6

The Principle of Limited Reduction*

Lars Mathiassen
Jan Stage

Abstract. This paper provides a simple but powerful theoretical framework for understanding and combining different approaches to software design. The main result is expressed in what we call The Principle of Limited Reduction. This principle states that relying on analytical behavior to reduce complexity introduces new sources of uncertainty and this requires experimental countermeasures. Correspondingly, relying on experimental behavior to reduce uncertainty introduces new sources of complexity requiring analytical countermeasures.

1. Introduction

Design of computer-based systems is a challenging and difficult task. Ideally, systems developers have to understand and appreciate the details, relations, and qualities of the user organization. They have to be professionals in evaluating and handling the technology involved. They must be able to meet the requirements of the user organization in a way that is both technically and socially implementable. Finally, they have to operate in a turbulent environment where ideas, preferences, and contractual arrangements change as they perform their task. From this point of view, it is not surprising to find that approaches to complexity and uncertainty has been a major concern within computer and information science.

Today, it is generally accepted that we can cope effectively with complexity in software design through abstraction and decomposition (Dijkstra 1972; Langefors 1966; Wirth 1973; Wulf 1977). Much

attention has also been drawn to the uncertainties related to software development (Boehm 1988; Davis 1982; Floyd 1984; Gomaa *et al.* 1981; Gould *et al.* 1985) and new approaches have been promoted. In the following, we will discuss effective approaches to software design by looking closer at complexity and uncertainty as important characteristics of a situation. In doing so, we will use a set of simple concepts for relating situational characteristics to different approaches (figure 1).

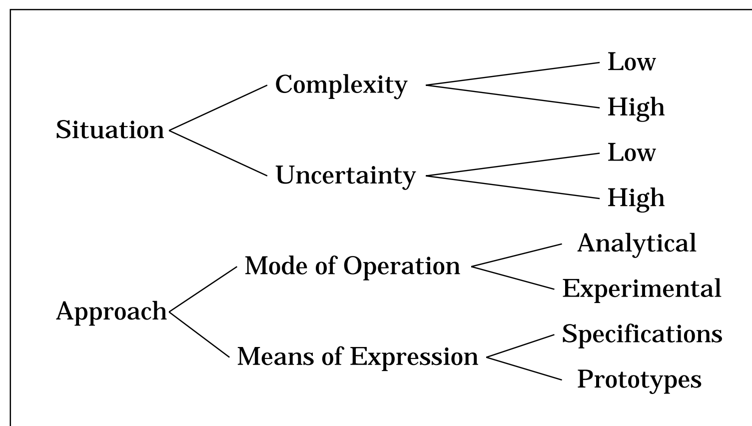

Figure 1. Situational characteristics and basic approaches to software design.

The basic characteristics of a design situation are described in terms of their degree of complexity and their degree of uncertainty. The degree of complexity represents the amount of relevant information that is available in a given situation. In contrast, the degree of uncertainty represents the availability and reliability of the information that is relevant in a given situation.

A *design approach* is characterized by its mode of operation and its means of expression. The *mode of operation* defines how systems developers process information in order to make design decisions. The mode of operation may in the one extreme be analytical and in the other experimental. When systems developers operate in an analytical mode, they simplify the available information through abstraction in order to reach a deeper and more invariant under-

standing. In contrast, when operating in an experimental mode they learn from experiences generating new information.

In addition, each design approach involves specific *means of expression. Specifications* can be used as means of expression to describe abstractly the properties and behavior of the system. As opposed to this, different models such as *prototypes* and mock-ups can be used as means of expression to illustrate the behavior of part of the system.

We are aware that some of these concepts appear simple. Our purpose is not to engage in a general discussion on the strengths and limits of analytical and experimental behavior in organizations. Instead, we want to identify, compare, and combine basically different approaches to software design. Our main result is expressed in:

> *The Principle of Limited Reduction:* Relying on an analytical mode of operation to reduce complexity introduces new sources of uncertainty requiring experimental countermeasures. Correspondingly, relying on an experimental mode of operation to reduce uncertainty introduces new sources of complexity requiring analytical countermeasures.

