AN ITERATIVE-CYCLIC SOFTWARE PROCESS MODEL

Matthias Rauterberg

Work and Organisational Psychology Unit, Swiss Federal Institute of Technology (ETH)
Nelkenstrasse 11, CH-8092 Zuerich, rauterberg@czheth5a.bitnet

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 is presented. The relationship between several different kinds of local optimisation cycles with known techniques for user-participation is discussed and integrated into a general concept of participatory software development. The pros and cons of essential problems known to obstruct optimal software development and possible ways of solving them are considered.

1. Introduction

Analysis of current software development processes 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 [5]. The literature contains an ample store of proposed solutions based on current practice in software development. These 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 goups of the component of the worksystem to be automated [14] — that is, 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 that are to be planned anew. Every work system comprises a social and a technical subsystem. An optimal total system must integrate both simultaneously.

In order 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 subsytem.

2. Barriers in the Framework of Traditional Software Development

The "specification barrier" is an immediate problem even at a cursory glance. How can the software developer ascertain that the client is able to specifiy the requirements for the subsystem to be developed in a complete and accurate way which 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 taken into consideration in the analysis and specification phases.

The *Applier's Perspective*: Every person in a position to make a contribution to defining the requirements for the total work system is considered to be an applier. This perspective often coincides with that of the client, and takes into consideration general requirements concerning the effectivity, 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's Perspective*: End-users are all those who directly use the software system as a work tool. This group

0-8186-2830-8/92 $03.00

is in the position to formulate pertinent requirements for the organisational, tool, dialogue and i/o interfaces.

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. This “gap” can only be bridged to a limited extent by purely linguistic means, because the fact that their respective semantics are conceptually bound make the ideas applied insufficiently precise. Overcoming this fuzziness requires creating 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.

At its best, 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 trove of very broadly applicable algorithms, software development is increasingly focussing attention on those facets of application-oriented software which are unamenable to algorithmic treatment. While the purely technical aspects of a software product are best dealt with by optimisation procedures attuned 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, exact and comprehensive view of the ideal for the work system to be set up. Only during the analysis, evaluation and planning processes are the people involvable to develop an 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 crystallisation should be allowed to unfold as completely, as pertinently and — from a global perspective — as inexpensively as possible. Completeness can be reached by ensuring that each affected group is involved at least through representatives. Iterative, interactive progress makes the ideal concept increasingly concrete. There are methods available for supporting the process of communication which ensure efficient progress [15] [16].

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 [6] [9] [17]. How then, can the both barriers mentioned above be overcome? The answer is: the concept of "optimisation".

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, that functions:
“(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 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” [18].

The application of the highly effective “regulation” principle actually only requires a knowledge of those controller outputs which steer the ouput in the desired direction. We designate the “Test-Action-Cycle” based on the "regulation" principle 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. 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 [9] [15] [16].

The *optimisation criteria* are all relevant technical and social factors [19] [21]. 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 as well as from technical and social problems in realising the project.

Of course current software development also avails itself of the principle of “control” in various places. What we

have in mind here are decisions made and directives issued by the client, 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 feed forward control systems operate more economically than is ever possible with regulated systems — but only if the named conditions prevail! This is one important reason why the attempt is made to come as close as possible to a particular control system in software development, 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 cycles are *indispensible* in a software process model.

3.2 The Analysis Phase

The analysis phase is frequently the one most neglected. This is essentially due to the fact that methods and techniques need to be used primarily the way occupational and organisational sciences have developed and applied them [13]. Inordinately high costs incur from the troubleshooting required because the analysis was less than optimal [6] [20]. The time has come to engage occupational and organisational consultants at the analysis stage who have been especially trained for 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 are to have a chance of finding consensus. Yaverbaum and Culpan [24] determined important criteria for further qualificational and organisational measures by means of “Job Diagnostic Survey (JDS)”.

3.3 The Specification Phase

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.

The first thing is to determine “if” and “where” it makes sense to employ 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” [10].

The intended division of functions between man and machine is decided during the specification of the tool interface. The tasks which remain in human hands must have the following characteristics [22]: 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 which should be taken is to test the screen layout on the end-users with hand-drawn sketches (the extremely inexpensive “pen and paper” method). 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. The use of prototyping tools is frequently inadvisable, because tool-specific presentation offers a too restrictive range of possibilities. The effect of the structuring measures taken can be assessed with the help of discussion with the end-users, or by means of checklists.

