HUMAN FACTORS AND MODERN SOFTWARE DEVELOPMENT

Matthias Rauterberg

Swiss Federal Institute of Technology (ETH)
Nelkenstrasse 11, CH-8092 Zuerich, rauterberg@rzvax.ethz.ch

Abstract

The current state of traditional software development is surveyed and essential problems are investigated on the basis of system theoretical considerations. The concept of optimisation cycle (ISO 9000) is proposed as a solution. The relation of several different kinds of local optimisation cycles to known techniques for user-participation is discussed and integrated into a general concept of participatory software development. Essential problems known to obstruct optimal software development and possible ways of solving them are discussed.

1 Introduction

Analysis of current software development procedures brings to light a series of weaknesses and problems, the sources of which lie in the theoretical concepts applied, the traditional procedures followed (especially project management) as well as in the use of inadequate cost analysis models. The literature contains a large store of proposed solutions based on current practice in software development. These results point to the significance of participation by all groups affected. Analysis of these cases shows that there are three essential barriers to optimisation: the specification barrier, the communication barrier and the optimisation barrier.

Speaking quite generally, one of the most important problems lies in coming to a shared understanding by all the affected groups of the component of the work system to be automated (Naur 1985) — 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. This involves, in particular, determining all the characteristics of the work system to be newly planned. Every work system comprises a social and a technical subsystem. An optimal total system must integrate both simultaneously.

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 a system to be optimised when coupled with the technical subsystem.

2 Barriers in the Framework of Traditional Software Development

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 — or perhaps is in part not in a position to do so — before the beginning of the software development process. It is therefore

necessary to find and implement other ways and means, using from 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. As a result, the following different perspectives must be considered in the analysis and specification phases.

The *Applier's Perspective*: Every person in a position to contribute to formulating the requirements for the total work system is considered to be an applier. This perspective often coincides with that of the clients, and takes into consideration general requirements concerning the effectiveness, organisational structures, project costs, global application and implementation goals for the technical subsystem, as well as the desired effect on the total work system. The applier's perspective thus embraces the requirements for the organisational interface.

The *User's Perspective*: Users are those persons who need the results obtained from using the software system for performing their tasks. The dominant factor influencing their perspective is human-to-human communication with the end-users (e.g., heads, secretaries, etc.) and their contribution usually includes requirements for the tool interface.

The *End-User Perspective*: End-users are all those who directly use the software system as a work tool. This group is well placed to formulate pertinent requirements for the organisational, tool, dialogue and input/output interfaces.

2.2 The Communications Barrier

The communications barrier between applier, user and end-user on the one hand and the software developer on the other is essentially due to the fact that "technical intelligence" is only inadequately imbedded in the social, historical and political contexts of technological development. 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.

The application-oriented jargon of the user flounders on the technical jargon of the developer (Carroll 1988:158). This "gap" can only be bridged to a limited extent by purely linguistic means, because that their semantics is conceptually bound makes the ideas applied insufficiently sharp. To overcome this fuzziness requires creating jointly experienced, perceptually shared contexts. Beyond spoken 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.

As a rule, software development is a procedure for optimally designing a product with interactive properties for supporting the performance of work tasks. Because computer science has accumulated quite a treasure throve of very broadly applicable algorithms, 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 application environment aimed at requires the implementation of optimisation procedures of a different nature.

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 the people involved develop and increasingly clear picture of what it is that they are really striving for. This is basically why the requirements of the applier seem to "change" — they do not really change but simply become concrete within the anticipated boundary constraints. This process of concretisation should be allowed to unfold as completely, as pertinently and — from a global perspective — as inexpensively as possible. Completeness can be reached by making sure that each affected group of persons is involved at least through representatives. Iterative, interac-

tive progress makes the ideal concept increasingly concrete. There are methods available for supporting the process of communication that ensure efficient progress (Nielsen 1993).

