Reading as Part of Computer Programming. An Ethnomethodological Enquiry

John Rooksby, David Martin and Mark Rouncefield

Department of Computing, Lancaster University

rooksby@comp.lancs.ac.uk, d.b.martin@lancaster.ac.uk, m.rouncefield@lancaster.ac.uk

Abstract. This paper examines reading as done by programmers engaged in software development. Reading is an activity we feel should be of fundamental interest to studies of programming, but the practical achievement of which has not been closely examined. We give examples of programmers reading in pairs, and reading alone, and show reading in both cases to be explainable in terms of shared social practices. These practices are not determined by the code but nor are they purely socially constructed; rather they lie in the linkage between the code and programmers' ways of reading the code. We discuss (1) how features of day-to-day coding work create pertinent occasions for reading a certain piece of code, (2) how programmers order and expect there to be an order to code, and (3) how programmers have ways of analysing code in order to make sense of it. This is an ethnomethodological study that draws from ethnographic fieldwork at a professional software development company.

Introduction

"Reading is neither in a text nor in the reader. It consists of social phenomena, known through the achievements which lie between the text and the readers eye, in the reader's implementation of society's ways of reading, in reading what a text says." Livingston [17, p16]

In this paper we argue that reading is social, practical and situationally grounded. We will demonstrate how: (1) some reading is obviously collaborative, (2) reading draws upon shared practices, and (3) even 'reading to oneself' is socially organized and oriented, and often publicly available. By this we are not arguing that people never 'read to themselves' silently or *sotto voce*, but rather that reading to oneself is undertaken using socially learned and shared practices, and is in many cases observable as the activity and practices of reading. As Livingston states in the quote above, reading is not to be located in the reader or the text but in the social practices the reader is able to apply to the text. We seek to clarify how programmers and programming technologies together implement programming's ways of reading. We have undertaken this study of reading as we see reading to be a fundamental aspect of programming, and hope that this research can help us in moving towards a better understanding of programming tasks or actions that involve reading (eg. testing, debugging, copy and pasting) and support in both the evaluation and development of methods and technologies associated with these. A general aim is also to show that social analysis – in our case ethnomethodologically informed ethnography – can provide a suitable means for studying the practices, methods and technologies of programming. Our approach is also interested in explicating those aspects of 'thinking', 'cognition', and 'reasoning' that are exhibited in talk and interaction – 'in the wild' (cf. Hutchins [13]) so to speak – and thus takes up a program of research that seeks to elucidate practices, methods and cognitive phenomena through empirical study of social action and interaction. We feel that this work can complement previous studies published in the Psychology of Programming arena (for example we cover similar territory to Green [10] Sharp [25] and others) but we take an approach to uncovering how programmers make sense of and get on with their work at hand, that only uses materials gained from observation and recording of 'real-life, real-world' work. Our line of enquiry is probably most comparable to that of Booth [2], Downey [9] or Button and Sharrock [4] in computing, but may be most recognisable to those from a psychology background through its similarities to the work of Hutchins (e.g. [15]; and Alač and Hutchins [1]).

Ethnography, Ethnomethodology, and the Ordinary Skills of Programming

This paper draws heavily upon writings by Eric Livingston [17][18][19]. Livingston's studies of mathematics are mentioned by Sharp in her paper 'An Ethnographic Study of XP Practice' [25]. Sharp notices a similarity between studies of mathematics and of programming, but does not engage with the ethnomethodological aspects of Livingston's research. Sharp and Livingston are in-fact doing very different things. Despite similarities in title, ethnography and ethnomethodology are separate. We use the term ethnography to refer to the gathering of data about people, in particular observational data about programmers at work. In contrast, 'Ethnomethodology' is an 'analytic orientation'; it pays close attention to the methods by which people (in this case programmers) construct and coordinate action and interaction, and assign meaning to, reason about and come to shared understandings about actions, interaction, texts and artefacts. As ethnomethodologists we are trying to reveal what is implicit and unremarkable for programmers in and about the methods they employ when they are reading computer code. Ethnomethodological studies often draw from ethnographic data, and as such our work is similar to Sharp's in that we have done an ethnographic study of practice. However our work diverges in that ours is what can be referred to as ethnomethodologically informed ethnography. Unlike Sharps' broad 'cultural' study we try to get towards a more detailed analysis of interaction and work with technology. We do not deny the value of Sharp's work, but wish to draw attention to differences in analysis, interests and outcomes of studies of very similar materials.

