J Gen Philos Sci (2010) 41:237–260
DOI 10.1007/s10838-010-9116-y

ESSAY REVIEW

Software Engineering Between Technics and Science

Recent Discussions about the Foundations and the Scientificness of a Rising Discipline

Stefan Gruner

Published online: 20 May 2010

> "Es steht gegenwärtig allen Wissenschaften eine Wiedergeburt in Ansehung ihrer Begriffe und der Geistlosigkeit bevor, die wissenschaftlichen Inhalt in bloßes Material verwandelt und die Begriffe, deren sie zu handhaben gewöhnlich nicht einmal weiß, unkritisch und bewußtlos handhabt" (G.W.F. Hegel, 1807).

1 Introduction

This essay-review presents and discusses relevant recent contributions to the science-philosophical and methodological discourse within the discipline of software engineering, about the scientificness of this discipline, also in comparison to other, related disciplines. The main problems in this context are exposed and explained especially for those readers who are not software engineers themselves.

Since the 'official' establishment of software engineering as a discipline at the historic NATO Science Conference 1968 in Garmisch, Germany, there has been a rising tide of science-philosophical and methodological discussions about the degree of 'scientificness' and/or 'engineeringness' of this still young and emerging discipline. Notorious experiences of spectacular technical accidents due to software failure seem to support the opinion of the skeptics who claim that software engineering is de-facto still only 'art' or 'craft', or at most 'technics', but neither 'science' nor 'engineering'. Indeed it was the crisis of software engineering—not its undeniable practical success—that has triggered its meta-scientific, methodological and philosophical reflections; people do not tend to philosophise during care-free times of success. On the other hand there is now a widespread normative agreement amongst many experts that software engineering should be 'engineering' and 'science' as far as possible, with older (and apparently more 'successful') branches of engineering being looked at as 'role models' and encouraging

S. Gruner (✉)
Department of Computer Science, Research Group for Software Science and Formal Methods, University of Pretoria, Pretoria 0002, Republic of South Africa
e-mail: sg@cs.up.ac.za

examples. This desideratum of software engineering has then motivated further meta-scientific and science-philosophical discussions about the criteria of scientificness and/or engineering-ness particularly for this discipline. However, those discussions have been carried out mostly within the discipline so far, by philosophically and meta-theoretically minded software engineers themselves, and not so much by 'professional' philosophers in the faculties of philosophy.

Throughout this essay I assume that the reader already has at least some very basic ideas about what software engineering is and what it entails in its daily practice. In the Appendix of this essay I have listed a small selection of interesting titles for further reading, in support of those readers who are not very familiar with this topic. Those titles listed in the Appendix are of course not the particular 'references' on which the discussions in the subsequent sections of this essay are based.

The aim of this essay—objective information about current philosophical approaches to software engineering, and their critique—is achieved by the classical method of literature review, for which some relevant, 'typical' or 'paradigmatic' recent papers have been carefully chosen. The selection criteria underlying this choice are related to the three large groups of science-philosophical positions which can currently be observed in software engineering's methodological meta-discourse. For the sake of argument I shall now call them the 'physicalist' group, the 'socio-humanist' group, and the 'classical engineering' group. They can be briefly characterised as follows:

1. The 'physicalist' group is guided by the classical leitmotiv of physics being 'the' paradigm of science; consequently the followers of this group are discussing software engineering and its methodology very much in terms of 'hypothesis', 'experiment', 'measurement', 'law', etc. Philosophically they often sympathize with 'classical' philosophy of science, which has also been very much physics-oriented all the time.
2. Followers of what I call the 'socio-humanist' group, on the contrary, emphasise that 'software is made by people for people'; consequently they look at the practice of software engineering very much through the lenses of sociology and psychology, thereby often rejecting the so-called 'positivist' viewpoints of the 'physicalists', especially the so-called 'positivist' desire for measurement and law-based predictability. Often the 'socio-humanists' also emphasise 'pragmatism' and 'flexibility' the face of the 'human needs', in contrast to the rigorous 'procedures' recommended by the 'engineers'. It comes to no surprise that various members of the 'socio-humanist' group have found quite some inspiration in various 'anti-establishment' philosophies, e.g.: the School of Frankfurt, the doctrines of socio-constructivism, etc.
3. Last but not least there is the third group of the 'engineers', who prefer to have software engineering like a classical 'engineering' discipline (rather than a 'science'), with a particularly strong emphasis on the industrial production of technical artefacts on the basis of well-established, time-proven production processes and a highly standardized 'body of knowledge' (BOK) gathered in readily applicable engineering hand-books. They have found much inspiration in various books about modern philosophy of engineering—see Appendix for some examples of such books.

Of course there can also be various 'hybrid' positions somewhere in the middle between those three extremes. There are even further science-philosophical positions about software engineering, such as Dijkstra's 'mathematicalism' which would prefer to see software construction almost as a branch of applied mathematics without much reference to any material processes. However, that position is rather rare and therefore not in the focus of this essay.

Each of the papers (Gregg et al. 2001; Rombach and Seelisch 2008; Maibaum 2008) which will be discussed in the subsequent sections of this essay can be regarded as a 'typical' example which is representative of one of those three science-philosophical positions about software engineering. On the bases of these discussions, this essay also proposes a preliminary definition of the term 'software science' as an original contribution to this discourse.

Last but not least this essay also briefly discusses one paper which does not fall clearly into any of the above-mentioned groups (Logrippo 2007), for its interesting elaboration on the notion of 'law'. This is a key concept at the intersection of the domain of the natural sciences ('law of nature'), the domain of the formal sciences ('law of mathematics') as well as the domain of the human sciences ('law of justice'); the discussion will also hint at why and how the notion of 'law' is relevant in the domain of software engineering (particularly its sub-domain of requirements specifications), too.

In summary, one could thus argue that (Logrippo 2007), via the notion of 'law', could perhaps offer some kind of bridge across the philosophical and methodological gaps between the 'physicalists' (Rombach and Seelisch 2008), the 'socio-humanists' (Gregg et al. 2001), and the 'engineers' (Maibaum 2008) in the science-philosophical meta-discourse of software engineering as an emerging discipline.

2 Philosophical Underpinnings of Software Engineering Research in Information Systems

In this section, an interesting article with the above-mentioned title, published by D.G. Gregg, U.R. Kulkarni, and A.S. Vinze (Gregg et al. 2001) in "Information Systems Frontiers", will be discussed. Their article is about the Information Systems (IS) discipline and related research which focuses on the development, understanding, and use of information technology for business needs. Thereby particularly software is the basis of IS research, thus making software engineering a critical issue of research in the IS domain. However, according to (Gregg et al. 2001), whilst the importance of software development is well accepted, what constitutes high quality software engineering research is not well defined. Perhaps this is, according to (Gregg et al. 2001), because some software development clearly is not 'research' and it is also hard to distinguish between pure application development, and systems development that pushes the boundaries of knowledge. Their article suggests that software engineering can meet Popper's criteria for scientific research. Drawing on well-established research philosophies, Gregg et al. (2001) proposed a software engineering research methodology (SERM) and discussed the utility of this methodology for contributing to and expanding the IS body of knowledge. They also described some considerations that need to be addressed by SERM to enhance acceptability of software engineering research in IS.

In the following discussion of that paper, all verbatim quotations in this section are taken from (Gregg et al. 2001); only the page numbers (p. xyz) will be explicitly given after every quotation. Also throughout this section, the term 'Information Systems' (IS) stands for a sub-discipline in informatics and software engineering, not for the concrete information systems as technical artefacts which are produced by the practitioners of that discipline.

