‘The next line’: understanding programmers’ work

Barry Brown
Department of Computing Science
University of Glasgow
www.dcs.gla.ac.uk/~barry/

Introduction

It is perhaps surprising that while computer programming has enabled so much change in the world, as a practice we are still relatively unaware of its details and practices. It is obvious, for example, that writing a program in a functional is quite significantly different from an object orientated language. What is less clear is how these differences actually shape and influence the practice of ‘coming to get a computer to regularly do what we want it to do’. Do programmers approach problems differently? Would a functional programmer start at a different part of a problem from an OO programmer?

More broadly, despite the pervasiveness of programming as work, we only have the most schematic understanding of what the work of programming *actually is* and the resources drawn upon. Indeed, programming as an activity has been systematically distorted in many accounts in the literature. For example, in the methods literature programming is laid out an orderly activity which has to be systematised and organised. Even, the agile methods literature skips over the details of much of the practices of applying methods. How, for example, is it that patterns are reinterpreted afresh in each new setting? How is re-factoring as a coding activity decided upon? While agile methods seem to be gaining increased support, there is still little understanding of *why* they are successful or popular - agile methods might be the answer, but we are still unclear about what the question is.

Indeed, to social scientists coding at times can suffer from being festishised as an activity. One can talk at great length about ‘code’ with little concern for what anybody actually does when coding, or the nature of code itself. Deleuze, for example, talks at length about various algorithms (such as in his discussion of the Koch snowflake (Deleuze and Guattari 1987)), but he has little interest in the work of applying and using those algorithms. What I am interested in discussing in this paper is understanding the work of coding. Indeed, I would argue that as earlier studies of the work of scientists uncovered, the actual practices of science often bear little recognition with the accounts given by either philosophers or practitioners. Coding is an expert practice that is often described in only the most cursory terms or even distorted terms by its practitioners, and at times in ways which distract attention from many of its most important details (for example, (Graham 2004)). Programming is a local practice - one which takes place at the computer and on the internet, and understanding code as it is written similarly has to take a fine grained approach to understanding that practice.

The current state of the field

Experimental work on programming, which dominate the field currently, suffer from a number of serious problems in getting at the practice of programming. There are a number of assumptions that these studies make which should be called into question. The first is that all software problems are equivalent. Experiments often take the from of setting a trial problem and then testing how the efficiency of programming is effected with different conditions. If one considers that programming is as varied a practice as say, painting, then different forms of painting will demand very different techniques. Moreover, programmers themselves all engage in different practices - not least because nearly all are self taught - and one cannot assume a homogeneity of practices. This is not just a question of skill (although one would want to question any sort of simple comparison between expert and novice programmers) but of approach and worldview.

Perhaps more seriously, these experiments lack an investigation into what would be most useful for understanding programmers work - *what they do*. Instead we tend to get analysis of numerical measures of the outcomes of experiments. The worst culprit here is of course the measure of time taken to solve a problem - as if a painting done quicker than another is obviously better. However these experiments contain, although unexamined, exactly what is of interest. Programmers in these experiments *program*. Yet this is on the whole

left unexamined, and if it is examined it is done simply through extracting types. In any sort of coding we have something very rich which we can examine. Unlike some activities, coding is a publically available and examinable activity in that it produces code as it is written which we can analyse. One reason that this is left unexamined is that for most approaches there a few resources which enable the examination of this activity. If one relies upon assuming that programming takes place 'in the head' then the fact that all this work is publicly done is if anything a nuisance. One instead has to go chasing cognitive structures - 'a beetle in the box' to use Wittgenstein's example, in a box that can never be opened. Even though coding is publicly available as programmers code, and much of it is talked about at length by programmers in meetings, and as they program, often the focus seems to be on what is hidden rather than what can be seen.

