J Gen Philos Sci (2008) 39:85–113
DOI 10.1007/s10838-008-9068-7

DISCUSSION

Towards a Philosophy of Software Development: 40 Years after the Birth of Software Engineering

Mandy Northover · Derrick G. Kourie · Andrew Boake · Stefan Gruner · Alan Northover

Published online: 6 August 2008

Abstract Over the past four decades, software engineering has emerged as a discipline in its own right, though it has roots both in computer science and in classical engineering. Its philosophical foundations and premises are not yet well understood. In recent times, members of the software engineering community have started to search for such foundations. In particular, the philosophies of Kuhn and Popper have been used by philosophically-minded software engineers in search of a deeper understanding of their discipline. It seems, however, that professional philosophers of science are not yet aware of this new discourse within the field of software engineering. Therefore, this article aims to reflect critically upon recent software engineers' attempts towards a philosophy of software engineering and to introduce our own philosophical thoughts in this context. Finally, we invite the professional philosophers of science to participate in this interesting new discourse.

Keywords Philosophy of technology · Philosophy of software development · Software science versus software engineering · Development process · Change · Evolution · Revolution · Paradigm shift · Popper · Kuhn · Feyerabend

1 Introduction

Software engineering (Sommerville 2007) is still a young discipline—just 40 years went by since its 'declaration of existence' in 1968—and, perhaps for this reason, it has not yet come to the attention of professional philosophers of science. There have been attempts by some software engineers (and also information scientists) to apply philosophical concepts

M. Northover · D. G. Kourie · A. Boake · S. Gruner (✉)
Department of Computer Science, University of Pretoria, Hatfield Campus, Lynnwood Road, 0002 Pretoria, Republic of South Africa
e-mail: sgruner@cs.up.ac.za

A. Northover
Department of English, University of Pretoria, Hatfield Campus, Lynnwood Road, 0002 Pretoria, Republic of South Africa

to their discipline in order to reflect on their activities, but these attempts appear rather limited or unsystematic. Our article aims to apply especially the ideas of Kuhn and Popper to relevant software engineering methodologies—with the School of Frankfurt as a critical corrective—and invites professional philosophers to participate in the debate.

The main emphasis of our work will be on epistemology and scientific methodology, but there will also be a brief exploration of the ontological status of software. However, *ethics* of software engineering, though certainly an important topic as well, is *not* within the scope of this essay.[1] It will be argued that Popper's philosophy is more suitable than Kuhn's to explain the nature and role of contemporary software engineering, in particular the so-called 'Agile' methodologies, in an 'open society' of accountable computer scientists and software engineers.

Computer software, though invisible and immaterial, is pervasive and ubiquitous. Our 'Lebenswelt' has reached a state of development in which, without software engineering, we could not make a phone call, watch television, receive any weather forecast nor fly by plane to a philosophy conference. Furthermore, software is truly global (or universal) and permeates all cultural barriers: oriental theocracies have it and use it as well as occidental democracies, although the cultural purposes and mindsets behind its construction and usage might be completely different from case to case.[2]

The difficulty of our subject, as far as philosophical reflections are concerned, is its hybrid character. Software engineering relates itself methodologically to classical engineering which is based on material sciences. However, software engineering is not equivalent to classical engineering: it does not deal with matter; Newton's law of gravity is not relevant here. Neither is software engineering to be identified with its parent—computer science—and its parent's parent—mathematics—though traces of both can still be found in it. Idiographic and hermeneutic issues play a role in software engineering, too, because software *projects* have unique histories which cannot be repeated, and software *documents* are made of texts and pictures which require human understanding and interpretation; see for example Kroeze (2007). Last but not least social and cultural aspects play a role as well, because software is engineered by people for people according to specifications stipulated by other people—which can lead to conflicts of interests that make our subject accessible through various philosophical concepts (e.g. Habermas' 'erkenntnisleitendes Interesse', or Apelt's 'Transzendentalpragmatik', etc.)

The application of classical philosophy of science to software engineering is made difficult by the fact that software engineering is not primarily aiming at 'Erkenntnis' for its own sake, but at the purposeful construction of usable artefacts, to which 'Erkenntnis' is only a means. Therefore, 'Erkenntnis' in our field seems to be more 'idiographic' than 'nomothetic' (to use Windelband's terminology), which raises the question whether or not the practice of software engineering can be justly called a 'science' at all, though there is no shortage of software engineers and computer scientists who speak—with a mixture of confidence and optimism—about 'software science' instead of 'software engineering' or

[1] We call our article 'essay', in appreciation of Brooks' famous 'Essays on Software Engineering' (Brooks 1995). As far as ethics of software engineering is concerned, we have not yet studied how it would differ specifically from the general ethics of engineering (or technics, or technology) which were already discussed by many contemporary philosophers.

[2] Recently it was reported in the news that Iranian programmers have (allegedly) programmed an Islamist computer game, which mimics the corresponding Western games but with reversed roles: the player of that game will find himself in a scenario in which he has to fight an anti-Western campaign in the role of an ardent religious Jihadist. The game seems to have become quite popular amongst the disgruntled Oriental youths.

'software technology'. Similarly, the application of classical philosophy of technics (or technology, or engineering) also has its limits, as far as the specific differences between software engineering and classical engineering are concerned.[3]

1.1 Conceptual and Terminological Preliminaries

'Evolution' and 'revolution' are two key concepts in this essay that need to be clarified before they are further used. In the usual sense of the word, 'evolution' can be defined as "a gradual development, especially to a more complex form", and 'revolution' as "a far-reaching and drastic change, especially in ideas, methods, etc." (Hanks 1991, pp. 539, 1325).[4] Evolutionary change thus involves small cumulative steps over a long duration, whereas revolutionary change occurs in sudden and radical leaps. With respect to the methodologies of science and software engineering, both these terms are significant. Other relevant notions are those of 'emergent' evolution, "a philosophical doctrine that, in the course of evolution, some entirely new properties, such as life and consciousness, appear at certain critical points, usually because of an unpredictable rearrangement of the already existing properties" (Hanks 1991, p. 509), as well as 'punctuated equilibrium', namely the theory that evolution proceeds mainly in fits and starts, rather than at a constant rate (Bullock and Trombley 1999, p. 771). These latter two terms enable us to refine our understanding of the opposition between the evolution and revolution.

We assume that the reader of our essay is generally familiar with the philosophy of Thomas Kuhn (1962), the philosophy of Karl Popper (1963), some key ideas of the School of Frankfurt, for example Marcuse (1964), as well as with the fundamental problems and main positions of a general philosophy of technics, engineering and technology (Mitcham and Mackey 1973). In-depth knowledge about software engineering, however, is not required, because a sufficient overview of software engineering is given in the subsequent section. For technical details we refer the reader to standard works such as Sommerville (2007).

The following abbreviations will be used throughout this article:

- AM = *Agile Methods, Agile Movement:* A software engineering discipline and community of software engineers which advocates iterative incremental development throughout the life-cycle of a project (Cockburn 2002).
- OOP = *Object-Oriented Programming:* An Aristotelian programming concept based on conceptual hierarchies of classes, super-classes, sub-classes, entities, properties and relationships.
- RUP = *Rational Unified Process:* A particular, iterative software development framework.
- SE = *Software Engineering*, SD = *Software Development:* SE and SD will be regarded as synonyms in this essay, since we are mainly concerned about the software

[3] For a general introduction to the philosophy of technics (technology, engineering), we refer to well-known modern and contemporary philosophers such as H. Beck, N. Berdjajew, F. Dessauer, J. Ellul, A. Gehlen, A. Huning, E. Kapp, H. Lenk, R. Mackey, C. Mitcham (Mitcham and Mackey 1973), F. Rapp, G. Ropohl, H. Sachsse, K. Schilling, H. Stork, K. Tuchel, A. Wenzl, S. Wollgast, W. Zimmerli, B. Zschimmer, and many others. For the difference between 'technics' and 'technology' see Sachsse (1994a, b). A bibliography newer than (Mitcham and Mackey 1973), comprising more than 100 titles on this topic, can be found on the internet at http://www.stefan-gruner.de/Bibl-Phil-Eth-Techn.zip.

[4] We have used a general dictionary here, instead of a specific historic-philosophical one, because specialist dictionaries, in all their 'scholastic' subtlety often obfuscate (rather than clarify) the term in question.

development methodologies in software engineering. (Thus, strictly speaking, SD occurs within SE.)
- UML = *Unified Modeling Language:* A standardized graphical notation for creating abstract models of software systems.
- XP = *Extreme Programming:* A discipline of SD which advocates the adoption of specific maxims, principles and practices and which is fundamentally focused on being adaptable to changing circumstances. XP emphasizes the activity of computer programming itself, rather than the activity of documenting and writing project reports.

1.2 Structure of this Essay

The remainder of this essay is structured as follows: In Sect. 2, we outline some historical aspects of SE, especially as far as the issue of *change* is concerned. The purpose of that section is to serve as a second introductory section for those readers who are not familiar with SE, such that they will be able to understand the philosophy of SE in the subsequent sections. *Related work* by other philosophically-minded software engineers is sketched in Sect. 3. In Sects. 4, 5, and 6 we offer our own philosophical thoughts about SE, whereby we emphasise the issue of change and use the philosophers Kuhn and Popper as our initial (though not necessarily final) landmarks for orientation in this difficult intellectual landscape. Those sections are based on two of our computer science symposium papers (Northover et al. 2006, 2007) which were well received in the related communities. In Sect. 7 we reach some—however preliminary—conclusions, and we invite the professional philosophers to join us in a new interdisciplinary discourse about this relevant, novel topic.

2 Change in Software Engineering and Software Development

To introduce the non-specialist reader to our topic, we shall briefly survey a chain of historical shifts from one type of SD methodology to another. It will be seen that, overall, change has taken place in a way that can be characterised as 'punctuated equilibrium'. These changes in SD methodology are related to—and mutually dependent on—changes in associated 'tools' and artefacts.The introduction of the *compiler* (a particular type of software artefact), for example, had an irreversible impact on the SD process, producing, in turn, better and more complex software artefacts. Attention below is focused on major historical changes in SD methodology, rather than the resulting software artefacts.

2.1 Historic Overview

In the early years of computing, there was no clear notion of SD methodology. Software was developed in a rather ad-hoc fashion, based on the available computer technology. Initially this meant hand-wiring connections, then coding in machine language, later in mnemonic code, and eventually in conceptually 'high-level' programming and specification languages. The main stimulus for adopting some sort of methodological standpoint about how software should be developed was the need to control the ever-growing complexity associated with SD, notwithstanding the continuous improvement of tools and symbolic notations for doing so.

'Software Engineering' as a term, and as a profession, was unknown until 1968 though programmable computers were already known and used in those days. Computer

engineering before 1968 was mainly concerned with hardware development (electronic engineering) on the one hand and conceptually 'low-level' programming (mostly scientific-numeric calculations known as 'number crunching') on the other hand (Methner et al. 1997). The situation changed when computers started to be equipped with what we now call 'operating systems' (which early computers did not have). Operating systems are software systems which can be understood as 'sitting in-between' a computer's hardware and its users' application programs, mediating between both of them, such that the user's application programs can be conveniently specified without tedious reference to the technical details of the underlying computer machinery.[5] Before the invention of operating systems the users' application programs had to run directly 'on hardware', without the mediation of system software. As soon as the advantages of operating systems were discovered there came a growing need for 'bigger' and 'nicer' operating systems, which confronted the makers of operating systems—the system programmers—with ever increasing and complex tasks. At the culmination of this early period of operating system development, operating systems were such complex and cumbersome software systems that even their developers got confused; they were not able to comprehend and maintain their own developments any more. At this point in history it was felt that a less 'home-grown', more systematic and better structured way of SD was urgently needed. From this perspective we can say that SE, as an activity and profession, was born out of operating systems development—which was, in turn, a consequence of ever more powerful and complex computer hardware, which could not have been controlled without the use of operating systems. At about the same time, the term *software crisis* was coined, when, for the first time in history, the *costs* of computer software started to grow above the costs of computer hardware—a trend which has never been reverted since then.