We will argue that this principle provides a simple, but powerful theoretical framework for understanding and combining different approaches to software design. One important and surprising implication of this principle is that effective software design requires a systematic effort combining analytical and experimental modes of operation, regardless of whether specifications or prototypes are used.

In section 2, we focus on means of expression and review the use of specifications and prototypes as two distinct, traditional approaches to software design. In section 3, we focus on fundamental modes of operation by revisiting well-known principles of human problem-solving. In section 4, we present a new way to relate means of expression to modes of operation in software design; and we argue that the effectiveness of software design is restricted by the Principle of Limited Reduction. In section 5, we conclude by discussing the limits and weaknesses of our argument, and by presenting ideas for further research.

2. Specifications and prototypes

Software development projects over a minimal size use variations or combinations of approaches based on specifications and prototypes

(Boehm *et al.* 1984; Davis 1982). In some situations, software development is based on a fairly formal specification of requirements. This requirements specification is transformed into the final system in several phases. Each phase develops a new specification or description by transforming the description produced in the previous phase. In other situations, software development is based on design, implementation, and evaluation of prototypes where each version models part of the total system. The design of the system is determined through realistic use of the prototypes. In the following, we examine more closely the basic characteristics and modes of operation related to the use of specifications and prototypes as basic means of expression.

2.1. Software design based on specifications

Approaches based on specifications have traditionally focused on reducing the complexity of software design (Wulf 1977). The typical mode of operation is analytical. Systems developers are expected to analyze and describe problems, options, and designs before actually implementing the system. The key technique is abstraction, providing clarity through selection and structuring of relevant information. The resulting information is expressed in specifications forming the basis for communication with users, design and implementation, measurement of progress, and division of labor among systems developers.

An analytical use of specifications is widely recommended in programming, where abstraction and decomposition are used as general techniques. The point made is that human beings have limited ability to handle large amounts of information. This implies that programmers facing a complex and elaborated set of requirements have to focus on certain aspects and ignore others. They have to dissect and decompose the problem in order to solve it (Dijkstra 1972). These observations and ideas have led to a variety of approaches, one of them known as stepwise refinement. This approach involves a series of decompositions where each decomposition yields a description of the program that is more detailed than the previous version (Wirth 1973).

Abstraction and decomposition are also widely accepted as fundamental techniques within the broader area of information systems development as illustrated by Langefors' basic principle for system construction. The general assumption is that the system is

too complex to comprehend because of its many parts and the complicated relations between them. The idea in Langefors' principle is to handle this problem through decomposition of the system into a specified set of elements. The overall properties of the total system are described, the system is divided into a set of elements, the relations between elements are specified, and, finally, the system consisting of these elements is checked to make sure it contains the specified properties. The principle may then be applied repeatedly to each of the elements (Langefors 1966).

The stagewise model is a simple implementation of an analytical mode of operation based on specifications. The basic idea is that development should proceed through a pre-defined set of successive stages or phases in a linear way. Each phase requires a well-defined input, utilizes well-defined tools and techniques, and produces well-defined outputs. The idea behind this model was presented in the mid fifties. It is still the basis for software standards in some development organizations.

The waterfall model refines the stagewise model by emphasizing feedback between subsequent phases. The first version of the waterfall model was presented as early as 1970 (Royce 1970); and since then it has had major influence on the field of software engineering. It has become the basis for software standards of most development organizations (Boehm 1988), many life-cycle methods integrate its basic ideas e.g. (Yourdon 1982), and many text books on software engineering use it as a basic frame of reference e.g. (Fairley 1985; Pressman 1987). The waterfall model is a modification of the more strict and analytical assumptions behind the stagewise model acknowledging the experimental aspects of software design.

Parnas and Clements argue that software can never be developed in a perfectly rational and analytical way. Requirements are vaguely and ambiguously defined, requirements change during the development process, implementation invalidates previous design decisions, and details left out in earlier phases turn out to be of major importance. In a typical software development project, programmers are often forced to make design decisions without having good reasons (Parnas *et al.* 1986).

