ELSEVIER

International Journal of Industrial Ergonomics 16 (1995) 391–410

International Journal of
Industrial
Ergonomics

Benefits of user-oriented software development based on an iterative cyclic process model for simultaneous engineering

Matthias Rauterberg *, Oliver Strohm, Christina Kirsch

Work and Organisational Psychology Unit, Swiss Federal Institute of Technology (ETH), Nelkenstr. 11 CH-8092 Zürich, Switzerland

Abstract

The current state of traditional software development is surveyed and essential problems are investigated on the basis of empirical data and theoretical considerations. The concept of optimisation cycle is proposed as a solution for simultaneous engineering. The relationships of several different kinds of local optimisation cycles to the specifications, the communications, and the optimisation problem are integrated into a concept of participatory software development. Software development without integrated work- and task-organisational development is sub-optimal. User participation and prototyping significantly decrease cost and time over-runs. Process moderation methods (e.g. workshops) are time effective and lead to best results for the analysis of requirements.

Relevance to industry

There is a growing realisation that the construction, implementation and performance of modern software products crucially depends on getting the social and organisational issues right. The user-oriented approach outlined in the paper explicitly aims to present a process model that incorporates method and techniques for the requirements analysis. The practical problems of this iterative cyclic process model are discussed and recommendations are made to guide industrial managers, engineers and human factor specialists involved in the process of simultaneous production.

Keywords: User-oriented requirements analysis and data modelling; Human-centred technology; Simultaneous software engineering; Social and organizational issues; Iterative cyclic process model; Industrial practice

1. Introduction

Analysis of current software development procedures brings to light a series of weaknesses and problems. Some reasons lie in the theoretical concepts applied, the traditional procedures followed as well as in the use of inadequate cost analysis models (Yeh, 1991). These aspects point to the significance of participation by all groups involved in the development process (cf. Rauterberg et al., 1994). Analysis of these cases shows that there are three essential barriers to optimisation: the specification barrier, the communication barrier and the optimisation barrier. Generally, one of the most important problems lies in coming to a shared understanding by all the affected

* Corresponding author. Tel.: +41-1-632-7082; Fax: +41-1-632 1186; E-mail: rauterberg@ifap.bepr.ethz.ch

SSDI 0169-8141(95)00021-6

groups in the component of the work system to be automated. That is to say, to find the answers to the questions of 'if', 'where' and 'how' for the planned implementation of technology, to which a shared commitment can be reached. An optimal total system must integrate the social and the technical subsystems simultaneously. "The most appropriate way to engender a Concurrent Engineering culture is by forming crossfunctional teams. However, simply placing people from different departments together in the same room will not overcome these deeply ingrained cultural barriers. It the benefits of Concurrent Engineering are to be maximised, management must review the entire product introduction process and adopt radical changes to long established procedures" (Tucker and Leonard, 1994, p. 67). An iterative-cyclic process model for software development with several, simultaneous optimisation cycles is presented, and the strengths and weaknesses of user-oriented methods are introduced.

2. Problems of traditional software development

To arrive at the optimal design for the total working system, it is of paramount importance to regard the social subsystem as a system in its own right, endowed with its own specific characteristics and conditions, and as a system to be optimised when coupled with the technical subsystem. "Human resources are at the core of future growth and Europe's innovation capability" (C.E.C., 1989). The work task plays a central role in this, since it is the 'interface' between the organisation and the individual.

In the context of the our research project the general software development processes of different software companies ($N = 22$) were analysed as field studies by document analysis and interviews. These field studies were carried out in firms that develop information systems for offices and administration. Several other representative software projects were analysed only by questionnaires ($N = 83$). The analysis considered not only technical aspects, but also work organisation, use of methods, user participation and the problems connected with different procedures. The whole sample ($N = 105$) consisted of companies with internal software development departments focussed either on general software development activities (74%), or on banks or insurance business (15%), or industrial affairs (7%), or to other issues (4%).

Table 1
Actual problems in software development projects ($N = 75$)

	N
Organisational / methodological aspects	
Requirements analysis	17
Project complexity	13
Changes of requirements	11
Project management	6
Communication	3
Total	**50**
Strategic aspects	
Keeping of term	17
Keeping of costs	8
Total	**25**
Technical aspects	
Software engineering environments	15
Hardware constraints	7
Total	**22**
Social aspects	
Continuity of project members	8
Motivation of project members	2
Total	**10**
Qualification aspects	N
Qualification of project members	9
Total	**9**
Other aspects	
Software quality	6
External influences	4
Total	**10**

The central problems of all software projects in our sample are organisational or methodological aspects, strategical aspects, technical aspects, social aspects, qualificational aspects, or other aspects (see Table 1). The most important topics are problems which have organisational or methodical causes. One topic is the difficulty to define requirements. Because users usually do not take part in this stage, the importance of this

stage is underestimated and/or there is a lack of adequate methods. A field experiment was conducted as a 'real-life' software project in a German organisation to test and to compare the effectiveness and usability of specified communication methods for user participation.

2.1. The specification problem

The 'specification barrier' is a problem that is in the foreground even at a cursory glance. How can the software developer ascertain that the client is able to specify the requirements for the subsystem to be developed in a complete and accurate way that will not be modified while the project is being carried out? The more formal and detailed the medium used by the client to formulate requirements, the easier it is for the software developer to incorporate these into an appropriate software system. But this presumes that the client has command of a certain measure of expertise. However, the client is not prepared to acquire this before the beginning of the software development process. It is therefore necessary to find and implement other ways and means, using informal through semi-formal to formal specification methods. It would be a grave error with dire consequences to assume that clients – usually people from the middle and upper echelons of management – are able to provide pertinent and adequate information on all requirements for an interactive software system.

The integration of various distributed information systems or databases, as in the case of computer-integrated manufacturing (CIM), requires an integrated and consistent data model. This is necessary to assure a 'mutually agreed' basis for further specifications and a universal, reliable glossary of terms for the co-ordination and accuracy of all data entries, as well as updates and exchanges. The independent design of data models as a stable form of the logical representation of data has been recommended as a possible solution.

Hence, the process of data modelling is becoming increasingly important for the construction of database systems (Martin, 1989, p. 57). The quality of these data models is a critical factor for further stages of the design process. The first step in data modelling is information analysis, which attempts to analyse, clarify and define the terms of entity types used in an organisation or in a specified business area. An entity type is anything which an organisation usually stores information about, such as customer, supplier, machine tool, etc. For each entity type certain attributes are stored. Applications or procedures that use the entity types are constructed on the basis of the data model and change frequently, whereas the data model remains stable over time. Systems are easier to build and cheaper to maintain when thorough data modelling has been carried out (Martin, 1989, p. 73).

The business perspective on the "semantic universe" of the organisation of different users has much overlap, but divergences in corporate language invariably occur, referred to as "multiple vocabularies" (Shlaer and Mellor, 1988, p. 2). These inconsistencies – synonyms, homonyms or vagueness – have to be eliminated in the mutual efforts of the user, considered as an 'insider' (work expert) and the information engineer, as an 'outsider' (design expert) (see Udris and Ulich, 1987, p. 65).

2.2. The communication problem

The communications barrier between customer, user and end-user on the one hand and the software developer on the other is essentially because 'technical intelligence' is only inadequately embedded in the social, historical and political contexts of technological development (Mai, 1990). Communication between those involved in the development process can allow non-technical facts to "slip through the conceptual net of specialised technical language, which therefore restricts the social character of the technology to the functional and instrumental" (Mai, 1990).

Every technical language not only dominates the concrete process of communication in the speciality concerned, but also determines the cognitive structures underlying it. The application-oriented jargon of the user flounders on the technical jargon of the developer. This 'gap' can

only be bridged to a limited extent by purely linguistic means. To overcome this fuzziness requires the creation of jointly experienced, perceptually shared contexts. Beyond verbal communication, visual means are the ones best suited to this purpose. The stronger the perceptual experience one has of the semantic context of the other, the easier it is to overcome the communications barrier.

