

DESIGNING A PROGRAM. PROGRAMMING THE DESIGN.

Kristoffersen, Steinar, Faculty of Computer Sciences, Østfold University College,
Halden N-1757, Norway, steinar.kristoffersen@hiof.no¹

Abstract

The objective of this paper is to examine in detail the activity of programming as such. In particular, it is concerned with those aspects of requirements analysis and design which are (and can probably only be) embedded into programming itself, regardless of the method or project model that is used. Software development is usually based on a rational organization of work into stages such as requirements engineering, analysis, design, programming, testing, etc. Each is seen as resulting in artefacts which are then further refined and concretized throughout later stages. Different methods are often contrasted to each other with regards to the amount of design that is required before implementation, the looping of stages and the extent of formal documentation which is handed from one stage to the next. However, little evidence can be found that one method is actually better than the others. This paper indicates that one reason might be that programming is poorly understood and that too many methods and process models still wrongly assume that programming simply translates an existing design into code.

Keywords: Ethnographic studies of programming, requirements analysis, design and programming as epistemic categories, tendentious talk and under-determination of designs and problems.

1 INTRODUCTION

Information systems development is usually seen as a predominantly rational process, for which proper accomplishment of each stage, such as requirements engineering and design, will greatly facilitate the work in the subsequent stages. Moreover, many part-products of each stage are usually seen as more abstract representations of products that are going to be realised later. Thus, development project models encompass a broad range of activities which directly are expected to contribute, ultimately, to the implementation of a software system. This is not only limited to the waterfall-model. Moreover, it is not obvious that it works:

[M]ost projects to develop production engineered software products spend more of the project's effort in activities leading to a document as their immediate end product, as compared to activities whose immediate end product is code (Boehm and Papaccio 1988).

It is usually believed to be better still, to work out problems sooner rather than later and on a higher level of abstraction rather than whilst obscured by technical details. However, bigger projects require more people and a wider set of competencies involved in order to carry this ambition through. This, in itself, represents a challenge, which needs to be addressed. It is then often recommended to see as a starting point that software development is highly interdisciplinary and deals with many aspects of development *other* than programming (Kemerer 1998). This does not prevent that obstacles involved in coordinating functionally separate activities and resource pools often are seen as major contributors to the “software crisis” (Kraut and Streeter 1995) as well. Others argue that the reason is that programmers and engineers are doing the design work and making marketing decisions (Cooper 2004). The answer may be a little of both, or neither, since the “crisis” seems to be rather permanent.

¹ The research described in this paper was conducted while the author was a post-doc at the University of Oslo

Numerous studies have documented the dismal success statistics of software projects: approximately one-third are delivered on time and within budget; another one-third are delivered late or over budget; and the remainder are not delivered at all (Denning 2004)

Although it is arguably a crass simplification, this paper sees requirements engineering and design, e.g., as a preparation for programming. Given that the cost of these preparations is substantial, and their effect seems somewhat disappointing, it is warranted to engage in a more detailed examination of the relationships between programming and various other systems development activities. This is the objective of this paper.

There are many common factors in the conceptualization of “software development” as it is seen from research in cognitive science, CSCW (Computer Supported Co-operative Work) and software engineering (Brooks 1987; Clases and Wehner 2002; Grinter 1999; Storey, Fracchia and Muller 1997): Starting from current and future users’ requirements (explicit, emerging or tacit), designers can create a *manifest* design which is exogenous to the task of programming it. Design is an activity *as well as an artefact in itself*, which expresses the same requirements on a more detailed level of abstraction from “needs”, but more abstract still than code. Stepwise refinement down through these levels of abstraction is one way of managing the complexity of software projects. A chronologically ordered sequence of functional activities comprises a life-cycle for the project, which drives it forward towards completion. It is commonly argued that different and distinguishable competencies ought to be brought to bear on each of the activities, one of which is, notably, design.

This paper challenges this view. It presents an ethnomethodologically-informed analysis of software development indicating that many activities which are categorically termed “design,” are not performed *at all* distinguishably from programming. It is, and can only be, performed by programmers as part of making a “cautious” set of specifications, instructions and mutual obligations about what they are going to do next. Other examples show how design is an invitation to discuss a problem, rather than the conclusion of a problem-solving exercise, in a fashion that carefully *avoids* pinning down the exact solution; rather it maintains a wide solution space for programmers to figure it out. In the first instance, it is inseparable from programming *as an activity*, and in the second it is inseparable from the programmer as an *individual member*.