The term *Software Engineering* itself was coined at the NATO Science Conference in Garmisch, 1968, which was attended by many prominent computer scientists of that time (Naur and Randell 1968).[6] That conference was organised in order to address problems such as achieving sufficient reliability in *data* systems (which were becoming increasingly integrated into the central activities of modern society), as well as the notorious difficulties of meeting the schedules and fulfilling the specifications and requirements of large software projects (carried out under the conditions of work-sharing amongst large numbers of specialised and hierarchically organised project members). The 40th anniversary of that event is being commemorated this year.[7]

At a 'Dagstuhl-Seminar' on the history of software engineering (Brennecke and Keil-Slawik 1996),[8] Mary Shaw, a leading researcher in her field, reflected on the use of 'software engineering' as a technical term (Shaw 1996). To a large extent, according to Shaw, the term 'software engineering' is (and has always been) a phrase of enthusiastic aspiration rather than a phrase of sober description: Software practitioners hope to reach the status of an 'engineering' discipline wherein the construction of artefacts of high quality is predictable, reliable, and repeatable. However, an honest appraisal of the field

[5] Thus, operating systems, too, are a manifestation of a general *principle of abstraction*, which is one of the most important methodological guidelines in computer science and SE.

[6] The fact that the conference was organised under the umbrella of the NATO in the middle of the 'Cold War' also indicates the importance of software systems from a *military* perspective—like every novel technology in its own historical time.

[7] See, for example, ICSE Conference 2008, on the internet at http://icse08.upb.de/program/40years.html.

[8] That seminar was attended, inter alia, by Naur and Randall of the 1968 NATO Garmisch event, as well as by 'celebrities' such as Parnas, Boehm (2002), Shapiro, etc.

reveals a practice that is falling short of this ideal (with too many software projects still being late, over budget, and not delivering what was expected). Indeed, the quality difference between classical engineering and SE, which is closely related to Arageorgis and Baltas' (1989, p. 213) discussion of 'craft technology' , is still widely (and sometimes painfully) recognized.

In this historical context, Dijkstra (who had also been at the above-mentioned Garmisch conference) identified a major cause of trouble in form of the *GOTO* commands (Dijkstra 1968) which were widely applied in computer programs in those days. Though Dijkstra's remedy was initially regarded as too radical,[9] the subsequent rapid and universal adoption of 'structured programming' as a SD style had all the hallmarks of a 'revolutionary' change.

As software projects grew in size it was perhaps natural for project managers with classical engineering backgrounds to look to their traditional style of management in an attempt to control the challenging extent of system complexity.[10] This meant viewing a project as a sequence of discrete phases, each of which needed to be articulated, documented, executed and 'signed off' *before* proceeding to the next phase. Thus, once the requirements had been elicited, documented and approved by the client, an analysis phase was entered. Here the task was to determine *beforehand* various modules of software that could be written by different developers. This task was supported technically by advances in the field of programming languages, where 'modularisation' was becoming an ever-stronger theme. For example, the language 'Modula' (Wirth's nomenclature for his programming language that followed on the language 'Pascal') bears testimony to this. Its successor, 'Modula-2', was widely used in the early 1980's, and 'Modula-3' was available in the 1990s; (see Bishop 1991 for a brief history of programming). Further software project phases were to follow the initial phase, including: design, implementation, testing, deployment, and maintenance. This phased approach was eventually termed the 'Waterfall Model'[11] and it hallmarked a change from informally to formally and rigorously managed SD projects. Such a change clearly involved an important conceptual shift, and in this sense, the adoption of the 'Waterfall' model could be seen as 'revolutionary'. However, since there were relatively few large projects in the emerging software industry of the time, the uptake in applying the 'Waterfall' model was gradual. Nevertheless, it became the orthodoxy of the day and also came to be considered mandatory for proper management of large software projects.

By the late 1980s, the inappropriate nature of this rigid 'ceremony-laden' phased SD process had become increasingly apparent, not least because of the large number of unsuccessful software projects (Brooks 1995). Particularly troublesome was the dogmatic requirement of committing to one phase before proceeding to the next. Consequently, there was a strong shift to 'Iterative Incremental Development', as advocated by Boehm (2002),

[9] It provoked, for example, Knuth's response about structured programming with GOTO commands (Knuth 1974).

[10] The website http://www.sereferences.com/software-failure-list.php lists a number of spectacular software failures which are all due to the intrinsic complexity of modern software systems and the limited ability of the human mind to cope with such high levels of structural and behavioural complexity; see also Brooks (1995) for comparison.

[11] The 'Waterfall' development process, adopted by SE from classical engineering disciplines, is nicely summarised and discussed also by Arageorgis and Baltas (1989, pp. 225–226); however they make the mistake of confusing the terms 'efficient' and 'effective': where they say 'effective' they should have said 'efficient'. ('Effective' merely means that a desired result is achieved within a finite—however long—period of time.)

which later evolved into the more elaborate RUP. This shift away from 'big upfront design' relied on object-oriented analysis and design and was, once more, supported by appropriate new programming languages. In other words, the way in which software was modularised was changed by the introduction of OOP. A module no longer represented a set of similar tasks that changed the state of a large number of heterogeneous objects in a domain. Instead, the focus shifted to identifying homogeneous software objects from the same software class, whose state is encapsulated and manipulated by functions particular to that class of objects. Since software objects closely reflect so-called 'real world'-objects in the problem domain, the problem and solution spaces were brought closer together. This shift towards gradual commitment of smaller parts of the project, coupled with a new way of partitioning the software into objects could arguably be classified as 'revolutionary', although the uptake of this approach was also somewhat gradual.

However, some people found even modernised concepts of SD processes, such as RUP, to be too 'heavy' on upfront requirements. Furthermore, these methodologies seemed unable to cope with the challenges of the 'Internet era', including tighter 'time-to-market' constraints, rapid technological progress, as well as increasingly volatile business contexts in a 'globalised' economy. Arguably, all these challenges can be reduced to the *problem of accommodating change.* This provided the impetus for various AM proposals. While these were 'lightweight' as far as formal and so-called 'ceremonial' requirements are concerned, they retained an iterative incremental development nature. For example, instead of relying on the modeling language UML to articulate requirements, emphasis was placed on frequent, informal interaction with the client. Instead of requiring comprehensive documentation of the system's code and architecture, emphasis was placed on co-ownership of program code, and the production of so-called 'self-documenting' program code.

Undoubtedly it was Kent Beck's methodology,[12] XP, which established the popularity of those so-called 'lightweight' SD methods. In February 2001, seventeen representatives of these lightweight SD methodologies met to discuss the similarities of their various approaches. The meeting resulted in an agreement to adopt the term 'agile' instead of 'lightweight' and to issue their 'Manifesto for Agile Software Development' (Beck and Fowler 2001), which includes a statement of the following four central values underlying the AM approach:

(i) *individuals and interactions* over processes and tools,
(ii) *working software* over comprehensive documentation,
(iii) *customer collaboration* over contract negotiation, and
(iv) *responding to change* over following a plan.[13]

Later, in order to support these four 'values' (or maxims), a further dozen principles were formulated. All this culminated in the formation of the 'Agile Alliance', a non-profit organisation whose stated 'mission' it was to promote these principles and values.

AM are therefore, in one sense, an 'evolutionary' offshoot of iterative incremental SD processes. However, in another sense, they represent a radical, 'revolutionary', departure from all previous SD processes, in that they spurn an 'ideological' commitment to the

[12] German-speaking readers should keep in mind that the English word 'methodology' is conceptually closer to the German words 'Methode' or 'Methodik' rather than to the German 'Methodologie'; it means a *framework of related methods* rather than a fully elaborated science or theory of methods: see Geldsetzer (1980) for comparison.

[13] There are, of course, ad-hoc 'micro-plans' for "responding to change". The process is not supposed to be chaotic.

velocity of software production, whilst at the same time taking cognisance of the human needs of the members of the 'agile' development team. Cockburn, for example, is one of a growing number of AM advocates who would eschew an upfront commitment to any particular SD process, albeit an AM one. Instead, he recommends that the SD process be tailored specifically to the demands of each individual project (Cockburn 1999).

2.2 What is Software?

In the historic overview of above we sketched the various ways in which software has been developed throughout the recent four decades. However, we did not yet speak about *what* software actually *is*, or how software can be characterised. Therefore we will round up this section with a few remarks in this regard. Unlike the classical sciences, SE does not produce theories in order to explain aspects of nature, but rather, like classical engineering, it produces artefacts to serve practical ends. However, unlike classical engineering, software engineering does not produce physical structures, hence our need to understand the ontological status of software as the product of SE.

The term 'software architecture' is often used to describe the structural aspects of software, relating it not only to classical engineering (the building of complex physical structures) but also to the arts (to which architecture is related). The transfer of structural knowledge ('design patterns') from the building architect Christopher Alexander into the software architecture community is widely acknowledged (Alexander 1999).[14] An important difference, however, is the fact that, whereas buildings can be more or less permanent and complete, software structures tend to be more prone to change and incompleteness. It was mentioned above that engineering produces artefacts. Etymologically, 'art' and 'artefact' share the Latin root *ars*, meaning craftsmanship or skill. The second part of 'artefact' is from the Latin *facere*, meaning 'to make'. Thus software and SD share features of the sciences, engineering and the arts; see Bishop (1991) for comparison.

Despite the term 'software architecture', software—computer programs, or 'code'—more closely resembles the *temporal* arts (music, literature) than the plastic arts (architecture, sculpture, painting, printing). Software code can be compared to the score (or notation) of a symphony or to the text of a novel or poem, which have to be interpreted and performed in time. We speak of 'programming languages', all of which have to be 'interpreted' by compilers until they are translated into the Zeros and Ones enacted at the most basic level of machine language. A programming language resembles the rules of inference of formal logic, although, unlike an argument, a program serves a practical end. Each time a program is run (executed) it can be considered a performance. Thus software is not solely to be identified with its code, but must also be regarded with respect to the performance (execution) of the code, and successful code is judged on its actual performance. Just as all artworks are physically embodied, but not identical with the physical body, so too is all software embodied in but not identical with hardware.

Nonetheless, software differs in its nature and function from art. Whereas art might be considered an end in itself, software is a means to some further end, hence its status as a type of engineering. Also, artworks often have an expressive or symbolic or representative function whereby the artwork expresses an emotion, or stands for something other than itself, or provides an interpretation of some aspect of reality. In this latter function,

[14] For this reason a philosophical analysis of SE might even be interesting for philosophers in the tradition of *Structuralism*.

artworks might resemble 'theories', whereas software serves a practical, non-interpretive function. Finally, unlike most artworks, software programs are almost never complete but rather are subject to continuous modifications and maintenance. From this perspective, software artefacts might perhaps better be named processes, not only in terms of their continual development by programmers, but also in terms of their ongoing application by users.

