Situated Software Design

Hubert F. Hofmann, Rolf Pfeifer, Erik Vinkhuyzen
AI Lab, University of Zurich
Winterthurerstr. 190, CH-8057 Zurich, Switzerland
Email: {hofmann, pfeifer, vinkhuyz}@ifi.unizh.ch

PLEASE NOTE THAT THIS ARTICLE HAS BEEN SUBMITTED FOR PUBLICATION TO THE FIFTH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, JUNE 1993, SAN FRANCISCO

IT APPEARS IN THE FORM OF THIS TECHNICAL REPORT ONLY FOR PRELIMINARY AND RESTRICTED DISTRIBUTION

Abstract

In computer science we are used to characterize systems in terms of levels of virtual machines. In other words, we are dealing with virtual worlds. For virtual worlds the rationalistic perspective is well suited, i.e. general rules can be logically applied to situations characterized by objects with well-defined properties.

However, software design is concerned with developing systems for the real world. The real world differs in several crucial points from virtual ones: it changes continuously, it is largely unpredictable, and it is only partially knowable. Many design approaches have not sufficiently taken these points into account. A number of alternative approaches have been proposed, but they have been largely ad-hoc and lack theoretical rigor.

We show the need for a more suitable approach to software design. We believe that the idea of situatedness can provide the basis for such an approach. Situatedness is based on the fact that people are involved in a continuous interaction with a changing environment. People, e.g. software engineers and users of software systems are viewed as situated agents that bring to bear their prior experience on the particular situation. We discuss the importance of this view for software design methodology.

Table of Contents

1 Introduction

Software design is a process of developing a software system. It is carried out within an organizational context and involves software designers and clients, i.e. customers and users. A customer is an individual or an organization who wants a software system to be built and is willing to bear its development costs. The term user denotes an individual or an organization working with the software system.

Existing approaches to software design are largely based on a rationalistic perspective. Winograd and Flores in their seminal book “Understanding computers and cognition” have characterized rationalism by three steps [29, p. 15]: (1) Characterize the situation in terms of identifiable objects with well-defined properties. (2) Find general rules that apply to situations in terms of those objects and properties. (3) Apply the rules logically to the situation of concern, drawing conclusions about what should be done.

While software design that relies exclusively on this perspective may be suited for virtual worlds, it has proved to be inappropriate in real worlds. Virtual worlds are formal, like logic, mathematics, formal games, etc. The real world differs from virtual ones in a number of important respects: it changes continuously, it is largely unpredictable, and it is only partially knowable. As obvious as the distinction between virtual and real world may seem to be, we demonstrate that software design approaches neglect this distinction and treat the real world as a virtual one.

To overcome the problems inherent in current design approaches, we have to take the dynamics of the system-environment interaction into account. A basic assumption is that the users of software systems are viewed as situated agents, rather than as actors following preconceived plans [27]. As situated agents, users interact with their environment and bring to bear their prior experience on the particular situation. If design is to be based on this perspective then it must be viewed as situated. We discuss the implications of this view and conclude that software design approaches based on a rationalistic perspective are intrinsically limited and that an alternative view is needed.

2 The Rationalistic Perspective

The traditional approach to software design promotes a rationalistic perspective. Following this perspective, the application domain is part of the objective reality. When investigating the application domain, a designer seeks to capture regular patterns of organizational processes by identifying key organizational objects and relationships. These objects and relationships are used to organize (human) work into formal tasks.

The rational analysis of tasks in turn supports the construction of evaluation criteria, which are used as an “objective” means to evaluate task performance. So described, a worker's job can be analyzed with respect to the optimal efficiency and effectiveness of organizational processes using established evaluation criteria. It is important to note that evaluation criteria are derived from abstract descriptions of the task rather than from the reality of the work situation.

Whether tasks are performed by people or by software systems their relationships are established by information exchange. For example, a software system performs financial and inventory functions. People have access to this system through terminals. Links between the software system and a central database allow terminals connected to the system to access data in the database and to generate pertinent reports using that data. Therefore, the traditional design approach favors techniques such as data flow diagrams and decision trees which aid in formulating information streams in organizations.

