APPENDIX B

Naur, Ehn, Musashi

Peter Naur and Pelle Ehn wrote the two most compelling and accurate accounts of software development I have yet seen. Neither is as well known as it needs to be, and Ehn's book is out of print. I am happy, therefore, to present extracts from their articles, for wider readership.

Peter Naur's "Programming as Theory Building" neatly describes the mental activity of creating software and explains the "metaphor building" activity in Extreme Programming (XP).

Pelle Ehn wrote the wonderful book *Work-Oriented Design of Software Artifacts*, in which he considers how Wittgenstein's idea of language games informs our view of software development.

Miyamoto Musashi, the 17th-century samurai champion, never wrote software. The competing schools of sword fighting in his day sound painfully like today's schools of methodology. He admonishes people to avoid getting infatuated with tools and schools, to use different tools and strokes for different moments, and to just "cut off the opponent's arm." His admonitions apply directly to software development—if you realize that the *opponent* is the problem, not your office mate.

Naur, Ehn, Musashi

Peter Naur, Programming as Theory Building

Peter Naur, widely known as one of the authors of the programming language syntax notation "Backus-Naur Form" (BNF), wrote "Programming as Theory Building" in 1985. It was reprinted in his collection of works, *Computing: A Human Activity* (Naur 1992).

This article is, to my mind, the most accurate account of what goes on in designing and coding a program. I refer to it regularly when discussing how much documentation to create, how to pass along tacit knowledge, and the value of the XP's metaphor-setting exercise. It also provides a way to examine a methodology's economic structure.

In the article, which follows, note that the quality of the designing programmer's work is related to the quality of the match between his theory of the problem and his theory of the solution. Note that the quality of a later programmer's work is related to the match between his theories and the previous programmer's theories.

Using Naur's ideas, the designer's job is not to pass along "the design" but to pass along "the theories" driving the design. The latter goal is more useful and more appropriate. It also highlights that knowledge of the theory is tacit in the owning, and so passing along the theory requires passing along both explicit and tacit knowledge.

Here is Peter Naur's way of saying it.

"Programming as Theory Building"

Introduction

The present discussion is a contribution to the understanding of what programming is. It suggests that programming properly should be regarded as an activity by which the programmers form or achieve a certain kind of insight, a theory, of the matters at hand. This suggestion is in contrast to what appears to be a more common notion, that programming should be regarded as a production of a program and certain other texts.

Some of the background of the views presented here is to be found in certain observations of what actually happens to programs and the teams of programmers dealing with them, particularly in situations arising from unexpected and perhaps erroneous program executions or reactions, and on the occasion of modifications of programs. The difficulty of accommodating such observations in a production view of programming suggests that this view is misleading. The theory building view is presented as an alternative.

A more general background of the presentation is a conviction that it is important to have an appropriate understanding of what programming is. If our understanding is inappropriate we will misunderstand the difficulties that arise in the activity and our attempts to overcome them will give rise to conflicts and frustrations.

In the present discussion some of the crucial background experience will first be outlined. This is followed by an explanation of a theory of what programming is, denoted the Theory Building View. The subsequent sections enter into some of the consequences of the Theory Building View.

Programming and the Programmers' Knowledge

I shall use the word *programming* to denote the whole activity of design and implementation of programmed solutions. What I am concerned with is the activity of matching some significant part and aspect of an activity in the real world to the formal symbol manipulation that can be done by a program running on a computer. With such a notion it follows directly that the programming activity I am talking about must include the development in time corresponding to the changes taking place in the real world activity being matched by the program execution, in other words program modifications.

One way of stating the main point I want to make is that programming in this sense primarily must be the programmers' building up knowledge of a certain kind, knowledge taken to be basically the programmers' immediate possession, any documentation being an auxiliary, secondary product.

As a background of the further elaboration of this view given in the following sections, the remainder of the present section will describe some real experience of dealing with large programs that has seemed to me more and more significant as I have pondered over the problems. In either case the experience is my own or has been communicated to me by persons having firsthand contact with the activity in question.

Case 1 concerns a compiler. It has been developed by a group A for a Language L and worked very well on computer X. Now another group B has the task to write a compiler for a language L + M, a modest extension of L, for computer Y. Group B decides that the compiler for L developed by group A will be a good starting point for their design, and get a contract with group A that they will get support in the form of full documentation, including annotated program texts and much additional written design discussion, and also personal advice. The arrangement was effective and group B managed to develop the compiler they wanted. In the present context the significant issue is the importance of the personal advice from group A in the matters that concerned how to implement the extensions M to the language. During the design phase group B made suggestions for the manner in which the extensions should be accommodated and submitted them to group A for review. In several major cases it turned out that the solutions suggested by group B were found by group A to make no use of the facilities that were not only inherent in the structure of the existing compiler but were discussed at length in its documentation, and to be based instead on additions to that structure in the form of patches that effectively destroyed its power and simplicity. The members of group A were able to spot these cases instantly and could propose simple and effective solutions, framed entirely within the existing

structure. This is an example of how the full program text and additional documentation is insufficient in conveying to even the highly motivated group B the deeper insight into the design, that theory which is immediately present to the members of group A.

In the years following these events the compiler developed by group B was taken over by other programmers of the same organization, without guidance from group A. Information obtained by a member of group A about the compiler resulting from the further modification of it after about 10 years made it clear that at that later stage the original powerful structure was still visible, but made entirely ineffective by amorphous additions of many different kinds. Thus, again, the program text and its documentation has proved insufficient as a carrier of some of the most important design ideas.

Case 2 concerns the installation and fault diagnosis of a large real-time system for monitoring industrial production activities. The system is marketed by its producer, each delivery of the system being adapted individually to its specific environment of sensors and display devices. The size of the program delivered in each installation is of the order of 200,000 lines. The relevant experience from the way this kind of system is handled concerns the role and manner of work of the group of installation and fault finding programmers. The facts are, first that these programmers have been closely concerned with the system as a full time occupation over a period of several years, from the time the system was under design. Second, when diagnosing a fault these programmers rely almost exclusively on their ready knowledge of the system and the annotated program text, and are unable to conceive of any kind of additional documentation that would be useful to them. Third, other programmers' groups who are responsible for the operation of particular installations of the system, and thus receive documentation of the system and full guidance on its use from the producer's staff, regularly encounter difficulties that upon consultation with the producer's installation and fault finding programmer are traced to inadequate understanding of the existing documentation, but which can be cleared up easily by the installation and fault finding programmers.

The conclusion seems inescapable that at least with certain kinds of large programs, the continued adaptation, modification, and correction of errors in them, is essentially dependent on a certain kind of knowledge possessed by a group of programmers who are closely and continuously connected with them.

Ryle's Notion of Theory

If it is granted that programming must involve, as the essential part, a building up of the programmers' knowledge, the next issue is to characterize that knowledge more closely. What will be considered here is the suggestion that the programmers' knowledge properly should be regarded as a theory, in the sense of Ryle [1949]. Very briefly, a person who has or possesses a theory in this sense knows how to do certain things and in addition can support the actual doing with explanations, justifications, and answers to queries, about the

activity of concern. It may be noted that Ryle's notion of theory appears as an example of what K. Popper [Popper, and Eccles, 1977] calls unembodied World 3 objects and thus has a defensible philosophical standing. In the present section we shall describe Ryle's notion of theory in more detail.

Ryle [1949] develops his notion of theory as part of his analysis of the nature of intellectual activity, particularly the manner in which intellectual activity differs from, and goes beyond, activity that is merely intelligent. In intelligent behaviour the person displays, not any particular knowledge of facts, but the ability to do certain things, such as to make and appreciate jokes, to talk grammatically, or to fish. More particularly, the intelligent performance is characterized in part by the person's doing them well, according to certain criteria, but further displays the person's ability to apply the criteria so as to detect and correct lapses, to learn from the examples of others, and so forth. It may be noted that this notion of intelligence does not rely on any notion that the intelligent behaviour depends on the person's following or adhering to rules, prescriptions, or methods. On the contrary, the very act of adhering to rules can be done more or less intelligently; if the exercise of intelligence depended on following rules there would have to be rules about how to follow rules, and about how to follow the rules about following rules, etc., in an infinite regress, which is absurd.

What characterizes intellectual activity, over and beyond activity that is merely intelligent, is the person's building and having a theory, where theory is understood as the knowledge a person must have in order not only to do certain things intelligently but also to explain them, to answer queries about them, to argue about them, and so forth. A person who has a theory is prepared to enter into such activities; while building the theory the person is trying to get it.

The notion of theory in the sense used here applies not only to the elaborate constructions of specialized fields of enquiry, but equally to activities that any person who has received education will participate in on certain occasions. Even quite unambitious activities of everyday life may give rise to people's theorizing, for example in planning how to place furniture or how to get to some place by means of certain means of transportation.

The notion of theory employed here is explicitly *not* confined to what may be called the most general or abstract part of the insight. For example, to have Newton's theory of mechanics as understood here it is not enough to understand the central laws, such as that force equals mass times acceleration. In addition, as described in more detail by Kuhn [1970, p. 187ff], the person having the theory must have an understanding of the manner in which the central laws apply to certain aspects of reality, so as to be able to recognize and apply the theory to other similar aspects. A person having Newton's theory of mechanics must thus understand how it applies to the motions of pendulums and the planets, and must be able to recognize similar phenomena in the world, so as to be able to employ the mathematically expressed rules of the theory properly.

