Human IT
Tidskrift för studier av IT
ur ett humanvetenskapligt perspektiv

# On Process Quality in Integrative Frameworks for Information Systems Development

## by Per A. Zaring, Dept. of Computer and Business Science, Högskolan i Borås, Sweden

Per Zaring

---

## *Abstract*

*Single and monolithic for information systems development are by many considered to represent outmoded ideas, and are today being substituted by more integrative perspects, in this paper exemplified by Boehm's spiral model. Risk handling is considered to be the new big area for benefits. Apart from explicit strategies for risk handling, both specific development approaches and software tools are included in the spiral model. Let prototyping as well as software tools denote two concepts both of whom aim to increase certainty about the intended properties of a system, as well as development speed. In this paper we investigate how prototyping, use of tools and an effective software process interplay. The result obtained indicate that prototyping, supported by tools, not automatically contributes to a more effective software process, not even within the spiral model.*

---

## Innehåll

1. Introduction

2. The Software Process

3. An Integrative Model

4. Optimizing the Software Process

5. Discussion

6. Conclusions

The Author

---

# 1. Introduction

Single, detailed and monolithic models for information systems (IS) development, are by some [13, 2] considered to represent outmoded ideas, and are today being substituted by more integrative perspectives. Taking into account all those approaches that in different ways have contributed to increased competence on IS development over the decades, integrative views enable substantial effectiveness enhancements within the process of developing information systems.

The spiral model proposed by Boehm, is one representative to the integrative perspective. The spiral model's primary contributions are, for our concerns, that it (a) takes a meta position in that it incorporates other known models and (b) that it handles environmental complexity by working with system increments and by the explicit recognition of risks. Complexity, if not tackled properly, could mean risk. To the developer of infor-mation systems, both tools and techniques play important roles in the pursuit of developing the right system in the right way and at the lowest possible cost. Integrative approaches tackle this by encouraging, e.g., a recurring production of prototypes. Approaches for prototyping:

- can be found in integrative development models
- for too many people, legitimate unstructured thinking and action
- are often supported by tools
- enable shortcuts in the development process that are in the shortterm appealing but in the long-term disastrous.

Developing systems by following the spiral model is likely to cause high development costs [21, 15]. However, these increased costs should be put in contrast to the significant increase in final quality of the systems produced. Some [15] therefore suggest regular use of software tools in the spiral process, to shorten the time between sweeps in the spiral.

Let *prototyping* and *software tools*denote two concepts both of whom aim to increase *certainty* about the intended properties of a system, as well as development speed. In this paper, we investigate how prototyping, use of tools and an effective software process interplay. In particular, we do this with reference to the spiral model.

(Åter till början av artikeln)

# 2. The Software Process

Software development involves the application of models and techniques to govern the creation, assimilation and maintenance of a software system. Taking into consideration all the steps and documents that follow from a "traditional" sequential life-cycle approach, software development is a slow process which can last for several years. As these are

activities that involve many people, a dominant concept in software development is *communication*. Source code, design documents and other work products are intended to be communicated

## 2.1 A General Process Model

Basically, the problem lies in how to transform an informal, generally incomplete, description of a computer application concept into a system specification and the transformation, in turn, of that specification into a final product as illustrated in the model below.