Software design is seen as an objective and rational process with a “neutral” software designer operating in a cooperative environment. A software designer must be an expert in technology, tools and methods of software design. The technical knowledge about hardware and software is at the center of interest. In other words, software design becomes primarily a technical process cf. [21]. The focus on technical knowledge implies that software design is viewed primarily in formal terms, placing less emphasis on human activities.

However, practical experience with this perspective also reveals its deficiencies ([16], [29], [9]). The traditional approach to software design investigates means rather than discussing ends. The ends are often taken as inputs to the process. In other words, the traditional approach implicitly assumes that the ends are agreed or at least clearly defined at the outset of the project. But experience shows that ends are often controversial and subject to considerable disagreement and debate ([17], [18]).

Software systems are based on assumptions about which information is relevant and which characteristics of the environment are essential. Usually these assumptions are implicit, but they may become invalid because the environment changes. Since it takes time to incorporate these changes in the software system, there will be discrepancies between reality and how it is reflected in the software system ([5], [4]). Thus, software systems are often quickly outdated and since their coding does not change they can become conservative forces. These problems may lead to systems that are judged unsatisfactory or unacceptable by the client and have high maintenance costs.

Design primarily focuses on information processing aspects of reality. However, companies function in a way different from the officially advertised one and the work of individuals in a company may differ significantly from its formal description. Individuals are also part of an informal network within the organization and not all of their work contributes directly to the official purpose of the organization. For example, domain specialists use not only their expert knowledge to the benefit of the organization, they also contribute to the organizational culture and play a role in the organizational power structure. Domain specialists are usually responsible for advice even when the advice is given by their assistants.

Formal task analysis and conventional knowledge acquisition do not show aspects of responsibility, significance of informal communication, and do not reflect the individual's social skills, support structures, etc. For instance, formal interviews with domain specialists and managers emphasize domain details or resource issues, but the real nature of the working process remains hidden from the interviewer.

Moreover, the adaptive aspects of expertise cannot be easily captured in a rationalistic way [19]. Human expertise is intrinsically adaptive, it can deal with novel situations, i.e. situations which have not been encountered before. Strictly speaking, in the real world two situations can never be identical. Therefore adaptability is always required when dealing with the real world.

3 Impact of the Rationalistic Perspective

The shortcomings of the rationalistic perspective have lead to alternative design perspectives, which recognize that reality changes continuously, is largely unpredictable, and only partially knowable. These alternative perspectives on software design do not stick to the "three steps" outlined by Winograd and Flores. They do not try to conceal the complexity of the real world by pretending that there is an underlying order that can be captured in rationalistic models as described by the "three steps."

In software design we do not deal with an objective reality, but we have to cope with conflicts of interest, complex social relationships, conventions and cultural norms. In an alternative perspective there is no single reality, there are different perceptions of it ([30], [18]).

Perception is not a passive process of absorbing information. Perceptions are actively constructed depending on the goals and the experience of people, are always changing and never the same for two persons. Thus, software design will normally be looking at the application domain in different ways. For example, one group may think in economic terms (will we gain a competitive edge, will we make money in the future), another in terms of status and responsibilities (will the systems give users or managers more control over the social and physical environment), another in terms of job satisfaction (will we get the opportunity for personal development).

Alternative perspectives not only consider technical aspects, but also take into account social aspects that people must deal with in software design. Thus, alternative perspectives underscore the role of social analysis as a cornerstone for effective and efficient design practices [2]. Recently, the importance of social analysis in software design has been emphasized by several authors, e.g. ([10], [12], [14]).

For most approaches acceptance or consensus is the critical issue. Consensus among affected parties is established through intensive communication. Any software system that finds the approval of the affected parties is viewed as legitimate. If all users identify with a software system, the likelihood of success increases.

