Techniques and Tools for Continuous User Participation

Carola Lilienthal, Heinz Züllighoven
Department of Computer Science
University of Hamburg
Vogt-Kölln-Str. 30
D-22527 Hamburg
+49 40 54 94-2307/2414
{lilienth, zuelligh}@informatik.uni-hamburg.de

ABSTRACT

A widely accepted approach in supporting the development of new software is user participation. Many different techniques have been suggested that cover analysis and prototyping. However, software development in general includes changing and extending software that is already in use. In addition, the further development of existing prototypes and pilot systems is the basis for evolutionary software development. This paper presents an approach that supports user participation during the further development of existing software. Various techniques as well as a tool will be presented, that relate the documents used during analysis and initial design to the process of further software development.

Keywords

Use quality, prototyping, hyper documents

USER PARTICIPATION

User participation is a concept with several connotations. Since the end of the seventies there has been a strong focus in Central and Northern Europe on the political and emancipatory aspects of user participation (cf. Briefs, Ciborra, Schneider 1983, Docherty, Fuchs-Kittowski, Kolm, Mathiassen 1987), i.e., user participation is seen as a means of giving people a say in the conditions and equipment used in their workplaces. This interpretation of user participation has led accordingly to suggestions as to how to organize the software development process. The decision making process in particular should be organized in such a way that the end users' ideas about a new system be heard and accepted.

Use Quality and User Participation

As software engineers however, our understanding of user participation stresses a different aspect. We are primarily concerned with the quality of use in a process outcome – the application software. The use quality has some obvious application-oriented characteristics:

- functionality of the system should suit the tasks of the application area.
- the handling of the system has to be adequate for its users.
- the implemented sequence of the system's functions and operations should correspond with the sequence of the actual work processes and so on.

These "external" factors of use quality are based on "internal" software technical factors such as modular design, information hiding and abstraction (cf. Meyer 1988).

Nevertheless, use is unavoidably related to the tasks, work processes, concepts and terms of the application area. As software developers we have to accept that the real experts in the application area are those who are traditionally called users or end users. At this point, our understanding of user participation becomes apparent. Software experts and application domain experts need to work together in order to provide a useful and usable software product. One could question though, whether this corporation should still go under the heading of participation. Some have suggested the alternative term of user involvement. We believe that every software engineer has the professional responsibility in ensuring that all parties cooperate in developing a system where the quality of use is optimal.

This cooperation, however, doesn't occur without difficulties. Much has been said about the gap between the developers' world and the users' world (cf. Floyd 1987). Bridging this gap in software development entails realizing that the parties involved all have different perspectives (cf. Floyd, Züllighoven, Budde, Keil-Slawik 1992). The development process should therefore be organized so as to provide the techniques for making these perspectives explicit. In meeting these requirements a software project should result in a model which is an adequate representation of the application domain for all parties concerned. It would be desirable if this model were based on a "democratic consensus" among the different groups, but from our point of view this need not necessarily be the case.

With the quality of use as the central issue, it is more important that the model reflects the essential tasks and concepts of the application domain. The model should be the basis of discussion between the developers and the application domain experts. Therefore, software development is seen as a communication and learning process (cf. Floyd 1987). Note that this position does not negate the importance of user participation as a means of democracy at the work place, but it focuses on what seems to be the "lower level" engineering prerequisites. Put more simply, a software engineer's job is to provide a useful and usable software system. Or, if the system isn't relevant, adequate or suitable to its users, what is left of participation? R. Brooks (1996) voices a similar idea saying that software engineers should consider themselves as toolsmiths for their users, which suggests that the ultimate test for quality is use.

We have chosen an evolutionary, object-oriented approach, called the Tools and Materials Approach, that aims at providing software components with this degree of quality in use. User participation here essentially means integrating users and other relevant parties actively in the development and further development of software. The aim is twofold: firstly users provide their domain knowledge as the basis for the system's design. Secondly they provide feedback in analyzing, modeling and evaluating activities.