Another important point about software is that it comes in different *versions* and is distributed to the end-users in plenty of identical *copies*. Also from this perspective—and not only from the perspective of creation or creativity—software has something in common with artworks such as books, printings, or musical performances. For this reason, it might be useful to have a look at the philosophy of arts, in order to gain deeper insights into the 'essence' or 'nature' of software, by comparison. The arts-philosopher Margolis, for example, argued that artworks cannot be universals, since artworks are created and can be destroyed, whereas universals cannot. Margolis referred to Glickman's conjecture that "particulars are made, types created" and assumed that all artworks are tokens of a type, the tokens being physically embodied and the types existing only in the form of its tokens (Margolis 1980, p. 17). Types are abstract particulars, and the type-token concept is distinguished from the kind-instance and set-member concepts.[15] For example, every performance of a symphony would be a token of that type, but in this case a temporal rather than spatial token. By analogy, software (programs) could be considered to be tokens of a type existing in a temporal and sequential mode like the performance of a symphony or the reading of a novel. A further elaboration of such an ontology involves the terms 'prime instance' and 'megatype', whereby "two tokens belong to the same megatype if and only if they approximately share some design from the range of alternative, and even contrary, designs that may be defensibly imputed to each" (Margolis 1980, p. 54). In art, the prime instance would be the original painting or manuscript of a poem or novel. All other tokens of that type would be variations or copies, including non-identical copies, of the megatype of which the original poem is the prime instance. Other art forms, such as printings of etchings will have no prime instance at all, while an original painting would be a prime instance and all printings or copies of it would be tokens of the megatype. This provides us with a terminology that allows us to identify different versions of an artwork as tokens of the same megatype, for instance, the different performances of a theatre play in different periods, cultures and languages, and even in different *media* such as different movie (film) adaptations of a theatre play for cinema. In terms of software engineering, this terminology could be helpful to describe the different versions of applications (programs) that are continuously released. Each new version of a piece of software which is released to a customer, for example some new photo-software, would be a token of the megatype 'photo-software', whereas each individual use (performance) of *this* specific copy or instance (probably with its own serial-number on the box in which it is sold) of the photo-software would be a token of this specific type.

Mathematical Platonists like Penrose have located *algorithms* (which are, metaphorically speaking, the 'soul' of a computer program) in a Platonic realm of eternal forms (Penrose 1989). We would locate software programs not in such a static Platonic realm but rather in Popper's dynamic 'World (iii)', the realm of 'objective knowledge' which is inhabited by scientific and metaphysical theories. Popper was willing to locate artworks in his 'World (iii)', but was equally happy to call the realm reserved for art "World (iv)"

[15] Margolis acknowledges his indebtedness to Peirce for the type-token-distinction, although he also notes that he departs from Peirce's usage who perceived types and tokens exclusively as signs.

(Popper 1999, p. 25). His point was that theories and artworks are not merely subjective experiences in the minds of their creators and contemplators, but have an objective and non-physical existence independent of individual minds.[16] Unlike Plato's realm of pure forms, however, Popper's 'World (iii)' is regarded as open to change, in the sense of an evolutionary development towards increasing complexity.

In their article on the distinction between science and technology, Arageorgis and Baltas have spoken about how "theoretical models (…) tend to bridge the gap between what a scientific theory accounts for, and what a particular technology aims to bring about" (Arageorgis and Baltas 1989, p. 214). In this context it is interesting to note that software can also play the role of such 'models'. Classically, science had known basically two methods of enquiry, namely the rationalist 'Gedankenexperiment', and the empiricist 'real' experiment. The advent of computer technology has brought us a new, *third* method of scientific enquiry, in the form of computer *simulations*, including both rationalist and empiricist aspects (namely: model development and programming, and runtime observation), from which a whole new branch of science, namely 'simulation theory', has emerged. 'Executable models' (in form of software) are thus crucial epistemological entities in the ontology of those new sciences (Vangheluwe 2008).

Finally, at the end of this sub-section, we also want to mention that software can be of 'Zeug' character in the sense of Martin Heidegger's 'fundamental ontology' (Heidegger 1927). However, it is only through the *mediation* of another 'Zeug', namely the computer underneath, that software can reveal its character as 'Zeug'. Whereas a simple material tool, for example a hammer, can be immediately grabbed by a human hand in a naive, pre-technological 'Lebenswelt' (as phenomenologically analysed by Husserl, Heidegger and followers), no such direct (manual) access to software is possible. Without the computer as underlying platform, the usage of software as 'Zeug' is impossible. Nevertheless the modern software engineer speaks about 'software tools' in the same attitude in which a naive handyman or farmer would speak about his simple material tools. In this regard we may speak of software as of 'abstract Zeug', or 'virtual Zeug', or 'Zeug of second degree'. This, again, corresponds well with Heidegger's notion of a 'reference property' ('Verweisungs-Charakter') of 'Zeug', in which (for example) a hammer refers to the nails, and so on. Software without a computer is as useless as a computer without software. In other words, the 'Verweisungs-Charakter' is evident even for this 'abstract Zeug' in the form of immaterial software and software systems.

3 Related Work

As this essay puts strong emphasis on the question of *change* in software engineering, we focus our literature review especially on those authors who also deal with this question. Our literature review has two aspects: a *technical* one and a *philosophical* one:

- Our technical review is only brief. Its main purpose is to provide some orientation for those readers who are familiar with the history and philosophy of classical engineering so they may find it easier to relate our new topic of software engineering to their existing knowledge about classical engineering and technology.
- Our review of the philosophy of software engineering constitutes the larger part of this section. Its main purpose is to introduce the professional philosopher to the ideas and

[16] This is related to questions of organisational knowledge, upon which we shall touch only briefly in Sect. 6.

thoughts that have been produced by philosophically-minded members of the SE community, so that the professional philosopher may be encouraged to participate in this interdisciplinary discourse.

Much current debate in the field of software engineering methodology concerns the question whether to use traditional SD methods or AM. Members of the traditionalist camp prefer to follow a strictly pre-defined development procedure, whereas members of the AM camp, possibly with sympathy even for the ideas of Feyerabend,[17] would prefer to act as independently as possible and react ad-hoc to events and circumstances as they emerge. From a social-philosophical viewpoint (which shall soon lead us to thinkers like Kuhn and the School of Frankfurt), it is interesting to note that membership to the one or the other methodological camp is not only a question of knowledge or reasonable insight, but also a question of authority and power.[18] For the historian and philosopher of technology it will be interesting to note that also in the field of classical mechanical engineering—long before software engineering came into existence—there were examples of AM. In the field of classical engineering, this is called 'concurrent engineering', as explained by Robert Smith (1997).[19]

Now let us now turn our attention from the technical to the more philosophical part of our literature review. In this part, we shall review how philosophically-minded members of the SE community have applied the theories or thoughts of various professional philosophers to the software engineering discipline. Firstly, we consider some who have applied the work of Plato, Aristotle, Lakatos, Feyerabend and Dooyeweerd. Secondly, we focus on the literature that specifically applies the theories of Kuhn and Popper to software engineering and especially to AM.[20]

Marick argues that the decision to discard an existing software methodology in favour of a new one requires a "*gestalt* switch" from one ontology to another (Marick 2004). In particular, he discusses the AM ontology and turns to science in an attempt to understand why so many software engineers have adopted this particular ontology. He conjectures that Kuhn's concept of paradigm shift may provide one possible explanation. However, Marick believes that the theory of Lakatos provides an even better understanding of this switch than Kuhn's or Popper's theories do. Marick describes the four central aspects of Lakatos' methodology of scientific research programmes and finds each of these aspects to be evident in the AM ontology in some form. The author takes an anti-rational stance by emphasising perception and practice rather than reflection in relation to ontology. The paper concludes that there is no "platonic right way to build software" (Marick 2004, p. 71) and that for a methodology to be progressive, one must find the correct ontology.[21]

[17] For a brief overview of Feyerabendianism in SE see Gruner (2007). For a more profound critique of Feyerabendianism in SE see Snelting (1997, 1998). Another example of Feyerabendianism in computer science can be found on the internet at http://www.dreamsongs.com/Feyerabend/ETAPS03/.

[18] Our colleague, Morkel Theunissen, hinted in a private communication at the powerful role of the American *DARPA* doctrine in the widespread adoption of the strictly hierarchy-procedural development process *both* in classical-mechanical *and* in SE.

[19] This 'rapid' engineering method was ignored the methodological discussion of Arageorgis and Baltas (1989), presumably because this detail was not central to their general 'science versus engineering' theme.

[20] This entire synopsis is intentionally limited to literature that explicitly applies the theories of professional philosophers and excludes any literature that uses the word 'philosophy' loosely in the senses of 'guiding principle' or 'general outlook on life'.

[21] 'Ontology' in SE does not have the same meaning as in philosophy. An 'ontology' in SE is, basically, a 'name space'.

Guigette provides a compelling argument for adopting a Platonic view of objects in an OOP language. The class to which an object belongs is compared to a 'Platonic Form' because it generalises all possible class instances (Giguette 2006).[22] Rayside and Campbell argue in a similar direction and claim that the object-oriented community at times maintains that OOP has drawn inspiration from philosophy, specifically that of Aristotle (Rayside and Campbell 2000). That paper goes into great detail applying Aristotle's system of syllogistic logic to OOP and points out the similarity, on the one hand, between Aristotle's concept of matter and OOP's notion of 'object' and, on the other hand, Aristotle's concept of form and OOP's notion of 'class'.

Furthermore, Brooks, a well-known software engineer and computer scientist, acknowledges Aristotle and uses the Aristotelian terms 'essence' and 'accident' in distinguishing the essential difficulties—those inherent in the nature of software, like complexity—from the accidental difficulties like a poor design—those that attend its production but are not inherently problematic (Brooks 1987).

Basden uses Dooyeweerd's philosophy to describe the development and application of information systems. He has recently published a monograph on the five main areas of research and practice in this SE-related field: the nature of computers and information, the creation of information technologies, the development of artefacts for human use, the usage of information systems and information technology as our environment (Basden 2008). However, the theme of his book is only marginally relevant to our specific SE discussion in this essay.

Kuhn's concepts of paradigm shift and scientific revolution have been applied to diverse aspects of the software engineering discipline, most notably by several members of an international IT advisory firm, the Cutter Consortium. These applications are especially pertinent, since many leaders of the AM in SD are members of this consortium. Below we summarise the literature in this regard.[23]

Beck, the founder of XP, explicitly cites Kuhn's 'Structure of Scientific Revolutions' in the annotated bibliography of his own book on XP (Beck and Fowler 2005). Moreover, Beck acknowledged that Kuhn's book was influential on his thinking and that he found it useful mostly in the adoption process; he stated that Kuhn's book would have helped him to predict "how the market would react"[24] to the introduction of XP.

Bach applies the work of Popper, Kuhn and several other philosophers of science to software engineering in the context of the 'process versus practice' debate (Bach 2000). He argues that, in the 20th century, even science was a battleground for this debate when the very idea of science as a rational enterprise came under fire. The resulting fallible view of science, according to Bach, is pertinent to our situation in information technology today. Bach also argues, perhaps controversially, that developing software is not much different from developing scientific theory, and developing processes for developing software is exactly like doing science. Finally, he argues that the trend to move from formalised SD processes towards more intuitive practices manifests itself not only in AM but also in the object-oriented 'design pattern' approach, which was an adoption of Alexander's ideas on housing architecture and civil engineering into the domain of SD; see Alexander (1999) for comparison.

[22] In our opinion, OOP should be better characterized as 'Aristotelian' rather than 'Platonic', but this discussion would lead us too far away from the central topic of our paper; see Rayside and Campbell (2000) for comparison.

[23] We cite several authors whose writings have been published in the Cutter Consortium's own book series. All of them can be found in the Cutter Consortium's own bookstore (Cutter 2007).

[24] K. Beck: Personal e-mail communication with M. Northover, December 2006.

Schwaber, another Cutter member, does not explicitly cite Kuhn although he frequently uses Kuhnian terminology in his writings. He describes the seminal meeting of the AM advocates in 2001 as a meeting of revolutionaries. However, Schwaber is using the term 'revolution' here in a general political sense rather than in the specific notion of Kuhn's theory of science (Schwaber 2001).

Marzolf and Guttman (2002) use Kuhn's theory to explain an approach called 'systems thinking' and how it relates to SD. They point out that, although systems thinking insists that systems should be addressed holistically, the most universal characteristic of SD is, rather, a short-term and piecemeal approach (Marzolf and Guttman 2002). The authors conclude that we should learn systems thinking and take a more holistic approach towards building software. This, in their opinion, will ensure the shift from the 'machine age paradigm' of the 19th century to the 'systems age paradigm' of nowadays.