Sharp's work aside, there are already a number of ethnographic and a few ethnomethodological studies of the work of writing software (e.g. [3][4][5][6][7][12][14][15][16][20][21][23]). These predominantly focus on the 'cultures' of programming and the forms of collaborative work undertaken by programmers when organising their coding activities. These studies have drawn attention to how different types of development work are organised, and involve different collaborative arrangements and collaborative practices. In general these usually make claims about the 'situated' (i.e. contingent and resourceful) achievement of work, and follow a mode of analysis that concentrates on how awareness is shared, what the forms of coordination are, and how plans and procedures are produced and made to work (e.g. as exemplified in the ethnographic work of Suchman [26]). Most of these studies make useful contributions, but (with notable exceptions [3][5][23]) tend to describe collaborative practices without reference to how they are specifically related to the production of - and work carried out on - specific pieces of code. In this study we wish to do this, and to exemplify how this can be done.

We find that Livingston's work is particularly useful for aiding an understanding of the technical work of programmers. Livingston is an ethnomethodologist, and is best known for his work on mathematical proofs [18]. The work of mathematics has some resemblance to the work of writing software (however we must be cautious because as Greiffenhagen [11] rightly points out, Livingston's findings are about 'proofs from a readers point of view'; we are interested in code from a coder's point of view). Livingston has also written a book on reading literature [17], and in it writes briefly on programming. In this following quote, Livingston talks about the ordinariness of skills to the programmer.

There are practical, laic skills of knowing how to program a computer and of knowing how a computer, in terms of programming, works. To the novice programmer, these may hardly seem to be laic sills, but they are for competent programmers, particularly when compared to the theoretical analyses of those skills that are found in mathematical logic and advanced computer science.
[17, p122]

Livingston describes here that the skills of programmers have a 'laic' ordinariness about them. By 'laic' he is referring to the skills shared amongst programmers – the skills programmers expect other programmers to have (e.g. the knowledge about what makes code 'readable' or the expectations about how a professional would structure code.) He acknowledges that to the novice, these skills are not ordinary but must be learned. Livingston then makes contrast between the ordinary skills of programming and the theory driven analysis of a program. To understand programming, the ordinary skills of programming are of interest, not theory, and a theory of programming is unnecessary. Ethnomethodologists are dismissive of the utility of theoretical explanations of social phenomena, in that they believe these explanations gloss the actual endogenous ways in which social phenomena are realized. As Watson [28] states "*this is why so many representatives of this form of analysis utilise audio and video-recordings of scenes of naturally-occurring action and interaction. This form of data imposes certain constraints on the analyst, e.g. to eschew prior theory-formed or theory-driven characterisations of the phenomenon under observation and to focus instead upon participants' own displayed understandings of such phenomena and upon their cultural procedures for achieving such displays*" (p207-8). Theory is very useful in science, but we feel that to take a theory driven or scientific approach to social phenomena is counterproductive. We are not opposed to the use of theory, for it has allowed the successful design of interactive systems. We claim however that a general theory of reading will miss some of the basic principles of reading as it happens in real situations. In programming, the mathematical analysis of an algorithm can be said to invoke theory of an algorithm that the average programmer will seldom use, or ever have reason for recourse to. Theory could only be seen to describe the everyday work of programming in situations that it appears in the everyday work and talk of programmers. Even in these 'special' or rare occasions it would only be part of and used in conjunction with the predominant 'laic' practices of programming. Instead theory has a place of value in separate activities such as the analysis of the efficiency and dependability of software.

Our second quote from Livingston refers to what he elsewhere [18] might describe as the 'lived work' of programming.

In writing an actual algorithm or "program", the programmer organizes these laic skills of programming; a "bubble sort" program organizes the skills of programming as a "bubble sort" procedure. [17, p122]

This statement suggests that the ordinary skills of programming are used to produce programs, but claims that there is no separation between the skill of writing the program and of the written program itself. The program is the emergent assembly of the skills involved in its production. By this we do not mean that code contains a step-by-step, blow-by-blow account of programming and design decisions, however, it reveals the methods that have been used to solve problems, realise operations and so on, and it exhibits a structure irrespective of the perceived 'messiness' or 'elegance' of that structure. The code likewise exhibits possibilities for continued or future work to proceed on it. This "lived-work" as Livingston might call it, has an intrinsic presence throughout any program [19]. The program has a 'logic', but more specifically is has a written and read logic that as a text is inseparable from the work of reading and writing (and from other features of programming work including practices of 'compiling'). Reading, writing and other programming work is as much an affordance of code, as code is a production of programming work. In a continuation of these ideas, Livingston writes:

The achievement of the program, however the algorithm is written, is that it is an algorithm, that it does what it does correctly in an exhibited mechanical fashion. [17, p123]

A program can be written in many different ways, for example there are many different methods to sort a list, but the program can be said to be a program when it does what a program does, and what this program needs to do. The working program is a collection of code that is demonstrably a working program; it compiles and can be executed. A program is an account of itself, but note that the compiled (and working) program stands in relation to the code. By compiling, code is changed into instructions for the machine. The programmer can see the compiled program to be working or not, and if it is not working, the programmer will make some comparison between what the program does, and what the code instructs the computer to do; the error in the program does not simply appear in the code but must be looked for. The situation is slightly different for interpreted and compiled programming languages, but the fact remains that code is not as it is often said to be "instructions and descriptions"; it is not two things at the same time but is workable and transformable. The code and the program are done in relation to each other but are never the same thing.