The dependence of a theory on a grasp of certain kinds of similarity between situations and events of the real world gives the reason why the knowledge held by someone who has the theory could not, in principle, be expressed in terms of rules. In fact, the similarities in question are not, and cannot be, expressed in terms of criteria, no more than the similarities of many other kinds of objects, such as human faces, tunes, or tastes of wine, can be thus expressed.

The Theory to Be Built by the Programmer

In terms of Ryle's notion of theory, what has to be built by the programmer is a theory of how certain affairs of the world will be handled by, or supported by, a computer program. On the Theory Building View of programming the theory built by the programmers has primacy over such other products as program texts, user documentation, and additional documentation such as specifications.

In arguing for the Theory Building View, the basic issue is to show how the knowledge possessed by the programmer by virtue of his or her having the theory necessarily, and in an essential manner, transcends that which is recorded in the documented products. The answers to this issue is that the programmer's knowledge transcends that given in documentation in at least three essential areas:

1) The programmer having the theory of the program can explain how the solution relates to the affairs of the world that it helps to handle. Such an explanation will have to be concerned with the manner in which the affairs of the world, both in their overall characteristics and their details, are, in some sense, mapped into the program text and into any additional documentation. Thus the programmer must be able to explain, for each part of the program text and for each of its overall structural characteristics, what aspect or activity of the world is matched by it. Conversely, for any aspect or activity of the world the programmer is able to state its manner of mapping into the program text. By far the largest part of the world aspects and activities will of course lie outside the scope of the program text, being irrelevant in the context. However, the decision that a part of the world *is* relevant can only be made by someone who understands the whole world. This understanding must be contributed by the programmer.

2) The programmer having the theory of the program can explain *why* each part of the program is what it is, in other words is able to support the actual program text with a justification of some sort. The final basis of the justification is and must always remain the programmer's direct, intuitive knowledge or estimate. This holds even where the justification makes use of reasoning, perhaps with application of design rules, quantitative estimates, comparisons with alternatives, and such like, the point being that the choice of the principles and rules, and the decision that they are relevant to the situation at hand, again must in the final analysis remain a matter of the programmer's direct knowledge.

3) The programmer having the theory of the program is able to respond constructively to any demand for a modification of

the program so as to support the affairs of the world in a new manner. Designing how a modification is best incorporated into an established program depends on the perception of the similarity of the new demand with the operational facilities already built into the program. The kind of similarity that has to be perceived is one between aspects of the world. It only makes sense to the agent who has knowledge of the world, that is to the programmer, and cannot be reduced to any limited set of criteria or rules, for reasons similar to the ones given above why the justification of the program cannot be thus reduced.

While the discussion of the present section presents some basic arguments for adopting the Theory Building View of programming, an assessment of the view should take into account to what extent it may contribute to a coherent understanding of programming and its problems. Such matters will be discussed in the following sections.

Problems and Costs of Program Modifications

A prominent reason for proposing the Theory Building View of programming is the desire to establish an insight into programming suitable for supporting a sound understanding of program modifications. This question will therefore be the first one to be taken up for analysis.

One thing seems to be agreed by everyone, that software will be modified. It is invariably the case that a program, once in operation, will be felt to be only part of the answer to the problems at hand. Also the very use of the program itself will inspire ideas for further useful services that the program ought to provide. Hence the need for ways to handle modifications.

The question of program modifications is closely tied to that of programming costs. In the face of a need for a changed manner of operation of the program, one hopes to achieve a saving of costs by making modifications of an existing program text, rather than by writing an entirely new program.

The expectation that program modifications at low cost ought to be possible is one that calls for closer analysis. First it should be noted that such an expectation cannot be supported by analogy with modifications of other complicated manmade constructions. Where modifications are occasionally put into action, for example in the case of buildings, they are well known to be expensive and in fact complete demolition of the existing building followed by new construction is often found to be preferable economically. Second, the expectation of the possibility of low cost program modifications conceivably finds support in the fact that a program is a text held in a medium allowing for easy editing. For this support to be valid it must clearly be assumed that the dominating cost is one of text manipulation. This would agree with a notion of programming as text production. On the Theory Building View this whole argument is false. This view gives no support to an expectation that program modifications at low cost are generally possible.

A further closely related issue is that of program flexibility. In including flexibility in a program we build into the program certain operational facilities that are not immediately demanded, but which are

likely to turn out to be useful. Thus a flexible program is able to handle certain classes of changes of external circumstances without being modified.

It is often stated that programs should be designed to include a lot of flexibility, so as to be readily adaptable to changing circumstances. Such advice may be reasonable as far as flexibility that can be easily achieved is concerned. However, flexibility can in general only be achieved at a substantial cost. Each item of it has to be designed, including what circumstances it has to cover and by what kind of parameters it should be controlled. Then it has to be implemented, tested, and described. This cost is incurred in achieving a program feature whose usefulness depends entirely on future events. It must be obvious that built-in program flexibility is no answer to the general demand for adapting programs to the changing circumstances of the world.

In a program modification an existing programmed solution has to be changed so as to cater for a change in the real world activity it has to match. What is needed in a modification, first of all, is a confrontation of the existing solution with the demands called for by the desired modification. In this confrontation the degree and kind of similarity between the capabilities of the existing solution and the new demands has to be determined. This need for a determination of similarity brings out the merit of the Theory Building View. Indeed, precisely in a determination of similarity the shortcoming of any view of programming that ignores the central requirement for the direct participation of persons who possess the appropriate insight becomes evident. The point is that the kind of similarity that has to be recognized is accessible to the human beings who possess the theory of the program, although entirely outside the reach of what can be determined by rules, since even the criteria on which to judge it cannot be formulated. From the insight into the similarity between the new requirements and those already satisfied by the program, the programmer is able to design the change of the program text needed to implement the modification.

In a certain sense there can be no question of a theory modification, only of a program modification. Indeed, a person having the theory must already be prepared to respond to the kinds of questions and demands that may give rise to program modifications. This observation leads to the important conclusion that the problems of program modification arise from acting on the assumption that programming consists of program text production, instead of recognizing programming as an activity of theory building.

On the basis of the Theory Building View the decay of a program text as a result of modifications made by programmers without a proper grasp of the underlying theory becomes understandable. As a matter of fact, if viewed merely as a change of the program text and of the external behaviour of the execution, a given desired modification may usually be realized in many different ways, all correct. At the same time, if viewed in relation to the theory of the program these ways may look very different, some of them perhaps conforming to that theory or extending it in a natural way, while

others may be wholly inconsistent with that theory, perhaps having the character of unintegrated patches on the main part of the program. This difference of character of various changes is one that can only make sense to the programmer who possesses the theory of the program. At the same time the character of changes made in a program text is vital to the longer term viability of the program. For a program to retain its quality it is mandatory that each modification is firmly grounded in the theory of it. Indeed, the very notion of qualities such as simplicity and good structure can only be understood in terms of the theory of the program, since they characterize the actual program text in relation to such program texts that might have been written to achieve the same execution behaviour, but which exist only as possibilities in the programmer's understanding.

Program Life, Death, and Revival

A main claim of the Theory Building View of programming is that an essential part of any program, the theory of it, is something that could not conceivably be expressed, but is inextricably bound to human beings. It follows that in describing the state of the program it is important to indicate the extent to which programmers having its theory remain in charge of it. As a way in which to emphasize this circumstance one might extend the notion of program building by notions of program life, death, and revival. The building of the program is the same as the building of the theory of it by and in the team of programmers. During the program life a programmer team possessing its theory remains in active control of the program, and in particular retains control over all modifications. The death of a program happens when the programmer team possessing its theory is dissolved. A dead program may continue to be used for execution in a computer and to produce useful results. The actual state of death becomes visible when demands for modifications of the program cannot be intelligently answered. Revival of a program is the rebuilding of its theory by a new programmer team.

The extended life of a program according to these notions depends on the taking over by new generations of programmers of the theory of the program. For a new programmer to come to possess an existing theory of a program it is insufficient that he or she has the opportunity to become familiar with the program text and other documentation. What is required is that the new programmer has the opportunity to work in close contact with the programmers who already possess the theory, so as to be able to become familiar with the place of the program in the wider context of the relevant real world situations and so as to acquire the knowledge of how the program works and how unusual program reactions and program modifications are handled within the program theory. This problem of education of new programmers in an existing theory of a program is quite similar to that of the educational problem of other activities where the knowledge of how to do certain things dominates over the knowledge that certain things are the case, such as writing and playing a music instrument. The most important educational activity is the student's doing the

relevant things under suitable supervision and guidance. In the case of programming the activity should include discussions of the relation between the program and the relevant aspects and activities of the real world, and of the limits set on the real world matters dealt with by the program.