3 Overcoming the Barriers

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 (Gomaa 1983; Scharer 1983; Baroudi, Olson & Ives 1986; Mantei & Teorey 1988; Rauterberg & Strohm 1992).

3.1 The Optimisation Cycle

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 for a feed forward or open loop structure: (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 affect on the system is detrimental to the intended influence (interference or perturbation, 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 is necessary. These conditions are hardly ever met in practice. That is why it is constantly necessary to supplement or to replace control by regulation.

The application of the highly effective "regulation" principle only requires a knowledge of those controller outputs which steer the output in the desired direction. We designate the "Test-Action- Cycle" based on "regulation" as *optimisation cycle*. 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 (see Table 1). The longer this period, the more costly the optimisation cycle. It is the aim of user-oriented software development to incorporate an as efficient optimisation cycle as possible into software development procedures.

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 (Rettig 1991). 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. It is frequently the case that control systems operate more economically than (even possible) regulation systems — but only if the conditions mentioned above prevail! This is also the important reason, why the attempt is made to come closely to a particular control system, namely the "waterfall" model. If, however, the barrier's discussed above, are taken seriously, then we must determine those places in software development procedures at which optimisation cycles are *indispensable*.

3.2 The Analysis Phase

The analysis phase is frequently the one most neglected. This is essentially because methods and techniques need to be used primarily the way occupational and organisational sciences have developed and applied them (Macaulay et al. 1990). Inordinately high costs incur from the troubleshooting required because the analysis was less than optimal (Rauterberg & Strohm 1992). The time has come to engage occupational and organisational scientists at the analysis stage who have been specially trained for optimal software development!

While traditional software development partly includes a global analysis of the tasks in the work system, analysis of work activities and their effect is largely excluded from consideration. The results of the detailed analysis of the objective conditions of a work system need to be supplemented with the subjective conditions experienced by the employees if the organisational measures to be drawn up jointly by all those affected is to have a chance of finding consensus.

3.3 The Specification Phase

Once the analysis of the work system to be optimised has been successfully completed, the next stage is to mould the results obtained into implementable form (Martin 1988). Methods of specification with high communicative value are recommended here. The first thing is to determine "if" and "where" it makes sense to employ modern technology (Malone 1985). 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.

The intended division of functions between man and machine is decided during the specification of the tool interface. The tasks that remain in human hands must have the following characteristics (Volpert 1987): 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.

Once those concerned are sufficiently clear about which functions are amenable to automation, the next step that should be taken is to test the screen layout on the end-users with hand-drawn sketches (the extremely inexpensive "pen and paper" method, Wulff, Evenson & Rheinfrank 1990). If the range of templates is very large, then a graphics data bank can be used to manage the templates produced on a graphics editor (Martin 1988:79). The effect of the structuring measures taken can be assessed with the help of discussion with the end-users, or by means of checklists (Nielsen 1993).

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 indiscriminately. 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.

3.4 The Implementation Phase

The implementation phase is made up of the following three steps (Boehm 1981): 1. Design of the programme architecture. 2. Design of the individual programme modules (object classes, etc.) 3. Coding and debugging. It is important to check before start coding the extent to which already available software can be re-used (Prieto-Diaz 1991). The use of software development environments can result in increases in productivity of up to 40% (Chikofsky & Rubenstein 1988).

3.5 The Trial and Assessment Phase

Once a working version is available, it can be put to test in usability studies ("user-oriented benchmark tests": Spencer 1985; Karat 1988; 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. Whiteside, Bennett and Holtzblatt (1988:805) point to the necessity of empirical evaluation techniques in concrete working situations. By contrast to laboratory studies, such field studies take into account the aspect of "ecological validity." An operational version of the system must be available for such methods to be used.

4 A Participatory Concept for Software Development

The literature contains a whole series of suggestions for incorporating optimisation cycles into software development procedures (Iivari 1984; Tavolato & Vincena 1984; Frühauf & Jeppesen 1986; Eason & Harker 1987; Grudin, Ehrlich & Shriner 1987; Boehm 1988; Macaulay et al. 1990). The various authors have different emphases in their concepts, depending on their own background and experience. The type of software to be developed has proved to be one of the essential factors governing software development. The following four types can be distinguished.