Because the languages of users and software engineers are quite different, techniques are needed to improve their communication. These communication techniques can be categorised as presentation techniques (natural language, formalised language, diagrams, etc.) and collection techniques (document analysis, questionnaire, interview, etc.). Abundant formalised models for the reproduction and visualisation of the idiomatic universe of an organisation as data models exist, e.g., Entity-Relationship (E-R) model, or Object-type model. However, which method is the most effective means to collect the information needed for entity analysis is a point in question.

2.3. The optimisation problem

As a rule, software development is a procedure for optimally designing a product with interactive properties for supporting the performance of work tasks. Software development is increasingly focusing attention on those facets of application-oriented software that are unamenable to algorithmic treatment. While the purely technical aspects of a software product are best dealt with by optimisation procedures attuned primarily to a technical context, the non-technical context of the targeted application environment requires the implementation of optimisation procedures of a different nature. Optimisation means the usage of all methods and techniques (even those of limited availability) within the context of an economical, technical and social process in such a way that the best result is achieved under the given constraints.

It would be false indeed to expect that at the outset of a larger reorganisation of a work system any single group of persons could have a complete, pertinent and comprehensive view of the ideal for the work system to be set up. Only during the analysis, evaluation and planning processes can be the people involved develop an increasingly clear picture of what it is that they are really striving for. This is basically why the requirements of the customer sometimes seem to 'change' – they do not really change but simply become concrete within the anticipated boundaries. This process of concretisation should be allowed to unfold as completely, as pertinently and as inexpensively as possible. Completeness can be reached by making sure that each affected group of persons is involved at least through representatives. Iterative, interactive progress in cross-functional teams makes the ideal concept increasingly concrete.

3. Analysis of traditional software development processes

3.1. The development process: An overview

Software projects are often realised with structured, linear stage models and defined milestones. The different tasks of software development can be assigned to the following stages: problem analysis, conception, specification, programming, test and implementation. Data modelling is accomplished during the early stages of the software development process, in the requirement specification or analysis phase. The mean effort of each stage is estimated by the percent-

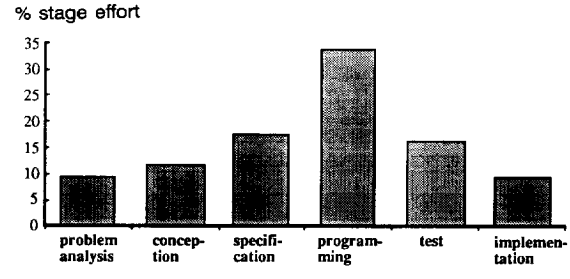

Fig. 1. Mean effort in the different steps of software development process ($N = 79$).

age proportion to the whole project effort without the maintenance phase (see Fig. 1).

3.2. The early stages: Analysis, conception and specification

The early stages are frequently the most neglected activities. This is essentially because methods and techniques need to be used primarily in the way occupational and organisational sciences have developed and applied them (Macaulay et al., 1990). Inordinately high costs are incurred from the troubleshooting required because the analysis was less than optimal. The time has come to engage occupational and organisational consultants at the analysis stage who have been specially trained for optimal moderation of software development processes. Introducing task orientation within the framework of socio-technological system conception makes the following conditions indispensable (Ulich, 1994):

(1) The employees must have control over the work process as well as the necessary resources.
(2) The structural features of the task must be such as to release in the working person the energy for completing or continuing the work.

Task planning is therefore the focus of attention in the analysis phase. The five features, Completeness, Variety of tasks, Opportunity for social interaction, Personal autonomy, and Opportunity for learning and development, must be addressed to plan the tasks suitably (Ulich, 1994).

Once the analysis of the work system to be optimised has been completed, the next stage is to mould the results obtained into implementable form. Methods of specification with high communicative value are recommended here.

(i) The specification of the organisational interface: The first thing is to determine 'if' and 'where' it makes sense to employ modern technology. Although the view is still widely held that it is possible to use technology to eliminate the deficiencies of an organisation without questioning the structures of the organisation as a whole, the conclusion is nevertheless usually a false one. It is important to understand the work system as a living organisation, as a self-sustaining organism, which must develop and change to reach the organisational aims. The purpose of defining the organisational interface, from this point of view, is to improve the viability of the organisation with the help of modern technology. An unavoidable consequence is that the necessary measures must be taken in such a way that the ease with which the employees can assimilate and adapt to the type of the organisation is maximised. The effects of the organisational measures undertaken can be assessed, for example, by means of the 'Activity Evaluation System' or 'Activity Evaluation System for Intellectual Work' (Ulich, 1994).

(ii) Specification of the tool interface: The intended division of functions between human and machine is decided during the specification of the tool interface. The tasks that remain in human hands must have the following characteristics (Zölch and Dunckel, 1991):

(1) sufficient freedom of action and decision-making;
(2) adequate time available;
(3) sufficient physical activity;
(4) concrete contact with material and social conditions at the workplace activities;
(5) actual use of a variety of the senses;
(6) opportunities for variety when executing tasks;
(7) task-related communication and immediate interpersonal contact.

(iii) Specification of the input / output interface: Once those concerned are sufficiently clear about which functions are amenable to automation, the next step that should be taken is to develop and test the screen layouts on the end-users with inexpensive, hand-drawn sketches. The use of prototyping tools is frequently inadvisable, because tool-specific presentation offers too restrictive a range of possibilities. The effect of the design decisions taken can be assessed with the help of discussion with the end-users, or by means of checklists.

(iv) Specification of the dialogue-interface: The use of prototypes to illustrate the dynamic and interactive aspects of the tools being developed is indispensable for specifying the dialogue interface. But prototypes should only be used very purposefully and selectively to clarify special aspects of the specification, and not indiscrimi-

nately. Otherwise there looms the inescapable danger of investing too much in the production and maintenance of 'display goods'. A very efficient and inexpensive variation is provided by simulation studies, for example, with the use of hand-prepared transparencies, cards, etc. which appear before the user in response to the action taken (Karat, 1990).

More and more companies try to practice user participation in the software development process. Statements of software engineers like "we must have the participation of the users" or "without users it does not work" and the selection of the users to participate in the project because of their professional background show that user participation as a necessity is becoming more and more accepted. Primarily, this concerns projects in which the system implementation is combined with greater organisational changes of the work system. The development of the data model especially requires the co-operation of software engineers and the employees of an organisation as potential users to analyse the particular business terms, to specify the names of the entity types and to create the definitions for the data elements. The participation of users in the information analysis phase is crucial for the adequacy of entity type definitions and later acceptance of the data model.

We differentiate user participation into three categories: active or passive participation, or without any participation. Further and more detailed analysis of our data show the following results (Strohm, 1991; Strohm and Ulich, 1991). In 29% of the projects ($N = 83$) *active* user participation was practised. This means that the users had decision possibilities, were frequently consulted regarding problems of task design, functionality, dialogue design, etc., and were involved in the early stages. In 57% of the projects a *passive* form of user participation was practised. This means that the users gave information, evaluated the ideas of the software engineers, but were not so deeply involved in the early stages. Fourteen per cent of all projects were realised without any kind of user participation.

Software developers put prototyping as one user-oriented method more and more into practice. The prototyping method was used in 55% of the projects ($N = 83$). Of all projects with prototyping ($N = 46$), most software engineers evaluated this procedure as 'very useful' (59%) or 'useful' (33%). These software engineers said that prototyping is primarily a good method to support the co-operation and communication with the users.

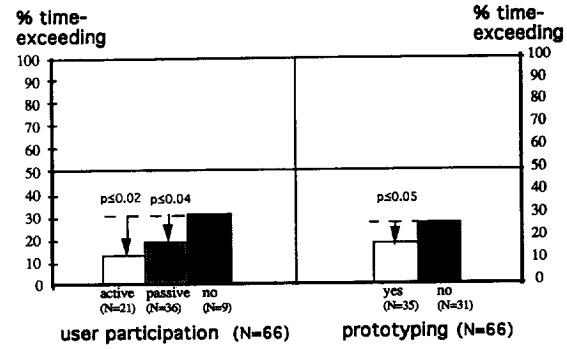

Fig. 2. The relation of cost over-runs with user participation and prototyping; 'cost exceeding' is the percentage cost proportion of total project budget.

