Interpretation, interaction and reality construction in software engineering: An explanatory model

Kari Rönkkö

Blekinge Institute of Technology, School of Engineering, Box 520, SE-373 25 Ronneby, Sweden

Available online 13 February 2007

Abstract

The incorporation of social issues in software engineering is limited. Still, during the last 20 years the social element inherent in software development has been addressed in a number of publications that identified a lack of common concepts, models, and theories for discussing software development from this point of view. It has been suggested that we need to take interpretative and constructive views more seriously if we are to incorporate the social element in software engineering. Up till now we have lacked papers presenting 'simple' models explaining why. This article presents a model that helps us better to understand interpretation, interaction and reality construction from a natural language perspective. The concepts and categories following with the model provide a new frame of reference useful in software engineering research, teaching, and methods development.

© 2007 Elsevier B.V. All rights reserved.

Keywords: Social; Interpretation; Interaction; Indexicality; Communication; Software development; Natural language; Methods development; Management; Software engineering practice

1. Introduction

There is a growing acceptance that so-called ‘soft’ issues are a valuable component in software engineering. The social, behavioral and cultural dimension is recognized as part of the international scientific agenda in the field, exemplified by [11,24,1,7]. This is a difficult area in which to work as the incorporation of behavioral and people issues in software engineering research is limited, and it is not yet clear what the relevant literature is. This is exacerbated by the fact that the field is cross disciplinary in nature, meaning that useful prior work may appear outside the software engineering community’s usual channels. One indication of the status of the social challenge in our community was given by Fuggetta in his software engineering process road map celebrating the turn of the millennium [15]. He stated that the existing isolation must end, and asked for the results of process research performed in and by other communities. This article contributes to the social agenda by presenting a model that explains from a natural language point of view, how interpretation takes place, and discusses the consequences of this in relation to interaction and reality construction in software engineering practice.

The presented model the documentary method of interpretation (explained in detail in Section 2) is originally from sociology and provides an explanation of how people understand each other and also are able to investigate their daily world. It focuses on how the language of daily life is built up through indexical expressions. Indexicality here refers to the fact that words only take on a complete meaning in the context of their production. Words have an indexical relation to the circumstances in which they are uttered. This phenomenon becomes especially visible in expressions such as ‘you’, ‘I’, and ‘that’, as they undeniably draw their meaning from the interplay of people and the objects of interest under concrete conditions within a specific setting. The documentary method of interpretation demonstrates that actually all words are indexical, i.e., not only the obvious ones from linguistics that are exemplified above. The model also demonstrates how words get their
meaning from the two dimensions of indexicality: the situated context and what is known.

One recent software engineering study that might serve as example of indexicality is Smolander’s study of architectures in software organizations. In this study he identified ambiguous meanings of the concept architecture: architecture emerges as a plastic concept including diverging and simultaneous connotations for different stakeholders [29]. Four different usages of architecture were found: architecture as blueprint, architecture as literature, architecture as language, and architecture as decision [29]. Those different usages can be explained through applying the model that is demonstrated in this paper, the documentary method of interpretation.

In the context of software engineering, the documentary method of interpretation has served as an underlying assumption in the present author’s PhD thesis [26]. In this thesis the model heavily influenced the understanding of the social element in the studied software development practices, the applied research methodology, and the industrial methods development cooperation. In the Computer Supported Cooperative Work community the power of the model is very much present in studies where ethnomethodologically informed ethnography has been applied, but it is seldom discussed explicitly (examples of such software engineering studies are [6,28,27]). In the continuation of this article the documentary method of interpretation will be referred to as DMI.

An era starting in the mid 1970s and extending to the late 1980s saw the focus of discourses on professional work shift from technology itself towards its use [20, p. 6]. During these years, several now ‘classic’ papers were published (although rarely in the mainstream SE literature). Some are experience-based, and more explicitly include the relation to the social context to which software engineering methods are applied (e.g., [19,23,4,25,12]). All of these papers are still influential and can be found referenced in recent publications. The present author found that many of the papers are adequate in providing software engineering with a context of contingencies in which the need for a model explaining the social elements interpretation, interaction and reality construction can be understood, also from a historical point of view.

For the purposes of this article, the model will be discussed in relation to three of the above enumerated papers, although they do not explicitly direct their discourses to the social elements as discussed in the present article. From a product oriented view, Brooks’ paper No silver bullet – essence and accident of software engineering [4] provides powerful insights into the nature of the material that software engineers must manage in software development projects, i.e., the software. In relation to the model demonstrated in this paper, these insights help to explain why indexicality of words and texts is so hard to achieve and communicate in software development projects. Floyd’s paper Outline of a Paradigm Change in Software Engineering [12] requested that we move from a product oriented paradigm to a process oriented paradigm. From her interpretative and constructive view it is claimed that software practitioners do not analyze, apply or refer to requirements, methods, and implementations; instead they unavoidably construct their understanding of them (I have borrowed the concept ‘reality construction’ from her work [14]). In relation to DMI as presented in this paper, these statements are reminiscent of the epistemology that follows from it. DMI explains how Floyd’s claims are played out in practice by the practitioners. Up till now we have lacked papers presenting ‘simple’ models explaining why the interpretative and constructive view is needed within our discipline. The concepts and categories included in DMI provide new information that is useful in management, in the design of software methods, processes and process improvements. Floyd’s paper also reminds us of the historically slow development of the social element in relation to software engineering. Finally Naur’s paper Programming as Theory Building [23] made it painfully clear to us that exemplary resources in the form of material and available support are not enough when modifying others’ programs. In fact, if Floyd’s claims had been taken seriously by the software developers in Naur’s study, and if the same developers had access to an explanatory model like DMI, their difficulties could have been both anticipated and prevented.