A very important consequence of the Theory Building View is that program revival, that is reestablishing the theory of a program merely from the documentation, is strictly impossible. Lest this consequence may seem unreasonable it may be noted that the need for revival of an entirely dead program probably will rarely arise, since it is hardly conceivable that the revival would be assigned to new programmers without at least some knowledge of the theory had by the original team. Even so the Theory Building View suggests strongly that program revival should only be attempted in exceptional situations and with full awareness that it is at best costly, and may lead to a revived theory that differs from the one originally had by the program authors and so may contain discrepancies with the program text.

In preference to program revival, the Theory Building View suggests, the existing program text should be discarded and the new-formed programmer team should be given the opportunity to solve the given problem afresh. Such a procedure is more likely to produce a viable program than program revival, and at no higher, and possibly lower, cost. The point is that building a theory to fit and support an existing program text is a difficult, frustrating, and time consuming activity. The new programmer is likely to feel torn between loyalty to the existing program text, with whatever obscurities and weaknesses it may contain, and the new theory that he or she has to build up, and which, for better or worse, most likely will differ from the original theory behind the program text.

Similar problems are likely to arise even when a program is kept continuously alive by an evolving team of programmers, as a result of the differences of competence and background experience of the individual programmers, particularly as the team is being kept operational by inevitable replacements of the individual members.

Method and Theory Building

Recent years [have] seen much interest in programming methods. In the present section some comments will be made on the relation between the Theory Building View and the notions behind programming methods.

To begin with, what is a programming method? This is not always made clear, even by authors who recommend a particular method. Here a programming method will be taken to be a set of work rules for programmers, telling what kind of things the programmers should do, in what order, which notations or languages to use, and what kinds of documents to produce at various stages.

In comparing this notion of method with the Theory Building View of programming, the most important issue is that of actions or operations and their ordering. A method implies a claim that program development can and should

proceed as a sequence of actions of certain kinds, each action leading to a particular kind of documented result. In building the theory there can be no particular sequence of actions, for the reason that a theory held by a person has no inherent division into parts and no inherent ordering. Rather, the person possessing a theory will be able to produce presentations of various sorts on the basis of it, in response to questions or demands.

As to the use of particular kinds of notation or formalization, again this can only be a secondary issue since the primary item, the theory, is not, and cannot be, expressed, and so no question of the form of its expression arises.

It follows that on the Theory Building View, for the primary activity of the programming there can be no right method.

This conclusion may seem to conflict with established opinion, in several ways, and might thus be taken to be an argument against the Theory Building View. Two such apparent contradictions shall be taken up here, the first relating to the importance of method in the pursuit of science, the second concerning the success of methods as actually used in software development.

The first argument is that software development should be based on scientific manners, and so should employ procedures similar to scientific methods. The flaw of this argument is the assumption that there is such a thing as scientific method and that it is helpful to scientists. This question has been the subject of much debate in recent years, and the conclusion of such authors as Feyerabend [1978], taking his illustrations from the history of physics, and Medawar [1982], arguing as a biologist, is that the notion of scientific method as a set of guidelines for the practising scientist is mistaken.

This conclusion is not contradicted by such work as that of Polya [1954, 1957] on problem solving. This work takes its illustrations from the field of mathematics and leads to insight which is also highly relevant to programming. However, it cannot be claimed to present a method on which to proceed. Rather, it is a collection of suggestions aiming at stimulating the mental activity of the problem solver, by pointing out different modes of work that may be applied in any sequence.

The second argument that may seem to contradict the dismissal of method of the Theory Building View is that the use of particular methods has been successful, according to published reports. To this argument it may be answered that a methodically satisfactory study of the efficacy of programming methods so far never seems to have been made. Such a study would have to employ the well established technique of controlled experiments (cf. [Brooks, 1980] or [Moher and Schneider, 1982]). The lack of such studies is explainable partly by the high cost that would undoubtedly be incurred in such investigations if the results were to be significant, partly by the problems of establishing in an operational fashion the concepts underlying what is called methods in the field of program development. Most published reports on such methods merely describe and recommend certain techniques and procedures, without establishing their usefulness or efficacy in any systematic way. An elaborate

study of five different methods by C. Floyd and several co-workers [Floyd, 1984] concludes that the notion of methods as systems of rules that in an arbitrary context and mechanically will lead to good solutions is an illusion. What remains is the effect of methods in the education of programmers. This conclusion is entirely compatible with the Theory Building View of programming. Indeed, on this view the quality of the theory built by the programmer will depend to a large extent on the programmer's familiarity with model solutions of typical problems, with techniques of description and verification, and with principles of structuring systems consisting of many parts in complicated interactions. Thus many of the items of concern of methods are relevant to theory building. Where the Theory Building View departs from that of the methodologists is on the question of which techniques to use and in what order. On the Theory Building View this must remain entirely a matter for the programmer to decide, taking into account the actual problem to be solved.

Programmers' Status and the Theory Building View

The areas where the consequences of the Theory Building View contrast most strikingly with those of the more prevalent current views are those of the programmers' personal contribution to the activity and of the programmers' proper status.

The contrast between the Theory Building View and the more prevalent view of the programmers' personal contribution is apparent in much of the common discussion of programming. As just one example, consider the study of modifiability of large software systems by Oskarsson [1982]. This study gives extensive information on a considerable number of modifications in one release of a large commercial system. The description covers the background, substance, and implementation, of each modification, with particular attention to the manner in which the program changes are confined to particular program modules. However, there is no suggestion whatsoever that the implementation of the modifications might depend on the background of the 500 programmers employed on the project, such as the length of time they have been working on it, and there is no indication of the manner in which the design decisions are distributed among the 500 programmers. Even so the significance of an underlying theory is admitted indirectly in statements such as that 'decisions were implemented in the wrong block' and in a reference to 'a philosophy of AXE.' However, by the manner in which the study is conducted these admissions can only remain isolated indications.

More generally, much current discussion of programming seems to assume that programming is similar to industrial production, the programmer being regarded as a component of that production, a component that has to be controlled by rules of procedure and which can be replaced easily. Another related view is that human beings perform best if they act like machines, by following rules, with a consequent stress on formal modes of expression, which make it possible to formulate certain arguments in terms of rules of formal manipulation. Such views agree well

with the notion, seemingly common among persons working with computers, that the human mind works like a computer. At the level of industrial management these views support treating programmers as workers of fairly low responsibility, and only brief education.

On the Theory Building View the primary result of the programming activity is the theory held by the programmers. Since this theory by its very nature is part of the mental possession of each programmer, it follows that the notion of the programmer as an easily replaceable component in the program production activity has to be abandoned. Instead the programmer must be regarded as a responsible developer and manager of the activity in which the computer is a part. In order to fill this position he or she must be given a permanent position, of a status similar to that of other professionals, such as engineers and lawyers, whose active contributions as employers of enterprises rest on their intellectual proficiency.

The raising of the status of programmers suggested by the Theory Building View will have to be supported by a corresponding reorientation of the programmer education. While skills such as the mastery of notations, data representations, and data processes, remain important, the primary emphasis would have to turn in the direction of furthering the understanding and talent for theory formation. To what extent this can be taught at all must remain an open question. The most hopeful approach would be to have the student work on concrete problems under guidance, in an active and constructive environment.

Conclusions

Accepting program modifications demanded by changing external circumstances to be an essential part of programming, it is argued that the primary aim of programming is to have the programmers build a theory of the way the matters at hand may be supported by the execution of a program. Such a view leads to a notion of program life that depends on the continued support of the program by programmers having its theory. Further, on this view the notion of a programming method, understood as a set of rules of procedure to be followed by the programmer, is based on invalid assumptions and so has to be rejected. As further consequences of the view, programmers have to be accorded the status of responsible, permanent developers and managers of the activity of which the computer is a part, and their education has to emphasize the exercise of theory building, side by side with the acquisition of knowledge of data processing and notations.

References

Brooks, R. E. Studying programmer behaviour experimentally. *Comm. ACM* 23(4): 207–213, 1980.

Feyerabend, P. *Against Method*. London, Verso Editions, 1978; ISBN: 86091-700-2.

Floyd, C. Eine Untersuchung von Software-Entwicklungs-Methoden. Pp. 248–274 in *Programmierumgebungen und Compiler*, ed H. Morgenbrod and W. Sammer, Tagung I/1984 des German Chapter of the ACM, Stuttgart, Teubner Verlag, 1984; ISBN: 3-519-02437-3.

Kuhn, T. S. *The Structure of Scientific Revolutions*, Second Edition. Chicago,

University of Chicago Press, 1970; ISBN: 0-226-45803-2.
Medawar, P. *Pluto's Republic*. Oxford, University Press, 1982: ISBN: 0-19-217726-5.
Moher, T., and Schneider, G. M. Methodology and experimental research in software engineering, *Int. J. Man-Mach. Stud.* 16: 65-87, 1. Jan. 1982.
Oskarsson, Ö Mechanisms of modifiability in large software systems *Linköping Studies in Science and Technology, Dissertations, no. 77*, Linköping, 1982; ISBN: 91-7372-527-7.
Polya, G. *How To Solve It* . New York, Doubleday Anchor Book, 1957.
Polya, G. *Mathematics and Plausible Reasoning*. New Jersey, Princeton University Press, 1954.
Popper, K. R., and Eccles, J. C. *The Self and Its Brain*. London, Routledge and Kegan Paul, 1977.
Ryle, G. The *Concept of Mind*. Harmondsworth, England, Penguin, 1963, first published 1949.