Active user participation decreases significantly in relation to the cost over-run portions ('active' versus 'no participation' $p \leq 0.03$, one-tail T-test; $N = 59$; see Fig. 2), as well as the time over-run portions ('active' versus 'no participation' $p \leq 0.02$, one-tail T-test; $N = 59$; see Fig. 3). The usage of prototyping decreases significantly with cost over-run ($p \leq 0.04$, one-tail T-test; $N =$

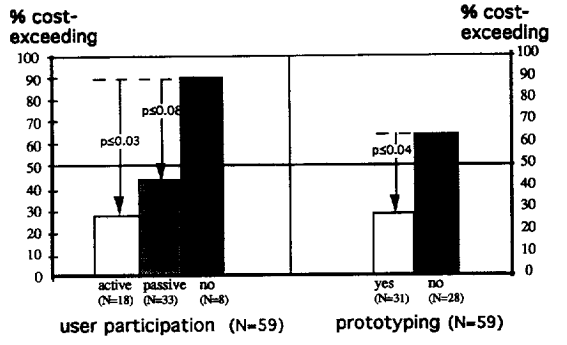

Fig. 3. The relation of time over-runs with user participation and prototyping; 'time exceeding' is the percentage elongation as a proportion of the total project time.

59; see Fig. 2), as well as time over-run ($p \leq 0.05$, one-tail T-test; $N = 66$; see Fig. 3).

Fig. 2 and Fig. 3 show that there are significant advantages in projects with active user participation and with usage of prototyping. We could not find a significant correlation between the category of user participation ('active', 'passive', 'none') and the project size ('small': ≤ 5 man years, 'big': ≥ 6 man years; χ^2-test, $df = 2$, $p \leq 0.18$); there is also no significant correlation between prototyping ('yes', 'no') and project size (χ^2-test, $df = 2$, $p \leq 0.84$).

We distinguish among four project types of software developments (Strohm, 1991): Type A is a singular software product by in-house development for a specific in-house department; Type B is a singular software product for external customers; Type C is a trade product for external companies; Type D is standard software for unknown users. Active user participation seems to be typical for project type A, passive user participation for project type C and no participation for project types B and D (χ^2-test, $df = 6$, $p \leq 0.07$).

3.3. The programming and implementation stages

The programming stage is made up of the following three steps (Boehm et al., 1981):

(1) design of the program architecture;
(2) design of the individual program modules (object classes, etc.);
(3) coding and debugging.

The distinction between design and specification is important. During specification, all relevant properties of the technical subsystem are defined as precisely as possible. In the programming stage all care must be taken to ensure that the technical subsystem being developed exhibits these properties to the greatest possible extent. It is pure software expertise that is of primary importance here. The implementation phase is characterised by the first tests of the software system in the concrete working context.

Once a working version is available, it can be put to test in usability studies ('user-oriented benchmark tests', see Rauterberg, 1991) in concrete working situations. This is the first place where it is possible to clarify the problems with the actual organisational and technical environment. In contrast with laboratory studies, field studies take into account the aspect of 'ecological validity' (Karat, 1990). Trials with real work tasks make it possible to check and assess the degree to which the planned organisational ideal has been reached. Although video is the data-recording medium preserving the most information, a combination of log-files and direct protocol capture makes a good compromise between performance and economy.

3.4. The maintenance phase

The mean effort for maintenance is 20% ($N = 55$). 33% of the maintenance effort is spent on debugging; hence 67% of the maintenance effort is needed for changing the systems (e.g. changed requirements caused by users). The cumulated effort for the early stages 'problem analysis', 'concept' and 'specification' has a significant, negative correlation ($r = -0.32$, $p < 0.05$, $N = 55$) with the effort for maintenance. This means that adequate effort in the early stages reduces the cost-and time-intensive repair and correction tasks in the maintenance phase (see also Boehm, 1981).

4. Communication methods for user participation

Ample evidence is available that shows the importance of user participation in software development. However, which method is the most effective means to collect the information required is a point in question. Questionnaires, interviews or workshops can be used as empirical techniques to improve the communication between software specialists and work experts. Financial and time constraints often force the designer to communicate with the end-users through interviews or questionnaires (Shlaer and Mellor, 1988, p. 86). These methods are characterised by the fact that they create as many differing definitions as the number of end-users involved. Workshops are recommended by many information engineering specialists (e.g. Oppermann, 1983; Martin, 1989), but are difficult to establish be-

cause at first glance they may seem to be time-consuming and a waste of personnel resources. Their advantage is that the inter-user communication reveals inconsistencies and vagueness in the use of the business terms and leads to a 'mutually agreed' definition of the entity types. This also prevents the engineers from exerting too much influence on the definition of entity types that could result in reduced acceptance of the semantics by the users. The purpose of this study was to investigate which of these information collection techniques provides the best means to improve the collaboration.

4.1. User participation in data modelling: A field experiment

In a co-project at Lufthansa AG (Germany) one researcher and two software engineers were entrusted with the design of a data model for specific business areas (personnel administration and office communication). To accomplish this task, three different communication techniques (questionnaire, interview and workshop) were used concurrently in three user groups for the information and requirement analysis. The goal was to empirically investigate and compare the efficiency of the above methods. As a first step in information analysis, a list of business terms extracted by a document analysis was scrutinised. The task was to determine whether each term was an entity, an attribute or a value of an attribute. Then the participants had to clarify and determine the definitions of the entity types and integrate them into a consistent and reliable data model.

A total of 18 employees from the participating departments were selected as users. They had comparable qualification levels and their tasks were related to the semantic world that was to be designed as a data model. With each method, six randomly assigned employees from the participating departments accomplished the same analysis task. During information analysis the motivation of the participating users, the amount of time spent on the analysis as well as the rating of the effort needed for preparation, execution and completion, were monitored.

As representation technique 'OrgMap' (Kirsch, 1992, p. 70) – a combination of a simplified data model and Metaplan technique – was used for visualisation. Hand drawn diagrams are difficult to revise and change, hence are not favoured as a communication means for the mutual analysis process. E-R-diagrams, on the other hand, impose additional cognitive strain on the participating users who are already concerned with the task of analysing and defining the corporate language. The demand of communicating in a highly stylised and unaccustomed language can lead to cognitive overload and interfere with the primary task.

The statements for the entity type definitions collected by the three groups constituted the basis for the construction of the data model by the engineers. The data model was designed as an E-R model, with the ICASE-Tool Information Engineering Workbench (IEW). The engineers graded all the statements as being either correct, incorrect, or a contribution to the detection of defective business terms (e.g. synonyms, homonyms, etc.). The final data model was then evaluated by the employees of the participating departments.

4.2. Efficiency of communication methods for user participation

The developed data model – a collection of definitions of entity-types, their attributes and relationships – was used to evaluate the results of the information analysis. The workshop led to the best results especially when – besides the quality of the entity type definitions – the effort for information analysis is taken into account. The motivation of the participating users was highest in the workshop and lowest with the questionnaire. However, the results of the evaluation questionnaire are difficult to interpret because of possible bias due to a low proportion of returns. Only users who were experts in the specific business area responded to the written questionnaire. Inference statistics were not administered because of the restricted sample and scope of the investigation.

The results of the evaluation of the business term definitions are shown in Fig. 4. The number

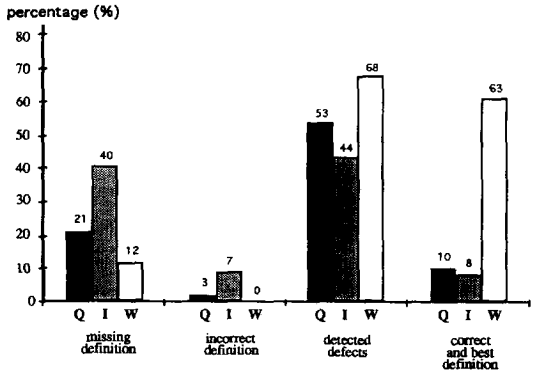

Fig. 4. Quality of entity type definitions by method (Questionnaire (Q), Interview (I), Workshop (W)). Percentage of missing definitions, incorrect definitions, correct definitions and detected defects.

of missing or incorrect definitions was smallest in the workshop group and highest in the interview group. As was hypothesised, correct definitions and detected defective terms were higher in the workshop group than in the interview or questionnaire groups.

