Knowledge and Reasoning about Code in a Large Code Base

David Martin and John Rooksby
Computing Department, Lancaster University, UK

In this paper we discuss how programmers maintain and develop code as part of a large code base. We discuss instances of how programmers reason about interdependencies in code, how programmers decide the location of the particular code to be changed in improving software functionality, and how programmers reason about what code can form the basis for a new piece of code. All three examples are of occasions where programmers, for one reason or another, have discussed with other programmers the issues to do with their code. Such talk makes reasoning, which might otherwise be done privately 'in the programmers head', observable. Through our analysis of the examples of observed reasoning we wish to draw attention to how knowledge is produced, exhibited, demonstrated and deployed in maintaining and developing the code base.

Introduction

How code should be written to be easily understood, how it should be constructed to produce elegant solutions that are easy to follow, how it should be made clean, be easily extendable and maintainable is the stuff of many books and practical and theoretical approaches. It is reasonable to believe that initially one can construct code to have an elegant structure, and also that in the early stages of projects it is possible that the programmers will have something approaching a comprehensive understanding of their code base. But as the code base grows the structure of the code becomes more complex, relationships between different areas of code proliferate and the code is no longer understandable in a comprehensive fashion, certainly at a fine level of detail. How then do we understand knowledge in relation to code – how do programmers know or find out how to successfully expand their code base to enhance their software? How do they understand problems with the code, and then remedy them? How is code realized as a social object? This paper presents an attempt to enlighten on some of these questions by reporting on an ethnomethodologically-informed ethnographic study of programmers[1].

An Ethnographic Study of Software Development

We have undertaken ethnographic fieldwork at a software company, observing work as it happens over a seven week 'iteration' of software development. We have followed this up with an in-depth interview. The company produces a 'write once, run anywhere' (w1re) development environment (in Java and C#) that can be used to develop applications (in XML) to run on mobile devices (such as mobile phones and pocket PCs). The software company has seven full-time employees, four of whom are programmers (Mark, Tom, Dale and Paul in figure 1). There is also a technical director (Shaun in figure 1), who has extensive knowledge of the software and some involvement with the work of writing it, and a 'customer relationship manager' Gordon. The programmers at the study site follow an XP (eXtreme Programming) or 'Agile' approach in developing the software. XP is one of a number of 'agile methods' that have been popularised in software development, whereby 'programming' is said to become the centre of the development work. The development work is all about programming and the project team seek to minimize the time that is spent in organising, scheduling, assessing and documenting that work. This approach runs contrary to 'traditional' software engineering where project management takes a more central role or can even appear to gain priority over programming itself. We do not seek to discuss the totality of the method in this paper, but give some brief explanations here as necessary. The method gives shape to much of the work that is done by the company, and therefore impacts upon the work of the programmers. It must be said that the XP method is implemented in different ways by different organisations, and that in any case the following of a method involves practices that are not prescribed by that method (Button and Sharrock, 1994; Suchman, 1987). There are many textbooks and websites that offer a description of XP, and also a number of empirical and

[1] See Button and Sharrock 1994;1995 for discussion of this approach

critical studies (see Chong, 2005; Chong et al., 2005; Mackenzie and Monk 2004; Sharp and Robinson, 2004). Here we will briefly discuss XP in terms of what we see as an 'organising dimension' and a 'technical dimension'.

The 'organising dimension' is the work of organising the technical work. XP has a particular form of customer focus. The customer is asked to write requirements, which are then worked on by the programming team, first of all by reformulating them into requirements that can be worked on and then by breaking them into tasks. An achievable number of these tasks are then taken up in an 'iteration' – a period of programming. The iterations at the study site lasted seven weeks, of which five were for programming and two for testing. A software 'release' is to be made after each iteration. Cards and a pin board are central technologies to this work. Programming in pairs is advocated in XP, but this was not done at the study site. Instead, as seems fairly typical, the programmers sat around a single large table, which afforded ease of interaction (figure 1).