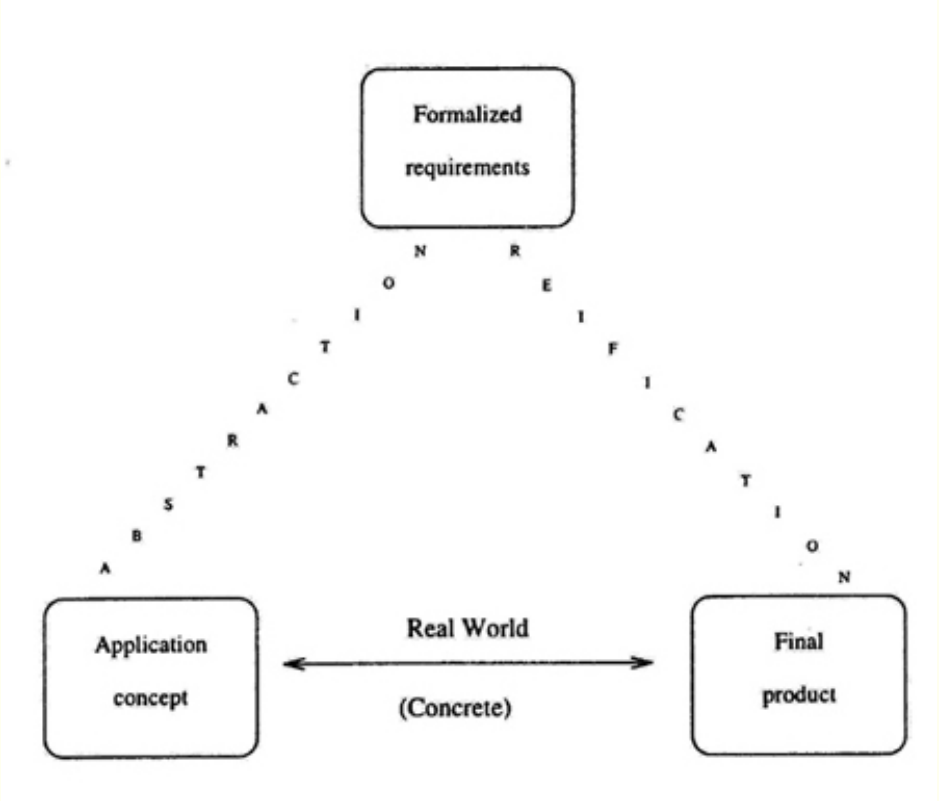


*Figure 1 A general process model*

Depending on the original problem formulation we may have to reformu-late it several times, each time making it more "workable". We achieve a problem statement of appropriate "workability" by moving between different levels of *abstraction*.

## 2.2 Analysis and Design as Process Components

Since problems are perceived differently among individuals, the abstraction must necessarily enable a broader understanding in general terms of a concept. The more novel, unfamiliar or vague an application concept, the more critical becomes the analysis phase to the total outcome. A designer is said to start with an initial specification which is a statement of the *requirements* of a system. The subsequent design meets these requirements by reifying them in *structural* terms. While analysis covers any activities to elaborate the application concept in order to reveal as many of its aspects as possible, the design essentially carries out the structural (technical) representation of some of them.

(Åter till början av artikeln)

# 3. An Intergrative Model

Starting during the mid 80'ies, ideas about the software process and how to assimilate a number of basic approaches began to form a new cohesive integrative paradigm. Of the

number of basic approaches began to form a new cohesive integrative paradigm. These driving forces, significantly changed conditions for both software producers and software users (These new or changed conditions are mainly due to major changes in society, forcing both individuals and companies to act in a considerably more *turbulent, uncertain* and *hostile* environment. See for definitions and further discussions [8, 9, 10, 11, 22].)were the most prominent. These conditions forced the need to more responsively take care of risk-related phenomena. Among a number of alternative, more or less coherent, process models, particular success as to general applicability, has been reported concerning Boehm's *spiral model* [2].

The spiral model is an attempt to manage the fact that all projects deal with risks, uncertainty and *ad-hoc* problem solving. When inadequate and/or insufficient information is available about a situation one whishes to be able to quantify the risk of making a decision. In his model, Boehm regards "risk" as a central concept which should be considered repeatedly in a development project. Risk management brings in a six-step program of activities classified as either risk *assessment* activities or risk *control* activities. See further [3].