The time invested in information analysis for the compared communication methods is listed in Fig. 5. On average, for all the participants – engineers and users – there is no significant

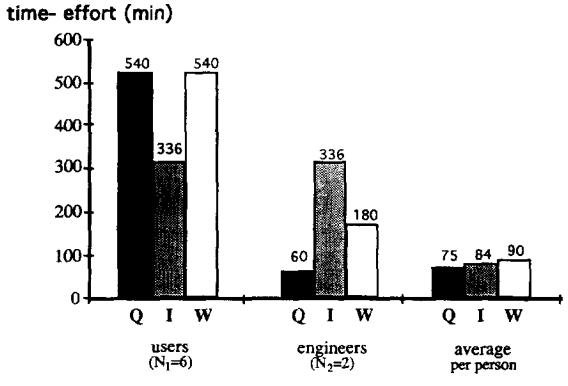

Fig. 5. Time effort required for information analysis. Average in minutes by communication method (Questionnaire (Q), Interview (I), Workshop (W)) used in the specific group for all group members (average) and for the subgroups of employees (user) and data administrators (engineer). ['average' = ('users' + 'engineers')/$N_1 + N_2$)].

Table 2
Motivation of the employees for participation. Self-rating scales from none (0) –low (1) –middle (2) – high (3). Sum (Σ motivation) and average (ø motivation) for each method is listed

Method	N	Σ motivation	ø motivation
Questionnaire	6	2	0.33
Interview	6	14	2.33
Workshop	6	18	6.00

difference in the amount of time that was needed to analyse the business terms.

A more differentiated comparison shows differences in the amount of time for users and engineers – workshops requiring more hours by users, less by engineers and questionnaires vice versa. Subjective ratings of motivation to participate were highest in the workshop and lowest in the questionnaire group (see Table 2).

As a communication method for the collaboration of users and engineers in information analysis, the workshop is to be recommended. It requires the least effort, resolves conceptual discrepancies quickly and evolves valid and best definitions. Motivation of the participating employees and later acceptance of the entity type definitions is highest. Interviews find partially overlapping semantic perspectives with separate and sometimes conflicting vocabularies. Differences must be resolved with additional effort. Questionnaires are characterised by low motivation for usage.

5. An iterative-cyclic process model for software development

Sufficient empirical evidence has accumulated by now to show that task- and user-oriented procedures in software development not only bring noticeable savings in costs, but also significantly improve the software produced (Boehm et al., 1981; Karat, 1990; Peschke, 1986; Rauterberg, 1991; Spinas and Ackermann, 1989). How then, can the problems mentioned above be solved?

5.1. The concept of optimisation cycles

Systems theory distinguishes between 'control' (i.e. 'feed forward' or 'open loop' control systems) and 'regulation' (i.e. 'feedback' or 'closed loop' control systems). The following are minimum conditions in the case of 'control' for the output to adjust as intended to the reference input:

"(1) precise knowledge of the response of the system being controlled, i.e. of the relation between the controller output on the one hand and the output and interference – such as changes in the specifications – on the other;

(2) precise knowledge of those quantities whose effect on the system is detrimental to the intended influence (interference or perturbance, such as technical feasibility, etc.); if the system has a response delay, then a prognosis is needed for these interferences at least for the period of the delay;

(3) knowledge of procedures for deriving controller output from such information.

These conditions are hardly ever met in practice. That is why it is constantly necessary to supplement or replace control by regulation" (Schiemenz, 1979, p. 1024).

Floyd and Keil (1983, p. 144) and Peschke (1986, pp. 143ff.) come to the same conclusion in the context of software development procedures.

The application of the highly effective 'regulation' control principle actually only requires a knowledge of those controller outputs which steer the output in the desired direction: "In regulation, the control apparatus takes into account the results of the control procedure. There is thus a feedback from the output of the regulated system to the input of the regulating one" (Schiemenz, 1979, p. 1024).

It is already quite common to prescribe feedback between two successive phases (Sommerville, 1989, p. 10; Boehm, 1988).

The distinction between boundary constraints on the one hand and interference on the other is that the boundary constraints are deliberately and purposefully set (e.g. development costs, duration of project, etc.), whereas interference is unintentional and unanticipated (see Fig. 6). We designate the 'Action-Test-Cycle' as an optimisation cycle (see also Deming, 1989; AMI, 1992; Rauterberg, 1992). An important dimension of the optimisation cycle is its *length*, i.e. the time required to complete the cycle once. Depending on the nature of the activity and the testing, the length can be anything from a matter of a few seconds to up to possibly several years. The longer this period is, the more costly is the optimisation cycle. It is the aim of user-oriented software development to incorporate an optimisation cycle that is as efficient as possible into software development procedures (Nielsen, 1989, 1990; Karat, 1990).

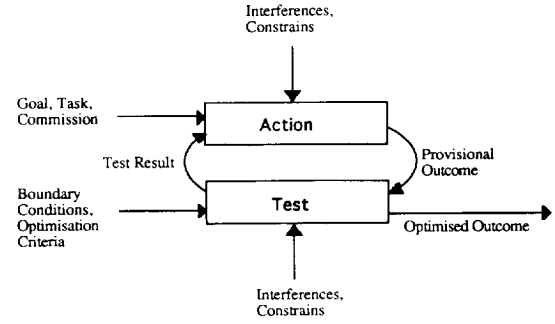

Fig. 6. The optimisation cycle in the context of an iterative cyclic concept for software development (see also Peschke, 1986, pp. 143ff. and Deming, 1989).

If the general scheme of regulation is applied to software development, then the individual components of regulation are assigned as illustrated in Fig. 6. The *optimisation criteria* are all relevant technical and social factors. Testing ascertains the extent to which the optimisation criteria are met, subject to the boundary constraints. The action taken could come from a range of extremely different procedures, methods or techniques. All of this depends on the nature of the work output. Interference could come from the three barriers discussed above as well as from technical and/or social problems in realising the project.

Of course current software development also avails itself of the principle of 'regulation' in various places. What we have in mind here are decisions made and directives issued by the client,