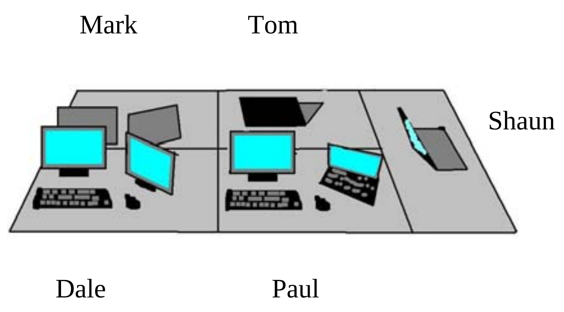

Figure 1: The Seating Arrangement

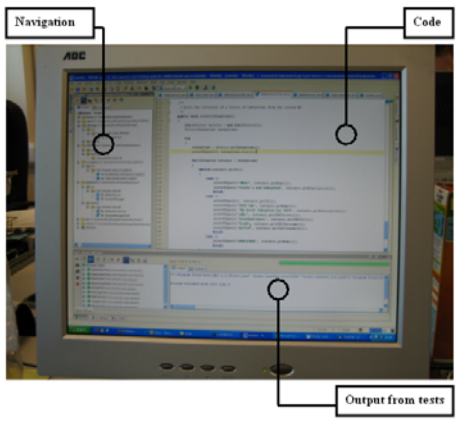

Figure 2: The IDE

The 'technical dimension', is that of the infrastructure and the work on that and on code. XP relies on a configuration of technologies to afford work, which include a CVS (Concurrent Versioning System), an IDE (Integrated Development Environment), testing software, and various other odds and ends. XP advocates a 'write the tests (in code) before the code' strategy, whereby tests give structure to what you code, and are used to regularly check the integrity of all the code when new code it added. The study site followed this at a fairly superficial level, writing a fairly minimal number of tests. A mantra of XP is that code should stand for itself and therefore should be well written, and should not be documented and should use comments only to describe why code has been written in a particular way and not to describe what the code does. 'Refactoring' is an important element of XP whereby working code is re-written so as to be more understandable by other programmers and therefore amenable for use by other bits of code, and amenable to future updates. Because the programmers were working on a development environment for other programmers, they would also write example code, and they produced documentation about how to use the development environment.

Examples

The material focused on in this paper consists of three detailed examples taken from different points in the observed iteration. The analysis seeks to draw out a number of interesting features observed in the activities of the programmers as they undertake some programming tasks as part of their day-to-day work. Of interest is the fact that actual programming clearly takes place as part of and in relation to various discussions about the programming tasks. For example, in relation to talk about why the task is required, how it is to be characterized, how it may be achieved, which areas of the code base may be implicated, how code can be re-used as part of the activity and what problems the coding may and has encountered. This analysis touches on issues such as how the coders conceive and *conceive of* programming tasks, how (and in which ways) they know their way around their code base, how they teach others and learn about their code base, how they reason about programming and how their skills are exhibited in this work.

Example 1: "What's going on with the palm stuff, what's the problem we're having?"

Our first example is re-constructed entirely from ethnographer's notes. It occurred fairly early on in the iteration. The focus of our attention is on Paul and Tom who are sitting down doing some paired programming. This is unusual at the company as although they follow an XP/Agile approach they do not systematically pair program, preferring instead to collaborate regularly in an ad hoc or occasioned manner. Paul and Tom are pair programming because Tom is a new graduate recruit being 'shown the ropes' by Paul, an experienced developer. Interestingly, however, Paul, too, is relatively new to the company. From the IDE (or IDE GUI) customer developers will be able to automatically test their XML code out on mobile phone and PPC emulators – i.e. in theory they will be able to see if their developing programs work on different mobile devices. In order to do this they are working on the code for the build scripts for the different platforms and devices they support. They will test their new coding by trying to get a piece of their demo software to work properly on the different emulators. This process is marred by problems, and it is difficult for Tom and Paul to work out just what the problem is, or how many problems they have. They try different things to try to understand what is going on and how to solve issues. Initially compiling the code causes the PC to shut down, an action which Paul repeats to the same effect:

> Paul then states: *"It's depressingly the same so it might be something we've done"*
>
> To which Tom replies: *"Put in a restart?"*
>
> Paul then says: *"Or done something to the windows kernel… oh dear not only have we broken it, we've made it explode and that's quite difficult in Java… I think it's the last lines entered* [there is an illegal exception flagged in red which Paul reads]*… no IDE instance or workspace"*