This paper proposes that this type of user participation shouldn't be restricted to the traditional software project notion which ends when the software is shipped and installed at the workplace. Software has to be further developed throughout its entire life cycle, i.e., as long as a system is used within an organization. Seeing this further development as an ongoing activity we have to provide means for maintaining the quality of use.

In the following, we will outline the essentials of the Tools and Materials Approach. This approach provides us with a frame of reference for the ensuing discussion on continuous user participation.

The Tools and Materials Approach

The Tools and Materials Approach can be seen as an application-oriented interpretation of object-oriented design (cf. Bäumer, Gryczan, Züllighoven 1995; Riehle, Züllighoven 1994). The essential point behind object orientation in the context of this paper is the close relationship between the tasks and concepts of the application domain and the software model. However, this modeling process is not geared towards work as a process itself. As Alan Kay (1977) says "do not automate the work you are engaged in, only the materials". Thus, application software should provide appropriate and flexible components to support the various ways of working.

This way of looking at software can be made more explicit by what we call a *leitmotif* or general guideline. A leitmotif, in general, should help developers and users to understand and design a software system. As our leitmotif we have chosen a workplace where a certain degree of individual responsibility is called for. This leitmotif becomes tangible with the help of a set of design metaphors which solidify the general guidelines in a pictorial way. Similar ideas have been discussed in Maaß and Oberquelle 1992. Design metaphors which have proven useful for application software in office-like environments are materials, tools, automatons and work environment (cf. Bäumer et al. 1995; Riehle et al. 1994).

The Tools and Materials Approach attempts to identify the relevant objects and means of work in order to design an electronic workplace. This workplace is equipped with a useful set of materials, tools and automatons (see Fig. 1), that aid in completing the task at hand.

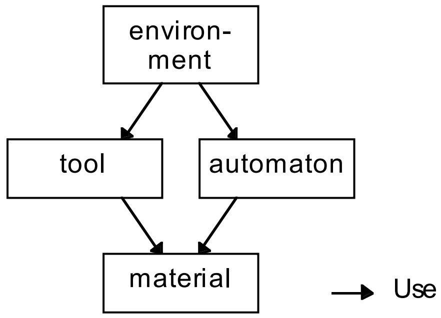

Fig.1: The metaphors

These central metaphors stem from the observation that in many work situations people make intuitive distinctions are made between those objects which are *worked on*, i.e., materials, and those which are the *means of work,* i.e., tools. These metaphors suggest certain use related characteristics:

- A *tool* (see Fig. 1) supports recurring work procedures or activities. It is useful for various tasks and aims.
- A tool is always handled by its user who decides when to take up a tool and what to do with it.
- *Materials* (see Fig. 1) are the objects of work which finally become the result or outcome of tasks. They incorporate "pure" application domain functionality.
- A material is worked on by tools according to professional needs. A material should be characterized by its potential behavior not its internal structure.

Not everything in an office environment is a tool or a material. There are certain cumbersome work routines a user wishes to delegate to a machine. To fulfill these requirements the metaphor *automaton* (see Fig. 1) was added. An automaton is started by a user and is active over a long period in the background. Once set and started it produces a predefined result without user interaction.

Tools, materials and automatons need to be presented and accessible to the user. There is always a place where work is done and where tools, materials and automatons can be found. The metaphor *work environment* (see Fig. 1) corresponds to this. Users should have their own work environment with its own arrangements of things and privacy.

Using metaphors for software design is not a new idea (cf. Carroll, Mack, Kellogg 1988). There has even been long discussion in Scandinavia on using the tool metaphor (cf. Ehn 1988). It should be noted, however, that we are offering a comprehensive set of design metaphors integrated within a uniform leitmotif. For each design metaphor there is a set of design patterns (in the sense of Gamma, Helm, Johnson, Vlissides 1994) describing the technical architecture of the appropriate software components (cf. Riehle et. al. 1994).