One potentially significant implication is that software engineering ought not to be treated primarily as a set of co-ordination challenges (Grinter 1998). At the very least, future improvement of software development methodologies, especially with a view towards creating a “design science” and tools aiming to support design (Rosson, Kellogg and Maass 1988), need to take the findings of this paper into account. Design might be intractably un-supportable by computers, and certainly if new groupware is based on an idealized understanding of how it is accomplished. Instead, a complete respecification of “programming” as a genuine form of co-operative work may be warranted. The term “design” might be a useful heading under which multifarious activities referring to problem solving, technical approaches and software engineering “competencies” are loosely compiled. However, they seem to bear very only “familiar resemblance” to the design work of other disciplines, some of which work closely *with* programmers in their projects (graphical and industrial designers, business developers, etc) and even *within* programming itself, which seems to comprise many *different* designs and design activities. A future research agenda of the “language games” of design could grow out of this appreciation that design ought to be treated as a deeper epistemic theme (Lynch 1997), rather than simply as “given”.

The main theoretical contribution of this paper is that it shows how a perspective of design as “under-determined” in a similar fashion to Garfinkel’s use of tendentious instructions (Garfinkel 2002), and that this is exactly the skilful and social accomplishments of a design space that is sufficiently open and “forgiving” to allow programmers to recognize and solve hard technical problems in a virtual world which they all see differently. This is more than an appreciation of Garfinkel’s ethnomethodology. It applies to programming as an archetypical example of “the social” in broader

terms, which from lessons learned then can be brought to bear on more general social theory in future research.

2 RESEARCH SETTING AND METHODS

This paper describes work in a small software company which has specialized in developing complete and partial application suites for mobile phones. This means that they work on contract, usually for a baseband-chip producer or handset vendor, in projects that are “owned” by vendors, usually in close alliance with one of the large operators. The company in question is currently engaged in projects for many of the big actors in this industry. Currently, the main thrust is on developing middleware for future phones. They need to integrate (or at least provide hooks for integrating) support for services that they expect will be in demand. An integrated development environment for mobile phones has to offer sophisticated call control (for unspecified handsets), a unifying event-control architecture (for future applications) and a tailorable set of user interface components so that it can get the “look-and-feel” of models from many different vendors. The development takes place in a stepwise fashion, loosely inspired by various agile software engineering methodologies. The programmers themselves talk about a requirements-driven, yet flexible life-cycle, which is well documented within the company’s intranet. Examining the web, however, shows that it is permanently in an initial and categorical state. The details of the projects, their enactment, success and failure are only found in the “lived practice” of programming. The focus of this paper, therefore, is on the common workaday interaction between the programmers themselves, if, when and as they do programming.

The research presented here is based on several full days of observation taking field notes and recording a small number of samples from conversations taking place. The excerpts presented are only a small sample from a big body of material showing stable and recurring patterns with regards to what constitutes design in this setting.

Ethnographic studies gained a strong foothold in CSCW in the nineties (Hughes, King, Rodden and Andersen 1994) and have become commonplace in related fields such as Information Systems (IS) and software technology research lately (Sakthivel 2005). As popularity as grown (Beynon-Davies 1997), it has become easy to fall into the trap of believing that ethnography is straightforward, which it clearly is not (Forsythe 1999). This paper is not only seeing ethnography as a suitable approach to collecting data from which, in the next instance, a research result can be compiled. Instead it sees ethnography as particularly well-suited to support the use of *ethnomethodology* as an analytical frame of reference (Garfinkel 1967). The “unique adequacy requirement” of ethnomethodology points toward needing detailed empirical accounts (Lynch 1999). Although this adds to the empirical bulk of argument and in some instances might make it more cumbersome to read, hopefully it also makes it more worthwhile.

It would be beyond the scope of this paper to offer a detailed account of ethnomethodology’s program. Ample discussion can be found in the CSCW literature (Button and Dourish 1996; Harper 1995; Hughes, Randall and Shapiro 1992; Shapiro 1994; Sharrock and Randall 2004) and beyond (Garfinkel 1967; Garfinkel 2002; Heritage 1984; Lynch 1999). This paper is under particular influence from the studies of epistemological practices in laboratory settings (Latour and Woolgar 1986; Lynch 1985). In terms of ethnomethodology, most of the analytical props of this paper might be traced back to Michael Lynch’s work on “the ordinary” practices of science and the use of commonly used categories of activities as epistemic topics (Lynch 1997). This paper ought to be seen as subscribing to a similar set of ambitions. Armed with more complete knowledge about the practices of programmers *and* (and *as*) designers, radically improved methodologies and tools for such work may be proposed. That, however, is outside ethnomethodology’s scope.

