"What's the Sense of Using Echo?" Social Interaction in a Pair Programming Session

Chris Bates	Kathy Doherty	Karen Grainger
C3RI/CCRC Sheffield Hallam University Sheffield, United Kingdom	C3RI/CCRC Sheffield Hallam University Sheffield, United Kingdom	C3RI/CCRC Sheffield Hallam University Sheffield, United Kingdom
c.d.bates@shu.ac.uk	k.h.doherty@shu.ac.uk	k.p.grainger@shu.ac.uk

1 Abstract

In this paper we report field observations of professional developers using pair programming as they work together to find a way to refactor and rebuild existing prototype code. We use discourse analysis to show how talk about problems in the code and possible solutions constructs the meaning of, and shapes the purpose for, their task. We present the use of ethnomethodological observations and discourse analysis as useful tools for the researcher who wishes to understand software development in the wild rather than in the laboratory. We show that the importance of talk is under-theorised in thinking about agile methods.

2 Introduction

The Agile Manifesto, (Beck et al. 2001) re-imagines software development as a socially embedded technical process. At its heart lie four principles including, in the manifesto's words, valuing "Individuals and interactions over processes and tools". The manifesto thus foregrounds the subjective experiences and communication practices of developers as an *integral* aspect of software programming and the management of projects. This reflects the day-to-day realities of delivering code - very little software gets developed by one person acting totally in isolation. Indeed, larger projects are highly managed cooperative social activities within specific institutions – the corporation, the Web services provider, the game development company and so forth. The Agile Manifesto is an attempt to recognise software programming as an essentially human activity based on creative problem-solving, co-operation and collaboration.

Working practices within Agile software engineering houses are far from fixed, but all Agile methods are predicated on the idea that improved communication between developers and management of relationships between customers and software houses will lead to better product or better projects. Extreme Programming (XP) for example, is one of the more popular instantiations of Agile, although it does pre-date the actual manifesto by a number of years. In his discussion of XP, Beck (2000) spends a lot of time describing the need for better oral and written communication. In the preface he writes that XP is different from other approaches to programming for a number of reasons. One of these is its "reliance on oral communication, tests, and source-code to communicate system structure and intent".

Beck writes that XP has four values: communication, simplicity, feedback and courage. Each of these is described in idealised, almost religious, terms giving the impression that if developers just talked more and trusted each other everything would turn out right. However, it seems to us that, in terms of theory, the "communication" element of the Agile Manifesto

and in discussions of agile methods, remains stubbornly implicit. Furthermore the social processes that are part and parcel of Agile would be better understood if grounded in established theories of talk-at-work and talk-in-interaction. Our work explicitly sets out to examine interaction within agile teams, employing an ethnomethodological framework to analyse interaction between pair programmers as they rework some failing code.

Section 3 introduces the use of ethnomethodology to study work, section 4 looks at two studies of pair-programming, section 5 looks at our data which is analysed in section 6.

3 Looking for Understanding

The agile community can give the impression that working code is all that matters. The artefacts developers produce are somehow self-explanatory and if we simply read the source we will understand what it means and how it should be used. But code is necessarily contextual; reading the source will not reveal to us what the developer was thinking as they wrote it. Kernighan & Pike, (1999) advise us to use in-code comments to "aid the understanding of a program by briefly pointing out salient details or by providing a larger-scale view of the proceedings". However, such comments tend to discuss the operation of the code rather than providing a justification for its structure.

Several ideas and techniques from sociology have found their way into software development but the most widely adopted is probably ethnomethodology. This theoretical approach examines how we create understandings of the world and use them to build social situations, which makes it particularly well-suited to the study of work and workplaces (Drew & Heritage, 1992). Ethnomethodology tells us that in everyday life we engage in practical social activities that are accountable, that is they can be observed, understood and evaluated by others and that this understanding is context sensitive (Garfinkel, 1986). We turn the ethnomethodological spotlight onto software developers and, since agile methods are explicit expressions of a set of social rather than technical practices, they provide an interesting case for study.

