Pair programming and the re-appropriation of individual tools for collaborative software development

Sallyann BRYANT, Pablo ROMERO and Benedict DU BOULAY
IDEAS Laboratory, University of Sussex, United Kingdom

Abstract. Although pair programming is becoming more prevalent in software development, and a number of reports have been written about it [10] [13], few have addressed the manner in which pairing actually takes place [12]. Even fewer consider the methods used to manage issues such as role change or the communication of complex issues. This paper highlights the way resources designed for individuals are re-appropriated and augmented by pair programmers to facilitate collaboration. It also illustrates that pair verbalisations can augment the benefits of the collocated team, providing examples from ethnographic studies of pair programmers 'in the wild'.

Keywords: Pair Programming, Collaboration, Artifacts, Software development.

Introduction

Collaborative programming is common in the commercial world, a fact that is borne out if one considers the regularity with which more than one programmer is seen at a computer terminal working on a debugging problem, assisting in design or simply providing 'another set of eyes'. One form of collaborative programming has been formalised as 'pair programming', one of the twelve core practices of the eXtreme Programming (XP) methodology. XP is classed as an 'agile' methodology, explained [4] 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."

In pair programming, "all production code is written with two people working at one machine, with one keyboard and one mouse" [3]. Two roles have been identified, the "driver", who is currently using the peripherals to manipulate the computer, and the "navigator", who contributes to the task verbally (and more subtley in other ways, as shown

later). Typical reports by practitioners talk about these roles in two ways. First, the navigator is seen as providing a 'constant design and code review' [36] by observing the work of the driver (see also [37] and [38]). Second, the navigator is considered to be thinking at a higher level of abstraction than the driver, considering strategic issues, such as 'how the code that is being written fits in with the overall design' [37] while the driver is involved in the tactical process of writing the code (see also [3]). These two themes are also seen in some of the academic pair programming literature. For example [33] talks of the navigator 'looking for…defects' and claims that he/she is the 'strategic, long-range thinker' while the driver 'is typing at the computer'. These reports also assume that a pair will work together for the whole of an assigned task or for a pre-determined amount of time.

A number of studies have considered the costs and benefits of pair programming [7] [18] [21] [22], and several experience reports have suggested that working collaboratively assists in producing better quality software, improving communications, facilitating knowledge transfer and increasing enjoyment [33]. Some studies have considered why this might be; Flor and Hutchins [12] suggest that when collaborating on software maintenance it is more likely that the correct plan will be chosen. Williams, Kessler et al. [33] suggest 'pair pressure' assists in focusing developers. However, none of these studies have closely considered how the roles of driver and navigator are dynamically realised and facilitated by the artifacts, environment and language used by the pair.

This paper uses the results of four, one-week studies of pairs of commercial programmers. It draws on a detailed ethnographic account to highlight how pair programming is practically accomplished, in particular focusing on how tools are re-purposed and used alongside dialogue to facilitate role management and communication.

The first part of this paper discusses the existing literature on representations and artifacts in software development, considers the methodology used in the studies and gives an overview of the teams observed. Peripheral awareness is discussed and it becomes clear that the benefits generally attributed to collocation are further facilitated by the transparency provided as a result of pair programmers verbalising. The paper then focuses in on the pair. In particular we consider the phenomenon by which tools explicitly designed for individual use are re-appropriated by the programming pair and instead used to assist collaboration. The conclusion then situates this work and suggests future directions.

1. Representations and Artifacts in Software Development

There is evidence that external representations and artifacts play an important role in software development. At a general level, Ackerman and Halverson [1] suggest that any organisation's memory is constructed and maintained by both people and artifacts and Schmidt and Simone [40] highlight the use of artifacts for coordination. More specifically, Gilmore and Green [14] suggest that external elements play an important role in a software developer's mental model. Similarly Davies [9] shows that experts often rely on their tools to compensate for the limitations of working memory. This approach may go some way to explaining why the role of tools and artifacts may be an important one in ensuring "accurate and effective communication about a product no-one can see" [24]. Work by

Grinter [34] and de Souza and Redmiles [35] among others has considered the challenges and tools required for team coordination of software development, however we consider the commercial programming pair and the special role of representations and artifacts with regards to communication and role management.

Comparing self-ratings of pair programming ability with those of peers and managers from pre-assessment questionnaires from the studies reported here suggests that these skills are far from obvious to the practitioner [39]. In fact, experienced pair programmers were more likely to under-rate their ability to work in pairs and those who are inexperienced were likely to be over-confident about their ability to work collaboratively.