These statements are supported by others. In a discussion of the waterfall model, Boehm argues that a major source of difficulties with this model has been its strong emphasis on fully elabo-

rated documents as a completion criterion for the early requirements and design phases. When developing interactive end-user applications, the earlier phases are often characterized by a very limited understanding of the work tasks and needs of the users. Elaborate specifications do not solve this problem. They only lead to production of a large quantity of unusable code. According to Boehm, the waterfall model is most effective for development of basic software like compilers and operating systems (Boehm 1988). McCracken and Jackson argue more generally that no single life cycle model is applicable for all software development projects. They claim that the life cycle concept is too simple as a general model of software development (McCracken *et al.* 1982).

These problems lead Parnas and Clements to conclude that the analytical mode of operation should only be considered as an ideal for software design. In practice, it is not possible to operate in an analytical way. Instead, systems developers should fake an analytical process. Parnas and Clements suggest that descriptions and specifications should be structured as if they had been developed in an analytical mode even though they are made, reviewed, extended, and modified in an experimental mode (Parnas *et al.* 1986).

In summary, the *use of specifications* has traditionally been related to an *analytical mode of operation* in software design. Systems developers are advised to take advantage of abstraction in using specifications to reduce complexity. This approach to software design has some basic limitations. It relies primarily on available information, it assumes that the available information is reliable, it implies serious simplifications, and it restricts the ways in which organizational actors can communicate and learn about the future system. There have been attempts to modify the use of specifications in a more experimental direction. However, the basic problems still remain.

2.2. Software design based on prototypes

Theoretical and practical problems with the use of specifications have led many authors to argue in favor of the use of prototypes (Basili *et al.* 1975; Boehm 1988; Davis *et al.* 1988; Ehn 1988; Floyd 1987; Gladden 1982; Gomaa *et al.* 1981; Gould *et al.* 1985; McCracken *et al.* 1982). Approaches based on prototypes are mainly concerned with reducing the uncertainty of software development. The typical mode of operation is experimental, where systems devel-

opers are expected to develop and evaluate realistic prototypes and to learn from experiences as the primary basis for making design decisions. Experiments generate new information and provide practical insight into the relevance of possible solutions. During development, prototypes serve as vehicles for communication and cooperation between systems developers and different user groups. The ideal systems development project includes learning, communication, and negotiation about problems and possibilities as essential activities. These activities are deemed necessary due to the variety of competing requirements that emerge in any system.

One of the early techniques that integrated an experimental mode of operation was iterative enhancement. With this technique, development starts with a simple initial skeleton implementation of a subset of the system. This subset is then gradually refined and extended in order to reach the full implementation (Basili *et al.* 1975).

Today, evolutionary development (Floyd 1984) is the most consistent implementation of an experimental mode of operation based on prototypes. The idea is to develop a number of operational versions of the system where each new version includes and extends the set of functions implemented in the previous versions. This kind of prototyping makes it possible to combine iterative development of the system with gradual adaptation to poorly understood or changing user demands and requirements. It has received much attention since it resembles the way systems developers typically develop systems to support their own work.

In most software development projects, prototypes are used in a more restricted way. Floyd discusses two such ways (Floyd 1984). First, a prototype can be used to clarify desirable features of the future system. Requirements are determined through discussion and evaluation of prototypes that implement different design proposal for parts of the system. This kind of prototyping provides possibilities for formulating different proposals for design. Secondly, a prototype can be used to determine the relevance and adequacy of a specific design. The proposal is evaluated through experiments with a prototype that implements parts of the design. This kind of prototyping makes it possible to investigate the properties of a specific solution before it is implemented in full scale.

The experimental mode of operation raises both technical and organizational problems. Parnas argues that software can never be developed in a perfectly experimental way. Software systems are

highly complex, requirements to reliability and correctness are increasing, and many software systems involve concurrency and multiprocessing issues. These challenges all require an analytical mode of operation (Parnas 1985). Based on small-scale experiments, Boehm, Gray and Seewaldt compare the relative strengths and weaknesses of specifications and prototypes. They conclude that the use of prototypes leads to less coherent designs and to integration difficulties due to lack of interface specifications. In addition, the use of prototypes requires more testing and fixing, compared to software development based on specifications (Boehm *et al.* 1984). These points are supported by Brooks who argues that top-down development is necessary to establish and maintain conceptual integrity in software design (Brooks 1987).