3 RESULTS

In the following sections, transcripts from field notes will be presented. The excerpts are in themselves “data” rather than a precise and positive representation of what “actually” happened and should be treated with caution just because of that. This is of course not unique to this data set in particular, and other ethnographers have made similar remarks (Lynch 1985). Most of the programming activities take place in one large room; four of the programmers have their desks in this room. The manager has a room adjacent to this, and there is another room with two programmers working next to the first room. Conversations between the programmers may generally be heard across the floor, and it often initiates “off the cuff” meetings which comprise discussions like those typically referred to below. A similar workspace of one room is provided on the floor below. The most prominent aspect of practical problem solving between programmers is how far the problem is from actually being solved in just about any popular or formal understanding of that notion. Consider the following examples:

Two programmers are in a discussion about the layout of a screen with a menu:

1: *(We could)² change between white and blue background there, but that is not the most important*

2: *There are two menus*

1: *That is one thing*

2: *That*

1: *No, no, no, yes, yes, yes,...*

2: *A parallel line (just) there*

So, what we see here is a design exercise, rather than simply programming a design. Image the strongly two-dimensional and limited display of a mobile phone. What is a menu? These (particular) devices do not have rich component libraries and 3D menus (although most mobile phones now have them). A menu is, in terms of its design, what the programmers makes out to be a menu from they lines that they get the graphics library to draw on-screen.

Design is usually considered to be a goal-oriented activity, and this is made explicit in many tools which support co-operative aspects of design (Conklin and Yakemovic 1991). But that is not what we observe. “Design” seems to start by working out and suggesting *descriptions*, rather than solutions. Moreover, it is hard to see any *problem* driving the dialogue. Actually, it is hard to see how somebody can, at all, understand from the beginning what exactly is being brought onto the design “agenda.” And this is exactly the point. Design is a lot about prioritising, figuring out what to do when, and eventually, how. From then on, it is the programming. This skilful working towards a problem definition “from the solution out” characterizes many of the instances of co-operative design observed in this study.

For instance the following excerpt represents a situation that starts when the manager (a trained programmer himself) comes back from the kitchen with cup of coffee. Walking through the office, he spots one of the programmers working with a calendar application.

1: *That calendar application, you have not*

2: *Hmmm,...*

1: *Yes, yes, but, it can be the case that we should*

2: *Okay, but, it is a bit strange*

1: *No I have not*

² The excerpts have not been formally marked up, and the paper does not pertain to be based on conversational analysis or any formal “grounding” of theories from data. Numbering distinguishes different people *within* excerpts, but do not identify individuals as such across them. (An explication in parenthesis) means that the field workers has filled in “blanks”, which could be a result of unintelligible conversation, poor note taking or simply that participants did not speak a complete sentence. When the next line is tabbed, there has been a perceived overlap of speech, if not the turn- taking was judged to be sequential. Ellipses means that the utterance has been drawn out ((Double parenthesis)) are used for ‘meta-comments and explanations.

2: Yes

1: Yes, well, but could it not be the case that we ought to, no! That was a little, 2: is it not a bit strange “user-wise” (when) you switch between the days, and then you switch?

1: Go up there ((points to the screen)), and then you can show, ..., do you have any options as well, ... okay, that’s fine

(starts with no introduction to speak about the videophone application that they are implementing) okay, but then take the picture

But then you can show it directly, come down here ((on the screen)),

And then one down and so, hello-o!

()

Yes, video and call control and (then) time (it) and such, yes that would have been really nice.

Here we see a fascinating disintegration of borders between two apparently very separate design concerns, namely that of calendar functionality of moving between the day and month views, and the event-handling mechanisms that make it possible to switch between the various applications. The latter is similar to re-focusing on a different application on the desktop by pointing to it using a mouse. The reason for the “switch” to talk about and assess the design of the videophone-functionality was the “strangeness” of the way user views on the calendar changed from *day* to *month*.

The most striking aspect of this conversation is the way in which options are “explored” in a way that is completely underdetermined, to the extent that it does not at first seem fitting simply to describe it using simply the term “indexical”. One has to start by asking, again, not only how do *we* know what they are talking about, but (how) do *they* know what they are talking about? We find some analytical bolstering in the description of tendentious use of instructions referred to by Garfinkel:

By tendentious, I mean that I want to talk about [instructions] with an abiding and hidden tendency. More, I am going to know what I’m talking about long before I reveal to you just what I am talking about. And I would hope that finally we’ll come to a point, after I’ve gone through some materials, that you will have seen, with some surprise, that in fact this thing that I had in mind all along, being in the end revealed, is in fact what you had in mind all along as well, except that I had a rival version of what you might have in mind, and more that that: you would see that I mean to be talking to what you really mean, before you’re able to say it, and that I mean to be talking to the point of a revealing corrective of what you might have in mind, and even a radical corrective (Garfinkel 2002).

But how do programmers “find out” what they do not know that they are looking for in an artefact that so effectively hides everything that is not currently displayed? It seems that they point into a possible solution space by suggesting (im)possible operations and manoeuvres with curiosity and stubbornness, keeping their specification open enough for it to be correctable. This goes on until the programmer who has implemented the module in question confirms that “this is really what we have been talking about all the time.”

We shall see another example of this, in which the work need to connect an application to the underlying hardware architecture seems intentionally to be left “underdetermined”:

1: The device layer is on top,...

2: Hmm, that is something that we can

1: But, come on, it is possible to make this real easy,...

2: We could simply implement it.