2. Study Methodology

The methodology used for this work is ethnographically informed, based on observational studies supplemented with informal interviews, photographic and video evidence of artifact use and the verbal protocol analysis of transcribed sessions. As an experienced commercial software developer, the lead author feels that this facilitated acceptance in the field, however she is also aware that her own experience may lead to different focus than, for example, a social anthropologist may have had (similar issues are reported in Sharp, Robinson et al [29]). In addition, although disruptions were kept to a minimum, the developers were being recorded in order to further analyse their interactions, therefore one should consider the impact of this on their behaviour.

An opportunistic sampling method was used, as there are only a limited number of companies available for study, however only sessions with programmers of at least six month's commercial pair programming experience are considered in this report as a pilot study [6] indicated that pair programmers without this level of experience behave somewhat differently. The data gathered for this paper originates from field notes, informal interviews, photographs, recorded sessions and observations during the studies.

The method used was inspired by the work of Grinter [41] and based on Grounded theory [15]. Grounded theory helps to ensure a solid foundation for hypotheses by basing them on observational studies in the real world. The methodology has also been greatly influenced by the work of Chi [42] who puts forward a compelling argument for analysing qualitative data in a quantifiable manner as a method of integrating the two approaches. Instances reported below relate to themes consistently seen in the data unless otherwise specified. In addition, all of the sessions were recorded in digital audio and three captured on video. These recordings were transcribed and combined with the field notes, informal interviews and photographs to create a rich picture of the interactions from each session. Where possible, examples of actual occurrences are given.

3. Teams Observed

The data was collected from four, one-week studies of pairs of experienced programmers (those with at least six month's continual commercial experience of pairing) in four

different companies. All the companies used an agile approach [4], and several of them used eXtreme Programming [3]. The studies took place in the workplace, with the programmers working on typical tasks. The profiles of the sessions are shown in Table 1.

Table 1. Profiles of sessions observed

	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

36 sessions were observed, transcribed and analysed. Each session was an hour long and a total of 45 programmers participated in the studies. As pair composition switched frequently and the organization of pairs and their work was not impacted by the studies, some individuals were observed in more than one pair. However, any particular pair was observed working together for 2 one-hour sessions and any individual a maximum of four times. In total 18 different pair combinations were observed.

4. A Typical Pair Programming Session

This section describes a typical pair programming session. The day begins with a stand-up meeting. Each pair gives an overview of what they worked on yesterday and any issuesthey encountered. Areas where one task might impact on another are identified. The pairs in the team consider the outstanding tasks and decide which to work on next. A task will usually take about one full 'ideal' programming day to complete. In some cases this will mean continuing to work as a pair on a task not yet completed, but in other cases there may be some negotiation. Here, John and Mary continue working on yesterday's unfinished task.

Once the meeting is finished, they agree to work at John's desk. As the team all pair program, the desks are set out with room for two chairs to fit side by side in front of the large screen. They spend some time discussing progress and decide that now that they have completed writing the automated test script that will prove their code works once it is done, they can get on with the writing the code itself. Mary remembers that there was an outstanding issue and they have a discussion with the allocated business 'customer' in order to clarify the requirement. Once resolved, John pushes the keyboard over to Mary and suggests "you drive". Mary starts up the Integrated Development Environment that the team use and it opens up two initial views. One view shows the suite of automated tests, including the one that they wrote yesterday. The second view shows the system source code, organized into classes and their methods. It is here that the new code will be written.

As they start working, they discuss the approach they are going to take on each sub-task together before continuing. Often they draw informal sketches, type a piece of example

code or point at something on the screen. They switch seamlessly between views and often transfer the keyboard and mouse between them, sometimes with utterances like 'show me what you mean' and sometimes simply indicating their intention to change roles with a gesture. Occasionally, whoever is navigating picks out a typing mistake or syntax error. At one stage Peter, who is working nearby, overhears them discussing an issue that they are having problems solving. Peter knows about this area and they have a three-way discussion. Occasionally one of the pair overhears their name being mentioned by another pair and gets involved in another issue. When there are short breaks in the development task, perhaps while the test suite is running or completed code is being integrated, they take the opportunity to have a break, a social chat or check their email.

Once the code is complete and the test suite runs successfully, Mary picks up a fluffy toy from on top of the integration machine and places it on top of their terminal to show that they are integrating their code with the most recent version of the whole system. This signals to the rest of the team that they should not be making changes at the same time. Once the code is copied across, a full set of integration tests are successfully run, and they place a green sticker on their paper task card and stick it back on the progress chart.