Type A: Specific application for an internal division; both, the division placing the order and the one developing the software belong to the same company.

Type B: Specific application for external users; the division placing the order and the one developing the software belong to different companies.

Type C: Standard solutions for external users; this often arises from projects of Type A or Type B, when individual software solutions (Type A, Type B) are specially adapted for further users.

Type D: Standard software for a largely anonymous circle of users.

To reach the goals of a work-oriented design concept (Ulich et al. 1991) the first project phases (requirement's analysis and definition) should be replete with a range of optimisation cycles. The system design is settled on after carrying out a complete as possible clarification of the requirements of the client (work and task analysis, division of functions between humans and computer, etc.). The question remains as to which design specifications need clarification by means of additional optimisation cycles.

Table 1 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 & verbal interpretation	Specific design-decisions	Minutes - hours
Simulation-I	Sketches, scenarios, "Wizard of Oz", etc.	Visual & verbal interpretation	Specification of the input/output interface	Minutes - days
Simulation-II	Draughting of structural blueprints, etc. with (semi)-formal methods	Visual & verbal interpretation with suitable qualification	(Semi)-formal descriptive documents	Hours - weeks
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	Run through entire development cycle	Inductive benchmark tests	First largely complete version	Months- years
Versions-II	Run through entire development cycle	Deductive benchmark tests	Several largely complete versions	Months- years

The global optimisation cycle begins at Start A of Figure 1 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.

4 A Participatory Concept for Software Development

The literature contains a whole series of suggestions for incorporating optimisation cycles into software development procedures (Iivari 1984; Tavolato & Vincena 1984; Frühauf & Jeppesen 1986; Eason & Harker 1987; Grudin, Ehrlich & Shriner 1987; Boehm 1988; Macaulay et al. 1990). The various authors have different emphases in their concepts, depending on their own background and experience. The type of software to be developed has proved to be one of the essential factors governing software development. The following four types can be distinguished.

Type A: Specific application for an internal division; both, the division placing the order and the one developing the software belong to the same company.

Type B: Specific application for external users; the division placing the order and the one developing the software belong to different companies.

Type C: Standard solutions for external users; this often arises from projects of Type A or Type B, when individual software solutions (Type A, Type B) are specially adapted for further users.

Type D: Standard software for a largely anonymous circle of users.

To reach the goals of a work-oriented design concept (Ulich et al. 1991) the first project phases (requirement's analysis and definition) should be replete with a range of optimisation cycles. The system design is settled on after carrying out a complete as possible clarification of the requirements of the client (work and task analysis, division of functions between humans and computer, etc.). The question remains as to which design specifications need clarification by means of additional optimisation cycles.

Table 1 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 & verbal interpretation	Specific design-decisions	Minutes - hours
Simulation-I	Sketches, scenarios, "Wizard of Oz", etc.	Visual & verbal interpretation	Specification of the input/output interface	Minutes - days
Simulation-II	Draughting of structural blueprints, etc. with (semi)-formal methods	Visual & verbal interpretation with suitable qualification	(Semi)-formal descriptive documents	Hours - weeks
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	Run through entire development cycle	Inductive benchmark tests	First largely complete version	Months- years
Versions-II	Run through entire development cycle	Deductive benchmark tests	Several largely complete versions	Months- years

The global optimisation cycle begins at Start A of Figure 1 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.