As mentioned in the introduction, Gregg et al. (2001) was chosen as an example of the 'socio-humanist' approach to software engineering, whereby technical or technological issues do not come to the foreground. It is a methodological paper that deals with the

question of the scientificness of software engineering, with a particular focus on a classification of published papers (i.e.: a meta-study) of software-engineering-related works in the domain of business-oriented IS. Positively we note about (Gregg et al. 2001) that an interdisciplinary attempt towards a philosophical understanding of software engineering is made. However, it must be critically asked if their understanding of 'software engineering' is fully adequate, and if their notion of 'philosophical' is accurate enough to justify the classification of their paper as a suitable resource for a philosophical meta theory of software engineering under the umbrella of a general philosophy of science and technology: "Software engineering as a research methodology […] can be defined as an approach that allows the synthesis and expression of new technologies and new concepts in a tangible product that can […] contribute to basic research and serve as an impetus to continuous research." (p. 169). Further it is stated that "the development of a software system, by itself, is usually not regarded as serious research" (p. 169). Consequently, "there is disagreement as to whether software engineering represents a scientific method of enquiry", and "no accepted standard and very few rules are available to govern the way software engineering research is conducted" (p. 170).

Here we can clearly see a general science-philosophical question at the horizon, namely the question about the criteria that a network of activities must fulfill in order to be rightfully called 'research'. Alas, one can find rather different answers to that question not only in different eras of history, but also in different academic domains within the same historical period—remember only the notorious 'Positivismusstreit' in sociology between the schools of 'Critical Rationalism' and 'Critical Theory' in the previous century. Consequently it would come to no surprise if an IS manager's opinion about the research-ness of 'software engineering research' would differ considerably from a computer scientist's opinion about the research-ness of 'software engineering research', which would then lead to the general science-philosophical question if those opinions could possibly be reconciled under a common conceptual roof.

The paper continues with the claim that "how software engineering research is performed is left almost entirely up to the individual researcher. This creates difficulties for reviewers trying to determine whether a given software development project constitutes high quality research" (p. 170). In introspection I found that statement rather surprising, for I usually do not subjectively feel such "difficulty" when acting as a reviewer for software engineering conferences; it seems that there is almost certainly an element of experience and intuition in the judgment of scientific work, which also relies upon a hard-to-describe sense of aesthetics (as Albert Einstein and his 'little finger' have always strongly asserted). Subjective introspection aside, three replies to the claim of above can now be given:

1. Following Popper, who strongly emphasised the element of creativity, it seems to be rather irrelevant "how research is performed" as long as the generated hypotheses are effectively testable: If testable, we do not need to care very much any more about how they came about, such that there is no need for any fundamentalist 'methodologism' as far as the creation of hypotheses is concerned. In (Gregg et al. 2001), however, we can find traces of such 'methodologism',[1] especially at the point where the authors complain that the "importance of a theoretical framework, although well understood and frequently referred to, is often sidestepped in software

[1] It is perhaps no surprise that particularly the social sciences clinge so strongly to the idea of methodologically correct ways of hypothesis generation as main criterion of their 'scientificness', since their hypotheses themselves are so much harder to test experimentally than the hypotheses generated in the natural sciences.

development research" (p. 171)—note that this research 'methodologism' of Gregg et al. (2001) must not be confused with the rigorous artefact production 'methodologism' advocated by the 'engineers' (as outlined above in Sect. 1).

2. Already in 1998, Gregor Snelting had published a strong warning against all sorts of 'quackery' in software engineering research (Snelting 1998a). Contrary to the picture painted in Gregg et al. (2001), Snelting's warnings are indeed well-known in the software engineering 'community', and they can also inform the peer reviewing process amongst diligent members of this 'community'.
3. More recently, Walter Tichy has given a comprehensive overview of empirical methods for software engineering research from within the domain of software engineering (Tichy 2007); needless to say that that later contribution was not yet know-able to the authors of Gregg et al. (2001). In his methodology, Tichy identified "experimental" versus "descriptive" software engineering research, whereby the latter one was then further divided into "qualitative" and "quantitative" studies (Tichy 2007), similar to the "qualitative" and "quantitative" studies which are known in the area of the social sciences.

The external perspective onto software engineering in (Gregg et al. 2001) in their enquiry about the scientific-ness of software engineering becomes obvious wherever the authors give examples of software engineering research, such as:

> "If a researcher is proposing an entirely new way of looking at a problem and wonders if a system can be developed that will address the problem, then the engineering of such a software would constitute research. For example, in the late 1980s researchers were wondering if information systems could be developed to aid group decision making" (p. 171).

From a computer scientist's perspective, however, it is hardly surprising that such decision support systems can somehow be implemented, because the Turing Machine as universal machine is theoretically known to be suitable for the implementation of "almost anything" (within the limits of the theory of computability). Internal problems of software engineering research (for example: the invention of suitable module concepts which can then be applied as principles to any concrete software development project)—i.e. the engineering perspective—were not sufficiently taken into account in (Gregg et al. 2001).

In another step of argumentation the question was raised whether software engineering would actually constitute a 'research paradigm' (p. 171) in comparison to other research paradigms known in the social sciences: "While these [research] paradigms provide a good basis for a majority of the IS research stream, they do not fully address the unique requirements of software engineering" (p. 172). Though their hint to some "unique requirements of software engineering" remains rather vague, the need for a software-engineering-specific approach in our science-philosophical enquiry is acknowledged. Nevertheless, I fail to see how software engineering, as a practice so open-ended and so tightly interwoven with other scientific disciplines, could possibly constitute a whole new 'research paradigm'—at least not in Kuhn's notion of 'paradigm' which he had characterized as epistemologically isolated and incommensurable with other paradigms. Anyway:

> "To alleviate limitations of the Positivist/Postpositivist and Interpretive/Constructivist paradigms and more directly describe the practice of IS software engineering research, we introduce the Socio-technologist/Developmentalist paradigm which addresses the valuable contribution of software systems and associated

> processes to scientific knowledge building. […] Within the Socio technologist/Developmentalist paradigm, reality is technologically created. […] The supporting methodology is primarily developmental, […] The methodology's focus is on the technological innovations […] which are intended to affect individual and organizational experience in a positive manner. The Socio-technologist/Developmentalist paradigm allows the creation of new systems and transfer of technology to domains that need them. The keyword is 'creation'. IS can be viewed as social systems that are technically implemented" (pp. 172–173).

Peculiarities of expression aside—it is certainly not their new "paradigm" which would now suddenly "allow" us to create new systems, which we have always done—the passages quoted above seem to speak between the lines about a larger phenomenon which could perhaps be analysed deeper in the terminological framework of Niklas Luhmann's philosophy of systems. Also their ontological assertion that "reality is technologically created" (p. 172) could possibly be linked to Luhmann's thoughts about the autopoiesis of systems—this might perhaps become an interesting topic for future philosophical enquiry in this context.

Next we can find in (Gregg et al. 2001) the description of their above-mentioned "SERM" (Software Engineering Research Methodology) framework comprising a triad of "conceptual", "formal" and "developmental" aspects of a software engineering project. In this context it is interesting to read that "while development is sometimes equated with software engineering, in the SERM framework conceptualization or the theoretical grounding of the system requirements is suggested as the focal point of the research effort" (p. 174) and that "in this phase the theoretical grounding for the needs and requirements of the research effort are defined" (p. 174).