It appears that this 'illegal exception' is the 'proximal' reason for the system shutting down, but why is there no IDE instance or workspace? Paul turns and has a chat with Dale and also engages in an IMS (instant messaging system) text-chat with Shaun (the technical director who is doing training at the moment). Shaun flags up the issue that older versions of the software were written to support Palm Pilots as a separate platform. However, since new Palms run on the same platform as mobile phones, there is some essentially 'obsolete' code for the old Palm. However, this is code that they do not want to try to just remove as they think it may well have important and complex relationships with other areas of code that they need to keep. Paul and Tom's job now is to try and find a way of stopping the computer from attempting to execute the old palm code. Their initial try is to simply insert a piece of code that stops the old palm code from executing. However this then leads to the red error message 'unreachable code', to which Paul states *"Java's too clever for its own good"*. Following this Paul and Tom continue trying to work on the problem. A concern then emerges that some of their difficulties may be due to Mark working on a similar area of the code base at the same time. This leads them into a sidetrack but they eventually work out that their problem is wholly in the area they are working by comparing their code, their attempts to compile the code, and their error results with Mark's. This sidetrack, however proves serendipitous as it has drawn Mark's interest and after some trial and error he produces a *"shortcut solution"* that works.

Analysis of Example 1

This excerpt is illustrates 'knowledge, skill, reasoning and learning at work' amongst a cooperating group of programmers. Firstly, as shown here and repeated across our data, it is clear that programmers learn about the code base primarily through coding and the attendant talk that surrounds coding, dealing with errors and so forth. It is not as if a new programmer will familiarise themselves with the code base by only reading through lines of code with only the purpose of learning it. Secondly, writing code and attempting to compile it helps you learn more about the code. Errors provoke reasoning in which possible causes of errors are posited and 'thought through' by looking over pieces of code, as Tom and Paul do here when the system crashes. Importantly, programmers know that the point of failure as a line of code is often just the starting point – the symptom rather than the underlying cause – and this also motivates their attempts to solve problems. Third, knowledge is shared and distributed, but this is a means to maintain a good group understanding of the code base. Here we see that due to their problems Tom and Paul engage with other members of the team, particularly Shaun, who provides (interestingly, remotely via IMS) a reason for the failures – the computer is trying to build for an old Palm Pilot but is unable to.

Knowledge of the code as demonstrated by Shaun, is partly knowledge of the code history, how it developed over time, what idiosyncrasies, weaknesses or inconsistencies it might contain, and how these might cause problems. Of course, through their work on the code, their problems, and the help of others Paul and Tom have just learned a whole lot about the code base. The Palm Pilot example, also illustrates another interesting feature of a developing code base – why do they not just remove the code, why do they leave it in and try and 'switch it off' one way or another? The answer to this is very much to do with how a code base develops over time. Shaun tells Tom and Paul that the old Palm code may well have important relationships with other core pieces of code and therefore just extracting it may not be straightforward and may be dangerous. As code grows dealing with 'obsolete' but potentially 'crucial' code becomes an issue. A final issue to draw out here relates to Mark's involvement. Mark gets involved when it appears that some of his coding may be causing problems for Tom and Paul. A key feature of Agile/XP development is that of 'continual integration'. Small teams, close cooperation and so forth are 'designed' to move such approaches away from excessive modularity and integration problems. Part of the way by which our programmers attempt to avoid such problems is by being aware of what each other is doing and especially being aware of problems, as Mark is here. What is also interesting is that Mark, and Tom and Paul go through a 'classic' 'is it a general problem, is it a specific problem' and 'is it me, is it you'[2] problem identification search process by comparing their code, their compiling and their errors to identify the problem as one that resides with Paul and Tom's code specifically, but it also helps identify just what the problem might be and helps them find a 'shortcut' (rather than rigorous?) solution.

Example 2: "I agree but it's not just a case of just checking that"

The transcript in this example presents part of a discussion between Paul and Dale (with a single comment from Mark) over how they might proceed in the re-writing of their 'connection manager' code, to enable their push server to handle simultaneous or near simultaneous connections to many more devices then it currently supports and in a more reliable manner. This small exchange is preceded by a protracted discussion (which we only have notes for) in which Dale, Paul and Mark have been talking to Gordon (who is in charge of relationship management with customers) about performance issues with their server that they have been experiencing, and that the multiple connections problem is one that they will need to address if their plans for the scalability of their product are to be realised. They have already had some complaints from a customer about performance problems when connecting to many devices and Gordon is hoping to market their product to a large company that would have many users. A day earlier Gordon experienced problems when 'demo-ing', and it is from discussing that the 'whole problem' is worked up in conversation. The example is an interesting one in that it illustrates issues about how the programmers understand and gain knowledge of their own code base for the purposes of developing it. The problem of performance and scalability is one that has been known for a while but has not manifested regularly as they have not had customers with many users.