Davies cited Kuhn's book in a conference address and claimed that understanding AM would require a paradigm shift (Davies 2006). Yourdon, like Davies, references Kuhn's book explicitly and applies his concept of paradigm shift to information technology organisations in the context of ad-hoc communication networks (Yourdon 2001a). In a related paper, Yourdon (2001b) applies Kuhnian terminology specifically to XP by describing classical SE as the old paradigm and AM as the new paradigm. In that paper, he also quotes Beck's presentation at the 2001 'Cutter Summit' Conference in which Beck had explicitly called XP a 'paradigm shift'. We shall come back to Yourdon's ideas in somewhat greater detail below.

From the selected references above it should be evident that Kuhn's philosophy has had a significant influence on the SE discipline, especially within the AM community. However, most of the authors have applied Kuhn's concepts rather uncritically to this domain. Consequently, one of the central aims of our essay is to provide a seemingly much needed critical perspective on the applicability of Kuhn's philosophy to the software discipline, and specifically to XP. This critique will follow in Sect. 5 of our essay. In the next few paragraphs, however, we will highlight some literature references that specifically apply Popper's philosophy to the SE discipline.

Snelting, another significant software engineer, has pointed out that in software technology sometimes things happen quite similarly to the way Feyerabend has described them (Snelting 1997, 1998). In particular, he notes that empirical studies are rare in SE, that there exists a chasm between theory and practice and that a particular form of 'constructivism' is rife amongst practical scientists who ignore theory and at the same time avoid empirical validation. He argues for a stronger empirical foundation of SE. In order to achieve this, he insists that basic scientific principles should not be neglected. Regarding that particular form of (social) constructivism, he points out that SE constructs its own reality since software engineers invent abstract concepts or devices, such as abstract data types, software architectures and design patterns. For this reason, he states, it is popular to call computer science a 'structural science', however, recently computer science is seen more as an engineering science. Snelting goes on to ask what the computer science equivalent of predictions and falsifying experiments are. He states that predictions in computer science are generally obtained from abstractions and theories. More recently, however, there have also been methods of prediction, such as the application of model-checking.[25] Experiments, on the other hand, are most importantly found in the form of software testing, although testing, like all experiments, can only be used to refute a specific prediction. Snelting concludes that the

[25] Model checking is a method of testing the possible behaviour of a finite-state machine against a formal specification in a temporal logic, to generate counterexamples or refutations if the specification is violated.

Popperian concept of falsifiability is as valid in SE as it is in the natural sciences, but is often ignored in SE. Furthermore, Snelting observes that the pioneering computer scientist Dijkstra's well-known dictum (that testing can only demonstrate the presence of defects, but not their absence) is a special case of Popper's falsifiability principle.

Another pioneer of computer science and SE, Hoare, made a similar observation (Hoare 2003a). Hoare pointed out that, following Popper's criterion of falsification for the meaning of a scientific theory, Roscoe and Brookes would have concentrated on failures of tests, with particular attention to the circumstances in which programs could deadlock or fail to terminate. This, according to Hoare, led to the now standard model of CSP (Hoare 1985, 2006; Roscoe 1997, 2005), with traces, refusals, and divergences (Hoare 2003b). Following Hoare, Aichernig pointed out that falsification can be applied to software development in the following way (Aichernig 2001): A formal specification of requirements helps to solve systematically the problem of software development by offering the apparatus of logic. The validation of the formal specification as well as the implemented solution is only feasible through falsification—which is testing. For being falsifiable (or testable), the requirements description has to be unambiguous and sound—ensured by formal specification and verification techniques.

Coutts uses many Popperian ideas to trace the similarities between software testing and the scientific method (Coutts 2007). He concludes that falsification is just as essential to software development as it is to scientific development. However, whereas in science the falsification criterion is used to demarcate science from non-science, in SE it is used to demarcate the testing discipline from the analysis and development disciplines.

Meyer (2007) stated that there is a view of science, first proposed by Popper, that any true science must contain open and testable claims. According to Meyer, Lakatos has shown that this same testability applies to mathematics in the form of methods (approaches) which are tested by evaluating their problem solving success. Meyer's intention in pointing out these two claims is to encourage SE work that will eventually lead to such scientific tests in computer program validation (Meyer 2007).

Hamlet contended in a conference address that "computer science is *not* science" (Hamlet 2002)—in opposition to 'mainstream' opinions like those expressed by Bishop (1991). Hamlet claimed, using Popperian terminology, that there are no falsifying experiments in computer science. Instead, according to him, to experiment in computer science means to implement an idea and force it to work, which is in contrast to science, since scientists cannot change reality to fit a theory. He contended further that a fundamental understanding of philosophy would help us to deal with the difficulties of SE, like unrealistic schedules and changing requirements. He discusses both Kuhn and Popper in this context.

Popper's ideas, especially his 'three worlds' ontology, have also been used in the context of 'knowledge management', which is related to SE from an organisational point of view. This is due to the fact that software is developed not so much by individuals rather than by groups of developers, who have to share their knowledge (about SD processes and products) in implicit (informal) or explicit (formal) ways. Three well-known Popperians in this SE-related field of study are Hall, Moss, and McElroy (Hall 2003; Moss 2003; McElroy 2002; Firestone and McElroy 2003), but an in-depth discussion of their work would lead us too far away from the main SE theme of this article.

Finally we should mention the work of Gregg et al. (2001) which tries to see "software engineering as a research paradigm" (Gregg et al. 2001, p. 171) in the light of the social sciences and their particular research methodologies, classified as "constructivist", "interpretative", "positivist" (Gregg et al. 2001, p. 172), and so on. "The impetus of our effort resulted from the inability to fit software engineering research comfortably into the

established research paradigms from the social sciences" (Gregg et al. 2001, p. 181). Such a sociologist perspective of SE—see also Sect. 4 below—may be partially justified by the social role of software, being built *by* people *for* people, in a modern society. However it should not be forgotten that a sociologist notion of 'research' might differ quite strongly from a scientist's or engineer's concept of 'research', and that a sociologist account of SE is thus unlikely to provide reliable insights into what really happens in this technological discipline. Nevertheless it is interesting to have a look at Gregg et al. (2001), not least for their informative bibliography on the philosophical foundations of information systems. An accurate account and classification of empirical research methods of SE has been provided from *within* the discipline by Walter Tichy (2007).

Concluding this related work overview we might conjecture that the professional philosophers amongst the readers of this essay will notice that the range of philosophers cited by those philosophically minded software engineers is rather limited, which seems to be a consequence of academic specialisation and the notorious 'knowledge gap' between the professions of engineering and the professions of the humanities.[26] Therefore we presume that many other relevant philosophers still have to be 'discovered' for SE. It is for this reason that we are writing our essay as a 'call for cooperation' between software engineers and professional philosophers in this new discourse towards a philosophy of SE and SD.

4 The School of Frankfurt

Whereas Snelting has expressed concern about tendencies of 'Feyerabendianism' in academic SE research (Snelting 1997, 1998), SE-related publications like Lyytinen and Klein (1985) or Gregg et al. (2001) reveal that 'critical theory' (Simon-Schaefer 1994), too, has been recognised—and partly even adopted as theoretical and practical guideline—in some socially oriented circles of the wider information technology (IT) and information systems (IS) community, to which the SE sub-community belongs. As indicated by papers such as Lyytinen and Klein (1985) or Kroeze (2007), there exist members of the IS community who regard IS as a 'social science' and their own work as emancipative in the sense of 'critical theory'. Several decades after the pointless 'Positivismusstreit' (Adorno et al. 1993) has ebbed down the outdated term 'positivist' (Schnädelbach 1994) is still in use with the aim of drawing a line between IS and the so-called "harder sciences" (Kroeze 2007, p. 38). For this reason—before coming back to Kuhn's philosophy in the SE context in Sect. 5—we shall briefly consider the School of Frankfurt in this section of our essay. Unlike Kuhn and Popper, leading members of this school had criticised science in total as a system of 'ideology' (Becker 1994), and have confronted Popper (as well as other critical rationalists) directly in a notorious debate on these issues (Adorno et al. 1993).

From the 1940s to the 1980s, the School of Frankfurt was a dominant participant in debates on the nature of social systems. Philosophers of that school had a dialectical notion of how social or technical systems change: This view included social subsystems, such as those relating to science and technology. As far as the theme of this essay is concerned, we read the Frankfurtians against their own intentions and regard a software system itself as a 'global' system. In this sense, a Frankfurtian analysis becomes (at least partly) applicable.

In the view of the School, a social or technical system, including a software system, is likely to appear (or become) globally irrational.[27] The roots of this global irrationality lie in

[26] This includes ourselves, the authors of this essay; none of us is a professional philosopher.

[27] Here we refer particularly to some notoriously anti-technological paragraphs in Marcuse (1964).

what might seem to be locally rational. Such points of local pseudo-rationality should be uncovered by critical analysis, and the entire global system has to change if it is to improve. A revolution would then be needed to achieve the desirable global state. As a corollary, it could be said that partial or small-scale rationalism within the parts of the system does not prevent its global irrationality. Thus, attempts at small-scale criticism or small-scale modifications from within the system would be seen as ineffective and reactionary, since they are absorbed by the system due to its self-stabilising conservative tendencies. The discrepancy between such a perspective and the already mentioned AM perspective in recent SD should be clear.

Such sources of irrationalism, in any such system, are due to driving forces in processes that are characterised by conflicting particular 'interests' (Lobkowicz 1994) which are supported by a corresponding 'ideology' (Becker 1994). Even the acquisition of 'pure' scientific knowledge and the development of 'neutral' technology is driven by such interests. Together with a corresponding ideology, technics and techniques form the system of 'technology', thus: technology is technics plus ideology.[28] In this context, *change* is analysed by the School of Frankfurt in dialectic terms, thus in the Hegelian/Marxist figure of thesis, anti-thesis and synthesis. In the SE context of our essay those terms could also be applied: For example, a client's specifications would be a 'thesis', the software engineer's system proposal would be an 'anti-thesis', and the emerging software product would be a 'synthesis'. Thereby, the so-called 'contradictions' ('Widersprüche') between thesis and anti-thesis are not merely mental concepts of the observer (as a matter of epistemology), but are regarded as objective realities caused by objective antagonistic forces and tendencies (as a matter of ontology). Also note in this context that the synthesis, though emerging out of thesis and anti-thesis, cannot be deterministically predicted out of them, therefore, the path towards progress of technology cannot be foreseen until it actually takes place.

Followers of the School of Frankfurt would therefore understand some of the changes in software methodologies in the following terms: The classical, top-down 'Waterfall' model of SD would be seen as globally irrational, despite the locally rational nature of completing one phase before embarking on another, of maintaining a paper trail of what has been done (documentation), of forbidding constant changes to the requirements and so on. That was indeed suited to middle managers, who needed to maintain their positions of authority and control, and to offer their seniors evidence (in the form of signed-off documents) of progress. Notwithstanding the high failure rate of projects managed in this fashion, conservative forces have ideologically clung to the model until its internal contradictions became intolerable. Only then were people willing to change to the iterative incremental approach, which became the new orthodoxy. The School would thus also hold that a system as a whole should be criticised from a hypothetical *outside* perspective. Attempts at small-scale criticism (or small-scale modifications) from within the system would be absorbed by the system. An example of how this perspective maps to SD is that, despite all small-scale software maintenance efforts, a legacy software system will typically grow increasingly obsolescent. The case for consequently discarding it completely and replacing it happens against conservative forces who try to protect their financial investments in the old system, their prestigious status as experts in the old framework, and the like.

The purpose of this brief excursion into the world of the School of Frankfurt was to provide an alternative and critical view on change and progress in science and society from

[28] Note that similar thoughts are evident in the later works by M. Heidegger—technics as 'Gestell' (Heidegger 1949)—in spite of the School of Frankfurt regarding itself as notoriously anti-Heideggerian.

that of Popper and Kuhn. As far as Kuhn is concerned, it seems fair to say that the School of Frankfurt had emphasized earlier (and more strongly) than Kuhn that the historic occurrence of paradigm shifts is not only driven by flaws and deficiencies in those paradigms themselves, but also by strong social forces with specific agendas. In the world of SE, such conflicting interests might be identified with the various stakeholders of a software development project, such as customers, project managers, programmers, and so on. Finally, we might say that, to the followers of the School of Frankfurt, any philosophy that condoned a piecemeal, evolutionary approach to change seemed superficial and reactionary. It is therefore not surprising that they repudiated the ideas of Popper who offered a contrary perspective of change, as discussed further in Sect. 6 below.