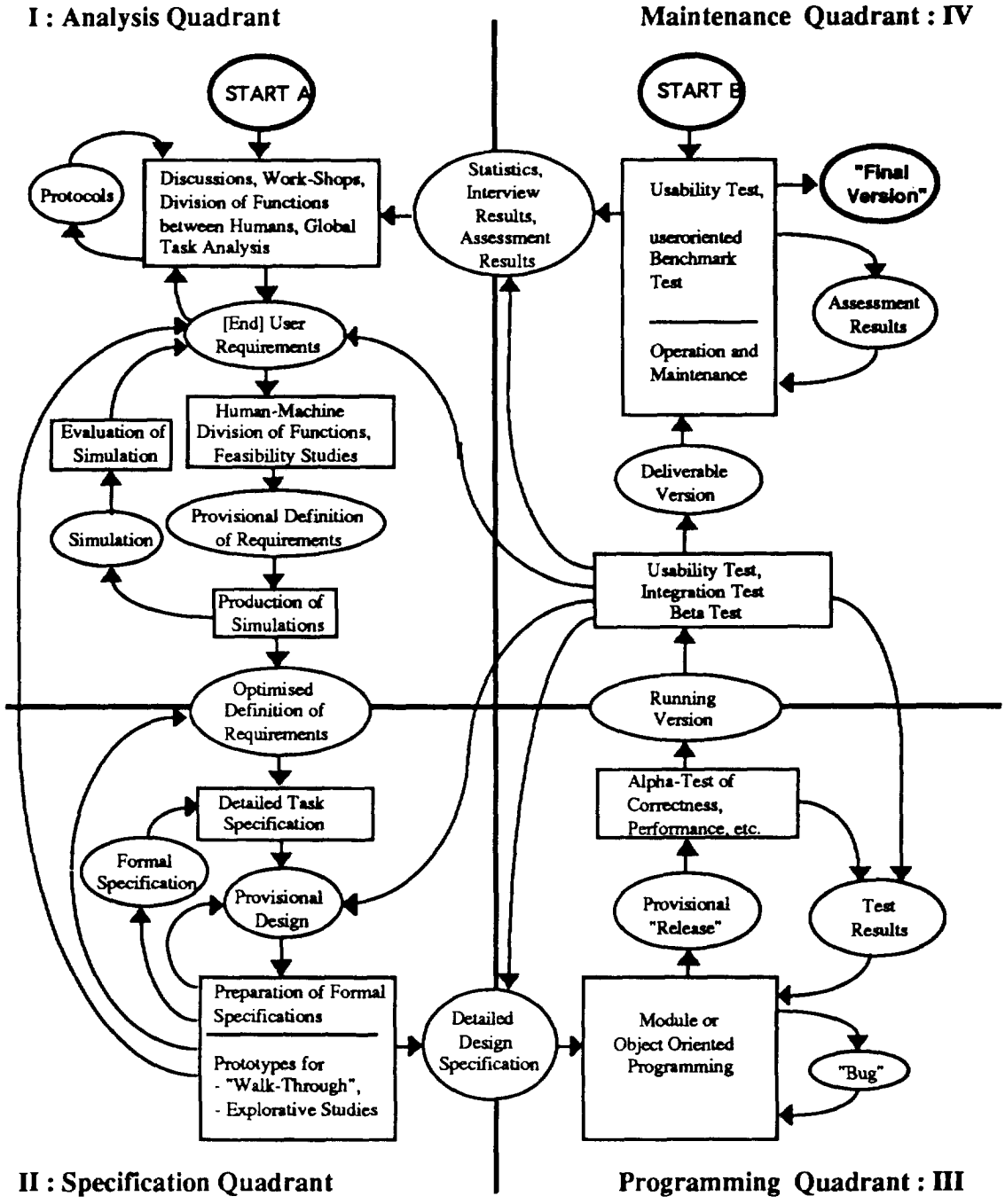

Fig. 7. Flow chart for a participatory software development model showing the local optimisation cycles within and between individual quadrants (I–IV).

the project management or other bodies as a consequence of experience, ignorance, exercise of power or purely and simply the pressure of time. "It is frequently the case that control systems operate more economically than (even possible) regulation systems" (Buhr and Klaus, 1975, p. 1035) – but only if the conditions mentioned above prevail! This is also the important reason why the attempt is made to come as close as possible to a particular control system, namely the 'Waterfall Model'. If, however, the barriers discussed above are taken seriously, then we must determine those places in software development procedures at which optimisation cycles are *indispensable* (see also Peschke, 1986, pp. 143ff.; Rettig, 1991).

5.2. Embarking on the global optimisation cycle

The type of software to be developed has proved to be one of the essential factors governing software development. The global optimisation cycle begins at Start-A of Fig. 7 when developing completely new software and at Start-B in the case of further development and refinement of existing software. Different concept-specific local optimisation cycles are used to optimise specific work tasks, depending on the particular type of the project at hand. It is up to the project management to settle on the actual procedure and this decision is reflected in the development form chosen.

5.3. Global and local optimisation cycles

The use of optimisation cycles in software development procedures depends on the following conditions being met (Peschke, 1986): "(1.) A modified project management model, which guarantees above all communication between those concerned and the developers. (2.) Computer supported version and documentation management, which includes also the results of evaluation and current criticism. (3.) Informing all those involved about the project's aims and the peculiarities of the procedure, as well as training the employees concerned. (4.) The fundamental willingness of the developers to produce incomplete software and to accept critique of it. (5.) The expansion of the expertise of the developer beyond purely data processing technology as regards measures in work structuring. (6.) The use of a largely integrated software tool environment,

Table 3
Survey of different methods of user-participation in the framework of optimisation cycles

Method	Action	Test	Outcome	Cycle-length
Discussion-I	Verbal communication	Verbal interpretation	Global design decisions	Seconds–minutes
Discussion-II	Meta-plan, Flip-charts, etc.	Visual and verbal interpretation	Specific design decisions	Minutes–hours
Simulation-I	Sketches, scenarios, 'Wizard of Oz', etc.	Visual and verbal interpretation	Specification of the input/output interface	Minutes–days
Simulation-II	Draughting of structural blueprints, etc. with (semi)-formal methods, e.g. 'OrgMap'	Visual and verbal interpretation with suitable qualification	(Semi)-formal descriptive documents. data modelling	Hours–days
Prototyping-I	Horizontal prototyping	'Thinking aloud', 'Walk-through'	Specification of dialogue component	Days–weeks
Prototyping-II	Partial vertical prototyping	Heuristic evaluation	Partial specification of application component	Days–weeks
Prototyping-III	Complete vertical prototyping	Task-oriented benchmark tests	Specification of application component	Weeks–months
Versions-I	First run through entire development cycle	Formative usability tests	First largely complete version	Months–years
Versions-II	Consecutive run through entire development cycle	Summative usability tests	Several largely complete versions	Months–years

which supports the developer in repeated preparation and modification of the software. (7.) The preparedness of all persons involved to learn throughout the course of the project".

Even if we assume that all the conditions listed are more or less fulfilled, there still remains the question of how to actually carry out the software development project. To reach the goals of a work-oriented design concept the first project phases (requirement's analysis and definition; Quadrant-I in Fig. 7) should be replete with a range of optimisation cycles.

6. Methods for user-participation

Simple and fast techniques for involving users include discussion groups with various communication aids (Meta-plan, layout sketches, 'screen-dumps', scenarios, etc.; see Spinas and Ackermann, 1989; Ehn and Kyng, 1991), workshops for determining the attitudes, opinions and requirements of the users, the 'walk-through' technique for systematically clarifying all possible work steps, as well as targeted interviews aimed at a concrete analysis of the work environment (Grudin et al., 1987; Macaulay et al., 1990). Very sound simulation methods (e.g., scenarios, 'Wizard of Oz' studies) are available for developing completely new systems without requiring any special hardware or software. Spencer (1985) presents a summary of techniques for the analysis and evaluation of interactive computer systems (see also Crellin et al., 1990). Comparative studies, e.g. user-oriented benchmark tests (Lewis et al., 1990; Rauterberg, 1991) and usability studies (Dumas and Redish, 1993), can be undertaken after the second time though, when working with a version concept, for then there are at least two versions available.

We used systems-theoretical considerations to explain how each optimisation cycle consists of an action and a test component that are suitably coupled (Fig. 6). Each component can be of a widely varying nature. Table 3 provides an overview of the main focus of effort, of the nature of activity, and of the tests, the outcome and the expected range for the length of the cycle. The shorter an optimisation cycle is, the more rapidly – and therefore the more often – it can be used to reach a truly optimal result.

A major problem in assuring adequate meshing of different, concurrent optimisation cycles is *synchronisation* (see also Stovsky and Weide, 1991). If several optimisation cycles are simultaneously active at different places in the iterative-cyclic process model (Fig. 7), then these concurrent cycles must be suitably synchronised. This is particularly important, being the only way to minimise inconsistencies within the overall development process. If, for example, there are additional consultations with the user and if specification analyses are undertaken parallel to the implementation phase, then it could easily happen that the programmers end up writing programs for the wastepaper basket because they are working to specifications that are superseded. This problem is caused by the differing lengths of the optimisation cycles involved and it becomes very evident whenever the separate optimisation cycles have not been adequately synchronised.

By the *functional synchronisation principle* we mean fixing the sequence of the individual optimisation 'quadrants' in the sense of a functional phase distribution. This principle is used primarily in the Waterfall Model (the 'milestone' concept) and leads necessarily to a dramatic increase in the total length of the cycle when applied exclusively to the version concept.