1: As long as it is an object, you don’t have to go ((no indication of where?!)).

(...)

2: Okay, but what he ((don’t know if this is the programmer or the code)) does not have is a, what’s it now, a directory scan

1: He ha ho ((laughing))

2: But we need to get that sorted

1: Yeah, right, that,...

2: Great stuff

1: *Fair enough*

2: *No, that was kind of just a “bulletpoint” ((assume figurative speech)), not a, you know, general, mucking about*

1: *No, No-o,...*

2: *That won't be a problem, really*

1: *Jolly good*

2: *Yes, yes*

The excerpt starts with a proposition which in an “architectural sense” seems either false or obvious, namely that “The device layer is on top,…” since the idea of the device layer roughly speaking is to be an abstraction sitting right on top the hardware. So either it is *not* “on top” of the application because that defies the purpose (and would be really “poor design”), or it is clearly *on* top of the hardware (but only figuratively speaking, of course, since a real time OS and basic input/output is on top of the machine code which is on top of the digital circuits, which are abstraction of physical wiring, etc). So, what do we see happening here? It seems that programmers intentionally furnish vague and vernacular descriptions about the latent artefact that they are working on, ostensibly suggesting what it is like currently and what it could be made to be, whilst leaving room for interpretation and re-action from the other party.

The following excerpt shows how tightly knit the projection of “design” and “implementation” is in the discussion between the programmers. It is vital that they get to a (perceived) common understanding of how everything is done so far, and to which extent the future design options must be limited by the overarching concern of not having *to do that work again*. The concrete implications of getting code to actually *execute* in a “stubborn” environment of device drivers and laboratory hardware within the deadline, *have to be brought to bear on* otherwise proudly “techie” mottos, which show deeper respect, on the surface, as it were, for clear cut designs, rational problem solving and general solutions. In practice programmers display a practical acknowledgement of starting from the ambition of making “beautifully designed” code and ending up with “working code.” Accounting for code “as is” in a fitting way is part and parcel of “design” and is not in any way heralded by it. In this excerpt the implementation on a pixel-by-pixel level is connected with its own design. It starts by “connecting with” how it is today, but carefully not really specifying that (either) in much detail.

1: *It is the way that it is done today*

2: *Yes, I agree, but we cannot just let the focus go*

1: *And then all the menus, well, that come this way, they can just render as if they are the root ((of a hierarchy of screens)), or be in the root, and so,...*

2: *That i-is the way that it is implemented today*

mmm, in,...

fact

1: *and then that key goes up to the root*

2: *Okay, that is the way the, ..., that it is implemented now*

The quoted discussion goes on like this, strikingly in harmony with ideas from ethnomethodology about accountability being a constitutive element in the action which is (then) accounted for. It seems that a “hack” is acceptable if it is unavoidable, but that is not the most important point, which is that it *becomes* an acceptable hack by accounting for it as such. It certainly did not become an acceptable hack “by design,” because the coding that comprised the hack had already been done and designating it as a hack conspicuously made it possible to keep it, for now.

1: *So, if we can implement implement ((repeats)) the keys this way, then we can keep it in the wrong,...*

(...)
Then we can simply keep it, internally,...

((interrupting himself?))

yes, yes, it has to be called something else, then

2: *The only point then, just that thing is a little hack, okay?*

I mean, it is okay, in one way

1: *If you are rendering menus on top of each other, there is not reason that they shall need to consider that issue, of things being behind each other*

2: *The only aspect that we're dependant on is if we are going to light and put out that one there ((points))*

1: *Eh, yes, that that should not*

2: *No, sure, I agree*

Only thing is that we need a focus manager

So, what we see is that some elements can be kept as an acceptable (and even admirable?) hack simply by renaming it, which also means that it architecturally can be kept in the "wrong" module. This is one example of design (modularization of functionalities) and implementation (naming), being tied together. Even more expressive, the discussion continues to make room for a particular type of dialog design for the screen of a mobile phone, the menus, to take on a new role as the (underlying) focus manager. This is observably a retrospective fit of existing code to an emerging design, which is coming out of requirements that are only now being firmly established; that a separate focus manager is needed. One succinct way of putting it is that coding is at this very moment producing a rationale for itself in terms of a "requirement," namely the requirement of having to comprise a focus manger. This requirement was not there before the programmers started talking about the code as part of their examination of the code together. The possibility of the menu system working as a focus manager was not in question when the menus were designed. It is unlikely that mixing such unrelated functionalities within one designated module would be passable as a quality design, anyway, had it been proposed. Now it "comes in handy," and it is co-opted into its new role on the backdrop of a continuous rehearsal of "what the code is like" today.

2: *Eh, yes, that menu is (of course) some sort of focus manager, as long as you ((as a user)) are navigating in the menu (there), then it is like a small focus manager, for those ((screens)) that are beneath (it)*

1: *Yes, yes,*

()

Dialogue windows that,...