5. Role Management, Communication and Transparency

As shown in the previous section, a typical pair programming session has many subtleties beyond the formal 'driver' and 'navigator' roles described in the XP literature. In fact, the pair programming session takes place in the context of a rich environment of artifacts and talk. Although 'artifacts have been in use for coordination purposes…for centuries' [40], here tools for individual software development are re-appropriated and combined with verbalization to assist in fluid resource and role management and the communication of complex technical issues. Conversations between the pair and specially assigned tokens with mutually agreed meanings provide transparency to the rest of the project about what the pair are doing as a means of highlighting any dependencies or potential areas for knowledge sharing. These issues are addressed individually in detail below.

6. Pair Utterances Assisting Peripheral Awareness

All of the teams studied worked in open-plan environments. This approach to team collocation has been seen to be highly effective. For example Teasley et al. [32] found this approach doubled productivity in terms of function points produced, and took only one third of the time to get to market. This type of layout allows a team to 'overhear' each other and pick up on useful or relevant information. This phenomenon is similar to that reported between journalists [17] but is facilitated by the fact that pair programming demands a high level of verbal communication and therefore renders transparent much information which might be hidden in a more traditional software development environment. In fact, through verbal protocol analysis of one of the four studies included in this paper, pair programmers were shown to produce more than 250 verbal interactions per pair programming hour [6].

This paper also shows that experienced pair programmers produced 27% fewer interactions per hour than those with less pairing experience. Observation suggests that with experience one might become more selective about interactions, better able to make assumptions about ones partner, more successful at using the environment and better able to negotiate a mutually agreeable way forward. Preliminary findings also suggest that, contrary to the XP literature, a pair work at the same level of abstraction, irrelevant of role.

'Overhearing' a pairs verbalizations not only allows a third party to tune in to relevant conversations from surrounding pairs (see Figure 3), but also allows a developer to highlight information that might be relevant to others (see Figure 2). Figures 2 and 3 below provides anonymised examples from different pair programming sessions where Zoe is used as the name of the project member who is external and Andrew and Betty are the names used for members of the pair.

Andrew: Because it'll fail won't it?
Betty: Yeah…that was in…(sighs)…package one wasn't it? And it's not here, so it needs to go into package two I think.
Andrew: OK, so that's something we can make (raises voice) Zoe aware of.
Zoe: What's that?
Andrew: Ummm…something which was, I think in (package name), which has just been abolished.
Zoe: Right, yeah. It's going to be constantly evolving unfortunately, isn't it?

Figure 2. Example of proximity facilitating peripheral awareness through name-dropping

Andrew: Reporting requirements…oh yeah.
Betty: Whenever he's free we're…
Zoe (overhearing): He's free now.
Andrew: Is he?!

Figure 3. Example of proximity facilitating peripheral awareness through over-hearing

On occasions overhearing triggers episodes where a third party joins the pair. In some cases, where the problem required specialist knowledge that the pair did not have, a pair change is negotiated. This allows the developer who had overheard to become part of the pair working on that problem. This fluid re-pairing is contrary to the static, formal nature of pair allocation typically described in the pair programming literature.

7. The Re-appropriation and Augmentation of Solo Software Development Artifacts

This section identifies a number of artifacts, designed for and usually used by individuals, which are re-appropriated or augmented for collaborative use and play an intrinsic role in pair programming. In particular, these artifacts assist in the dynamic negotiation of driver and navigator roles, assist within-pair communication, render work visible and help assure that the programming pair are maintaining a common mental model of the task at hand.

7.1. Keyboard

The keyboard, designed as a solo data input device, consistently became the primary token for 'floor control' - possession of the keyboard signalled who was in the 'driver' role and who was 'navigating'. This is an example of constraints being built into the tools [20] as complications from having both programmers simultaneously editing code are avoided. The keyboard was often used to indicate intention of role change: the driver might slide the keyboard over to the navigator to suggest an exchange of roles, sometimes with an accompanying utterance (see Figure 4 for an example). Interestingly, although relinquishing control of the keyboard in this way seemed acceptable, initiating control of the keyboard was rare. That is, the keyboard was often 'offered and accepted' but very rarely 'taken without offering'.

Andrew: If you…go to…
Betty: (sliding the keyboard over to him) (You) drive…it's easier.

Figure 4. An example of dialogue during keyboard hand-over.

As well as being used for both it's traditional role and as a token for 'floor control', the keyboard also assisted intra-pair verbal communication. One of the methods by which the object of conversation might be highlighted is by use of the keyboards cursor keys. This seemed to take place for a number of reasons including: avoiding the overhead for the driver of switching to another medium; overcoming difficulties with mouse control/dexterity; ensuring accuracy of communication and allowing multi-modal pointing (one partner could highlight with the keyboard while the other used her finger).