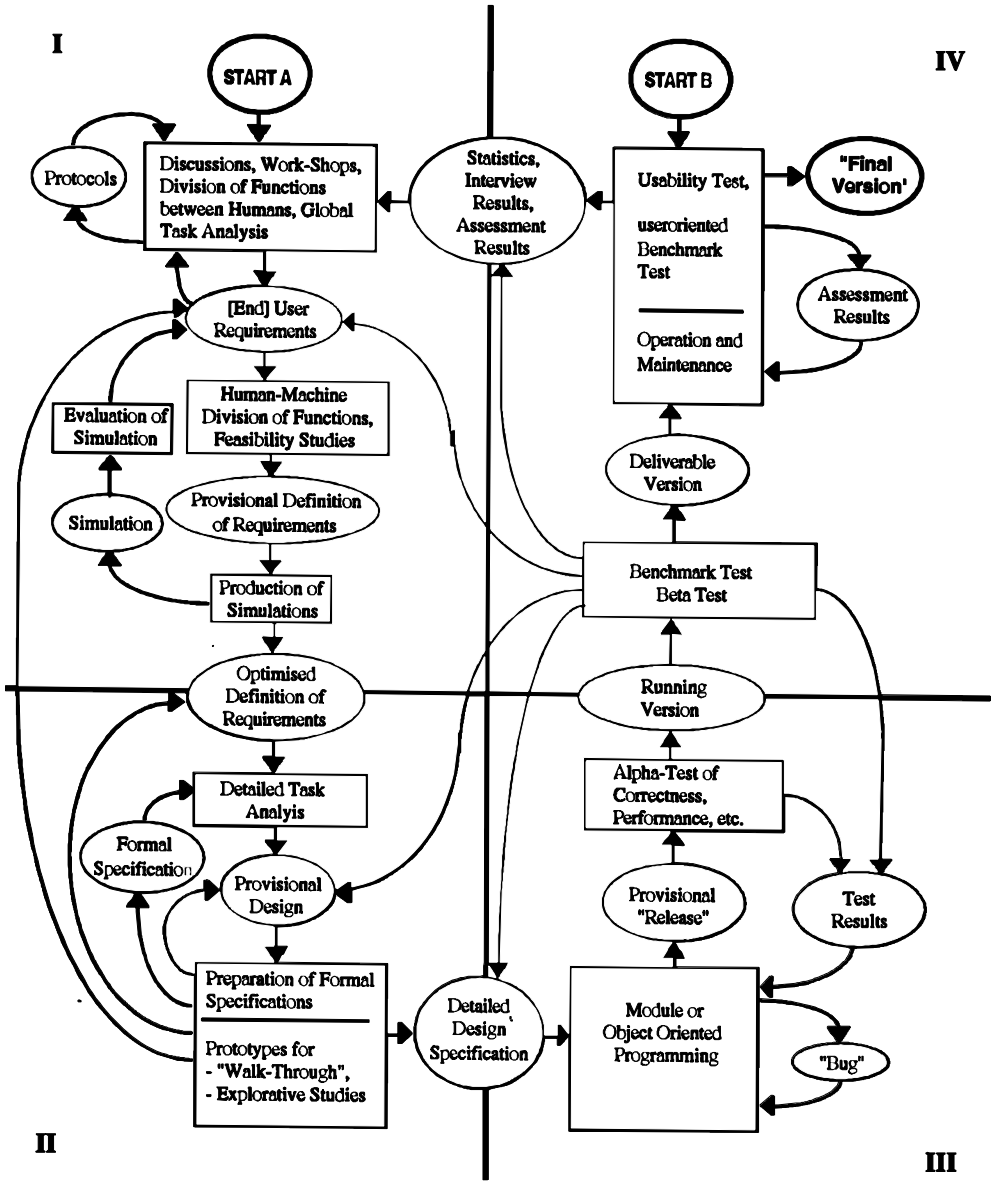


Figure 1 Flow chart for an iterative-cyclic software process model showing the local optimisation cycles within and between individual quadrants (I - IV). The systematic use of application and maintenance cycles with feedback to the requirement phase thus subsumes the version concept as the global optimisation cycle (cf. Grudin, Ehrlich & Shriner 1987).