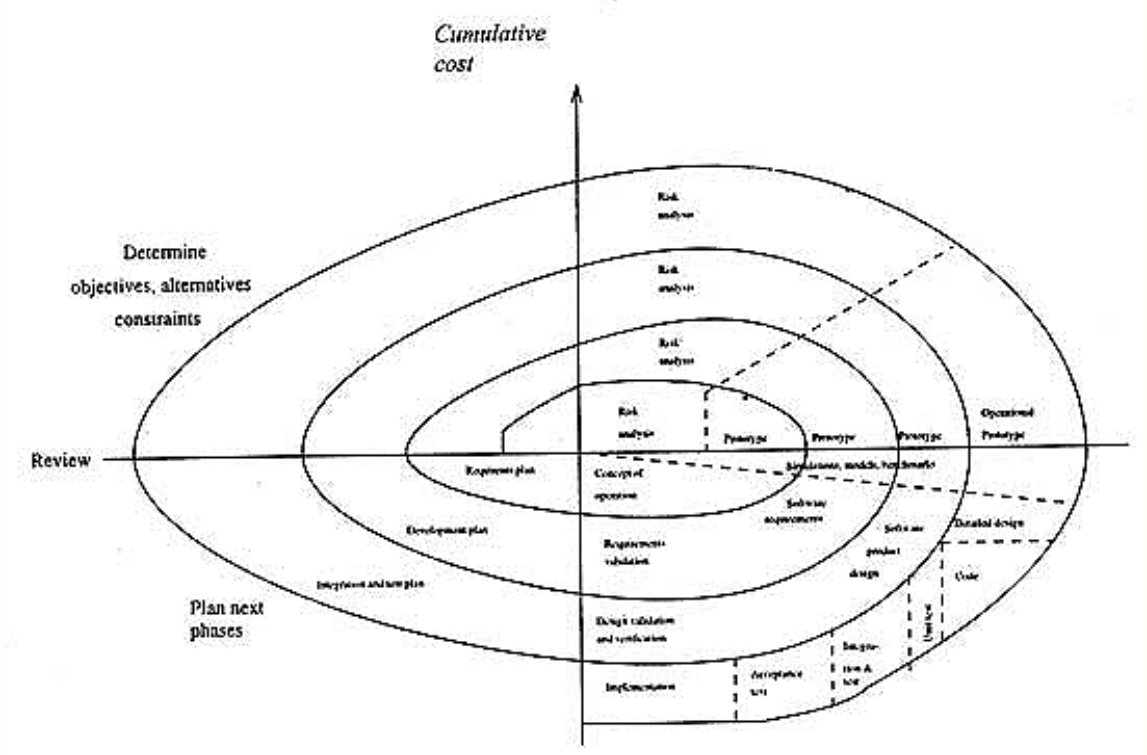


*Figure 2: The spiral model according to Boehm [2].*

The model embraces, to a large extent, other models as special cases. The most significant difference between this model and other, more traditional approaches, is the explicit recognition of alternative means of meeting project objectives and the identification of risks associated with each alternative. Basically, the idea is incremental development, using the Waterfall model for each step. However, the Waterfall model alone requires that all specification and design be done first and followed by the coding. Unfortunately, it is too difficult for the human mind to comprehend and understand all unknowns at first. Therefore, the spiral approach suggests specification, design, and coding of one subsystem at a time. Consequently: do not define in detail the entire system at first. The developers should only define the highest priority features and implement those. With this knowledge, they should then go back to define and implement more features in smaller chunks.

The number of cases based entirely on the original spiral model are tively few. However, a

The number of cases based entirely on the original spiral model are fairly few. However, a number of variants have been evaluated. For instance Wolff [21] bases his work on the assumption that the idea of the spiral model is based on prototyping. Here, the spiral process does not perform one spiral per sub-system, but one spiral for each level of completeness of the *entire* system. The first spiral cycle is a prototyping cycle, where some portion of the requirements that are representative of the eventual full requirements, and that are well understood, drive a full life cycle spiral. Based upon the understanding attained by this prototyping cycle, a more mature complete set of requirements is defined. These new requirements are the start of the second cycle, which again is a full life cycle spiral. As each spiral is completed, the understanding of the system matures, the individual life cycle objects (requirements, specification, design, implementation, testing materials) also mature. To this, one may object that Wolff's view on the development process is strongly idealized. This is because once a prototype is established, requirements are often ignored since customers are only interested in the end result. Too often prototypes become "operational prototypes" that are released without any more actual design being done. These operational prototypes are incorporated also in Boehms original model (above). Since prototyping is a vital component of the spiral model, we shall investigate the concept in more detail.