Fieldwork

We have undertaken ethnographic fieldwork at a software company, observing work as it happens over a seven week 'iteration' of software development. We followed this up with an in-depth interview one week after the iteration, and then conducted one further week of fieldwork nine months later. The company produces a 'write once, run everywhere' 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. There is also a technical director, who has extensive knowledge of the software and some involvement with the work of writing it. The programmers at the study site follow an XP (eXtreme Programming) 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 technical work, and the focus of the work of organising the technical work. The method gives shape to much of the work that is done by the company, and therefore impacts upon the reading-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 [4][26]. There are many textbooks and websites that offer a description of XP, and also a number of empirical and critical studies (see [6][20][25]).

In our study of the programmers we observed many situations in which they were required to read. There is not room to describe all of these in full detail but here we can provide an informal taxonomy of some of these before going on to focus on three illustrative instances in detail. Most of this reading took place at the screen or in conjunction with doing something at the screen. The situations were (1) When writing code (2) When debugging (3) When writing tests (4) Searching for information on the internet (5) Reading emailed information (6) Reading from textbooks (7) When using the cards (8) When discussing models (9) Documenting (10) Sharing information on whiteboards. These ten 'occasions' of reading were all similarly situated in the physical space of the company, many using largely similar computer based tools, and each done within the contingencies of getting work done. Each however still had its own peculiarities according to the specific task at hand, the specific situation that task occurred in and the specific tools in use. We will give three examples of reading. The first of these is intended to be a simple example. The second is a lengthier example allowing us to build upon the first. The first and second examples detail people reading together to get work done. The third example is of a programmer working in isolation and is used to demonstrate that what can be said of reading in pairs holds true for reading alone. What we come to is a framework for understanding reading which states reading is 'occasioned' (i.e. it is provoked by specific circumstances), and that by and for reading, text is 'analysable', and 'orderly'.

Example 1

This first example is a short vignette taken from our notes. For these we wrote down what the workers were saying and what they were doing but were unable to record the specific code that was being worked on.

P and D are working together on some code. P has run the tests, one of which has failed. P clicks on the hyperlink created in the test window as part of a message when the test failed, which brings to the screen the code where the test failed.

D (Reads aloud) "If node has children"

P "I just cast to string stringL"

Reading is central to the work referred to by this short vignette. Talk is used for the coordination of reading-work so as to solve the problem of why the code fails. The reading work is mundane and is 'obvious' enough to each programmer so as to be unremarkable. Analysis of this talk for the purposes of reading allows us to discuss reading. P has evidently read that the code has failed, and needs not tell this to D or confirm that D has also read this. It is obvious that the code has failed, but only obvious in that they know where to read for messages when compiling (in the bottom window of the IDE). P understands that D is a competent programmer and will be doing the same reading-work as he. The hyperlink generated as part of the error message is intelligible to both people and it is obvious that that hyperlink can be and should be followed. It is then obvious that they read the code that the hyperlink brings them to. This line of code that they are brought to is highlighted in pale yellow. The reading aloud of the code by D in line 1 is necessary because it is not obvious to D what the code does. We could interpret D's reading aloud in several ways, but it is P's interpretation of D's utterance that allows this conclusion. D's reading aloud of the code is understood by P to mean that D is attempting to understand the code. P's response is to state what is done if, as D reads, the node has children. This response focuses the reading on another part of the code, which follows the logical order of the code, and does so in a way that explains logical order in terms of how the code was written. By qualifying it as 'just', D appears to think casting to `String StingL` has some simplicity. We could follow the conversation further to see how work continues. However, we use just a few lines of description and two utterances here to demonstrate the amount of reading work being done, its centrality, and the extent to which reading work is a taken-for-granted skill for workers at the study site.