2: *And that is okay, actually*

1: *I still think that we have to keep in mind that a menu is something special*

2: *Yeah*

and that other dialogues are different

1: *And that dialogues that can get the focus from an application is one thing and that a menu is something else*

2: *((With emphasis, louder:)) What kinds of dialogue windows ((orig. "screens," which would technically translate into display for most GUI programmers)); is it that has a need to let it ((the key press)) leak through to other windows?*

1: *(Well, I think that we have,...)*

Yes, yes

1: *You see, the point is, isn't it, that it comes (it comes) the problem comes back again one day if we are going to implement something that we do not have today and that we will probably not have in a long time, because we're making a...*

2: *Yes*

1: *()*

and if you have a S-view ((from a previous phone GUI-project)), or something like that, then you have to, it is after all the input box which needs to receive the keys, but if you push it down ((Not the key itself? It is a touch screen, but more likely the input box 'down' into the hierarchy in terms of 'design')) then it ought to leak through unless we're talking about the browser and then focus needs to be shifted onto the one that is

2: *And by implementing it in the fashion that we've been talking about just now*

1: *Now we are ready to do just that*

Drawing again our attention to the underdetermined and indexical aspects of the design, the excerpt above shows that design is not at all stipulating of what is implemented, but rather yielding to “the requirements of working code” when a design is being truly co-constructed in order to make the code passable as sufficiently “good” within requirements, standards, company kudos, etc. The code is where it is, and the “input box” (whatever that is) is already contained in a module somewhere. “If you push it down” does not imply that it can be pushed further down or lifted higher up, although, of course, the code can be re-factored according to some plan relying on such terms. Similarly, it does not matter (for the object) if it “ought” to let key presses “leak through” its even handler to another module. It either leaks or it does not. That is not the (main) point, which is: The code can be read with the essential property of “leaking through” from the current input box, from where it is, as meaningfully, justifiably and understandably complying with a design in which it is “pushed down” and this reading makes that particular design in itself an expression of an order which is meaningful, justified and understandable.

In exactly the same way the documentation that exist is referred to as either “coming into” fit with the “code in working order,” or as documentation that is no longer accurate:

1: *But, then, the documentation that we have is not too bad, then. I gather? It might end up having to be changed a tad, but*

*Yes, we are very confident that this is the way that it will be lifted
it just sort of*

2: *But, we’re ought to get around to concluding,...*

Do you ever need more than the parent

1: *(But, but,...)*

Yes, but that does not have anything to do with this?

3: *Maybe you do not really need it, but it is sometimes nice to be able to ask if you have got focus*

4: *The way that we are doing this today, then,...*

1: *The way that we are doing it today is how we do it*

I think, I think, ..., halo-o! folks, defining the focus, that is, using what we have today,

That is what seems to work

And then we can use the focus manager

2: *It is so simple, what we have today, because it just works.*

Yes, yes

I really suggest that we do not change too many things at once. Then suddenly it stops working and, as (we) have said, defining focus, that is an implementation issue

Having demonstrated in many excerpts how a compiled and running implementation of a program, subjected “design” to continuous recasting (rather than the other way around!), it is worthwhile to consider how in this last excerpt *first* the documentation is in a similar way “retrofitted” to match “how we do it today” and *then* at the end it is actively underspecified to the extent that the team can wait and see what comes out of the next few hours of programming before they sit down and look at this again. As a “design” issue, the implementation of “focus” was meticulously scrutinized to devise a matching design for it. Re-classifying it as an implementation issue happens when it is stated that; “as (we) have said, defining focus, that is an implementation issue,” because before that happens it has, to the contrary, been treated as a design problem. One could say that this re-classification now leaves the problem back with the responsible programmer to re-design and implement until it is time (again) to bring the topic of the focus manager into the social domain of design. From an analytical point of view, we could say that referring a design chore to another epistemic category of implementation itself becomes a way of underdetermining code to the extent that it is design-wise dealt with, “for now.”

The implicit treatment of an artefact as “in the need of” one type of epistemic attention or the other, as exemplified in these excerpts with reference to design, implementation and documentation, are not guided by the normal and normative framework of methods within software engineering. In this respect, there is a clear difference between the rules that extend from software engineering, and their practical achievement. Programmers rely on rules and methods to do design, implementation and

documentation activities. They perform them in a network of actions and treat one as if it was of a separate category from another if, indeed, it is recognizable as such:

One knows whether one has “instructably reproduced” an action by its “praxeological validity,” whether the action works, or has coherence for others, not by whether one has followed rules. If the action “works,” that is, has coherence and recognisability and is mutually intelligible to others, then the instructed action has praxeological validity (Garfinkel 2002).

The observations of this paper lie within this analytical thrust, but the main contribution that comes out of from this fieldwork is that the coherent actions “which work for others” under the epistemic headings of software development, such as “design” and “implementation,” are different from how they are specified within the software engineering metaphor. Especially, it is hard to see how they can be supported by tools purporting to make goal achievement more effective, notwithstanding that goals and the actions directed towards fulfilling them are tendentially underspecified as well and “made known” by opening up the number of alternatives, rather than closing them down.