Ronkko (2007) suggests that ethnomethodology offers firm theoretical grounding from which to explore and understand the activities and challenges that are part and parcel of software engineering. He argues that software programmers are inevitably engaged in a search for "adequate indexicality" as they deal with changing requirements and strive to develop a product – software - that is essentially complex and 'invisible' (Brooks, 1995). Source code is inherently "plastic", changing through the life of a product as it is maintained. As requirements change source code is frequently remodelled such that it can lose its original form and intent.

Indexicality refers to the way in which meaning (e.g. the specification of a requirement) depends on the specific context of use or production, including mutually understood norms and expectations. Programmers are always searching for adequate indexicality to make sense of the task in hand and thus routinely work in a context of "incomplete communication". Attempts to disambiguate the current programming task, or at least establish a framework of shared understanding, thus become a feature of programming related social interaction. We see this in the pair programming analysis that follows, where the history of previous instantiations of the code is made relevant (and actively constructed) as part of a delicate and dynamic negotiation of its existing design problems and proposed solutions. A key focus for our analysis is the way in which accountability for the existing design is invoked and

responded to by participants. Within ethnomethodology, accountability refers to the various ways that people present and explain their activities and the activities of others so as to render them sensible, understandable and "proper" – aligned with prevailing norms and expectations (Button & Sharrock, 1998, Dourish, 1995). Being held accountable can operate as a powerful constraint on social action as members orient to, and actively renew, what counts as an understandable action or acceptable conduct. When developers talk about code several potential layers of accountability are in play - to the company, the project, the team, each other and to themselves - within a complex context of work based roles and reputations and where the ownership of creative design input and relationships within the team are potentially at stake. Software design talk is thus potentially 'face-threatening', and, as we show, requires careful interactional management. In our analysis we apply the notion of 'face' and 'face-work' taken from Goffman (1955). Goffman defined face as the way that we present ourselves to others when we interact. It is "the presentation of a civilised front to another individual" which, Ting-Toomey (1994) writes, is culturally located and which provides a "claimed sense of self in an interactive situation". Goffman (1972) further argues that social interaction is structured by rules, which are protective of one's face and the face of others, within a shared (and negotiated) understanding of what counts as acceptable conduct. We shall see that face-work is an important part of the interaction between the paired developers in our analysis.

4 Studying Pair-Programming

There are a few studies of pair-programming. Working within an experimental paradigm, Muller & Padberg (2004) hypothesised that "the performance of a pair is dominated by how comfortably the developers feel during the pair session". They tested their hypothesis through simple experiments. A group of student programmers was paired so that those with most experience worked with the least experienced. The students performed some simple tasks then answered a question about how they individually felt during the experience. The researchers found a correlation between the performance of a pair, as measured through time taken to complete the task, and their comfort, or *feelgood*, during the task.

Performing a task under laboratory conditions is very different to the kind of activity that programmers engage in at work. With this is mind, Sharp et al. (2004) undertook an ethnography of a mature XP team to examine how work practices such as pairing affect the quality of code. One advantage which ethnographic fieldwork has over laboratory experiments is that all phenomena found in the workplace become part of the researchers' data set. Sharp and colleagues are clear that talk was central to the processes of the developers they were observing both in daily stand-ups and in paired coding sessions: "[t]he oral tradition of the developers we observed, based on minimal transient documentation, was particularly significant and they went to some trouble to ensure that appropriate communication was maintained. The vehicle for facilitating this shared understanding was one of the XP practices: that of metaphor, a simple story or model to share amongst the team about how the software fits together". However, the detail of the talk-in-interaction of the developers is not examined in this study so we learn little about how software programming is embedded in communication practices or how design problems and solutions are negotiated and resolved.

5 Data from the field

The work reported here is based on observations taken during summer 2010. As part of an

on-going study into the ways in which developers communicate about programming we spent some time at a small software house that is avowedly agile, watching them write code. We made audio recordings and took contemporaneous field notes. Background information was gathered through informal conversation rather than through structured interviewing and is being used to provide context for our analysis. In our work the context within which the talk happens is central to our analysis. The main field worker is an experienced software engineer for whom the details of programming are commonplace knowledge. Analysing these interactions with an understanding of the culture of programming means that we can focus on detailed phenomena without the distraction provided by trying to understand and interpret specialised technical work. This approach follows that of Goodwin (2000) in contrast to those ethnomethodologists who view context as anathema, wishing to let their respondents' words quite literally speak for themselves.