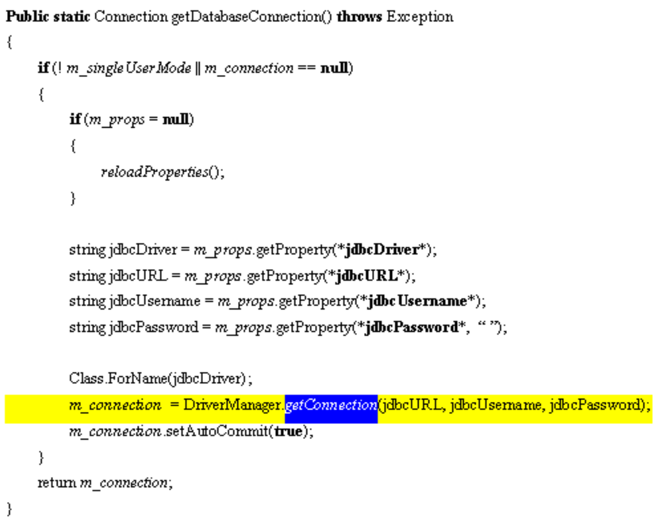
Example 2

This second, longer example is taken from a video. The example is made up of a transcript (below) and a copy of some of the code discussed (figure 1). In this example P and D are discussing an issue to do with multi-threading, which is the running of multiple processes at one time. The debate, as with most at the study site has the character of 'trying-to-work-it-out' rather than of a dispute. We do not discuss the actual issue in debate, and note that the debate is left unresolved. What we are interested in here is how reading is used in and for the purposes of the debate.

D sits with paper and pen in hand P looks towards paper
P:"it just sends a quick message to that thread"
D:"Yeah but this (D points) is what?"
[4.0] D looks at P with a faint smile and holds it over 4 seconds
P:"Yeah but (D looks to paper) that doesn't, that doesn't matter. You can (P points to diagram with pencil) [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 global variable"
D:"You can (D points with pencil) set a global variable [on it]"
P:"[on that] when this does come back with the thing [0.9] you just check that we've been told to quit [0.5] we have been told to quit [a]"
D: [I]Know (…) getting the results of the thread issue. Its not just a matter of (D focuses pencil to paper) changing garbage collect's connection. There are all sorts of thread groups such as (…)"
D gazes at P, D lowers paper and P turns to computer screen simultaneously
P:"yeah well there are but we change this so when this is working in threads as well"
[3.0]
D: "Yeah [0.5] well yeah but I agree but its not just a case of just checking that"
P: "no, no-no (…)"
D: "[(…) [1.0] thread safe [1.0]"
M: "On server app?"
P: "Yes [7.0]" (P flicks between `getDatabaseConnection()` *and* `getConnection()` *using a reserved keystroke)*
P: "erm"
[15.0]
P:(deep breath)
[11.0]
P: tck tck-tck (sniffs)
P sits back
M starts conversation about another issue, P starts reading email.

The transcript recounts a dispute between P and D over how to proceed in the (re)writing of code. The two parties in the debate have differing opinions on both how and where in the code, new code is to be written to solve an agreed problem. The debate involves a variety of activities including various formulations in talk, the use of a drawn model, gesturing, and reading.

The talk at the beginning of the transcript relates to gesturing at a diagram on paper, which has already been drawn and read as a part of this debate. The gesturing is done to focus the debate on a part of the model and so a part of the corresponding code. This gesturing demands a reading of what is gestured to, but a form of reading that does not involve the actual reading of anything new or unfamiliar to that reader. The gestured reading is an interactional resource in the holding of the debate. Attention is shifted to the screen part way through the debate, relating reading of the diagram to a reading of the code. It is not actually clear to us whether P shifts D's attention to the screen, or whether the holding of the debate has naturally led back to the screen. However we do see a reorientation by P and D to the screen, both physically, and in their conversation. When reorienting to the screen, P makes two uses of the word “this”. “This” refers directly to some thing, and this is achieved by gesture. The first this is ambiguous to us (and is possibly also so to D) in what the ‘this-gesture’ actually refers to: it may be to a certain line of code on the screen or to a specific section of code, or perhaps simply to the screen itself. It can definitely be said however that the this-gesture is a cue to read something on screen. The second “this” is combined with the highlighting of `getConnection` by the action of a mouse click. This second this-gesture orients reading to a specific place. Again this is not reading to gain new information, but is reading to orient the conversation to a particular place in the code.



```
Public static Connection getDatabaseConnection() throws Exception
{
    if (! m_singleUserMode || m_connection == null)
    {
        if (m_props = null)
        {
            reloadProperties();
        }

        string jdbcDriver = m_props.getProperty(*jdbcDriver*);
        string jdbcURL = m_props.getProperty(*jdbcURL*);
        string jdbcUsername = m_props.getProperty(*jdbcUsername*);
        string jdbcPassword = m_props.getProperty(*jdbcPassword*, “ ”);

        Class.ForName(jdbcDriver);
        m_connection = DriverManager.getConnection(jdbcURL, jdbcUsername, jdbcPassword);
        m_connection.setAutoCommit(true);
    }
    return m_connection;
}
```

Fig. 1. Program Code.