The principal result of this article is the presentation of a model that explains how social accomplishments take place; a model that is repeatable and true for different situations. The model provides software engineering with a shared conceptual apparatus explaining how interpretations take place and which dimensions of information are necessary in the process of interpreting. By relating the explanatory model to the three classic papers and to own study, a better understanding of interaction and reality construction in the context of software engineering is furthermore achieved. Altogether, this provides software engineering research, teaching, methods and process development, and professional practice with a new frame of reference. In the following section DMI is presented and exemplified. Thereafter the three classic papers are presented in Section 3. In the following section DMI is discussed in relation to these papers. Here Brooks’ four product oriented categories, found in Section 3.1, are converted to three new socially oriented categories in order to fulfill demands from the interpretative paradigm found in Section 3.2, that Floyd requests. Through this act, we move a step away from the product oriented and explanatory perspective, towards a social and action oriented perspective. The contents of the suggested categories are in no way final; more work is needed here. Finally, in the conclusions, improvements to software engineering are suggested, based on the knowledge that DMI provides.

2. The documentary method of interpretation

The documentary method of interpretation was borrowed from Mannheim, who originally reserved it for scientific
knowledge [16, p. 78]. Harold Garfinkel demonstrated that the very same method is at work in the processes through which people understand each other and investigate their daily world. It is a method that makes possible the recognition of what a person is talking about despite the fact that he/she does not say exactly what he/she means. The method consists of treating an actual appearance as “the document of,” as “pointing to,” as “standing on behalf of” a presupposed underlying pattern. Not only is the underlying pattern derived from its individual documentary evidences, but the individual documentary evidences, in their turn, are interpreted on the basis of “what is known” about the underlying pattern. Each is used to elaborate the other [16, p. 78]. Hence, the intelligibility of what is seen rests upon the reader’s ability to make out what is meant from what is seen in that specific situation. The tenor of this text will be elaborated in detail in this section with help of one early study by Garfinkel, and in the next sub-section with the help of the present author’s own study.

To clarify the function of the DMI, Garfinkel performed an experiment with 10 undergraduates. In the experiment, the students were supposed to obtain advice about personal problems from a counsellor. The students first presented their problems and related contexts. The students then had to ask at least 10 questions to which the counsellor could only answer either ‘yes’ or ‘no’. The counsellor and the consulting student were located in different rooms connected by means of a communication system. After each of the 10 questions the student unplugged the communication system so that the counsellor could not hear his/her obligatory comments on the answer; the comments were recorded. After the student had asked all the questions s/he also gave a summary of his/her impressions of the entire counselling experience. What the students did not know was that the counsellor’s ‘yes’ and ‘no’ answers were determined on a random basis. Below, a chosen set of Garfinkel’s findings are presented [16, pp. 79–94]:

- None of the students found it difficult to ask all their questions.
- The responses were perceived as answers to the questions.
- Students often immediately constructed a sense of what was meant even if this was not directly expressed by the counsellor.
- The answers were seen as advice leading to a solution to their problem.
- All expressed some form of appreciation and criticism in response to the advice received.
- Attempts were made to find hidden meanings.
- Each new question emerged after reflection upon the course of conversation previously provided by the student in relation to the given answer to the question.
- Present answers provided answers to further questions that were never asked.
- Incongruous answers were resolved by imputing personal knowledge and guessing the counsellor’s intent.

Explained in Garfinkel’s terminology: the students treated the counsellor’s ‘yes’ or ‘no’ answers as “the document of,” as “pointing to,” as “standing on behalf of” a presupposed underlying pattern. Not only is the underlying pattern derived from its individual documentary evidences (i.e., ‘yes’ or ‘no’ in the physical situation and context), the individual documentary evidences in their turn are interpreted on the basis of “what is known” (i.e., personal experiences, expectations, normative values and knowledge of the subject) about the underlying pattern. Hence the students understood each answer from the counsellor in the context of their own comments upon it, including non-spoken common sense knowledge and normative expectations in the subject. One famous aphorism that captures the challenge that the DMI explores is Wittgenstein’s aphorism: if a lion could talk, we could not understand him [30, p. 223].

Despite the presence of the term ‘method’ in the name, DMI is not a method in the usual software engineering sense of that word, i.e., prescription, guideline, set of techniques and tools. DMI is instead an explanatory framework of how humans interpret information. As such a framework it can be used as a reference model during discussions of problems in past and ongoing software development projects as well as when planning complex future projects.