This strong focus on requirements elicitation reveals that we are confronted with a predominantly idiographic notion of "research", because the requirements for any software project are always the requirements for this project, which is historically unique, whereas the nomothetic quest for general principles or laws is not in the scope of erkenntnisleitendes Interesse (enquiry-guiding interests) behind such an approach. Nevertheless the authors have also acknowledged that "the accumulation of experiences and knowledge acquired during the systems development process represents a viable research goal in and of itself" (p. 175). Thus, in spite of its predominantly idiographic perspective, the methodology presented in (Gregg et al. 2001) at least concedes that there are also some nomothetic interests involved in software engineering research and development.

The remainder of that paper (Gregg et al. 2001) presents a meta-review of previously published papers (by various authors) with the intention of classifying those papers along the lines of the above-mentioned "SERM" framework. Finally it becomes clear that "in this paper we investigated research philosophies as a foundation for conducting software engineering research in the IS discipline. The impetus of our effort resulted from the inability to fit software engineering research comfortably into the established research paradigms from the social sciences" (pp. 180–181). Also the practical purpose of that paper is finally declared: "Thus it provides a useful metric for software engineering research in IS" (p. 181). The usefulness of that "metric" (strictly speaking, however, it is not 'metric' since it was not defined in a metric space) is somewhat debatable. Their systematic overall conclusion—"we find that for software engineering to qualify as rigorous research, it must address issues in at least two of the three phases [of the SERM framework]" (p. 175)—appears as rather 'scholastic' and comes to no surprise for any serious software engineer. Indeed, their 'scholastic' recommendation ("at least two of the

three…") does hardly reach beyond Snelting's well-known concerns about 'quackery' in software engineering research (Snelting 1998a, b) and does thus not add much to our science-philosophical understanding of software engineering as an industrial and academic discipline. The notion of "philosophical" in (Gregg et al. 2001) seems to be restricted to a few rather particular issues (e.g.: the publish-ability of research papers) and does not seem to attempt to reach out for 'typical' philosophical questions, such as questions about transcendental pre-conditions, questions about the 'Anfangsproblem' of a science, and the like.

The reason for the above-mentioned shortcomings is probably the too application-oriented, external view of software engineering in (Gregg et al. 2001) which fails to behold the discipline from 'inside'. A sheer 'socio-humanist' meta-theory of software engineering forgets the equally important issue of software engineering's 'engineering-ness'. Evidence in support of this critique can also be found in (Gregg et al. 2001):

> "To examine its utility and to illustrate the various approaches undertaken by IS researchers, the current software engineering research efforts in the IS domain were mapped to our SERM framework. We focused on software engineering research reported in seven leading IS journals: Decision Sciences, Decision Support Systems, Information and Management, Information Systems Research, Journal of Management Information Systems, Management Science, and Management Information Systems Quarterly" (pp. 175–176).

This choice of research focus is too restricted and does not even take into account such important and globally known software engineering venues like ICSE (Internat. Conf. Softw. Eng.), ESEC (Europ. Softw. Eng. Conf.), or the widely distributed journal "Software Practice and Experience". Contributions such as (Gregg et al. 2001) can thus only be regarded as interesting pieces in a much larger 'puzzle' of philosophy of software engineering, to which many further pieces must still be found.

3 Software Engineering Between Formalism and Empiricism

This section is focussed on two rather short but relevant position papers, co-authored by Manfred Broy, Dieter Rombach, and Frank Seelisch, on the topic of the position of software engineering amongst the sciences, especially the sciences of engineering (Broy and Rombach 2002; Rombach and Seelisch 2008). In terms of the classification introduced in Sect. 1, these papers were chosen for this essay because of their notable 'physicalist' perspective; however they including the perspective of the 'engineers' (which will be further discussed below in Sect. 4). Paper (Rombach and Seelisch 2008) especially is dealing with the problem that many of the software engineering research results do not make it into practice, whereby the gap between software engineering research and practice widens constantly. According to (Rombach and Seelisch 2008), the reasons for not making it into practice range from insufficient commitment to professionalization to insufficient consideration for practical scale-up issues and a considerable lack of empirical evidence regarding the benefits and limitations of new software engineering methods and tools. The major focus of their paper is to motivate the creation of credible evidence which in turn should allow for less risky introduction of new software engineering approaches into practice. According to (Rombach and Seelisch 2008), in order to overcome this progress-hindering lack of evidence, both software engineering research and practice have to change their 'paradigms'.

Rombach and Seelisch started their discussion with the notorious problem that "software engineering today, seen as a practically highly relevant engineering discipline, is not mature enough" and that most of the results from scientific or academic software engineering research are not finding their way into the industrial or commercial software engineering practice—in other words, that the knowledge gap between software engineering research and practice is widening (Rombach and Seelisch 2008). The authors argued that this gap between theory and practice is due to "a tremendous lack of empirical evidence regarding the benefits and limitations of new software engineering methods and tools on both sides".

This problem leads consequently to some more science-philosophical questions about the status of software engineering as an 'empirical' discipline—yes or no, and, if yes, to what extent:

> "The major claim of this work is that typical shortcomings in the practical work of software engineers as we witness them today result from missing or unacknowledged empirical facts. Discovering the facts by empirical studies is the only way to gain insights in how software development projects should be run best, i.e., insights in the discipline of software engineering" (Rombach and Seelisch 2008).

In this short paragraph only we can already detect two non-trivial science-philosophical problems, which Rombach and Seelisch did not mention themselves, namely: how to bridge the categorial gap from ontology ("discovering facts") to deontology ("how projects should be run best") without running into the notorious 'naturalist fallacy', and which types of investigations may be methodologically admitted as valid 'empirical studies' under the practical condition that software engineering can obviously not happen in the closed environment of a well-controlled chemical laboratory for the sake of the highest possible level of repeatability and experimental precision.

Rombach and Seelisch further identified two main reasons for the current practical problems of software engineering, namely "non-compliance with best-practice principles" and "non-existence of credible evidence regarding the effects of method and tools" (Rombach and Seelisch 2008), which leads them to the meta-disciplinary discussion of the relationship between software engineering and computer science (informatics), in continuation of an older contribution to this discourse by Broy and Rombach (Broy and Rombach 2002). About this relation we can read:

> "Computer science is the well-established science of computers, algorithms, programs and data structures. Just like physics, its body of knowledge can be characterized by facts, laws and theories. But, whereas physics deals with natural laws of our physical world, computer science is a body of cognitive laws", and "when software engineering deals with the creation of large software artifacts then its role is more similar to mechanical and electrical engineering where the goal is to create large mechanical or electronic artifacts. […] In this sense, software engineering can be seen as an analog set of methods for developing software, based on fundamental results from computer science" (Rombach and Seelisch 2008).

In this context, we can nicely illustrate the relationships between (computer) science, (software) engineering, and (programming) craft by a small example (which I had discussed with my colleague Markus Roggenbach some years ago); see also (Arageorgis and Baltas 1989) for comparison:

1. Science: There exists an algorithm A which has an average-case runtime complexity O(f(x))—proof thereof.
2. Engineering: There is a problem P which could be solved by algorithm A or by algorithm B. Algorithm A has an average-case runtime complexity O(f(x)), Algorithm B has an average-case runtime complexity O(g(x))—proof details are not needed, yet a well-informed decision between the choice of A and B for solving P is made.
3. Crafts: A programmer can implement algorithm A in a suitable programming language to build an executable computer program—no need to know anything about its average-case runtime complexity O(f(x)).

