Designing a program. Programming the design.

Steinar Kristoffersen
steinkri@ifi.uio.no
University of Oslo
Norway

ABSTRACT

For most rational systems development methods, distinct phases of construction work is envisaged: Analysis, design, programming, testing, etc. Different methods are often contrasted to each other with regards to the amount of design that is required before implementation, the importance for testing continually or the prominence of future users' involvement. However, little evidence can be found that one method is actually better than the others. This paper is concerned with design activities within programming. It presents an ongoing study of programmers developing software for mobile telephones, and in particular it looks at the ways in which a design is "re-fabricated" as part of the common practices of programming. Such re-fabrication is social and under-determined. Knowing more about how design and programming is inextricably linked with a set of rather vague instructions, practically explicates problems and possibilities of distributed programming projects, e.g., in an outsourced setting. In the theoretical domain, this paper shows how well ethnomethodologically-informed notions of "epistopics" (of design) and "tendentious instructions" apply to programming as a scientific as well as deeply social undertaking.

Keywords

Ethnographic studies of programming, design as epistopic category, tendentious talk and under-determination.

1. INTRODUCTION

Creating the software is arguably the most costly part of information technology development projects [1]. This is not due to programming alone, however:

[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 [2].

This quote can be taken as inspiration to ask: Do these activities contribute to the executable result at all, notwithstanding the aspect that they do not contribute directly to create code? And vice versa, what do the non-coding activities of programming look like (with reference to their social achievements and output in terms of documentation, plans, design rationale, etc.)?

Many properties of the envisaged IT development projects are highly similar across software engineering methods and life-cycle models [3]: 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. This design is an artifact 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 several levels of abstraction is one way of managing the complexity of software projects. One typically moves from requirements via some form of conceptual design, a technical- or functional design and the software architecture. A chronologically ordered sequence of functional activities comprises a life-cycle for the project, which drives it forward through such horizontal levels of abstraction or vertically compartmentalized designs, towards completion [4]. Different and distinguishable competencies ought to be brought to bear on each of the activities, one of which is, notably, design.

On the background of the abysmal results coming out of almost 50 years of systematically attempting to se software development as a software engineering activity [5, 6], this paper challenges this view. It is particularly motivated by the recognition that many software engineering problems arise, arguably, due to factors which are not covered by the methods. "the thin spread of application domain knowledge, fluctuating and conflicting requirements, and communication bottlenecks and breakdowns [7]."

This paper 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 or on another level of abstraction. It is, and can only be, performed by programmers as part of making a “cautious” set of specifications, tendentious instructions [8] 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 of these observations is that software engineering ought not to be treated primarily as a stepwise refinement of a set of requirements’ specifications from one level of abstraction to the next, or one phase accumulating the results of the previous one, at least not solely or prescriptively, inasmuch as this perspective does not stipulate how the programming takes place. Instead, a complete respecification of “programming” as a genuine form of co-operative work is 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). 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 epistopic theme [9], 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 [8], 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 view on design can compose an “epistopic” for ethnographic as well as software engineering research, to the extent that one fruitfully sees programming more broadly as constituting ways of creating knowledge, technically as well as socially, about the problem at hand, instead of simply an applications of “boxed” skills. Thus, it is also an application of theories from social studies of science to the “more” ordinary activities of programming.

2. RESEARCH SETTING AND METHODS

2.1 The setting

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.

2.2 The ethnography

The research presented here is based on approximately five full days of observation taking field notes and recording a small number of samples from conversations taking place. The project is still ongoing, and more material will continue to be compiled. The excerpts presented here, however, 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 [10] and have become commonplace in related fields such as Information Systems (IS) and software technology research lately [11]. As its popularity continues to grow [12], it has perhaps become easy to fall into the trap of believing that ethnography is straightforward, which it clearly is not [13]. This paper is not seeing ethnography as simply a suitable approach to data collection; it is driven by e*thnomethodology* as its analytical frame of reference [14] and for serving that ambition ethnography seems particularly suited. The "unique adequacy requirement" of ethnomethodological analysis certainly points toward needing detailed empirical accounts [15], from which a minimal level of detail needs to be excerpted in a paper such as this. 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 discussions can be found in the CSCW literature [16-20] and beyond [8, 14, 15, 21]. This paper is under particular influence from the studies of epistemological practices in laboratory settings [22, 23]. In terms of ethnomethodology, most of the analytical support 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 [9].

A methodological ambition of this paper has been to show how the practical achievements of design observably offer themselves as a starting point for a necessary respecification of "design" and "programming" in software development. The fieldwork excerpts are examples, a "real-life" flavor of programming as and when it unfolds. Abstract concepts such as "design" and "programming" (rather than the practices of designing or programming) and the relationships between them are not seen as functionally or structurally governing the behavior programmers, in this case.