In the next sub-section, the scope and effect of this model in the context of software engineering project practice will be explained and elaborated in detail. The effect of not including the lessons learned from this model in the official planning of a project will be demonstrated through an industrial case. In this case, a software development project member struggles hard with creating the foundation for interpreting his requirements. One important finding of this article is that if humans are to interpret behaviour, they cannot ‘just look at’ the phenomenon, since they have to impute motive and rationale to actions in order to explain them; in these situations the documentary method of interpretation is at work without the possibility to take time out.

2.1. The documentary method of interpretation applied at software development

The present author’s way of demonstrating the DMI in this section is to split it into the following two ethnomet hodological concepts: member and indexicality. Ethnomethodology has developed its own understanding of the notion of member, and borrowed the concept indexicality from linguistics (see Coulon for history and relations to other sciences [8]). Figs. 1–4 will be elaborated upon in order to explain the terminology and how the DMI helps us to explore and understand an unclear requirements situation in a software development project. The example is taken from an ethnographic study carried out by the author during a five month study of an internationally distributed software development project. The project was performed in and by a Swedish part of the organization, with the main
customers placed in Germany and with the end users in USA, Spain, Sweden and Germany. In Sweden, several departments of the company were situated in different locations and a sub-contractor was also involved. The aim of the project was to develop a graphical programming environment, a set of tools and methods (including documentation templates for different tasks, guidelines on how to apply these, etc.) for its application and training in its use. (For further details see [27].)

In this study we asked one software engineer, Arne, working on the code compiler in one sub-project (also the compiler expert in this sub-project), how he went about interpreting the requirements described in the specification. The very same requirements that he was supposed to implement. He answered:

I don’t … I mean, I can’t understand them, not from the Requirements Specification document on its own.

Arne

Fig. 1. Requirements interpretation problem.

In the ethnomethodological terminology, the compiler expert Arne’s comment about requirements revealed that even though he was a highly competent member of the project, he was not a competent member in the ordinary language sense. This means that he lacked the adequate *indexicality* for the purpose of understanding ‘his’ requirements. The ethnomethodological notion of *member* refers to the competence and mastery of natural language and not membership of a social group [8, pp. 26–27]. By becoming a ‘competent’ member in this view one is adapted to an institutional language. In the worldwide distributed software development project that was studied, many such institutional languages existed. In those institutions, the understandings depended on culture, organization, contexts, profession, and local politics. Arne confronted a situation in which he had to impute motive and rationale to his given requirements in order to understand them. As already mentioned, he was confronted with the problem of lacking an adequate indexicality. In fact, the inability to identify a need for adequate indexicality caused a lot of troubles for the software engineers in the studied project (presented last in this section).

Indexicality points to the incompleteness of words. Terms take on full meaning in the context of their production. The central problem with indexical terms is that the referents of the terms vary with the circumstances in which they are uttered, i.e., the true values of statements are to be found in the situation and context in which they were first spoken. In conversations the significance of the speaker’s intention comes from:

- Context – the situated circumstances, e.g., observable body language and behavior, what is said and what is left unspoken, and the physical environment.
- What is known – past conversations, the listener’s knowledge about the speaker’s normal behavior, mutually understood and agreed rules of conduct, and what is believed to be the shared domain of the subject discussed.

As is also demonstrated by Garfinkel’s study, the indexicality of terms can be found in these dimensions.

A text in a requirements specification is a set of terms chosen for specific reasons. It is also a formalization that includes expressions that in their turn are extensions of the human mind that once wrote them (also identified by Naur [22] and cited by Mathiassen [20, p. 142]). Hence, precisely because of the fact that expressions are extensions of the human mind, the problem of how to achieve shared understanding is a key issue in every software development project. How can we ensure that we understand each other well enough? Precisely because of the fact that terms are indexical in local situations, this is a problem that grows with the number of people, roles, and organizational cultures involved. Often, to be sure of what is meant by a formalized expression (e.g., a requirement specification) it must show up later on in concretizations produced in the software development process. In Fig. 2, Garfinkel’s two dimensions of indexicality, context and what is known, are now included to demonstrate why the compiler expert could not understand his requirements.

Arne’s context and what is known differs from the context and what is known possessed by the requirements originators who originally documented the requirements. In this situation, of course, Arne possessed relevant knowledge about compilers, programming and project execution; these skills are not questioned. What Arne meant by his answer was that he actually lacked knowledge about the interplay between the local situations and contexts, including the politics and reasons for initiating the requirements he was supposed to implement. Arne continued:
What I do is, I contact the technical person that is responsible for that particular requirement. Of course, the problem with this is that the person's name is never mentioned in the Requirements Specification. I mean the person who originally figured it out. I have to do my own research. I start by contacting people... asking around to establish where a given requirement might have its origin. That means finding the technical person responsible for the requirement, finding the person who originally wrote it down, and also finding other people who might have an interest in these specific requirements. I start by asking around with the help of e-mail and telephone... questions like ‘what does X mean, and why?’ Then negotiations start about the requirements in question. I almost always use my contacts, and we might have, say, six people involved in a discussion about the relevant requirement, people from all over the world. Of course, this gives rise to other problems because of the time difference. It can mean the process is slow... it often takes days to get an answer. It shouldn’t be this way... but it is...