(Åter till början av artikeln)

# 4. Optimizing The Software Process

Different means can be used to optimize the process of developing a system. By optimization, we here mean activities to keep the development process cost-effective. In practice this is achieved by minimizing development time (DT). With reference to figure 1, we use the following notion:

*DT1*:
- Development time for one version => 360°

*DTtot = DT1 x ∑versions:*
- Total development time

The development time consists of all activities leading towards a final product. Where can time reductions be made, and how? Roughly, we can do it by using methods, tools or a combination of the two. As creating *versions* is a recurring concept, it is reasonable to start looking at it as a potential area for optimization. Let us, in this chapter start with the methods discussion. We restrict ourselves to prototyping approaches for three reasons:

1. prototypes are essential in the spiral model
2. prototyping is "dangerous" because of the relatively low degree of process formalization.
3. it is and old area for tool support

As the last section of the chapter, we will elaborate the impact of tools in the pursuit of trying to reduce both the number of sweeps in the spiral models as well as the time it takes to make a single sweep, which is what it is all about.

## 4.1 Prototyping Approaches

As mentioned in the previous chapter, traditional working patterns mean that definition is

followed by specification, construction and testing of the complete system. This method can work if *all* requirements of the final system are known from the outset. This is however not always the case [22]. What about when knowledge of the system grows progressively as work progresses, i.e. when software requirements are not known and understood from the outset? It is exactly this situation which Bohem suggests should be resolved by building prototypes. Prototyping is the well known approach that allows requirements to evolve as experience is gained. It is an approach which is exploratory in the sense that the first phase of development involves developing a program for user experiment. To be meaningful to use this approach, it is assumed [20, 12, 19] that the requirements are known only partially at the beginning. Humphrey [12] elaborates this by using a taxonomy of *requirements instability* ranging from known and stable requirements to not known requirements. The more an application is changed, the less accurate are its requirements. There are some criteria that should be satisified to successfully employ prototyping:

1. The initial prototype must be "low cost" [20],
2. The use of any method and tool available should be encouraged, such as *visual interface programming* [18] and very high-level languages [1].
3. There is no "right" way to prototype. Even if it is only intended as a quick experiment, the prototypes' overall objectives should be clearly established before starting to build it [12].
4. When the prototype is to be included as part of the final product, the need for design records, user documentation and other facilities must be recognized [22].

There is a number of risks reported by Boehm [4] related to the use of prototyping which deserve to be mentioned:

A. Prototyping tends to create proportionally less effort planning and designing, and proportionally more testing and fixing.
B. Prototyping tends to result in more difficult integration due to lack of interface specifications.
C. Prototyping tends to create a less coherent design.

These effects become proportionally more critical with increasing system size. It is our experience [22] that the final system size as well as the range of application for it is initially hard to estimate. This suggests that prototyping should be accompanied by a reasonable level of specification of the product. This is exactly what is done in the spiral model. It should be emphasized once more, that prototypes in the spiral model are developed with different purposes, all depending on the current position in the spiral.

## 4.2 Tools for the Software Process

Tools for the development process have, until recently, been available useful only to

proffesional developers. We are here referring to e.g. pre-compilers, debuggers and more or less sophisticated editors for different purposes. The situation today is different.