Simple and fast techniques for involving users include discussion groups with various communication aids (meta-plan, layout sketches, "screen-dumps", scenarios, etc.), questionnaires 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, Ehrlich & Shriner 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 (Nielsen 1993). Spencer (1985) presents a summary of techniques for the analysis and evaluation of interactive computer systems (see also Crellin, Horn & Preece 1990; Vainio-Larsson & Orring 1990). Comparative studies, e.g., user-oriented benchmark tests (Lewis, Henry & Mack 1990; Rauterberg 1991), can be undertaken after the second time through, when working with a version concept, for then there are at least two versions available.

The global optimisation cycle together with its incorporated local cycles, can be subdivided into four regions (Quadrants-I – IV of Figure 1). Quadrant-I includes the analysis and approximate specification. Communicative, informal methods are mainly applied here. Detailed specifications are optimised in Quadrant-II using prototypes. The specified hardware and software are implemented in Quadrant-III and the test data assessed. Quadrant-IV comprises assessment and optimisation of the system in a real-life operating environment.

The effort spent on optimisation in each quadrant varies according to the type of the project and of the task. However all software development project analyses completed to date indicate that increasing the effort expended on optimisation in Quadrant I reduces maintenance in Quadrant IV and saves costs (Macaulay et al. 1990).

5. Participatory Techniques, Methods and Concepts in an iterative-cyclic Software Process Model

In the introduction we used systems theoretical considerations to explain how each optimisation cycle consists of a test and an action component that are suitably coupled. Each component can be of a widely varying nature. Table 1 provides a survey 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.

Discussion Methods I & II: Discussion is the method most frequently used, because it is fast, familiar and to a certain extent informative (see the "communications barrier"). But, because it rests essentially on purely verbal communication, a series of misunderstandings can arise which often never become known or only do so when it is already too late. Discussion must therefore be supplemented with methods using visual communications techniques.

Simulation Methods I & II: Simulation methods comprise all techniques that illustrate the work system to be optimised in as realistic, visually perceivable a way as possible. This ranges from simple quickly completed sketches, through template layouts to formal description techniques (SADT: Ross 1977; SA/SD: Yourdon 1989).

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. The more formal the method of representation, the more time consumed by its preparation.

Prototyping Methods I, II & III: 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. It is in this sense that prototyping provides a particularly effective mean of communication between the user and the developer. Since the use

of prototypes is always within the test component of some optimisation cycle, they must be *readily modifiable.*

Two kinds of prototypes can be distinguished: the vertical and the horizontal. *Horizontal prototypes* 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. *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 in the traditional industrial background is.

The disadvantages of prototyping lie in the fact that the prerequisites — the developer must produce incomplete software ("rapid prototyping") and then deal with critique from the user — are difficult if not impossible to meet. "The sad truth is that as an industry, data processing routinely delivers a prototype under the guise of a finished product" (Boar 1984). "The fundamental idea of prototypes is to iterate the design, not to FREEZE it" (Jörgensen 1984:287). Several authors place great value on simpler and quicker participatory techniques to banish this danger (Grudin, Ehrlich & Shriner 1987; Nielsen 1993).

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 that being occupied with a concrete prototype can blind one to fundamentally different alternatives (Floyd 1984:15). One remedy is to precede prototyping with and to superimpose suitable optimisation cycles aimed primarily at application contexts (see Figure 1). 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 1984:287). Industrial norms and standards (ISO 9000, Smith 1986), and design criteria for interactive software (Ulich et al. 1991) can be of assistance here.

Versions' Methods I & II: As vertical prototyping is extended by increasingly enriching the prototype with programmed functionality, it undergoes a smooth metamorphosis into the version method. This procedure gained in significance in the 80's (Frühauf & Jeppesen 1986) because it clearly best matches the "Waterfall Model" in a software life cycle.

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. 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, CASE tools, etc.)