One point about this network which is well worth making is that it existed entirely independently of project management. This fact can be understood as one effect of an insufficient understanding of the social elements of interpretation, interaction and reality construction amongst software engineers. The network was also extensive and ongoing. It was common for Arne to receive communications from people asking him what solutions he had finally adopted, following on from discussions taken at an earlier stage. Fig. 3 below demonstrates how Arne, on his own initiative, established an informal network consisting of individuals located around the world; a network entirely independent of the formal project network.

Illustrated above are the two types of networks in the project and the different types of competent members. The members at the top of Fig. 3 are competent project members from a formal projects qualification point of view. They have knowledge and experience of application related subjects, and process related subjects. The members at the bottom are competent members from a natural language point of view. They know the history and local politics that were played out when specifying the requirements Arne must deal with. They are able to identify who might be interested in what, why, and when, if existing requirements have to be changed. Hence, these members had the local knowledge needed for explaining what the requirements meant, who initiated them, and which changes could be made without violating the original aims.

In this new situation adequate indexicality was established through interactive efforts with members of the informal network. These efforts secured an adequate understanding, whereby design and re-design could be made without violating the original aims. Hence, Arne solved his unclear requirements situation through the establishment of a situation where adequate indexicality could be established, represented in Fig. 4 below.

Requirements stand as a proxy for the purposes of those formulating them. Whereby handling the requirements specification is partly a matter of recognizing who is in a position to require an account, and arrange interaction for allocating and accepting responsibility for carrying out the work. Thus Arne’s efforts consisted of identifying what the requirement initiators (the competent members from a natural language point of view) really meant, and also identifying who might have a different version. This meant locating geographically distributed project members who might have initiated a given requirement, placed in different parts of the organization, and sometimes tracing their history within the project.

In the same study from which the above example was taken, three sub-project leaders reported a diverse set of innovative tricks that they used to concretize their requirements specifications during the project. They all
had in common that the requirements could not be adequately handled without extensive interactive efforts, and a lack of formal support for achieving this interaction. In an effort to solve misunderstandings in a problematical requirement situation, one sub-project leader presented a proactive strategy with the aim of prompting a reaction from the customer. When they did not manage to clarify a requirement in this sub-project they simply decided to produce a solution anyway. In this way they turned the interpretation problem around, making it a problem for the requirement originator’s organization, who then had to interpret if it was the correct solution to the requirements. Another sub-project leader sought a consensus through iterative cycles of interacting with product management and other acknowledged expert roles and groups in the project. At one point of time he traveled to his appointed requirement interface person and refused to leave before they tracked down and talked to all requirements initiators. A third project leader contacted the people behind the previous project in order to form a basic understanding which could subsequently be presented to the customer. He then iterated and negotiated new understandings of the requirements based on these two stakeholders’ iteratively provided feedback. As is visible through these project members’ acts, software development is truly a human process of interpretation, interaction and reality construction. In Rönkkö et al. [27] the interaction presented is discussed at length with help of Gerson and Star’s concept *due process* [17].

The above explanation was designed to explain what is required of us as social and human beings to understand one another, and to identify the interactivity needed to achieve this understanding. It is demonstrated that Garfinkel’s two dimensions of reference are needed when constructing the world (over and over again, as there actually is no ‘time out’ from the DMI). The dimensions of reference needed are context and what is known. An alternative explanation of DMI including more ethnomethodological concepts can be found in [26, Chapter 6]. Taken together, the examples from the distributed software development project also provide a recent software engineering case that demonstrates how the lack of subject knowledge leads to poor formal support from the software development organization.

3. The social element in software engineering discourses

In this section we examine three now historic papers that represent requests and reasoning for taking the social element into serious consideration in order to improve software development. In the first paper, Frederick P. Brooks [4] adopts a product perspective where he provides powerful categories capturing the nature and difficulties inherent in the building material which practitioners must manage in software development projects, i.e., the software. The product view was for a long time a dominating view in the software engineering community. In relation to the model demonstrated in this paper, these categories help to explain why indexicality of words and texts are so hard to achieve and communicate in software development projects. In the second paper Christiane Floyd [12] revealed ongoing controversy between rivaling ideas and attitudes underlying scientific and technical work in software engineering. She argued for a paradigm shift in order to handle aspects of the living human world in a more systematic manner. She demonstrated that software engineering lacks adequate approaches to meet this challenge. In her argumentation, software practitioners do not analyze methods, apply requirements, or refer to implementations; instead they unavoidably interpret and construct their own understanding of these. DMI explains how her claims are played out in practice by the practitioners. In the third paper Peter Naur [23] identifies contingencies caused by the fact that handling methods is in practice a social achievement. Surprisingly, despite extensive and exemplary support, it transpired that in several major cases developers failed to make use of the facilities inherent in an old program. Naur thereby concluded that programming is primarily about building up certain kinds of knowledge that are possessed by the programmers themselves, and that documentation can only have an auxiliary function. In relation to the present article Naur’s study demonstrates what can happen when developers lack the kind of knowledge revealed in this article, which explains and guides what is required to effectively take advantage of the results of others’ work.

In retrospect these three papers provide important challenges for the software engineering community to handle. To these insights a model is presented, providing a useful explanatory frame of reference that can help software engineering research, teaching, and methods development. Each of these perspectives will be returned to and summarized in Section 4. Discussion in relation to the model presented in this paper.