In turn interview studies of programmers have serious shortcomings in how they rely upon the reflective skill of programmers, beyond what is usually possible or practical to examine. It is just too complex to get to describe what you do when you program in an interview - much like asking someone in interviews how to play the piano will hardly teach you how to play the piano. Surveys also rely upon the (extremely high) *post hoc* justification skills of managers rather than the practices of *in situ* programming. These methodological and theoretical shortcomings are perhaps behind our lack of understanding of what a software method actually is, and how it effects and takes form in the day to day work of programmers, managers and so on.

Unexcavated aspects of programming

An alternative, and the one pursued here, is the examination of recordings of code as it is written, alongside introspection, and self-reflection, into what one does when coding. Each of these can assist and help each other, in that it is difficult to make sense of videos of programming without reflection on ones own practices of coding - they are not data alone but rather that stand side by side with the ability to explore ones own practices. I am suggesting here that through looking at a record of programming as it is done alongside self reflection can help us understand what it is, as competent practitioners, is done as the work of programming. The model here is Sudnow's work on Jazz musicians (Sudnow 2001), where he extracts from his own experience a narrative of his own learning to play jazz piano. The problematic of learning, and the differences between manual skill, tactics and strategies are expertly dissected. Yet, taking a slightly more empiricist turn than Sudnow's use of recordings, I would suggest that recorded video data can be a supplement here, not only for its rhetorical contribution, but for how it can focus us on both the variety of practices (not everyone programs like us) and also the details which can be missed in our experiences in the moment.

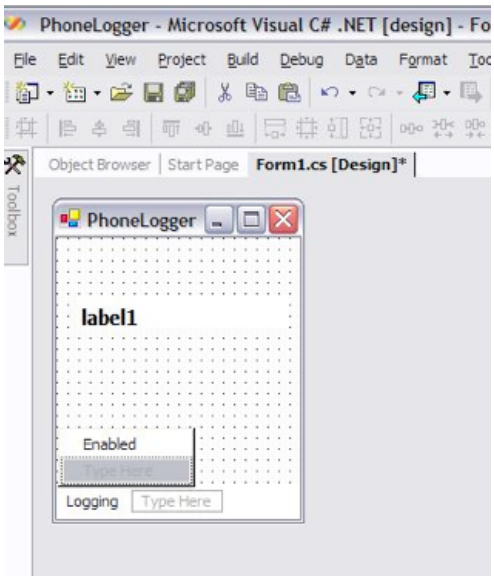
In particular, here I draw on a video of some coding done by a colleague of mine, Malcolm Hall, using Microsoft Visual studio in c#. While coding Malcolm ran a logging program which took screen captures of his screen while he was programming a small PDA application to log bluetooth devices on a PDA. I won't go into great depth with the program, and the screencaptures themselves have some limitations in analysing his work. However, from the captures we can extract (alongside self-reflection) some lessons about how coding proceeds. In particular, this *in vivo* recording methods lets us see problem as they are approached and the code as it is written line by line until there is finally a working program.

Our problem in starting analysis is how to frame the problem of understanding code? I suggest that one question we might seek an answer to is 'How does a programmer write the *next* line of code?', where *next* is any particular line proceeded by code, and followed by code (with two possible exceptions). Of course, one draws on lots of different resources when writing that line of code - no line of code is written blind, and there are lots of things we draw on to decided which line of code we will write.

Let us take a first extract of Malcolm's code, one of the very first lines of code which Malcolm writes on the project:

```
Private void loggingEnabledMenuItem_Click(object sender, System.EventArgs e)
{
     loggingEnabledMenuItem.Checked = !loggingEnabledMenuItem.Checked;
}
```

Prospective/Retrosepctive



Malcolm is writing some code which is linked into a menu on the phone. When the user clicks Logging/Enabled, his code will add a checkmark to that menu item 'selecting' it, and when the user clicks again it will 'unselect it'. This bit of code is written after Malcolm starts to draw the basic interface for his application (pictured above). He creates a menu and VisualStudio creates a 'stub' (an empty method) for loggingEnabledMenuItem. Malcolm then writes the line which toggles the value of the Checked value which is part of the loggingEnabledMenuItem object.