7.2. Mouse

Despite also being designed as a solo data-entry device, the mouse was used as a collaborative resource. Control of the mouse was less formal than the keyboard and while in the majority of cases, the driver would control the mouse and the keyboard, in three of the sessions this was not at all the case. It was not uncommon across sessions for the navigator to lean over and use the mouse to 'point' at something on the screen, rather than pointing with their finger or describing the target of interest verbally (see Figure 5 for an example). Presumably this was to avoid both the physical inconvenience of finger-pointing and the time and cognitive load associated with verbally describing.

Andrew: …just test it…and that means you don't have to start faffing about with this…
(uses mouse to point at screen)
Betty: Yeah…I know.

Figure 5. An example of navigator use of the mouse for pointing
(Betty is driving and Andrew is navigating)

In two cases, a wireless mouse was placed on the desk between the two programmers and used as a communal resource to point at and highlight code during discussions, and to

position the cursor. This was possible because the pair were close enough to easily reach the mouse with the appropriate hand. Interestingly, neither pair had any difficulty coordinating mouse or cursor control although this was never discussed or mentioned during observations.

7.2.1. Surrogate mouse

In one session a small ball of paperclips was used as a very informal role control mechanism. Assume the programmer using the paperclips is called B and his pairing partner is A. When A was the driver, B (currently the navigator) would take up the paperclips and make movements and finger-twitches similar to those that the driver was making with the actual mouse. When B wished to assume the role of driver he would let go of the paperclips as a signal to A, who would then relinquish control of the mouse (and keyboard). Once finished as the driver, B then let go of the mouse and once more picked up the paperclips, at which point A almost immediately took up the driver role (and the mouse) once more. Use of the surrogate mouse can be seen in Figure 6.

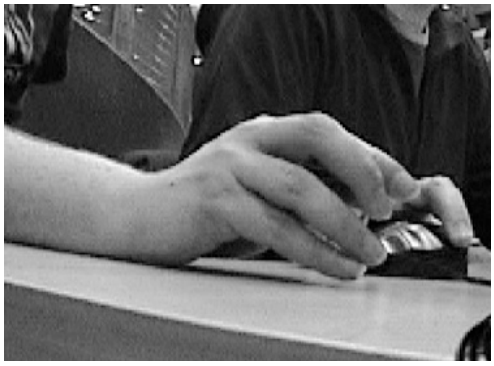

Figure 6. The surrogate mouse

7.3. Interactive Development Environment (IDE)

The code itself played an important role in communication and did not seem to be merely the driver's 'translation' of the collaborative effort. For example, on occasions a period of silence did not indicate the end of an interaction. Sometimes verbal communication between the two programmers would trail off and the interaction would be continued by the driver typing at the keyboard. This was clearly the case where the navigator interjected using agreement protocols normally reserved for conversations (e.g. Uttering "mmmn' or' yes' or 'uh huh'). Examples of this type of interaction is shown in Figure 7.

Andrew: A slightly different side. That's got a…(types code)
Betty: Uh huh

(later)
Betty: Yeah, I think so. Yeah it makes it easier to write accessor methods as well, I think, if you do…(types code)
Andrew: Yeah
Betty: OK, so that's cool

Figure 7. Examples of the code being used to continue conversation

Often the target piece of code being referred to would be identified via pointing as previously discussed. In such cases, the distributed cognition afforded by this representation often led to underspecified statements, as reported elsewhere [12]. An example of this is given in Figure 8, where '*this'* and '*that*' are used to refer to parts of the code being pointed at in a variety of manners (emphasis added).

Andrew: Err…get *this* version of *that*….so *that*'s got *that*….so it's come through *there* now.
Betty: So if you try and run *that* through *there* now.
Andrew: Is *this* a problem?
Betty: *That* should be included in the project.
Andrew: Yeah

Figure 8. An example of the code being used to supplement verbalisation

The Interactive Development Environment (IDE) that was being used facilitated this form of interaction by providing a readily visible and comprehensible representation of the program for both parties. The physical layout of the screen and the programming pair ensured that this representation could be easily read by both and referred to by gesturing either using the mouse or keyboard, or by physically pointing at the screen. On occasions the IDE actually initiated a 'conversation'. This was particularly evident when, for example, the programmers' attention was drawn to an error that had been introduced by a 'red light' appearing next to a particular automated test. This representation would trigger a conversation between the programmers and often initiate a new episode of problem solving.

8. Other Artifacts in New Roles

The role of diagrams and other paper-based external representations in software development is well-documented and key to many development methodologies. The documented benefits of diagrams are many, including their ability to "show complexity in a simple, retainable form" [11], to disambiguate mental representations [8] and to assist in offload, ease problem solving and provide constraints [28].