3.1. No silver bullet – essence and accident of software engineering

Brooks provided the software engineering community with a much discussed and quoted explanation as to why software development is so difficult [4, pp. 177–203]. Brooks’ product-oriented explanation is one possible interpretation and categorization of the challenges making up the environment in which practitioners make software methods work. Brooks divided the software development context into *essence* and *accident* [4, p. 182]. He advocated that software engineers must address the issue of essence if substantial success is to be achieved. *Essence* entails the mental crafting of the conceptual construct. *Accident* refers to the implementation process. Essence concerns the nature of software entities, i.e., is a construct of interlocking concepts such as data sets, relationships among data items, algorithms, and invocations of functions. Accident refers to those difficulties that attend its production today but are not inherent in its nature. (From a social point of view
DMI actually provides essence to Brooks’ accident part, i.e., through the concern of the nature of social accomplishment. Accident relates to words such as incidental or opportunity, and not to interpretations such as occurring by accident, or misfortune. The inherent properties of the essence of software systems are placed in four categories: complexity, conformity, changeability and invisibility. Short explanations of these categories follow.

The first property of essence: software entities are complex because no two parts are alike, i.e., repeated elements with which to build do not abound in software construction, as may be the case in other engineering disciplines. Digital constructs also often have a very large number of possible states. If a system is scaled up, elements often interact in non-linear fashion. As a result, complexity is much larger when compared to other engineering disciplines that often allow building using repeated elements. Challenges arise: how can one enumerate and understand all possible states of a large program? How much can one remove through abstraction? Bearing in mind that complexity is an essential property of the program, if one removes too much complexity by abstraction, the essence is also removed. How should one communicate the construct? From a management point of view such complexity makes an overview difficult. How does one find and control all loose ends?

The second property of essence: conformity refers to the fact that software does not obey the laws of nature; software is a human construct and thus includes considerable arbitrary complexity. The software construct is formed without the stable laws of nature that predict rhythm and reasons. Instead, it is the human institutions and systems with which the construct must interact that create predictability in the software, i.e., the stability must be as a social construct.

The third property of essence: a software entity is constantly subject to pressures for change. Society and culture change, as do people’s goals and means. People often try out the software at the edge of, or beyond, its original aim. As a result, the life of the machine vehicle for which it is constructed has developed and continues to develop very rapidly, i.e., new hardware introduces and requires that new demands be made on the software.

The fourth property of essence: invisibility refers to the fact that software has no ready geometric representation. An architect might investigate the results of his/her work with the aid of geometric representations in a way that a software developer cannot. Software is not embedded in space. As soon as we represent the software structure we need a number of general directed graphs, superimposed one upon the other: data flow diagrams, state diagrams, flow of control, patterns of dependency, class diagrams, object diagrams, architectural overviews, conceptual overviews, use cases, use scenarios, requirement specifications, and prototypes, etc.

In a follow-up in 1995 [4, pp. 207–226] Brooks concluded that although the trouble caused by the four essentials can be ameliorated, his arguments still hold. There are no silver bullets, and never can be.

In relation to the present article, Brooks’ paper demonstrates the nature of and challenges that follow with the building material inherent in software development. The building material is part of the explanation of why indexicality is so hard to establish in software development practice. In this paper he also represents the product orientated view.

3.2. Outline of a paradigm shift in software engineering

Floyd has looked at the human side of software development, but still the main uptake on Floyd is on use oriented design of software and not software engineering. Her work is also much more taken up in the participatory design context than in the software engineering context. As early as 1987 [12] Christiane Floyd argued that there was an ongoing controversy between rivaling ideas and attitudes underlying scientific and technical work in software engineering. She suggested a paradigm shift in order to handle social aspects in a more systematic manner. She also demonstrated that software engineering lacked adequate approaches with which to meet this challenge. In this paper she contrasted the predominant product-oriented view within software engineering with a process-oriented view.

According to the former, the contexts of future applications to be developed are considered to be fixed and clearly understood, thereby allowing software requirements to be determined in advance. According to the latter view, software is seen in relation to human learning, work and communication. It is an evolving world in which needs change constantly. In this view the product emerges from the totality of interleaved processes of analysis, design, implementation, evaluation and feedback. Different groups of people and roles, including users, carry out the development activities.

Floyd required a richer process perspective in research, teaching and professional practice. The process perspective as the primary framework needed to be developed, in order to incorporate the product view. With such a starting point it is possible to rethink our basic assumptions and concerns and include human needs. Floyd points to symptoms that need to be addressed: we need open ‘information’ with respect to human processing of meaning, development systems that are dynamic and evolving in time, developers must be included as part of our reference system when considering developments, program results must be considered as versions, interaction between software and its environment and users is necessary. The different paradigms provide various frames of reference, suggesting different questions to be asked, quality considerations to be aimed for, and guidelines for interpreting research and technical results.

Floyd demonstrated that the product-oriented view has been the ruling view since the first software engineering conference 1968. The problem with such a view is that
3.3. Programming as theory building

That Naur’s early work is still considered is visible in the book Social Thinking – Software Practice by [2,10], and also demonstrated by authors like Mathiassen and Cockburn. Mathiassen included ideas from Naur when defining what a method is, in his Reflective Systems Development [20]. In Cockburn’s book Agile Software Development [7], a reprint of Programming as Theory Building can be found. In this paper Naur discusses programming from the perspective of human activity.