The first point to be drawn from this very simple example is the way in which code which is written is embedded in an environment. Gone are the times when programmers would write code with no reference to any other code. Code is written linked into an OS, tool, foundation, and so on. In building houses, the tools and materials of building are rich with the knowledge and history of building, each tool encapasulates the knowledge of millions of previously built houses. So it is in little extract of programming. Indeed, the first line of code here isn't even written by Malcolm, it is generated from the GUI builder. This line of code thus depends upon what has gone before - the class definitions, using other methods, the OS code, Visual Studio, a whole world which links all this together. It is remarkable quite how much software reuse is going on here, especially since software reuse was so much of a goal of 80s software research. In many ways they have been gloriously successful. This embedding of code in ac context goes beyond the tools in use. In writing this line of code there is considerable *retrospective* orientation to what will fit. What to write depends to a large extent on what has been written before. Each line of code is unique in some way, but it also gets its sense from what has been written before. In this case it is not strictly a line of code but the creation of the 'loggingEnabledMenuItem' menu item. Code depends for its sense on what has gone before - what has been written in the previous lines.

This line of code that is written also displays a second feature of code - its *prospective* nature. The code itself here doesn't do anything. It is on all accounts a useless bit of code since it will only toggle an item on a menu. Yet the code does make lots of sense as a starting point on an application where the user will want to turn logging on or off. In this way the code is written with a prospective orientation to what will happen next. The code gains its sense from how the program will unfold, rather than simply where it is now.

These two orientation in code writing - the prospective/ retrospective nature, are in some senses fairly obvious. Language has an indexicality which is similar, but it does have a number of interesting ramifications. One is the way in which code can seem to 'write itself' - that the next line of code is obvious when one is following what is being done. When reading code 'surprises' are problems. The sense of the code develops in a body of code, of course, but for this it relies very much on the code around it.

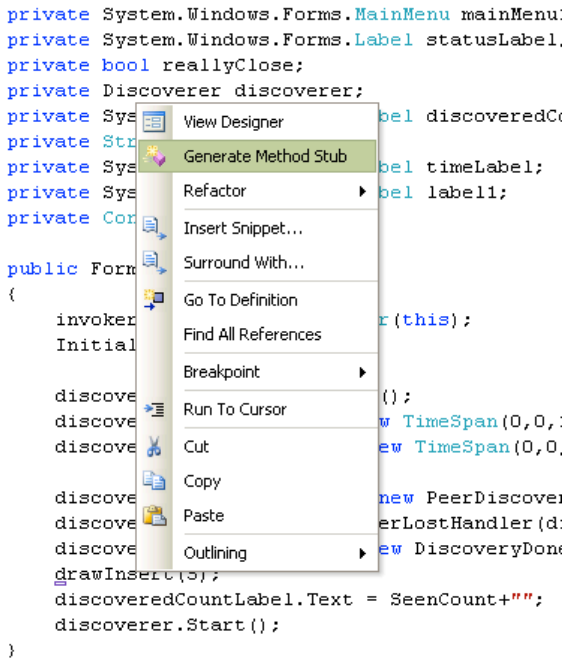
Another interesting point that follows here is how code *projects* into the future its use. That is to say, that one can guess from a reading of code what comes next. As it is written you can come back to code and read what can now be done using what is there. So in this case, you have an item which can be toggled that turns logging on and off. So code can now be written which uses that toggle - later this code becomes:

```
private void loggingEnabledMenuItem_Click(object sender, System.EventArgs e)
          {
               loggingEnabledMenuItem.Checked = !loggingEnabledMenuItem.Checked;
               if(loggingEnabledMenuItem.Checked)
               {
                    discoverer.Start();
               }
               else
```

```
            {
                discoverer.Stop();
            }
        }
```

So here the code has expands out from something which only toggles a menu item, to code which then either starts or stops the discoverer based on this menu item. This aspect of code - it's 'projection' also explains some features of code as it is written. Button and Sharrock in their excellent discussion of coding (Button and Sharrock, (Button and Sharrock 1995) discuss how programmers make use of ' scaffolding' code - code which doesn't do anything, but is there to indicate that code should be written, in the future to do such and such. Any coder will recognise scaffolding code.