The development of working manners along with technology advances over the decades, can be described as a circle starting 30 years ago with many small systems, locally developed for local purposes. The systems then grew larger and so did the range of applications. This growth were soon recognized as a problem which resulted in a number of attempts to manage the software process. IS management became crucial for organizations

to such an extent that development and related activities became centralized. New computer architectures made possible less centralized IS configurations. However, development activities were still centralized. As industrial progress was made (personal computers, application software packages) the potentials for local users were recognized regarding e.g. local customization of applications. More and more sophisticated technology today permits the local user to become less and less dependent on that "bottleneck" called the *"System Department"*. Even though the conditions aren't the same today, we are back were we started, i.e. local development for local purposes. This development is made possible by a wide range of tools and high-level languages for different purposes. Current technology, thus, enables even managers not only to use, but also to build their own computerized systems (The situation was recognized as early as 1982 with the change of availability of microcomputers, see James Martin [17]. Today, however, we are no longer struggling with hardware, but instead with software.) .

In parallell with the increase of availability of software support tools for amateurs, proffessional tools are becoming more and more capable as well. We are here primarily referring to that large class of software which are usually goes under the name *CASE tools*. These tools provide automated or semi-automated support for one or many system development methods. Also to the professional system developer, rapid prototyping is valuable. All high-level software, including CASE tools, therefore support prototyping. The features included for this purpose, are e.g. interface tools, code generators and expert system tools designed to automate as many of the development phases as possible.

## 4.3 Tools and Prototyping

### 4.3.1 Planning for Change

The "hype" surrounding high-level tools often leaves the developers/users with extremely high expectations which mislead them to disregard software managerial issues such as those regarding planning for change. The problem of software changes is central in any software engineering context. There is always a need to change a system in order to improve it, add new features, or fix any problems still left after the warranty is over. Most of the time, the user's business will change with time and so will his requirements. These changes or enhancements are called maintenance. Software maintenance is not a trivial task but consumes alarmingly huge resources, a problem which is thoroughly investigated by Boehm [5, 6] and others. Consider in this context that the amount of effort spent on software maintenance is between 65% and 75% of total effort according to Sommerville [19]. Macro [16] states that the apparent ease of changing programs misleads people to think that software is flexible. As software tools continue to find their ways into organizations, a critical success factor is whether they fit into the current (established) development environment.

### 4.3.2 Software Crisis Revisited?

The most fundamental reason for a new, or exacerbated, software crisis is that methods and most existing software tools mostly adress the implementation part of the software development process and, furthermore, do this poorly. They provide no, or limited, assistance to the software specifiers and software designers. Except the "old" problems, identified for more than 30 years ago, the crisis today consists of new ones brought by new technology such as the lack of methods to support "ad-hoc" development and the problem of integrating odd development styles with an ordinary software development environment.

### 4.3.3 Methods and Tools -A Problem of Optimization?

Both tools and prototyping is used to reduce (a) the time to make one sweep in the spiral and (b) the number of sweeps required. The more accurate the requirements specification, the less there is a need to building additional versions. The ideal case would then be:

DT*tot* = DT1

Assume that there exists a linear relationship between length and time. When you shorten the length of the spiral, you gain time. Consequently, several small sweeps could be performed to the same cost as one large sweep. This is typically done by building prototypes. However, to optimize the process it would not suffice to only shorten the way, because you would have to pay for it by making a larger number of sweeps (a larger number of versions). To be able to also reduce the number of sweeps, we have to consider qualitative aspects of the process, i.e. means that help us to do the same things, but faster with the same quality. With this as a constraint, many expensive, over-functional software packages become disqualified. Those that remain have all in common that they in one way or another directly support the software life cycle phases. Consequently, very simple drawing tools would qualify. It all presumes that prototyping (when used in this context) can be nothing but an approach to sequre requirements. Prototyping and tools together, within the integrative model described in this paper, should thus complement the shortcomings of each other:

- Prototyping for doing the right thing
- Tools for doing the things right

Thus prototyping, to be able to function as a complement in the spiral model, should be used in accordance with the three first criteria in section 4.1. If used in that way, the risks listed in the same section would be limited to the first (A).