Applying "Theory Building"

Viewing programming as theory building helps us understand "metaphor building" activity in Extreme Programming (XP), and the respective roles of tacit knowledge and documentation in passing along design knowledge.

The Metaphor as a Theory

Kent Beck suggested that it is useful to a design team to simplify the general design of a program to match a single metaphor. Examples might be, "This program really looks like an assembly line, with things getting added to a chassis along the line," or "This program really looks like a restaurant, with waiters and menus, cooks and cashiers."

If the metaphor is good, the many associations the designers create around the metaphor turn out to be appropriate to their programming situation.

That is exactly Naur's idea of passing along a theory of the design.

If "assembly line" is an appropriate metaphor, then later programmers, considering what they know about assembly lines, will make guesses about the structure of the software at hand and find that their guesses are "close." That is an extraordinary power for just the two words, "assembly line."

The value of a good metaphor increases with the number of designers. The closer each person's guess is "close" to the other people's guesses, the greater the resulting consistency in the final system design.

Imagine 10 programmers working as fast as they can, in parallel, each making design decisions and adding classes as she goes. Each will necessarily develop her own theory as she goes. As each adds code, the theory that binds their work becomes less and less coherent, more and more complicated. Not only maintenance gets harder, but their own work gets harder. The design easily becomes a "kludge." If they have a common theory, on the other hand, they add code in ways that fit together.

An appropriate, shared metaphor lets a person guess accurately where someone else on the team just added code, and how to fit her new piece in with it.

Tacit Knowledge and Documentation

The documentation is almost certainly behind the current state of the program, but people are good at looking around. What should you put into the documentation?

That which helps the next programmer build an adequate theory of the program.

This is enormously important. The purpose of the documentation is to jog memories in the reader, set up relevant pathways of thought about experiences and metaphors.

This sort of documentation is more stable over the life of the program than just naming the pieces of the system currently in place.

The designers are allowed to use whatever forms of expression are necessary to set up those relevant pathways. They can even use multiple metaphors, if they don't find one that is adequate for the entire program. They might say that one section implements a fractal compression algorithm, a second is like an accounting ledger, the user interface follows the model-observer design pattern, and so on.

Experienced designers often start their documentation with just

- The metaphors
- Text describing the purpose of each major component
- Drawings of the major interactions between the major components

These three items alone take the next team a long way to constructing a useful theory of the design.

The source code itself serves to communicate a theory to the next programmer. Simple, consistent naming conventions help the next person build a coherent theory. When people talk about "clean code," a large part of what they are referring to is how easily the reader can build a coherent theory of the system.

Documentation cannot—and so need not—say everything. Its purpose is to help the next programmer build an accurate theory about the system.

Pelle Ehn, Wittgenstein's Language Games

In *Work-Oriented Development of Software Artifacts* (Ehn 1988), Pelle Ehn describes a series of projects that explored ways of making software more appropriate to its final use, easier to use, and made by both programmers and end users.

The high point of the book for me is the way in which he considers software development in the context of four philosophers: Descartes, Marx, Heidegger, and Wittgenstein.

A person working in the style of Descartes thinks of an external reality worth describing and turns her efforts toward capturing that reality. She is therefore interested in the match to reality of the requirements, models, and code. This Cartesian approach filled our field's first half-century.

A person working in the style of Marx first asks, "Whom does this new system benefit? How does its deployment change the social power structure?" This is a meaningful question to consider, whether you like Marx's political theories or not.

A person working in the style of Heidegger considers the efficacy of the system as a tool. Ideally, the user should not "see" the system at all. She should see through the system to the task being performed. When I am typing a document, for example, I see the page growing text; I don't "see" the word processor. An accomplished pianist sees the music being formed, not the piano; a good carpenter sees the nail going into the wood, not the hammering tool. Heidegger's frame of evaluation helps us produce systems more fit for use.

It is only the style of Wittgenstein that opposes the style of Descartes. A person working in this style views the unfolding of the software design as the unfolding of a language game, in which new words are added to the language over time.

This immediately links to software development as a cooperative game of invention and communication. I probably owe a good deal of my construction of the cooperative game model to Ehn's writings. I had read and forgotten the following article years before working out the cooperative game idea. As I started to write this book, I reviewed this article and was shocked to see how many of my words echoed Ehn's.

Ehn is concerned with the building of shared experience through shared practice, of using practice directly as a basis for discovering needs. In other words, he is working with tacit knowledge. More than that, he highlights the place of *skill* in carrying out practices (it is interesting to read Musashi's words pointing out much the same). Although skill is a topic I have mentioned, Ehn develops it much more thoughtfully and completely.

I took the game thinking in a different direction. I am concerned with playing a group game amicably, so that communication can take place at all. You will see that Ehn's ideas complement the rest of the ideas in this book.

Pelle Ehn expresses it much better in his own words than I can through summaries. *Work-Oriented Development of Software Artifacts* is out of print, sadly. However, this excerpt from "Scandinavian Design: On

Participation and Skill" (Ehn 1992) contains the line of thinking I feel is so important.

The article is longer than I can reproduce here. In this extract, I added italics to emphasize points relevant to the notion of cooperative games.

"On Participation and Skill"

. . .

In the following, I will propose that this new understanding can be buttressed by an awareness of language games and the ordinary language philosophy of Ludwig Wittgenstein. My focus is on the shift in design from language as description towards language as action.

Rethinking Systems Descriptions

A few years ago I was struck by something I had not noticed before. While thinking about how perspectives make us select certain aspects of reality as important in a description, I realized I had completely overlooked *my own presumption that descriptions in one way or another are mirror images of a given reality.* My earlier reasoning had been that because there are different interests in the world, we should always question the objectivity of design choices that claimed to flow from design as a process of rational decision making. Hence, *I had argued that we needed to create descriptions from different perspectives in order to form a truer picture*. I did not, however, question the Cartesian epis ontology of an inner world of experiences (mind) and an outer world of objects (external reality). Nor did I *question the assumption that language was our way of mirroring this outer world* of real objects. By focusing on which objects and which relations should be represented in a systems description, I took for granted the Cartesian mind-body dualism that Wittgenstein had so convincingly rejected in Philosophical Investigations (1953). Hence, although my purpose was the opposite, my perspective blinded me to the subjectivity of craft, artistry, passion, love, and care in the system descriptions.

Our experiences with the UTOPIA project caused me to re-examine my philosophical assumptions. Working with the end users of the design, the graphics workers, some design methods failed while others succeeded. *Requirement specifications and systems descriptions based on information from interviews were not very successful*. Improvements came when we made joint visits to interesting plants, trade shows, and vendors and had discussions with other users; when we dedicated considerably more time to learning from each other, designers from graphics workers and graphics workers from designers; *when we started to use design-by-doing methods and descriptions such as mock-ups and work organization games*; and when we started to understand and use traditional tools as a design ideal for computer-based systems.

The turnaround can be understood in the light of two Wittgensteinian lessons. The first is not to underestimate the importance of skill in design. As Peter Winch (1958) has put it, "A cook is not a man who first has a vision of a pie and then tries to make it. He is a man skilled in cookery, and both his projects and his achievements spring from that skill." The second is not to mistake the role of description methods in design: Wittgenstein argues

convincingly that what a picture describes is determined by its use.

In the following I will illustrate how our "new" UTOPIAN design methods may be understood from a Wittgensteinian position, that is, why design-by-doing and a skill-based participatory design process works. *More generally, I will argue that design tools such as models, prototypes, mock-ups, descriptions, and representations act as reminders and paradigm cases for our contemplation of future computer-based systems and their use. Such design tools are effective because they recall earlier experiences to mind*. It is in this sense that we should understand them as representations. I will begin with a few words on practice, the alternative to the "picture theory of reality."

Practice Is Reality

Practice as the social construction of reality is a strong candidate for replacing the picture theory of reality. In short, practice is our everyday practical activity. It is the human form of life. It precedes subject-object relations. Through practice, we produce the world, both the world of objects and our knowledge about this world. Practice is both action and reflection. But practice is also a social activity; it is produced in cooperation with others. To share practice is also to share an understanding of the world with others. However, this production of the world and our understanding of it take place in an already existing world. The world is also the product of former practice. Hence, as part of practice, knowledge has to be understood socially—as producing or reproducing social processes and structures as well as being the product of them (Kosik, 1967; Berger & Luckmann, 1966).

Against this background, we can understand the design of computer applications as a concerned social- and historical-conditioned activity in which tools and their use are envisioned. This is an activity and form of knowledge that is both planned and creative.

Once struck by the "naive" Cartesian presumptions of a picture theory, what can be gained in design by shifting focus from the correctness of descriptions to intervention into practice? What does it imply to take the position that what a picture describes is determined by its use? Most importantly, *it sensitizes us to the crucial role of skill and participation in design*, and to the opportunity in practical design to transcend some of the limits of formalization through the use of more action-oriented design artifacts.