It should have become clear in the above that application-oriented quality in use is our central issue. It becomes, in fact, the primary consideration in user participation. We have outlined the leitmotif and the matching design metaphors which help to guide and shape the design of the future system. To allow user participation a set of documents is needed. These are presented in the following section.

DOCUMENT TYPES

Software development as described above should be seen as a communication and learning process. The need for application-oriented documents as the basis for this process should be fairly obvious. Hence, there seems to be a general consensus amongst those interested in the quality of use and user participation that new document types are needed (e.g. Carroll, Rosson 1990; Jacobson, Christerson, Jonsson, Övergaard 1992).

Appropriate Document Types

We have evaluated the various proposals and have selected a set of matching document types which fit our approach. The predominant prerequisite is that these document types be based on the professional language used in the application domain. In most cases they have to be written in prose. We have successfully used a set of application-oriented document types that are well-known under various names in the literature (cf. Bäumer et al. 1995):

- *Scenarios* (see Fig. 2) describing the current work situation, the everyday tasks and the objects and means of work. Scenarios are written by developers based on interviews with users and the various other groups involved.
- *Glossaries* defining and reconstructing the terminology of the professional language in the application domain.
- *System visions* (see Fig. 3) anticipating future work situations. They are comparable to use scenarios (cf. Jacobson et al. 1992) and are frequently supported by prototypes (cf. Lichter, Schneider-Hufschmidt, Züllighoven 1994).

In recent projects we have also successfully used cooperation pictures, a variation of rich pictures based on pictograms, to represent joint tasks (cf. Krabbel, Ratuzki, Wetzel 1996).

The crucial point in using these documents is the transition from the current work situation to the future system. While documents that describe current work situations (e.g. scenarios, glossary entries) can usually be discussed and evaluated by the application domain experts without major difficulties, discussing and evaluating the design of the future system is different. People tend to find it difficult to anticipate future ways of coping with tasks or handling a system. For this reason and not surprisingly prototypes play a central role in our approach (cf. Budde, Kautz, Kuhlenkamp, Züllighoven 1992).

An advisor fetches a *customer advice file*, looks for the required *product* in the index and opens the *file* at the desired spot. In addition to the customer advice file, there is a *form file* in which *standard forms* (e.g., *contracts* with third parties) are deposited, and a specimen file in which completion guides and *code sheets* are deposited.

Fig. 2: A scenario

Prototypes are tangible objects for anticipating future situations both from use-related and technical perspectives. It is, however still important that the emerging vision of the future system is documented beyond the actual prototypes. A prototype does not show the intended use context, explain design decisions or outline the anticipated handling of tasks. Therefore a set of different "subtypes" of *system visions* which relate to the different aspects of the future system is provided:

- *General visions* represent an overview of how a system's functionality is embedded in its context. The context can be both the technical environment and the use context. One general vision could describe for example to what extent existing tasks will be supported by the future system and which tasks will vanish due to rationalizing effects of the system.
- *Procedural visions* (see Fig. 3) present how individual tasks can be accomplished by utilizing tools, automatons and materials. Again technical and application-oriented procedural visions are written. While the technical visions describe the algorithms and state transitions of the software components the application-oriented visions portray a task from a use perspective.
- Within *component visions* the functionality of components is described without considering the tasks. There are three different types of component visions: tool visions, material visions, and automaton visions. Tool visions consist of tool-interface sketches and a description of the offered functionality. Material and automaton visions include the functionality of automatons and materials.

Scenarios, a glossary and system visions are employed to initiate the process of user participation. Since all documents are written in prose the users are able to understand and develop their own opinion of the various texts. This cooperation between developers and users can be increased by an appropriately organized development process.

An advisor opens a *customer advice file* by double-clicking the mouse, first selects all *products* from a visible table of contents, and then selects the desired *variant* from a second table of contents (also with a double-click) - thus also causing the corresponding sales help facility to be dispayed.