An experimental mode of operation also raises organizational problems. It is hard to create opportunities for users to be constructively involved. Conflicts of interests among groups of users imply different demands to the next versions. It isn't easy to convince the customer that it is worthwhile experimenting with different designs. Iteration and increased involvement of users make management of the development process more complicated. And it isn't clear when an experiment should finish. Using prototypes as primary means of expression may create blindness towards other solutions, knowledge about the users' work is put in the background, and repeated extensions of a prototype may result in a poor and incomprehensible structuring of the program (Andersen *et al.* 1990; Stage 1989).

In summary, the *use of prototypes* have traditionally been linked to an *experimental mode of operation* in software design. Systems developers are advised to take advantage of learning through experiments. This approach to software design is a constructive response to some of the more serious problems of the analytical mode of operation based on specifications. But the strong emphasis on prototypes, learning, and involvement of users generate other problems. More analytical ways of developing and using prototypes are needed to handle these problems effectively.

3. Problem solving and bounded rationality

In the following we look closer at Simon's theory of human problem solving (Simon 1955, 1956, 1957, 1969, 1972, 1976, 1982) with the purpose of learning about fundamental conditions for effective soft-

ware design. We use Simon's more elaborate terminology and apply it to software design as a specific kind of human behavior.

Traditional theories of human behavior in organizations postulate that man is economic and rational. This ideal economic man (Simon 1955) is assumed to know relevant features of his environment. He is assumed to have clear goals and priorities, and to have the skills and resources to process and evaluate relevant information about the alternative courses of action that are available to him. Theories of rationality imply behavior that is appropriate to achieve given goals, within the limits imposed by given conditions and constraints (Simon 1972).

Throughout his work Simon has questioned this ideal. In real organizations rationality is bounded in several ways. Organizational actors face risks and uncertainties, they have incomplete information about alternative courses of action, and the complexity in processing and evaluating available information can be so great as to prevent the actor from selecting the best course of action (Simon 1972). Consequently, Simon has replaced the global rationality of the economic man with a kind of rational behavior based on the limited information and the limited computational capacities that are possessed by real organizational actors (Simon 1955).

Simon is concerned with bounded rationality—not to be confused with irrationality. This type of behavior is the result of the actor's intention to be rational and the limited conditions for being so. Simon is concerned with how we can approximate rational behavior in a world where we are often unable to predict the relevant future with accuracy. Facing uncertainty we must adopt a whole range of actions to reduce uncertainty, or at least to make outcomes less dependent on it. We must try to predict or forecast future developments, we must be able to learn from what actually happens, we must reduce the sensitivity of outcomes to the behavior of other actors, and we must enlarge the range of alternatives whenever the perceived alternatives involve high risks (Simon 1976). In short, we must take the world as it is. If we want to deal with uncertainty we have to understand how human beings behave in the face of uncertainty, and what limits of information and computability bind them.

Simon's alternative approach to human problem-solving is based on the principle of bounded rationality: "The capacity of the human mind for formulating and solving complex problems is very small compared with the size of the problems whose solution is re-

quired for objectively rational behavior in the real world—or even for a reasonable approximation to such objective rationality" (Simon 1957). The first consequence of this principle is that the intended rationality of an actor requires him to develop a simplified model of the real situation. He can then attempt to behave rationally with respect to this model, but such behavior is not even approximately rational with respect to the real world. The second consequence of this principle is that organizations—providing actors with mechanisms for coordination and division of labor—becomes necessary and useful instruments in dealing effectively with complex problems.

Based on the assumption of bounded rationality, heuristic search is the principal engine for human problem solving (Simon 1969). Simon argues that learning theories appear to account rather better for the observable behavior of organizational actors than do the traditional theories of rational behavior (Simon 1956). The actual learning behavior of a human problem-solver depends not only on the physiological and psychological characteristics of the actor, but equally upon the characteristics of the environment, representing among other things the needs, drives, and goals related to the organizational situation in question. Simon describes the actor's behavior metaphorically as a kind of maze-running, in which the actor learns by random search or search controlled by choices based on clues in the environment (Simon 1956).