5 Appreciation of Kuhn

"Can Thomas Kuhn's paradigms help us to understand software engineering?" (Wernick and Hall 2004). As usual in philosophy, the answer is neither an unconditional 'yes', nor an unconditional 'no'. Our discussion especially of Yourdon in the literature review above (Sect. 3) led us to reflect upon our own understanding of Kuhn's philosophy with respect to software engineering. This section will assess the extent to which Kuhn's popularity in parts of the software engineering community is justified. For this purpose, a distinction will be made between *large-scale* and *small-scale* change that both occur in the SE domain. We will argue throughout that Kuhn's concepts of 'scientific revolutions' and 'paradigm shift' seem more suited to explaining large-scale change in the field of SE as a whole, whereas Popper's concepts of 'evolutionary epistemology' and 'falsificationism' seem more suited to explaining small-scale change in the course of particular SD projects.

Applying Kuhnian terminology to SE naively, we could thus consider AM and XP to be a new paradigm of SE, whereas 'Waterfall' or other traditional SE methodologies would be considered part of the old paradigm. Therefore, we must now assess if Kuhn's theory adequately accounts for the change from classical SD to AM, and if this change can be best described as a revolution or paradigm shift in the Kuhnian sense.

One could ask, for example, whether software developers could be rightfully described as 'scientific researchers' at all. To us it seems obvious that software programmers are engineers of sorts, since they produce something that has direct application to the ordinary world, whereas physical scientists are usually concerned with specialised research that often has no obvious bearing on everyday life. In other words, the engineer synthesises (constructs) to produce new artefacts, whereas the scientist analyses and takes apart to acquire knowledge about an existing (natural) entity. Furthermore, scientists are not normally held accountable to the public to the same extent as engineers who do not merely solve theoretical problems but produce software with clear applications. Nonetheless, both are making use of the rigorous standards of modern mathematical logic, and computer technology has become essential to any modern scientific research as well.

Moreover, may SD methodologies rightfully be regarded as 'paradigms'? We concede that software practice can indeed be made to fit both of Kuhn's definitions of a paradigm, at least in the broadest senses. However, what is not evident from Kuhn's definition is that he was writing specifically about *scientific* communities in their particular historic form. Taken in this specific and historic sense, it is questionable whether the term can be re-applied to software-*engineering* communities without any modification of its meaning (though Kuhn himself has indicated in his 1969 'Postscript' that his theses were compiled by taking several fields of research into account).

The proliferation of various SD methodologies since the early 1990s has been interpreted by some AM followers in terms of Kuhn's 'scientific crisis' or 'extra-ordinary science'. AM are seen as the emerging paradigm that might completely replace the traditional methodologies of the old paradigm. The questioning of the fundamental principles, values and practices of SD methodology that accompanies the emergence of new methodologies, compares with the similar activity of physical scientists in a state of crisis. This presupposes that SE could already be regarded a 'mature science' dominated by a single paradigm, as opposed to a 'pre-scientific' state of affairs which is characterised by many competing schools of thought. This does, indeed, seem to be the case in software engineering since, before the present crisis, the 'Waterfall' SD method was dominant. However, one should question the assumption that there needs to be a single dominant paradigm in SE as Kuhn argues there is in science. Cockburn's suggestion of one SD method per SD project (Cockburn 1999) seems reasonable, though it is a thoroughly non-Kuhnian approach since it implies that rational choices can be made between different SD methodologies.

Furthermore, are SD methodologies really incommensurable and an all-or-nothing affair, as Kuhn alleges scientific paradigms to be? According to many articles in the software literature that compare methodologies, such a claim is rather unfounded. Boehm, for example, makes a detailed comparison between AM and plan-driven SD methods and argues for their synthesis into a hybrid one (Boehm 2002). Similarly, other authors claim to have successfully adapted the 'Waterfall' model by integrating key elements of SD approaches such as 'Rapid Application Development' and XP (Lux 2007).

Another important question in this context is: 'Was the emergence of AM in the mid 1990s a result of *cumulative anomalies?* If not, what triggered the *crisis?*' For Kuhn, anomalies could take the form of "discoveries, or novelties of fact" on the one hand or "inventions, or novelties of theories" (Kuhn 1962, p. 52) on the other hand. Both forms of novelty however are usually not actively sought out (initially even resisted) by scientific communities. Since SD methodologies do not aim to explain physical phenomena, it is difficult to see how Kuhn's theories can be applicable in this case. Nonetheless, a case can be made for crucial events causing a *crisis* in SE, for example, the advent of the Internet-era: document-oriented SD methodologies were the dominant paradigm at the time and most document-oriented software engineers came to recognise that developing at 'Internet time' required a reconsideration of the document-oriented SD process. As a result, AM emerged and many advocates of the old paradigm became the advocates of AM. Also, the emergence of SE itself as a new sub-discipline of computer science about forty years ago was the result of a crisis, namely, the inability to construct and maintain robust operating systems for a new generation of hardware with the rather limited SD techniques of the 1960s (see the history section of above). In this context, it is also interesting to remember that the so-called 'software crisis', which started from the moment the financial costs of computer software exceeded the financial costs of computer hardware, has never since been overcome: We have been living with this state of affairs for four decades now, and it seems that the perpetuation of Kuhnian 'anomalies' have become a new kind of normality in the SE domain.

Another relevant question in this discourse is: 'Can AM be called 'revolutionary' if they have not already become the dominant paradigm in ten years since their initiation?' Kuhn himself faced a similar criticism: for example, the period between Copernicus and Newton is often termed the 'Scientific Revolution' but the time-span involved, over 150 years, makes the process seem more like evolution than revolution. On the other hand, the definition of 'revolution', as used by us, does not refer to time-span at all. Whereas

political revolutions seem to occur rather swiftly, scientific and technological revolutions can take considerable time to occur, as for example in the 'Industrial Revolution'. Nonetheless, given the quick rate of change in the information age, perhaps the 'AM revolution' should have occurred already, if it is to take place at all. The most important question in this context, however, remains open to date: whether or not SE is actually a *science* already, or whether it is still in the state of a pre-scientific (although technical) human practice.[29]

In the following, we come back to the papers of Yourdon (2001a, b), since he seems to have made the most influential remarks on Kuhn within the context of the AM community. In spite of his various references to Kuhn, Yourdon's account of Kuhn's ideas is sometimes inaccurate and seems to be influenced by the same revolutionary (or, rather, rebellious) spirit already identified in Schwaber (2001)—a spirit which runs contrary to that of Kuhn's *Structure*, its title notwithstanding. Yourdon's definition of a paradigm omits an important Kuhnian sense of the term, namely, "the concrete puzzle-solutions which, employed as models or examples, can replace explicit rules as a basis for the solution of the remaining puzzles of normal science" (Kuhn 1962, p. 175). This omission by Yourdon is strange, given that implicit in this definition is the concept of 'tacit knowledge' which is an important concept to the AM community. Yourdon provides a definition of 'paradigm' that departs significantly from Kuhn's other central explication of the term,[30] namely where Yourdon speaks of paradigms as of performing a reasonably good job of describing and explaining the events and phenomena that we encounter in our day-to-day life. This misses Kuhn's crucial point concerning the "unparalleled insulation of mature scientific communities from the demands of the laity and of everyday life" (Kuhn 1962, p. 164). Most software engineers are not as isolated from everyday experience as most scientists since they are producing artefacts for use in everyday life. Moreover, Yourdon's reference to "scientists, engineers, soothsayers, or priests" is also problematic because Kuhn's 'paradigm shift' specifically relates to physical scientists, whereas software engineering is quite remote from the physical sciences. At no point in Kuhn (1962) does he explicitly refer to *engineers*, (and certainly not to soothsayers and priests). Nonetheless, the reference to soothsayers and priests, if not true to the word of Kuhn's theory, might come quite close to its spirit, since it suggests that paradigm changes are not completely rational, but require a leap of faith. Similarly one could object to Yourdon's descriptions that "meanwhile, there's likely to be a band of renegade scientists, engineers, or priests looking for a new paradigm" and that "In the past, the rebels promoting a new paradigm were likely to be burnt at the stake". Note how this echoes Schwaber's questionable use of the term 'revolution', as mentioned above.

Fuller, on the contrary, pointed out that Kuhn's enthusiastic followers generally "ignored that Kuhn, far from being a scientific revolutionary, argued that revolutions were only a last resort in science—indeed, an indication of just how fixated scientists tend to be on their paradigm [is] that they have no regular procedure for considering fundamental

[29] D. Knuth's seminal book 'The *Art* of Computer Programming' (Knuth 1968) was followed by D. Gries' book 'The *Science* of Programming' (Gries 1981): this shows how slowly the computer science community is trying to come to terms with an appropriate philosophical understanding of their own practices and activities; see Bishop (1991) and Arageorgis and Baltas (1989) for comparison.

[30] Whilst some authors have claimed that Kuhn would use his term 'paradigm' in a large variety of different notions throughout his writings (see for example Masterman 1970), Kuhn himself, in his 'Postscript' (Kuhn 1970), has rejected such allegations and insisted—including a reference to Masterman—that he would use the term 'paradigm' in *two* notions only (Kuhn 1970, pp. 181–182). These are the ones which we have mentioned above.

changes in research direction" (Fuller 2003, p. 22). Yourdon seems to have shifted from Kuhn's scientific sense of 'revolution' to a political sense that is, in fact, alien to Kuhn's theory (despite the fact that Kuhn actually compared the two). Fuller's critique shows how close Kuhn's paradigms are to the blueprints advocated by 'big upfront design' methodologies, similar to those advocated by the software engineering traditionalist. This is in complete contrast to XP with its many practices for accommodating change. Furthermore, Yourdon's claim that the new paradigm "explains all of the known phenomena much more cleanly and simply [than the old one]" also misrepresents Kuhn, who argues that the new paradigm does not solve all of the old puzzles, but rather abandons many of them, and, indeed, reviews, in the terms of the new paradigm, puzzles previously thought solved. Thus, the phenomena themselves have changed in the new paradigm, so radical is the break with the old paradigm. On the other hand, the new paradigm can solve many of the new puzzles that the older paradigm failed to solve. A final problem with Yourdon's depiction, from a Kuhnian point of view, is that it presents an objective criterion that allows one to make a rational choice between paradigms, a possibility which Kuhn rejected, and which more accurately fits the philosophy of Popper, as discussed in the following section.

6 Appreciation of Popper

> *Die Theorien aber sind wie dürre Blätter, welche abfallen, wenn sie den Organismus der Wissenschaft eine Zeit lang in Athem gehalten haben*[31]—Ernst Mach.

In this section, Popper's 'critical rationalism' (Albert 1994), which relates—like Mach's motto quoted above—to a long tradition of skepticism and fallibilism of various kinds, is used in an attempt to illuminate the values and principles underlying contemporary SD. Whilst some authors have sporadically applied Popperian concepts to aspects of the SE discipline (see Sect. 3), we aim to do so more systematically in this section. In particular, Popper's ideas will be used in an attempt to provide a comprehensive and unified philosophical basis for understanding AM. Here we argue that important aspects of the AM approach are strongly endorsed by Popper's philosophy of critical rationalism. To begin with, Popper's principle of falsificationism is transferred from the domain of physical science to the domain of SE. Whether or not it is legitimate to do so is another philosophical question that has already been raised in the previous section. In what follows, Popper's philosophy is applied both to AM in general and to software testing in particular.