It is not the purpose of this essay to discuss the extent of structural and methodological similarity between computer science and physics as asserted in (Rombach and Seelisch 2008). Subjectively I believe that the alleged similarity between computer science and physics is (at least to date) more wishful thinking than observable reality, but that is a question for the philosophy of computer science, about which some volumes of publications already exist.[2] Here we are mainly concerned about software engineering and its relation to other disciplines, not with the meta-scientific disputes about those other disciplines themselves. Anyway, it should be clear that there must be some topical overlaps between the philosophy of computer science and the philosophy of software engineering in analogy to the topical relationships between computer science and software engineering themselves.

Rombach and Seelisch continued their argument about software engineering 'principles', especially the "general pattern of divide and conquer", which is a reductionist method of dividing a large problem into a set of smaller (and thus easier solvable) sub- and sub-sub- problems under the a-priori assumption that the whole will not be more than the sum of its parts. Terminologically one might criticize, perhaps somewhat pedantically, that a 'principle' in the terminology of (Rombach and Seelisch 2008) should be better called a 'maxim', such as not to confuse ontology and deontology, Sein and Sollen, world and method. Terminology aside, there arises the question about the limits of 'principles' (like "divide and conquer") themselves: Classically we have almost always tacitly presumed rather simple hardware structures, such as the Zuse/von-Neumann computer architecture, to be the material basis onto which software systems were to be developed. However, with the possible emergence of other hardware systems, such as massive-parallel cellular automata, in the not-too-far future our cherished 'principles' (like methodical reductionism) might perhaps falter. In the words of Victor Zhirnov and his co-authors: "When we consider the use of these systems [cellular automata] to implement computation for general applications, a vexing set of software challenges arise […] and we are aware of little work in this area" (Zhirnov et al. 2008). At stake is thus, from a science-philosophical perspective, the principle-ness of those software engineering concepts which were so far regarded as 'principles' under un-reflected, accidental historical circumstances (such as the technical and technological dominance of the Zuse/von-Neumann machine) and were taken for granted during several decades.

About the requested empirical evidence in software engineering, the authors stated "that having definitions and measures at one's disposal does not automatically guarantee that they be used. Enforcing their usage must be part of the project and can often only be accomplished by organizational changes or even changes in the working culture" (Rombach and Seelisch 2008). Here I can see a door into the domains of philosophical

[2] See the Philosophy of Computer Science website with many literature references at http://pcs.essex.ac.uk/.

ethics and philosophical anthropology with the question whether or not any change of our "working culture" is arbitrarily at our disposal, or if there is anything like a "human nature" by which our "working culture" might be constrained in a non-arbitrary manner.

In this context it is also interesting to note that Tom DeMarco, previously known as a strong supporter of metrics and quantitative measurements in the software engineering process (DeMarco 1982), has recently dissociated himself from his earlier positions and is now emphasising the importance of ethical concepts such as 'value' and 'purpose' beyond the borderlines of quantitative control and controllability (DeMarco 2009). Contrary to Rombach and Seelisch's remarks regarding software engineering and physics, De Marco claimed that

> "software development is inherently different from a natural science such as physics, and its metrics are accordingly much less precise in capturing the things they set out to describe. They must be taken with a grain of salt, rather than trusted without reservation" (DeMarco 2009).

In the above-mentioned dispute between the 'physicalists', the 'engineers' and the 'socio-humanists', DeMarco has declared his *new* socio-humanist position as follows:

> "I'm gradually coming to the conclusion that software engineering is an idea whose time has come and gone. I still believe that it makes excellent sense to engineer software. But that isn't exactly what 'software engineering' has come to mean. The term encompasses as specific set of disciplines including defined processes […] For the past 40 years […] we've tortured ourselves over our inability to finish a software project on time and on budget. But […] this never should have been the supreme goal. The more important goal is transformation, creating software that changes the world […] Software development is and always will be somewhat experimental. The actual software construction isn't necessarily experimental, but its conception is. And this is where our focus ought to be" (DeMarco 2009).

The word 'experimental' which DeMarco has used in his phrase above does clearly *not* mean the same kind of experimental-ness which the software 'physicalists' who think of experiments under well-controlled and almost laboratory-like conditions have in mind. DeMarco has used the phrase clearly synonynmous to 'heuristical' or 'explorative', just meaning to 'play around' and to 'try things out' in an almost Feyerabendian fashion without too much concern for strict methodology.

This topic of experimental-ness, as vaguely mentioned by DeMarco, will now lead us back into the discussion of the issues as they were contrarily emphasised by Rombach and Seelisch:

> "Laying the foundations of software engineering thus means to: state working hypotheses that specify software engineering methods and their outcome together with the context of their application, make experiments, i.e. studies to gain empirical evidence, given a concrete scenario, formulate facts resulting from these studies […], abstract facts to laws by combining facts with similar, if not equal, contexts, verify working hypotheses, and thereby build up and continuously modify a concise theory of software engineering as a theoretical building block of computer science" (Rombach and Seelisch 2008).

This quote clearly contains the core of Rombach and Seelisch's 'empiricist' or 'physicalist' software engineering philosophy. Formally we can clearly spot their adherence to the model of physics, with their mentioning of hypotheses, experiments, facts and laws.

But once again the science-philosophical question arises whether or not—and, if yes, to what extent—such a formal analogy is materially justified: For example, the authors did not clarify their notion of 'experiment' and its classical criterion of repeatability. How would software engineering 'experiments' be controlled and isolated from their environment (which was the classical precondition of repeatability)? Has any software engineering 'experiment' ever been repeated de-facto? And if full repeatability cannot be granted, what is then the degree of validity of the 'laws' which are supposed to emerge from such 'experiments'? Note that we are not talking here about the repetition of 'program-runs' of particular software products on particular computer platforms—we are talking about the repeatability of wide-ranging and far-reaching design and development processes. These are the kind of questions which philosophically minded software engineers from the 'physicalist' school must try to answer seriously—otherwise they would immediately run into the same kind of difficulties as Auguste Compte with his empiricist conception of sociology as the 'physics of society' more than 150 years ago.

Regarding Rombach and Seelisch's requirements about the "verification" of hypotheses in the domain of software engineering, see Karl Popper and the related discussions of software verificationism in (Northover et al. 2008). Even earlier than (Northover et al. 2008) there is a Popperian discussion of software engineering in (Hernandez-Orallo and Ramirez-Quintana 2000), a contribution which also challenges the wide-spread 'formalist' opinions that software programs would resemble mathematical-scientific 'theories' and that the software development process would be a deductive rather than an inductive one. (The ontological status of computer programs, whether they resemble mathematical-scientific 'theories' or not, is an issue of the philosophy of computer science rather than philosophy of software engineering.) On the other hand we can also find in (Hernandez-Orallo and Ramirez-Quintana 2000) the explicit normative request that software engineering should be predictable, in analogy to the predictability criterion of the scientificness of the natural sciences (Rombach and Seelisch 2008); however also in (Hernandez-Orallo and Ramirez-Quintana 2000) the question remained un-answered how such predictability in a more 'scientific' type of software engineering could be method(olog)ically and epistemologically guaranteed.