Yet, compared to other construction activities what is perhaps surprising about coding is how little scaffolding there is. Programmers do not always go ahead of themselves sketching out a structure before it is actually coded. Far more common is that the written code projects forward, to the competent practitioner, what needs to be written, what is missing and so on. So actually not much in the way of scaffolding needs to be written, beyond what the compiler requires so as to run. The projected future of the code can be 'read' from what is already written. When programmers do explicitly project forward their code into what will be written, frequently it takes the form of a comment rather than scaffolding: an example of this is the %TODO comments in code allow programmers to explicitly project into the future what their code should do.



The latest release of visual studio has a design feature which displays some of this projection. As you are writing code, if you write a method which has not yet been declared, you can choose to have the system generate a 'stub' for you of the declaration. This 'declaration by use' supports the building up of the necessary declarations for code where it is used. The bits of code that are written 'project' what is going to have to be written later, and this feature allows you to put in the code to indicate where that will be called.

Planning problems

Returning to the notion of plans, I would argue that programming does not take place as a planful activity where a 'plan' is outlined in advance of what the whole of the code is, a plan which is then filled in over time. Instead it is more a dialogical relationship between the code, the developing practice of the machine and what the programmer arranges as to what they will do. Suchman's comments on plans here stand as a useful counterpoint to the description of programming in terms of 'plans' and their 'implementation' in some studies. Programming is a discovering activity, and as such is not determined in advance - it involves the interaction and engagement with what could be, an exploration of what a computer can do.

Following Visual Studio's lead here, one might explore technically how we could better support the projection of code by programmers as part of *ad hoc* planning. Support for the generation of structure as a side effect, or as part of, writing code would support better the 'bootstrapping' relationship between program structure and code. One could, for example, write methods without having to move from one part of the program by inlining them into the code and then having that code 'sent' to the respective method. Another extention could be instantiation by use. Often code contains objects which will only ever be instantiated once - such as a controller object for example. This distinction is often made by using a lowercase version of the class name

for this single object - BluetoothController for the class, bluetoothController for the object. One could imagine implicitly referring to this object and having it constructed as a singleton from that class[1].

This not to downplay the importance of the programmers sense of what they are doing and their position in solving the overall problem. Programming does include considerable *planning* but rather in the sense of an outline (or projection) of how the problem will be approached rather than a fixed organisation of what will be done. This plan does involve considerable development and change over time.

It is important to point out that there are two different plans at work. One plan is the way in which the programmer will approach the problem, the order in which they will solve the problems. The second is the plan of how the computer eventually will work. A sensible approach is often for the programmer to treat these as equivalent and to program the computer in terms of stages of the final solution. This is particularly useful for testing, since often previous parts of a problem depend on earlier parts (although not in the case of unit testing. However, they are *not the same* and often the work of the programmer and the computer are conflated.

One approach to problem solving is problem decomposition whereby a problem is decomposed into smaller problems which can be solved one by one - as (Abelson and Sussman 1996) puts it programming is breaking large problems into small ones. So, if one wants to do a word count program one approach might be to write code that reads in text and prints out numbers first. Then you would write code which will go through the text and count the number of words. Yet the solution (or the computer) also involves decomposing problems — such as in quicksort which will divide the problem of sorting numbers into sub problems involving smaller lists.