In traditional software engineering, the concept of a process life-cycle comprised by distinguishable stages of "design" and programming" which so often fails to "deploy" properly in software engineering, seems to have maintained its stability and "meaning" through theoretical reflection rather than empirical inspection [24][1]. Instead, the methodology of this paper has guided it towards providing empirical evidence of the production of such "orderly properties," as and when programmers discuss their code.

Ethnomethodology is not concerned with creating big theories about external "normative" structures that (may or may not) stipulate behavior; rather, it sees such "theories" as part-and-parcel of the production of social order which all members of a community is engaged in, through and for the sake of "getting the job done". However, that should not be taken as an indication of ethnomethodology only yielding conservative or uncommitting results. Its position is certainly one from which poorly formulated theories about social activities can be successfully identified, attacked and respecified. This paper ought to bee seen as subscribing to such ambitions.

3. RESULTS

In the following sections, transcripts from field notes will be presented. Perusing the excerpts, the reader one might find it useful to be aware, however, that the author doing the fieldwork of this paper is a trained programmer and has been working as a programmer for many years. Some might think that programming thus appears different and perhaps more orderly or understandable to this fieldworker than to others who are not themselves programmers. One of the main implications of the analysis if the material which follows is that this does not seem to be the case: Not because everybody can do programming, but because the observable acts of programming lend themselves are social rather than technical. "Lived programming" consists of a discourse which seems, word by word, just as intelligible (or perhaps mainly exactly the opposite) to non-programmers as to programmers. Understanding the instructions and opinions that programmers exchange in discussion does not require great technical skills, however, and it might not even refer to an objectively distinct piece of code,

[1] An interesting observation in this respect is that there are actually several journals devoted to the empirical study of software engineering, e.g., *Empirical Software Engineering. An International Journal*. Editors-in-Chief: V.R. Basili; L.C. Briand. ISSN: 1382-3256 (print version). ISSN: 1573-7616 (electronic version). Journal no. 10664. Springer US.

functionality or requirement. Rather, it is about *creating* those entities as recognizable for all, with the capabilities that they have in terms of functionality. Requirements and design are *made* programming. Realizing the code that compiles and executes requires a specific skill, of course. Code has to be written and written correctly. Designing (for) it, on the other hand, does seem to be a different matter.

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 room(s), and it often initiates ad-lib 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 of 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

The notion of design as a precise and rational problem-solving exercise, which is prevalent within in software engineering is challenged in this sample. Similarly, the idea that programmers “divide and conquer” a problem either by partitioning it into manageable smaller problems or by stepwise refinement from higher to a lower level of abstract, which is in many ways the hallmark of software engineering [25], is not much present. The excerpts presented here do not reject that this is how programming as an individual activity, as practical *programming*, takes place; that is outside the scope of this investigation. On the other hand, there is something else entirely that may be observed when more than one programmer go about solving their software development problems *together*, and we need to bear in mind that *this* is the domain of most software engineering and its methods. Design is usually considered to be a highly goal-oriented activity, and this is made explicit in many tools which support co-operative aspects of design [26]. But that is not what we observe in our fieldwork. “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,..

[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.

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: 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

(switches 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.

The most striking aspect of this conversation is the way in which options are "explored" in a way that is completely under-determined, 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 do *we* know what they are talking about, but (how) do *they* know what they are talking about? Assuming that they are not trying to *eliminate* alternatives, which is what one would normally find in an idealized conceptualization of "design [27]," and instead that they are trying to establish the "fact" of what it is that they are looking at; of what it is that they are trying to achieve; this makes programming similar to the more general *tendentious* use of instructions described 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 [8].

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 impossible operations and manoeuvres with curiosity and stubbornness, keeping their "local-and-continuous" specification work open enough for it to be impossible for them to get it wrong, and this wait simply holding off until the programmer who has implemented the module in question, himself or herself proposes what it can be. We shall se another example of this, in which the work need to connect an application to the underlying hardware architecture seems intentionally to be left "under-determined":

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 level 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 happing here? It seems that programmers intentionally offer rather vague descriptions about the invisible parts of the 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 practical 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 deep respect for clear cut designs, rational problem solving and general solutions. This means that is not only recognized that although one usually starts 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" rather than design being a set of instruction for what and how to code. 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 seems to be in harmony with ideas from ethnomethodology about accountability being a constitutive element in the action which is (then) accounted for. It seems here 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; namely 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', which is the requirement of having to have a focus manger. This requirement was not there before programming 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 design. 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 dalogues 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 under-determined and indexical aspects of the design, these excerpts show that it 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, is subjected to "design" as a continuous recasting of its properties, 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 for it a matching design. 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 transferring a design job to another epistopic category of implementation, in itself becomes a way of "under-determining" inasmuch as it is design-wise dealt with, "for now," whilst still leaving room for the programmer to decide how to do it

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.