A key difference in going from traditional rationality to bounded rationality is the substitution of optimizing behavior with satisficing behavior. An organizational actor that satisfies will not need a complete and consistent set of goals and priorities. Instead, he attempts to find a course of action that satisfies his needs (Simon 1957). Simon suggests that the actor can try to make approximate optimal decisions for an imaginary simplified world, but basically he tries to satisfice for a world approximating reality more closely (Simon 1969, 1972). It is difficult to draw a sharp distinction between approximate optimizing and satisficing behavior. Typically, it is possible to reinterpret the one in the frame of the other. In practice, however, the difference in emphasis that results from adopting one viewpoint or the other is often great (Simon 1972). In some situations, the actor starts out behaving in an approximate optimizing fashion operating in highly simplified spaces that abstract most of the detail from the real situation. When a schematized design of a solution has been elaborated, the details can be reintro-

duced and the design used as a guide in the search for a satisfactory design. In other situations, the overall design process may employ a satisficing search strategy, while optimizing techniques may be used to handle the details (Simon 1972).

Now, let us for a moment look at software design as a specific kind of human problem solving, and use Simon's theory to outline fundamental characteristics and principles related to effective software development. In doing so, we return to our conceptual framework (figure 1).

The actual behavior of software developers is bound by uncertainty and complexity. Even if they intend to operate in a rational and analytical mode—always trying to know why and how before actually doing—they can at most hope to achieve a kind of approximate rational behavior. Facing the real conditions of software design, the rational ideal of knowing before doing is less useful than the experimental ideal of learning. Learning seems to be the fundamental mode of operation; and learning theories account better for the observable behavior of software developers than the traditional theories of analytical behavior do. Software developers can at most hope to design satisfactory systems as a response to the demands and needs of the users. In doing so, they can try to make design decisions based on abstract specifications representing imaginary simplified worlds. But basically they have to rely on models representing a world approximating reality more closely.

4. Complexity and uncertainty

We have examined main traditions in software development and we have looked at Simon's general principles for human problem solving. We are now ready to review the relation between key features of approaches to software design and the complexity and uncertainty of the situation in question.

4.1. Focusing on means of expression

The problems that characterize an analytical mode of operation based on specifications have made some authors suggest that this approach is of little or no relevance to software design e.g. (Ehn 1988; Gladden 1982; Gould *et al.* 1985). To them, the issue is specifying or prototyping, and the answer is strongly in favor of prototyping. Most contemporary writers do, however, reflect a more

balanced understanding of the relative advantages and disadvantages of specifications and prototypes as basic means of expression. It is typically suggested that systems developers should apply a mixture of an analytical mode of operation based on specifications and an experimental mode of operation based on prototypes. Yet there are only few authors who provide a systematic framework for selecting and mixing these basically different approaches to software design. Most authors claim that a combination is possible; and they mention some advantages and disadvantages of each approach. However, they give very little advice on how to actually do it e.g. (Fairley 1985; Pressman 1987). In the following, we discuss some interesting exceptions.

The contingency approach has a long tradition in organization theory. In software development, Davis was one of the first to suggest that selection of the basic approach to determination of requirements to a computer system should be based on an analysis of the uncertainties that characterize the project (Davis 1982). Davis analyses uncertainty in terms of the following four factors: the organizational and technical context of the system, the future computer system, the experience and skills of the users, and the experience and skills of the systems developers. If uncertainty is low in terms of these factors, he suggests to base requirements determination on an informal approach or on analysis of existing systems. If uncertainty is high, he suggests the use of specifications or prototypes.

Burns and Dennis have extended and refined Davis' idea by introducing a distinction between complexity and uncertainty (Burns *et al.* 1985). Uncertainty is analyzed in terms of the following three factors: the degree of structuredness that characterizes the users' work, the degree of understanding the users have about their work, and the degree of experience and training of the systems developers. Complexity is analyzed in terms of the following four factors: project size, number of users, volume of new information, and complexity of new information. Figure 2 illustrates how Burns and Dennis suggest that the choice of basic approach in a software development project should be based on this analysis of complexity and uncertainty.