One of the core values of agile projects is the focus on working software rather than documentation. In particular, the XP methodology downplays the role of system architecture diagrams. Each of the projects observed had some communal representations posted up either in the physical project space (in three of the four companies) or on the intranet (in one of them). The role of these representations seemed to be in allowing the wider implications of a pair's work to be visible and to provide a means of facilitating communication across pairs and ensuring an understanding of the system as a whole.

In addition to these 'project representations', a form of informal, paper-based representation was produced or used during nearly every session observed. These were either informal sketches or lists. Sketches were widely used, they featured in 20 of the 36 sessions observed. These sketches were highly informal (e.g. Figure 9) and in some cases near illegible. This was considered preferable to using more formal or communal diagrams. For example, one programmer suggested "if it's pre-drawn you feel like there's nothing you can contribute", and another that "it feels more comfortable than an official document".

While the representations appeared useful in facilitating communication, the extent to which the non-sketching partner engaged differed widely. On some occasions these representations seemed to be produced to clarify the thoughts of the programmer doing the sketching, and in one particular case, the 'sketch' was merely traced on the table with a finger. In an informal interview, one programmer referred to these diagrams as "like a brain-dump" and another stated "If I scribble it down I can find out if I'm thinking absolute rubbish". This implies that such sketches may at best be playing a highly ephemeral role in communication with the partner, or be used as part of the pragmatics of the interaction (for emphasis, say), or may simply be acting as a cognitive aid for one member of the pair. If this is the intention, then their role may be simply to lower the load on working memory and assist in discovering inferences as documented in [31] or to attempt to externally work with very rich, multi-dimensional models [25].

Where both parties appeared to engage with the representation, its role seemed to be to highlight structure or logic regarding how things related to each other. In one session a timeline was drawn to show the relationship between three conceptual dates and in another a diagram was produced to show how one code method called a number of other sub-routines. Interview data suggests that these were used to assist communication: "It helps communication better than just talking", "Some things are hard to articulate…so it gives you a common diagrammatic language". See Figure 10 for an example of a verbal exchange about creating an informal representation. However, the usefulness of diagrams was considered limited, with comments such as "Between a pair it's easier to just whack out a piece of code" or "You work in small mini-steps, so you can keep it all in your head".

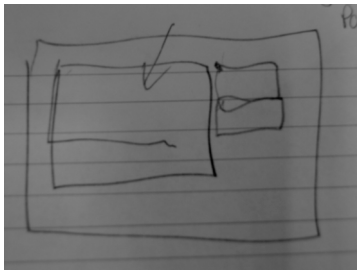

Figure 9. An example informal external representation

Andrew: Oh god (laughs)…Shall we draw the hierarchy?
Betty: Mmn.
Andrew: Because it's…it's more than just one.
Betty: It's loads isn't it.

Figure 10. An example exchange about creating an external representation

Lists were mainly produced as an aide memoire. They were produced in mutual view and both partners often contributed, either at the time the list was produced, or when additional items required adding later. Usually these lists were created in a programmer's personal 'day book' (a kind of diary for each day) or on a separate sheet of paper. In one case items were noted on post-it notes and stuck to the screen. This fits findings by Adelson and Soloway [2], who found that experienced software engineers tend to work in a roughly hierarchical manner, taking notes if something comes to their attention which is not at the current level of detail.

Figure 11 shows an example of the type of list produced. As is obvious from the degree of informality, these lists do not seem to be produced for anyone other that the pair who produce them. Informal interview data shows that they represent more of a check-list, for personal assurance that all the necessary sub-tasks are complete before a piece of work is deemed finished. However, in one case a programmer claimed that they would be useful for another pair who might later work on the same or a similar task. Interestingly, while a number of these lists were produced, they were rarely referred back to and 'checked off' in the sessions that were observed, and never seen to be transferred from one pair to another. This implies that their value might lie more in their creation than in their persistence. Perhaps their very existence was enough of a 'memory jog' without a need to refer to them.

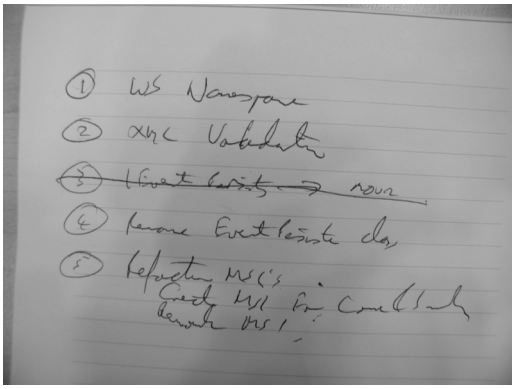

Figure 11. An example list

8.1. Toys

On three of the projects, soft toys were used as tokens. A programming pair would collect the token toy and place it on top of their computer terminal to indicate that they were currently loading new code onto the integration machine. Essentially these tokens were an informal 'locking mechanism' for integration. In fact they were so informal that their effectiveness relied entirely on members of the project understanding and conforming to their rules of use. This is interesting as a more formal, technology based locking mechanism might just as easily have been put in place. It is also contrary to an example in Rogers and Ellis [26], showing that software developers were inconsistent in using a manual whiteboard system for file locking as this was extraneous to their work activities.