This example illustrates a lot of the 'preparatory' investigative work that is part and parcel of code development in this company. Their first consideration in their discussion is how serious the problem is, and furthermore when will they have to solve it. In order to gauge the seriousness of the problem they calculate hypothetical 'load' on the server by considering e.g. what would happen if 500 connections were established at once, and they realise then that they have an issue. Their next step is to consider other companies that achieve multi-threading with many simultaneous connections to a push server. This is a common technique employed by the programmers as a means of understanding how they might 'solve' a problem, and it may even provide a code resource (or template) that they can utilise. As Gordon says *"how do *** do their code? can we nick It?"* Dale and Paul then break off as Paul asks Dale to explain how the current connection to their server works by creating threads. Paul is an experienced developer but is new to the company, therefore in-and-through their discussion of how the connection problem might be solved Dale is also teaching Paul

[2] This type of approach to problem (and source) identification has been identified as especially common in situations where technical problems abound, particularly using distributed or remote collaborative technologies, e.g. Martin and O'Neill (2002); Crabtree (2004)

about how the current system works, and therefore how the code is organized, what it means and what it does. To explain to Paul, Dale and Paul have been consulting a diagram Dale has drawn. As we begin our excerpt they are still consulting the diagram as a means to understand which areas of code will need to be worked upon.

1.		*D sits with paper and pen in hand P looks towards paper*
2.	P	"it just sends a quick message to that thread"
3.	D	"Yeah but this (D points) is what?"
4.		*[4.0] D looks at P and holds it for next 4.0 with faint smile*
5.	P	"Yeah but (D looks to paper) that doesn't, that doesn't matter. You can (P points to diagram with pencil)
6.		[0.5] You can sss, That's a class you can still send (D points) from here, put a handle on you can set a
7.		global variable"
8.	D	"You can (D points with pencil) set a global variable [on it]"
9.	P	"[on that] when this does come back with the thing
10.		[0.9] you just check that we've been told to quit [0.5] we have been told to quit [a]"
11.	D	[I]Know (…) getting the results of
12.		the thread issue. Its not just a matter of (D focuses pencil to paper) changing garbage collect's connection.
13.		There are all sorts of thread groups such as (…)"
14.		*D gazes at P, D lowers paper and P turns to computer screen simultaneously*
15.	P	"yeah well there are but we change this so when this is working in threads as well"
16.		[3.0]
17.	D	"Yeah [0.5] well yeah but I agree but its not just a case of just checking that"
18.	P	"no, no-no (…)"
19.	D	"[(…) [1.0] thread safe [1.0]"
20.	M	"On server app?"
21.	P	"Yes [7.0]" (P flicks between getDatabaseConnection() and getConnection() using a reserved keystroke)
22.	P	"erm"
23.		[15.0]
24.		*P takes deep breath*
25.		[11.0]
26.	P	Tck tck-tck (*sniffs*)
27.		*P sits back*

At the start of the example we can see that Dale and Paul are moving from what might be termed a more 'abstract' discussion of threading to a more 'concrete' discussion of just how multiple connections might be realised in code within the code base. Of interest are the ways in which Dale and Paul begin to scope out how the code might be written and where, and where they agree and disagree. With reference to the diagram Paul suggests that they need to set a 'global variable' which is basically accepted by Dale (lines 5-7). However, as shown by Dale's comments in lines 10-12 and line 16, while he agrees with Paul's conception of some of the problem, he also believes that Paul does not have a full grasp of all that needs to be coded as there are multiple types of thread groups (i.e. related to particular processes, whereas 'garbage collection' probably runs as a default). This means that Dale is essentially telling Paul that the problem is to be solved with code of some other location or combination of locations.

Analysis of Example 2

This provides another rich example of 'knowledge, skill, reasoning and learning in action'. Firstly, we see how an issue with the operation of the company's system develops to finish with an examination of where and how the code base might be worked on or added to. The problem with the code is already known – they have not instantiated a means to multi-thread connections to their server – but it is made particularly prescient by a confluence of events; their customer base is growing, an individual customer wants to develop applications for 'enterprise wide' implementation, they have had previous customer problems and today Gordon could not demo successfully. With some ad hoc calculations the team attempt to conjecture the magnitude of the problem and decide they need to start thinking about a solution. This exhibits some the practical ways in which problems manifest themselves as 'something we need to do something about now or soon', how known issues bubble up into specific requirements. We also see the practical ways in which they are measured via informal calculation – 'if the result suggests a serious problem we take it seriously'. Consequently, the talk of the programmers moves towards looking for a solution. This initially focuses on where they might find another company that has already 'solved' this problem and whether they can learn how they did so, and even borrow (or steal!) code as a template solution. The work then progresses by examining, diagrammatically, the current threading model. The diagram is then related to actual areas of code to discuss how and where code would have to be modified and added to achieve multiple connections.