In this collaborative reading (and ultimately writing) of code we observe ‘premising’ [27] work – work to bring parts of text to one another’s attention for particular purposes. Premising work is directed for the other person to recognise why the text being shared; why the shared text is significant and why it is relevant for solving the task in hand. That it is undertaken without specifically ‘spelling everything out’ indicates that interactants proceed (at least initially) assuming that one another will understand why something is being pointed at in this way. As a feature of such premising work, highlighting fine-tunes what is intended, relevant, and significant and effectively pre-motivates the reading, making it apparent, for example, that the intended object, the text, is not so much the whole screen as just a certain entry on it. But we can also see how co-presence of the interactants is central, since part of the premising resides in the way the sharer physically points at the text shared, whether with a casual waving at the screen, or pouncing on a line and stabbing at it. The sharer can manipulate the ways in which the object becomes accountable. A key feature of premising work is that it is not only directed at sharing an understanding but it is also directed at witnessing that shared understanding. The sharer can see the recipient providing recognition, and the recipient can see the sharer witnessing his recognition –

that, in some sense: ‘we are reading this the same way’. So highlighting alone does not constitute premising. But premising may rely heavily upon the way in which highlighting is achieved.

Reading is unremarkable to P and D, but it is understood by both of them that they must read and that by using particular gestures that they can achieve reading the same thing. P uses a reserved keystroke to bring to the screen the implementation of `getConnection`. This is intelligible to D and likely expected (We can’t say for sure); we can see (and P can see) that the switch was viewed without remark. P is inviting the reading of the implementation of `getConnection` as a means to do writing-work to solve the agreed problem. We can say that reading has been ‘occasioned’.

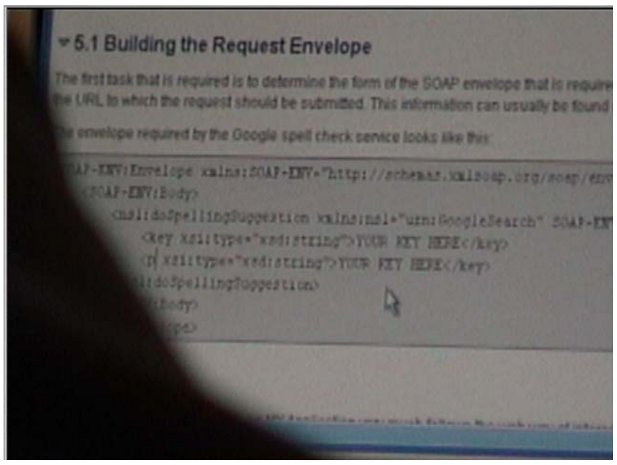
Reading code has a family resemblance to other forms of reading, but there are methods that are specific to it that arise from the code itself. Code has what we might refer to as a natural analysability (see [24]) to the programmers. It is a feature of this type of programming language that a line of code, ended with a semi-colon is interpreted by the compiler to be executed as a single and self descriptive instruction. A programmer is able to look at a line of code and (usually) to understand what it does. The highlighted line in figure 1 can be understood by the programmers to do the `getConnection` method using the three variables `jdbcURL`, `jdbcUsername` and `JdbcPassword` and to assign this to `m_connection`.

Reading of code is done by the reading of lines of code, but there is also an order to reading the lines of code that is understood and unremarkable to the programmers. This order follows in part the required logical order of code but also in part the agreed conventions of how to order code. Firstly, to comment on code’s required orderliness, the highlighted line is seen to be the sixth of seven lines that follow an ‘if statement’, all of which are executed sequentially if the criteria of the preceding if statement is fulfilled. It is naturally intelligible to the programmers that these lines will be executed in order one, by one. It is also intelligible to the programmers that the code in the lines refers to code elsewhere. The programmers understand that `getConnection` entails the execution of lines of code in the method `getConnection` which is written elsewhere. P uses a shortcut to bring this code to the screen, and it is obvious to both P and D what they are looking at and why they are looking at it. This highlighted line also refers to the variables instantiated earlier in this segment of code. There is a methodical, unambiguous relationship between code and other code and the methods of reading relate directly to this relationship; they must do in order to achieve a reading of the code that can be used in order to write more of it. There is also an orderliness that derives from social conventions and is of no consequence to the actual compiling and execution of code. This is typified in the declaration of the four variables in that they could be written in any order, but are written in a specific order to give intelligibility (e.g. Username is followed by Password). This is also typified in the laying out of the code, in its indentation to make the status of a section of code obviously visible as a section of code, and in the grouping of lines of code, such as the grouping of the four variable declarations

Example 3

Our third example is one of reading alone. This example is of pasting XML code into documentation. This is done by P when writing the user documentation for the development environment that is produced by the company. At the end of each development iteration any new features and functionality are described in the documentation. XML code is written as part of the testing of an application and is left as demonstration code for the users of the software. The demonstration code is usually described as part of the description of the features that that code invokes.