In keeping with a number of studies, the physical presence of the toy and the manner by which it is manipulated may play an important role in alerting others to peripheral events which might be of interest (in this case use of the integration machine). This is consistent with studies of news rooms, police operations, traffic control centres and operating theatres, by Heath, Sanchez, Svensson et al. [17] in which participants were seen to "design and produce actions to render features of their conduct selectively available to others" and "encourage another..without interrupting what they are doing, to…notice something..which may have to be dealt with". Robertson [27] stresses the human ability of peripheral awareness as particularly pertinent to this phenomenon. In the pair programming teams, each team member is given the opportunity to notice the change in integration machine control by the action of the developer retrieving the toy. If this is not attained, the toy's placement on top of the developer's monitor is still continually available to the team.

9. Discussion

The account above has given a rich picture of how pair programming is practically achieved. In particular we have focused on the augmented role of artifacts and talk with regards to role management and communication within and outside of the programming pair.

9.1. Role Management

Whereas the XP literature suggests that the roles of 'driver' and 'navigator' have specific properties regarding focus and level of abstraction, little has been written about the management of these roles. The observational studies discussed have shown that the roles not only do not appear as formal in nature as is suggested, but also that they are managed in a number of subtle ways, and practically realised via the interplay of verbalizations and the re-appropriation of traditional software development tools. The keyboard in particular has an important role to play in managing the relationship between driver and navigator. While it might be considered preferable to provide a separate keyboard for each programmer, in fact the use of a single shared keyboard facilitates role management by enforcing a method of floor control and providing an informal means of negotiating role change-over. It also

becomes a common reference, embodying a set of social rules for changing role. For example, one can now relinquish the role of driver but not take it without being offered simply by following a known social protocol for physical items - one is not accustomed to 'snatching' an item that is being used by someone else. More subtly, alternative tokens like the 'surrogate mouse' might be used to facilitate role change by implying a request to drive in a socially acceptable manner.

9.2. Within Pair Communication

Software development is a taxing task. One might consider that the overhead of communicating at the same time as producing software would be cognitively exhausting. The studies considered show that a mixture of verbalizations and artifacts work together to lessen this cognitive load, and in fact, produce tools that not only assist pair communication, but may also help an individual programmer.

The manner in which the programming pair combine verbalizations, gestures, the use of mouse, keyboard, the code and the IDE, external representations and other tokens and seamlessly weave these many artifacts together as a means of communication is nothing short of amazing. In addition to using this rich array of 'props', the pair programmers observed had an implicit understanding of the role and appropriateness of each item and it's role as a method of communication. Only on one occasion, where a partner was less able to physically manipulate the mouse, was this management of resources explicitly discussed.

9.3. Extra-Pair Communication

One of the additional benefits of the verbalizations required to successfully program in a pair is the transparency this lends to the work the pair is engaged in. Traditionally, the work of an individual programmer has little visibility to the rest of the team. However, where a pair is actively discussing their work in an open-plan environment they can easily be overheard by others. This provides opportunities to easily identify potential dependencies, conflicts or areas where assistance might be provided. Where additional attention needs to be drawn to an issue, a pair may raise their voices, or 'name drop' the person whose attention they wish to gain. On occasions a more formal mechanism for gaining attention is required. For example, when integrating new code onto the existing code base, three out of the four projects observed used a soft toy to indicate control of the integration machine.

These studies show pair programmers interacting seamlessly within a rich environment, using artifacts and verbalizations to assist in collaboration within the pair and provide transparency outside the pair. Most interestingly, a number of the tools used to assist with collaboration were initially created for individual use and have been re-appropriated to embody additional constraints or skills which are now required. Perhaps the key to understanding expertise in pair programming lies in acknowledging the skills required to actively situate oneself and interact with fluency within this rich environment.

Conclusion

The analysis draws on ethnographic data in the form of field notes, photographs, video sessions, audio tapes and transcriptions to begin to describe an ecology within which pair programming takes place in four organisations observed. It focuses on the use of artifacts and speech as a mean of easing some of the challenges faced by the pairs, particularly regarding role management and the communication of technical information.