The role of the system designer is to interact with the clients (i.e. management and users) and to work out which system fulfills their needs [22]. The needs or the “wishes” of the client for the system to be built are subject to debate. The exploration of the needs involves the users, since they have to work with the software system and experience it as part of their actual working environment, and not as an abstract task description. Thus, the cyclic process in which the software is developed is the most important factor for its success. Software design is viewed as a learning process for those concerned and affected by the proposed software system. For example, Greenbaum and Kyng [13] start in the working environment of users with the envisionment of future work situations allowing users to experience the effect of software systems on their working practice.

Software designers apply alternative perspectives for various reasons. Differences in perceptions of reality exist and often lead to conflict, e.g. about the appropriate action to take in a particular situation. Some approaches such as the “Scandinavian school” ([14], [10]) are based on conflict rather than on compromise between co-operation and conflict. Conflict is viewed as the driving force of organizational change and personal enhancement. The lack of conflict is viewed as undesirable since it can reinforce the status quo.

Alternative perspectives as stated in the literature identify problems with the traditional design perspective and help to overcome some of its deficiencies. The introduction of software systems in companies always entails organizational problems and acceptance is typically a critical issue. The literature suggests that these problems may be overcome by the early participation of users ([11], [14]). Exploratory and evolutionary approaches cf. [6] can aid in shaping the software to the needs of the clients. For example, evolutionary approaches advertise feedback loops to evaluate design artifacts as early as possible by building and testing operational software versions.

4 Towards Situated Software Design

The more recent approaches to software design have addressed some deficiencies of the traditional design approach. Nevertheless these approaches have their problems as well. For example, alternative approaches show a marked tendency to focus on invisible artifacts. Though software systems — the visible artifacts — can be expected to emerge, they do so largely as an indirect result of these approaches. The more direct effect is on the design participants' perceptions, outlooks, and judgments of those participating in the design process. These are the invisible artifacts of alternative approaches.

Social interaction is viewed as crucial for acceptance, but the lack of (managerial) goals makes it difficult to match the consensus achieved in the participatory process against the organizational needs. We think that alternative approaches often ignore the effects of social differentiation. Social differentiation is introduced, for example, by unions, political parties, or other forms of worker organization. Ehn [10, p. 358] gives the following example: "[...] the lack of trade union co-operation, not the technology, not the newspaper owners, suppliers, may ironically become the decisive factor frustrating the dream of UTOPIA[1]. That the UTOPIA team first contacted the graphics worker union made the other unions critical toward UTOPIA, and thus frustrated the dream of a joint design."

Sometimes managers and organizations (like trade unions) refuse to participate fully in the design process. They argue that the software designer is the systems expert and should be designing appropriate systems for them.

There seems little agreement among the advocates of different approaches on the underlying principles behind the choice to divert from the traditional approach to software design. Their motives are usually of a practical nature or stem from the historical social context of the group practicing the design approach. For example, the traditionally strong position of the unions in Scandinavian countries constituted the driving force behind user participation [11].

We think these approaches could benefit from a sound theoretical framework [25]. The idea of situatedness or situated cognition can provide the basis for such a framework.

Situatedness states that people are involved in a continuous interaction with a changing environment, make sense of this environment, and take action in it. Acting is based on previous interactions and on the current situation. A situation is not an

---

[1] In the UTOPIA Project software designers and researchers worked with a group of typographers to help them formulate the ways that computer technology could be used to enhance their skills and better the typographic quality of newspapers.