Language as Action

Think of the classical example of a carpenter and his or her hammering activity. In the professional language of carpenters, there are not only hammers and nails. If the carpenter were making a chair, other tools used would include a draw-knife, a brace, a trying plane, a hollow plane, a round plane, a bow-saw, a marking gauge, and chisels (Seymour, 1984). The materials that he works with are elm planks for the seats, ash for the arms, and oak for the legs. He is involved in saddling, making spindles, and steaming.

Are we as designers of new tools for chairmaking helped by this labeling of tools, materials, and activities? In a Wittgensteinian approach the answer

would be: only if we understand the practice in which these names make sense. To label our experiences is to act deliberately. To label deliberately, we have to be trained to do so. Hence, the activity of labeling has to be learned. Language is not private but social. The labels we create are part of a practice that constitutes social meaning. We cannot learn without learning something specific. To understand and to be able to use is one and the same (Wittgenstein, 1953). Understanding the professional language of chairmaking, and any other language-game (to use Wittgenstein's term), is to be able to master practical rules we did not create ourselves. The rules are techniques and conventions for chairmaking that are an inseparable part of a given practice.

To master the professional language of chairmaking means to be able to act in an effective way together with other people who know chairmaking. To "know" does not mean explicitly knowing the rules you have learned, but rather recognizing when something is done in a correct or incorrect way. To have a concept is to have learned to follow rules as part of a given practice. Speech acts are, as a unity of language and action, part of practice. They are not descriptions but below I will elaborate on language-games, focusing on the design process descriptions in design, design artifacts, and knowledge in the design of computer applications.

Language-Games

To use language is to participate in language-games. In discussing how we in practice follow (and sometimes break) rules as a social activity, Wittgenstein asks us to think of games, how they are made up and played. We often think of games in terms of a playful, pleasurable engagement. I think this aspect should not be denied, but a more important aspect for our purpose here is that games are activities, as are most of the common language-games we play in our ordinary language.

Language-games, like the games we play as children, are social activities. To be able to play these games, we have to learn to follow rules, rules that are socially created but far from always explicit. *The rule-following behavior of being able to play together with others is more important to a game than the specific explicit rules*. Playing is interaction and cooperation. *To follow the rules in practice means to be able to act in a way that others in the game can understand*. These rules are embedded in a given practice from which they cannot be distinguished. To know them is to be able to "embody" them, to be able to apply them to an open class of cases.

We understand what counts as a game not because we have an explicit definition but because we are already familiar with other games. There is a kind of family resemblance between games. Similarly, professional language-games can be learned and understood because of their family resemblance to other language-games that we know how to play.

Language-games are performed both as speech acts and as other activities, as meaningful practice within societal and cultural institutional frameworks. To be able to participate in the practice of a specific language-game, one has to share the form of life within which that practice is possible. This form of life includes our

natural history as well as the social institutions and traditions into which we are born. This condition precedes agreed social conventions and rational reasoning. Language as a means of communication requires agreement not only in definitions, but also in judgments. Hence, intersubjective consensus is more fundamentally a question of shared background and language than of stated opinions (Wittgenstein, 1953).

This definition seems to make us prisoners of language and tradition, which is not really the case. Being socially created, the rules of language games, like those of other games, can also be socially altered. *There are, according to Wittgenstein, even games in which we make up and alter the rules as we go along. Think of systems design and use as language games. The very idea of the interventionistic design language-game is to change the rules of the language-game of use in a proper way.*

The idea of language-games entails an emphasis on how we linguistically discover and construct our world. However, language is understood as our use of it, as our social, historic, and intersubjective application of linguistic artifacts. As I see it, the language-game perspective therefore does not preclude consideration of how we also come to understand the world by use of other tools.

Tools and objects play a fundamental role in many language-games. A hammer is in itself a sign of what one can do with it in certain language-games. And so is a computer application. *These signs remind one of what can be done with them*. In this light, an important aspect in the design of computer applications is that its signs remind the users of what they can do with the application in the language-games of use (Brock, 1986). The success of "what-you-see-is-what-you-get" and "direct manipulation" user interfaces does not have to do with how they mirror reality in a more natural way, but with how they provide better reminders of the users' earlier experiences (Bødker, forthcoming). This is also, as will be discussed in the following, the case with the tools that we use in the design process.

Knowledge and Design Artifacts

As designers we are involved in reforming practice, in our case typically computer-based systems and the way people use them. Hence, the language-games of design change the rules for other language-games, in particular those of the application's use. What are the conditions for this interplay and change to operate effectively?

A common assumption behind most design approaches seems to be that the users must be able to give complete and explicit descriptions of their demands. Hence, the emphasis is on methods to support this elucidation by means of requirement specifications or system descriptions (Jackson, 1983; Yourdon, 1982).

In a Wittgensteinian approach, the focus is not on the "correctness" of systems descriptions in design, on how well they mirror the desires in the mind of the users, or on how correctly they describe existing and future systems and their use. Systems descriptions are design artifacts. In a Wittgensteinian approach, the crucial question is how we use them, that is, what role they play in the design process.

The rejection of an emphasis on the "correctness" of descriptions is especially important. In this, we are advised by the author of perhaps the strongest arguments for a picture theory and the Cartesian approach to design—the young Wittgenstein in Tractatus Logico-Philosophicus (1923). The reason for this rejection is the fundamental role of practical knowledge and creative rule following in language-games.

Nevertheless, we know that systems descriptions are useful in the language-game of design. The new orientation suggested in a Wittgensteinian approach is that we see such descriptions as a special kind of artifact that we use as "typical examples" or "paradigm cases." *They are not models in the sense of Cartesian mirror images of reality* (Nordenstam, 1984). *In the language-game of design, we use these tools as reminders for our reflection on future computer applications and their use. By using such design artifacts, we bring earlier experiences to mind, and they bend our way of thinking of the past and the future*. I think that this is why we should understand them as representations (Kaasboll, forthcoming). And this is how they inform our practice. *If they are good design artifacts, they will support good moves within a specific design language-game*.

The meaning of a design artifact is its use in a design language-game, not how it "mirrors reality." Its ability to support such use depends on the kinds of experience it evokes, its family resemblance to tools that the participants use in their everyday work activity. Therein lies a clue to why the breakthrough in the UTOPIA project was related to the use of prototypes and mockups. Since the design artifacts took the form of reminders or paradigm cases, they did not merely attempt to mirror a given or future practice linguistically. They could be experienced through the practical use of a prototype or mockup. This experience could be further reflected upon in the language-game of design, either in ordinary language or in an artificial one.

A good example from the UTOPIA project is an empty cardboard box with "desktop laser printer" written on the top. There is no functionality in this mockup. Still, it works very well in the design game of envisioning the future work of makeup staff. It reminded the participating typographers of the old "proof machine" they used to work with in lead technology. At the same time, it suggested that with the help of new technology, the old proof machine could be reinvented and enhanced.

This design language-game was played in 1982. At that time, desktop laser printers only existed in advanced research laboratories, and certainly typographers had never heard of them. To them, the idea of a cheap laser printer was "unreal."

It was our responsibility as professional designers to be aware of such future possibilities and to suggest them to the users. It was also our role to suggest this technical and organizational solution in such a way that the users could experience and envision what it would mean in their practical work, before the investment of too much time, money, and development work. Hence, the design game with the mockup laser printer. The mockup made sense to all participants—users and designers (Ehn & Kyng, 1991).

This focus on nonlinguistic design artifacts is not a rejection of the importance of linguistic ones. Understood as triggers for our imagination rather than as mirror images of reality, they may well be our most wonderful human inventions. Linguistic design artifacts are very effective when they challenge us to tell stories that make sense to all participants.

Practical Understanding and Propositional Knowledge

There are many actions in a language-game, not least in the use of prototypes and mockups, that cannot be explicitly described in a formal language. What is it that the users know, that is, what have they learned that they can express in action, but not state explicitly in language? Wittgenstein (1953) asks us to "compare knowing and saying: how many feet high Mont Blanc is—how the word 'game' is used—how a clarinet sounds. If you are surprised that one can know something you are perhaps thinking of a case like the first. Certainly not of one of the third."

In the UTOPIA project, we were designing new computer applications to be used in typographical page makeup. The typographers could tell us the names of the different tools and materials that they use such as knife, page ground, body text, galley, logo, halftone, frame, and spread. They could also tell when, and perhaps in which order, they use specific tools and materials to place an article. For example, they could say, "First you pick up the body text with the knife and place it at the bottom of the designated area on the page ground. Then you adjust it to the galley line. When the body text fits you get the headline, if there is not a picture," and so forth. What I, as designer, get to know from such an account is equivalent to knowing the height of Mont Blanc. What I get to know is very different from the practical understanding of really making up pages, just as knowing the height of Mont Blanc gives me very little of understanding the practical experience of climbing the mountain.

Knowledge of the first kind has been called propositional knowledge. It is what you have "when you know that something is the case and when you also can describe what you know in so many words" (Nordenstam, 1985). Propositional knowledge is not necessarily more reflective than practical understanding. It might just be something that I have been told, but of which I have neither practical experience nor theoretical understanding.