This example illustrates some common features of coding work we have observed. Firstly, it is common for the team to come together on an issue and discuss it even if particular individuals go off to work on it (but this work is often still available to others through their close proximity as a group). This is part of the process of maintaining shared knowledge of their system and their code base. Secondly, it illustrates the propensity towards re-use and economy in finding solutions rather than working out a solution from scratch – if someone else has already solved the problem you can learn and even borrow from them. Thirdly, it illustrates how the understanding of your code (and what to do with it) is elaborated in relation to models, diagrams, other code and conversation in the flow of on-going practically oriented work. Finally, it also illustrates how knowledge is produced and learning occurs as an unremarkable part of working on and with the code base. Paul and Dale work together cooperatively on the problem, Dale as an experienced developer who has knowledge of issues to do with threading, but also with a good knowledge of this system, of this code base. Paul, on the other hand, has good knowledge of the former but not so much the latter. In their talk we see them trying to work up a solution together but we can also witness the ways in which Dale is guiding and directing Paul around the system to see the other areas of the code that may be implicated in the threading issue. It illustrates how knowledge of the code base is knowledge of your way around it, how things might be connected, and what the implications of changing a piece of code may be and is fundamentally based on experiences in the mundane work of doing coding. Dale is demonstrating his knowledge and 'teaching' Paul implicitly in their on-going work on the multi-threading task.

Example 3: "It's only a noddy server but it's worked pretty damn well for uh good couple of years"

Our third example is drawn entirely from videotape, however here we just focus on a transcription of the talk going on in part of it. This material is taken from the 'testing' phase at the end of the development iteration. The testing phase involves some performance testing, some continued coding and some testing of the code that has been added to the code base. As a basic test of whether the code they have added works, to ensure that programs written in the XML development environment are write once run anywhere, they write small applications in XML on their workstations and these are tried out on various emulators and 'pushed' to actual devices.

When we join our transcript Paul is in the middle of writing and testing just such a trial program, one that will provide a means by which web services can be accessed and utilised by mobile devices. Web services are small web applications which, for example, would allow you to authenticate from a mobile device and then retrieve records – i.e. they allow the exchange of data over the web. It is envisaged (and has been requested) that customers will want to utilise web services as the means of exchanging data between their back office systems and mobile devices deployed in the field. To test whether this will be possible Paul is trying to write a program to successfully access demonstration web services (www.salesforce.com, see line 67) through a mobile phone. If he is successful it will essentially be taken as proof of concept that this is possible. However, he has hit a hurdle. He is not managing to successfully access the demonstration services – when he sends a request he does not get the desired response (e.g. a login screen appearing on his mobile 'test' phone). This presents a problem as he would like to understand the nature of the 'traffic' or dataflow between the phone and the web service as a means of diagnosing and remedying the problem, however, he cannot think of a way to capture this dataflow (this is difficult with mobile phones; his usual approach to such a problem uses telnet, but this is not available).

When we join the transcript Mark is suggesting a possible solution – one in which Paul can take a small program from another area of their code base and use it to 'sit between' the mobile phone and the web service and capture the dataflow, by taking an image of it, or recording it in some way. This is a clear example of instruction from Mark. Firstly he is telling Paul that there is code in the 'signature capture demo'[3] that he can use within his code to record the dataflow between the mobile phone and the web service (re-use or template use of your own code is a major feature of software development for our company). Secondly he

[3] Signature capture is considered a classic demo application by the team – e.g. so someone can sign for deliveries on a mobile device using a stylus as a record of receipt.

is guiding Paul through the file structure (1-22) to locate the code he requires – again showing that knowledge of the code base is in part about knowledge of how to find your way about it, how to locate sections of code and so forth, which is also about familiarity with the organisation of the file system. The application he is directing Paul to is the ‘socket image reader’ (14) – a small application that allows you to read and view the dataflow. Mark also instructs Paul as to how the code might be used – ‘just add it into your source’ – suggesting it can be used without much re-writing (21-22). He also comments on the nature of the code ‘it’s only a noddy server…’ (23) i.e. it works like a (proxy) server[4] and it is simple and straightforward rather than elegant or sophisticated but it works fine.