Regarding the issue of software testing, we argue that the susceptibility of software to testing demonstrates its falsifiability, and thus the scientific nature of software development. Indeed, software programmers seem to resemble 'theoreticists' since they are responsible for establishing, by careful *a-priori* reasoning, an overall 'theory' that guides the development of working software programs. Moreover, software testers seem to resemble 'experimental scientists' since each test case they write is like a 'scientific experiment' which attempts to falsify a part of the developer's overall theory. Thus, SD seems to take physical science as the exemplar since it is testable, logical, mathematical, rigorous, repeatable, refutable and deductive.

With regard to software *verification*, which is complementary to software testing, Popper's falsificationism is useful in providing an understanding of the view that no

[31] The theories, however, are like dry autumn leaves which are falling off, after having enabled the tree of science to breathe for another while. Our translation, from Mach (1871, p. 46).

amount of software testing can prove a program correct. Snelting and Dijkstra were mentioned previously in this context. However, in spite of the similarity between their concepts of testability, Popper and Dijkstra argue from different perspectives: Dijkstra, arguing from a mathematical perspective about programs as *mathematical* entities, believes that a computer scientist should try to formally verify the correctness of an algorithm he has constructed, whereas Popper, arguing from the perspective of the natural sciences, rejects 'verification' in the *natural* sciences,[32] since this would require a positive result in every possible instance, (which is clearly unattainable). Whilst Popper can be seen as putting more emphasis on empirical methods, Dijkstra emphasises mathematical methods, with the aim of making the practical job of software testing, which is usually tedious and error-prone itself, somewhat easier.[33]

In order to adopt a purely Popperian approach to SD, software engineers must regard software systems as quasi-physical entities. Under this assumption they can then attempt to eliminate as many defects—via test cases—as possible, instead of striving to prove that programs are correct (from a mathematical perspective)—the goal of testing is to detect defects, not to demonstrate their absence. A principle difficulty of 'pure' Popperianism, also in SE, is highlighted by the questions: Who tests the testers? How far can we meta-test that a test was properly conducted? Therefore, to support the tedious procedure of rigorous software testing, programmers should aspire to write 'cleaner' software which would better enable such falsification attempts—whereby the word 'should' points towards a whole array of *normative* issues in SE, such as the enforcement of coding standards, and the like. Anyway, the *invention* of relevant and significant 'experiments' for software testing remains a difficult task which requires ingenuity and creativity (as Popper mentioned in his writings), expertise, as well as the aid of formal logical (mathematical) theories.[34]

One of the fundamental characteristics which distinguish AM, especially XP, from traditional SD methodologies is their concept of 'test-first' programming, the main focus of which is on defining test cases before developing the deliverable software. Initially, each of the test cases fails when executed, since the software is yet to be written. The failure of each test case points out lacking functionality. Then, the developers iteratively and incrementally implement the required functionality according to the most important or urgent requirements. This cycle is repeated until all the required functionality is implemented and all test cases pass. From a Popperian perspective, this practice of test-first programming can be understood as continuous attempts at falsification since the failing test cases (analogous with scientific experiments) point out that the program (analogous with the scientific theory or hypothesis) does not function according to requirements. The fact that test-first programming is an important basis of the development approach of AM means that these methodologies may be considered scientific in a Popperian sense.

[32] Note that Popper did not dispute the possibility of a positive mathematical or logical proof within the realm of mathematics; in fact Popper needs the validity of logics itself in order to be able to claim that one counter-example is sufficient to disprove an all-quantified expression of a disputed hypothesis.

[33] Mathematically verified software tends to reveal less defects in the testing phase than software which had been created without formal verification. With respect to the density of flaws in program code, the SE practitioner's classical 'rule of thumb' states: 'one flaw per thousand lines of software code', which would sum up to approximately thousand flaws in a larger software system consisting of about one million lines of code.

[34] Entire conferences are dedicated to this specific sub-field of SE; see for example the IEEE-ICST Conference on Software Testing, Verification and Validation, on the internet at http://www.cs.colostate.edu/icst2008/.

Another XP practice called 'pair programming' also supports and encourages falsificationism through its emphasis on interaction and collaboration. Pairs of programmers who sit together and co-program software, continuously subject their source code to a critical peer review process, which facilitates falsification. This practice encourages objectivity (or at least inter-subjectivity) and error detection due to the presence of an observer. The consensus resulting from pair programming seems to be achieved through an objective Popperian process emphasising criticism and error elimination, rather than through mere subjectivist (conventional) consensus as it would appear from a Kuhnian viewpoint. Yet another AM practice encouraging falsification is collective ownership of program-code within the society of a software factory. Individual code contributions made by group members are stored in collectively owned databases and, through a democratic peer review process, the functioning of the source code is thoroughly assessed. In principle, the collective ownership of code is similar to pair programming, however, the number of observers is larger than two. In short, AM in SE are 'human-centred'. They advocate continuous and frequent collaboration between all stakeholders of a project, especially between customers and developers. In AM, customers form an integral part of a development group and accompany the programmers throughout a project. They provide continuous feedback to the programmers which helps to detect and eliminate errors in the software as early as possible. This approach also resembles the role of falsificationism in Popper's method and it fits into the framework of Popper's political and social philosophy. This human-centred orientation of AM also manifests itself in the practice of using *stories* (instead of formalisms) for the *description* (instead of definition) of software requirements. In this context, the creative imagination required for story telling might be related to Popper's remarks on the invention of hypotheses (anti-inductionism) before the process of falsification can start.

In line with Popper's fallibilism, AM advocates acknowledge that people invariably make mistakes. Embracing this fact, they propose an iterative and incremental approach to SD. This approach, which derives historically from RUP, is now central to AM. It allows mistakes to be detected and repaired as early as possible, by reworking pieces of the software, before the errors accumulate to an un-manageable number. This iterative incremental approach is more similar to Popper's method of error detection and elimination than in the classical 'Waterfall' model of SD, because in AM the software is built in small steps allowing designs to be reworked and errors to be rectified after each increment.

According to Popper, all knowledge begins with problem-solving (Popper 1999). He proposed his cyclic 4-step model to represent the way in which scientific knowledge advances into an open future:

$$P1 \rightarrow TS \rightarrow EE \rightarrow P2.$$

From what has been said about SE above,[35] it should be clear that the small-step SD cycles of AM can be described in terms of this model, whereby a preliminary solution *P2* from a previous SD cycle poses further problems to the programmers (due to detected errors or due to modified expectations on the customer's part) and becomes thus the initial problem *P1* of the subsequent SD cycle. Moreover, *TS* would correspond to a new solution proposed by a developer, and *EE* would represent the attempt of the software tester to

[35] One might feel slightly reminded of Hegel's well-known 3-step dialectics: as far as the analogy holds, a Popperian double step ($TS \rightarrow EE$) would resemble some 'operational refinement' of an Hegelian 'antithesis'—the well-known fundamental differences between Hegel's and Popper's general philosophies of history not being taken into account at this specific point.

detect any flaws. Popper's remarks on evolution and selection (Popper 1999) are relevant in this context, too, because the iterative incremental AM approach makes it feasible to experiment with alterative software designs, whereby unsuccessful designs can easily be discarded. Note that software projects often keep several competing versions of a software product under development, whereby the least successful ones will eventually 'die' by not being pursued any further.

As mentioned earlier, software is ultimately developed by organizations, rather than by individuals. This is also true for SD in the AM paradigm. For this reason it might be an interesting research topic for the future to compare Popper's 'three worlds' of knowledge which must be shared (implicitly or explicitly, informally or formally) by the members of an organisation, a community or a society, with the concepts of organisational knowledge discussed, for example, by Nonaka and Takeuchi (1995), based on ideas of Polanyi (1967). This topic also relates to the question of if-and-how to do documentation (of the intermediate or final results) in a SD project: a question about which the AM community is also at odds with followers of the classical 'Waterfall' model of SD. In this essay, however, we refrain from such a discussion, in order not to deviate too far from our central theme, which is the SD process as such. Only summarily we conjecture that Popper's 'three worlds' metaphysics (combining ontology and epistemology) could be central to such a study, whereby we should also take notice of an interesting similarity between Popper's three 'worlds' and the four ontological realms ('Reiche') in the techno-philosophy of Friedrich Dessauer (1933); see also Mitcham (1994, p. 29).[36]

There is also some similarity between the AM community's 'baby-step' approach to SD and Popper's 'piecemeal' approach to social engineering (in an 'open' society), namely in their common rejection of an overall 'blueprint' for design. AM supporters would argue that SD projects are usually too complex and requirements too volatile to guarantee the correctness of comprehensive up-front design. Moreover, the 'human factor' introduces further unpredictability. All this potential for change means that there will inevitably be deviations from the blueprint. Therefore, AM are adaptive rather than anticipatory, which means they place less emphasis on comprehensive up-front design and more emphasis on principles which enable them to adapt to changing circumstances. Another similarity between AM and Popper's 'piecemeal' social engineering is their common focus on the present, rather than the future. AM emphasize the present in their core value of simplicity. To achieve simplicity, AM advocates assume a position of deliberate short-sightedness and they minimalistically implement software functionality only for those requirements which are immediately testable. This focus on the immediate problem is similar to the piecemeal engineer's focus on identifying the most urgent problems in society and bringing about a *rapid* solution, even at the price of sub-optimality.

7 Summary, Reflection and Conclusion

Forty years after its 'birth declaration' at the NATO Science conference in Garmisch, the young discipline of SE is still in a state of ambiguity which makes its philosophical

[36] Dessauer's fourth 'Reich' (realm), as the "Inbegriff aller eindeutig prästabilierten Lösungsgestalten" (Dessauer 1933, p. 50), comprises the 'ideas' of all technically feasible solutions which are 'awaiting' their 'discovery'—similar to the realm of algorithms and mathematical solutions in the quasi-platonic ontology of Penrose (1989). Even more (namely five) 'worlds' (of 'discovery', 'invention', 'development', 'being', and 'becoming'), have been distinguished by Jürgen Mittelstraß in several of his works; see for example Mittelstraß (2001).

assessment rather hard. Whilst authors like Mary Shaw (1996) tend to believe that SE is still in a state of 'craft' technology in the terminology of Arageorgis and Baltas (1989, p. 213) and has not even reached 'engineering' status at all, academic SE organisations, such as EASST,[37] already speak of software 'science' (as well as of software 'technology').[38] Where SE has common features with 'real' (i.e., material) engineering, the analysis provided by Arageorgis and Baltas (1989) is relevant; however the specific differences between SE and real engineering are still in need of philosophical attention and reflection.

Considering how questionable Kuhn's concept of 'paradigm shift' is when applied to questions of change in SD methodology, his popularity amongst prominent members of the AM community is quite surprising, especially when we take into account that Kuhn's theory had already been regarded as rather insignificant and criticised as "pure historicism" (Seiffert 1994) well *before* the 'agile movement' took off. Moreover, Kuhn was not the revolutionary that the rebellious young American students of the late 1960s and 1970s thought he was. On the contrary, according to Fuller, his critics saw Kuhn as "the official philosopher of the emerging military-industrial complex" (Fuller 2003, p. 35) and rather than having killed positivism, "as the Popperians saw it, Kuhn simply replaced the positivist search for timelessly true propositions with historically entrenched practices. Both were inherently uncritical and conformist" (Fuller 2003, p. 35). As Fuller also pointed out, part of the popularity of Kuhn is "the innocence [of his admirers] of any alternative accounts of the history of science (…) with which to compare Kuhn's" (Fuller 2003, p. 22), such as the fallibilism of Popper or Peirce (Buchler 1955) or, even earlier (19th century), Johann Eduard Erdmann (1896) in his reflections on the validity of hypotheses in the field of history. Popper's philosophy has been used in our essay as a corrective, although other philosophers, such as Peirce may have suited the purpose equally well. Indeed, the fallibilism of the latter two philosophers seems to be better suited to the principles and practices of AM than the uncritical authoritarianism and elitism of Kuhn's philosophy. In this aspect of technology assessment, Habermas and Popper seem closer to each other than they would like to admit, notwithstanding their differences as far as Hegelianism and other issues (including questions of politics) are concerned.