The decomposition of the programmer is *different* from that of the computer, however, in many ways. One is that the stage of problems which are to be solved in writing a large program are ordered in terms of what problems can be solved now, and what problems will depend upon others for their testing. So the division of problems involves working out 'what can I do now' and separating this from 'what can I do later'. Since most systems rely on some sort of rudimentary UI, coding often starts (as in this case) by working on a user interface. Problem decomposition for the programmer is therefore one of also problem *ordering* - of finding some solvable beginning problems, some place where you can start. Following that you need to work on harder problems and then work on fitting all these problems together.

What is important then about programming is getting some sort of handle on how one might order the problems that one has to solve, and how they might interact with each other. Indeed, one might approach difficult problems first and then move onto the easier problems, since how the difficult problems are solved will have more impact on the other problems. Or one might seek to attempt the more general problems first. There is a problem ordering alongside a problem decomposition.

This sort of arrangement can be seen in Malcolm's code. Here he starts by writing the logger code which tracks how many bluetooth devices have been seen and writes this to the user interface. This is developed alongside the bluetooth code (in a different class) which will actually find the required devices and do the bluetooth searching. Malcolm moves between each problem in line: first get it to find bluetooth devices. Then handle loosing bluetooth devices. All the time he keeps updating the user interface code to test the other code. His progress here follows an ordering of problems which is in some sort of rough tandem with how the computer will proceed, although with some differences.

As an aside, along with the decomposition of problems for users, and the executing of the computer, a programmer must also decide how to decompose the problem in terms of the structuring entered into the computer. Most languages take some sort of decomposition of a program in terms of its structure, and one that is connected to but not exactly similar to, the decomposition of the problem execution. It is interesting to

---

[1] Christian Greiffenhagen has pointed out to me that this is the same as in maths, where objects and classes are often defined simultaneously and differentiated using case - e.g $e \ni E$ to mean any e which is a member of the class E, for which the properties of e can then be explored.

note that much of the code that is written in object oriented languages, for example, concerns linking together different objects in the code, calling other objects instantiating objects and so on.

Relationship between the code as written and the code as executed

Programs must be written for people to read, and only incidentally for machines to execute. (Abelson and Sussman 1996, xv)

In the process of writing code one enters into a dialogue with a computer about what is not, and what is, acceptable. While the ability of others to read code, and the readability of code, are important - and often a way of differentiating a good from an excellent coder - it is the computer itself who must be impressed upon all others. Without a compiling and running program, whatever the quality of the resulting code, it is left wanting. Much of programming, then, is interacting with the computer - with the compiler, the running of code and the debugger so as to get closer to a finished program. Outside classroom situations, the computer is the final judge of whether code works or not, however elegant.

In this sense the quote from Abelson and Sussman's classic textbook is incorrect. Computers are not in any sense incidental to the job of programming, and to claim so is simply to confuse what is maths and what is computer science. Computer languages do *double duty* in that they work as an understandable notation for humans, but also as a mechanically executable representation suitable for computers. Computer languages are quite different from mathematical notation which in many cases depends upon what what a competent mathematician can follow (see for example the debates around the three colour problem (MacKenzie 1999)). Computer code *has to be* automatically translatable to a form which can be executed by a machine, one with little if any reasoning ability. Abelson and Sussman's quote is applicable to *psuedocode*, but not to real code. Programming languages thus sit in an unusual and interesting place - designed for human reading and use, but bound by what is computationally possible.

All this means that one of the key and most important jobs of programming is the dialogue with the computer to work out what works. The relationship between the code, and its behaviour on the executing computer is in many ways the *essential* role of code. This takes many different forms: in some ways simply writing the code is a form of working out what will work out. A programming language itself in its syntax and semantics constrain what is possible to write. It is also in the edit-compile-test cycle, where the code is compiled to see if the code checks out in terms of static checks (syntax, typechecking, etc.), and then by testing to see if it does what it should. Code is often executed during its writing to see the effects of certain parts of code that have been written - experiments in a way. Since the execution of code in a complex codebase is not predictable 'at a glance', running the code is often a much easier way of seeing if it will do what is intended.