In the following lines Mark goes on to elaborate how the socket image reader works on the signature capture demo (25-27) – it sits between the device and the server, captures the signature image being transmitted and then ‘sucks it’ and displays it on a page on screen on the workstation. He then relates it to how it might be used to solve Paul’s problem (29-35) – just ‘whack it around’ (i.e. rework the code) so it can work as a proxy server to capture the data flow and then display this on Paul’s workstation. In the rest of the transcript, during which Paul is working with the code for the socket image reader, and attempting to work it into the code for the application to access the web services, we see Mark, Dale, Paul and Tom trying to work out just how the code for the image reader should be deployed. This involves working out how the connection between the mobile phone and the web services works and working out how the socket image reader would be deployed in this connection, and consequently Paul is trying to work out how this would be realised in code. As we leave the transcript the work is on-going, and continues for some time.

1. P hhhhh-God-er
2. M yeah
3. M yeah, in err QX demos (p) QX demos project [p2 speaking] this in the signature capture ** sub project
4. P nyeah
5. M and in there theres an image reader
6. P [er
7. M [and theres a call [
8. P [where?
9. M where are we at? MidP MidP2?
10. P say that again, *** QX demos. (p) oh right
11. M in there there’s an image reader
12. P right
13. M and in there there’s a source called Q net, and in there there’s a socket image reader
14. P ***
15. M Tiny little, program, it won’t take much modifying to do whatever you want to do with whatever you want
16. to read.
17. P erm, sorry I just, I just can’t, I’m in QX demo
18. M in there theres a sub directory called signature capture demo
19. P Signature capture demo, and the source for the directory in there
20. M If you, you y’know, open the source for the QX demos project you just add that into your source.
21. M Its only a noddy, server but its (P OK) worked pretty damn well for uh good couple of years.
22. P oh right, so
23. M retrieving byte steams for uh, images and, basically if you take a photo on one of these, on the signature,
24. the signature (P yeah) capture demo and say send photograph it sends it to that sucks it in and shows it on
25. a page so it, it definitely works
26. P so, OK
27. M Just whack it round to do whatever you want
28. P OK
29. M But I presume if you (P) what, what would you be doing? Sending off your request to a different port
30. number on the server and just using, using one of these to just suck in what came down through the socket.
31. P Yeah
32. M That is, I mean. That is really all it
33. D **** Will be passed to send it off.
34. M S**
35. D * that back on so

[4] A proxy server works as a temporary cache between device and web service

36.	M	Right so you're setting like a proxy are you?
37.	P	Yeah
38.	M	To sit, between, the two, so you, could read what's coming around.
39.	D	It's a bound socket isn't it
40.	P	What
41.	D	… a bound socket between the device and the, server
42.	P	Yeah, Yeah …
43.	D	I don't think
44.		(Pause)
45.	M	Oh yeah in that its uh
46.		(Pause)
47.	D	Connection (m state ??) or state
48.	M	yeah
49.	D	Yeah
50.	M	You just get a, request and response, there no state between them to (Pause) communication so
51.	P	Tha, tha doesn't
52.	M	I don't think that matters because you're only hitting the server, once, and then, you you just putting
53.		something in between it aren't you. That this.
54.	P	Right sorry you mean how do I get the response
55.	D	MMM
56.	P	If I get a socket
57.	D	Yeah
58.	T	*eh
59.	P	What?
60.	T	**want the same socket
61.	P	I'm just coming back to it
62.	P	**[
63.	T	[Your just going to have to keep, have to keep, keep the socket open from the phone
64.	P	yeah
65.	T	and , one two salesforce seven
66.	P	Yeah
67.	M	and come back down that one and throw it back here
68.	P	yeah
69.	M	It could have been kept open for the phone (P yeah) I would have thought
70.	D	I thought, I [thought
71.	M	[its worth experimenting
72.	D	I thought there was a bit more to it, is, is there not

Analysis of Example 3

There are a number of interesting features about this example that build upon our previous analysis. Firstly, the example gives us an insight into the development process and testing regimen of the company. One of the new features on this iteration of the platform and development environment is that customers should now be able to develop programs by which they can deploy to mobile devices using web services to exchange data with their backroom systems. The team do not know just how any customer would like to do this in any particular fashion so test the possibility 'hypothetically' (and generically?) by trying to write a program to communicate between various mobile devices and demonstration web services. In our observations this was shown to be difficult and complicated (as shown here) but possible. It is interesting that the programmers do not observably take this to indicate a failure of their work in the design and programming of their development environment but rather that the design basically works. That the developer working for a customer might have to spend considerable time and encounter a number of problems developing an application appears to be a taken for granted feature of work – these are routine features of the job of coding and development.