The second case, corresponding to knowing how the word game is used, was more complicated for our typographers. How could they, for example, tell us the skill they possess in knowing how to handle the knife when making up the page in pasteup technology? This is their practical experience from the language-games of typographic design. To show it, they have to do it.

And how should they relate what counts as good layout, the complex interplay of presence and absence, light and dark, symmetry and asymmetry, uniformity and variety? Could they do it in any other way than by giving examples of good and bad layouts, examples that they have learned by participating in the games of typographical design? As in the

case of knowing how a clarinet sounds, this is typically sensuous knowing by familiarity with earlier cases of how something is, sounds, smells, and so on.

Practical understanding—in the sense of practical experience from doing something and having sensuous experiences from earlier cases—defies formal description. If it were transformed into propositional knowledge, it would become something totally different.

It is hard to see how we as designers of computer systems for page makeup could manage to come up with useful designs without understanding how the knife is used or what counts as good layout. For this reason we had to have access to more than what can be stated as explicit propositional knowledge. We could only achieve this understanding by participating to some extent in the language-games of use of the typographical tools. Hence, participation applies not only to users participating in the language-game of design, but perhaps more importantly to designers participating in use. Some consequences of this position for organizing design language-games will be discussed in the following.

Rule Following and Tradition

Now, I turn to the paradox of rule-following behavior. As mentioned, many rules that we follow in practice can scarcely be distinguished from the behavior in which we perform them. We do not know that we have followed a rule until we have done it. The most important rules we follow in skillful performance defy formalization, but we still understand them. As Michael Polanyi (1973), the philosopher of tacit knowledge, has put it: "It is pathetic to watch the endless efforts—equipped with microscopy and chemistry, with mathematics and electronics—to reproduce a single violin of the kind the half-literate Stradevarius turned out as a matter of routine more than 200 years ago." This is the traditional aspect of human rule-following behavior. Polanyi points out that what may be our most widely recognized, explicit, rule-based system—the practice of Common Law—also uses earlier examples as paradigm cases. Says Polanyi, "[Common Law] recognizes the principle of all traditionalism that practical wisdom is more truly embodied in action than expressed in the rules of action." According to Polanyi this is also true for science, no matter how rationalistic and explicit it claims to be: "While the articulate contents of science are successfully taught all over the world in hundreds of new universities, the unspecifiable art of scientific research has not yet penetrated to many of these." The art of scientific research defies complete formalization; it must be learned partly by examples from a master whose behavior the student trusts.

Involving skilled users in the design of new computer application when their old tools and working habits are redesigned is an excellent illustration of Polanyi's thesis. If activities that have been under such pressure for formalization as Law and Science are so dependent on practical experience and paradigm cases, why should we expect other social institutions that have been under less pressure of formalization to be less based on practical experience, paradigm cases, and tacit knowledge?

Rule Following and Transcendence

If design is rule-following behavior, is it also creative transcendence of traditional behavior. Again, this is what is typical of skillful human behavior, and is exactly what defies precise formalization. *Through mastery of the rules comes the freedom to extend them*. This creativity is based on the open-textured character of rule-following behavior. *To begin with, we learn to follow a rule as a kind of dressage, but in the end we do it as creative activity* (Dreyfus & Dreyfus, 1986). Mastery of the rules puts us in a position to invent new ways of proceeding. As the Wittgenstein commentator Alan Janik has put it: "There is always and ineliminably the possibility that we can follow the rule in a wholly unforeseen way. This could not happen if we had to have an explicit rule to go on from the start . . . the possibility of radical innovation is, however, the logical limit of description. This is what tacit knowledge is all about" (Janik, 1988). This is why we need a strong focus on skill both in design and in the use of computer systems. We focus on existing skills, not as to inhibit creative transcendence, but as a necessary condition for it.

But what is the role of "new" external ideas and experiences in design? How are tradition and transcendence united in a Wittgensteinian approach? It could, I believe, mean utilizing something like Berthold Brecht's theatrical "alienation" effect Verfremdungseffekt to highlight transcendental untried possibilities in the everyday practice by presenting a well-known practice in a new light: "the aspects of things that are most important to us are hidden because of their simplicity and familiarity" (Wittgenstein, 1953). However, as Peter Winch (1958, p. 119) put it, in a Wittgensteinian approach: "the only legitimate use of such a Verfremdungseffekt is to draw attention to the familiar and obvious, not to show that it is dispensable from our understanding."

Design artifacts, linguistic or not, may in a Wittgensteinian approach certainly be used to break down traditional understanding, but they must make sense in the users' ordinary language-games. If the design tools are effective, it is because they help users and designers to see new aspects of an already well-known practice, not because they convey such new ideas. It is I think fair to say that this focus on traditional skill in interplay with design skill may be a hindrance to really revolutionary designs. The development of radically new designs might require leveraging other skills and involving other potential users. Few designs, however, are really revolutionary, and for normal everyday design situations, the participation of traditionally skilled users is critical to the quality of the resulting product.

The tension between tradition and transcendence is fundamental to design. There can be a focus on tradition or transcendence in the systems being created. Should a word processor be designed as an extension of the traditional typewriter or as something totally new? Another dimension is professional competence: Should one design for the "old" skills of typographers or should new knowledge replace those skills in future use? Or again, with the division of labor and cooperation: Should the new design

support the traditional organization in a composing room or suggest new ways of cooperation between typographers and journalists? There is also the tension between tradition and transcendence in the goods or services to be produced using the new system: Should the design support the traditional graphical production or completely new services, such as desktop publishing?

Tradition and transcendence, that is the dialectical foundation of design.

Design by Doing: New "Rules of the Game"

What do we as designers have to do to qualify as participants in the language-games of the users? What do users have to learn to qualify as participants in the language-game of design? And what means can we develop in design to facilitate these learning processes?

If designers and users share the same form of life, it should be possible to overcome the gap between the different language-games. It should, at least in principle, be possible to develop the practice of design to the point where there is enough family resemblance between a specific language-game of the users and the language-games in which the designers of the computer application are intervening. A mediation should be possible.

But what are the conditions required to establish this mediation? For Wittgenstein, it would make no sense to ask this question outside a given form of life: "If a lion could talk, we could not understand him" (1953). In the arguments below, I have assumed that the conditions for a common form of life are possible to create, that the lions and sheep of industrial life, as discussed in the first part of this chapter, can live together. This is more a normative standpoint of how design ought to be, a democratic hope rather than a reflection on current political conditions.

To develop the competence required to participate in a language-game requires a lot of learning within that practice. *But, in the beginning, all one can understand is what one has already understood in another language-game*. If we understand anything at all, it is because of the family resemblance between the two language-games.

What kind of design tools could support this interplay between language-games? I think that what we in the UTOPIA project called design-by-doing methods—prototyping, mockups, and scenarios—are good candidates. Even joint visits to workplaces, especially ones similar to the ones being designed for, served as a kind of design tool through which designers and users bridged their language-games.

The language-games played in design-by-doing can be viewed both from the point of view of the users and of the designers. This kind of design becomes a language-game in which the users learn about possibilities and constraints of new computer tools that may become part of their ordinary language-games. The designers become the teachers that teach the users how to participate in this particular language-game of design. However, to set up these kind of language-games, the designers have to learn from the users.

However, paradoxical as it sounds, users and designers do not have to understand each other fully in playing language-games of design-by-doing together.

Participation in a language-game of design and the use of design artifacts can make constructive but different sense to users and designers. Wittgenstein (1953) notes that "when children play at trains their game is connected with their knowledge of trains. It would nevertheless be possible for the children of a tribe unacquainted with trains to learn this game from others, and to play it without knowing that it was copied from anything. One might say that the game did not make the same sense as to us." As long as the language-game of design is not a nonsense activity to any participant but a shared activity for better understanding and good design, mutual understanding may be desired but not really required.

User Participation and Skill

The users can participate in the language-game of design because the application of the design artifacts gives their design activities a family resemblance with the language-games that they play in ordinary use situations. An example from the UTOPIA project is a typographer sitting at a mockup of a future workstation for page makeup, doing page makeup on the simulated future computer tool.

The family resemblance is only one aspect of the methods. Another aspect involves what can be expressed. In design-by-doing, the user is able to express both propositional knowledge and practical understanding. Not only could, for example, the typographer working at the mockup tell that the screen should be bigger to show a full page spread—something important in page makeup—he could also show what he meant by "cropping a picture" by actually doing it as he said it. It was thus possible for him to express his practical understanding, his sensuous knowledge by familiarity. He could, while working at the mockup, express the fact that when the system is designed one way he can get a good balanced page, but not when it is designed another way.

Designer Participation and Skill

For us as designers, it was possible to express both propositional knowledge and practical understanding about design and computer systems. Not only could we express propositional knowledge such as "design-by-doing design tools have many advantages as compared with traditional systems descriptions" or "bit-map displays bigger than 22 inches and with a resolution of more than 2000 x 2000 pixels are very expensive," but in the language-game of design-by-doing, we could also express practical understanding of technical constraints and possibilities by "implementing" them in the mockup, prototype, simulation, or experimental situation. Simulations of the user interface were also important in this language-game of design.