Last but not least an academic problem re-arises from (Rombach and Seelisch 2008), namely whether software engineering should be categorized as a sub-discipline of and within computer science, or whether software engineering should be regarded as a discipline in its on right, with computer science as its basis and auxiliary discipline. The authors seem to oscillate between these two classification alternatives and do not commit themselves to a final decision in this regard. Rightly, however, it was pointed out in (Rombach and Seelisch 2008) that the immaturity of software engineering as an "engineering" discipline is closely related to the following practical shortcomings and flaws: "Clear working hypotheses are often missing. There is no time for, or immediate benefit from empirical studies for the team who undertakes it. Facts are often ignored […]" and "often replaced by myths, that is by unproven assumptions", which leads the authors to conclude that "up to now, a concise, practical theory of software engineering does not exist" (Rombach and Seelisch 2008).

The remainder of that paper deals with particular examples of popular software engineering myths, and the suggestion of some concrete research questions to stimulate research programmes with the aim of eventually being able to replace those myths (for example: about the usefulness and effectiveness of software testing) by more solid and better corroborated long-term knowledge.

As Rombach and Seelisch (2008) was directly related to an earlier contribution by Broy and Rombach (2002) it makes sense now to round up this section of this essay by taking

into account a few additional aspects from Broy and Rombach (2002), in the light of which the discussions of Rombach and Seelisch (2008) can be even better understood. For the remainder of this section, all verbatim quotations will be taken from Broy and Rombach (2002), and only the according page numbers (p. xyz) will be given after every quotation. The central issue of Broy and Rombach (2002) is that software engineering aims at the development, maintenance and evolution of large software systems in an engineering manner, thereby using well-established, systematic principles, methods and tools. 'Role models' therefore are, according to Broy and Rombach (2002), the methods from the classical engineering sciences, such as machine engineering, electrotechnics, or civil engineering. As an initial definition of software engineering we can find in Broy and Rombach (2002):

> "Purpose of the industrial engineering of software is the development of large scale software systems under consideration of the aspects costs, deadlines, and quality" (p. 438).

Already this initial definition by Broy and Rombach could lead us further into a discussion of general philosophy of technics, namely what constitutes an 'industry': is 'industry' a large number of people and how they organise their work in a Taylorist/Fordian manner, or is it the application of accumulated 'capital' (i.e.: machinery and automated tools) in the production process? Is the term 'software industry' materially justified if we observe that most software producing enterprises in our days are in fact hardly any larger—in terms of numbers of workers—than the workshop of a traditional craftsman and his helpers? Is 'industry' a rather misleading metaphor in this context which does not do justice to the actual way in which software is actually being produced? Or are we here already dealing with a completely new notion of the term 'industry' itself, which is now no longer associated with traditional images of iron, smoke, and armies of workers marching through the gates of their factory? Those would be interesting new questions for a general philosophy of technics and technology; however these questions cannot be further discussed within the scope of this essay.

The general theme of Broy and Rombach (2002) was the degree of difference and similarity between software engineering and other engineering disciplines, on the basis of the immateriality of software as 'virtual thing'. Particularly they mentioned in this context the difficulties arising from the software's abstractness, the software's multiple aspects of syntax and semantics, the intrinsically hard-to-understand, complex and dynamic system behaviour to which software is only a static description, as well as the absence of natural physical constraints as protection against weird constructions. On these premises Broy and Rombach (2002) concluded—whereby their conclusions can still be regarded as valid today—that software engineering as a discipline has not yet reached the degree of professional maturity which classical engineering disciplines have already reached, that "the discipline is still struggling with its self-understanding" (p. 439), and that "foundations and methods are partially still missing" (p. 438).

On the ontological status of software, which was more comprehensively discussed elsewhere (Northover et al. 2008), it was rightly pointed out by (Broy and Rombach 2002) that software enhances the intellectual abilities of its users, whereas material hardware enhances the bodily abilities of its users; the latter thought leads straight back to the classical machine theory, formulated already in 1877 century by Kapp in his Grundlinien einer Philosophie der Technik. Related to Heidegger's notion of 'Zeug' (equipment), a software-plus-computer system has elsewhere been dubbed as 'Denkzeug'

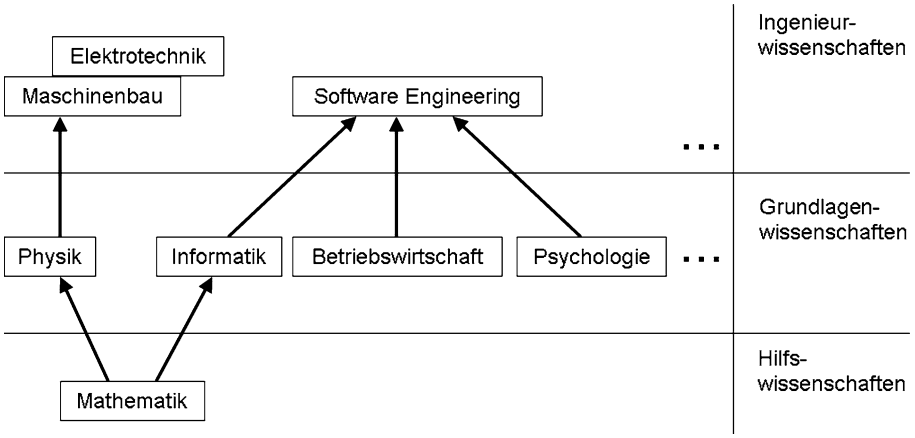

Fig. 1 Classification of software engineering according to Broy and Rombach (2002)

(think-equipment),[3] in contrast to the Werkzeug (work-equipment) of the material world. The same philosophical thought, though not through the same words, was thus expressed by Broy and Rombach (2002).

The most interesting part of Broy and Rombach (2002) in the context of this essay is their attempt to classify software engineering in a category of related disciplines, with the purpose of contributing to the self-understanding (the lack of which they had previously identified) of the software engineering discipline. Their classification scheme is interesting enough to keep us occupied for the remainder of this section. The main question is, like in Rombach and Seelisch (2008), whether software engineering is included as a sub-field of informatics (as it is currently enshrined in the curricula of many universities, with software engineering courses being lectured as part of the informatics degree), or whether software engineering is a field on its own with informatics as its separate basis. Also Broy and Rombach (2002) did not reach a decisive conclusion in this regard, though they clearly tend towards the latter solution with the analogy argument: "Imagine that physicists with a specialisation in mechanics would be employed as mechanical engineers!" (p. 441). On the other hand one could ask back, naively, if that is not only an argumentum ad hominem, especially if we take into consideration that physicists are de-facto employed in all sorts of jobs and positions, including positions as programmers in the software industry. Anyway, the basic classification by Broy and Rombach (2002) looks as depicted in Fig. 1.

In Fig. 1, taken directly out of (Broy and Rombach 2002), we can see three categories of sciences, namely auxiliary sciences (Hilfswissenschaften), foundation sciences (Grundlagenwissenschaften), and engineering sciences (Ingenieurwissenschaften). Software engineering appears here in the latter category, with electrical and mechanical engineering as some examples of 'sister' sciences. Sciences so different from each other as physics, informatics, and psychology appear here all in the category of foundation sciences (middle layer of Fig. 1), whereas mathematics appears in the bottom layer of Fig. 1 as auxiliary science; note that a similar viewpoint had already been expressed in (Hoare et al. 1998).

[3] The term *Denkzeug* is now so popular in German philosophical language that I was not able to trace its origin, but I know that it was also used by Walter Zimmerli in his philosophy of (computer) technology in the 1990s.