As already noted the example also highlights, again, features indicating what knowledge of the code base might consist in and how this knowledge is passed on within the practical business of coding, of doing development. Mark suggests that Paul should be able to reconfigure a small application from the signature capture demo (i.e. used to sell the system to potential customers) to solve a seemingly quite different type of issue within testing. We might consider this to be a creative or clever form of code re-use; if it works for capturing and displaying signatures, it could be made to work for capturing and displaying data flows. That the whole episode is treated by participants as unremarkable rather than as a spectacular discovery would suggest that such creative re-use or 'lateral' thinking is part and parcel of the mundane work of programmers. Within making his suggestion Mark also guides Paul through the file structure to where it may be located (and therefore through some of the structure of the code) and provides other information about the application - its 'status' as an application and some of its history (it's a noddy server, but has worked well for two years). All sorts of things are therefore being communicated about the code base, how it is organized, the status and history of particular bits of code, how they may be re-used so forth.

Other interesting insights that the example provides us with relate to the process of programming, testing, and the process of code re-use within this. This example only arises out of a difficulty in testing. If the application for accessing web services had worked straight off there would have been no need to try and work out what went wrong, to try to work out how to capture the dataflow between mobile phone and web service. However, from our observations this is precisely the type of thing that happens in programming for our coders. Problems are encountered in the 'task' in hand and this stimulates all sorts of 'branched' work to try and work out what the problem is (or problems are) and how they may be solved. In our example here the failure of the application has lead to an attempt to try and understand why the failure has happened (initially by inspecting the code), which has lead to Paul wanting to capture the dataflow, which involves re-using another application and working out how to code it to work for his purposes. Incidentally, we can see how the whole group of programmers get involved in 'thinking-through' how the connection between device and web service works, how the proxy server/socket image capture may be deployed within this connection and how this might be coded. This involves thinking about the 'physical' relationships between devices, how they connect, how the process of communication works and how this might be coded – in a way they are thinking (i.e. talking through, reasoning through talk) from a multiplicity of perspectives to work up how to code. Of course, this does not have the nature of working everything out in advance – it more has the nature of thinking about possibilities and then 'its worth experimenting' as Mark says (73). The experimenting helps to discard unlikely possibilities and refine the problem space.

Discussion

In this paper we wish to draw attention to a number of features about how knowledge is produced, exhibited, demonstrated and deployed in relation to the practical activities involved in maintaining and developing a code base. As we have seen through our examples knowledge is thoroughly practical in nature – it has been and is produced through a practical engagement with code and a code base over time. By practical engagement we mean that programmers gain knowledge of the code base by interacting with it through their business of work – writing, 'experimenting', maintaining, developing, troubleshooting, testing and so forth. As the code base develops over time and in size, and since the coding is a distributed task, so individual, and even group knowledge of the code base becomes less comprehensive, less systematic. But was it ever so? This is an interesting question. Certainly Mark and Dale, in our study (and maybe Shaun) have a 'strong' or 'good' knowledge of the code base. But what does this knowledge consist in? Knowledge is certainly not knowing what every bit of code is and does and how it might relate to every other (or even just some selection of other pieces of code), certainly not a systematic, detailed, comprehensive and complete knowledge of the code base and its 'structure'. Instead, they have a good knowledge of the history and the status of the code, e.g. what is good, stable, elegant, unproblematic, problematic, inconsistent, incongruous, 'noddy' etc. This helps them to identify where problems might lie, how code is *relevantly* connected to other code in terms of what we are trying to program now, or where something might be found to re-use (and in what way it might serve as a template). They have worked by finding their way around the code for doing coding and all its attendant activities and much of their knowledge is in their ability to find their way around the code base and 'see' the relevant connections between different parts of the code base.