Fig. 3: A procedural vision

Setting Documents and Prototypes to Work

To get the learning and communication process going, it is important that the traditional life cycle strategies be substituted with an evolutionary concept of fast design and feedback cycles (see Fig. 4).

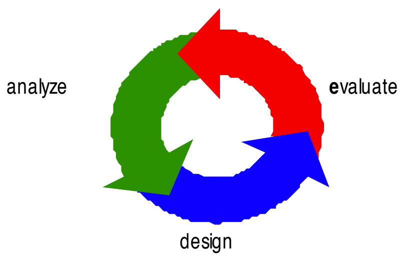

Fig. 4: Feedback cycles

These cycles come about by employing documents combined with prototypes (s. Fig. 5). For example developers interview users at their workplaces and prepare scenarios which are evaluated by the interviewees and other users. In designing a new system on the basis of these scenarios, system visions are written by developers and respective prototypes are realized. These prototypes are afterwards evaluated in workshops or "hands-on sessions" by the users.

It is important to note that during these feedback cycles any problems that occur should direct us to the appropriate documents that need modification (see Fig. 5). There is no predefined sequence of access to the

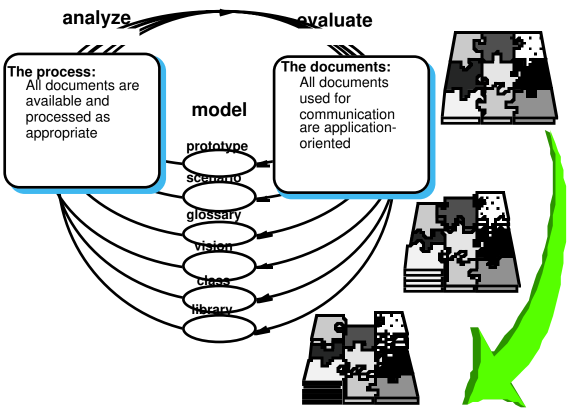

Fig. 5: The evolutionary process

documents; in principle, all are at any point available.

On the one hand the various documents are a necessary basis for the development of prototypes. On the other they are of equal importance for evaluating prototypes. Some advocates of "rapid prototyping" go as far as claiming that written requirements and design documentation are superfluous with prototyping. The domain knowledge and professional experience of the users seem to be a sufficient basis for prototype experiments and review discussions with developers. Experience shows however, that this is usually not the case.

A prototype represents only a partial solution for the software support needed by the user. To enable a sufficient evaluation of a prototype, the part of the application domain for which the prototype offers a solution has to become clear to the user. This part of the application domain has to be described in a text in order to make proper use of prototyping. Therefore scenarios, a glossary and system visions are important for evaluating prototypes.

A Net of Documents

Preparing and evaluating the various documents and developing prototypes in parallel leads to a lot of cross-referencing. A glossary and scenarios, for example, describe the concepts and work tasks found in the application domain under static and dynamic perspectives. In this way a glossary and scenarios are related to one another. When reading a scenario, a person will frequently want to access the corresponding glossary entries for terms used in the task description. At the same time these documents are the conceptual basis for writing system visions and building prototypes. Consequently, a net of potential relations among the documents has to be constructed and maintained.

This means that scenarios, a glossary and system visions should be organized as a hyper document. By using a hyper document the documents are not only the subject of joint evaluation but their relationships become clear and traceable. The net of documents shows the degree of communication during analysis and design. It also reflects the level of understanding of the application domain and the future system. By means of a hyper document the users are then able to reexamine this understanding. Misunderstandings between developers and users can already be identified by comparing the various linked documents.

In order to allow reasonable access to the hyper document, appropriate tool support is necessary. Typical tool support for on-line documentation is a hypermedia system, for example a WWW-server. These tools offer the technical support for linked document structures. Two other problems make the use of hypermedia systems seem advisable. First, it is usually difficult to keep linked documentation manually up to date. This problem is alleviated by keeping the documentation on-line. Secondly, during a software project a number of different document versions will be produced. These different versions can then be archived in the on-line documentation to allow reexamination of earlier positions and designs in the communication between developers and users.