5.1 Background to the fieldwork

E* is a small UK company which develops mobile telephony applications. Their software runs on the network and on the telco's servers rather than on the handset and is used to send bulk SMS messages. The software manages all stages of the messaging process including auditing and billing as well as message creation. E* employs fewer than ten developers with a similar number in operations and another ten or so working in marketing and sales.

Management and developers at E* describe their software development approach as agile. They are actually very evangelical about the benefits they get from agility, being involved in local user groups and blogging about agile methods. This has caused them problems in the past as experienced developers often find the working environment difficult so that E* now tends to recruit only recent graduates who it can train in its own approach. The development team might be evangelical about agile but they are not adherents to any particular approach. Rather than defining and restricting their approach to scrum or XP or kanban as so many agile shops do, E* use a pick-and-mix approach to tools and techniques, selecting those which fit best with their working practices and ethos. The most obvious practices they use are story cards, estimation and pair programming and test-driven development.

This paper covers a session early one afternoon. The pair is Darren, an experienced developer who has been at E* for a number of years and Andy, a recent graduate in his first year at the company. Darren and Andy are an experienced pair who work together often but not exclusively.

6 Analysis

In the interaction that we observe here, Darren and Andy are re-working code, which was started and abandoned a few months previously. The code synchronizes contact lists between the server and a user's phone. Our field notes include briefings from the developers about their work and about the specific task we see them undertaking here. The task is framed by the knowledge of the earlier failure – made evident by the previous abandonment of this work - without which they would not be engaged in the present task.

At the start of the session the two developers are discussing the data that they need to store.

The figures in this paper are transcriptions of this talk. Coding conventions are listed at the end of the paper.

1	Darren	This is kind of the start of a process and over time it might evolve

2		(1.0) or it might not
3	Andy	ha ha
4	Darren	tends to depend how things are used (.)
5		Errm (1.8) yes I think for the implementation side of it the first pass our implementation is about 'cause what we kinda doin' is a provider that's got its own little database here
6		(1.5)
7		tha' that's isolated from that so it can store (.) contacts (.) these'll be probably creatin' it's probably a session-type table
8		(1.3) a user-type table (0.8) ah (1.4) and then the data store (1.5)

Figure 1: Introducing the task

Darren's initiating turn informs Andy as to what they are about to do. This immediately positions him in the role of 'expert'. He frames the conversation they are about to have as "the start of the process" but, rather than directing Andy to undertake tasks, he launches into an exposition of the future, of what will happen as they work. His use of hedges such as "kind of", the modal verb, 'might', and the expression of doubt ("or it might not") reduce any certainty in the statement and hence mitigate any potential face-threat to Andy. Pair programming is in principle egalitarian and if one programmer directs the definition of the task too much, the balance is potentially threatened.

In this opening sequence of talk, Darren holds the floor for the first few moments. At line 2 Andy acknowledges the light-heartedness of Darren's contribution but Darren maintains his turn with a combination of fillers ("erm"), false starts (tha' that's isolated…) and pauses while he refers to/adds to a sketch he is making which shows the rough structure of the code Although Darren appears to be informing and consulting with Andy, notice that on line 5 he says "yes" after a pause, as if responding to himself.

Using our field notes alongside their conversation we understand that they are going to develop a service, the provider, which will supply contact details. The concluding "here" on line 5 refers to an object on the diagram that he is drawing. With "what we kinda doin' is a provider that's got its own little database here" once again he is actively performing and constructing an identity as ‘the expert’, the one who does the thinking and has the answers. But he mitigates this, by using "we" and "kinda", constructing a sense of provisionality for the task at this stage.