Knowledge turns us to learning and it is clear from our examples how learning proceeds in such a setting. It is gained from the practical experience of doing coding as part and parcel of development. We see this in everything Paul is doing. He is not doing work at one point and learning at another. He does both at once, and his problems in achieving his programming tasks give ample room for others to guide him in learning different things out about the code base, and for him to branch off into side projects, and even dead ends that teach him much along the way. It is not as if learning is of a clearly different order for Dale and Mark, it is just that they can specifically help Paul to gain knowledge about the code base in a way that he currently cannot do for them. What is skill in relation to knowledge and reasoning in relation to knowledge? Much of this can be understood as simply putting the knowledge to work, using it to reason about problems as they arise, working out why something might be causing something based on the evidence (see the Palm Pilot example), being able to experiment in ways that will help you find a solution, or skillfully, artfully and inventively re-using of a piece of code (the noddy server example). It will manifest in many ways, in the mundane methods and practices employed by the programmers, some of which we have begun to explicate.

Conclusion

Now on the theme of this workshop we might now like to ask where this paper takes us in inquiring as to how ethnographies of code might sit in respective sociological and computing disciplines. It is easier to answer the question in relation to the former rather than the latter subject area. Sociologically we can say that studies of programming, particularly field studies that attempt to explicate mundane reasoning and practice as a part of working specific programming problems in commercial settings are thin on the ground (Button and Sharrock, 1995, alone appears to get closest to achieving this). Studies tend to look at organising and coordinating activities (or more abstract notions like programming 'culture') but as something disembodied from the actual coding tasks being worked on. As such we hope to have attempted a start at redressing this omission. Sociologically it is interesting simply to try to understand more about the lived work of programmers. It is worth noting that 'unique adequacy' (Garfinkel and Wieder, 1992) is an issue here, as because programming is so specialised and diverse, and that in order to understand a new code base even an experienced programmer needs quite a lot of 'familiarisation' getting a clear understanding of what coders are doing, how and why, even at an 'acceptable' level of gloss is difficult.

Turning to the question of computing – we might first answer that sociological studies do hold some computing interest. However, usually such studies are expected to have some enlightening and improving relationship to computing – e.g. helping to understand how computing (i.e. programming) work could be better organized, or better supported (by technologies, techniques and practices), or as a means to generate requirements for technologies used within that work or for the technology being designed by those workers. In our study we are not in a position to produce any of these 'results' as yet but we would argue that sociological studies have not really done this successfully in relation to programming itself, but that if such studies are really going to have an influence they really need to understand programming at a decent level of depth, and in a manner that explicates how programmers observably reason and 'think' in relation to actual programming work.

References

Button, G., Sharrock, W. Occasioned Practices in the Work of Software Engineers. In Jirotka, M., and Goguen, A. (eds.) Requirements Engineering. Social and Technical Issues. Academic Press, London, 1994.

Button, G., Sharrock, W. The Mundane Work of Writing and Reading Computer Programs. In Have, P. ten, and Psathas G. (eds.) Situated Order. Studies in the Social Organization of Talk and Embodied Activities. University Press of America, Boston, 1995.

Chong, J. Social Behaviors on XP and Non-XP Teams: A Comparative Study. In Proc. Agile United Conference (2005).

Chong, J., Plummer, R., Leifer, L., Klemmer, S.R., Eris, O., Toye, G. Pair Programming: When and Why it Works. In Proc. The 17th Workshop of the Psychology of Programming Interest Group PPIG17, (2005), 43-48.

Crabtree, A. (2004) Technomethodology. Paper presented at the Proceedings of the 6thInternational Conference on Social Science Methodology, August 17-20, Amsterdam: International Sociological Association.

Garfinkel, H., D.L. Wieder (1992) "Two incommensurable, asymmetrically alternate technologies of social analysis", In: G. Watson, R.M. Seiler, eds. Text in context: studies in ethnomethodology, Newbury Park. Sage: 175-206

Mackenzie, A. Monk, S. From Cards to Code: How Extreme Programming Re-Embodies Programming as a Collective Practice. JCSCW 13,1 (2004), 91-117.

Martin, D. and O'Neill, J., (2002), "Practically Accomplishing Participation", XV World Congress of Sociology, Brisbane, Australia, July 2002.

Sharp, H., Robinson, H., An Ethnographic Study of XP Practice. Empirical Software Engineering 9 (2004), 353-375.

Suchman, L.A. Plans and Situated Actions. The Problem of Human Machine Communication. Cambridge University Press, Cambridge, 1987.