The use of prototypes to illustrate the dynamical and interactive aspects of the tools being developed is indispensable for specifying the dialogue interface. However, prototypes should only be used very purposefully and selectively to clarify special aspects of the specification — 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 [9].

3.4 The Implementation Phase

Having invested the necessary effort in optimising both the analysis and specification phases, it is time to enter the implementation phase. This phase is made up of the following three steps [2]: 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 the extent to which already available software can be re-used before commencing coding. The use of software development environments can result in increases in productivity of up to 40% [4].

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”: [17] [19]) 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 et al. [23] point to the necessity of empirical evaluation techniques — as opposed to laboratory tests — in concrete working situtations. By contrast to laboratory studies, such

field studies take into account the aspect of "ecological validity" [9] [23]. An operational version of the system must be available for such methods to be used. This is only possible with the framework of a version concept.

4. Local and Global Optimisation Cycles

The literature contains a whole series of suggestions for incorporating optimisation cycles into software development procedures [3] [7] [13] [17]. The various authors have different emphases in their concepts, depending on their own background and experience.

We now turn to a discussion of those aspects which need to be borne in mind when commencing as well as throughout the course of an individual optimisation cycle. 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 [20]:
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 an anonymous user group.

The global optimisation cycle begins at Start A (Figure 1) when developing completely new software and at Start B in the case of further development and refinement of existing technology. Different concept-specific local optimisation cycles are used to optimise specific work tasks, depending on the particular type of the project. It is up to the project management to settle on the actual procedure and this decision is reflected in the development form chosen.

In order to reach the goals of a work-oriented design concept [21] [22] the first project phases (requirements analysis and definition) should be replete with a range of different 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.

Simple and fast techniques for involving users include discussion groups with various communication aids (metaplan, 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 [7] [13] [23]. 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 [19] presents a summary of techniques for the analysis and evaluation of interactive computer systems. Comparative studies, e. g. user-oriented benchmark tests [17], 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, maintenance and optimisation of the system in the real-life operating environment.

The effort spent on optimisation in each individual quadrant varies according to the type of the project and of the type. 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 [13] [20].

5. Participatory Methods and Techniques

In the introduction we used systems theoretical considerations to explain how each individual optimisation cycle consists of a *test* and an *action* component which are suitably coupled. Each component can be of a widely varying nature. Table 1 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 optimisation cycles is *synchronisation*. If several optimisation cycles are simultaneously active at different places in the iterative-cyclic software development concept (Figure 1), then these must be suitably synchronised. This is particular important, being the only way to minimise inconsistencies with to 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 programmes for the waste paper basket because they are working to superseded specifications. This problem is caused by the differing lengths of the optimisation cycles involved and it becomes blatant whenever the separate optimisation cycles have not been adequately synchronised.