6 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 occupational and/or organisational planning. Were software development 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 software development concept 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 a test and an action 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. This 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 (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 lead to optimally adapted hardware and software solutions. This is because all subsequent users are involved at least through representatives, and can therefore inject their relevant knowledge into the design of the work system.

As more the effort expended on optimisation in the first quadrants, so less is needed in Quadrant-IV (Boehm 1981). The amount of effort required for optimisation in the second and third quadrants 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 that the user finds simply to understand. Employing powerful development environments and suitably qualified software developers minimises the investment in Quadrant-III. But first and foremost, we must start learning to plan jointly technology, organisation and the application of human qualification.

Acknowledgements

The preparation of this paper was supported by the German Minister of 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

BAROUDI J, OLSON M & IVES B, 1986: An Empirical Study of the Impact of User Involvment on System Usage and Information Satisfaction. Communications of the ACM 29(3): 232-238

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

BOEHM B W, 1981: Software Engineering Economics. Englewood Cliffs: Prentice Hall.

BOEHM B W, 1988: A spiral model of software development and enhancement. Computer (May 1988):61-72

CARROLL J, 1988: Integrating Human Factors and Software Development. In: Proceedings of CHI '88 (Washington, 15th - 19th May 1988). New York: ACM. 157-159

CHIKOFSKY E J & RUBENSTEIN B L, 1988: CASE: Reliability engineering for information systems. IEEE Software 5(2): 11-17

CRELLIN J, HORN T, PREECE J, 1990: Evaluating Evaluation: A Case Study of the Use of Novel and Conventional Evaluation Techniques in a Small Company. In: DIAPER D et al. (eds.) Human-Computer Interaction - INTERACT '90. Amsterdam: Elsevier Science. 329-335

EASON K D & HARKER S D P, 1987: A User Centered Approach to the Design of a Knwoledge Based System. In: BULLINGER H-J. & SHACKEL B (eds.) Human-Computer Interaction - INTERACT '87. Amsterdam: Elsevier Science. 341-346

FLOYD C, 1984: A Systematic Look at Prototyping. In: BUDDE R, KUHLENKAMP K, MATHIASSEN L, ZÜLLIGHOVEN H (eds.) Approaches to Prototyping. Berlin: Springer. 1-15

FRÜHAUF K & JEPPESEN K J, 1986: Software Development: the Staircase Approach. IFAC Experience with the Management of Software Projects. 115-123

GOMAA H, 1983: The impact of rapid prototyping on specifying user requirements. ACM SIGSOFT Software Engineering Notes 8(2):17-28

GRUDIN J, EHRLICH S F, SHRINER R, 1987: Positioning Human Factors in the User Interface Development Chain. In: Proceedings of CHI + GI (Toronto, 5th - 9th April 1987). New York: ACM. 125-131

IIVARI J, 1984: Prototyping in the Context of Information System Design. In: BUDDE R, KUHLENKAMP K, MATHIASSEN L, ZÜLLIGHOVEN H (eds.) Approaches to Prototyping. Berlin: Springer.

JÖRGENSEN A H, 1984: On the Psychology of Prototyping. In: BUDDE R, KUHLENKAMP K, MATHIASSEN L, ZÜLLIGHOVEN H (eds.) Approaches to Prototyping. Berlin: Springer. 278-289

KARAT J, 1988: Software Evaluation Methodologies. In: HELANDER M (ed.) Handbook of Human-Computer Interaction. Amsterdam: Elsevier Science. 891- 903

LEWIS J R, HENRY S C & MACK R L, 1990: Integrated Office Software Benchmarks: A Case Study. In: DIAPER D et al. (eds.) Human-Computer Interaction - INTERACT '90. Amsterdam: Elsevier Science. 337-343

MACAULAY L, FOWLER C, KIRBY M & HUTT A, 1990: USTM: a new approach to requirements specification. Interacting with Computers 2(1):92-118

MALONE T W, 1985: Designing organizational interfaces. In: BORMAN L & CURTIS B (eds.) Pro-ceedings of CHI '85 (San Francisco, Special Issue of the SIGCHI Bulletin). 66-71

MANTEI M M & TEOREY T J, 1988: Cost/Benefit Analysis for Incorporating Human Factors in the Software Lifecycle. Communications of the ACM 31(4):428-439

MARTIN C F, 1988: User-Centered Requirements Analysis. Englewood Cliffs: Prentice Hall.