4 DISCUSSION OF THE RESULTS

Before looking into the academic implications of this research with regards to design within Information Systems Development, one has of course to discuss what is entailed by the concept of *design* in the relevant contexts. HCI (Human-Computer Interaction) is seen by many as the science of design (Carroll 1997), and it is used here as one point of departure for the remainder of the discussion. This tradition somehow represents the cognitive perspective towards design. One important ambition of the design science of HCI was to orient developers towards “the characteristics of human beings (Carroll 1997).” It, moreover, saw psychology (in particular) as a discipline that could be called upon to develop general models of how people interacted with computers, and wanted to apply such models prescriptively as a framework for design. Another, more process-oriented perspective on design, is represented by the strand of work concerned with implementing a structured approach to development building on top-down approaches, well-documented and verified progress hand-off between stages and precise metrics (Royce 1987). Much critique has been raised against the waterfall-model of software development (McCracken and Jackson 1982). It has also been forcefully defended (Curtis, Krasner, Shen and Iscoe 1987), and even today most “modern” life-cycle models (and reflexion upon them) really rely on some of the same philosophies (Guimarães and Vilela 2005). Even radical revisions of the life-cycle idea itself, is often seen as encompassing some of the same process-structural assumptions (Davis 1992). The findings presented above indicate that this is quite problematic. HCI contributed to point out the discrepancies between the organization of work in a “waterfall model” and design:

Design work is frequently piecemeal, concrete, and iterative. Designers may work on a single requirement at a time, embody it in a scenario of user interaction to understand it, reason about and develop a partial solution to address it, and then test the partial solution— all quite tentatively— before moving on to consider other requirements. During this process, they sometimes radically reformulate the fundamental goals and constraints of the problem. Rather than a chaos, it is a highly involuted, highly structured process of problem discovery and clarification in the context of unbounded complexity (...) (Carroll 1997).

This was at the time a radical critique, but the findings of this paper indicate that perhaps it did not go far enough in many respects, because it still documented the design process as starting from requirements, rather than seeing it also as a process which *produces* requirements. Similarly, although looking at it with much finer granularity, it still conceived design as governmental to programming, e.g., by seeing programming as a way of testing if the design is, indeed, correct, rather than *the other way around*.

The applications of cognitive user models have had a prominent place in HCI (Gong and Kieras 1994; John and Kieras 1996; Sutcliffe 2000), and although the interaction models tradition, initiated by

Hutchins et al (Hutchins, Hollan and Norman 1985) and nicely documented by e.g., Fisher (Fischer 2001), has been somewhat criticized for not taking social aspects into account (Moran and Anderson 1990), it is still driving interesting research in this area (Beaudouin-Lafon 2000; McTear 1993). From the observations reported here, there seems to be little support for extending such modelling to the domain of CSCW and put that down as a foundation of future design tools. Storey et al. suggest on the basis of cognitive models of program comprehension, that a tool for programmers ought to comprise at least functionality to support goal-driven examination, an overview of the architecture, the construction of multiple mental models, and indicate focus path (Storey, Fracchia and Muller 1997). However, these are not activities in which programmers seem to be particularly engaged, *in order to do programming*. Some of these they, indeed, resourcefully and systematically avoid pinning down in order for them to maintain an “open design space” of hints and clues which in a tendentious manner allows a common design to be established that “with some surprise“ is revealed as in fact what they had in mind all along (Garfinkel 2002). Jahnke and Wahlenstein maintain that tools for software reverse engineering lean toward requiring a precise, complete and consistent knowledge representation (Jahnke, Walenstein and 31 2000); they therefore recommend enforcing predefined process templates. This would rather seem to limit their usefulness, on the background of the excerpts discussed in this paper. Exogenous problem descriptions, background competencies and steps in the procedure towards resolving “the” problem is *exactly* what the programmers in our case did not use, as part of their co-operative design work. Indeed, this is perhaps the most explicit and radical new contribution of this paper towards showing that “social design” as part of programming might be computer-*unsupportable*. The common explication of knowledge and specification of a problem-solving procedure thus needed, e.g., recommended not only by Jahnke and Wahlenstein, but similarly by innovators in design rationale systems such as gIBIS (Conklin and Begeman 1988) and expert systems (cf. AnswerGarden (Ackerman and McDonald 1996)) seem to represent, namely, the opposite of what the programmers struggle “methodologically” to be doing.

Programming might be seen as exactly a social activity which produces, rather than is subjected to, the production of, motivations, requirements, features and design trade-offs. Thus, it might be seen as not entirely trivial to claim that a computer system is not its own elucidation (Carroll 1997). Perhaps there is no other, better representation of the computer system’s “design; the user requirements it was intended to address; the discussions, debates, and negotiations that determined its organization; the reasons for its particular features; the reasons against features it does not have; the weighing of trade-offs; and so forth (Carroll 1997),” than exactly the computer system itself.

