

DISCUSS PROGRAMMING AS THEORY BUILDING

JASON CATENA

FEBRUARY 8, 2009

The present survey republishes major articles, reprints leading discussion, and collects erudite comment on the theory-building view of software programming.

Programming as Theory Building

Some views on programming, taken in a wide sense and regarded as a human activity, are presented. Accepting that programs will not only have to be designed and produced, but also modified so as to cater for changing demands, it is concluded that the proper, primary aim of programming is, not to produce programs, but to have the programmers build theories of the manner in which the problems at hand are solved by program execution. The implications of such a view of programming on matters such as program life and modification, system development methods, and the professional status of programmers, are discussed.

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.¹

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.**²

Section Programming as Theory Building reprints Naur 1985 (**emphasis mine**), which presents programming as theory building. On page 14 **Applying "Theory Building"** excerpts Cockburn 2006, which applies theory-building. On page 16 **Silver Bullets, Theory, and Agility** excerpts West 2008, which discusses tools to promote a common gestalt or group theory. On page 16 **References** lists sources of margin comments.

NAUR, P. 1985. Programming as theory building. *Microprocessing and Microprogramming* 15, 5, 253–261. Reprinted as §1.4 (pp. 37–49) of NAUR, P. 1992. *Computing: A human activity*. ACM Press, Addison-Wesley, NY.

¹ Naur bases his theorizing on observations that cannot be accommodated by the prevailing 'methodical view' on programming. Such observation can be made especially if unexpected situations arise, such as erroneous program executions, and if modifications of a program are required. In such situations it becomes evident that programming is not just about producing programs and documentations thereof. While the focus on "unexpected situations" seems to be somewhat arbitrary, it is actually well-supported by a philosophical doctrine called "falsificationism" (Popper 1934).
Wyssusek 2007

² In this argument, Naur actually combines two insights. First, presuppositions enter our judgments, thus motivating an analysis of presuppositions in the Kantian (1929) sense of critique. Second, our actions towards the world are not only mediated by our conceptualizations of world. Our actions are materializations of our conceptualizations.
Wyssusek 2007

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.³

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 first hand 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

³ As a major result of these studies I described programming as a human activity: theory building. By this description the core of programming is the programmer's developing a certain kind of understanding of the matters of concern.
Naur 2007

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 adaption, modification, and correction of errors in them, is essentially dependent on a certain kind of knowledge possessed by a group⁴ 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.^{5 6} 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.⁷ It may be noted that Ryle's notion of theory appears as an example of what K. Popper⁸ 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.

Ryle⁹ 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

⁴ Computing and especially software engineering is a social activity.

Dittrich et al. 2005

⁵ RYLE, G. *The concept of mind*. Harmondsworth, England, Penguin, 1963, first published 1949. Applying "theory building"

⁶ Analogously to Ryle's (1949) distinction between *knowing-that* (declarative knowledge) and *knowing-how* (procedural knowledge), scientific knowledge can be characterized as being comprised of two different yet complementary types: knowledge about the subject under investigation—the ultimate goal of science—and knowledge about how to achieve this goal. *Wyssusek 2007*

⁷ Programmers need to develop theories in order to be able to understand how a certain program will support a certain environment. Thus, programming requires not only knowledge about computers, programming languages, and the like, but also about the context in which the program developed will eventually be put in use. It is the latter knowledge that is not considered by 'methodical view' on information systems development. *Wyssusek 2007*

⁸ POPPER, K. R. and ECCLES, J. C. 1977. *The self and its brain*. London, Routledge and Kegan Paul

⁹ RYLE, G. *The concept of mind*. Harmondsworth, England, Penguin, 1963, first published 1949. Applying "theory building"

into such activities; while building the theory the person is trying to get it.¹⁰

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,¹¹ 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 answer to this issue is that the programmer's knowledge transcends that given in documentation in at least three

¹⁰ Programming shares the same kind of doubleness, being both interpretable representation and hardware construction. Naur (1985) rejects the idea that programming can be seen as the mere production of machine executable code—isolated construction without elements of conception and cooperation. He explains that a theory about what the program does, and how it does it, is built simultaneously with the construction of the executable code. Thus, in the terminology introduced above, programming work has the double character of being both construction and conception. This theory cannot be written down or otherwise formalised, and is only accessible to the programmers working on the particular project. In Wartofsky's (1973) terminology Naur's "program theory" is a secondary artefact conserving the acquired knowledge and skills in working with the program. However, Naur bases his analysis on the individualist philosophy of Ryle, thus neglecting the societal/cultural aspect of representation. The concept of secondary artefacts helps us understand that the way humans understand their surroundings (including programs) is culturally mediated. Secondary artefacts not only conserve knowledge and skill among the individuals whose experience they are based on, but secondary artefacts also transfer these across a given culture, e.g. the programming profession. *Bertelsen 2000*

¹¹ P. 187ff in KUHN, T.S. 1970. *The structure of scientific revolutions*, 2 ed. Chicago, U of Chicago Press; ISBN: 0-226-45803-2.

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.¹²

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.

¹² We emphasize ... [t]he widespread expectations in the scientific community to formal methods as an answer to the continued problems in software development. Naur emphasizes that only formal specifications which contribute by supporting the intuitive understanding of the matter at hand can be recommended. *Soeinsdottir and Frøkjar 2002*

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.¹³ 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.¹⁴ 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 man-made 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

¹³ Theory is dynamic in at least two dimensions. First, the theory is about an affair of the world and hence is dynamic to the same extent that the world is. Second, theory is emergent over time, as it expands and morphs. West 2008

¹⁴ Obviously, both developer and 'end-user' are users. For the 'end-user' to be able to make use of an information system it is necessary to have a sufficient understanding of the logic of the system and how it relates to the task at hand. With other words, the 'end-user' just like the developer needs to have a theory—in Naur's sense. Consequentially, the seemingly universal character of certain information systems is not due to some inherent properties of the respective system. Rather it is due to a network of practices that 'keep the information system alive' (see also Wyssusek 2005). Wyssusek 2007

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.¹⁵

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.¹⁶ 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.¹⁷ This problem of education of new programmers in an existing theory of a program is quite similar to that of the

¹⁵ Effectively, the programmer abstracts the cultural context being addressed, and maps that into the program text which freezes those behaviours into software. *Clear 1998*

¹⁶ Interestingly for the teacher of programming, Naur eschews documentation as a secondary construct to the programmers internalised theory of the program ... [f]or the creative process of theory building is inherently not a method or rule driven activity. *Clear 1998*

¹⁷ This insight has ramifications for the anthropology and life-cycle of large projects, where in order to produce modifications to an existing program without compromising the structure the programmers must understand the underlying theory. Since such an understanding is not easily acquired, we have Fred Brooks' observation that **adding more manpower to a late project makes it later** as a corollary: The people who understand the theory have to induct the newcomers, thus soaking up both groups' productivity. *Prager 2007*

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.¹⁸

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.¹⁹

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.^{20 21}

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 has 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

¹⁸ By a project we mean the work going into solving a partly defined, not too small, but definite problem, involving design and planning for new construction as essential parts. ... [F]irst, it involves *problem solving*. Second, in addition to solving, project activity involves *problem definition*. Indeed the initial definition of the problem to be solved in a project is normally quite incomplete, or even logically inconsistent. It is an important part of the project activity to discuss the definition of the problem, to clarify it, to make it more definite, to modify it, to discover contradictions in it, and to discuss and resolve the contradictions. Third, project activity involves the *contact* and *organization* of the group of people engaged in the project. Typical projects can only be successfully tackled by having several, or many persons working in parallel. Naur 1970

¹⁹ With other words, were information systems development methodical and reducible to epistemic knowledge, the 're-creation' of the theory underlying the program would not be a problem at all—following the dictum: the artifact, *i.e.*, the information system, is its best documentation. Conclusively, with the 'methodical view' of information systems development bound to epistemic knowledge, every problem occurring while 're-creating' the theory underlying the information system must be due to some error. Wyssusek 2007

²⁰ Naur takes implicitly a hermeneutical stance, drawing our attention to the knowledge that is always already required when 'acquiring' new knowledge, *e.g.*, through the 're-creation' of the theory underlying an information system. The unlikelihood of different teams to come up with identical 're-creations' is due to the different horizons of meaning (*e.g.*, Gadamer 1999) possessed by the different teams and which are at the basis of any theorizing. Wyssusek 2007

²¹ People familiar with different tools understand problems and their solutions differently. Ullall-Espersen 2008

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.²²

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.²³ 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,²⁴ taking his illustrations from the history of physics, and Medawar,²⁵ arguing as a biologist, is that the notion of scientific method as a set of guidelines for the practising scientist is mistaken.²⁶

This conclusion is not contradicted by such work as that of Polya^{27 28} 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

²² Naur also draws our attention to the important role of practice. Information systems development is not just the mere application of methods in order to achieve a given end. It is also not just about creating some artifact, be it an information system, some concepts, or a theory. Both information systems development and the resulting information system are bound to practices through which the meaning of the information system is continuously 'created' and 're-created'. If these practices get interrupted, the information system "dies".
Wyssusek 2007

²³ *Representational documentation*. Western culture, since the Age of Enlightenment, has believed it possible to construct formal written or mathematical models that represented reality. This general belief is exemplified in computer science by the belief that you can formally capture requirements, specifications, and models and that these documents are sufficiently representative of reality that they carry all the semantic and syntactic information necessary to create the software.
West 2008

²⁴ FEYERABEND, P. 1978. *Against method*. London, Verso Editions; ISBN: 86091-700-2.

²⁵ MEDAWAR, P. 1982. *Pluto's republic*. Oxford, University Press; ISBN: 0-19-217726-5.

²⁶ In critiquing the notion of design as a scientific activity, Naur [1985] recalls that Feyerabend [1975 *sic*] and Medawar [1982] have both come to the conclusion that "the notion of scientific method as a set of guidelines for the practicing scientist is mistaken." Naur concludes that such a set of guidelines for design is also mistaken.

McPhee 1997

²⁷ POLYA, G. 1954. *Mathematics and plausible reasoning*. NJ, Princeton U Press

²⁸ POLYA, G. 1957. *How to solve it*. NY, Doubleday Anchor Book

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.²⁹ 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* Brooks³⁰ or Moher and Schneider³¹). 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-workers³² 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.³³ This study gives extensive information on a considerable number of modifications in one

²⁹ Peter Naur has also contributed to a deeper understanding of the serious and varied problems connected to design, production and service of large software systems, now treated under the term software engineering—a term coined at the conference in *Garmisch Partenkirchen* held in October 1968 [sponsored by NATO Science Committee]. This conference created a sensation by the first open admission of the software crisis, as mentioned by Dijkstra [1972]. Still to day our profession has serious problems in meeting the demands for more and better software systems. According to Naur we shall not expect these difficulties to be overcome mainly through new methods in software development. This is one of the conclusions in the paper *Programming as Theory Building*, where a thorough treatment of programming viewed as a human activity is given. *Sveinsdottir and Frøkjær 2002*

³⁰ BROOKS, R. E. Studying programmer behaviour experimentally. *Comm. ACM* 23(4): 207–213, 1980.

³¹ MOHER, T., and SCHNEIDER, G. M. Methodology and experimental research in software engineering, *Int. J. Man-Mach. Stud.* 16: 65–87, 1. Jan. 1982.

³² FLOYD, C. 1984. Eine untersuchung von software—entwicklungs—methoden. Pp. 248–274 in *Programmierungsumgebungen und compiler*, ed H. MORGENBROD and W. SAMMER, Tagung I/1984 des German Chapter of the ACM, Stuttgart, Teubner Verlag; ISBN: 3-519-02437-3.

³³ OSKARSSON, Ö. 1982. Mechanisms of modifiability in large software systems. *Linköping studies in science and technology, Dissertations*, no. 77, Linköping; ISBN: 91-7372-527-7.

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.³⁴

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

³⁴ Not only does Naur question the claimed methodical nature of software development. He also questions an anthropological view that enters as a presupposition into the favorable evaluation of the use of methods in software development. Naur clearly establishes a link between our conceptualizations of the nature of software development with our judgment of the effort that goes into the activity of software development. There, the capturing, codification, and scientific 'enhancement' of workers' knowledge dictated by the principles of scientific management led to the popular understanding of human work as being nothing more than the application of methods in order to achieve a given end. Both the limitations of codification as well as the conditions of their application, *i.e.*, the conditions of industrial production, found their way as presuppositions into the conceptualization of work. *Wyssusek 2007*

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.³⁵ 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.

³⁵ Along with Peter Naur (1985) we believe that computer systems development is about gaining insight and knowledge of "how the matters at hand can be supported by the execution of a program", rather than simply "the production of programs and certain other texts." In other words, we seek to build systems in interaction with the world, constructing theories, as Naur advises, such that the resulting programs can respond over time "to the changes taking place in the real world activity." *Anderson and Anderson 1995*

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.

Excerpt from COCKBURN, A.A.R. 2006. Appendix B: Naur, Ehn, Musashi. *Agile software development—The cooperative game*, 2 ed. Addison-Wesley. Reprints Naur 1985. <http://alistair.cockburn.us/ASD+book+extract:+Naur,+Ehn,+Musashi>

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 ten 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 jiggle 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.

Silver Bullets, Theory, and Agility

A **story card** is an evocative device³⁶ that recalls to mind past conversations and context that are far too rich to capture in any representational model. (A story card can also be a reminder of the need for future conversations.) A story card is an exemplar of the kind of external artifact necessary for gestalt formation.

A story is a fragment of a theory. A backlog of stories is a model of the current understanding of a theory—or a subset of that theory. The product backlog, captured and displayed as a set of story cards is a visual and tactile artifact of the group's current understanding of a theory. A sprint backlog is the same with added focus.

Stories are written by customers/users because the point of theory is an understanding of an affair of the world. Software developers, traditionally, have very little understanding of the world; and, stereotypically, have very little understanding of affairs.

Stories are written and told to develop and expand a whole-team gestalt. Stories are not created to be units of work that enable Scrum project management.

Pair programming is micro-theory building using tests and code as the external artifacts. Each pair extends the shared theory with character development (objects) and dialog (messages, object interactions) in service of the overall plot (the affair of the world) being advanced by the code. Judicious mixing of pairs ensures that the entire group shares in the same gestalt. **Collective code ownership** accomplishes much the same goal—ensuring that everyone shares the same theory.³⁷

Even something as prosaic as a **burn-down chart** can contribute to theory. If we truly have a theory we know things about the world and about the software—including how difficult it will be to translate a world thing into a software thing. Burn down charts, along with similar charts for estimates, provide the feedback that confirms—or denies—our current theory.

References

Margin commentary from papers which refer to Naur 1985 or Naur 1992.

ANDERSON, W. L. and ANDERSON, S. L. 1995. Socially grounded engineering for digital libraries. *SIGOIS bull.* 16, 2 (Dec 1995), 3–5.

Excerpt from West, D. 2008. Silver bullets, theory, and agility. *Agile Journal*, Oct 2008. <http://www.agilejournal.com/content/view/862/195/>

³⁶ *Evocative device.* Human memory seems to be highly associative with the added feature that single part of a memory can evoke—return to conscious attention—the entire memory. Smell is often considered to be the strongest evocative device with a mere wisp of cinnamon in the air causing a flood of memories about mom and baking. Religious icons are another example—recalling to mind complicated stories and myths.

³⁷ Often at the expense of long-term support of individual theories of each subsystem, in systems with millions of lines. When feature-team developers implement a feature across many subsystems, and each subsystem has its own theory or set of tools, then long-term development and maintenance of each subsystem's theory or tools suffers in time because each feature team bears responsibility for only one set of changes across the entire system's code base. For each subsystem with its own theory or tool set, better to designate a team responsible for its long-term maintenance, to develop a long-term interest in a coherent theory. The size of the code base under collective code ownership should be set to minimize the number of separate theories or tools. *Catena*

<http://doi.acm.org/10.1145/226188.226189>

- CLEAR, T. 1998. Programming in context—the next step. *SIGCSE bull.* 30, 4 (Dec 1998), 13–14. <http://doi.acm.org/10.1145/306286.306295>
- DIJKSTRA, E. W. 1972. The humble programmer, *Comm. ACM* 15, 10.
- DITTRICH, Y., RÖNKKÖ, K., LINDBERG, O., ERICKSON, J., and HANSSON, C. 2005. Co-operative method development revisited. In *Proc. of the 2005 workshop on human and social factors of software engineering* (St Louis, Missouri, 16 May 2005). HSSE '05. ACM, New York, NY, 1–3. <http://doi.acm.org/10.1145/1083106.1083111>
- EKDAHL, B. 2002. Implications of the view of programs as formal systems, Peter J. Nürnberg (ed.), *Metainformatics. International Symposium, MIS 2002*, (Esbjerg, Denmark, 7–10 Aug 2002), Revised Papers, Springer, LNCS 2641, pp. 100–111. <http://www.springerlink.com/content/72p74gfv5feb3fl3/>

Abstract. The prevailing view of software development as the production of program systems is abandoned in favor of software development as theory building. This suggests that software development properly should be regarded as a deductive activity following the same methodology as is used in all deductive sciences. Programming is merely the description process in the activity of building theories about things in reality. Being the vehicle of theory building, programming may further be considered from a linguistic view. Programs are written in a language and have a proposed meaning; semantics. The main idea is that description and interpretation are complementary in a language; they cannot be fragmented within a language. Seeing programs as linguistic objects in this way has a profound influence on most aspects of software development.

Introduction. In [3, p.86] Floyd expresses doubt about “the validity of the established model of thought in software engineering as the sole foundation for our work as computer scientists”. Her doubt is, among other things, due to the basic assumption of “software development as the *production* of program systems on the basis of fixed requirements” [ibid]. Floyd thinks that a richer view is needed and the soundest available conceptual model in her opinion is Peter Naur’s view of programming as theory building. However, as Floyd points out, “Naur says little about what theory building consists in. And he does not account for the interpersonal nature of communal theory building” [3, p.87].

Naur’s paper takes up an interesting aspect of programming which is still not known to many programmers and is certainly not the view that is propagated in software education. Software engineering is still in its infancy and is mainly directed towards methods for program production and little effort is spent on foundational questions. I share Naur’s “conviction that it is important to have an appropriate understanding of what programming is” [9, p.253].

In this paper I will give answers to Floyd’s questions, both what theory building consists in and the nature of it. Particularly I will spell out the implications of the idea of programming as theory building.

- KAUTZ, K. and PRIES-HEJE, J. 1999. Systems development education and methodology adoption. *SIGCPR comput. pers.* 20, 3 (Jul. 1999), 6–26. Quotes NAUR, P. 1970. Project activity in computer science education. *Lezione 'Leonardo Fibonacci' 1969*, Pisa, 1970 April 8. Reprinted as §4.5 of NAUR, P. 1992. *Computing: A human activity*. ACM Press, Addison-Wesley, NY: <http://doi.acm.org/10.1145/568508.568509>

§4.5 Project activity in computer science education (1970). Introduction. What project activity is. Projects in computer science. Problem solving. Design techniques. Project documentation and the design process. The typewriter as a tool in documentation. Use of check lists. The need for group work. Introducing

project work in higher education. Guidance of project work via the evaluation procedure. Students' mutual evaluation. Automatic grading of programs. Evaluation of group work. Conclusion.

MCPHEE, K. 1997. Design theory and software design. Dept of CS, U of Alberta. Tech report.

<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.6361>

Some researchers attack the design-as-science viewpoint by questioning the validity of the philosophies of science. By arguing that a philosophy of science is flawed, they state that applying such a philosophy to design is flawed and therefore without validity. The logical-empiricist design theorists were the first who were criticized using this line of argument. Rzevski [1980] notes that even Hume, the intellectual forefather of logical empiricism, found two major flaws in the traditional scientific method. First, the inductive step has no logical explanation that adequately can describe how a theory is derived from a set of specific observations. Second, a natural law that is discovered as a result of applying the scientific method cannot be independently verified. It wasn't until Popper's [1965] ideas of "falsification", wherein it is always possible that a new experiment might be devised to contradict the conclusions embodied in the natural law, that these ideas were fully expressed in the philosophy of science. Even Alexander [1964], in attempting to apply empirical traditions to design, realizes that there are no fundamental truths in design. Instead, he adopts a principle of fallibility to test designs for success [March 1984]. Alexander acknowledges that designs may not entirely satisfy the constraints imposed upon them. There exists always the possibility that another design can better satisfy the constraints.

In criticizing the link between science and design, Cross, Naughton, and Walker [1980] claim that the antecedent observations regarding the nature of science are flawed. Citing several authors who argue against Popper's [1968] and Kuhn's [1970] models of science, they point out the "epistemological chaos" that plagues the philosophies of science. They claim that design cannot be equated to science on the grounds that the epistemology of science is unstable. They don't wish to use science as a reference for describing design, because the reference is itself a moving target and poorly described. In critiquing the notion of design as a scientific activity, Naur [1985] recalls that Feyerabend [1975] and Medawar [1982] have both come to the conclusion that "the notion of scientific method as a set of guidelines for the practicing scientist is mistaken." Naur concludes that such a set of guidelines for design is also mistaken.

NAUR, P. 2007. Computing versus human thinking. *Commun. ACM* 50, 1 (Jan 2007), 85–94.

<http://doi.acm.org/10.1145/1188913.1188922>

PRAGER, D. 2007. Programming is theory-building. *The daily kibitz: Un-called for advice and occasional musings*, 19 Apr 2007.

<http://dailykibitz.blogspot.com/2007/04/programming-is-theory-building.html>

SVEINSDOTTIR, E. and FRØKJÆR, E. (eds.). 2002. *Bibliography of Peter Naur*, DIKU-rapport 88/22, 35 p. including supplement—version 2002 maj 6.

<http://www.naur.com/bibliography.html>

ULDALL-ESPERSEN, T. 2008. The usability perspective framework. In *CHI '08 extended abstracts on human factors in computing systems* (Florence, Italy, 5–10 Apr 2008). CHI '08. ACM, New York, NY, 3813–3818.

<http://doi.acm.org/10.1145/1358628.1358935>

WEST, D. 2008. Silver bullets, theory, and agility. *Agile Journal*, Oct 2008. Discusses tools to promote a common gestalt or group theory.

<http://www.agilejournal.com/content/view/862/195/>

WYSSUSEK, B. 2007. A philosophical re-appraisal of Peter Naur's notion of "programming as theory building". *Proc. ECIS2007*, University St Gallen, 2007, 1505–1514.

<http://is2.lse.ac.uk/asp/aspecis/20070190.pdf>