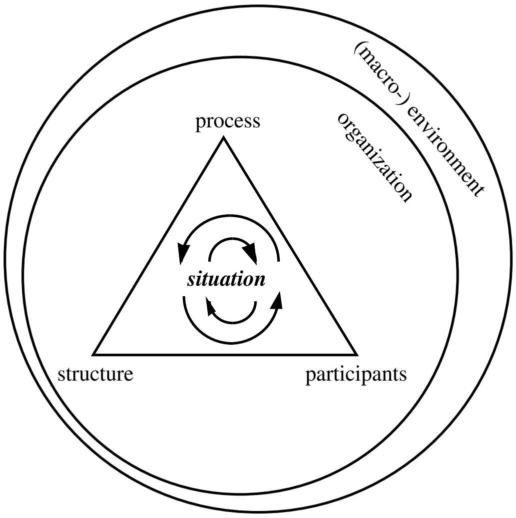

Figure 1. Situation and Context

objective entity but is conceptualized by an agent. This conceptualization comprises the agents, the actions they are currently performing as well as their interactions, and the organizational processes of which the actions are a part (Figure 1). Individual structures, cognitive and motivational ones, and external structures (role expectations, group norms, etc.) are coordinating the behavior of participants and come into being during the interaction. In other words, the behavior of participants and organizations emerge from a dynamic interaction of individual and external structures.

This idea that is the basis for situated cognition ([27], [8]) contrasts with the prevalent view in cognitive psychology, which has had — and still has — great influence on software engineering, especially on the design of expert systems. Many cognitive psychologists view people as having representations of the environment and taking action based on inferences with this knowledge. Although cognitive psychology has a long tradition and is a well-established research discipline, much of its insight is based on evidence from tightly controlled experiments in laboratory settings and it is questionable to what extent its results generalize to real-world environments [23].

Situated cognition dismisses this view, because it ignores the importance of the dynamics of system-environment interaction. Knowledge is embedded in a particular frame of reference. It is no longer viewed as a substance. Representations are not structures in the minds of people and knowing is not only possessing representations such as rules and algorithms. Knowledge representations are created while agents are interacting with the real world. Viewed in this way, they are external, perceivable structures, which are open to interpretation, negotiation, and debate.

Knowledge and its symbolic representations result from a sense-making process in which people attribute meaning to patterns of behavior in physical, informational, and organizational processes. In a real world setting the adaptive component of human behavior is far more important than static cognitive models can account for. When dealing with people in working environments, it is better not to view people as functional units, but as individuals who are involved in a sense-making process of their environment.

5 The Influence of Situatedness on Software Design

Situatedness gives us an understanding how (knowledge) representations can be related to practice, how they promote reflection, and why reflection is important. Situatedness focuses on the dynamic interplay between design participants, context, and technology.

5.1 Design Process

Software design first analysizes the working environment and the application domain. The current practice to focus primarily on data flows when designing software systems neglects that information is an interpretation of data, and that an analysis of the working environment, respectively the organization, in terms of data may only introduce the software designer's interpretation thereof. An interpretation that does not necessarily meet the client's objectives and requirements.

How can software designers cope with these difficulties? Certainly not, as we noted earlier, by relying solely on formal tasks. For example, people in the field of human-computer interaction that are concerned with user models, a field directly derived from artificial intelligence, often view workers as solely performing (formal) tasks and totally neglect the reality of work situations ([7], [15]). Some proponents of this idea even go as far as to introduce executable user models which make the involvement of workers redundant, even in the testing of the software. The idea that one can make models of users stems from the idea that the software designers' conceptualizations of the work are “correct” and do not need to be validated, except against the software designers' own standards. But workers will not function as these executable models predict, they adapt and change and attribute new meaning to their environment and ultimately it is this capability that makes them experts of the application domain, not the software programs. Careful software designers will therefore always check their conceptualizations with the workers and involve them in this process of conceptualization.

The ability of the participants to interact and coordinate behaviors is the foundation of the design process. The continuously increasing knowledge possessed by the design participants is used to improve the software system.

We clearly separate viewpoints emerging during design and we clearly specify whose point of view is assumed: the client's, the system's, the designer's. Viewpoints reveal how situations are framed by the perceptions of people as opposed to being defined as objective facts. Their comparison helps to investigate the advantages and limitations of each viewpoint. Moreover, the use of viewpoints prevents the software designer from monopolizing the decisions on what to represent and what not to represent in the system.

Situated software design reveals to design participants how (knowledge) representations are created and brings representations into the community of the workers. The development of representations, e.g. conceptual models, is a recurrent process. In the case of conceptual models, representations are adapted to meet the perceptions of customers and users. While the customers' objectives and requirements for a particular software system may be static, the users' perceptions are usually not. They are influenced by the whole of the users' working environment. The users[2] are situated agents. The situated agent is part of the environment and as such influenced by it, but the agent's actions in turn influence the environment.