9	Andy	what's the sense around using echo?
10		(1.7)
11	Darren	at the moment that functionality's all kind of been turned off 'cause it it got to a certain point (1.1) and it wasn't really
12		(2.8)
13		to [improve the user experience]
14	Andy	[what was the point of th]at?
15	Darren	the (.) the pla' when we did it
16		to echo (.) then that was (.) the database was here the synchronisation was going

		and the contacts were pushed up
17		(1.6) into this database. THERE WAS various issues (.) and from a usability point of view it wasn't (1.0)
18	Andy	Hmm mmm
19		(1.0)
20	Darren	err >working that< well
21		and also echo (0.9) echo was kind of err (1.7) Yeah (1.0) it had problems
22		(0.3) it was kind of a big
23	Andy	laughs
24	Darren	laughs
25		a big test (.) but this is you know this is kinda tryin' a make it easier

Fig 2: Taking a turn

At line 9, Andy's question "what's the sense around using echo?" Here, Andy is referring to a sketch we saw them make earlier in the day, which started as a drawing of the existing, failed, design. Andy introduces a new topic by asking for an 'account', thus indicating that the state of affairs under discussion is problematic. The implied criticism of the previous developers (querying the "sense" of their actions) can be seen as a challenge and thus face-threatening for Darren, in terms of authorship of the existing code and also the forward-looking direction of the task initiated by Darren. Darren's response is hesitant – the pauses around this statement total over 5.5 seconds. When discussing the database design he was clear and the pauses were due to his sketching. On lines 10 through 13 the pauses are longer and he is not distracted by any other activity.

Darren says that the functionality was "kind of" turned off, placing it in a nebulous third state which is neither on nor off. This is more as a face-saving device than an accurate technical description. Darren provides the justification, "cause it got to a certain point" and that the code wasn't improving the user experience. The latter statement is a perfectly normal reason for removing or refactoring a piece of code yet Darren's statement that the code is turned off does not stop Andy from pursuing an explanation. Indeed Andy's turn at line 14 overlaps Darren . This interruption comes at a possible turn transition point following Darren's long delay and overrides the explanation. This question challenges the actions of the previous programmers and continues to threaten Darren's face in terms of his claimed identity as the experienced leader of the pair that wants to get on with things.We can't know what Darren's motivations are in asking for the account but can see how it manifests and becomes consequential in the interaction in terms of the ongoing negotiation of the work. As he starts to talk on line 16 his volume increases when he says "there was", after the pause he rushes through the explanation. Darren is involved in more facework here. Darren's joke that the code wasn't "working that well" diffuses the tension and creates a turn away from the problem code. On line 21 Darren begins to lighten the mood by moving attention to the echo class and joking about its state. By describing it as problematic Darren lets Andy know that, as the 'expert' he is aware of its limitations and that they won't be re-using this code. When he says "it was kind of a big" then pauses. Andy takes this up as a joke by starting to laugh demonstrating that a shared understanding has been reached, enabling them to move on from the echo issue.

7 Discussion

Although this paper examines just a few minutes talk from a single working day, those fragments are enough to begin to assess some of the ideas we introduced earlier. How are indexicality and accountability made manifest, does context matter when examining code? And, does ethnomethodology help us understand programming?

In this session the two developers are clearly trying to understand code which is new to one of them and which the other hasn't seen for some time. Yet they approach this from different directions. Andy tries to understand the code and design. When he asks "what's the sense of using echo?" he has found something which he regards as accountable. To use Ronkko's formulation, echo presents an indexicality problem for Andy whereas for Darren it is simply something which is "switched off".

This discussion about the meaning of code demonstrates the importance of context and that it is a live issue for programmers. The differing contexts of production and use alter meaning. Some of the code in an application represents domain objects and exposes its intent and context relatively easily. Domain classes often have clear and meaningful names, for example, with interfaces that many programmers will understand. Other classes implement functionality within the application: processing data for the domain classes or linking to other applications. The meaning of these classes is inherently less clear. When Andy looks at the echo class he sees it outside of the context of its production. He cannot understand what it is meant to do or why it exists, his question positions Darren as the expert who is accountable for the code but also challenges Darren’s direction of the task, insisting that they deal with the issue of echo before moving on. We see that management of face becomes thus important for Darren. The question, and positioning, threatens his face by both forcing him to act as the expert and by challenging his expertise. If Darren were involved in the original creation of the code then he ought to be able to explain its structure and meaning.-

Beck, (2000), repeatedly writes that talking about code is an essential activity for developers. No doubt Beck's idealised programmers would follow Andy's line and investigate echo in detail. However, our excerpt shows that there is much more at stake in agile programming than the ultimate goal of effective design: the two are also engaged in the task of managing face here, as Goffman (1955) tells us they must.

