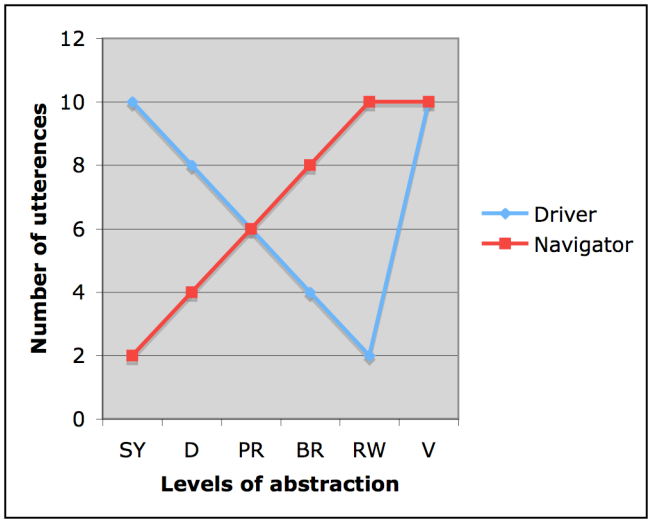


Figure 3. Chart showing theoretical levels for utterances by the driver and navigator were they to work at different levels of abstraction.

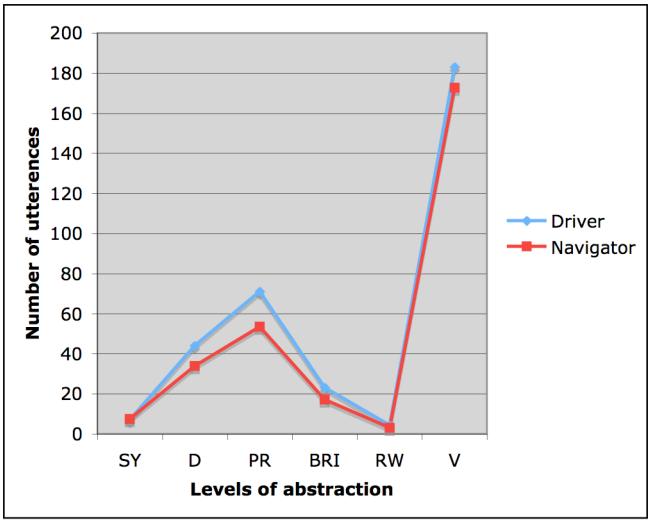


Figure 4. Chart showing **actual** levels for utterances by the driver and navigator.

This is contrary to studies by Chintakovid, Wiedenbeck, Burnett and Grigoreanu (2006) who found that the navigator tended to talk more during session of spreadsheet debugging. However, there are a number of differences that could account for this disparity. First, Chintakovid et al. (2006) were counting the amount of verbalisations in a general sense, rather than contribution of new information. Second, the domain of spreadsheets is different in terms of complexity and problem type to that of object-oriented programming. Third, the users groups were very different (students as opposed to professional software engineers) and finally, the task of debugging plays only a small role in the more general nature of the sessions studied here.

6. Discussion

In general, our findings indicate no significant differences in the level of abstraction between the verbalisations of the driver and navigator. Below we focus more generally on how these findings relate to the perceived roles of the navigator.

6.1 The navigator as 'reviewer'

It would seem from our findings that contrary to what has been previously 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 and on the infrequent occasions in which they do occur are almost evenly distributed between driver and navigator roles, with the driver accounting for 47% of 'SY' utterances and the navigator

53%. 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.

6.2 The navigator as 'foreman'

Similarly 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 a general difference in the level of abstraction of their discussions according to role. 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. Both partners tended to speak more at the 'PR' level, which further contradicts existing suggestions as it shows the driver working consistently at one of the higher levels of abstraction along with the navigator.

6.3 Pair programming as a 'Tag Team'

The results described in section five indicate that the driver and navigator roles may not be as strongly distinguished as has been suggested. Rather, this implies that the pair works more as a kind of 'tag team', sharing the additional cognitive load of typing. In this context we examine two further questions: First, why do both the driver and navigator speak at the level of 'abstract chunks of code' significantly more than other levels, and second, if the roles of driver and navigator may not be defined by constant review or levels of abstraction, then how might they contribute to the production of higher quality software?