NAUR P, 1985: Programming as Theory Building. Microprocessing and Mircoprogramming 15: 253-261

NIELSEN J, 1993: Usability Engineering. Boston: Academic Press.

PRIETO-DIAZ R, 1991: Implementing Faceted Classification for Software Reuse. Communications of the ACM 34(5):88-97

RAUTERBERG M, 1991: Benutzungsorientierte Benchmark-Tests: eine Methode zur Benutzerbeteiligung bei Standardsoftwareentwicklungen. In: ACKERMANN D &ULICH E (Ed.) Software-Ergonomie '91. (Reports of the German Chapter of the ACM, Vol. 33). Stuttgart: Teubner. 96-107

RAUTERBERG M & STROHM O, 1992: Work organization and software development. In: ELZER P & HAASE V (eds.) Proceedings of 4th IFAC/IFIP Workshop on "Experience with the Management of Software Projects" Annual Review of Automatic Programming 16(2):121-128

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

SCHARER L L, 1983: Prototyping in a production environment. In: CURTIS B (ed.) Proceedings of the ITT Conference on Programming Productivity (June 1983). 440-455

SMITH S L, 1986: Standards versus guidelines for designing user interface software. Behaviour and Information Technology 5(1): 47-61

SPENCER R H, 1985: Computer usability testing and evaluation. Englewood Cliffs: Prentice Hall.

TAVOLATO P & VINCENA K, 1984: A Prototyping Methodology and Its Tool. In: BUDDE R, KUHLENKAMP K, MATHIASSEN L, ZÜLLIGHOVEN H (eds.) Approaches to Prototyping. Berlin: Springer. 434-446

ULICH E, RAUTERBERG M, MOLL T, GREUTMANN T, STROHM O, 1991: Task Orientation and User-Oriented Dialogue Design. International Journal of Human Computer Interaction 3(2):117-144

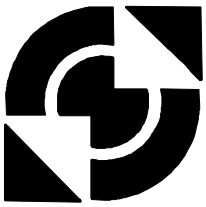
VAINIO-LARSSON A, ORRING R, 1990: Evaluating the Usability of User Interfaces: Research in Practice. In: DIAPER D (ed.) Human-Computer Interaction - INTERACT'90. Amsterdam: Elsevier Science. 323-328

VOLPERT W, 1987a: Kontrastive Analyse des Verhältnisses von Mensch und Rechner als Grundlage des System-Designs. Zeitschrift für Arbeitswissenschaft 41:147-152

WHITESIDE J, BENNETT J, HOLTZBLATT K, 1988: Usability Engineering: Our Experience and Evolution. In: HELANDER M (ed.) Handbook of Human-Computer Interaction. Amsterdam: Elsevier Science. 791-817

WULFF W, EVENSON S & RHEINFRANK J, 1990: Animating Interfaces. In: Proceedings of the Conference on Computer-Supported Cooperative Work (Los Angeles, 7th - 10th October, 1990). 241-254

YOURDON E, 1989: Modern Structured Analysis. Englewood Cliffs: Prentice-Hall.



University of Twente
university for technical
and social sciences

Proceedings of the fifth workshop on the next generation of case tools

Babis Theodoulidis
(Editor)

Memoranda Informatica 94-25
May 1994

ISSN 0924-3755

University of Twente
Department of Computer Science
P.O. Box 217
7500 AE Enschede
The Netherlands

Order-address: University of Twente
TO/INF library
The Memoranda Informatica Secretary
P.O. Box 217
7500 AE ENSCHEDE
The Netherlands
Tel.: 053-894021