As designers, our practical understanding will mainly be expressed in the ability to construct specific language-games of design in such a way that the users can develop their understanding of future use by participating in design processes.

As mentioned above, there is a further important aspect of language-games: We make up the rules as we go along. A skilled designer should be able to assist in such transcendental rule-breaking activities. Perhaps, this is the artistic competence that a good designer needs.

To really learn the language-game of the use activity by fully participating in that language-game is, of course, an even more radical approach for the designer. Less radical but perhaps more practical would be for designers to concentrate design activity on just a few language-games of use, and for us to develop a practical understanding of useful specific language-games of design (Ehn & Kyng, 1987). Finally, there seems to be a new role for the designer as the one who sets the stage for a shared design language-game that makes sense to all participants.

Some Lessons on Design, Skill, and Participation

As in the first practice-oriented part of this paper on designing for democracy at work, I end this second philosophically oriented part on skill-based participatory design with some lessons for work-oriented design.

General lessons on work-oriented design include:

1. Understanding design as a process of creating new language-games that have family resemblance with the language-games of both users and designers gives us an orientation for doing work-oriented design through skill-based participation—a way of doing design that may help us transcend some of the limits of formalization. Setting up these design language-games is a new role for the designer.
2. Traditional "systems descriptions" are not sufficient in a skill-based participatory design approach. Design artifacts should not be seen primarily as means for creating true "pictures of reality," but as means to help users and designers discuss and experience current situations and envision future ones.
3. "Design-by-doing" design approaches such as the use of mockups and other prototyping design artifacts make it possible for ordinary users to use their practical skill when participating in the design process.

Lessons on skill in the design of computer-based systems include:

1. Participatory design is a learning process in which designers and users learn from each other.
2. Besides propositional knowledge, practical understanding is a type of skill that should be taken seriously in a design language-game since the most important rules we follow in skillful performance are embedded in practice and defy formalization.
3. Creativity depends on the open-textured character of rule-following behavior, hence a focus on traditional skill is not a drawback to creative transcendence but a necessary condition. Supporting the dialectics between tradition and transcendence is the heart of design.

Lessons on participation in design of computer-based systems include:

1. Really participatory design requires a shared form of life—a shared social and cultural background and a shared language. Hence, participatory design means not only users participating in

design but also designers participating in use. The professional designer will try to share practice with the users.

2. To make real user participation possible, a design language-game must be set up in such a way that it has a family resemblance to language-games the users have participated in before. Hence, the creative designer should be concerned with the practice of the users in organizing the design process, and understand that every new design language-game is a unique situated design experience. There is, however paradoxical it may sound, no requirement that the design language-game make the same sense to users and designers. There is only [the] requirement that the designer set the stage for a design language-game in which participation makes sense to all participants.

Beyond the Boredom of Design

Given the Scandinavian societal, historical, and cultural setting, the first part of this chapter focused on the democratic aspect of skill-based participatory design, especially the important role of local trade unions and their strategies for user participation. In the second part, some ideas inspired by Ludwig Wittgenstein's philosophical investigations were applied to the everyday practice of skill-based participatory design. Practical understanding and family resemblance between language-games were presented as fundamental concepts for work-oriented design.

The concept of language-games is associated with playful activity, but what practical conditions are needed for such pleasurable engagement in design? Is the right to democratic participation enough?

In fact, the experiences from the work-oriented design projects indicates that most users find design work boring, sometimes to the point where they stop participating. This problem is not unique to the Scandinavian work-oriented design tradition. It has, for example, been addressed by Russell Ackoff (1974), who concluded that participation in design can be only successful if it meets three conditions: (1) it makes a difference for the participants, (2) implementation of the results is likely, and (3) it is fun.

The first two points concern the political side of participation in design. Users must have a guarantee that their design efforts are taken seriously. The last point concerns the design process. No matter how much influence participation may give, it has to transcend the boredom of traditional design meetings to really make design meaningful and full of involved action. The design work should be playful. In our own later projects, we have tried to take this challenge seriously and have integrated the use of future workshops, metaphorical design, role playing and organizational games into work-oriented design (Ehn & Sjogren, 1991).

Hence, the last lesson from Scandinavian designs is that formal democratic and participatory procedures for designing computer-based systems for democracy at work are not sufficient. Our design language-games must also be organized in a way that makes it possible for ordinary users not only to utilize their

practical skill in the design work, but also to have fun while doing so.

. . .

Reflections on Ehn's Writing

Each time I read Ehn's article, I discover that I may be more in debt to his writing than I previously thought. Rereading it just prior to writing this paragraph, I was struck by his use of the Shu-Ha-Ri construct, to his attention to "understanding through doing," and his understanding of how people grow new understanding through the act of doing.

I evidently wasn't ready to read very many of his words in 1993 and have grown into them over the years. It makes me wonder how many other concepts he mentions, but which I haven't yet noticed.

I hope you will take the time to reread this article in another year or two.

Musashi

Miyamoto Musashi was a 17th-century samurai who never wrote software.

He claimed never to have lost a fight. Losing a fight meant serious body damage, and it was quite an accomplishment to be alive with all limbs in place at the age of 70.

A romantic novel series about Musashi depicts his early life, fights, and mental development. It is a wonderful read and also vividly portrays his fighting approach, which his personal book describes.

His personal book is the *Go Rin No Sho*, in English *The Book of Five Rings* (I have the Thomas Cleary translation, Shambhala, 2000), which he wrote at age 70. That book outlines his approach as clearly as he can make it, describing mental states, specific moves, and the use of large groups. It is short, clear, and wonderfully absent of the usual Zen doubletalk, "Be by not being, fight by not fighting, win by losing," and so on.

I include Musashi here because three characteristics of his fighting style match my software development style, and he describes them so well:

- Do not develop an attachment to any one weapon or any one school of fighting.
- Practice and observe reflectively.
- Win.

The first recommendation is to use any and all schools and techniques, without great attachment.

At the time of his writing, warriors formed schools around particular stances, styles, weapons, and tactics. His view was that each had its merits and weaknesses; one should use the range of them without getting stuck in any one.

The same is true in software design techniques. Don't get stuck in UML, RUP, CMM, SEI, XP, CRC (insert your favorite school's or tool's acronym here). Use whichever you need at the instant you

need it. Discover what you need at different moments, so you can develop a tool- and method-attack strategy that will tell you which one to pick up and when to put it down.

The second recommendation is to reflect on what you do and how you do it. Reflective practice has been discussed throughout this book.

The third recommendation is to pay more attention to winning than to looking good.

Winning the software development game is shipping the software. If you can do so without process, do so. My favorite-ever recommendation to a group was:

"What? You have a five-week project, with three developers who have done this before in the same technology? You don't need a development coordinator—just do it and go home."

Musashi said, "Do not do anything useless."

Musashi cared about winning the game, which in his case was life-or-death. I am attached to delivering the software. The prettiness of the dance doesn't matter if the software comes out at the wrong time.

In the following, notice that even in the 17th century, Musashi describes Shu-Ha-Ri and the importance of developing skill.

The "opponent" in software development is the problem to be solved. "Killing the opponent" is delivering the software and winning the game. Here are some of his words (or Cleary's translation of them), presented as individual excerpts.

The Book of Five Rings

1. Now, in composing this book, I have not borrowed the old saying of Buddhism or Confucianism, nor do I make use of old stories from military records or books on military science . . .
2. The field of martial arts is particularly rife with flamboyant showmanship, with commercial popularization and profiteering on the part of both those who teach the science and those who study it. The result of this must be, as someone said, that "amateuristic martial arts are a source of serious wounds." . . .
3. The master carpenter, knowing the measurements and designs of all sorts of structures, employs people to build houses. In this respect, the master carpenter is the same as the master warrior. . . . As the master carpenter directs the journeymen, he knows their various levels of skill and gives them appropriate tasks. . . . Efficiency and smooth progress, prudence in all matters, recognizing true courage, recognizing different levels of morale, instilling confidence, and realizing what can and cannot be reasonably expected-such are the matters on the mind of the master carpenter. The principle of martial arts is like this. . . .
4. Speaking in terms of carpentry, soldiers sharpen their own tools, make various useful implements, and keep them in their utility boxes. . . . An essential habit for carpenters is to have sharp tools and keep them whetted. . . .