One way to at least partly overcome this drawback is to use the *informational synchronisation principle* – appropriate information links are established between the various optimisation cycles, so that every person in each cycle is kept informed about the current state of the cycles that are active in parallel (Mambrey et al., 1986, pp. 79ff.). This can be achieved using such simple aids as document folders at a fixed location and regular conference times. But technical support can also be used (mailboxes, version data banks, information repositories, etc.) (Jones, 1987, p. 193). Recent results from research into computer-supported co-operative work (CSCW) can be applied to the work environment of the developer (Brothers et al., 1991).

Another important synchronisation principle is

to ensure that participation in the different optimisation cycles is by the *same* set of people (see also Floyd and Keil, 1983, pp. 171ff.). However, this principle often flounders on organisational forms based on the division of labour, which is frequently encountered in software companies. These software development divisions require a reorganisation according to the occupational psychological criterion of 'completeness' of work tasks (Ulich, 1994). Since nobody can be in two places simultaneously, we call this principle the *personal synchronisation principle*.

6.1. Strengths and weaknesses of user-oriented methods

We list some further aspects to be considered when applying the various participatory methods, going beyond their influence on the length of the cycle. A method is fast, if the time to execute a complete cycle is short ('length of the cycle').

Discussion methods I and II

Discussion is the method most frequently used, because it is fast, familiar and to a certain extent informative (see the 'Communications barrier' section). An optimisation cycle in the discussion method is limited to communicative units such as 'statement–counterstatement', 'question–answer', etc. But because it rests essentially on purely verbal communication, a series of misunderstandings can arise which often never come to light or only do so when it is already too late. Discussion must therefore be supplemented with methods using visual communications techniques.

Simulation methods I and II

Simulation methods comprise all techniques that illustrate the work system to be optimised in as realistic, visually perceivable a way as possible. These range from simple, quickly completed sketches, through template layouts (Ackermann, 1987; Ehn and Kyng, 1991) and semi-formal methods (Kirsch, 1992) to formal description techniques (SADT: Ross, 1977; RFA nets: Oberquelle, 1987; SA/SD: Yourdon, 1989; Martin and McClure, 1985, and Boose, 1989, give an overview of other techniques).

The unequivocal advantage of formal analytical and descriptive tools is that they force one to perform a thorough and detailed investigation of the domain to be described. The analysis focuses on different aspects, depending on the particular procedure involved. However, the concrete work environment of the end-user is almost completely neglected by most descriptive techniques. Another caveat: "The more detailed this specification is, the more incomprehensible it becomes (not only for Electronic Data Processing (EDP) amateurs)" (Budde et al., 1986, p. 201). The more formal the method of representation, the more time consumed by its preparation. This is partly due to the fact that when users participate, first they need adequate training in operating and interpreting such methods. Unfortunately this circumstance is often used to justify steering clear of user participation when using formal methods of representation.

Prototyping methods I, II and III

As has already been mentioned, prototyping methods make it possible to acquaint end-users with the procedural character of the system being developed. "Prototyping is about adequately imaging a part of or the entire application system in a working model for the future user to be able to grasp the way the planned system works" (Rüesch, 1989, p. 49). It is in this sense that prototyping provides a particularly effective method of communication between the user and the developer: "The possibility of economically applying prototyping in applications development nevertheless requires three tools:

- a non-procedural language
- a data bank management system and
- a data dictionary" (Rüesch, 1989, p. 49).

Since the use of prototypes is always within the test component of some optimisation cycle, they must be *readily modifiable*: "Thus the period elapsed between the suggestion by the user for modification and his assessment of the modified prototype must be as short as possible, for otherwise motivational problems arise" (Kreplin, 1985, p. 75).

Two kinds of prototypes can be distinguished: the vertical and the horizontal. *Horizontal proto-*

types contain only a very small number of application-oriented functions from the end-product, the emphasis being mainly on the presentation of the sequence of templates incorporated in a dialogue structure (Friedman, 1989, p. 291). *Vertical prototypes*, on the other hand, go deeper. In a partial vertical prototype only a few applications functions are implemented and only in a rather rudimentary fashion, whereas a complete vertical prototype implements nearly every application function. This last procedure comes closest to the traditional notion of what a prototype is (see also Pomberger et al., 1987, pp. 20–21). This is the very reason why Pomberger et al. (1987) prefer to speak of a pilot system, even in the case of a complete vertical prototype. As the vertical prototype becomes complete, so does the distinction from a version become more and more slurred.

The disadvantages of prototyping lie in the fact that the prerequisites – the developer must produce incomplete software ('rapid prototyping') and then deal with critiques from the users – are difficult if not impossible to meet. Another aspect is that the traditional industrial notion of a 'prototype' refers to a fully functional product. But in the context of software development, this is more properly called an 'end-product' and not a preliminary variant: "The sad truth is that as an industry, data processing routinely delivers a prototype under the guise of a finished product" (Boar, 1984).

Both of these aspects support the observation that when prototyping is adopted, "the best prototype is often a failed project" (Curtis et al., 1987); "The fundamental idea of prototypes is to iterate the design, not to FREEZE it" (Jörgensen, 1984, p. 287).

Several authors place great value on simpler and quicker user-oriented techniques to ameliorate this danger (Grudin et al., 1987; Nielsen, 1989, 1990; Ehn and Kyng, 1991).

The method of prototyping in the context of an optimisation cycle faces the peril of leading to an inadequate optimum. One reason for this is the fact that being occupied with a concrete prototype can blind one to fundamentally different alternatives (Floyd, 1984, p. 15). One remedy is to precede prototyping with, and to superimpose, suitable optimisation cycles aimed primarily at application contexts (see Fig. 7). Nevertheless, there is no guarantee that the user is also a good designer. One possible consequence of this is that only less than optimal solutions end up being produced iteratively (Jörgensen, 1984, p. 287). Industrial norms and standards, and design criteria for interactive software can be of assistance here. "Prototyping is undoubtedly helpful but there are practical limitations to its applicability. In many cases it is essential to build a complete system to create the necessary interaction before judging a prototype" (Sutcliffe, 1988, p. 182).

Versions methods I and II

As vertical prototyping is extended by increasingly enriching the prototype with programmed functionality, it undergoes a smooth metamorphosis into the version method (Floyd et al., 1990). This procedure gained in significance in the 80's (Frühauf and Jeppesen, 1986; Friedman, 1989, p. 297) because it clearly best matches the Waterfall Model in a software life cycle. The version cycle as a global optimisation cycle is seen here in the proposed participatory development concept in the feedback between the application phase and the prescription phase (Quadrant IV and Quadrant I in Fig. 7). Floyd and Keil (1983) describe a compatible process-oriented procedure. Some authors speak of 'evolutionary' software development in this context (Lehman and Belady, 1985).

The basic advantage of the global optimisation cycle lies unequivocally in the fact that it provides the first opportunity to determine and test all interactions between the usability and practicality of each version within the context of the concrete work environment. If the system being developed is sufficiently complex, then certain flaws in its design can only be detected in the real-life situation of the implementation phase (Quadrant IV in Fig. 7). To keep to a minimum the modifications required, the system must be developed from its very inception according to modern programming concepts (documentation, modular structure, object-oriented programming, etc.).

Repeatedly running through the global optimisation cycle makes it impossible to avoid a certain

measure of upwards compatibility. It is therefore necessary to include preceding optimisation cycles in Quadrant I and Quadrant II to minimise the risk of fundamental design flaws (Peschke, 1986, p. 161). Peschke (1986) also correctly points out that the very substantial cycle length in the version method can lead to a breakdown in contact between developer and user.

6.2. User-developer distance for participatory methods

Users cannot be adequately involved in participatory software development without a whole range of aspects being given serious consideration (Vartiainen, 1989). The aspects we discuss here can be brought together in the six-dimensional umbrella concept of 'user–developer distance'. The user–developer distance varies according to the type of project, so that different methods become necessary to realise participation.

The application-related distance: This refers to the extent to which the concrete application context is known. Thus, for example, users might be 'unloaded' from their sections for the participation because their unsatisfactory qualification renders them dispensable, or because they only acquire the technical qualifications required during their 'participation period'. A serious problem arises from the fact that the application context is still largely unknown, or is extremely heterogeneous. This is especially true in the case of the development of basically new, universal or standard software.