A basic problem in the approach suggested by Burns and Dennis is that complexity and uncertainty are analyzed only at the beginning of a project. They do not describe how the basic approach

Complexity		Uncertainty: *Low*	Uncertainty: *High*
	High	System Life Cycle	Mixed Methodology
	Low	Prototyping	Prototyping

Figure 2. Selection of mode of operation based on complexity and uncertainty (Burns et al. 1985).

of a project can be reanalyzed and changed if complexity and uncertainty increase or decrease during development. In this sense, they consider complexity and uncertainty as stable factors.

This problem is handled systematically in Boehm's spiral model (Boehm 1988). Each cycle of the spiral includes a re-evaluation of risks or uncertainties and subsequent development of a specification or prototype. The main emphasis is on resolving sources of uncertainty. In effect, the spiral model seems to be a constructive attempt to combine dynamically specifications and prototypes during a development effort.

The contingency approach and the spiral model give useful advice on the relevance of specifications and prototypes. But they focus exclusively on the means of expression as they only discuss whether one should apply specifications or prototypes in a specific situation. The mode of operation seems to be implied by this choice of a certain means of expression. In addition, neither the contingency approach nor the spiral model discusses how systems developers should handle the basic problems related to specifications and prototypes that were treated in section 2. None of these problems are handled by the contingency approach or the spiral model. The issue set is too simple.

4.2. Shifting to modes of operation

Let us summarize the argument. We have discussed effective approaches to software design by looking at complexity and uncer-

tainty as important characteristics of the situation in question. In doing so, we have assumed that modes of operation and means of expression are key features of approaches to software design.

The use of specifications has traditionally been linked to an analytical mode of operation. The strength of this approach is its narrow focus on reduction of complexity. But as a consequence of this focus, it suffers from some serious weaknesses by not questioning the availability and reliability of relevant information, by implying radical simplifications, and by restricting the ways in which organizational actors can communicate and learn about the system.

In contrast, prototypes and mock-ups have been linked to an experimental mode of operation. This approach is a constructive response to some of the more serious problems of the analytical mode of operation based on specifications. But due to the strong emphasis on prototypes, learning, and involvement of users, other problems arise.

Looking more broadly at software design as a specific kind of human problem-solving, we found theoretical support for what we believe to be the experience of most practitioners: an effective design effort should be based on variations or combinations of different modes of operation and means of expression. Yet the traditional views contradict this experience. They seem to be based on the fundamental assumption that complexity and uncertainty are independent (figure 2). It is, however, difficult to find any evidence supporting this viewpoint. On the contrary, it seems more reasonable to base effective software design on the assumption that complexity and uncertainty are intrinsically related, and there is no evidence that we can hope to reduce one of these without affecting the other. Acting analytically, we have to rely on an imaginary simplified world, thereby introducing new sources of uncertainty as to how well this view accords with the complex real world. Acting experimentally, we produce information as we go along, thereby introducing new sources of complexity. We have expressed this view in the The Principle of Limited Reduction, which we restate:

- Relying on an analytical mode of operation *to reduce complexity* introduces new sources of uncertainty requiring experimental countermeasures.

- Relying on an experimental mode of operation *to reduce uncertainty* introduces new sources of complexity requiring analytical countermeasures.

By now it should be clear that effective software design requires a systematic effort combining analytical and experimental modes of operation, regardless of whether specifications or prototypes are used. Software developers should adopt an experimental attitude towards the use of specifications as a supplement to the traditional analytical mode of operation. Similarly, they should adopt an analytical attitude towards the use of prototypes as a supplement to the traditional experimental mode of operation.

Tradition constrains us to focus on means of expression as the salient difference in approaches to software design. The Principle of Limited Reduction by contrast focuses on the relationship between the development situation itself and the method to be used. It deemphasizes the means of expression. It is assumed that applying either of the two basic means of expression requires a combination of analytical and experimental modes of operation. This perhaps surprising point can be illustrated by examples.