5. You should observe reflectively, with overall awareness of the large picture as well as precise attention to small details. . . .
6. Having attained a principle, one detaches from the principle; thus one has spontaneous independence in the science of martial arts and naturally attains marvels: discerning the rhythm when the time comes, one strikes spontaneously and naturally scores. . . .
7. In my individual school, one can win with the long sword, and one can win with the short sword as well. For this reason, the precise size of the sword is not fixed. The way of my school is the spirit of gaining victory by any means. . . .
8. When your life is on the line, you want to make use of all your tools. . . . We find that whatever the weapon, there is a time and situation in which it is appropriate. . . . Both the spear and the halberd depend on circumstances; neither is very useful in crowded situations. . . . they should be reserved for use on the battlefield. . . . [the bow] is inadequate for seiging a castle. . . .
9. In the present age, not only the bow but also the other arts have more flowers than fruit. Such skills are useless where there is a real need. . . .
10. You should not have any particular fondness for a particular weapon, or anything else for that matter. Too much is the same as not enough. . . . Pragmatic thinking is essential. . . .
11. Whatever guard you adopt, do not think of it as being on guard; think of it as part of the act of killing. . . .
12. Whether you adopt a large or small guard depends on the situation; follow whatever is most advantageous. . . .
13. (FIRST TECHNIQUE) . . . your sword now having bounced upward, leave it as it is until the opponent strikes again, whereupon you strike the opponent's hands from below. . . .
14. (SECOND TECHNIQUE) . . . If your sword misses the opponent, leave it there for the moment, until the opponent strikes again, whereupon you strike from below, sweeping upwards. . . .
15. (THIRD TECHNIQUE) . . . as the opponent strikes, you strike at his hands from below. . . . as he tries to knock your sword down, bring it up in rhythm, then chop off his arms sideways. The point is to strike an opponent down all at once from the lower position just as he strikes. . . .
16. Having a position without a position, or a guard without a guard, means that the long sword is not supposed to be kept in a fixed position. . . Where you hold your sword depends on your relationship to the opponent, depends on the place, and must conform to the situation; wherever you hold it, the idea is to hold it so that it will be easy to kill the opponent. . . . Even though you may catch, hit, or block an opponent's slashing sword, or tie it up or obstruct it, all of these moves are

opportunities for cutting the opponent down. This must be understood. . . .

17. . . . how to win using the long sword according to the laws of martial arts. This cannot be written down in detail; one must realize how to win by practice.

18. . . . the power of knowledge of the art of the sword. This is something that requires thorough examination, with a thousand days of practice for training and ten thousand days of practice for refinement. . . .

19. Other schools become theatrical, dressing up and showing off to make a living, commercializing martial arts. . . . Do you think you have realized how to attain victory just by learning to wield a long sword and training your body and your hands? This is not a certain way in any case. . . .

20. . . . the views of each school, and the logic of each path, are realized differently, according to the individual person, depending on the mentality. . . .

21. Thus in my individual school there is an aversion to a narrow, biased attitude. . . .

22. In my school, no consideration is given to anything unreasonable; the heart of the matter is to use the power of the knowledge of martial arts to gain victory any way you can. . . .

Applying Musashi to Software Development

I share three views with Musashi. I differ on the fourth.

Appropriate Tool, Appropriate Technique

Know your tools, know what you need at the moment, and you will know how to get value out of the tools at your disposal, even if they aren't perfect. You can even profitably use tools that were not originally constructed for software development.

Here is how I work in two different circumstances.

When given a CASE tool to use, I first exclude from use all of the tool's capabilities that do not lend value to the project at hand. Although this is an underutilization of an expensive tool, my goal is not to use a tool to its maximum, it is to deliver software.

On a different project, we may select as our primary strategy having the CASE tool create the final code. On this project, we plan on extending the tool as we need to so that it performs the job we want it to do.

Know your favorite tools and techniques for key tasks without getting overly attached to any one. Learn to adapt to whatever is available.

Direct Solution

See if you can just "cut off your opponent's arm with a single blow," as in sword fighting. In software terms, see if you can just "do it and go home." Avoid waste.

When you have to feint, block, and parry, understand that you are doing that because there is no alternative. Do just

enough of it to win. Avoid flamboyant showmanship, because it does not help deliver the system.

In software development, look for simple solutions to your process problems just as you look for simple solutions to your technical problems. Recall the one-sentence summary of Crystal Clear: "Put the people in a room with printing whiteboards, give them access to user experts, and have them deliver running tested software every two months." If you can do that, then just do that.

Reflection and Skill Development

Continue to develop your skill, and take time to reflect at regular intervals.

Microtouch Intervention

I do part company with Musashi in one area. He was in the business of killing or getting killed. I am in the business of helping people deliver software. There is a dramatic difference.

I like to cut quickly to the heart of the problem but keep the people fully intact. Arm-chopping is not an effective intervention strategy. I am after the smallest possible changes to the people on a project that accomplishes the job: *microtouch intervention*. (Actually, I suspect Musashi would agree with me, if he were in my business.)

Microtouch intervention is based on two ideas:

- With better understanding, smaller interventions are required.
- Many microscopic changes can produce a very large effect in unison.

Better understanding, smaller interventions. Two centuries ago, syphilis patients died. A century ago they underwent near-fatal arsenic treatments. These days they are given antibiotics. Early antibiotics were broad-spectrum bacteria killers; nowadays the antibiotics are targeted to the specific bacteria they are to kill.

Early computers were made with large vacuum tubes. Then, they were made with transistors. Now they are made with only a few thousands of atoms, recently even just single atoms.

Less energy is needed to effect a needed change the better we understand what we are doing. When we understand enough, we need only move molecules small distances, and the consequences will ripple out to produce the macro-effect we are interested in.

In software development, we are still in the amputation stage. As we better understand the forces underlying our profession, we can make smaller and smaller changes to improve a situation. I know that asking people to change their personal habits is a big request, so I prefer to change team seating or a few job assignments and let the human communications mechanism effect the much larger changes.

Small changes add up. I find it remarkable that just aligning many, microscopically small magnetic domains in a metal converts a nonmagnet to a strong magnet.

Aligning people's purposes has the same effect on a project.

Imagine many people, working to their own value systems, pursuing whatever goals happen to hit them each day. They

will sometimes, almost randomly, help each other or thwart each other.

Suppose you ask each person to make a tiny change, one that they find acceptably small. You can arrange the changes so that the people thwart each other less, help each other more. They are oriented in the same direction. With almost no energy change, the project team achieves a resulting power all out of proportion to the changes made (this is shown graphically in Figure 3-17 and Figure 3-18). This is summed up in Kent Arett's statement, "Paint the vision and get motivated people, and it's 'Game Over.'" (see "Fewer and Better" on page 195).

Microtouch intervention has its limits, of course. Sometimes, the correct move is not to continue with microtouch intervention but to replace the entire project structure with a new one. This happened once when we saw that a 30-person, colocated team could deliver the same as the failing 300-person multinational team.

The art, of course, is knowing when to rebuild the project and when microtouch intervention will work. Makes me wonder how Musashi would express that.

This page intentionally left blank

APPENDIX B.I

Naur, Ehn, Musashi: Evolution

I have referred at length in this book to the ideas of the three people quoted in Appendix B: Peter Naur, Pelle Ehn, and Musashi.

After five years of living with their words, I can mostly suggest that you read and reread those extracts. I continue to find value in them.

Naur, Ehn, Musashi: Evolution

Naur

From Peter Naur's writing, we get the idea that the team is working to create a common theory for their work. In terms of the Swamp Game (p. 49), the team starts off not knowing what they are supposed to build, where in the swamp to build it, or what the layout of the swamp is. The theory they are building is the answer to those three questions.

Part of the communication aspect of the cooperative game is establishing a shared direction for the team and a shared view of what the results need to look like. This is called *common vision* in some writings. Naur's theory includes this idea and also a common understanding of why the thing is put together the way it is.

Common vision and common understanding of why the thing is put together the way it are both part of any cooperative game, and most certainly our cooperative games of invention and communication.

Naur's discussion of theory building as a personal activity helps us to understand modes of transmitting understanding from one person to another. There is nothing that says that written documentation is the best way to convey understanding; possibly it is the worst. If we take the challenge to "convey understanding," then we can experiment with different ways until we find some that work better.

Ehn

From Pelle Ehn's writing, we get the idea that the understanding of the task to be done may never be perfect, but it may never need to be perfect. The magic lies in the back-and-forth between developer and user, creating new understanding about the task at hand and the tools being created.

It is easy to look at Ehn's team's assignment from 1986 and think that we are long past the days when people couldn't understand how the computer could help them. However, every organization working on improving their organizational process is faced with this problem. Until the system gets delivered and put into use, there is really no way that the users can tell how the presence of the new system will change the ways they work with each other, and the ways they carry out their jobs.

Musashi

I use the Musashi quotes to start off my one-day workshops introducing agile development. At first, it seems strange to use samurai quotes to understand software development, but then later it becomes so obvious how much his writings

have in common with modern software development.

I highlight three of his motifs:

- Waste no movement.
- Learn each tool's strength; don't become attached to any one tool.
- Reflect and adapt.

The one difference to resolve between Musashi's writings and software development is that Musashi keeps referring to killing the opponent. Who or what is the opponent in software development?

I was shocked in one organization when someone answered, "The users!" Looking for other opinions, I asked another person, who said, "The other specialists, like the database administrator!" Another person tried to clarify: "Sometimes you have to take out one of the other people on your team in order to get your work done."

Just in case you are tempted to make a similar response, I wish to stress that your teammate is not the opponent, nor is the user, your manager, or the sponsor.

The "opponent" is the problem you are trying to solve, the obstacles to delivering the system. "Killing the opponent" is solving the problem, delivering the system. Your situation will throw enough obstacles in your way that you don't need to consider your teammates as opponents.

Remember: "There's only us."