It is likely that when driving, a programmer must maintain a good concept of both the overall structure of the program and the task at hand (i.e. typing in code). As such, the driver needs to be operating at several different levels of abstraction at once (as mentioned by Pennington, Lee and Rehder (1995)), for example: considering whether they are successfully solving the business problem at hand; ensuring that they are conforming to project coding standards; making good use of existing modules of code and ensuring that what they type in is 'grammatically correct'. Therefore it is possible that the 'PR' level of utterance provides the driver with a mediating device to help tame the complexity of working at all these different levels at once. In other words, one hypothesis might be that 'PR' utterances provide the 'glue' that holds together the upper and lower levels of abstraction in order to ease the process of navigating and progressing through the programming task. Further work is required to ascertain whether this is, in fact, the case.

It is also possible that the 'PR' level is used as a means of keeping the navigator up to speed in intermediate level terms in preparation for a role change. The driver and

navigator change role regularly and role changes appear to be very fluid with little accompanying explanatory conversation. This implies that the navigator also maintains a firm grasp of what is happening during the session at a number of levels of abstraction. Lower level comprehension is assisted by the ready availability of the code to the navigator. This is helped by the physical layout of the pair (they can both see the screen) and the Interactive Development Environment (IDE) they are using which on many occasions made it easy to simultaneously see the code currently being written and where it fitted into the overall code base. In addition, the IDEs that were used on the projects facilitated an understanding of the structure of the code, by providing a visualisation of the various sub-systems, objects and methods as a nested tree. Perhaps through 'PR' level utterances the programming pair are together providing a missing level of information that is essential if the navigator is to be able to 'take over' in a fluid manner. Therefore articulating what is happening at this level will be rewarded when the driver needs to change roles as the navigator has maintained a clear mental model of their current state.

Finally, it is possible that the proliferation of utterances at the 'PR' level is a side effect of using the XP methodology. In particular, it may be that that as eXtreme Programming intertwines the design and coding tasks, more intermediate level work needs to be done at the programming stage than when using a conventional methodology.

6.3 Study limitations

Before discussing the findings presented above, we must first highlight some limitations of our studies. First, the sample of companies and projects studied was purely opportunistic. That is, they were those companies who 'fitted the bill' in terms of software development approach and pair programming and were also willing to participate in a study. The only further selection for this article was the exclusion of sessions where both programmers had not been commercially pair programming for a minimum of six months. Similarly, the pairs observed were those who were happy to be watched and participate in the study (although only two pairs declined). Therefore the results reported may not generalise to programmers with differing levels of experience, inclination to be observed or a different approach.

Another limitation of this work is the method of data collection. Ideally the transcribed verbalisations would be complemented with video and screen information and other such contextual information. Unfortunately due to the sensitive nature of the projects involved, video recording was only possible at one of the participating companies and screen-recording equipment was not available for use on this project.

We have also assumed that much can be gleaned from the discussions between the programming pair. Although we have based our findings here on the pair's utterances, we are aware of the many other subtle ways in which the pair (and others on the project) communicates. Some of these have been reported elsewhere (Bryant, Romero and du Boulay, 2006b) but are excluded from the analysis reported here. In addition, this focus

assumes that talk is a good indicator of the level at which the pair was working, rather than just the level at which they were interacting.

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. Nor have we analysed who initiates a change in the level of abstraction being discussed, but rather we have focused on how frequently particular levels occur in the pair's discussion. It is, however, possible that a particular role consistently initiated a change in the level of abstraction being discussed.

It should also be noted that role was treated here as a between subjects factor, but could be more accurately treated as within subject. In fact the manner in which the data was coded and analysed allows analysis of role behaviour and participant behaviour, but does not permit insights into how a particular participant behaved according to the role they were performing.

7. Future work

Although the work reported assists in furthering our understanding, the question of what is key to the driver and navigator roles in pair programming remains unanswered. In particular, what does the role of the navigator really entail? Especially given the relative 'cost' to both the driver and navigator of ensuring a suitable collaborative model of the system is maintained.