The analytical mode of operation based on specifications has been discussed at length (section 2.1). But an experimental mode of operation based on specifications is also widely recommended and used. In quality assurance, we are encouraged to plan for quality following the analytical ideals. On the other hand, we are advised to perform systematic evaluations of specifications and products through activities like structured walk-throughs, formal reviews, and tests. These experimental countermeasures are designed to offset the uncertainties introduced by the analytical mode of operation.

The proposal by Parnas and Clements (Parnas *et al.* 1986) is a more radical example of how to use specifications in an experimental mode of operation. As we have already seen, they argue that the analytical mode of operation is an idealization of software design. In practice, descriptions and specifications have to be developed, reviewed, extended, and modified experimentally. Parnas and Clements realize that software design is inherently experimental. Yet they hold to specifications as the primary means of expression through a post hoc analytical process, i.e. by making the documentation look as if the process had been analytical.

The experimental mode of operation based on prototypes has also been discussed (section 2.2). But an analytic use of prototypes is

equally relevant. In most software development projects, prototypes are in fact used analytically. They are used to clarify desirable features of the future system and to determine the relevance and adequacy of a specific design. The use of prototypes also requires careful management. It is advisable to clarify what specifically needs to be learned or discovered through the evaluation of a prototype. These considerations should be expressed in the project plan. Such analytical countermeasures will effectively reduce the uncertainties introduced by an experimental mode of operation.

5. Conclusion

We have proposed to focus on both complexity and uncertainty when facing a specific design task, and to take modes of operation and means of expression into account as different but related features of design approaches. Our major conclusion is stated in the Principle of Limited Reduction emphasizing the intrinsic relationship between complexity and uncertainty in software design.

Our conclusions are as yet hypothetical. We are conducting experiments on how systems developers handle complexity and uncertainty in specific design situations. In one of our ongoing experiments, we compare a mixed approach based on the spiral model (Boehm 1988) with earlier experiments contrasting the use of specifications with the use of prototypes (Boehm *et al.* 1984).

In conclusion, we caution that our discussion of approaches to design has been limited in its focus. Simon has been criticized for his narrow and instrumental view on problem-solving (Lanzara 1983; Schön 1983). Much of this criticism is relevant and valid. We have chosen to focus on the relation between knowing and learning in software design, without thoroughly taking the relation between understanding and change into account. We acknowledge that software design is an intervention. Our aim has been to look for common grounds for comparing and combining approaches to software design, and to add practicability to emerging paradigms in computer and information science.

Acknowledgments

The research behind this article has been partially financed by the Danish Natural Science Research Council, Program No. 11-8394. In writing the paper, we have received valuable comments from many

colleagues. Especially, we wish to thank Gro Bjerknes, Lars Bækgaard, Bo Dahlbom, Pentti Kerola, Heinz K. Klein, Pasi Kuvaja, Peter A. Nielsen, David L. Parnas, Carsten Sørensen, Kari Thoresen, Heinz Züllighoven, Ivan Aaen, and David Novick.

References

Andersen, N. E., F. Kensing, M. Lassen, J. Lundin, L. Mathiassen, A. Munk-Madsen & P. Sørgaard (1990): *Professional Systems Development. Experience, Ideas, and Action.* Englewood Cliffs, New Jersey: Prentice-Hall.

Basili, V. R. & A. J. Turner (1975): Iterative Enhancement: A Practical Technique for Software development. *IEEE Transactions on Software Engineering.* Vol. 1, No. 4 (390–396).

Boehm, B. W. (1988): A Spiral Model of Software Development and Enhancement. *Computer.* Vol. 21, No. 5.

Boehm, B. W., T. E. Gray & T. Seewaldt (1984): Prototyping Versus Specifying: A Multiproject Experiment. *IEEE Transactions on Software Engineering.* Vol. 10. No. 3 (290–303), May.

Brooks, F. P. (1987): No Silver Bullet. Essence and Accidents of Software Engineering. *Computer.* Vol. 20, No. 4 (10–19).

Burns, R. N. & A. R. Dennis (1985): Selecting the Appropriate Application Development Methodology. *Data Base.* (19–23), Fall.

Davis, G. B. (1982): Strategies for Information Requirement Determination. *IBM Systems Journal.* Vol. 21, No. 1 (4–30).