FROM DEVELOPMENT TO USE

So far we have outlined a set of guidelines, techniques and document types which support the development of a new software system. This approach focuses strongly on the quality of use based on user participation and employs various documents and prototyping. Up to this point user participation forms an important part of the various other approaches (cf. Greenbaum, Kyng 1991). However once a system is shipped and installed at the users' site - participation seems to disappear.

The process of further development, occasioned by adjustments in the software to changing needs and requirements, is accomplished in a very different way. First of all, this process is rarely seen as development, but as maintenance. The users play a different and often more inferior role, being restricted to writing bug reports or proposals for new features. These reports are then added to a long list of backlogged change requests. Change requests are not usually handled by the original system developers but by maintenance programmers. They often lack the necessary application knowledge and know little about the history of the development process. Furthermore, the changes are described and dealt with on a primarily technical level as features (e.g., a new procedure or a new data field for a screen layout).

User participation, thus, isn't evident during the whole period of actual system use. Lack of insight into the necessity of user participation as well as an insufficient document basis are reasons for this. In the following, we will present the various aspects of our *continuous participatory* approach, present during the entire life cycle of a software system.

Continuous Participation

Discussion about prototypes between users and developers is from our point of view only part, albeit an important one, of user participation. By discussing a prototype with the user, the developer is able to see how a prototype is used and to gain hints about its usefulness. This information, however, is not acquired in an actual work situation but during experiments. The main focus of attention is in fact directed at evaluating the prototype, during which the user imagines various typical work situations.

Although a lot of problems and misunderstandings will already become clear during these initial stages, further cycles are necessary to ensure the usefulness of the software system. These are done with a small test group of users working with a pilot system. The users are left to go about their daily work with the pilot system and the on-line documentation.

This leads to a first productive version of the software system and installation at several work places. The users continue their appraisal of the system with the aid of the on-line documentation. Problems will surface but the focus will shift to further development of the system.

A software system can only remain useful over a long period of time when adjustments occur to suit the changing surroundings. Some changes are occasioned by new products or new regulations. Using the software system will stimulate users to see their work from a new perspective; new organizational ideas will occur as well as insights into which new concepts to introduce. These all form a crucial starting point for the further development.

Inclusion of the evaluation results as well as the users' new ideas in the further development of pilot and software systems require extension of the on-line documentation.

User Comments

The first extension allows the user to add to every text in the documentation. If a concept seems to be misinterpreted, a corresponding note can be made in a material vision or to the glossary. Difficulties with the handling of the software system can be commented on within the tool visions or scenarios. To explain a misunderstanding of various related concepts and work tasks the comments can form links to other documents.

The comments will consist of notes on problems arising with the tools, materials and automatons of the software system. The communication between users and developers will thus improve and be directed towards concrete parts of the software system described in the system vision. If misunderstandings aren't restricted to technical aspects but include problems on an application domain level, the glossary and scenarios will need to be discussed anew.

Relating software and documentation

The second documentation system extension concerns the strong relationship between the actual work situation, supported by the software system, and the documents accessible within the documentation system. A documentation system by rights shouldn't only be available as an additional element of the software system. Users are disinclined to search in the on-line documentation for the appropriate area to place their comments.

To improve document access we have included a context sensitive component. A system vision describing the corresponding part of the software system exists for each tool, material or automaton. When the user calls the documentation system, the context sensitive component identifies which tool or automaton the user is currently handling and which materials are involved. In accordance with this information - system visions, glossary entries or scenarios are displayed.

This section has introduced how extended on-line documentation serves users and developers. The context sensitive component is a step towards more effective user participation. Discussing the software system with the help of the documentation supports user participation in a way that would otherwise end with evaluating prototypes.