4. DISCUSSION OF THE RESULTS

Much critique has been raised against the waterfall-model of software development, which is implicated by the first strand of work mentioned here. [28]. It was forcefully defended, however [29], and even today most "modern" life-cycle models rely on some of the same philosophy [3]. Even radical revisions of the life-cycle idea itself, relies on some of the same assumptions:

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 (…) [30]."

This is interesting, because it still documents the design process as:

- Starting from requirements (rather than a process which *produces* requirements)

- A structured process (only with finer granularity), oriented cyclically towards resolving goals, (re)discovering problems and on the basis of that *qua* evaluation reformulating the goals.

Storey et al. suggest that on the basis of cognitive models of program comprehension, 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, indicate a path of focus and current focus [31], etc. 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" a design is revealed which is in fact what they had in mind all along [8].

Jahnke and Wahlenstein maintain that tools for software reverse engineering lean toward requiring a precise, complete and consistent knowledge representation [32]; they enforce predefined process templates. Although this intuitively limits their usefulness, on the background of the excerpts discussed in this paper it seems still quite open whether a turn towards tool support for refining "imperfect knowledge," would be much better. The concern with knowledge and the specification of a problem-solving procedure which is recommended not only by Jahnke and Wahlenstein, but similarly by innovators in design rationale systems such as gIBIS [33] and expert systems (cf. AnswerGarden [34]) seem to represent, indeed, the opposite of what the programmers struggle "methodologically" to be doing. Exogenous problem descriptions, background competencies and steps in the procedure towards resolving "the" problem is *exactly* what the programmers in out case did not use, as part of their co-operative design work.

It has been claimed that computer systems do not display their rationale:

A computer system does not itself elucidate the motivations that initiated its 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 [30].

This is perhaps just stating the obvious, since these elucidations were never made clearly by programmers in the first place. Programming might (of course) be seen as exactly a social activity which *produces*, rather than is subject to the production of, motivations, requirements, features and design trade-offs. Moreover, and more radically, these "design artefacts" are skillfully and carefully under-determined and perhaps even made exactly "unspecifiable" enough, to work for programmers.

The practical implications are formidable: Programming cannot (or should not) be seen as a "result" of design, which ideally and productively could be mapped from "a" design. 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 [35]. The alternative hypothesis, indicated by this paper, would be that it can *only* be performed "as is" because it is a contituive element of programming itself. And, there is, in a sense nothing to "move". Design "works as design," but is not and does not produce "a" design. Taken out of its programming context, it becomes non-sensical.

Taking one step back, the biggest theoretical contribution from this paper is therefore perhaps only a formulation and thus a critique (admittedly from an ethnomethodological point of view) of the functionalist character of many theoretical perspectives in software engineering, CSCW and HCI. They are manifest in the waterfall life-cycle model and the cognitive psychologists' models of interaction [36], as well as in ideas of a design "rationale [26]." Numerous design support tools have hence been suggested, explicating the work programmers do as designers, i.e., by encouraging issues to be listed and prioritized, opinions made public and trade-offs weighted. This paper seems to suggest that this established tradition needs to be complemented by research looking at how programming relies on under-determined "matter" in the social web of programming, how opinions and alternatives seem to become manifest and shared *when-and-through* they emerge as manifest and shared by the programmers and consequently recognizing that there is room for much more radical re-specifications of the methodological aspects of software development.

5. ACKNOWLEDGMENTS

Thanks to the programmers who allowed their work to be the inspiration and immense source of inferences made within this paper.