We suggest that the driver and navigator system provides a 'cognitive tag team' such that the effort of keeping the navigator up to speed is countered with the benefits of having a partner to take over when the driver becomes overloaded. As mentioned in Hutchins (1995) the navigator may also help to avoid confirmation bias, where the driver might otherwise select a plan of action and then selectively accept and ignore information based on this. It is also likely that simply encouraging the driver to talk might have some effect on performance similar to the reported effects of self-explanation in other domains (e.g. Chi, de Leeuw et al, 1994). Further studies are required to ascertain whether these aspects contribute to the driver-navigator relationship and to what extent.

In addition, it would be interesting to see whether the driver or navigator might more often initiate a change to a particular level of abstraction. It is feasible that the results reported here indicate that it is difficult for two people to converse at different levels of abstraction rather than showing whether a particular role has a specific interest in a particular level of abstraction. It would also assist in confirming that the results shown are not due to a response requiring the same coding as the initial utterance to which it is replying.

Our studies indicate that utterances at the intermediate 'PR' level of abstraction may play a key role in pair programming. This suggests that further empirical studies exploring the

relationship between intermediate utterances and software quality may prove fruitful. In addition, it may be possible to define a number of factors that increase the amount of talk at the 'PR' level. For example aiming to optimise 'PR' talk through studies that enforce different frequencies of role change, or providing supporting representations that may encourage discussions at this level may both be useful endeavours. However, it is possible that providing such representations may, in fact, be counter-productive, and that their absence in XP may be the very factor that encourages intermediate talk in the first place. Further investigations are also required in order to ascertain whether 'PR' level utterances do, in fact, provide the cognitive 'glue' required to consider multiple levels of abstraction in parallel. One possibility might be further studies specifically focusing on occurrences of utterances at this intermediate level and categorisation of their perceived purposes.

Finally, a more granular coding by both participant and role could lead to a more fruitful analysis regarding the interplay of differences according to participant and those according to role. A pilot study that begins to investigate this is reported in Bryant (2004).

8. Conclusion

Although pair programming has been shown to be beneficial in both academic and commercial environments, little is known about the mechanisms by which it is realised. Here we have used verbal protocol analysis to consider three main issues: Is there a general 'shape' to a pair programming session in terms of the level of detail which is discussed? Does the navigator act as a reviewer, catching syntax and spelling errors? Does the navigator act as a foreman by working at higher levels of abstraction than the driver?

Here we found a marked and significant difference between the number of utterances at different levels of abstraction. Surprisingly we found that, rather than the navigator primarily acting as a reviewer, pairs rarely discussed syntax or spelling. In fact, the few utterances that mentioned syntax and spelling were very evenly distributed between the driver and the navigator roles. This is contrary to suggestions in the literature that one of the key features of the navigator role is the provision of such a review.

Similarly contradictory to suggestions that the navigator acts as a kind of 'foreman', our findings indicate that the driver and navigator both work at similar levels of abstraction, but that they are significantly more likely to talk about abstract parts of the code (e.g. "the error handling") than to discuss their work at other levels. This is contrary to suggestions that the different roles are characterised by a focus on different levels of detail. However, it may be an indication that verbalising at this intermediate level helps to tame the complexity of having to consider many different levels of abstraction at the same time. This abstract programming level may assist by providing the 'cognitive glue' between the highly abstract problem domain and the very detailed programming levels. Verbalisations at this level may also provide a mechanism for keeping the navigator up to speed with

progress in intermediate level terms and thus assisting a fluid role change when the driver tires and the navigator steps into the driving seat.

The work reported clearly assists our understanding of the nature of pair programming. However, a number of further studies are required before we can identify the key characteristics of the driver and navigator roles that it necessitates.

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. Thanks also to the reviewers of this article for their valuable and constructive comments. The lead author would like to thank Dr. Amanda Harris for her statistical assistance.

References

Ainsworth, S. & Loizou, A. T. (2003). The effects of self-explaining when learning with text or diagrams. *Cognitive Science, 27*, 669-681.

Beck, K. (2000). *Extreme programming explained: Embrace change*: Addison Wesley.

Beck, K. & Andres, C. (2004). *Extreme Programming Explained: Embrace Change - 2nd Edition* (2nd ed.). Upper Saddle River, NJ, USA: Pearson Education.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., et al. (2006). The Agile Manifesto.