In software design one must carefully decide which problems to investigate to which degree of detail, i.e. one has to find the correct balance between problems and their solutions. Failure to find this balance can lead to insufficiently investigated problems, or to elaborate solutions of non-problems. The correct balance is project-specific, and once found it is subject to revisions. Other projects and other situations may require to find a new balance. In other words, the design process itself is situated, e.g. there is considerable scope for situated variation in the way design methods and tools are applied.

5.2 Design Artifact

One implication of situatedness is that the design artifact should enable workers to employ a variety of approaches to their work. In practical terms, this means that software design should be directed toward developing software systems that are flexible, friendly, and that provide a variety of options cf. [17]. If this focus is adopted, a priori determination of the worker's behavior as a basis for identifying the most adequate software system becomes largely irrelevant.

In addition to the earlier arguments, there is an observation that supports this position. Many software systems have multiple users, either at one point in time or

[2] The very word "user" illustrates the limited view that software design has of those they design their products for. Therefore we will use the term worker, which captures that the computer is only one object they interact with.

across time because of job fluctuation. Thus, flexible rather than artificially constrained software systems seem necessary.

Although software designers can develop a software system that provides a variety of options, individual differences will undoubtedly influence the workers' preferences for the system. While organizational processes and structures will often suggest to the software designer that certain design options should be used, workers will have preferences concerning, at the very least, how and when they should be used.

Moreover, systems with features differing from the features of the organizational setting can lead to worker resistance to software systems. Although this is a necessary condition for worker resistance, it is not believed to be sufficient. In any given case, resistance may not occur for several possible reasons: people may like the changes embodied in the system, they may be too apathetic to resist, or they may find ways to go around the changes the system implies [20]. More accurate predictions are only possible when these additional ingredients of the situation are known and their (temporal) relations are understood.

Since every representation will always be incomplete, given the properties of the real world, the system is bound to fail at some point. We use expert systems to elaborate this point. One problem with expert systems is that they have a static representation of the application domain ([3], [28]). But more important the reasoning (or inference) mechanisms of expert systems freeze the interpretation of these representations. Therefore they do not allow for the reinterpretation of the data, which is necessary to cope with the changing environment. In a loan-assessment expert system, for example, data about companies are interpreted qualitatively with respect to the amount of loan for which they are eligible. The rules that are used to determine this “interpretation” in the system become outdated over time. For example, the economic situation of a company should be differently interpreted when the macro-economic milieu changes. But even more immediate smaller environmental changes, such as a visit to a company will change the economic expert's interpretation of it. Expert systems can by definition only reason about data represented in the program and implicitly contain a ceteris paribus clause. As the example illustrates, however, this ceteris paribus clause is often violated by only small environmental changes. The practical success of expert systems has been limited because their designers have not taken sufficiently into account that there is always a lot of information lacking in knowledge bases and that the interpretation of the data changes over time.

5.3 Design Reflection

Reflection is an integral part of software design. It is an ongoing activity to investigate what we are paying attention to, what is important, what we are trying to achieve.

As advertised by exploratory and compositional software development ([1], [13]), we should place into the working environment[3] stable components of the intended software system which can function on their own. Such components — partial implementations of the software system to be built — allow individuals to reflect on the application domain and to understand the software system implemented so far by experiencing it first hand. This experience can in turn be used to further guide the design process.

The introduction of software systems will not only change the environment of the workers, but also their perceptions of it. As software designers we have to be aware that this will happen and should observe and discuss the changes that software has brought.

However, perceptual changes do not only result when introducing a software system. Traditional accounting systems are good illustrations of this point. They reflect a historical situation when (economic) environments were perceived as rather simple, describable by two essential factors: price and quantity. Other factors, such as competition and skill, which have been recognized to be at least equally important, have lacked impact on the software system. Changing values in our society forced the development of, for example, personnel accounting or environmental exchange accounts, but changes in accounting software have been only modest so far cf. [26].