8 Conclusion

Software development is a social activity. The agile movement foregrounds the social nature of development and builds work practices and processes which embody it. Pair programming is an example of such a practice. Two developers working together have to talk about the problem and the solution and have to do so with an appropriate level of detail. Because it is a social interaction, they have to establish and construct their relationship with one another via face work. Programmers cannot talk about algorithms or data structures in a social vacuum.

Using discourse analysis we were able to discover the conversational strategies through which the two developers accomplished professional and relational goals. Each of them had interpersonal and professional goals to achieve. This is typical of our interactions at work where we try to complete tasks cooperatively whilst managing relationships with our colleagues. In analysing a few minutes of talk we were able to uncover this range of talk and to show that the way programmers talk about a problem can impact directly upon the way they choose to solve it.

The agile manifesto and its implementation in methods such as XP privileges talk. We have shown that talk about programming cannot be separated from other social interactions which compose our working lives. Whilst we would agree that talk, as a living communication, is more useful than static documentation, we would not see it as problem-free.

9 Bibliography

Beck, K., 2000. *Extreme Programming Explained: Embrace Change*, Addison-Wesley.

Beck, K. et al., 2001. Manifesto for Agile Software Development. *Manifesto for Agile Software Development*. Available at: http://www.agilemanifesto.org/ .

Brooks, F., 1995. *The Mythical Man-Month anniversary ed.*, Addison-Wesley Longman.

Button, G. and Sharrock, W., 1998,*The Organizational Accountability of Technological Work*, Social Studies of Science, 28:1, Sage Publications, pp73-102.

Dourish, P., 1995, *Accounting for System Behaviour:Representation, Reflection and Resourceful Action*, Proceedings of Computers in Context '95.

Drew, P. and Heritage, J., 1992, *Talk at Work*, Cambridge University Press.Garfinkel, H., 1986, *Ethnomethodological Studies of Work*, Routledge and Kegan Paul.

Goodwin, C., 2000. Action and embodiment within situated human interaction. *Journal of Pragmatics*, 32, pp.1489-1522.

Goffman, E., 1955. On face-work: an analysis of ritual elements in social interaction. *Journal for the Study of Interpersonal Processes*, 18, pp.213-231.

Kernighan, B. and Pike, R., 1999. *The Practice of Programming*, Addison Wesley.

te Molder, H. and Potter, J., Eds., *Conversation and Cognition*. Cambridge University Press, 2005.

Muller, M. and Padberg, F., 2004. An Empirical Study about the Feelgood Factor in Pair Programming. In *Proceedings of the 10th International Symposium on Software Metrics*. METRICS'04. Chicago, USA: IEEE Computer Society.

Ronkko, K., 2007. Interpretation, interaction and reality construction in software engineering: An explanatory model. *Information and Software Technology*, 49, pp.682-693.

Sharp, H. and Robinson, H., 2004. An ethnographic study of XP practice. *Empirical studies of software engineering*, 9, pp.353-375.

Ting-Toomey, S. ed., 1994. *The Challenge of Facework: cross-cultural and interpersonal issues* 1st ed., Albany, New York, USA: State University of New York Press.

10 A note on the transcriptions

The examples of speech given in this paper are transcribed using the following conventions taken from (Te Molder & Potter, 2005):

- Parentheses are used to identify pauses. Short pauses are shown with (.) whilst longer ones show the time in seconds inside the parentheses as in (1.8)
- indented passages show where talk is synchronised
- [ ] indicate overlapping passages
- Capital letter show an increase in volume
- Angle brackets > < are used to show pieces of talk which are faster than normal