The practical implications are formidable: Programming should not (or cannot) be seen as a *result* of design, which ideally and productively can be mapped from a set of requirements. This does not mean that design does not happen. There is always a need to do design work, and probably an approximately equal type and amount of design work, regardless of life-cycle or documentation techniques chosen. The dominant hypothesis for improved methods and tool support programmers seem to be that removing design from programming simply moves it “upstream” and that can make it more transparent and more manageable (Humphrey 1988). The alternative hypothesis, indicated by this paper, would be that it can *only* be performed “as is” because it is a constitutive element of programming itself. Programming can partly be seen as *the* design rationale. This paper is therefore also an indication of the need to readdress the “taken-for-granted” functionalist character of many theoretical perspectives in software engineering, CSCW and HCI. They are manifest in the waterfall life-cycle model as well as the cognitive psychologists’ model of interaction, as well as in ideas of a design rationale. The research presented in this paper suggests that these traditions need to be complemented by research looking at how programming is independently constituted *from within* by a wider variety of epistemic categories. Since information systems development ultimately has to be concerned with supporting the work of programmers, a deeper understanding of what programming is really about seems to be a necessary prerequisite to further progress in this field as well.

References

- Ackerman, M.S. and McDonald, D.W. Answer Garden 2: merging organizational memory with collaborative help, Proceedings of the 1996 ACM conference on Computer supported cooperative work, ACM Press, Boston, Massachusetts, United States, 1996, pp. 97-105.
- Beaudouin-Lafon, M. Instrumental interaction: an interaction model for designing post-WIMP user interfaces, Proceedings of the SIGCHI conference on Human factors in computing systems, ACM Press, The Hague, The Netherlands, 2000, pp. 446-453.
- Beynon-Davies, P. Ethnography and information systems development: Ethnography of, for and within is development, Information and software technology (39:8) 1997, pp 531.
- Boehm, B.W. and Papaccio, P.N. Understanding and Controlling Software Costs, IEEE Transactions on Software Engineering (14:10), Oct 1988, pp 1462-1477.
- Brooks, F.P. No silver bullet: essence and accidents of software engineering, Computer (20:4) 1987, pp 10-19.
- Button, G. and Dourish, P. Technomethodology: paradoxes and possibilities, Proceedings of the SIGCHI conference on Human factors in computing systems: common ground, ACM Press, Vancouver, British Columbia, Canada, 1996, pp. 19-26.
- Carroll, J.M. Human-computer interaction: Psychology as a science of design, Annual Review Of Psychology (48) 1997, pp 61-83.
- Clases, C. and Wehner, T. Steps Across the Border - Cooperation, Knowledge Production and Systems Design, Computer Supported Cooperative Work (CSCW) (11:1 - 2) 2002, pp 39.
- Conklin, E.J. and Yakemovic, K.C.B. A Process-Oriented Approach to Design Rationale, Human-Computer Interaction (6:3&4) 1991, pp 357-391.
- Conklin, J. and Begeman, M.L. gIBIS: a hypertext tool for exploratory policy discussion, Proceedings of the 1988 ACM conference on Computer-supported cooperative work, ACM Press, Portland, Oregon, United States, 1988, pp. 140-152.
- Cooper, A. The inmates are running the asylum, or, why high technology products drive us crazy and how to restore the sanity., (Second ed.) Sams, Indianapolis, 2004, p. 288.
- Curtis, W., Krasner, H., Shen, V. and Iscoe, N. On building software process models under the lamppost IEEE Computer Society Press, Monterey, California, United States, 1987, pp. 96.
- Davis, A.M. Operational prototyping: a new development approach, Software, IEEE (9:5), September 1992, pp 70-78.
- Denning, P.J. The field of programmers myth, Commun. ACM (47:7) 2004, pp 15-20.
- Fischer, G. User Modeling in Human-Computer Interaction, User Modeling and User-Adapted Interaction (11:1 - 2) 2001, pp 65.
- Forsythe, D.E. It's Just a Matter of Common Sense: Ethnography as Invisible Work, Comput. Supported Coop. Work (8:1-2) 1999, pp 127-145.
- Garfinkel, H. Studies in ethnomethodology Prentice-Hall, Englewood Cliffs, NJ, 1967.
- Garfinkel, H. Ethnomethodology's Program. Working out Durkheim's Aphorism Rowman & Littlefield Publishers, Inc., Lanham, Maryland, 2002, p. 297.
- Gong, R. and Kieras, D. A validation of the GOMS model methodology in the development of a specialized, commercial software application, Proceedings of the SIGCHI conference on Human factors in computing systems: celebrating interdependence, ACM Press, Boston, Massachusetts, United States, 1994, pp. 351-357.
- Grinter, R.E. Recomposition: putting it all back together again, Proceedings of the 1998 ACM conference on Computer supported cooperative work, ACM Press, Seattle, Washington, United States, 1998, pp. 393-402.
- Grinter, R.E. Systems architecture: product designing and social engineering, Proceedings of the international joint conference on Work activities coordination and collaboration, ACM Press, San Francisco, California, United States, 1999, pp. 11-18.
- Guimarães, L.R. and Vilela, P.R.S. Comparing software development models using CDM, Proceedings of the 6th conference on Information technology education, ACM Press, Newark, NJ, USA, 2005, pp. 339-347.