Moreover, it can be argued that Beck shifts his emphasis in the 1st and 2nd editions of his book (Beck and Andres 2005) from a Kuhnian revolutionary approach to a Popperian evolutionary one, since in the 1st edition, Beck advocates that, in order to be truly practising Extreme Programming, *all* agile values, principles and practices should be adopted and strictly adhered to, whereas in the 2nd edition he suggests, instead, a piecemeal approach to adopting the new methodology. A Kuhnian approach would entail a total replacement of the previous methodology with that of XP and would require a leap of faith, whereas Popper would allow a piecemeal approach whereby AM practices and principles can be adopted individually and be withdrawn if unsuccessful. Furthermore, Kuhn's insistence that paradigms are incommensurable suggests that the AM paradigm cannot be compared to any other software paradigm. However, it was shown earlier that several authors have indeed made such comparisons.

[37] European Association of Software Science and Technology; on the internet at http://www.easst.org/.

[38] This philosophical limbo is reflected by the organisations of different universities, in which we can find computer science and SE sometimes attached to the faculties of engineering, sometimes to the faculties of mathematics and natural sciences. Moreover, within the discipline itself, there is also growing dissent about whether SE still is (or should be) a sub-discipline of computer science, or whether it is (or should be) already a discipline in its own right, emancipated from computer science (in analogy to the similar debate about the relationship between computer science and artificial intelligence some time ago).

On the other hand, the incommensurability of paradigms implies that choosing between them is not completely rational, an implication that Beck would presumably resist. Instead, he would most probably argue that the XP principles and practices are all rational. However, why then does he use Kuhn's irrationalist theory of scientific revolutions? As mentioned above, Beck's statement regarding market prediction might provide a clue: Since XP will require significant change and since most people fiercely resist change, its adoption will not be largely due to rationality and good arguments, but will instead require an emotional conversion or leap of faith.

It would seem that Beck gained this insight about the irrationality of paradigm shifts from Kuhn despite the fact that Kuhn never mentioned markets. Nonetheless, as Fuller writes, "Kuhn saw science as a knowledge enterprise" (Fuller 2003, p. 15). This may thus be a solution: Although the values, principles and practices of XP are highly rational according to Popper's critical rationalist criterion, the adoption of XP as a whole will be more a matter of emotion rather than logic, as full of 'hype' and persuasive techniques as the selling of a new product on the market. This is, however, a significant departure from the strict use of Kuhn's theory of scientific revolutions.

In this essay we have conducted an exercise in 'applied philosophy'—in contrast to 'pure' philosophy—in the sense that we did not discuss a general problem (e.g. the problem of knowledge or the problem of science as such). Instead, our discussion was motivated occasionally, by the specific problems of SE, as outlined in the introductory section.

Our essay would be unsatisfactory without a brief reflection on the possible *limitations* of such an approach. Especially in our context, two questions need to be asked: Is Kuhn's view of the (history of the) scientific world largely correct, or is his opinion itself merely a 'paradigm'—a temporary fashion which is soon likely to be superseded by another, with its own set of believers and dedicated followers? Secondly: Regardless of whether or not Kuhn's model is intrinsically plausible, did we apply it consistently and appropriately to our specific problem in the context of SE? As far as the first of these two questions is concerned, we refer the reader to the ongoing philosophical discourse with inputs from a great diversity of sources such as 'critical theory' (School of Frankfurt), 'critical rationalism' (Popper, Albert, etc.), 'pragmatism' or 'pragmaticism' (Peirce, James, etc.), various versions of (meta)-scientific 'relativism' (Nietzsche, Feyerabend, etc.) and many others. Future work might delve somewhat deeper into the details of these issues. As far as the second question is concerned, we concede that it is methodologically daring to apply something like Kuhn's model, which was initially conceived as a historic meta-theory about scientific theories, to the field of SE which is, more than anything else, a technical practice and not a mathematically formulated, predictive scientific 'theory' in the classical sense of the term.

As far as the School of Frankfurt is concerned, it is also not yet fully understood how Habermas' well-known three categories of research processes (namely: empirical-analytic, historical-hermeneutic, critical-reflective) can be *adequately* (and not just 'roughly') mapped to those various complicated activities which all together constitute a SD process. Modally speaking we could ask: What degree of 'necessity' would Habermas' category-mapping possess with respect to SE, and what degree of 'sufficiency'? Moreover: Is a Frankfurtian (or maybe even Althusserian) notion of 'theory' compatible with the with the notions of 'theory' as they are understood by software engineers and software scientists themselves? Or are 'construction' and 'interpretation' (i.e. 'theory') glued together in a "Leibniz-world" (Mittelstraß 2001, p. 21) to such an extent that they cannot be separated into different categories of research processes at all—and, if so, what would this imply as far as our philosophical understanding of SE is concerned?

Some minimal kind of 'theory', however 'hidden' or un-reflected, *must* exist to motivate the followers of *any* SD methodology—even if such a 'theory' would not express anything more than the simple assertion that, if you would produce software in this and this particular way, then you would be effective and efficient in delivering products of the required quality. Yet it is this lack of explicit (meta) theory—where it occurs—which renders the practice of SE akin to other practice-oriented disciplines, for example pre-scientific forms of medicine ('Heilkunde') or pre-scientific forms of biology (e.g. cattle breeding), which have to rely for their predictive purposes on the heuristics of a 'Lebenswelt' rather on than the precision of mathematics. Practically (and inter-subjectively) such 'hidden background theories' (to any SE approach) manifest themselves in a multitude of frequently proposed and published (yet rarely applied) software *metrics*, quality *criteria*, capability maturity *models* (like CMM or TMM), 'best practice' *recommendations* based on anecdotic 'success stories' transmitted at meetings and workshops, corporate or governmental *norms*, and the like. It would seem, therefore, that it is the more or less implicit adherence to various versions of such un-clarified background 'theory' that makes the discipline of SE so prone to the formation of those kinds of opinion-based 'schools' which Kuhn (as well as Feyerabend and others) have mentioned in their writings.

From a small-scale (i.e. more technical) perspective of SD, however, there is *no* room for 'opinion' and hence no room for a Kuhnian approach. More specifically, a Kuhnian perspective has nothing to say about a computer program that would, for example, calculate the square-root of 16 as '5'. At this small-scale level of SD (respectively computer programming), where non-negotiatable issues of 'true' or 'false' are at stake, Kuhnian thought seems inapplicable. It is at this small-scale (technical) level of SD where Popperian ideas of conjecture and refutation are best applied.

At this point we interrupt our discussion towards a philosophy of SE, and we invite the reader to participate in this emerging discourse. Considering the practical context of our theme we might expect reactions especially from philosophers belonging to various pragmaticist, constructivist, culturalist traditions (in the footsteps of William James, Hugo Dingler, etc.), with their special emphasis on human purposes and actions, at which we have only hinted in this essay. Analytic philosophers might be interested in software from the perspective of language, whereas structuralists might want to search for hidden patterns or schemes. Perhaps even Kantian thinkers, with their characteristic focus on questions of transcendental preconditions, could make some interesting contributions too.

Acknowledgments Thanks to our colleagues *Morkel Theunissen* and *Markus Roggenbach* for inspiring discussions about 'Software Engineering versus Software Science'. Thanks to our colleague *Tim Grant* for directing our attention to Smith (1997). *Elaine Byrne* is acknowledged for providing a copy of Lyytinen and Klein (1985), and *Judith Bishop* for a copy of Bishop (1991). Many thanks also to *Gert König, Helmut Pulte* and *Lutz Geldsetzer*, editors, as well as the anonymous reviewers of this journal for their helpful remarks and comments on various draft versions of the manuscript; their hints have been taken into account for this published version of our essay. Moreover we would like to thank the editors for sending us a copy of Arageorgis and Baltas (1989). Last but not least thanks to *Marlene Fischer* at the editorial office for friendly communications and efficient handling of the manuscripts.

References

Adorno, T., Albert, H., & Dahrendorf, R. (1993). *Der Positivismusstreit in der deutschen Soziologie*. München: DTV.

Aichernig, B. K. (2001). *Systematic Black-Box Testing of Computer-Based Systems through Formal Abstraction Techniques*. Dissertation, Technische Universität Graz, Austria.

Albert, H. (1994). Kritischer Rationalismus. In H. Seiffert & G. Radnitzky (Eds.), *Handlexikon zur Wissenschaftstheorie* (2nd ed., pp. 177–182). München: DTV Wissenschaft.
Alexander, C. (1999). The origins of pattern theory. *IEEE Software*, September/October, 71–82.
Arageorgis, A., & Baltas, A. (1989). Demarcating technology from science—Problems and problem solving in technology. *Zeitschrift für allgemeine Wissenschaftstheorie, 20*(2), 212–229.
Bach, J. (2000). Skill over process. *Cutter Consortium*. Retrieved November 2007 from https://cutter.com/cgi-bin/catalog/store.cgi.
Basden, A. (2008). *Philosophical frameworks for understanding information systems*. USA: IGI Publishing. ISBN-10: 1599040360.
Beck, K. (2002). XP and culture change. *Cutter Consortium*. Retrieved November 2007 from https://cutter.com/cgi-bin/catalog/store.cgi.
Beck, K., & Andres, C. (2005). *Extreme programming explained: Embrace change* (2nd ed.). London: Addison-Wesley.
Beck, K., & Fowler, A. (2001). Manifesto for agile software development. Retrieved November 2007 from http://www.agilemanifesto/org/.
Becker, W. (1994). *Ideologie*. In H. Seiffert & G. Radnitzky (Eds.), *Handlexikon zur Wissenschaftstheorie* (2nd ed., pp. 144–150). München: DTV Wissenschaft.
Bishop, J. M. (1991). Computer programming: Is it computer science? (Inaugural Lecture). *South-African Journal of Science (Suid-Afrikaanse Tydskrif vir Wetenskap), 87*, 22–33.
Boehm, B. (2002). Get ready for agile methods, with care. *IEEE Software, 35*(1), 64–69.
Brennecke, A., & Keil-Slawik, R. (Eds.). (1996). *Position Papers for Dagstuhl-Seminar 9635 on the History of Software Engineering*. Schloß Dagstuhl, Germany, August 1996. Retrieved November 2007 from http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=199635.
Brooks, F. P. (1987). No silver bullet: Essence and accidents of software engineering. *Computer, 20*(4), 10–19.
Brooks, F. P. (1995). *The mythical man-month and other essays on software engineering* (2nd ed.). London: Addison Wesley.
Buchler, J. (Ed.). (1955). *Philosophical writings of peirce*. New York: Dover.
Bullock, A., & Trombley, S. (Eds.). (1999). *The new fontana dictionary of modern thought* (3rd ed.). Glasgow: Harper Collins.
Cockburn, A. (1999). *A methodology per project*. Retrieved November 2007 from http://alistair.cockburn.us/index.php/Methodology_per_project.
Cockburn, A. (2002). *Agile software development* (2nd ed.). London: Addison-Wesley.
Coutts, D. (2007). *The test case as a scientific experiment*. Retrieved November 2007 from http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectId=8965.
Cutter (2007). *Cutter consortium*, Online Bookstore. Retrieved November 2007 from https://cutter.com/cgi-bin/catalog/store.cgi.
Davies, R. (2006). *Agile paradigm shift*. Keynote lecture to the Agile North Conference, 2006. Retrieved November 2007 from http://www.agilenorth.net/DownloadFiles/2006Conference/.
Dessauer, F. (1933). *Philosophie der Technik: Das Problem der Realisierung* (3rd edn.). Bonn: Verlag von Friedrich Cohen (1st ed., 1927).
Dijkstra, E. W. (1968). Go-to statement considered harmful (Letter to the Editor). *Communications of the ACM, 11*(3), 147–148.
Erdmann, J. E. (1896). *Grundriss der Geschichte der Philosophie* (Reprint 1993: Petra Wald).
Firestone, J. M., & McElroy, W. (2003). *The open enterprise: building business architectures for openness and sustainable innovation*. Hartland Four Corners, VT: KMCI On-line Press. Retrieved November 2007 from http://www.dkms.com/papers/openenterpriseexcerptnumb1final.pdf.
Fuller, S. (2003). *Kuhn versus Popper: The struggle for the soul of science*. Cambridge: Icon Books.
Geldsetzer, L. (1980). Methodologie. In J. Ritter & K. Gründer (Eds.), *Historisches Wörterbuch der Philosophie* (Vol. V, pp. 1379–1386). Basel: Schwabe-Verlag.
Giguette, R. (2006). Building objects out of plato: Applying philosophy, symbolism and analogy to software design. *Communications of the ACM, 49*(10), 66–71.
Gregg, D. G., Kulkarni, U. R., & Vinze, A. S. (2001). Understanding the philosophical underpinnings of software engineering research in information systems. *Information Systems Frontiers, 3*(2), 169–183.
Gries, D. (1981). *The science of programming* (1st ed.). Heidelberg/New York: Springer-Verlag.
Gruner, S. (2007). The path to innovation. *Innovate, 2*, 96. Pretoria: ISSN 1814-443X.
Hall, W. P. (2003). Managing maintenance knowledge in the context of large engineering projects: Theory and case study. *Journal of Information and Knowledge Management, 2*(3), 1–17.
Hamlet, D. (2002). *Science, computer science, mathematics and software development*. Keynote Lecture to the SR Quality Week Conference. Retrieved November 2007 from http://web.cecs.pdx.edu/~hamlet/QW.pdf.