(Åter till början av artikeln)

# 5. Discussion

Software industry presents us with an ever increasing number of analysis-, design-, implementation, validation- and “all-in-one” software packages which are aimed to facilitate gathering, storage, retrieval, processing and presentation of information which is relevant within the process of creating a system. In the pursuit, for organizations, of trying to cope with a more and more *complex* or *imperceivable* [14] environment, focus has shifted from manufacturing of goods to the "manufacturing" of services, e.g. information. As information as a concept gains in importance, so does the use of computers and consequently also the methods and techniques that guide us in the quest of formulating and achieving certain information system (IS) characteristics.

Current software for this purpose builds on the ambition to provide valid representations of complex real world phenomena within a short time frame. Software manufacturers often promise that their technology will improve productivity significantly. One would therefore expect that they also provide solutions to the problems of assimilating this technology with the rest of the computing environment in an organization. This is the backside of the coin. We argue here that organizations, to be successful, always must consider the adoption of

new technology as a completely new concept. This means that to adopt new tools, they should always be prepared to adopt new methods as well. The prototyping capabilities in modern high-level software is one example of technological progress which, if not handled properly, would violate sound methods in the long run. Thus, prototyping and tools together, from this perspective could mean a threat to the organization, rather than an opportunity.

### 5.1 Tools -Some Remarkable Observations

The "jump to the code" philosophy supported by software tools, naturally appeals to result-oriented people. However, we have experienced that management seem to bring in sophisticated support software to rescue failing projects [22]. This serves as a divergence from the real problems (e.g. inadequate analysis of tasks). The problem is that software technology, if it is considered "new" to an organization, will slow down the project due to learning curve. Even when an experienced consultant is assisting, there is usually so much retrofitting needed that whatever time is saved, it is lost in this activity. The learning required, if based on the vendors' material (which is more or less a necessity), becomes to shallow in that the *tools* are focused and not the *concepts* and *methods* such as structured techniques, engineering principles and teamwork concepts. Those areas are all neccesary in order to make applications smoothly fit into the "old" orinary data processing environment.

(Åter till början av artikeln)

## 6. Conclusions

In this paper, we have investigated how *prototyping, tools* and a *high-quality software process* relate to each other. In particular we have used Boehm's spiral model [2] as a reference. Society was said to impose new requirements upon artifacts such as information systems today. By defin-ing the software process as a flow from *abstraction* to *reification*, these new requirements were related to risk and uncertainty in development. Both the project *process* as well as the *contents*, e.g. what to do, were shown to be affected. To be able to act under more certain conditions, risk mitigation strategies exemplifie new and central concepts for development. These concepts, together with a mix of established development approaches contribute to more dynamic software models, here exemplified by the spiral model.

Apart from specifict strategies for risk mitigation, both software tools and techniques were said to contribute to a more certain development environment. In particular, prototyping and software tools, were investigated and found to be strongly related to each other. What is stated here about the dependencies beween prototyping and the tools used, has the potential to make a significant impact on software development. When using tools, no matter in which context, a property like transparency is of great value. Buchanan [7] points out that transparency is the key idea for making the system understandable despite the complexity of the task. As the system, when practicing prototyping, matures through incremental improvements, thorough understanding of previous versions is crucial. Thus, development history must be documented even in prototyping, in particular since the system improves through criticism from persons who are not familiar with the system specific details. Within an integrative framework, the different parts must not counteract each other. We here view risk management as a pro-active management tool. Any interesting and challenging project involves uncertainties associated with known-unknowns and/or

unknown-knowns. The objective of risk management is to identify and assess these risks, and to mitigate them before they manifest themselves as significant performance inadequacies, cost over-runs, or significant delays. This is very different than simply reacting to a problem once it has surfaced. The spiral evolutionary model provides a good framework for dealing with program uncertainties and risks, provided that tools and methods (here prototyping) are chosen so that they complement each other as discussed in section 4.3.3.