There are some obvious omissions in the diagram of Fig. 1 which do not need to be discussed any further, for example: mathematics is obviously also an auxiliary science to economics (Betriebswirtschaft in the diagram), and economics must surely be taken into account not only for commercial software engineering (as shown in Fig. 1) but also for commercial mechanical engineering (not depicted in Fig. 1). More interesting is the question here why psychology does only point to software engineering but not to mechanical engineering—are there not any psychological issues to be considered when any potentially dangerous apparatus shall be constructed?—and why mathematics does not point directly also to the engineering sciences (only indirectly via the foundation sciences)? This diagram can only suggest that (Broy et al. 2002) seem to believe that whenever a software engineer is applying mathematics, then he is actually already doing informatics, not software engineering any more. This is in contrast to other, more 'formalist' schools of thought, according to which mathematical methods are genuine software engineering methods,[4] not only informatics-supporting methods at the basis of software engineering.

As far as the mathematicalness of engineering in general and software engineering in particular is concerned, it was Tom Maibaum, who has recently pointed out two further issues: On the one hand he asserted that "engineers calculate, mathematicians prove",[5] which means that engineers are mostly applying 'distilled' handbook-mathematics which had been developed outside the world of engineering (Maibaum 2008). On the other hand, the branch of mathematics most relevant to classical engineering is the infinitesimal differential calculus (as it was developed since Leibniz and Newton) whereas the branch of mathematics most relevant to software engineering is discrete mathematics, set theory and formal logics (Maibaum 2008).

Of course it is necessary to 'calculate' in order to 'prove', and of course also an engineer (not only a mathematician) wants to 'prove' (by means of calculation) that some design concept or model is consistent and feasible before the according artefact is produced. But that was not Maibaum's point in this discussion. The issue here is: Whereas the classical engineering disciplines already have a large volume of distilled handbook-mathematics available for application, a corresponding formula-handbook readily applicable for software engineering calculations is—at least at the moment—nowhere to be seen, which must be highly problematic for those philosophers of science, who—like Kant—tend to bind their notion of 'scientificness' very strongly to the condition of 'mathematicalness'.

The notion of 'mathematicalness' itself is still problematic in software engineering, too, as it was comprehensively explained in (Kondoh 2000): it seems as if the old Grundlagenstreit in pure mathematics (between the logicists/formalists and the anti-logicists during the early twentieth century) is now experiencing an 'applied revival' in the context of software engineering, whereby many software engineers would like to operate as 'mathematically' as possible (in order to resemble the classical engineers with their status of professional maturity) but cannot always agree on the concrete forms and methods of mathematics which any 'mathematicalness' would entail (Kondoh 2000). Here I can actually see the some deeper philosophical problems which were summarily characterised as "immaturity" and "lack of foundations" in (Broy and Rombach 2002).

Returning to the discussion of Fig. 1, informatics—here regarded as foundation science to software engineering—is also an issue in itself. As it was rightly remarked in (Broy and Rombach 2002), informatics itself is not a monolithic science. Instead, informatics has various parts and aspects, such that "informatics structures itself [further] into informatics

[4] See for example the organisation Formal Methods Europe (FME), online at http://www.fmeurope.org/.

[5] Maibaum attributed this quotation to Tony Hoare.

as foundation science and informatics as engineering sciences" (p.441). If I may give two simple examples: A formalised theory of Chomsky grammars and a large body of empirical, practical knowledge about the design and development of operating systems are both included in the wide domain of informatics, whereby the formal grammars would be 'mathematics' whereas the operating systems would be 'engineering' (in this simplified picture). Academically this diversity within the field of informatics itself is reflected by the distribution of informatics departments across the faculties at various universities—typically (with few exceptions) either in faculties of mathematics and natural sciences, or in faculties of engineering and technology.

The problem now with the classification by (Broy and Rombach 2002), as depicted in Fig. 1, is that it treats informatics too simplistically as mathematics-based foundation science, thereby ignoring other, engineering-related sub-sciences of informatics, such as the above-mentioned operating systems.

Consequently, a string of further problems emerges: If software engineering has been "lifted out" of the domain of informatics into the domain of engineering, should then not—by analogy—also the field of operating systems be lifted out of informatics into the domain of engineering?—and so on, until informatics, stripped bare of all practical aspects would be nothing more than formalised Chomsky grammars and discrete algebra? On the other hand, if we would leave the operating systems where they are, namely in informatics, would not then be also the operating systems, according to Fig. 1, belong to the foundations of software engineering?

Though this is not wrong, it is only half of the picture: In fact, operating systems are, in the end, nothing else but large software systems, which means that software engineering should also be listed, vice versa, as the foundation science for operating systems (Northover et al. 2008) within the domain of informatics. In Fig. 1, however, the link between informatics and software engineering is only unidirectional. In this context it is also interesting to note that the mutual dependency between informatics and software engineering, or parts thereof (which is not depicted in Fig. 1), corresponds quite well with the old Constructivist argument about the mutual dependency between physics and engineering of technical artefacts to be used for physical measurements—in contrast to the classical interpretation of physics as the unidirectional foundation of engineering as it was also depicted by Broy and Rombach (2002) in Fig. 1.

Let us now ask the question: What is it that "lifts" software engineering up onto the level of engineering, above informatics, in Fig. 1? The answer, given by Broy and Rombach (2002) is: "experience", such that, for example: "A new method is perhaps a remarkable result of informatics, but without robust empirical experiences about its effectivity and limits of applicability it is not a contribution to software engineering!" (p. 446). This position is schematically depicted in Fig. 2, which is also taken directly from Broy and Rombach (2002).

Problematic with the above-mentioned quote is the tacit equation 'Software Engineering is Informatics plus Empiricism', which reduces by implication the science of informatics to purely rationalist, non-empirical science. On the other hand, informatics was compared to physics in Fig. 1 of Broy and Rombach (2002)—would they then, by analogy, also assert the equation 'Engineering is Physics plus Empiricism', thereby implicitly reducing physics to pure rationalist scholastics (as it has been historically the case throughout the Latin middle ages)?

Interesting in Fig. 2 is, however, the terminological distinction between Software Technics (Softwaretechnik in Fig. 2) and Software Engineering. This terminological distinction should not only ring a bell in the house of general philosophy of technics and

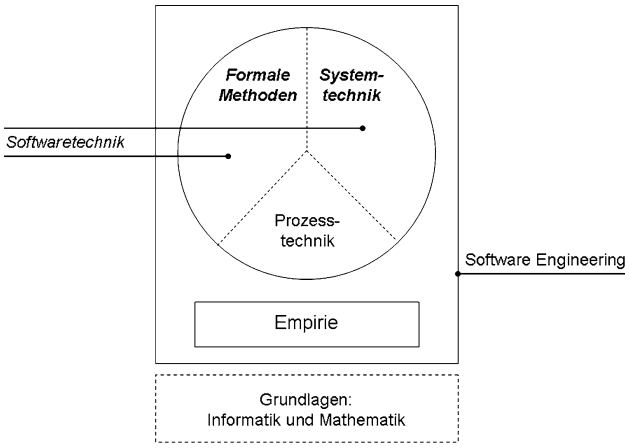

Fig. 2 Components of software engineering according to Broy and Rombach (2002)

technology; this distinction is also interesting because it breaks the usual translation-synonymy between the English term 'Software Engineering' and the equivalent German term 'Softwaretechnik'. Braking this usual synonymy, (Broy and Rombach 2002) have indeed made a remarkable step towards a deeper understanding of software engineering as a theoretical and practical discipline. Software Technics, as depicted in Fig. 2, is now regarded by Broy and Rombach (2002) as a part of Software Engineering, whereby other disciplines (like economics, general systems and process theory, etc.) are needed to bridge the gap between technics and engineering.