Moreover, it is important to realize that there is no single correct interpretation of a situation, although there is no reason that there cannot be a consensus on the interpretation. Therefore it is important that the software system must always be validated against the interpretations of the clients.

Although usability-labs may give good information on some inadequacies of the interface, the validity of software can only be established in its use. This can only be done in the workplace, where workers depend on the software to do their work and are under time-pressure to perform certain tasks. It is really only the productive system that is the proof of the pudding, not merely the implementation in a prototype [24]. Although prototyping is certainly a step in the right direction. So the key problem is how to conceive the successive production system — and that is done incrementally.

---

[3] Of course, it is not possible in great environments (e.g. design for thousands of users) or when standard software is developed to provide each user with a partial, though stable, implementation of the intended software system. Instead, a representative group evaluates design options.

Observations of the interactions with the software systems *over extended periods of time* are necessary to provide feedback for the design process. Examples are to provide automatic recording facilities (case recording, statistics, etc.) and to ask questions like: Which parts are used how frequently? Do these distributions change over time? Do they stabilize? Although the initial phase is crucial for the iteration to start, the real test is a long-term one. Studies only covering the time just before and after installation are insufficient.

6 Discussion and Conclusions

We have shown how the rationalistic perspective on software design leads to problems, since it largely neglects crucial aspects of the system-environment interaction. It is impossible to anticipate all changes. While the problems with the rationalistic perspective have been recognized within the software engineering community, they have not been analyzed appropriately. Consequently the proposed alternatives or remedies for software development are mostly of an ad-hoc nature, they have the form of “recipes” and lack theoretical rigor. This is true for both participatory approaches and CSCW-oriented work.

We therefore propose a new view on software design that takes into account that workers, which are to be supported with information technology, are situated agents. Situatedness provides the theoretical basis for deriving a design methodology — a framework for software system design we are working on.

One word of caution should be mentioned. Situated software design is not a “methodology” in the traditional software engineering sense, in which a choice is made between different means to a predefined end. Situated software design is a process in which the ends of a software system are subject to debate between clients and designers. It provides a theoretical framework that supports the understanding of design participants and the development of quality software systems.

Of course, the proof of the pudding is in the eating. Only after having applied the framework to several projects we can judge its quality. We do have some preliminary experience in projects on train scheduling and loan-assessment in a bank. Although the results are only of a preliminary nature they are highly promising and we will continue to pursue them.

Furthermore, a new method to develop software called Compositional Software Development [13] has been defined, and partially implemented. Compositional Software Development centers on heterogeneous executable compositions that are accessed by tools to modify them and to present them in various external representations thus accommodating multiple view points. The method does not prescribe a fixed order of development activities and can thus be adapted to various working situations.

7 References

[1] W. Bischofberger, G. Pomberger, *Prototyping-Oriented Software Development — Concepts and Tools*. Springer, Berlin, 1992.

[2] B.P. Bloomfield, "Understanding the Social Practices of Systems Developers". *Journal of Information Systems*, vol. 2, 1992, pp. 189-206.

[3] D.G. Bobrow, S. Mittal, M.J. Steffik, "Expert Systems: Perils and Promise". *Communications of the ACM*, vol. 29, no. 3, 1986, pp. 880-894.

[4] B.W. Boehm, *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, 1981.

[5] F.P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, 1975.

[6] R. Budde, K. Kautz, K. Kuhlenkamp, L. Mathiassen, H. Züllighoven, *Prototyping: An Approach to Evolutionary System Development*. Springer, Berlin, 1992.

[7] S.K. Card, T.P. Moran, A. Newell, *The Psychology of Human-Computer Interaction*. Erlbaum Hillsdale, New York, 1983.

[8] W. Clancey, "The Frame of Reference Problem in the Design of Intelligent Machines". In K. Van Lehn (ed.), *Architectures for Intelligence*. Erlbaum, Hillsdale, 1991, pp. 357-423.