6. REFERENCES

[1] G. Ooi and C. Soh, "Developing an activity-based costing approach for system development and implementation," *SIGMIS Database*, vol. 34, pp. 54-71, 2003.
[2] B. W. Boehm and P. N. Papaccio, "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering*, vol. 14, pp. 1462-1477, 1988.
[3] L. R. Guimarães and P. R. S. Vilela, *Comparing software development models using CDM*. Newark, NJ, USA: ACM Press, 2005.
[4] W. S. Humphrey and M. I. Kellner, *Software process modeling: principles of entity process models*. Pittsburgh, Pennsylvania, United States: ACM Press, 1989.
[5] P. Naur and B. Randell, "Software Engineering: Report of a conference sponsored by the NATO Science Committee," Brussels, Scientific Affairs Division, NATO (1969), Garmisch, Germany 7-11 Oct. 1968.
[6] B. Randell and J. N. Buxton, "Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee," Brussels, Scientific Affairs Division, NATO (1970), Rome, Italy 27-31 Oct. 1969.
[7] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Commun. ACM*, vol. 31, pp. 1268-1287, 1988.
[8] H. Garfinkel, *Ethnomethodology's Program. Working out Durkheim's Aphorism*. Lanham, Maryland: Rowman & Littlefield Publishers, Inc., 2002.
[9] M. Lynch, *Scientific Practice and Ordinary Action: Ethnomethodology and Social Studies of Science*. New York: Cambridge University Press, 1997.
[10] J. Hughes, V. King, T. Rodden, and H. Andersen, *Moving out from the control room: ethnography in system design*. Chapel Hill, North Carolina, United States: ACM Press, 1994.
[11] S. Sakthivel, "Virtual workgroups in offshore systems development," vol. 47, pp. 305, 2005.
[12] P. Beynon-Davies, "Ethnography and information systems development: Ethnography of, for and within is development," vol. 39, pp. 531, 1997.
[13] D. E. Forsythe, "It's Just a Matter of Common Sense: Ethnography as Invisible Work," *Comput. Supported Coop. Work*, vol. 8, pp. 127-145, 1999.
[14] H. Garfinkel, *Studies in ethnomethodology*. Englewood Cliffs, NJ: Prentice-Hall, 1967.
[15] M. Lynch, "Silence in context: Ethnomethodology and social theory," *Human Studies*, vol. 22, pp. 211, 1999.
[16] G. Button and P. Dourish, *Technomethodology: paradoxes and possibilities*. Vancouver, British Columbia, Canada: ACM Press, 1996.
[17] R. H. R. Harper, "Radicalism, beliefs and hidden agendas," *Comput. Supported Coop. Work*, vol. 3, pp. 43-46, 1995.
[18] J. A. Hughes, D. Randall, and D. Shapiro, *Faltering from ethnography to design*. Toronto, Ontario, Canada: ACM Press, 1992.
[19] D. Shapiro, *The limits of ethnography: combining social sciences for CSCW*. Chapel Hill, North Carolina, United States: ACM Press, 1994.
[20] W. Sharrock and D. Randall, "Ethnography, ethnomethodology and the problem of generalisation in design," *Eur. J. Inf. Syst.*, vol. 13, pp. 186-194, 2004.
[21] J. Heritage, *Garfinkel and Ethnomethodology*. Cambridge: Polity Press, 1984.
[22] M. Lynch, *Art and Artefact in Laboratory Science: A Study of Shop Work and Shop Talk in a Research Laboratory*. London: Routledge and Kegan Paul, 1985.
[23] B. Latour and S. Woolgar, *Laboratory Life: The Construction of Scientific Facts*: Princeton University Press, 1986.
[24] A. M. Davis, E. H. Bersoff, and E. R. Comer, "A strategy for comparing alternative software development life cycle models," *Software Engineering, IEEE Transactions on*, vol. 14, pp. 1453 - 1461, 1988.
[25] N. Wirth, "Program development by stepwise refinement," *Commun. ACM*, vol. 14, pp. 221-227, 1971.
[26] E. J. Conklin and K. C. B. Yakemovic, "A Process-Oriented Approach to Design Rationale," *Human-Computer Interaction*, vol. 6, pp. 357-391, 1991.
[27] F. M. Mehmet Aksit, "Deferring elimination of design alternatives in object-oriented methods," *Concurrency and Computation: Practice and Experience*, vol. 13, pp. 1247-1279, 2001.
[28] D. D. McCracken and M. A. Jackson, "Life cycle concept considered harmful," *SIGSOFT Softw. Eng. Notes*, vol. 7, pp. 29-32, 1982.
[29] W. Curtis, H. Krasner, V. Shen, and N. Iscoe, *On building software process models under the lamppost*. Monterey, California, United States: IEEE Computer Society Press, 1987.
[30] J. M. Carroll, "Human-computer interaction: Psychology as a science of design," *Annual Review Of Psychology*, vol. 48, pp. 61-83, 1997.
[31] M.-A. D. Storey, F. D. Fracchia, and H. A. Muller, "Cognitive design elements to support the construction of a mentalmodel during software visualization," presented at Fifth Iternational Workshop on Program Comprehension. IWPC '97., Dearborn, MI, USA, 1997.

[32] J. H. Jahnke, A. Walenstein, and P. s.-. 31, "Reverse engineering tools as media for imperfect knowledge," presented at Seventh Working Conference on Reverse Engineering, 2000.

[33] J. Conklin and M. L. Begeman, *gIBIS: a hypertext tool for exploratory policy discussion*. Portland, Oregon, United States: ACM Press, 1988.

[34] M. S. Ackerman and D. W. McDonald, *Answer Garden 2: merging organizational memory with collaborative help*. Boston, Massachusetts, United States: ACM Press, 1996.

[35] W. S. Humphrey, "Characterizing the Software Process: A Maturity Framework," *IEEE Softw.*, vol. 5, pp. 73-79, 1988.

[36] E. L. Hutchins, J. D. Hollan, and D. A. Norman, "Direct Manipulation Interfaces," *Human-Computer Interaction*, vol. 1, pp. 311-338, 1985.