Hanks, P. (Ed.). (1991). *Collins English dictionary* (3rd ed.). Glasgow: Harper Collins.
Heidegger, M. (1927). *Sein und Zeit* (19th ed. 2006). Tübingen: Niemeyer-Verlag.
Heidegger, M. (1949). Die Frage nach der Technik (Reprinted 1954 in Heidegger, M., *Vorträge und Aufsätze*. (Pfullingen: Neske-Verlag)).
Hoare, C. A. R. (1985). *Communicating sequential processes*. London/New York: Prentice-Hall.
Hoare, C. A. R. (2003a). Towards the verifying compiler. *Lecture notes in computer science* (Vol. 2757, pp. 151–160). Heidelberg: Springer-Verlag.
Hoare, C. A. R. (2003b). Assertions: A personal perspective. *IEEE Annals of the History of Computing, 25*(2), 14–25.
Hoare, C. A. R. (2006). Why ever CSP? *Electronic Notes in Theoretical Computer Science, 162*, 209–215. Retrieved November 2007 from http://dx.doi.org/10.1016/j.entcs.2006.01.031.
Knuth, D. (1968). *The art of computer programming* (1st ed.). London: Addison-Wesley.
Knuth, D. (1974). Structured programming with go-to statements. *ACM Computing Surveys, 6*(4), 261–301.
Kroeze, J. H. (2007). Linguistic information—A humanistic endeavour. *Innovate, 2*, 38–39. Pretoria: ISSN 1814-443X.
Kuhn, T. (1962). *The structure of scientific revolutions*. Chicago: University of Chicago Press.
Kuhn, T. (1970). *The structure of scientific revolutions—Postscript* (2nd ed.). Chicago: University of Chicago Press.
Lobkowicz, N. (1994). *Interesse*. In H. Seiffert & G. Radnitzky (Eds.), *Handlexikon zur Wissenschaftstheorie* (2nd ed., pp. 158–160). München: DTV Wissenschaft.
Lux Group. (2007). *Project lifecycles—Waterfall, rapid application development, and all that.* Retrieved November 2007 from http://luxworldwide.com/whitepapers/project_lifecycles_waterfall_rapid_application_development.asp.
Lyytinen, K. J., & Klein, H. K. (1985). The critical theory of Jürgen Habermas as a basis for a theory of information systems. In E. Mumford et al. (Eds.), *Research methods in information systems* (pp. 219–236). Amsterdam: North-Holland/Elsevier.
Mach, E. (1871). *Die Geschichte und die Wurzel des Satzes von der Erhaltung der Arbeit*. Lecture to the Royal Bohemian Society of Sciences, 15th November 1871. (1st ed., Prague: 1871; 2nd ed., Leipzig: 1909).
Marcuse, H. (1964). *One-dimensional man—Studies in the ideology of advanced industrial society*. Boston: Beacon Press.
Margolis, J. (1980). *Art and philosophy*. Sussex: Harvester Press.
Marick, B. (2004). Methodology work is ontology work. *ACM SIGPLAN Notices, 39*(12), 64–71.
Marzolf, T., & Guttman, M. (2002). Systems minus systems thinking equals big trouble. *Cutter Consortium.* Retrieved November 2007 from https://cutter.com/cgi-bin/catalog/store.cgi.
Masterman, M. (1970). The nature of a paradigm. In I. Lakatos & A. Musgrave (Eds.), *Criticism and the growth of knowledge* (pp. 59–89). Cambridge: Cambridge University Press.
McElroy, W. (2002). *Deep knowledge management*. Retrieved November 2007 from http://www.macroinnovation.com/images/DeepKMbyM.W.McElroy.pdf.
Methner, H., Oberhoff, W. D., Schmidt, P., Swiridow, A. P., Unger, H., & Vollmar, R. (Eds.). (1997). *Computer und Kybernetik—Anmerkungen zu ihrer Geschichte und zu Perspektiven in der Zeit von 1940 bis 1965.* III Russisch-Deutsches Symposium, Heidelberg, Germany, November 1997 (Bonn: Gesellschaft für Informatik, and Moscow: Russian International Informatization Academy).
Meyer, S. (2007). *Pragmatic versus structured computer programming.* Retrieved November 2007 from http://www.pragmatic-c.com/docs/structprog.pdf.
Mitcham, C. (1994). *Thinking through technology: The path between engineering and philosophy*. Chicago: University of Chicago Press.
Mitcham, C. & Mackey, R. (1973). Bibliography of the philosophy of technology. *Technology and Culture*, 14/2, Part II.
Mittelstraß, J. (2001). Konstruktion und Deutung: Über Wissenschaft in einer Leonardo – und Leibniz-Welt. Festvortrag anläßlich der Verleihung der Ehrendoktorwürde, January 2001. Technical Report, Series *Öffentliche Vorlesungen*, 110, Faculty of Philosophy, Humboldt-University to Berlin. (Berlin: ISSN 1618-4866, ISBN 3-86004-144-4).
Moss, M. (2003). *Why management theory needs Popper: The relevance of falsification.* Retrieved November 2007 from http://www.markwmoss.com/falsificationism.htm.
Naur, P., & Randell, B. (Eds.). (1968). *Software engineering—Report on a conference sponsored by the NATO Science Committee*. Garmisch, Germany, October 1968, published January 1969. Retrieved November 2007 from http://homepages.cs.ncl.ac.uk/brian.randell/NATO/.
Nonaka, I., & Takeuchi, H. (1995). *The knowledge-creating company*. Oxford: Oxford University Press.
Northover, M., Boake, A., & Kourie, D. G. (2006). Karl Popper's critical rationalism in agile software development. *Lecture notes in artificial intelligence* (Vol. 4068). Heidelberg: Springer-Verlag.

Northover, M., Northover, A., Gruner, S., Kourie, D. G., & Boake, A. (2007). *Agile software development: A contemporary philosophical perspective*. ACM International Conference Proceeding Series (Vol. 226, pp. 106–115). doi:http://doi.acm.org/10.1145/1292491.1292504.

Penrose, R. (1989). *The emperor's new mind*. Oxford: Oxford University Press.

Polanyi, M. (1967). *The tacit dimension*. New York: Doubleday.

Popper, K. R. (1963). *Conjectures and refutations: The growth of scientific knowledge*. London: Routledge.

Popper, K. R. (1999). *All life is problem solving*. London: Routledge.

Rayside, D., & Campbell, G. T. (2000). *An Aristotelian understanding of object-oriented programming*. 15th ACM SIGPLAN Conference on OOP Systems, Languages and Applications (pp. 337–353). doi: http://doi.acm.org/10.1145/353171.353194.

Roscoe, A. W. (1997). *Theory and practice of concurrency*. London/New York: Prentice-Hall.

Roscoe, B. (=A.W.) (Ed.). (2005). Communicating sequential processes—The first 25 years. *Lecture notes in computer science* (Vol. 3525). Heidelberg: Springer-Verlag.

Sachsse, H. (1994a). *Technik*. In H. Seiffert & G. Radnitzky (Eds.), *Handlexikon zur Wissenschaftstheorie* (2nd ed., pp. 358–361). München: DTV Wissenschaft.

Sachsse, H. (1994b). *Technologie*. In H. Seiffert & G. Radnitzky (Eds.), *Handlexikon zur Wissenschaftstheorie* (2nd ed., pp. 361–365). München: DTV Wissenschaft.

Schnädelbach, H. (1994). *Positivismus*. In H. Seiffert & G. Radnitzky (Eds.), *Handlexikon zur Wissenschaftstheorie* (2nd ed., pp. 267–269). München: DTV Wissenschaft.

Schwaber, K. (2001). The agile alliance revolution. *Cutter Consortium*. Retrieved November 2007 from https://cutter.com/cgi-bin/catalog/store.cgi.

Seiffert, H. (1994). Die Theorie Thomas S Kuhns. In H. Seiffert & G. Radnitzky (Eds.), *Handlexikon zur Wissenschaftstheorie* (2nd ed., pp. 411–412). München: DTV Wissenschaft.

Shaw, M. (1996). Three patterns that help [to] explain the development of software engineering. In A. Brennecke & R. Keil-Slawik (Eds.), *Position Papers for Dagstuhl-Seminar 9635 on the History of Software Engineering*. Schloß Dagstuhl, Germany, August 1996 (pp. 52–56). Retrieved November 2007 from http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=199635.

Simon-Schaefer, R. (1994). *Kritische Theorie*. In H. Seiffert & G. Radnitzky (Eds.), *Handlexikon zur Wissenschaftstheorie* (2nd ed., pp. 172–177). München: DTV Wissenschaft.

Smith, R. P. (1997). The historical roots of concurrent engineering fundamentals. *IEEE Transactions on Engineering Management, 44*(1), 67–78.

Snelting, G. (1997). Paul Feyerabend und die Softwaretechnologie. *Softwaretechnik-Trends, 17*(3) (Bonn: Gesellschaft für Informatik). Reprinted in *Informatik Spektrum, 21*(5), 273–276, 1998. Retrieved November 2007 from http://pi.informatik.uni-siegen.de/stt/17_3/17_3_LB_snelting.ps.

Snelting, G. (1998). Paul feyerabend and software technology (English Translation of Snelting 1997). *Software Tools for Technology Transfer, 2*(1), 1–5. doi:10.1007/s100090050013.

Sommerville, I. (2007). *Software engineering* (8th ed.). London: Addison-Wesley Publ.

Tichy, W. (2007). *Empirical methods in software engineering research.* Invited Lecture to the 4th IFIP WG 2.4 Summer School on Software Technology and Engineering, March 2007, Gordon's Bay, South-Africa.

Vangheluwe, H. (2008). *Foundations of modelling and simulation of complex systems.* Invited Lecture to the 7th GT-VMT International Workshop on Graph Transformation and Visual Modeling Techniques at ETAPS'08, March 2008, Budapest, Hungary.

Wernick, P., & Hall, T. (2004). Can Thomas Kuhn's paradigms help us [to] understand software engineering? *European Journal of Information Systems, 13*, 235–243.

Yourdon, E. (2001a). Paradigm shifts. *Cutter Consortium*. Retrieved November 2007 from https://cutter.com/cgi-bin/catalog/store.cgi.

Yourdon, E. (2001b). The XP paradigm shift. *Cutter Consortium*. Retrieved November 2007 from https://cutter.com/cgi-bin/catalog/store.cgi.