It is debatable why only the two components 'formal methods' (Formale Methoden in Fig. 2) and 'systems technics' (Systemtechnik in Fig. 2) should constitute the core of software technics (Softwaretechnik in Fig. 2), but the terminological refinement is remarkable and, so I believe, in principle fruitful. Though Broy and Rombach (2002) did not mention Kondoh (2000), their distinction between 'software technics' and 'software engineering' is quite similar to Kondoh's distinction between "Abstract Software Engineering as a research field and Precision Software Engineering as an engineering discipline" Kondoh (2000); however Broy and Rombach (2002) and Kondoh (2000) clearly differ about the validity of the physics/computer-science analogy.

Now having 'software technics' and 'software engineering' available as distinguishable terms in our terminology, one question arises immediately, namely: what about 'software science'? This term is indeed in use elsewhere,[6] but (Broy and Rombach 2002) did not attempt to work it explicitly into their terminological schema, though various questions and problems regarding the scientificness of the discipline had already been discussed (Snelting 1998a; Gregg et al. 2001).

In a first and provisory attempt to complete the schema of Broy and Rombach (2002), I suggest to define—however debatable—'software science' as a comprehensive science about software engineering and software technics (software technology), comprising not only theoretical and practical aspects of software (its conceptualisation, its construction, its application) itself, but also meta-scientific and methodological self-reflexions.

[6] See for example the European association of software science and technology (EASST), online at http://www.easst.org/.

This definition of 'software science', though perhaps still premature, fulfills at least the minimum condition of definition-ness by distinguishing 'software science' from other sciences. At the same time the definition wide enough to comprise new (and much-needed) research activities into the direction of cause-effect-analyses, with the purpose of shedding some brighter light onto the software "myths" mentioned in Rombach and Seelisch (2008), as well as onto the opinion-based software "fashions" and "gurus" mentioned in Broy and Rombach (2002). The definition of above also comprises Kondoh's notion of "Abstract Software Engineering" (Kondoh 2000).

Of course such 'software science' cannot be value-free: it is indeed driven by the values of 'better' insight and 'better' understanding in almost the same way in which engineering is driven by the value of designing and producing 'better' products, whereby the word 'better' indicates the presence of an underlying ethos.

4 Radical Versus Normal Design

In Sect. 2 we had discussed a 'socio-humanist' approach to software engineering which had by-and-large ignored the engineering-ness of that discipline. In Sect. 3 we had discussed a rather 'physicalist' approach to software engineering which took its engineering-ness in principle for granted and mainly requested some further methodological (including: mathematical and experimental) maturation in order to accelerate the historical process of "catching up" with the other (older) engineering disciplines. In this section I shall discuss yet another position which neither ignores the question of engineering-ness, nor takes it for granted as an established fact. This position was recently expressed by Tom Maibaum, especially in his lecture on "Formal Methods versus Engineering" (Maibaum 2008). The key distinction here is the one between "radical Design/Engineering" versus "normal Design/Engineering", which is obviously derived from Kuhn's well-known distinction between 'normal science' and 'science in crisis'.

According to Maibaum (2008), who also referred to Rogers' Nature of Engineering (Rogers 1983), the distinction between radical and normal design/engineering is closely related to the difference between systems and devices, whereby radical design creates systems, whereas normal design produces devices. In this terminology, a 'device' has to be understood as something 'usual' and 'well known', the production of which does not confront us with any fundamental challenges any more and only requires small-scale improvements every now and then, in analogy to the 'puzzle solving' in Kuhn's 'normal science'. Motor cars are typical examples of devices in this sense. Since about hundred years they are being designed with four wheels on a more or less rectangular chassis and a combustion engine sitting usually between the front wheels, such that the differences between a car model of the year 2009 and a car model of the year 1929 can be found in all the details but not in its fundamental principles. On the contrary a 'system' is, in this terminology, any product that is not a device; a system thus carries all the flavour of newness, novelty, complicatedness, challenge, difficulty, irritation, non-manageability, and the like.

On these premises, said Maibaum, engineering research aims at turning systems into devices. Engineering research thus aims at making a transition from radical to normal design or normal engineering. But what are the operational pre-conditions of normal engineering? At least the following three can be identified: a high level of professional specialisation, a high level of component standardisation, and a well established, standardised methodology with readily available and applicable 'handbook knowledge',

whereby it is also obvious that these three conditions will mutually support and reinforce each other.

For example, a railway engineer will typically not design a production plant for agricultural fertilisers, but the railway engineer will also typically not design completely new bolts and screws for every new railway project; there are standard bolts and screws in various sizes readily available to be used in the project. On these premises Maibaum argued that software engineering is still on the developmental level of crafts, far away from actually being engineering: "Unfortunately there is not all that much engineering material to communicate to students" (Maibaum 2008).

To find evidence in support of this verdict, we have to revisit those three criteria of 'normal engineering' and see to what extent they match our current reality of software engineering in its industrial practice as well as in its academic (re)presentation:

1. Imagine we would only have 'hardware engineers' in contrast to 'software engineers', and that such an imaginary 'hardware engineer' would be expected to design railway bridges, plants for the production of agricultural fertilisers as well as electronic micro chips. Whereas such a suggestion would be deemed insane in the world of classical engineering, in the domain of software engineering this is, to date, mostly the case (and has its ultimate basis in the concept of the Turing machine as the 'universal machine').
2. There is no shortage of academic research and conferences on the topic of software components and component-based software development.[7] However, the acceptance of these ideas in the industrial software business is only slow, the level of software re-use from project to project is de-facto low, and most software modules in industrial or commercial software projects tend to be developed from scratch again and again. This is only one example of the widening "gap between research and practice" mentioned in Rombach and Seelisch (2008).
3. With regard to the third criterion of above, one can find in the history of software engineering three 'success stories' about particular classes of software systems which can now be produced quite easily following standard approaches and procedures, such that they can be regarded as having reached "device" status in the terminology of above. These three well-understood classes of software systems are compilers, operating systems, and relational database systems. Here, however, we have made the academic mistake of removing these successful 'device' types out of our regular software engineering curriculum, treating them as something completely different, such that software engineering (minus compilers, databases and operating systems) is still doomed to appear as the un-known domain of 'radical design', with its consequent dominance of opinion-based "gurus" and "schools". (Artificial Intelligence research, by the way, had a structurally similar problem: For every puzzle solved, the goal posts of the very definition of 'intelligence' were ex-posteriori shifted further, such that Artificial Intelligence continued to appear as un-intelligent as ever.)

As a consequence of this discussion it should have become clear by now that another objective of software science, as I have tried to define it in Sect. 3, should be the identification of further, suitable (sub) classes of software systems, which can then be treated as 'devices' (like compilers, databases, and operating systems) in procedures of 'normal'

[7] See for example *CBSE*, the annual symposium on component based software engineering, published regularly in the *Lecture Notes in Computer Science*.

engineering, if software engineering wants to make the transition from 'crafts' to genuine 'engineering'.

Sociology of science will of course tell us that such a transition from radical to normal design in software engineering will face some resistance from those "gurus", whose status of guruness depends exactly on the continuation of the current predominance of the radical design situation—see for example: "creating software that changes the world" (DeMarco 2009)—however this discussion is no longer in the scope of this essay. I am also not (yet) able to answer the question whether such a transition from crafts to engineering, from radical to normal design, would make a 'paradigm shift' in the Kuhnian sense of the term, though I tend to believe that such a transition would come gradually and evolutionary, thus missing the 'revolutionary' criterion with which a Kuhnian paradigm shift is usually associated (Northover et al. 2008).