The qualification distance: Whenever analytical and descriptive techniques are used whose application requires special training, all those who come into contact with these techniques in their regular routine must be suitably qualified. Fruitful communication between user and developer also requires that the developer be sufficiently acquainted with the occupational activities of the user to be able to ask the right questions. If users are involved only to an insufficient extent, then the developer will be faced with the problem of acquiring the technical expertise himself. But within the constraints of a development project this is anything but an optimal path.

The organisational distance: If the development department and the contracting department belong to the same organisation, then it is frequently possible to come to a less formal contractual agreement than when they are parts of completely different organisations. The greater the organisational distance the more extensive are the contractual agreements needed to secure the project.

The motivational distance: If users are insufficiently informed about the current stage of development, if they have no opportunity for direct influence, or if they lack the requisite qualification, then serious motivational disturbances can arise which hinder further co-operation. In particular, motivational distance can grow if the users gain the impression that rather than having their work burden alleviated, they could be falling victim to rationalisation measures.

The geographical distance: This always becomes an important aspect whenever the workplace of the users is at a substantial distance from the location of the software development department. It can prove to be extremely useful to carry out as much of the development as possible in the same building as the users to facilitate the quickest possible communication between users and developer.

The temporal distance: This embraces all those problems originating in the fact that the user's time is precious. Even when the user is in close proximity, extensive feedback can be hindered by the fact that the user may not be able to find the time to participate in intensive discussions about specifications. This problem frequently arises in projects aimed at providing EDP support for technically highly specialised users.

7. Conclusion

One of the principal problems of traditional software development lies in the fact that those who have been primarily involved in software development to date have not been willing to recognise that software development is, in most cases, mainly a question of task, job and/or organisational planning. Were software develop-

ment to be approached from such a perspective, it would be planned from the beginning to engage experts in occupational and organisational planning in the process of software design. This, however, would require interdisciplinary co-operation between occupational and organisational experts on the one hand and software development experts on the other. The extensive qualification required in each of these fields makes it virtually impossible to dispense with such interdisciplinary co-operation.

We have presented here an iterative-cyclic process model that integrates solution proposals developed to date for overcoming the specification, communication and optimisation barriers on the basis of the notion of an optimisation cycle. This consists of an action and a test component, coupled to each other by feedback. The feedback loops recommended at various places in the literature have been incorporated into a global cycle as local optimisation ones. The global optimisation cycle can be subdivided into four regions: the region where requirements are determined (Quadrant I), the region of specification (Quadrant II), the region of implementation (Quadrant III) and the region of application and maintenance (Quadrant IV).

Different aspects of the work system to be designed can be progressively optimised as one moves from quadrant to quadrant. The various perspectives of the ideal sought take on progressively more concrete form. An appropriate investment in optimisation in Quadrants I and II not only helps to reduce the total cost (development costs and application costs), but also leads to optimally adapted hardware and software solutions. This is because all subsequent users are involved at least through representatives, and can therefore incorporate their relevant knowledge into the design of the work system.

As more effort is expended on optimisation in the first quadrants, less is needed in Quadrant IV. The amount of effort required for optimisation in the second and third quadrant depends in essence upon the complexity of the work system to be designed. The investment in Quadrant II can be minimised for example with the help of modern prototyping tools and specification methods which the user finds easy to understand. Employing powerful development environments and suitably qualified software developer minimises the investment in Quadrant III. Simultaneous engineering with cross-functional teams is one important factor to increase productivity in terms of cost- and time-reduction. Another important factor is the adequate synchronisation of concurrent processes to avoid waste and conflicts.

Simultaneous software engineering discussed in this paper is a systematic approach to the integrated concurrent design of a software product and its related processes. The presented iterative-cyclic process model is intended to cause the developers, from the outset, to consider all elements of the software product life cycle. But first and foremost, we must start learning to plan jointly technology, organisation and the application of human qualification. Technology should be viewed as one way of providing the opportunity to organise our living and working environments in a manner that is better suited to human needs.

Acknowledgements

The preparation of this paper was supported by the German Secretary of State for Research and Technology (BMFT, AuT programme) grant number 01 HK 706-0 as part of the BOSS 'User oriented Software Development and Interface Design' research project.

References

Ackermann, D., 1987. Handlungsspielraum, Mentale Repräsentation und Handlungsregulation am Beispiel der Mensch–Computer-Interaktion. Doctoral Thesis, Zürich: Work and Organisational Psychology Unit, Swiss Federal Institute of Technology.

AMI, 1992. A quantitative approach to software management. Centre for Systems and Software Engineering, South Bank University, 103 Borough Road, London SE1 0AA, UK.

Boar, B.H., 1984. Application Prototyping: A Requirements Definition Strategy for the 80s. John Wiley, New York.

Boehm, B.W., Gray, T. and Seewaldt, T., 1981. Prototyping versus specifying: A multiproject experiment. IEEE Transactions on SE, 10(3): 224–236.

Boehm, B.W., 1981. Software Engineering Economics. Prentice Hall, Englewood Cliffs, NJ.

Boehm, B.W., 1988. A spiral model of software development and enhancement. Computer, May: 61–72.

Boose, J.H., 1989. A survey of knowledge acquisition techniques and tools. Knowledge Acquisition, 1: 3–37.

Brothers, L., Sembugamoorthy, V. and Muller, M., 1991. ICICLE: Groupware for code inspection. Proceedings of the Conference on Computer-Supported Cooperative Work. Association for Computing Machinery, New York, pp. 169–181.

Budde, R., Kuhlenkamp, K., Sylla, K.H. and Züllighoven, H., 1986. Prototypenbau bei der Systemkonstruktion – Konzepte der Systementwicklung. Angewandte Informatik, 5: 198–204.

Buhr, M. and Klaus, G., 1975. Philosophisches Wörterbuch. Das Europäische Buch, Berlin.

C.E.C. Commission of the European Communities, 1989. Science, technology and societies: European priorities. Results and recommendations of the FAST II programme, Summary Report. Directorate-General Science, Research and Development, Brussels.

Crellin, J., Horn, T. and Preece, J., 1990. Evaluating evaluation: A case study of the use of novel and conventional evaluation techniques in a small company. In: D. Diaper, G. Gilmore, G. Cockton and B. Shackel (Eds.), Human–Computer Interaction – INTERACT '90. Elsevier Science, Amsterdam, pp. 329–335.

Curtis, B., Krasner, H., Shen, V. and Iscoe, N., 1987. Initial results from a field study of large software development projects. Unpublished manuscript (Reference in: J. Grudin, S.F. Ehrlich and R. Shriner).

Deming, W.E., 1989. Out of the Crisis. Massachusetts Institute of Technology, Cambridge, MA.

Dumas, J.S. and Redish, J.C., 1993. A Practical Guide to Usability Testing. Ablex, Norwood, NJ.

Ehn, P. and Kyng, M., 1991. Cardboard computers: Mocking-it-up or hands-on the future. In: J. Greenbaum and M. Kyng (Eds.), Design at Work: Cooperative Design of Computer Systems. Lawrence Erlbaum, Hillsdale, NJ, pp. 169–195.

Floyd, C., 1984. A systematic look at prototyping. In: R. Budde, K. Kuhlenkamp, K.H. Sylla and H. Züllighoven (Eds.), Approaches to Prototyping. Springer, Berlin, pp. 1–15.

Floyd, C. and Keil, R., 1983. Adapting software development for systems design with user. In: U. Briefs, C. Ciborra and L. Schneider (Eds.), System Design for, with, and by the Users. North-Holland, Amsterdam, pp. 163–172.

Floyd, C., Mehl, W.M., Reisin, F.M. and Wolf, G., 1990. Projekt PETS – partizipative Entwicklung transparenzschaffender Software für EDV-gestätzte Arbeitsplätze. Technical Report, Department of Computer Science, Technical University of Berlin.

Friedman, A.L., 1989. Computer Systems Development – History, Organization and Implementation. John Wiley, Chichester.