Naur observed teams dealing with unexpected and erroneous program executions as well as with modification of programs. The programming activity as outlined in his paper includes the complete activity of design and its implementation. Two cases are presented: the first concerns the development of a compiler, and the second, installation and fault diagnosis in a real-time system. In the former, a second team was given the task of writing an extension to the first version of the compiler. The technological task was considered to be a modest language extension for another type of computer. A contract was agreed upon which provided the new team with full documentation support including annotated program text, documented design discussions and personal advice from the first team. The new team developed the compiler and sent it to the old team for review. It transpired that in several major cases the new team had failed to make use of the facilities inherent in the original program. This failure arose despite the fact that the facilities were inherent in the structure of the old program and that the implemented structure was discussed at length in the documentation. The power and simplicity that were inherent in the old program were destroyed in the new version of the compiler. The members of the old team were able to spot these cases instantly and were also able to suggest simple and effective solutions based on the original program structure. Over the years, new programmers in the organization took over the compiler. A follow-up of the compiler was carried out 10 years later, together with a programmer from the original team. The result was disappointing; the original power structure was still visible but made entirely ineffective through additions of different kinds.

In the second case, one group of programmers had been occupied full-time and for several years with new installations and fault finding of a real time system for monitoring industrial production activities. When faults occurred the programmers relied almost exclusively on their knowledge of the system and on the annotated program text. These programmers were unable to identify any additional documentation that would be useful to them. Other programming groups, who had access to full guidance on the program’s use as well as program documentation, regularly encountered difficulties. These were difficulties that upon consultation with the producer and fault-finding programmer could be traced back to inadequate understanding of the documentation. These faults were also easily cleared up by the installation and fault-finding programmers.
Based on those cases Naur concluded that programming is primarily about building up certain kinds of knowledge that are possessed by the programmers themselves, and that documentation can only have an auxiliary function. These observations led to the conclusion that the production view is misleading and that a theory building view could be a useful alternative. Programs are not only designed and produced; they are also modified at a later stage. Using the perspective of change, Naur concludes that the proper, primary aim of programming is, not to produce programs, but to have the programmers build theories of the manner in which the problems at hand are solved by program execution [23]. In his view programming should be viewed as an activity for gaining insights and generating theories. In relation to the present article Naur’s industrial case also demonstrates how exemplary documentation and support might not be enough to successfully build new software based on existing software. It is interesting to note that Naur’s theory building view is in agreement with Floyd’s reality construction view. Both views suggest that human processing of meaning is an act of interpretation and construction.

4. Discussion

Naur’s paper provides an example of a study, extended over a longer time period, of what happens when we apply standard (at that point of time) software development procedures to approach the challenge of ‘taking over’ existing software. The failure appeared despite the fact that the facilities were inherent in the structure of the old program and that the implemented structure was discussed at length in the documentation. The members of an old team were able to spot failures instantly and also able to suggest simple and effective solutions based on the original program structure. Thus, as they were the only ones who had built up the adequate theories when developing the software, they were also the ones who possessed the adequate indexicality to re-create those theories. Also, in a 10 year perspective, the results of a follow up study remained disappointing; the original power structure was still visible but made entirely ineffective through additions of different kinds. Obviously such development efforts did not do the job. From the point of view of the present author, Naur identifies software practitioners’ difficulties which can be explained by the model presented in the present article. In this view the latter development team did not realize that they were “merely” competent members from a formalized project point of view, and not from the natural language point of view. They did not realize their difficulties of adequately understanding the extensive documentation; neither did they seriously consider taking advantage of the fact that the old team that possessed the original theories was there to be interacted with.

In Floyd’s interpretative view these programmers’ cooperative difficulties could be expected, and also remedied to a large extent by accepting her underlying assumptions and implementing the iterative and incremental processes belonging to her interpretative paradigm. Floyd identified the predominant product-oriented view within software engineering and contrasted it with a process-oriented view. She argued that there was an ongoing controversy between rivaling ideas and attitudes underlying scientific and technical work in software engineering. To resolve the situation she suggested a paradigm shift. Today, process thinking is widespread and well-established, but it has still not reached the level Floyd advocated. The social element is still not adequately understood. In her process oriented paradigm, software developers have to interpret and impute motive in order to understand their requirements, their methods, and how to implement their decisions in existing development environments. What is still lacking despite her valuable insights and concrete advice is explanatory models explaining how these assumptions are played out in practice. In fact, both Naur and Floyd suggest that human processing of meaning is an act of interpretation and reality construction, but neither of them provide a model of how this is done. Starting in the concept of interpretation, DMI is a model that can explain how both Naur’s theory building and Floyd’s reality construction assumptions are played out in practice, with reference to the identified dimensions of indexicality.