(Åter till början av artikeln)

**The Author Per Zaring is Senior Lecturer in Informatics, Department of Computer- and Business Science, Högskolan i Borås, Borås, Sweden. In his current research he focuses on communication processes and new technology. The present paper gives a retrospective of software engineering research carried out in the early nineties.**

## References

[1] **Bell**, Doug, Ian Morrey, and John Pugh in Software Engineering -A Programming Approach, Prentice Hall (1992).
[2] **Boehm**, Barry, ‘‘A Spiral Model of Software Development and Enhancement,’’ ACM SIGSOFT Software Engineering Notes (August 1986.).
[3] **Boehm**, Barry, ‘‘Software Risk Management: Principles and Practices,’’ IEEE Soft-ware (January, 1991).
[4] **Boehm**, Barry, ‘‘Prototyping Versus Specifying: A Multiproject Experiment,’’ IEEE Transactions on Software Engineering, vol. SE-10, no. 3 (1984).
[5] **Boehm**, Barry, ‘‘Software and its Impact: A Quantitative Assessment,’’ Datama- tion, no. 19 (1973).
[6] **Boehm**, Barry, ‘‘Software Engineering,’’ IEEE Transactions on Computers (1976).
[7] **Buchanan**, Bruce G. and Edward H. Shortliffe, ‘‘RULE-BASED EXPERT SYS-TEMS, The MYCIN Experiments of The Stanford Heuristic Programming Project,’’ Addison Wesley (1984).
[8] **Huber**, George P., ‘‘The Nature and Design of Post-Industrial Organizations,’’ Management Science, vol. 30, no. 8 (1984).
[9] **Huber**, George P. and Reuben R. McDaniel, ‘‘Decision-Making Paradigm of Orga-nizational Design,’’ Management Science, vol. 32, no. 5 (1986).
[10] **Huber**, George P. and Richard L. Daft, ‘‘The Information Environment of Organi-zations’’ in Handbook of Organizational communication, ed. F.M. Jablin and L.L. Putnam, Sage (1987).
[11] **Huber**, George P., ‘‘A Theory of the Effects of Advanced Information Technologies on Organizational Design, Intelligence and Decision Making,’’ Academy of Man-agement Review, vol. 15, no. 1 (1990).
[12] **Humphrey**, Watts S., ‘‘Managing The Software Process,’’ Addison-Wesley (1989).
[13] **Ivari**, J., ‘‘Hierarchical Spiral Model for Information System and Software Development. Part 1: Theoretical Background,’’ Information and Software Technology, vol. 32, no. 6 (1990).
[14] **Langefors**, B. in Essays on Infology, ed. Bo Dahlbom, Gothenburg Studies in Infor mation Systems (1993).
[15] **Lewis**, T.G. in CASE: Computer-Aided Software Engineering, Van Nostrand Rein-

[15] **Lewis**, T.G. in CASE: Computer-Aided Software Engineering, Van Nostrand Reinhold (1991).
[16] **Macro**, Allen, ‘‘Software Engineering -concepts and management,’’ Prentice Hall (1990).
[17] **Martin**, James in Applications Development Without Programmers, Prentice Hall (1982).
[18] **Shiller**, Larry in Software Excellence, Yourdon Press Computing Series (1990).
[19] **Sommerville**, Ian, ‘‘Software Engineering 4 ed.,’’ Addison-Wesley (1992).
[20] **Turban**, Efraim in Decision Support And Expert Systems, Maxwell Macmillan Int. (1988).
[21] **Wolff**, Gerard, ‘‘The managemnet of risk in system development: “Project SP” and the “New Spiral Model”,’’ Software Engineering Journal (May, 1989).
[22] **Zaring**, Per A., ‘‘The Impact of Tool Based Application Development on Software Management,’’ Licentiate Thesis, Department of Information Systems (1994).

---



Åter till Human IT 1/1998