Davis, A. M., E. H. Bersoff & E. R. Comer (1988): A Strategy for Comparing Alternative Software Development Life Cycle Models. *IEEE Transactions on Software Engineering.* Vol. 14, No. 10 (1453–1461).

Dijkstra, E. (1972): Notes on Structured Programming. (1–82) in *Structured Programming.* London: Academic Press.

Ehn, P. (1988): *Work-Oriented Design of Computer Artifacts.* Stockholm: Arbetslivscentrum.

Fairley, R. (1985): *Software Engineering Concepts.* McGraw-Hill.

Floyd, C. (1984): A Systematic Look at Prototyping. (1–17) in R. Budde *et al.* (Eds.): *Approaches to Prototyping.* Berlin: Springer-Verlag.

Floyd, C. (1987): Outline of a Paradigm Change in Software Engineering. (191–210) in G. Bjerknes *et al.* (Eds.): *Computers and Democracy.* Avebury: Aldershot.

Gladden, G. R. (1982): Stop the Life-Cycle, I Want to Get Off. *Software Engineering Notes.* Vol. 7, No. 2 (35–39).

Gomaa, H. & D. B. H. Scott (1981): Prototyping as a Tool in the Specification of User Requirements. (333–342) in *Proceedings. 5th IEEE International Conference Software Engineering.*

Gould, J. D. & C. Lewis (1985): Designing for Usability: Key Principles and What Designers Think. *Communications of the ACM.* Vol. 28, No. 3 (300–311).

Langefors, B. (1966): *Theoretical Analysis of Information Systems.* Lund: Studentlitteratur.

Lanzara, G. F. (1983): The Design Process: Frames, Metaphores, and Games. In U. Briefs *et al.* (Eds.): *Systems Design For, With and By the Users.* Amsterdam: North-Holland.

McCracken, D. D. & M. A. Jackson (1982): Life Cycle Concept Considered Harmful. *Software Engineering Notes.* Vol. 7, No. 2 (29–32).

Parnas, D. L. (1985): Software Aspects of Strategic Defense Systems. *Communications of the ACM.* Vol. 28, No. 12 (1326–1335).

Parnas, D. L. & P. C. Clements (1986): A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering.* Vol. 12, No. 2 (251–257).

Pressman, R. S. (1987): *Software Engineering. A Practitioner's Approach.* Maidenhead, Berkshire: McGraw-Hill. Second edition.

Royce, W. W. (1970): Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings WESCON.* Aug.

Schön, D. (1983): *The Reflective Practitioner—How Professionals Think in Action.* New York: Basic Books.

Simon, H. (1955): A Behavioral Model of Rational Choice. *Quarterly Journal of Economics.* Vol. 69, No. 1 (99–118).

Simon, H. (1956): Rational Choice and the Structure of the Environment. *Psychological Review.* Vol. 63, No. 2 (129–138).

Simon, H. (1957): *Models of Man: Social and Rational.* New York: John Wiley and Sons.

Simon, H. (1969): *The Science of the Artificial.* Cambridge, Massachusetts: MIT Press.

Simon, H. (1972): Theories of Bounded Rationality. (161–176) in C. B. Radner and R. Radner (Eds.): *Decision and Organization.* Amsterdam: North-Holland.

Simon, H. (1976): From Substantive to Procedural Rationality. (129–148) in S. J. Latsis (Ed.): *Method and Appraisal in Economics.* Cambridge: Cambridge University Press.

Simon, H. (1982): *Models of Bounded Rationality: Behavioral Economics and Business Organization.* Cambridge, Massachusetts: MIT Press.

Stage, J. (1989): *Between Tradition and Transcendence. Analysis and Design in Systems Development.* Ph.D. thesis, Institute for Electronic Systems, Aalborg University. (In Danish)

Wirth, N. (1973): *Systematic Programming. An Introduction.* Englewood Cliffs, New Jersey: Prentice-Hall.

Wulf, W. (1977): Languages and Structured Programs. In R. T. Yeh (Ed.): *Current Trends in Programming Methodology.* New Jersey: Prentice-Hall.

Yourdon, E. (1982): *Managing the System Life Cycle.* New York: Yourdon Inc.