TOOL SUPPORT FOR USER PARTICIPATION

To realize our ideas about continuous user participation we have used WWW-servers in several university and industrial projects. The underlying concept of WWW-documents, though didn't quite meet our requirements. With the help of a WWW-server, developers can arrange documents for users to read and provide comments, however fulfilling our user participation criteria requires more support. In the following, the handling of the documentation system (see Fig. 6) will be described. This system has been implemented and used in university projects (cf. Lilienthal 1995).

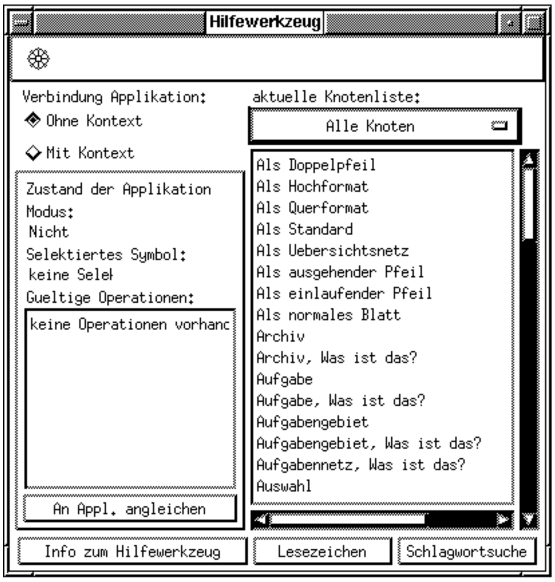

Fig. 6: The documentation system

Hypertext with comments

The glossary, scenarios and system visions are represented as nodes in a hypertext, and are linked to one another. Comments recorded by the users are also realized as nodes in the hypertext.

If a user decides to add to the documentation, a new hyper node is created. In order to outline which documents a comment refers to, users are able to create links between their comments and other hyper nodes. In addition, the documentation system allows the user to indicate whether a comment is for public or for private access. Public comments can be read by every user or developer who has access to the documentation system.

Orientation in a Hyper Document

A frequently reported problem with hypertext systems is the lack of orientation offered. Net-like structures only give users scanty clue in finding their way.

Fig. 7: A browser belonging to the documentation system

In employing the documentation system the user will open various browsers (see Fig. 7) to read system visions, glossary entries or scenarios. Editors will appear on the screen to be filled with comments. Each browser or editor offers more than just the text of a document or a comment and links to other documents. To support orientation some basic information is displayed: the name of the tool (browser or editor), the title, the type and the version of the document worked upon as well as the name of the author (see Fig. 7). This information enables users to distinguish between the documents on a higher level of abstraction than just textual. The users are able to identify different categories of documents and develop their own ways of working with them.

To equip users with more possibilities to work with and to gain orientation within the documentation system three other features were added. Users are able to mark hyper nodes and return to them later on (see Fig. 7). A search function allows access to documents by looking for a specific word (see Fig. 6). Each browser and editor is provided with a history, so that users can trace back to the hyper nodes already visited (see Fig. 7).

CONCLUSION

We have introduced concepts andtechniques to enable user participation during the entire software development process. Document types that encourage user participation during analysis and initial design have been linked to prototyping and further development. An on-line documentation system has been described. This documentation system offers possibilities such as comments, context sensitivity and several handling facilities to support users in evaluating pilot systems and software systems. On the basis of this, ongoing discussion between developers and users can be supported and organized more efficiently.

ACKNOWLEDGMENTS

We would like to thank Yvonne Dittrich, Charles MacInnes and the reviewers for their advice and help.

REFERENCES

Briefs, U., Ciborra, C., Schneider, I. (1983) (Eds) *Systems Design For, With and By the Users*. North-Holland, Amsterdam.

Brooks, F.P. (1996) The Computer Scientist as Toolsmith II. *Communications of the ACM*, 39, 3 (March 1996), p. 61 – 68.

Budde, R., Kautz, K., Kuhlenkamp, K., Züllighoven, H. (1992) *Prototyping*. Springer-Verlag, Berlin.