It highlights some of the ways pair programmers facilitate collaboration by re-appropriating or augmenting existing 'solo' tools or by using everyday artifacts in novel ways. It shows some of the rich and subtle ways in which pair programmers communicate and indicates that the verbalisations produced can make activities more transparent and accentuate the benefits of the 'war-room' type environment. Further verbal protocol analysis work is currently being done to analyze these verbalizations with regards to their level of abstraction, the contribution of new information by each partner and the decision making process.

The table below is a summary of the artifacts in question, and the activities they appeared to facilitate:

Table 2. Summary of the roles of artifacts and the activities they facilitate

	Role management	**Within pair communication**	**Extra-pair communication**
Verbalisation	√	√	√
Keyboard	√	√	
Mouse		√	
Code		√	
ERs		√	
Tokens	√	√	√
IDE		√	
Gestures	√	√	

The analysis described above should help provide a clearer understanding of pair programming for those inexperienced in its use. The re-appropriation and augmentation of tools designed for individual use also suggests that programming pairs have some very specific extra requirements from their environments. While this 're-purposing' shows ingenuity and flexibility on the part of the programmers, it suggests that there is scope for the design of more specialised tools for use in collocated pair programming. To the authors' knowledge this has so far only been considered in distributed pair programming environments (e.g. the Additional hand cursor [16] and the Transparent Video Facetop [30]). One must question whether it would be more appropriate to provide specifically

tailored tools for collocated collaborative software development rather than shoe-horning existing resources into collaborative use.

In addition, focus should be given to the skills involved in coordinating and manipulating the variety of tools and artifacts required when considering how to characterize an experienced pair programmer. Perhaps the lack of focus in this area provides some insight into the difficulties described in the introduction that have been seen in assessing ones own and others level of pair programming competence.

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] Ackerman, M. S. and Halverson, C. (1998). 'Considering an Organization's Memory'. *Proceedings of the 1998 conference on Computer supported cooperative work*, Seattle, Washington, United States: 39-48.

[2] Adelson, B. and E. Soloway (1988). A model of software design. *The nature of expertise*. M. Chi, R. Glaser and F. MJ. Hillsdale, New Jersey, Lawrence Erlbaum Associates**:** 185-208.

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

[4] Beck, K., M. Beedle, et al. (2001) The Agile Manifesto. http://agilemanifesto.org

[5] Blackwell, A. (2002). 'What is programming?' 14th workshop of the Psychology of Programming Interest Group, Brunel, Middlesex, UK. J Kuljis, L. Baldwin & R. Scoble (eds): 204-218.

[6] Bryant, S. (2004). "Double trouble: Mixing quantitative and qualitative methods in the study of extreme programmers" Visual languages and human centric computing, Rome, Italy. IEEE Computer Society.

[7] Cockburn, A. and L. Williams (2001). 'The costs and benefits of pair programming'. *Extreme programming examined*. G. Succi and M. Marchesi, Addison Wesley: 223-243.

[8] Cox, R. (1999). 'Representation construction, externalised cognition and individual differences.' *Learning and instruction* **9**: 343-363.

[9] Davies, S. (1993). 'Expertise and display-based strategies in computer programming'. *People and Computers VIII* - HCI '93 conference: 411-423.

[10] Dick, A. and B. Zarnett (2002). 'Paired programming and personality traits'. Third International Conference on eXtreme Programming and Agile Processes in Software Engineering, Alghero, Sardinia, Italy.

[11] Dogan, F. and N. Nersessian (2002). 'Conceptual diagrams: Representing ideas in design'. Second Internation Conference on Diagrammatic Representation and Inference, Callaway Gardens, GA, USA, Springer: 353-355.

[12] Flor, N. and E. Hutchins (1991). 'Analyzing distributed cognition in software teams'. *Empirical studies of programmers: Fourth workshop*, J. Koenemann-Belliveau, T. Moher and S. Robertson (eds). Ablex publishing corporation: 36-64.

[13] Gallis, H., E. Arisholm, et al. (2002). 'A transition from partner programming to pair programming - an Industrial Case Study'. Workshop: "Pair programming installed" at Object-oriented programming, systems, languages and applications (OOPSLA) 2002, Seattle, USA.

[14] Gilmore, D. and T. Green (1984). 'Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies* **21**(1): 31-48.

[15] Glaser, B and A. Strauss (1967). The discovery of grounded theory: Strategies for qualitative research. Hawthorne, New York, Aldine de Gruyter.

[16] Hanks, B. F. (2002) "Tool support for distributed pair programming". Workshop "Distributed pair programming, extreme programming and agile methods" at XP/Agile Universe 2002 , Chicago, USA.