In relation to this, Brooks’ work helps us to understand the extent of the challenge that Naur and Floyd’s work identified. In Brooks’ product oriented perspective, four essential properties have to be understood and handled in order to reach sustainable progress. It is interesting to note that the properties identified by Brooks are still very much present and even more challenging today as software has broadened in many ways and become even more complicated. In relation to this article, Brooks’ work provides valuable categories that help us identify different sorts of challenges that complicate and sometimes hinder us from having clear and unambiguous indexicality in software development projects. Brooks’ choice of categories is focused on pinpointing the nature of the problems following from ‘software’; they answer the question of ‘why’ software is such difficult material to handle. In relation to this article I have chosen a different categorization to be able to emphasize ‘what’ needs to be approached. The ‘what’ is carried out by people and is hence part of the missing knowledge and social agenda within software engineering. Hence Brooks’ four categories, of complexity, conformity, changeability and invisibility, are here converted from the product perspective to the interpretative and constructive paradigm that Floyd made requests for. In a sense the new categorization is a second step, a stage away from the product and explanatory perspective towards a social and action oriented perspective: here, it is not complexity and invisibility that is the main interest – it is instead accepting the manifold views needed to ‘grow’ software constructs. It is not the lack of conformity that needs to be approached – it is individuals’ free choice. Finally, the changeability we need to keep track of and predict is the evolution of society itself. Besides pointing to what needs to be approached, this new
view also opens up for extending the contents of the categories with social challenges, e.g., intuition, change of mind, memory, politics, attitude, trends, etc.

Brooks’ first and fourth entities are here merged into one first category (of three) to identify what is the main challenge in relation to the nature of software, i.e., accepting the manifold views needed to ‘grow’ software constructs. Digital constructs often have a large number of possible states that interact in a non-linear fashion. Hence, when scaled up the complexity is greatly increased in comparison with other engineering disciplines. Identified challenges were: how can one enumerate and understand all possible states of a large program? How much can be removed through abstraction? Bearing in mind that complexity is an essential property of the program, if one removes too much of the complexity, the essence is also removed. How should one communicate the construct? From a management point of view the complexity makes an overview difficult. How does one find and control all loose ends? It seems that this kind of material does not lend itself to be represented in its entirety; instead we can only capture relevant aspects of it. This means that software has no ready geometric representation. In order to represent the software structure we need a number of general directed graphs superimposed one upon the other. If different aspects of the same software are captured in different sorts of diagrams, what does that say about the understanding of the software in its entirety? Can we even claim that such a thing as a coherent view of it exists, or is all that can exist different possible views that are presentable in a number of possible combinations? These social insights suggest that massive interaction has a crucial role to play.

The second category that complicates clear and unambiguous indexicality is individuals’ free choice. Software does not obey the laws of nature. It is a human construct and thus includes considerable arbitrary complexity. Obviously this second category enlarges the complexity of the first category when the two are combined. Standardization, larger building blocks of code, and introducing shared concepts such as design patterns can bring some order to this arbitrariness. This second category might also be the most difficult one to deal with as it is directly related to individuals’ personal tastes and opinions. It is well known that, despite similar education and work experience, individuals may still prefer totally different solutions. In this article we only see the tip of the iceberg, and to this category in particular, many other challenges can be added: the fact that humans may lack clear ideas or ways to express what they mean; intuitive feelings and tacit knowledge are sometimes included that cannot fully be spelled out in words; sometimes rough experience-based ideas of direction are included in formalizations. As if this were not enough, people also change their minds, or do not remember what they actually meant at a specific point in time. Ongoing politics on different levels, inter-subjectively favoured ways and desires are aimed at and played out in organizations. This latter factor of politics may contradict the most clearly thought-through formalization. People may deliberately say that they are doing one thing while they are in fact consciously doing it another way. Again social insights points to the need for massive interaction. This time we need to introduce a new layer of reflection including other social elements on top of interaction, e.g., humility, sensitivity and ethics, to mention but a few (see for example Humphrey’s The commitment ethic [18], and DeMarco & Lister’s The making of community [9]). These are two only examples of social elements found within the community, which need to be discussed and studied to a larger extent in software engineering.

The third category to be handled is the evolution of society. Software entities are constantly subject to pressures for change. Society and culture change, as do people’s goals and means. Hence, new hardware introduces and requires that new demands be made on ‘old’ software. Internet access together with software development communities such as shareware and open source increase the development speed of software applications. One major problem here is that no individual can keep up with the thousands of improvements and changes that are continuously produced by others. Instead, when it comes to this dimension of indexicality, we are often captured in outdated frames of reference.

All of these categories are related to the explanatory model DMI through the concept of indexicality. And DMI provides a new frame of reference that from a social point of view helps us to understand the nature of interpretation, and thereby also how we can better approach the design of methods, processes and software process improvement, as well as interpretation problems related to industrial software development. One recent software engineering example of the latter is the author’s own study used in this article to elaborate DMI. In relation to this study, the model explains and motivates why and what sort of support is needed for tracing requirements initiators in distributed software development projects (see [27]).