Executing code, for most programmers, is their main way of testing the code. In doing this the programmer usually has some sort of expected behaviour - and often unexpected behaviour is what happens. If the program doesn't 'work' then the programmer must work out from what happened what has actually happened with the code. Often the bug is immediately obvious, and at other times not. The behaviour of the computer is interpreted through understanding what the program code is meant to do, and how its potentially aberrant behaviour can correspond with a potential bug.

Some codebases are so large that compilation takes too long to make testing frequently practical, although mode codebases are split into parts so that the whole code does not need to be compiled for testing. Debuggers are also extensively used here to interrogate the execution of code as it runs, although debugging a large program where changes are distributed through the code.

While debuggers are obviously been a great assistance for programmers, the relationship between the code as it is run, and the code as it is edited could support more technology. Just now there is something of a divide between the representation of written code and the execution of that code. For example, it would be straightforward for a IDE to track what bits of code are executed when a program is run in debug mode. This could be displayed on top of the code in the form of a light representation of what lines have been executed and what lines have not. This would show - at a glance - to a programmer what parts of the code are being tested. For example most coders have experienced the frustration of attempting to debug a clause in an if statement, only to find that the bug is with the condition and the clause is not being executed. Showing what

code was executed and what has not been would display this sort of information at a glance. It is also promising to explore how the history of code might be better brought into the representations used in computer science (Chalmers 2004; Bell, Hall et al. 2006).

Social relationships in code

A second important relationship which figures in code is that between coders. Multiple programmers often work on the same program, and as the open source movement shows, spend considerable time working together to solve problems. The social relationship around code has obviously been a key part of the literature on coding, and indeed this is one of the rationales behind peer programming.

This social relationship is shown in the way that program code is written - *pace* Abelson - for others to read. Yet a relatively unexamined aspect of this is how exactly program code is written for others to read. As we described above, program code has a certain trajectory - one can read what is going to happen next in some code. Yet code is also deliberately written such that it can be read by others at a later date, and also the programmer themselves who may have to come to the code later. Comments are one obvious example of this, but also program code might be structured in such a way as to enable its future comprehension. This is a sort of predictive work of programmers.

An interesting question here, to be examined in programmers practice, is how programmers make judgements about who will read their code in future and what they need to understand. The writing of code can display an understanding by the programmer not only of what the system needs to do to execute their code, but also who is likely to need to read that code in the future - if anybody. Some code for example is characterised by its quick and throwaway nature - code that isn't likely to be read by anybody in the future (although as is often the case it is). Programmers often also spend time 'cleaning up their code' making it presentable to others.

Language features often explore this future redability of code. An example of this is the use of generics in code. Generics are general data structures that are specialised in a particular way only to deal with one type - for example a list of ints. While generics do not add anything to the speed of programs, their key pay off is in how they make it clearer in code what is actually being done with a particular paramatarised data structure. A particular list, when seen by someone later, is visibly readable as a list of whatever it is, rather than simply as a list. This, through an extension to code, adds to the readability of the code in question to others. Providing general awareness of code reading has also been provided by systems such as 'edit ware and read ware' (Wexelblat and Maes 1999).

Structures in code

A last, and final, point I will make concerning coding is its structured nature. Ever since the 'structured programming' movement considerable effort has been made into making code modular, with a lack of dependencies, refactored code when necessary and so on. In modern object oriented programming a large amount of program text is concerned with this structuring - declaration of methods and classes, calling methods, setting events and so on. This structure sits on top of the textual organisation of program text - its separation into files, the line by line nature of text and so on.

In the video of Malcolm coding his rearrangement of the structure of the program as it develops can be seen - he starts with one UI class, starts putting code into that, and then creates another class - a controller which will run the code called from the UI class. He then creates a last class, a discoverer, to abstract out some of the more generic code from the controller class. Much of his work of programming is thus organising the code as it is written.