[17] Heath, C., M. Sanchez Svensson, et al. (2002). 'Configuring awareness.' *Computer Supported Collaborative Work* **11**: 317-347.
[18] Heilberg, S., U. Puus, et al. (2003). 'Pair-programming effect on developers productivity'. Fourth International conference on extreme programming and agile processes in software engineering. Springer-Verlag: 215-224.
[19] Holy, L. (1984). Theory, methodology and the research process. *Ethnographic research: A guide to general conduct*. R. Ellen (ed.). San Diego, CA, Academic Press**:** 12-34.
[20] Hutchins, E. (1995). *Cognition in the wild*. Cambridge, MA, The MIT Press.
[21] Jensen, R. (2003). 'A pair programming experience.' *The Journal of Defensive Software Engineering* **16**(3): 22-24.
[22] Lui, K. and K. Chan (2003). 'When does a pair outperform two individuals?' *Fourth international conference in Extreme Programming and Agile Processes in Software Engineering*. Springer-Verlag: 225-233.
[23] Navrat, P. (1996). 'A closer look at programming expertise: critical survey of some methodological issues.' *Information and software technology* **38**: 37-46.
[24] Perry, D., N. Staudenmayer, et al. (1994). 'Understanding software development: Processes, organisations and technologies.' *IEEE software* **11**(4): 36-45.
[25] Petre, M. and A. Blackwell (1999). 'Mental imagery in program design and visual programming.' *International Journal of Human-Computer Studies* **51**: 7-30.
[26] Rogers, Y. and Ellis, J. (1994). 'Distributed cognition: an alternative framework for analysing and explaining collaborative working'. *Journal of Information technology* **9(2)**: 119-128.
[27] Robertson, T. (2002). 'The public availability of actions and artefacts'. *Computer Supported Collaborative Work* **11 (3-4)**, Kluwer Academic Publishers: 299-316.
[28] Scaife, M. and Y. Rogers (1996). 'External cognition: How do graphical representations work?' *International Journal of Human-Computer Studies* **45**: 185-213.
[29] Sharp, H., H. Robinson, et al. (2000). 'Using ethnography and discourse analysis to study software engineering practices'. *Twenty-second International conference on software engineering*, Limerick, Ireland.
[30] Stotts, D., J. McC.Smith et al (2004). "Support for distributed pair programming in the transparent video facetop". XP/Agile Universer 2004, Calgary, Canada. Springer Verlag.
[31] Suwa, M. and B. Tversky (2002). 'External representations contributing to the dynamic construction of ideas'. *Diagrammatic representation and inference.* M. Hegarty, B. Meyer and N. Narayan (eds)**:** 341-343.
[32] Teasley, S., Covi, L., Krishnan, M.S. and Olson, J.S. (2000). 'How does radical collocation help a team succeed?', *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. Philadelphia, Pennsylvania, United States: 339-346.
[33] Williams, L., R. Kessler, et al. (2000). 'Strengthening the case for pair programming.' *IEEE software* **17**(4): 19-25.
[34] Grinter, R. E. (1995). 'Using a configuration management tool to coordinate software development'. Proceedings of conference on Organizational computing systems: 168-177, August 13-16, 1995, Milpitas, California, United States.
[35] de Souza, C. R. B., Redmilles, D. F. and Dourish, P. (2003) 'Breaking the code', Moving between private and public work in collaborative software development'. Proceedings of the 2003 International ACM SIGGROUP conference on Supporting group work, November 09-12, 2003, Sanibel Island, Florida, USA.
[36] Wake, W. (2002). 'Extreme Programming Explored', Addison-Wesley, Boston, USA: 63.
[37] McBreen, P (2003), 'Questioning Extreme Programming', Addison-Wesley, Boston, USA: 80.
[38] Auer, K. and Miller, R. (2002), 'Extreme Programming Applied: Playing to win', Addison-Wesley, Boston, USA: 171.
[39] Bryant, S. (2005), 'Rating expertise in collaborative software development', 17th workshop of the Psychology of Programming Interest Group, Brighton.
[40] Schmidt, K. and Simone, C. (1996), 'Coordination mechanisms: Towards a conceptual foundation of CSCW systems design'. Computer Supported Cooperative Work: The Journal of Collaborative Computing. 5 (2-3). Kluwer Academic Publishers: 155-200.
[41] Grinter, R. (2001) From local to global coordination: Lessons from software reuse. International acmsiggroup conference on supporting group working.
[42] Chi, M. Quantifying qualitative analyses of verbal data: A practical guide. The journal of the learning sciences 6(3): 271-315. (1997).[18],