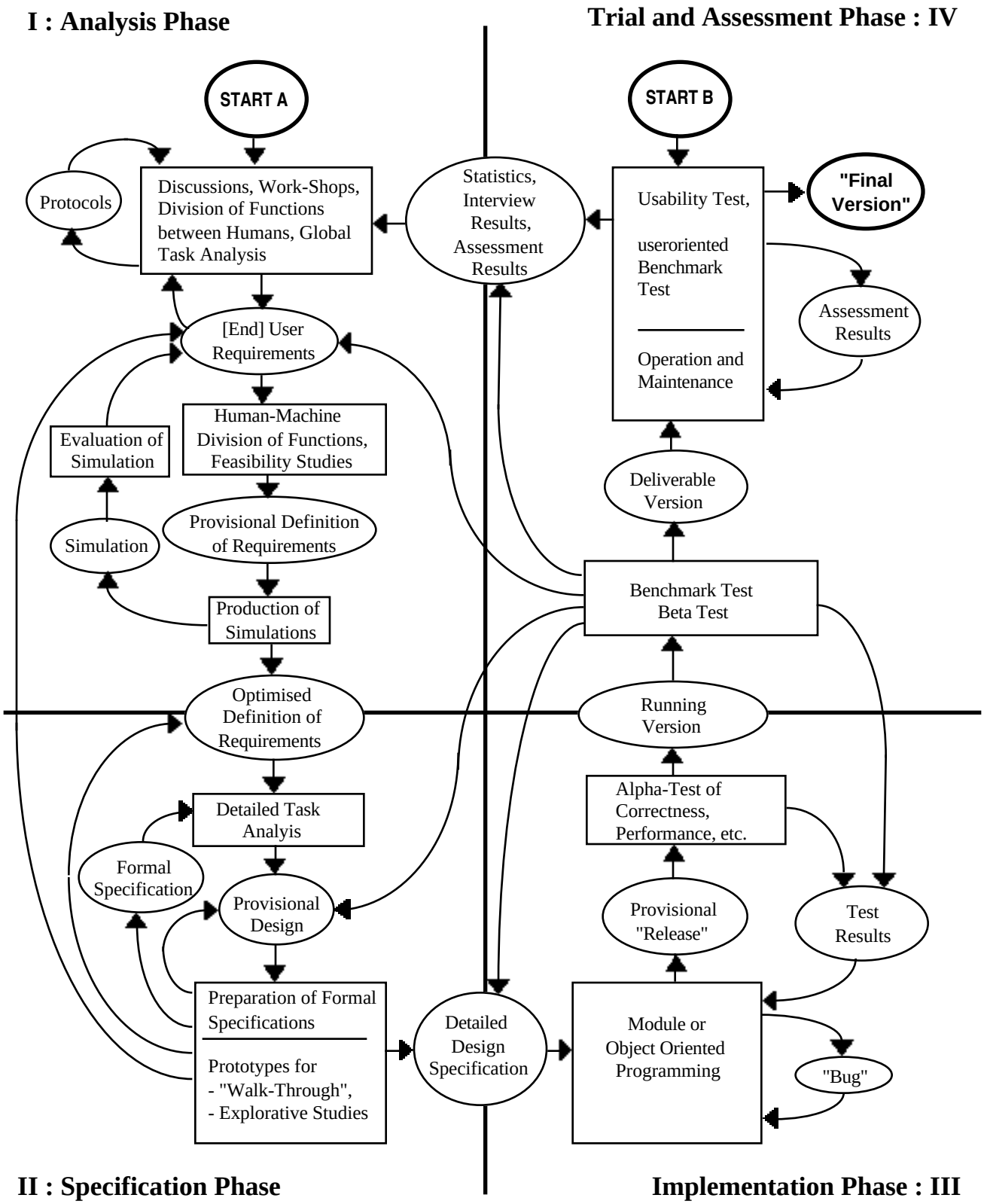

Figure 1 Flow chart for an iterative-cyclic software process model showing the local optimisation cycles within and between individual quadrants (I - IV). The systematic use of application and maintenance cycles with feedback to the requirement phase thus subsumes the version concept as the global optimisation cycle (see also [7]).

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 stuctural 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	useroriented benchmark tests	first largely complete version	months- years
Versions-II	run through entire development cycle	useroriented benchmark tests	several largely complete versions	months- years

Table 1 Overview of Different Methods of User-Participation in the Framework of Optimisation Cycles

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 which are active in parallel. 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).

Another important synchronisation principle is to ensure that participation in the different optimisation cycles includes the *same* circle of people. 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 [21]. Since nobody can be in two places simultaneously, we call this principle the *personal synchronisation principle*. By their own admission software developers significantly underrate the benefits of the *informational* and the *personal synchronisation principle* [11]. We list some further aspects to be considered when applying the various participatory methods, going beyond their influence on the length of the cycle.

Discussion Methods I & II (see Table 1): Discussion is the method most frequently used, because it is fast[1], 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 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 & II (see Table 1): Simulation methods comprise all techniques which illustrate the work system to be optimised in as realistic, visually perceivable way as possible. This ranges from simple quickly completed sketches, through template layouts to formal description techniques.

The unequivocal advantage of formal analytical and descriptive techniques is that they force one to perform a thorough and detailed investigation of the domain to be described. The analysis focusses 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. Additionally, the more formal the method of representation, the more time consumed by its

[1] In the sense of the cycle length; an optimisation cycle in the discussion method is limited to communicative units such as "statement-counterstatement", "question-answer".

preparation. This is partly due to the fact that when users also participate, they first need adequate training in operating and interpreting formal specifications.

Prototyping Methods I, II & III (see Table 1): As already mentioned, prototyping methods make it possible to acquaint end-users with the procedural character of the system being developed. Prototyping is an attempt to adequately image 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 means of communication between the user and the developer [1] [8].

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.