5. Conclusion

It is identified that the social element has long been a neglected area in software engineering. The social element refers here to software practitioners and their interpretative achievements to make software development work in practice. Today it seems to be the agile methodologies that have taken on this challenge [7, p. 1]. Still, despite these new actors’ seemingly successful implementation of software development methodologies, in practice they share the same situation as the rest of us, i.e., a lack of explanatory models of the social elements. One indication of this is Highsmith’s expression mushy stuff. According to Miller and Larson [21] Highsmith has identified values and culture as central when evaluating software methodologies. In this context mushy stuff illustrates the difficulty we all share today; expressing ideas about human values with language that is not as precise or articulate as the language engineers routinely use to express technical ideas [21]. Hence, the agile
methodologies continue to be based mainly on knowledge-
able practitioners’ personal experiences of developing soft-
ware. In this article one understanding of above shared diffi-
culty is presented with help of a model borrowed from soci-
ology, the documentary method of interpretation. The model helps us to explore how software practitioners’ inter-
pretative achievements are played out in practice from a
natural language point of view. To help develop an under-
standing of the model, it was applied to a software develop-
ment project member’s requirements situation. The results
from the present author’s study of the software develop-
ment project were also presented as a recent example of the
effect of not including the lessons learned from DMI in the
official planning of the project. Thereafter three classic
papers were presented. All of them provided valuable his-
torical contributions: Brooks by exploring the difficulties in
the building material from a product perspective, Floyd by
exploring the weak relation to the social element from a
process perspective, Naur by exploring the nature of and
difficulties in understanding software produced by others.
Both Naur and Floyd suggest that human processing of
meaning is an act of interpretation and reality construction,
but neither of them provide a model of how this is done.
With help of the concept indexicality, Brooks’ work helps
us to understand the extent of the challenge that Naur’s
and Floyd’s work identified. The DMI is identified as a use-
ful complement to these papers.

The model demonstrates how perfect communication is
an impossible mission. Hence, what software practitioners
unavoidably have to learn is how to manage the incom-
pleteness of communication, and as identified in this article,
handling incomplete communication can be seen in the
light of indexicality. In fact adequate indexicality of terms
is here suggested as ‘the’ crucial factor for practitioners in
order to reach success in gaining an understanding of and
cooperation around continuously growing software applica-
tions in development projects.

In the context of software engineering DMI provides an
explanation of how project members understand each other
and how they are also able to investigate their daily world.
Meaning is constituted in the interplay of project members
and their objects of interest under the concrete conditions
of a particular setting. The understanding depends on the
situated context and what is known. That these two dimen-
sions of indexicality are especially difficult in our discipline
was demonstrated in the discussion section by the three cat-
egories: 1 accepting the manifold views needed to ‘grow’ soft-
ware constructs, 2 individuals free choice, and 3 the evolution
of society. Notice that there is room for improvements in
the three suggested categories where indexicality can be
found. In relation to the first category it was questioned if
such a thing as a coherent view of software can exist. It
seems that all we can claim is the existence of different views
that are presentable in a number of possible combinations.
Hence, massive interaction has a crucial role to play in soft-
ware engineering. Obviously, it has not yet received the
attention it rightly deserves. In relation to the second cate-
gory the need for massive interaction was once again
pointed out. This time it was argued that we need a new
layer of reflection, including social elements such as humili-
ity, sensitivity and ethics, added on the top of interaction.
Here, it is also possible to introduce more ethnomethod-
ological concepts to deepen our understanding of interac-
tion, i.e., accountability and reflexivity (see [8,26 Chapter
6,27,5] to learn more about interaction and those concepts
in software engineering practice). In relation to the third
category we have to search for research results performed
in and by other communities. Future work would be to
continue to develop our understanding of interaction. One
way to address this challenge is to follow the suggestions
provided by this paper’s social and action oriented perspec-
tive, i.e., to continue the work with a starting point in the
categories of indexicality.

Acknowledgements

This work was partly funded by The Knowledge Foun-
dation in Sweden under a research grant for the software
development project “Blekinge – Engineering Software
Qualities” http://www.idp.bth.se/besq. Thanks also to
the members of the U-ODD research group, Jeff Winter, Olle
Lindeberg, Jeanette Ericsson and Christina Hansson for
their valuable comments, which helped me to improve the
article. Thanks to the reviewers who helped to improve the
article.

References

[1] I. Aaen, Software Process Improvement: Blueprints versus Recipes,
[2] U. Andelfinger, On the intertwining of social and technical factors in
software development projects, in: Y. Dittrich, C. Floyd, R. Klisches-
ski (Eds.), Social Thinking – Software Practice, MIT Press, Cam-
bridge, Massachusetts, 2002, pp. 185–204.
[3] B. Boehm, A Spiral Model of Software Development and Enhance-
ing (Anniversary Edition with four new chapters), Addison Wesley,
1995.
engineers, in: M. Jirokta, J. Goguen (Eds.), Requirements Engineer-
pp. 217–240.
ратive Design and Development in Software Engineering, In the Jour-
[8] A. Coulon, Ethnomethodology, Sage series Qualitative Research
[9] T. DeMarco, T. Lister, Peopleware: Productive Projects and Teams,
Loci” in a global context, in: Y. Dittrich, C. Floyd, R. Klischewski
(Eds.), Social Thinking – Software Practice, MIT Press, Cambridge,


[23] P. Naur, Programming as Theory Building, Microprocessing and Microprogramming. 15, can also be found in: Computing: A Human Activity – Selected Writings From 1951 To 1990, Chapter 1.4, 1992.