- Harper, R.H.R. Radicalism, beliefs and hidden agendas, *CSCW* (3:1) 1995, pp 43-46.
- Heritage, J. Garfinkel and Ethnomethodology Polity Press, Cambridge, 1984.
- Hughes, J., King, V., Rodden, T. and Andersen, H. Moving out from the control room: ethnography in system design, Proceedings of the 1994 ACM conference on Computer supported cooperative work, ACM Press, Chapel Hill, North Carolina, United States, 1994, pp. 429-439.
- Hughes, J.A., Randall, D. and Shapiro, D. Faltering from ethnography to design, Proceedings of the 1992 ACM conference on Computer-supported cooperative work, ACM Press, Toronto, Ontario, Canada, 1992, pp. 115-122.
- Humphrey, W.S. Characterizing the Software Process: A Maturity Framework, *IEEE Softw.* (5:2) 1988, pp 73-79.
- Hutchins, E.L., Hollan, J.D. and Norman, D.A. Direct Manipulation Interfaces, *Human-Computer Interaction* (1:4) 1985, pp 311-338.
- Jahnke, J.H., Walenstein, A. and 31, P.s.-. Reverse engineering tools as media for imperfect knowledge, Seventh Working Conference on Reverse Engineering, 2000, pp. 22-31.
- John, B.E. and Kieras, D.E. The GOMS family of user interface analysis techniques: comparison and contrast, *ACM Trans. Comput.-Hum. Interact.* (3:4) 1996, pp 320-351.
- Kemerer, C.F. Progress, obstacles, and opportunities in software engineering economics, *Commun. ACM* (41:8) 1998, pp 63-66.
- Kraut, R.E. and Streeter, L.A. Coordination in software development, *Commun. ACM* (38:3) 1995, pp 69-81.
- Latour, B. and Woolgar, S. *Laboratory Life: The Construction of Scientific Facts* Princeton University Press, 1986.
- Lynch, M. *Art and Artefact in Laboratory Science: A Study of Shop Work and Shop Talk in a Research Laboratory.* Routledge and Kegan Paul, London, 1985.
- Lynch, M. *Scientific Practice and Ordinary Action: Ethnomethodology and Social Studies of Science* Cambridge University Press, New York, 1997.
- Lynch, M. Silence in context: Ethnomethodology and social theory, *Human Studies* (22:2 - 4) 1999, pp 211.
- McCracken, D.D. and Jackson, M.A. Life cycle concept considered harmful, *SIGSOFT Softw. Eng. Notes* (7:2) 1982, pp 29-32.
- McTear, M.F. User modelling for adaptive computer systems: a survey of recent developments, *Artificial Intelligence Review* (7:3 - 4) 1993, pp 157.
- Moran, T.P. and Anderson, R.J. *The workaday world as a paradigm for CSCW design* ACM Press, Los Angeles, California, United States, 1990, pp. 381-393.
- Rosson, M.B., Kellogg, W. and Maass, S. The designer as user: building requirements for design tools from design practice, *Commun. ACM* (31:11) 1988, pp 1288-1298.
- Royce, W.W. Managing the development of large software systems: concepts and techniques, Proceedings of the 9th international conference on Software Engineering, IEEE Computer Society Press, Monterey, California, United States, 1987, pp. 328-338.
- Sakthivel, S. Virtual workgroups in offshore systems development, *Information & Software Technology* (47:5) 2005, pp 305-318.
- Shapiro, D. The limits of ethnography: combining social sciences for CSCW, Proceedings of the 1994 ACM conference on Computer supported cooperative work, ACM Press, Chapel Hill, North Carolina, United States, 1994, pp. 417-428.
- Sharrock, W. and Randall, D. Ethnography, ethnomethodology and the problem of generalisation in design, *Eur. J. Inf. Syst.* (13:3) 2004, pp 186-194.
- Storey, M.-A.D., Fracchia, F.D. and Muller, H.A. Cognitive design elements to support the construction of a mental model during software visualization, Fifth International Workshop on Program Comprehension. IWPC '97., Dearborn, MI, USA, 1997, pp. 17-28.
- Sutcliffe, A. On the effective use and reuse of HCI knowledge, *ACM Trans. Comput.-Hum. Interact.* (7:2) 2000, pp 197-221.