Bergantz, D. & Hassell, J. (1991). Information relationships in PROLOG programs: how do programmers comprehend functionality? *International Journal of Man-Machine Studies, 35*, 313-328.

Blackwell, A. (2002). *What is programming?* Paper presented at the 14th workshop of the Psychology of Programming Interest Group, Brunel, Middlesex, UK. (pp. 204-218).

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *Int. J. Human-computer studies, 18*, 543-554.

Bryant, S. (2004). *Double Trouble: Mixing quantitative and qualitative methods in the study of extreme programmers.* Paper presented at the IEEE Symposium on Visual languages and human centric computing, Rome, Italy. (pp. 55-61).

Bryant, S. (2005). *Rating expertise in collaborative software development.* Paper presented at the 17th Annual Workshop of the Psychology of Programming Interest Group, Brighton, UK. (pp. 19-29).

Bryant, S., Romero, P. & du Boulay, B. (2006a). *Pair programming and the re-appropriation of individual tools for collaborative software development.* Paper presented at the 7th International Conference on the Design of Cooperative Systems, Carry-le-Rouet, France. (pp. 55-70).

Bryant, S., Romero, P. & du Boulay, B. (2006b). *The collaborative nature of pair programming.* Paper presented at the The 7th International Conference on

Extreme Programming and Agile Processes in Software Engineering., Oulu, Finland. (pp. 53-64).

Chi, M. (1997). Quantifying qualitative analyses of verbal data: A practical guide. *The journal of the learning sciences, 6*(3), 271-315.

Chi, M., de Leeuw, N., Chiu, M. & Lavancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive Science, 18*, 439-477.

Chintakovid, T., Wiedenbeck, S., Burnett, M. & Grigoreanu, V. (2006). *Pair collaboration in end user debugging.* Paper presented at the IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2006), Brighton, UK. (pp. 3-10).

Curtis, B. (1986). By the way, did anyone study any real programmers? In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 256-261).

Dick, A. & Zarnett, B. (2002). *Paired programming and personality traits.* Paper presented at the Third International Conference on eXtreme Programming and Agile Processes in Software Engineering, Sardinia, Italy. (pp. 82-85).

Good, J. & Brna, P. (2004). Program comprehension and authentic measurement: A scheme for analysing descriptions of programs. *International Journal of Human-Computer Studies, 61*(2), 169-185.

Hazaan, O. & Dubinsky, Y. (2003). *Bridging cognitive and social chasms in software development using extreme programming.* Paper presented at the Fourth International conference in Extreme Programming and Agile Processes in Software Engineering. (pp. 47-53).

Hughes, J. & Parkes, S. (2003). Trends in the use of verbal protocol analysis in software engineering research. *Behaviour & Information Technology, 22*(2), 127-140.

Hutchins, E. (1995). *Cognition in the wild*. Cambridge, MA: The MIT Press.

Jensen, R. (2003). A pair programming experience. *The journal of defensive software engineering, 16*(3), 22-24.

Lave, J. & Wenger, E. (1991). *Situated learning: Legitimate peripheral participation.* New York, NY, USA: Cambridge university press.

Nosek, J. T. (1998). The case for collaborative programming. *Communications of the ACM, 41*(3), 105-108.

Pennington, N. (1987). *Comprehension strategies in programming.* Paper presented at the Second Workshop on Empirical Studies of Programmers. (pp. 100-112).

Pennington, N., Lee, A. & Rehder, B. (1995). Cognitive activities and levels of abstraction in procedural and object-oriented design. *Human-Computer Interaction, 10*, 171-226.

Petre, M. & Blackwell, A. (1999). Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies, 51*, 7-30.

Wake, W. (2002). *Extreme programming explored*. NJ, USA: Addison Wesley.

Williams, L. & Kessler, R. (2000). *The effects of "pair-pressure" and "pair-learning" on software engineering education.* Paper presented at the Thirteenth conference on software engineering education and training, Austin, Texas, USA. (pp. 59).

Williams, L. & Kessler, R. (2003). *Pair programming illuminated*. Boston: Addison-Wesley.

Williams, L., Kessler, R., Cunningham, W. & Jeffries, R. (2000). Strengthening the case for pair programming. *IEEE software, 17*(4), 19-25.