[9] B. Curtis, H. Krasner, N. Iscoe, "A Field Study of the Software Design Process for Large Systems". *Communications of the ACM*, vol. 31, no. 11, 1988, pp. 1268-1287.

[10] P. Ehn, *Work Oriented Design of Computer Artifacts*. Arbetslivsecentrum, Stockholm, 1988.

[11] C. Floyd, W.-M. Mehl, F.-M. Reisin, G. Schmidt, G. Wolf, "Out of Scandinavia: Alternative Approaches to Software Development and Design". *Human Computer Interaction*, vol. 4, 1989, pp. 253-350.

[12] C. Floyd, F.-M. Reisin, G. Schmidt, "STEPS to Software Development with Users". In C. Ghezzi, J.A. McDermid (eds.), *European Software Engineering Conference*, Springer, Berlin, 1989, pp. 48-64.

[13] N.E. Fuchs, *Compositional Software Development*. Technical Report 93.06, Institute for Informatics, University of Zurich, 1993.

[14] J. Greenbaum, M. Kyng, *Design at Work: Cooperative Design of Computer Systems*. Erlbaum, Hillsdale, 1991.

[15] N. Hammond, M. Gardiner, B. Christie, C. Marshall, "The Role of Cognitive Psychology in User-Interface Design". In M. Gardiner, B. Christie (eds.), *Applying Cognitive Psychology to User-Interface Design*. John Wiley & Sons, New York, 1987, pp. 13-53.

[16] R. Hirschheim, H.K. Klein, M. Newman, "Information Systems Development as Social Action: Theoretical Perspective and Practice". *OMEGA International Journal of Management Science*, vol. 19, no. 6, 1991, pp. 587-608.

[17] G.P. Huber, "The Nature and Design of Post-Industrial Organizations". *Management Science*, vol. 30, no. 8, 1984, pp. 928-951.

[18] M. Jirotka, N. Gilbert, P. Luff, "On the Social Organization of Organizations". *Computer Supported Cooperative Work*, vol. 1, no. 1, 1992, pp. 95-118.

[19] K. Lamberts, R. Pfeifer, "Computational Models of Expertise: Accounting for Routine and Adaptivity in Skilled Performance". In K. Gilhooly, M. Keane (eds.), *Advances in the Psychology of Thinking*. New York, Simon & Schuster, in press.

[20] L.M. Markus, D. Robey, "Information Technology and Organizational Change: Causal Structure in Theory and Research". *Management Science*, vol. 34, no. 5, 1988, pp. 583-598.

[21] R. Miles, "Computer Systems Analysis: The Constraints of the Hard Systems Paradigm". *Journal of Applied Systems Analysis*, vol. 12, no. 2, 1985, pp. 98-115.

[22] E. Mumford, M. Weir, *Computer Systems in Work Design - The ETHICS Method*. Associated Business Press, London, 1979.

[23] U. Neisser, *Memory Observed*. W.H. Freeman, San Francisco, 1982.

[24] R. Pfeifer, H.F. Hofmann, E. Vinkhuyzen, P. Rademakers, "Problemlösende Systeme: Implikationen für die Validierung" (in German). *KI*, vol. 6, no. 4., 1992, pp. 71-73.

[25] R. Pfeifer, P. Rademakers, "Situated Adaptive Design". In W. Brauer, D. Hernandez (eds.), *Künstliche Intelligenz und Kooperatives Arbeiten*. Springer, Berlin, 1991, pp. 53-64.

[26] A.L. Slay, "Balancing the Books". *MacWorld*, no. 11, 1992, pp. 116-123.

[27] L.A. Suchman, *Plans and Situated Actions*. Cambridge University Press, Cambridge, 1987.

[28] B.J. Wielinga, A.T. Schreiber, J.A. Breuker, "KADS: A Modelling Approach to Knowledge Engineering". *Knowledge Acquisition*, vol. 4, no. 1, 1992, pp. 5-53.

[29] T. Winograd, F. Flores, *Understanding Computers and Cognition*. Ablex, New York, 1986.

[30] S. Zuboff, *In the Age of the Smart Machine*. Basic Books, New York, 1989.