Bäumer, D., Gryczan, G., Knoll, R., Züllighoven, H. (1996) Large Scale Object-oriented Software Development in a Banking Environment. In: Cointe, P. (Ed.) *ECOOP '96 - Object-Oriented Programming*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 73-90.

Bürkle, U., Gryczan, G., Züllighoven, H. (1995) Object-Oriented System Development in a Banking Project: Methodology, Experience, and Conclusions. *Proceedings of HCI*, Vol. 10, pp. 293-336.

Budde, R., Züllighoven, H. (1992) Software Tools in a Programming Workshop. In Floyd, C., Züllighoven, H., Budde, R., Keil-Slawik R. (Eds.) *Software Development and Reality Construction*. Springer, pp. 252–268.

Carroll, J.M., Mack, R.L., Kellogg, W.A. (1988) Interface Metaphors and User Interface Design. In Helander, M. (Ed.) *Handbook of Human-Computer Interaction*, pp. 283–307.

Carroll, J.M., Rosson, M.B. (1990) Human Computer Interaction Scenarios as Design Representation. *Proceedings of the Hawaii International Conference on System Sciences*, *Los Alamitos CA* IEEE Computer Society Press, pp. 555-561.

Docherty, P., Fuchs-Kittowski, K., Kolm, P., Mathiassen, L. (Eds.) (1987) *System Design For Human Development and Productivity: Participation and Beyond*. North-Holland, Amsterdam.

Ehn, P. (1988) *Work–oriented Design of Computer Artifacts*. Almquist and Wiksell International, Stockholm.

Floyd, C. (1987) Outline of a Paradigm Change in Software Engineering. In Bjerknes, G., Ehn, P., Kyng, M. (Eds.) *Computer and Democracy*, Avebury, Gower Publishing Company Limited, Aldershot.

Floyd, C., Züllighoven, H., Budde, R., Keil-Slawik, R. (Eds.) (1992) *Software Development and Reality Construction*. Springer-Verlag, Berlin.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994) *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading.

Greenbaum, J., Kyng, M. (Eds.) (1994) *Design at Work. Cooperative Design of Computer Systems*. Erlbaum, Hillsdale.

Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G. (1992) *Object-oriented Software Engineering. A Use Case Driven Approach*. Addison-Wesley, Reading.

Krabbel, A., Ratuski, S., Wetzel, I. (1996) Requirements Analysis of Joint Tasks in Hospitals. In Dahlbom, B., Ljungberg, F., Nuldén, U., Simon, K., Sorensen, C., Stage, J. *Proceedings of IRIS 19 II*. (August 10-13, Lökegerg, Sweden) Gothenburg Studies in Informatics, Report 8, June, pp. 733-750.

Kay, A. (1977) Microelectronics and the Personal Computer. In *Scientific American*, Vol.237, No.3, Sepember, pp. 230-244.

Lichter, H., Schneider-Hufschmidt, M., Züllighoven, H. (1994) Prototyping in Industrial Software Projects - Bridging the Gap Between Theory and Practice. In *IEEE Transactions on Software Engineering 20*, 11, pp. 825-832.

Lilienthal, C. (1995) *Konstruktion und Realisierung eines an der Anwendung orientierten Hilfesystems nach der Werkzeug-Material Metapher*, Diploma Thesis, University of Hamburg, in German.

Maaß, S., Oberquelle, H. (1992) Perspectives and Metaphors for Human-Computer Interaction. In Floyd, C., Züllighoven, H., Budde, R., Keil-Slawik, R. (Eds.) *Software Development and Reality Construction*. Springer Verlag, Berlin.

Meyer, B. (1988) *Objet-oriented Software Construction*. Prentice Hall, Hemel Hempstead.

Riehle, D., Züllighoven, H. (1995) A Pattern Language for Tool Construction and Integration Based on the Tools & Material Metaphor. In Coplien, J.O., Schmidt, D.C. *Pattern Languages of Program Design*. Addison-Wesley, Reading, pp. 9-42.