Frühauf, K. and Jeppesen, K.J., 1986. Software development: The staircase approach. In: IFAC Experience with the Management of Software Projects, Heidelberg, pp. 115–123.

Grudin, J., Ehrlich, S.F. and Shriner, R., 1987. Positioning human factors in the user interface development chain. Proceedings of CHI + GI '87. Association for Computing Machinery, New York, pp. 125–131.

Jones, T.C., 1987. Effektive Programmentwicklung. McGraw-Hill, Hamburg.

Jörgensen, A.H., 1984. On the psychology of prototyping. In: R. Budde, K. Kuhlenkamp, K.H. Sylla and H. Züllighoven (Eds.), Approaches to Prototyping. Springer, Berlin, pp. 278–289.

Karat, C.M., 1990. Cost–benefit analysis of iterative usability testing. In: D. Diaper, G. Gilmore, G. Cockton and B. Shackel (Eds.), Human–Computer Interaction – INTERACT '90. Elsevier Science, Amsterdam, pp. 351–356.

Kirsch, C., 1992. Evaluation der Erhebungsmethoden zur Benutzerbeteiligung bei der Datenmodellierung aus psychologischer Sicht. Unpublished Diploma Thesis at the University of Konstanz, Germany.

Kreplin, K.D., 1985. Prototyping – Softwareentwicklung für den und mit dem Anwender. Handbuch der Modernen Datenverarbeitung, 22(126): 73–126.

Lehman, M.M. and Belady, L.A., 1985. Program Evolution – Processes of Software Change. Academic Press, London.

Lewis, J.R., Henry, S.C. and Mack, R.L., 1990. Integrated office software benchmarks: A case study. In: D. Diaper, G. Gilmore, G. Cockton and B. Shackel (Eds.), Human–Computer Interaction – INTERACT '90. Elsevier Science, Amsterdam, pp. 337–343.

Macaulay, L., Fowler, C., Kirby, M. and Hutt, A., 1990. USTM: A new approach to requirements specification. Interacting with Computers, 2(1): 92–118.

Mai, M., 1990. Sprache und Technik. Zeitschrift des Vereins Deutscher Ingenieure für Maschinenbau und Metallbearbeitung, 132(7): 10–13.

Mambrey, P., Oppermann, R. and Tepper, A., 1986. Computer und Partizipation. Westdeutscher Verlag, Opladen.

Martin, J., 1989. Information Engineering, Book I: Introduction. Prentice Hall, Englewood Cliffs, NJ.

Martin, J. and McClure, C., 1985. Diagramming Techniques for Analysts and Programmers. Prentice Hall, Englewood Cliffs, NJ.

Nielsen, J., 1989. Usability engineering at a discount. In: G. Salvendy and M.J. Smith (Eds.), Designing and Using Human–Computer Interfaces and Knowledge Based Systems. Elsevier Science, Amsterdam, pp. 394–401.

Nielsen, J., 1990. Big paybacks from 'discount' usability engineering. IEEE Software, 7(3): 107–108.

Oberquelle, H., 1987. Sprachkonzepte für benutzergerechte Systeme (Informatik-Fachberichte, No. 144). Springer, Berlin.

Oppermann, R., 1983. Forschungsgegenstand und Perspektiven partizipativer Systementwicklung. Oldenbourg, München.

Peschke, H., 1986. Betroffenenorientierte Systementwicklung (Europäische Hochschulschriften, Reihe XLI, Informatik, Vol. 1.1). Peter Lang, Frankfurt.

Pomberger, G., Bischofberger, W., Keller, R. and Schmidt, D., 1987. Prototypingorientierte Softwareentwicklung, Teil I. Technical Report No 87.05, Department of Computer Science, Winterthurerstrasse 190, CH-8057 Zurich.

Rauterberg, M., 1991. Benutzungsorientierte Benchmark-Tests: eine Methode zur Benutzerbeteiligung bei Standardsoftwareentwicklungen. In: D. Ackermann and E. Ulich (Eds.), Software-Ergonomie '91 (Reports of the German Chapter of the ACM, Vol. 33). Teubner, Stuttgart, pp. 96–107.

Rauterberg, M., 1992. Optimisation cycle: A concept for optimal software development. In: R. Trappl (Ed.), Cybernetics and System Research, Vol. 1. World Scientific, Singapore, pp. 279–286.

Rauterberg, M., Spinas, P., Strohm, O., Ulich, E. and Waeber, D., 1994. Benutzerorientierte Softwareentwicklung – Konzepte, Methoden und Vorgehen zur Benutzerbeteiligung. Teubner, Stuttgart.

Rettig, M., 1991. Testing made palatable. Communications of the ACM, 34(5): 25–29.

Ross, D.T., 1977. Structured analysis (SA): A language for communicating ideas. IEEE Transactions on Software Engineering, SE 3(1): 16–34.

Rüesch, P., 1989. Entwicklungsumgebung. Output, 11: 45–51.

Schiemenz, B., 1979. Kybernetik. Handwörterbuch der Produktionswissenschaft. Poeschel, Stuttgart.

Shlaer, S. and Mellor, S.J., 1988. Object-Oriented Systems Analysis: Modeling the World in Data. Prentice Hall, Englewood Cliffs, NJ.

Sommerville, I., 1989. Software Engineering. Addison Wesley, Wokingham.

Spencer, R.H., 1985. Computer Usability Testing and Evaluation. Prentice Hall, Englewood Cliffs, NJ.

Spinas, P. and Ackermann, D., 1989. Methods and tools for software development: Results of case studies. In: F. Klix, N. Streitz, Y. Waern and H. Wandke (Eds.), Man–Computer Interaction Research, MACINTER-II. North-Holland, Amsterdam, pp. 511–521.

Strohm, O., 1991. Arbeitsorganisation, Methodik und Benutzerorientierung bei der Software-entwicklung. In: M. Frese, C. Kasten, C. Skarpelis and B. Zang-Scheucher (Eds.), Software für die Arbeit von morgen. Springer, Berlin, pp. 431–441.

Strohm, O. and Ulich, E., 1991. Arbeitsteilung und Benutzerorientierung bei der Software-Entwicklung. In: F. Elzer (Ed.), Multidimensionales Software-Projektmanagement. Hallbergmoos, pp. 261–289.

Stovsky, M.P. and Weide, B.W., 1991. Access control strategies for coordinating teams of software engineers. International Journal of Software Engineering and Knowledge Engineering, 1(1): 57–73.

Sutcliffe, A., 1988. Human–Computer Interface Design. Macmillan, London.

Tucker, D.E. and Leonard, R., 1994. Overcoming the cultural barriers to implementing concurrent engineering. In: P. Kidds and W. Karwowski (Eds.), Advances in Agile Manufacturing. IOS Press, Amsterdam, pp. 67–70.

Udris, I. and Ulich, E., 1987. Organisations-und Technikgestaltung: Prozeβ und partizipationsorientierte Arbeitsanalysen. In: K. Sonntag (Ed.), Arbeitsanalyse und Technikentwicklung – Beiträge für Einsatzmöglichkeiten arbeitsanalytischer Verfahren bei technischorganisatorischen Änderungen. Wirtschaftsverlag Bachem, Köln, 49–68.

Ulich, E., 1994. Arbeitspsychologie (3rd edition). Poeschel, Stuttgart.

Vartiainen, M., 1989. Participation: Planning for themselves or planning for others? Psychological task analysis, design and training in computerized technologies. Technical Report No. 113, Helsinki University of Technology, Otakaari 4 A, SF-02150 Espoo, Finland, pp. 121–135.

Yeh, R.T., 1991. System development as a wicked problem. International Journal of Software Engineering and Knowledge Engineering, 1(2): 117–130.

Yourdon, E., 1989. Modern Structured Analysis. Prentice-Hall, Englewood Cliffs, NJ.

Zölch, M. and Dunckel, H., 1991. Erste Ergebnisse des Einsatzes der 'Kontrastiven Aufgabenanalyse'. In: D. Ackermann and E. Ulich (Eds.), Software-Ergonommie '91. (Reports of the German Chapter of the ACM, Vol. 33). Teubner, Stuttgart, pp. 363–372.