Alongside this class decomposition there are also a rich number of other structures which pepper program code. For example, the graph of which objects call other objects ,the call graph of a program and data paths through a program. There are also more descriptive structures such as the 'hot' and 'cold' parts of the code, the different languages used, age of code and so on, all these can be used to structure and separate code beyond what is written in the text. Much of programming involves the editing of code which spans across two different classes (and thus two different text files) but which is intimately connected in terms of data flow and

method calls. In these ways the editing of programs over time, while it may involve the editing of multiple files or classes, and so have changes distributed across multiple files, may take place across parts of a program that are very close in terms of their structure. The representation of code thus can contribute to how 'local' changes are in a program. One disadvantage of object oriented programming is how it can distribute changes across many different objects at times, rather than encapsulating changes. Indeed, much of the job of object oriented programming is fighting with the structure of a program to effectively do what one wants - structuring and object oriented decomposition give lots of advantages to programs, but they can also make development difficult or 'jumpy', as one is working on multiple objects which call each other. This is one reason behind the use of aspect oriented programming where code can be written which 'cuts across' parts of the program - such as by specifying code which would be run after a particular variable is changed or a method is called.

The ideas of aspect oriented programming can be taken a little way further if one understands the close interaction of parts of code. Above I mentioned the notion of 'inlining' methods into program code. One could also specify clauses such as calling a method but adding extra code (a 'but' clause) which would be executed at that point in the method, causing a modification of the methods behaviour. Again this supports the crosscutting of functionality across different modules in the code.

Conclusions

Here I have outlined some on the different contingencies which programmers must be concerned with when working on the 'next line' of code, covering four different aspects of how programmers write 'the next line'. Each of these aspects of coding are orientated to 'in the moment' as programmers work on writing code: the prospective/retrospective situation of code, the role of plans - in particular how problems are decomposed and ordered, the 'dual duty' nature of code and the relationship between code as executed and code as written, the social aspects of code and lastly the structural aspects of code.

Programmers must relate to and bring these different aspects of coding together. However, the generic analysis here presents a number of shortcomings. Namely, the problems presented here are *generic* to programming. What adds and produces the pleasure of programming, of course is that it is a distinct practice for each program. Each program presents its own problems and issues. Future work will endeavour to focus more specifically on outlining and understanding what specific problems are entered in each specific case, and how this variety of programs interacts with the generic concerns outlined here.

In closing I would point the direction back to understanding code, its writing, reading, situation and production. Extensive work here will enable a concise examination of how programming can be better understood and designed for as a practice.

References

Abelson, H. and G. Sussman (1996). Structure and Interpretation of Computer Programs, MIT Press.

Bell, M., M. Hall, et al. (2006). Domino: Exploring Mobile Collaborative Software Adaptation. To appear in Proc. Pervasive 2006, Dublin.

Button, G. and W. Sharrock (1995). The mundane work of writing and reading computer programs. Situated Order: Studies in the Social Organization of Talk and Embodied Activities. P. t. Have and G. Psathas. Boston University Press of America.

Chalmers, M. (2004). "A Historical View of Context." CSCW Journal **13**(3): 223-247.

Deleuze, G. and F. Guattari (1987). 1440: The Smooth and the Striated. A Thousand Plateaus: Capitalism and Schizophrenia. Minneapolis, University of Minnesota Press.

Sudnow, D. (2001). The ways of the hand: a rewritten account. Cambridge, Mass., MIT Press.

Graham, P. (2004). Painters and hackers, O'Reily.

MacKenzie, D. (1999). "Slaying the kraken: the sociohistory of a mathematical proof." Social studies of science **29**(1): 7-60.

Wexelblat, A. and P. Maes (1999). Footprints: History-Rich Tools for Information Foraging. Proceedings of CHI 1999.