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.