P copies and pastes some XML into the documentation. P formats the code by removing "□" characters that appear in place of tabbed indentation, and reproduces the indentations. P pauses momentarily on the last line, apparently (to us) realizing that this is not after all the last line. P then goes back to the XML and copies the line occurring immediately after the final line in the fragment, and pastes it to the bottom of the code. After adding this final line, P changes some of the code. He highlights the string of letters that are contained in the line beginning "<key… ", deletes them and writes "YOUR KEY HERE". He then adds a blank line below this line. He pauses momentarily and then highlights this line and pastes it to the blank space immediately below. He then deletes the word key and adds "phrase" (the screen grab in figure 2 is produced at this point) and deletes the YOUR KEY HERE and adds WordToCheck. He then sits back, stretches, leans forward and continues with typing the English language description of "Building the Request Envelope" beneath the code just added.



5.1 Building the Request Envelope

The first task that is required is to determine the form of the SOAP envelope that is require
the URL to which the request should be submitted. This information can usually be found

The envelope required by the Google spell check service looks like this

```
AP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/env
  <SOAP-ENV:Body>
    <ns1:doSpellingSuggestion xmlns:ns1="urn:GoogleSearch" SOAP-EN
      <key xsi:type="xsd:string">YOUR KEY HERE</key>
      <p xsi:type="xsd:string">YOUR KEY HERE</key>
    1:doSpellingSuggestion>
  Body>
  ope>
```

Fig. 2. The XML being worked on (The dark area to the left is the programmer's right arm)

P is working alone, but his work is still social, it is hopelessly so. The writing of the documentation involves reading code and is directed at producing a specialised version of code to be read by a user (also a programmer but not a co-worker!), when it is envisaged that they will be doing particular things – in this case sending a request to a web service. Here the reading is occasioned by the need to provide user documentation and the need to re-orient the pasted code to be intelligible to the user-developer as part of an instruction or 'recipe' for 'Building the Request Envelope'. It is to be read in relation to other instructions and the user-developer should perform some specific substitutions in the code itself. We can see P making sense of code as he writes a document that will be used to make sense of that same code. This sense making activity involves understanding how others read code, and understanding that they do this according to the same methods of ordering and analysability that P would invoke. We also see P formatting the code so as to be read, and in-fact changing the code into code that requires and obviates changes in order to get it to work (e.g. inclusion of "YOUR KEY HERE"). It might be argued that a novice may read the documentation, who may not hold the competencies in reading and writing code that P does, but the pasting of code here is pedagogical, the code exemplifies itself.

P has not had to coordinate his reading work with another as he did in examples one and two. However, in writing documentation he has necessarily and unavoidably followed and reproduced the conventions of reading code. An explanation of reading that involves some determination of how reading is done 'in the mind' is unnecessary and we find that it is possible and appropriate to rely upon socially available phenomena.

Discussion

"*… whatever a text's ultimate properties, it takes on its observed properties from within the work of reading. Reading consists of work that is always done in conjunction with a particular text. Rather than having two separate things – texts and reading – the two together constitute one object – a "text/reading" pair*" [17, p14].

In our ethnography we witnessed reading to be an extremely common part of programming work and, in line with Livingston's above statement, have attempted to demonstrate such reading work to be done by programmers in conjunction with code. We believe that most work in programming ought to be studied with a view on both technologies and practices. In this section we will summarise and discuss findings from the examples in the last section, and then make some more general remarks about a role for ethnomethodologically informed ethnography in understanding programming and in evaluating and contributing to technologies and methods of programming.

Code as Occasioned, Orderly and Analysable

In our three examples we came to discuss how reading code was 'occasioned', and how the code was 'orderly' and 'analysable'. By occasioned we mean that we can locate the circumstances in on-going programming work that make the reading of code the relevant next action for the programmer or programmers to undertake. By orderly we mean that there were expected and reproduced ways of ordering the layout of code, and by analysable we mean that programmers had consistent and expected ways of making sense of code. These three terms are each with their precedent as a topic of enquiry in sociology, but our intention is not to try and introduce them as obscure technical terms, but to demonstrate their relevance to understanding the situations in which programmers read code, and the practices and forms of knowledge they rely on when doing so. By using these three terms we are giving names to common patterns of work we observed in programming. They are descriptions of observable practice and not hidden forces or constructs that somehow give rise to observable practice.

Reading is occasioned in that it comes about due to particular circumstances that make reading the obvious next thing to do. The reading of code is occasioned in several ways in the examples. In example one, running a test highlights an error message, which provokes following a hyperlink to the location of the error within the code. D's reading the code aloud leads to P suggesting a solution to the error. Example two recounts attention being drawn away from a model on paper and to the screen. It is occasioned by a difference of opinions on what a problem is and how it is to be solved and is premised by use of body movement, talk and highlighting of text. In example three, reading code is occasioned by having to produce a user document detailing code. Cutting and pasting code into that document occasions a reading of it to see if it fits that purpose (of a document to be read by a user). As should be clear, reading is occasioned regularly within the flow of other activities such as writing and talking. Reading of code is occasioned by practical requirements and achieved by a tangle of embodied actions, and facilitated by available text, tools for reading text, work tasks and conventions of the community.

Code to be read is analysable and orderly. As Livingston states: "*The contextual clues in a text offer the grounds, from within the active, participatory work of reading, for finding how those clues provide an adequate account of how the text should be read*" [17, p14]. While we agree wholeheartedly with Livingston's above conception of reading, it clearly, as he admits, needs to come with two provisos: are the contextual clues clear, and does the reader posses the unique adequacy (the required skills and knowledge) to work them out in their reading work? Therefore, when we talk of the natural order and analysability of code for programmers it is important to examine these notions. (1) The degree to which a given piece of code is analysable is dependant on the reader's understanding and knowledge of the syntax and semantics of the language it is programmed in. (2) Issues of granularity and engagement are important – understanding what fragments of code do may be more easy than achieving an understanding of the overall structure of a program. And deciphering a given piece of code may take more or less time dependant on the source of the code, the methods used in its construction, the way it is written and the laic practical reasoning skills of the readers. And (3) that code is understandable to given readers, may well be separate to potential disagreements about whether the code is 'good', or whether the readers of code would have written it in the same way to achieve the same ends.

The Need for Observations and Understandings of Practice

Ethnographers, as well as programmers, can observe programmers reading, they can see programmers do it, they can listen to them talking about what they read, and they can observe how they treat other programmers with relation to whether that programmer is reading or not. To understand the 'reading work' the ethnographer should benefit from some understanding of programming. Although some interesting aspects of reading are available to be seen in what other people do, the other half of the 'availability' of reading code is that reading is a socially learned skill. In becoming a programmer you learn, amongst other things, to read code in the way that programmers should, can and do read code. We think that for a full understanding of reading code, it would be interesting for ethnographers to read code for themselves. To do so is following the ethnographic technique of Livingston who, upon recognizing maths was "not a spectator sport", proceeded to work on mathematical proofs for himself.

We have found very little written on what 'jobbing' programmers actually do (in an on-going fashion, in practice), let alone how they read. For instance, one topic that takes reading programs as its subject, and yet fails to address reading is graphical programming. For example Green [10] works "assuming that programs are written by translating cognitive structures into code [and] that part of the process of comprehending a program is parsing it back into the original cognitive components" (p25). Graphical programming is not without critique: Petre [22] states that it unproblematically assumes graphical programmers have the "unfettered meandering eye of the casual art viewer" rather than acknowledging the "purposeful perusal" required. Petre however does not elaborate on purposeful perusal, and tends to assume that reading code can be understood by a consideration of the inherent qualities of text. She makes vague statements such as "readers learn rules of interpretation so that plain text is read serially (although it may be accessed at random)" (p41). In Petre's study, and in many other studies of programming, laboratory experiments are relied upon to generate the data. In an experiment, as much as in real world programming, reading is an unavoidable feature of programming work. However, it is problematic that studies of programming are undertaken in which the usual situation of programming is not studied. Experiments often try to iron out regular features of programming work such as being able to talk to other programmers, having access to other code, etc.. Devito Da Cunha & Greathead [8], e.g., devised programming tests and gave them to programmers in a university setting in order to make remarks about how to select professional programmers based on personality. It is intriguing that while the authors might readily acknowledge problems with the size of their sample, or the representativeness of their subjects, they fail to comment on the fact that their study in no way focuses on the process and methods by which their subjects pass or fail the tests. The actual, embodied, on-going, work of programming was not studied by those authors, and they did not acknowledge that programming by students for the purposes of getting a coursework grade has many differences to professional programming (see [12]).

A 'Turn to the Social' in Studies of Programming

When considering human-computer interaction, as part of programming, as elsewhere, the 'turn to the social' in computing is important. This encompasses the idea that the single-user-single-computer paradigm is only sometimes relevant, while a consideration of the social activities that computer use takes place in, and is a part of, is often crucial to understanding patterns of use, problems of use and possibilities for better computer support. The better the activity being designed for is understood, the less chance there is to make design errors that make difficult or disrupt work practices.

We find XP particularly pertinent and interesting in the context of our study and this paper since it seeks to implement an ethos and series of practices that promote good coding (writing, reading, talking) as a *socially shared and agreed upon practice* through activities like paired programming. In the work presented here – taking an ethnomethodological approach to understanding reading code - we seek to show how such an approach may be useful for understanding reading in the XP environment studied. We believe that this work can be generative for future studies of reading code, and explicating the practices and psychology of programming at the points and places they can be found 'in the wild' within social action. In the future we believe that through understanding XP in this manner we may be able to contribute to refinements of its methods and practices As Button and Sharrock discussed a decade ago:

"[given that] *attempts within computer science to transform the mundane practices of programming into professional practices recognizes the role of mundane practices in the professional work of programming… it may, in principle, be amenable to having its reliance in other areas explicated, and in that explication it may find ethnomethodological studies of its work a resource*" [5, p254].

Conclusions

Programmers share and rely on "programming's ways of reading code". The code is the thing that is read, and reading often follows through the logic of that code, but programmers also work to organise and coordinate when code will be read and for what reasons, and in doing so rely on the understanding that (when it comes to reading) other programmers will work in similar and predictable sorts of ways. We have argued that it would be useful in the design and evaluation of programming methods and technologies to understand reading in this way as an observable, practical achievement. We suggest such work could take account of ways in which occasion reading is occassioned, and that code to be read is orderly and analysable.

REFERENCES

[1] Alac, M., Hutchins, E., "I see what you are saying" Action as Cognition in fMRI Brain Mapping Practice, Journal of Cognition and Culture 4, 3 (2004), 629-661.
[2] Booth, S. Learning Computer Science and Engineering in Context. Computer Science Education 11, 3 (2001), 169-188.
[3] Brown, B. 'The next line': Understanding programmers' work. TeamEthno-online Issue 2, June 2006, 25-33.
[4] 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.
[5] 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.

[6] 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.
[7] De Souza, C.R.B., Redmiles, D., Dourish, P. "Breaking the Code", Moving Between Private and Public Work in Collaborative Software Development. In Proc. SIGGROUP, ACM Press (2003), 105-114.
[8] Devito Da Cunha, A., Greathead, D. Code Review and Personality: Is Performance Linked to MBTI Type?, CS-TR-837 School of Computing Science, University of Newcastle upon Tyne , 2004.
[9] Downey, G.L. The Machine In Me. An Anthropologist Sits Among Computer Engineers. Routledge, London, 1998.
[10] Green, T.R.G. Instructions and Descriptions: Some Cognitive Aspects of Programming and Similar Activities. In Di Gesù, V., Levialdi, S., and Tarantino, L. (eds.) In Proc. AVI 2000, ACM Press, 2000, 21-28.
[11] Greiffenhagen, C. Sharrock, W. Durkheimian Aspects of Formal Constructs: Rranscendental 'Versus' Social Properties of Mathematical Proofs. Paper presented at Sociology after Durkheim (University of Surrey, Guildford, June 21, 2006).
[12] Holland, D., Reeves, J.R. Activity Theory and the View from Somewhere: Team Perspectives on the Intellectual Work of Programming. In Nardi, B.A. (ed.) Context and Consciousness. Activity Theory and Human-Computer Interaction. The MIT Press, Cambridge, MA, 1996, 257-281.
[13] Hutchins, E. Cognition in the Wild. MIT Press, Cambridge MA, 1995.
[14] Kim, M., Bergman, L., Lau, T., Notkin, D. Ethnographic Study of Copy and Paste Programming Practices in OOPL. In Proc. International Symposium on Empirical Software Engineering ISESE'04, (2004), 19-20.
[15] Kristoffersen, S. Designing a program. Programming the design. TeamEthno-online Issue 2, June 2006, 34-51.
[16] Law, A., Charron, R., Effects of Agile Practices on Social Factors. In Proc. HSSE 05, ACM Press (2005).
[17] Livingston, E. An Anthropology of Reading. Indiana University Press, Bloomington, 1995.
[18] Livingston, E. The Ethnomethodological Foundations of Mathematics. Routledge, London, 1986.
[19] Livingston, E. Making Sense of Ethnomethodology. Routledge, London, 1987.
[20] Mackenzie, A., Monk, S. From Cards to Code: How Extreme Programming Re-Embodies Programming as a Collective Practice. JCSCW 13,1 (2004), 91-117.
[21] Martin, D., Rooksby, J. Knowledge and reasoning about code in a large code base. TeamEthno-online Issue 2, June 2006, 3-12.
[22] Petre, M. Why Looking isn't Always Seeing: Readership Skills and Graphical Programming. Communication of the ACM 38, 6 (1995), 33-44.
[23] Reeves, S. The code document's structure and analysis. TeamEthno-online Issue 2, June 2006, 34-51.
[24] Sacks, H. On the Analyzability of Stories by Children. In Turner, R. Ethnomethodology. Selected Readings. Penguin Education, England, 1974.
[25] Sharp, H., Robinson, H., An Ethnographic Study of XP Practice. Empirical Software Engineering 9 (2004), 353-375.
[26] Suchman, L.A. Plans and Situated Actions. The Problem of Human Machine Communication. Cambridge University Press, Cambridge, 1987.
[27] Tolmie, P., Hughes, J.A., O'Brien J. and Rouncefield, M. The Affordances of Paper and Co-Presence. Unpublished paper, Department of Sociology, Lancaster University, 1999.
[28] Watson, D.R. Ethnomethodology, Consciousness and Self. Journal of Consciousness Studies, 5, No. 2, 202-223.