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 prevailing notion of what a prototype in the traditional industrial context 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. Another aspect is that the prevailing 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" [1]. Both of theses aspects support the observation that when prototyping is adopted, "the best prototype is often a failed project" [5]. "The fundamental idea of prototypes is to iterate the design, not to FREEZE it" [8]. Several authors place great value on simpler and quicker participatory techniques [7] [9] [15] [16] to banish this danger.

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. 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 [8]. Industrial norms and standards, and design criteria for interactive software [21] can be of assistance here.

Versions Methods I & II (see Table 1): 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 because it clearly best matches the "Waterfall Model" in a software lifecycle. 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 & Quadrant I). Some authors speak of "evolutionary" software development [12].

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 and maintenance phase. In order 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).

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 would require interdisciplinary cooperation 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 cooperation.

We have presented here an iterative-cyclic software development concept which integrates solution proposals developed to date for overcoming the specification, communication and optimisation barriers based on 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 due to the fact that all subsequent users are involved at least through representatives, and can therefore inject their relevant knowledge into the design of the work system.

The more effort that is expended on optimisation in the first quadrants, the less is needed in Quadrant IV. 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 which the user finds easy to understand. Employing powerful development environments and suitably qualified software developers minimises the investment in Quadrant III. But first, we must start learning to jointly plan 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

1. BOAR B H, 1984: Application Prototyping: A Requirements Definition Strategy for the 80s'. New York: Wiley.
2. BOEHM B W, 1981: Software Engineering Economics. Englewood Cliffs: Prentice Hall.
3. BOEHM B W, 1988: A spiral model of software development and enhancement. Computer (May 1988):61-72
4. 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
5. CURTIS B, KRASNER H, SHEN V, ISCOE N, 1988: A Field Study of the Software Design Process for Large Systems. Communications of the ACM 31(11):1268-1287
6. FOIDL H, HILLEBRAND K & TAVOLATO P, 1986: Prototyping: die Methode - das Werkzeug - die Erfahrungen. Angewandte Informatik 3:95-100
7. 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
8. 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
9. KARAT C-M, 1990: Cost-Benefit Analysis of Iterative Usability Testing. In: DIAPER D et al. (ed.) Human-Computer Interaction - INTERACT '90. Amsterdam: Elsevier Science. 351-356
10. KLOTZ U, 1991: Die zweite Ära der Informationstechnik. Harvard Manager 13(2):101-112
11. KRAUT R E & STREETER L A, 1992: Coordination in Large Scale Software Development. Communications of the ACM:(in press)
12. LEHMAN M M & BELADY L A, 1985: Program Evolution - Processes of Software Change. London: Academic.
13. MACAULAY L, FOWLER C, KIRBY M & HUTT A, 1990: USTM: a new approach to requirements specification. Interacting with Computers 2(1):92-118
14. NAUR P, 1985: Programming as Theory Building. Microprocessing and Mircoprogramming 15: 253-261
15. NIELSON J, 1989: Usability Engineering at a Discount. In: SALVENDY G, SMITH M J (eds.) Designing and Using Human-Computer Interfaces and Knowledge Based Systems. Amsterdam: Elsevier Science. 394-401
16. NIELSON J, 1990: Big paybacks from 'discount' usability engineering. IEEE Software 7(3):107-108
17. 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
18. SCHIEMENZ B, 1979: Kybernetik. In: KERN W (ed.) Handwörterbuch der Produktionswissenschaft. Stuttgart: Poeschel. 1022-1028
19. SPENCER R H, 1985: Computer usability testing and evaluation. Englewood Cliffs: Prentice Hall.
20. STROHM O, 1991: Projektmanagement bei der Software-Entwicklung. In: ACKERMANN D &ULICH E (Ed.) Software-Ergonomie '91. (Reports of the German Chapter of the ACM, Vol. 33). Stuttgart: Teubner. 46-58
21. 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
22. VOLPERT W, 1987: Kontrastive Analyse des Verhältnisses von Mensch und Rechner als Grundlage des System-Designs. Zeitschrift für Arbeitswissenschaft 41:147-152
23. 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
24. YAVERBAUM G J & CULPAN O, 1990: Exploring the Dynamics of the End-User Environment: The Impact of Education and Task Differences on Change. Human Relations 43(5): 439-454

Proceedings

Fourth International Conference on Software Engineering and Knowledge Engineering

SEKE'92

June, 15-20, 1992

Europa Palace Hotel

Capri, Italy

IEEE Catalog Number: 92TH0438-2
ISBN (paper): 0-8186-2830-8

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo