2007

A Philosophical Re-Appraisal of Peter Naur's Notion of "programming as Theory Building"

Boris Wyssusek
Queensland University of Technology, b.wyssusek@qut.edu.au

A PHILOSOPHICAL RE-APPRAISAL OF PETER NAUR'S NOTION OF "PROGRAMMING AS THEORY BUILDING"

Wyssusek, Boris, Queensland University of Technology, 126 Margaret Street, Brisbane 4000, Queensland, Australia, b.wyssusek@qut.edu.au

Abstract

The recent resurgence of interest in design science as meta-theoretical framework for information systems research is taken as a motivation for the re-appraisal of Peter Naur's notion of "programming as theory building". In stark contrast to the artifact orientation of current design science frameworks, Naur considers the artifact orientation as one of the most fundamental misconception in the information systems discipline (in its widest sense). Naur's work is philosophically challenging since it has not only meta-theoretical relevance, e.g., as an opposition to design science frameworks, but also practical relevance, e.g., for the conduct and management of information systems development—thus bridging the traditional theory–practice gap. Moreover, Naur's work not only exhibits the tension between relevance and rigor in information systems research but also illustrates that rigorous and relevant research may still yield questionable results. The goal of the paper is to direct the interested reader's attention to a view on information systems development that challenges a number of presuppositions not only of the meta-theoretical design science frameworks in contemporary information systems research but of other meta-theoretical frameworks as well. It might thus serve the reader as a starting point for a critical reflection on meta-theoretical presuppositions underlying her/his understanding of information systems development.

Keywords: information systems development, programming as theory building, design science, philosophy.

1 INTRODUCTION

The recent resurgence of interest in design science as a meta-theoretical framework for information systems research (e.g., Hevner, March, Park and Ram 2004) provides ample motivation for the exploration of other dated ideas which had been put forward in response to issues at the very core of the discipline.[1] Quite in contrast to the artifact orientation of current design science frameworks in information systems research, this paper is concerned with the work of an author who considers the artifact orientation of the information systems discipline (in its widest sense) as one of its most fundamental misconceptions.

The motivation for this paper is threefold: First, design science research in information systems commenced around the time of publication of the work re-appraised in this paper. The resurgent interest in design science research indicates that there is still need for it, which may be true for the re-appraised work as well. Second, the controversial views of the re-appraised work and early publications on design science research in information systems created a productive intellectual tension in the discipline. With the resurgent interest in design science it may be worthwhile to trigger a resurgence of interest in the re-appraised work in order to re-create this productive intellectual tension. Third, the re-appraised work not only exhibits the tension between relevance and rigor in information systems research but also illustrates that rigorous and relevant research may still yield questionable results.

While the boom (or hype?) of design science research within the information systems discipline provides a large part of the motivation for this paper, its focus is on the re-appraised work but not on design science.[2] The goal of the paper is to direct the attention of the interested reader to a work that was written more than twenty years ago and that can be viewed as an challenging opposition to design science's artifact orientation. The philosophical perspective of the re-appraisal sets the stage at a meta-theoretical level, thus allowing for a focus on presuppositions underlying the re-appraised work. This focus shall support the curious reader in establishing connections to presuppositions underlying contemporary design science research in information systems—or any other meta-theoretical framework: Following Kant (1929) and Heidegger (1962), the questioning of presuppositions paves the way towards a critique or critical reflection.

2 CONTEXT: SOFTWARE DEVELOPMENT BETWEEN ART AND ENGINEERING

Questions regarding the nature of software engineering or programming or information systems development tend to make us uncomfortable, since they aim at our identity as software engineers, programmers, or information systems developers.[3] Critical reflection is certainly not the daily business of the members of this profession who understand themselves as solvers of real-world problems.

[1] The reader shall be reminded that Simon's most influential classic "The Sciences of the Artificial" was first published in 1969 and the main ideas remained the same in the third edition in 1996 (for a critical review see, e.g., Eisenberg 2003).

[2] One of the reviewers of this paper recognized my references to design science throughout this paper as a "straw-man argument." I agree—and I do not think there is something wrong with such an approach. It allows me to deal with design science rather superficially, basically reducing design science to an "artifact-oriented meta-theoretical framework."

[3] Following Naur (1985, p. 253), I use the terms "software engineering," "software development," "programming," and "information system development" interchangeably throughout the paper. Early efforts showing an interest in a design science framework for information systems research are, for example, Weber 1987; Walls, Widmeyer and El Sawy 1992; March and Smith 1995. Given the decade-long silence between early and current publications on such frameworks it may be adequate to speak of a "second wave of research on meta-theoretical design science frameworks for information systems research."

However, the need for critical reflection is occasionally acknowledged, especially when reports of yet another disaster in information systems development make the headlines. It is thus not a coincidence that the notion of software engineering—which provides a professional identity to a rather heterogeneous group of people concerned with software development in the most general sense of the terminus technicus—is a result of critical reflection triggered by the recognition of a crisis.

Since the famous NATO conference in 1968 the software development research community has been familiar with the notion of *software crisis* (Naur and Randell 1969). Plagued by the ever-increasing complexity of hardware and software, the development of reliable, effective, and efficient information systems had become the major challenge for researchers and practitioners alike. It was also on the occasion of this very conference that the notion of *software engineering* was introduced. "The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering" (Naur and Randell 1969, p. 13). Today it is generally acknowledged that this conference was the birth of the discipline of software engineering. Engineering was identified as the silver bullet to (almost) all the problems encountered when engaging in the development of complex hardware and software systems. Why *engineering*?

Definitions of engineering abound, yet share some common clauses: "Creating cost-effective solutions […] to practical problems […] by applying scientific knowledge […] to building things […] in the service of mankind" (Shaw 1990, p. 15). Obviously, the difference between craft or art and engineering rests on the application of scientific knowledge. 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. Thus, the latter type of knowledge is about means–ends relationships and, when codified, describes "a specification of steps which must be taken, in a given order, to achieve a given end" (Caws 1967, p. 339). In general, we refer to such a specification as method, derived from the Greek μετα (*meta*: along) and οδοσ (*odos*: way)—following a way.

With Descartes (1637), the application of the appropriate method of inquiry became the guarantor for obtaining truth.[4] Consequently, when engineering makes use of scientific knowledge it draws on knowledge about the subject matter and on knowledge about means–ends relationships, i.e., methods. But whereas in the realm of science the goal of applying (or following) a method is to gain knowledge, in the realm of engineering the goal is "building things." Reaching this goal is supposedly ensured—in the tradition of Cartesianism—when using the appropriate method: "The predisposition to believe in the power of methodologies comes from Descartes who proposed that truth is more a matter of proper method than genial insight or divine inspiration" (Hirschheim, Klein and Lyytinen 1995, p. 21). Given this belief, it is only conclusive that ever since the emergence of the discipline software engineering, information systems development has been understood as a "planned, deliberate activity—bounded in time and carried out in a systematic and orderly way" (Bansler and Havn 2003, p. 51). Hence, it has been understood as a methodical process, to an extent that "the modern concept of method has been so strongly impressed on our thinking about systems development, that the two concepts, information systems development and information systems development method, are completely merged in systems development literature" (Truex, Baskerville and Travis 2000, p. 56).

[4] For the contemporary critic of 'method-ism' it might be of interest to know that severe criticism of the scientists' obsession with method is even older than Descartes' thoughts: "Method—no word is more popular in our lectures these days, none more often heard, none gives off a more delightful ring than that term. Everything else, if you use it often enough, will end by nauseating your readers. This is the only thing that never makes them sick. If you leave it out, they think the feast you set before them is disgustingly seasoned and poorly prepared. If you use it often, they will believe that anything you give them is the ambrosial and nectared food of the gods" (Turnèbe 1600; quoted in Ong 1958, p. 228).

The sheer abundance of methods available and applied in software development (e.g., Oei, Van Hemmen, Falkenberg and Brinkkemper 1992) means that the once prescriptive metaphor *software engineering* has been taken seriously and now appears as description. But still the question remains: Have the expectations been met, or, is the description valid? So far, the hypes around innovations in software development methods have failed to meet the expectations raised: "In the 1960's for example, it was packaged software that was slated for replacing the software developer […]. The 1970's was the era of the structured revolution […] with its claims of error-free software […]. The 1980's was the decade of the fourth-generation language and the debut of end-user computing. […] In the latter half of the 1980's and into the 1990's, the software industry is still searching for dramatic solutions, this time with automated software engineering (CASE) tools […]" (Glass, Vessey and Conger 1992, pp. 183/184). Also, recurring failures in the development of information systems (e.g., Standish Group 1994; OASIG 1996; Standish Group 2003)[5], the persistence of the software crisis (Gibbs 1994), as well as the productivity paradox (e.g., Brynjolfsson 1993; Attewell 1994; Strassmann 1997) undermine the validity of the *description* of software development as engineering.

In contrast to the claimed adherence to methods, empirical findings question the very idea of methodical information systems development, since "methods are often unsuitable for some individuals (Naur 1993) and settings (Baskerville, Travis and Truex 1992). Similar methods in similar settings yield distinctly different results (Turner 1987). Developers may claim adherence to one method while ignoring this method in actual practice (Bansler and Bødker 1993)" (Truex et al. 2000, p. 54, formatting of references adapted). Adopting a technological innovation perspective, Kautz (2004) suggests that information systems development methodologies (meaning "methods") do not get *adopted* but *enacted*. Floyd (1992, p. 86) criticizes the discipline's "view of methods as rules laying down standardized working procedures to be followed without reference to the situation in hand or the specific groups of people involved." This suggests that while research has been preoccupied with method, it has simultaneously, neglected (or ignored) the *a*methodical aspects of information systems development, as explained by Truex et al. (2000, p. 74): "When the idea of method frames all of our perceptions about systems development, then it becomes very difficult to grasp its non-methodical aspects." In a similar vein, Bansler and Havn (2003, p. 51) note that these aspects "become marginalized and practically invisible, e.g. how ISD [i.e., information systems development] is subject to human whims, talents and the personal goals of the managers, designers and users involved." Already in the year 1985, in a paper with the suggesting title "The poverty of scientism in information systems," Klein and Lyytinen argue that the dominance of the 'methodical' understanding of information systems development is a result of the methodological[6] short-sightedness of the information systems discipline in general. More than 20 years later, perusal of pertinent contemporary literature reveals that despite numerous publications going beyond the confines of scientism, the methodical understanding of information systems development and, with it, the methodological suppositions of scientism are still dominating.

The question regarding the methodical or *a*methodical nature of software engineering or programming or information systems development is, however, just a proxy for more fundamental questions. For example, for Glass et al. (1992) the question regarding the nature of software development is bound to the justification of the very existence of the discipline. Based on a literature review they claim: "There are two divergent views of the complexity of developing software. In the first, software is so mechanistic that it can be automated. In the second, software development is one of the most complex activities undertaken by humans" (Glass et al. 1992, p. 183). Thus, Glass et al. adopt a cognitive-psychological point of view and put the complexity of software development in the centre of their

[5] For fundamental criticisms of the reports by the Standish Group see Glass 2005; Glass 2006; Jørgensen and Moløkken-Østvold 2006.

[6] Given the widespread, but as I believe often inappropriate use of the word "methodology" in the information systems literature, I feel the need to emphasize that here "methodology" is used to refer to a sub-discipline of philosophy of science, i.e., a sub-discipline which is concerned with the study of methods of inquiry.

investigation in which they aim at determining whether software development is largely a clerical or an intellectual task: “By intellectual, we mean that the process requires non-routine thought processes; by clerical, we mean that the process can be accomplished using routine procedures” (Glass et al. 1992, p. 184). Obviously, only when software development is *clerical* it will lend itself easily to the extensive application of methods. However, Glass et al. (1992, p. 189) found that the ratio of *intellectual* vs. *clerical* task during software development is four to one—rendering, again, software development as a mostly *a*methodical effort and thereby justifying the existence of a research discipline dedicated to the study of software development.

Summing up, the critical reflection on the nature of software development resulted in dichotomy of competing conceptualizations and respective conclusions. On the one hand, software development is understood as a number of interrelated but distinct and separable activities that can be generalized and formalized, and thus be supported by methods. It is this understanding that led to the coining of and vision for *software engineering*, a notion that in turn led to the conceptualization of *software factories* (e.g., Akima and Ooi 1989; Cusumano 1989). On the other hand, supported by an ever increasing amount of empirical evidence, software development is understood as a number of interrelated and inseparable activities that require high levels of sophistication and creativity, and can hardly be generalized and supported by or based on methods. This latter understanding emphasizes the role of human capabilities within software development. However, from a philosophical point of view the critical reflections that eventually led to the dichotomy outlined above remained shallow. Nevertheless, the depiction of the critical reflection that led to this dichotomy provides both a historical and a material context giving meaning to Naur’s conceptualization of programming as theory building.

3 TEXT: 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 (Naur 1985, p. 253).

Naur’s conceptualization of programming as theory building is obviously rooted in a critical reflection which led to the recognition that the (then and now) prevailing understanding of programming is flawed. The conceptualization he subsequently develops is remarkable in a number of ways. First, he does not solely focus on the process of programming but takes the outcome of this process into consideration. Second, in line with later critics but in contrast to the proponents of the prevailing understanding of programming, he bases his arguments on empirical findings. Third, he likens program development to the activities of theory building in the sciences.

From a methodological point of view, Naur motivates his inquiry into the nature of software development with a critique in the Kantian tradition. He argues, “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” (Naur 1985, p. 253). 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. The latter insight found its most notable expression in the dictum: “If men define situations as real they are real in their consequences” (Thomas and Thomas 1928, p. 572). As we will see later on, Naur’s critique and subsequent re-conceptualization of software development is substantiated to a large extent by his

practical experiences, thus by empirical evidence. However, the focus of his reasoning is not on the practice of software development. It is on the concept of software development.

Naur uses the questioning of presuppositions for the destruction of this very concept—as it was established through the vision expressed by the term *software engineering* (Naur and Randell 1969). In doing so, he aims at leading the very discipline of software engineering into a crisis—but in a quite a different sense than which has been associated with the discipline ever since its christening (e.g., Naur and Randell 1969; Gibbs 1994): "The real 'movement' of the sciences takes place when their basic concepts undergo a more or less radical revision which is transparent to itself. The level which a science has reached is determined by how far it is capable of a crisis in its basic concepts. In such immanent crises the very relationship between positively investigative inquiry and those things themselves that are under interrogation comes to a point where it begins to totter. […] Basic concepts determine the way in which we get an understanding beforehand of the area of subject-matter underlying all the objects a science takes as its theme, and all positive investigation is guided by this understanding." (Heidegger 1962, pp. 29/30). Heidegger's thought is mirrored in Kuhn's (1962) "Structure of Scientific Revolutions" and both Kuhn and Heidegger seem to be inspired, again, by Kant's (1929) method of critique, i.e., the questioning of the presuppositions that are at the foundation of our knowledge claims.

To do justice to the participants of the NATO conference on software engineering in 1968, I provide the following quote from the respective conference proceedings: "The general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcomings is the primary condition for improvement" (Edsger Dijkstra in Naur and Randell 1969, p. 121). However, even if Dijkstra acknowledged the positive in a crisis, we can say with hindsight that the sole acknowledgement fell short of a catharsis. It appears that only Naur managed to make use of the crisis in Heidegger's (1962) sense by drawing on Kant's (1929) method of questioning presuppositions underlying the discipline's popular conceptualizations of software and software development.

Naur's critique of the then (and still now) prevailing understanding of software development goes beyond the criticism that can be found in later accounts (e.g., Floyd 1992; Glass et al. 1992; Truex et al. 2000; Bansler and Havn 2003): "[M]uch 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 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" (Naur 1985, p. 260).

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. He emphasizes that the 'methodical view' of software development is actually a perversion of ideas developed in the domain of artificial intelligence. In order to simulate so-called intelligent behavior, algorithms had to be devised that could be executed on a computer. Simple-minded people subsequently interpreted these algorithms as representations of mental activity, claiming an algorithmic nature of intelligent behavior. This naïve line of thought can already be found in Taylor's (1911) "Scientific Management". 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.

The "theory building view of programming" was conceived by Naur when contemplating on the role of programmers' knowledge that goes well beyond what is documented by the programs and the respective documentations eventually created. 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).

In his theorizing, Naur draws on the notion of "theory" as conceived by Ryle (1949). With Naur's words: "[A] person who has or possesses a theory in this [i.e., Ryle's] 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" (Naur 1985, p. 255). 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.

Probably more severe than the negligence of certain subject matters under the 'methodical view' may be its ignorance towards any form of knowledge other than the one that builds on the Aristotelian concept of *episteme*—a logically and terminologically elaborated system of situation-invariant (generally) true propositions. The focus on *episteme* in the Western sciences has lead to an unjustified and systematic prioritization of *episteme* and at the same time to a disparagement and exclusion of alternative forms of knowledge. Before the invention of the *episteme*, the ancient Greeks also considered *techne* (the technical know-how enabling to get things done) and *phronesis* (the practical wisdom, drawn from social practices) as forms of knowledge. While *episteme* is not embedded in the everyday practice of action and communication among humans, both *techne* and *phronesis* are (Wyssusek and Totzke 2005).[7] Evidently, Naur's understanding of knowledge (and of theory) goes beyond *episteme*; it includes *techne* and *phronesis* as well. In contrast, meta-theoretical frameworks that exclude the latter two forms of knowledge—such as the artifact-oriented design science frameworks (e.g., March and Smith 1995; Hevner et al. 2004)—cannot accommodate the unique characteristics and challenges of concrete situations in information systems development and elsewhere.

Naur exemplarily illustrates the merits of the theory building view of information systems development by reference to the "life" of a program. He argues that a program "dies" if its programmer team, holding its theory, dissolves. A "[r]evival of a program is the rebuilding of its theory by a new programmer team." Naur emphasizes that such a revival is fraught with difficulties since it is unlikely that a different team will be able to develop a theory identical to the one held by the team that originally 'wrote' the program (Naur 1985, p. 258). With other words, were information systems development methodical and reducible to epistematic 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 epistematic knowledge, every problem occurring while 're-creating' the theory underlying the information system must be due to some error.

[7] While *episteme* is not embedded in the everyday practice of action and communication among humans, it is embedded in the everyday practice of the ones who are concerned with the creation of logically and terminologically elaborated systems of situation-invariant (generally) true propositions, i.e., scientists who have adopted this concept as a description of the ultimate goal of science.

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. 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".

At this point it should have become evident that Naur's notion of "programming as theory building" has implications that go well beyond the scope of information systems development. If—from the perspective of information systems development—the understanding of a program requires developers to create or 're-create' a theory, then what does this mean for the use of the information system by the 'end-user'? 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).

4 CONCLUSION

The philosophical re-appraisal of Naur's "Programming as Theory Building" shows that his work not only provides a philosophically rich source for an argument against the artifact orientation of contemporary meta-theoretical design science frameworks in information systems research. It is also an illustration of how our understanding of both information systems development and information systems is deeply rooted in meta-theoretical presuppositions, thus implicating that any critique of this understanding should be grounded in an analysis of the underlying presuppositions. In the light of the current resurgence of interest in meta-theoretical design science frameworks for information systems research Naur's work provides a challenging alternative calling for rigorous debate.

Naur's implicit criticism of *episteme* as the only form of knowledge accepted by the 'methodical view' on information systems development as well as his illustration of the need for *techne* and *phronesis* provides a starting point for media-philosophical considerations of the different forms of knowledge and their role in the information systems development process. Additionally, Naur's criticism of episteme lends support to the well-established critique of the rationalist understanding of information systems development in the tradition of Descartes.

Not addressed by Naur but already implicated in his work is the need for the conceptualization of information systems use consistent with the conceptualization of information systems development. If we chose to follow Naur, such a conceptualization will bring the active role the 'end-user' in the adoption and subsequent use of information systems to the fore.

Naur's work (and the work of other critics of the rationalist methodical understanding of information systems development) illustrates that (scientific) rigor is no guarantor for universally meaningful results. What makes sense on the basis of one set of meta-theoretical presuppositions does not necessarily make sense on the basis of another, different set. For Naur, the 'methodical view' on information systems development may be based on rigorous research—but it is still wrong and thus irrelevant.

Last but not least, even if the reader does not follow Naur's reasoning, perusal of his work may still be worthwhile since it provides a point of reference that can help to find one's very own position in the field of competing meta-theoretical frameworks. Enjoy!

Acknowledgements

I am grateful to the anonymous reviewers whose comments and suggestions resulted in improvements of this paper.

References

Akima, N. and Ooi, F. (1989). Industrializing software development: a Japanese approach. IEEE Software, 6 (2), 13–21.

Attewell, P. A. (1994). Information technology and the productivity paradox. In Organizational linkages: understanding the productivity paradox (Harris, D. H. Ed.), pp. 13–53, National Academy Press, Washington, D.C.

Bansler, J. and Bødker, K. (1993). A reappraisal of structured analysis: design in an organizational context. ACM Transactions on Information Systems, 11 (2), 165–193.

Bansler, J. P. and Havn, E.C. (2003). Improvisation in action: making sense of IS development in organizations. In Proceedings of Action in Language, Organisations and Information Systems (ALOIS) (Goldkuhl, G., Lind, M. and Ågerfalk, P. J. Eds.), pp. 51–64, Linköping, Sweden.

Baskerville, R., Travis, J. and Truex, D. P. (1992). Systems without method: the impact of new technologies on information systems development projects. In The impact of computer supported technologies in information systems development (Kendall, K. E., Lyytinen, K. and DeGross, J. I. Eds.), pp. 241–260, North-Holland, Amsterdam.

Boehm, B. W. (1981). Software Engineering Economics. Prentice-Hall, Upper Saddle River.

Brynjolfsson, E. (1993). The productivity paradox of information technology. Communications of the ACM, 36 (12), 67–77.

Caws, P. (1967). Scientific method. In The encyclopedia of philosophy (Edwards, P. Ed.), pp. 339–343, Macmillan, New York.

Cusumano, M. A. (1989). The software factory: a historical interpretation. IEEE Software, 6 (2), 23–30.

Descartes, R. (1637). Discours de la méthode. Pour bien conduire sa raison, et chercher la verite dans les sciences. Jan Maire, Leyden.

Eisenberg, M. (2003). A classic problem. The Journal of the Learning Sciences, 12 (3), 445–450.

Floyd, C. (1992). Software development as reality construction. In Software development and reality construction (Floyd, C., Züllinghoven, H., Budde, R. and Keil-Slawik, R. Eds.), pp. 86–100, Springer, Berlin.

Gadamer, H.-G. (1999). Truth and method. Continuum, New York.

Gibbs, W. W. (1994). Software's chronic crisis, Scientific American, 271 (3), 86–95.

Glass, R. L. (2005): IT Failure Rates--70% or 10-15%? IEEE Software, 22(3):112, 110–111.

Glass, R. L. (2006): The Standish report: does it really describe a software crisis? Communications of the ACM, 49(8):15–16.

Glass, R. L., Vessey, I. and Conger, S. A. (1992). Software tasks: intellectual or clerical? Information and Management, 23 (4), 183–191.

Heidegger, M. (1962). Being and time. Harper and Row, New York.

Hevner, A. R., March, S. T., Park, J. and Ram, S. (2004). Design science in information systems research. MIS Quarterly, 28 (1), 75–105.

Hirschheim, R., Klein, H. K. and Lyytinen, K. (1995). Information systems development and data modeling: conceptual and philosophical foundations. Cambridge University Press, Cambridge.

Jørgensen, M., Moløkken-Østvold, K. (2006): How large are software cost overruns? A review of the 1994 Chaos Report. Information and Software Technology, 48(4):297–301.

Kant, I. (1929). Critique of pure reason (N. Kemp Smith, Trans.). Macmillan, London.

Kautz, K. (2004). The enactment of methodology – The case of developing a multimedia information system. In Proceedings of the International Conference on Information Systems, Washington D.C., USA, pp. 671–684.

Klein, H. K. and Lyytinen, K. (1985). The poverty of scientism in information systems. In Research methods in information systems (Mumford, E., Hirschheim, R. A., Fitzgerald, G. and Wood-Harper, T. Eds.), pp. 131–161, Elsevier, Amsterdam.
Kuhn, T. S. (1962). The structure of scientific revolutions. University of Chicago Press, Chicago.
March, S. T. and Smith, G. F. (1995). Design and natural science research on information technology. Decision Support Systems, 15 (4), 251–266.
Naur, P. (1985). Programming as theory building. Microprocessing and Microprogramming, 15, 253–261.
Naur, P. (1993). Understanding Turing's universal machine: personal style in program description, The Computer Journal, 36 (4), 351–372.
Naur, P. and Randell, B. (1969). Software engineering: a report on a conference sponsored by the NATO Science Committee, 7–11 October 1968, Garmisch, NATO Scientific Affairs Division, Brussels.
OASIG (1996): Why do IT projects so often fail. OR Newsletter, 309:12–16.
Oei, J. L. H., Van Hemmen, L. J. G. T., Falkenberg, E. and Brinkkemper, S. (1992): The meta model hierarchy: a framework for information systems concepts and techniques. Technical Report No. 92–17, Department of Information Systems, University of Nijmegen.
Ong, W. J. (1958). Ramus, method, and the decay of dialogue: from the art of discourse to the art of reason. Harvard University Press, Cambridge.
Popper, K. R. (1934). Logik der Forschung. Springer, Wien.
Ryle, G. (1949). The concept of mind. University of Chicago Press, Chicago.
Shaw, M. (1990). Prospects for an engineering discipline of software. IEEE Software, 7 (11), 15–24.
Simon, H. A. (1969). The sciences of the artificial (1st ed.). MIT Press, Cambridge, MA.
Simon, H. A. (1996). The sciences of the artificial (3rd ed.). MIT Press, Cambridge, MA.
Standish Group (1994). The CHAOS Report. http://www.standishgroup.com/chaos/toc.php
Standish Group (2003). CHAOS Chronicles v3. http://www.standishgroup.com/chaos/toc.php
Strassmann, P. A. (1997). Will big spending on computers guarantee profitability? Datamation, 43 (2), 75–85.
Taylor, F. W. (1911). The principles of scientific management. Harper, New York
Thomas, W. I. and Thomas, D. S. (1928). The child in America: behavior problems and programs. Knopf, New York.
Truex, D. P., Baskerville, R. and Travis, J. (2000). Amethodical systems development: the deferred meaning of systems development methods. Accounting, Management & Information Technologies, 10 (1), 53–79.
Turnèbe, A. (1600). De methodo libellus. In: Libelli de vino, calore, et methodo nunc primum editi. folio 2. Morel, Paris
Turner, J. (1987). Understanding the elements of system design. In Critical issues in information systems research (Boland, R. J. and Hirschheim, R. A. Eds.), pp. 97–111, Wiley, Chichester.
Walls, J. G., Widmeyer, G. R., and El Sawy, O. A. (1992). Building an information system design theory for vigilant EIS. Information Systems Research, 3 (1), 36–59.
Weber, R. (1987). Toward a theory of artifacts. A paradigmatic base for information systems research. Journal of Information Systems, 1 (2), 3–19.
Wyssusek, B. (2005). Enterprise system implementation and the linguistic shaping of organizational knowledge. In: Proceedings of the Pacific Asia Conference on Information Systems. Bangkok, Thailand, pp. 402–413.
Wyssusek, B. and Totzke, R. (2004). Data → Information → Knowledge? The perspective of media philosophy on knowledge and its management. In: Proceedings of the 15th Australasian Conference on Information Systems. Hobart, Tasmania.