Anyway, at this point of our discussion of Maibaum (2008) we can now also see the notorious quarrel between the school of the 'Agile Alliance' versus the school of 'Rigorous Processes', as it was described in Northover et al. (2008), in a considerably clearer light: the agile methods obviously belong to the exciting domain of radical design of 'systems', whereas the rigorous processes (as they are strongly advocated, for example, by the software engineering school of Japan) belong to the calmer and less spectacular domain of normal design and engineering of software products with the ontological status of 'devices'.

Last but not least, at the end of this section, it might also be interesting to note from an onto-philosophical perspective that those 'devices' (of normal engineering) are coming much closer to Heidegger's early notion of Zeug (which Heidegger had characterised by its familiarity) than the un-familiar 'systems' (in the little-known domain of 'radical design'), which seem to be closer related to Heidegger's later notion of the Gestell (i.e.: the technological form of being which always haunts and challenges us in our modern existence). Put into Heideggerian terms, normal engineering would thus aim at a re-transformation of Gestell to Zeug, however software engineering, according to Maibaum, has (by-and-large) not yet reached this status of normal engineering.

5 Norm and Form

The main difference between science and engineering, from the perspective of theoretical philosophy, is the difference between 'what is' and 'what should be'. Engineering is, in the end, a normative discipline, for example: across this river there should be a bridge, it should withstand a strong storm, and it should allow for a traffic capacity of 1,000 vehicles per hour. Software engineering is not different from other fields of engineering in this regard. This normative aspect which (software) engineering shares with the old discipline of jurisprudence was nicely explained by Luigi Logrippo, in his recent paper on normative systems (Logrippo 2007), which I have chosen as the last 'paradigmatic' example to be mentioned in this essay-review. The papers which I had discussed in the previous sections did not sufficiently mention this normative aspect of engineering, such that this essay-review would be rather incomplete if it would finish without any remarks in this regard:

> "It is argued here that there are many concepts and methods in common between policy systems used in Information Technology and Jurisprudence, i.e. legal theory. These concepts are found in the research area of 'normative systems' which encompasses them and provides a framework for unifying research. It is further

argued that advantages can be accrued to both research areas by favoring interchanges of methods and principles in this unifying framework. A distinction is made between norms in rule-style and norms in requirements-style. Issues of completeness, consistency and conflict are considered. Concepts that are useful in this research area include defeasible logic and ontologies. Useful tools are theorem provers and model checkers" (Logrippo 2007).

What Logrippo has called the 'requirements style' are norms of the form which we can find, for example, in the codex of Moses: 'you shall not steal'. What Logrippo has called the 'rule style' are norms of the form which we can find, for example, in the codex of Hamurabi: 'if anyone steels cattle or sheep… then the thief shall pay 30-fold…' (Logrippo 2007). Thus, rule-style norms, with their more precise specification of applicability conditions and action consequences, are formally more similar to the laws of nature as they are formulated by the natural sciences and are then productively used in engineering. Consequently, what we can call an implementation of a software system in software engineering must be a valid transformation from requirements ('Moses-style' norms) into behavioural system patterns ('Hamurabi-style' norms) by which the implemented system is then fully characterized (Logrippo 2007). This transformation must be carried out in such a way that conflicts and inconsistencies are avoided and resolved as far as possible (Logrippo 2007). Here we can indeed find a meeting point between the philosophies of science and engineering, and the philosophies of norms and laws, or, in other words: a meeting point between ontology and deontology, with methodology as the binding 'glue'.

6 Summary, Conclusion, Outlook

In this essay-review, intended as continuation of (Northover et al. 2008), I have discussed several science-philosophical issues around the emerging disciplines of software engineering and software science. The question about the extent of 'engineering-ness' in software engineering has been a main theme throughout this essay, though other issues, including terminological ones, were also addressed. This essay-review presented and discussed relevant recent contributions to the philosophical and methodological discourse within the discipline of software engineering, about the scientificness of this discipline, also in comparison to other, related disciplines. The main problems in this context were exposed and explained—though not solved—especially for those readers who are not software engineers themselves. The discussion in this essay-review was focussed on typical 'socio-humanist' versus 'physicalist' approaches to the understanding of software engineering as a modern science. An original contribution of this essay-review was a preliminary definition of the term 'software science', subject to further discussion and critique. Similar to what Kuhn has described in his history of science, these fields of software engineering and software science are still in need of further science-philosophical clarification, for the sake of a better understanding and self-understanding of this still 'young' discipline between theory and practice, academia and industry.

There exists an ongoing meta-scientific, science-philosophical debate 'behind the scenes' on software engineering amongst philosophically minded software engineers themselves, though not yet sufficiently amongst professional philosophers of science. Several positions and 'schools' participating in this debate have been identified in this essay, and several example papers, representative of those 'schools', have been discussed. In terms of Kuhn, however, it has not yet been clarified whether this multiplicity of

software engineering 'schools' would indicate a 'pre-paradigmatic' state of affairs (i.e.: 'science' not yet established), or an 'inter-paradigmatic' state of affairs (i.e.: science 'in crisis'). The long-term goal of this discussion is to establish a special—yet interdisciplinary—philosophy of software engineering, being part of general philosophy of science as well as part of general philosophy of technics and technology, in analogy to other already existing special philosophies of science, such as the philosophy of biology (meta biology), the philosophy of mathematics (meta mathematics), and the like. In the table of Fig. 3 I have summarised four examples of engineering disciplines, including their 'parent sciences' and some further important characteristics. This table shows clearly that software engineering is less than other ('classical') engineering disciplines constrained by the laws of nature; this allows for even more freedom of human ingenuity and consequently leads also to an even greater potential of errors and mistakes. Note, however, the grammar constraint: Programming languages must always have a well-defined formal syntax, and they must also be inherently consistent from a logical point of view.

Moreover, there are further differences between different engineering disciplines as far as their efforts in the categories design and (re-) production are concerned. Take, for example, the design and construction of a bread-toaster in the classical manufacturing industry: In a first process, an industrial designer is designing a prototype of the bread-toaster, which is a comparatively simple thing. Then, in a second—and considerably more complicated—process the production engineers have to design the machinery by which the components of the bread-toaster can be produced and assembled in a factory for mass-production. In software engineering, on the contrary, all mental efforts have to be invested into the design of the software prototype, which has a considerably higher structural complexity than the bread-toaster of this example. Mass-multiplication of the software product, on the other hand, comes almost for free: because it is software (i.e. immaterial information) it can simply be copied on almost any available computer. This also implies that software, as 'entity', comprises both model aspects and product aspects. Figure 4 summarizes various engineering disciplines, including software engineering, in a two-dimensional matrix with respect to their efforts in design and (re-) production.

Engineering Discipline	Mechanical Engineering	Chemical Engineering	Electronic Engineering	Software Engineering
Product	Machines	Chemical Substances	Computer Components and Circuits	Computer Programs
Type of Product	material	material	material	non-material
Parent Science	Classical Physics	Chemistry	Electro-Physics	Computer Science
Type of Parent Science	empirical	empirical	empirical	semi-empirical
Constraints	Law of Nature	Law of Nature	Law of Nature and Logic	Law of Logic and Grammar

Fig. 3 Classification and characterisation of several engineering disciplines

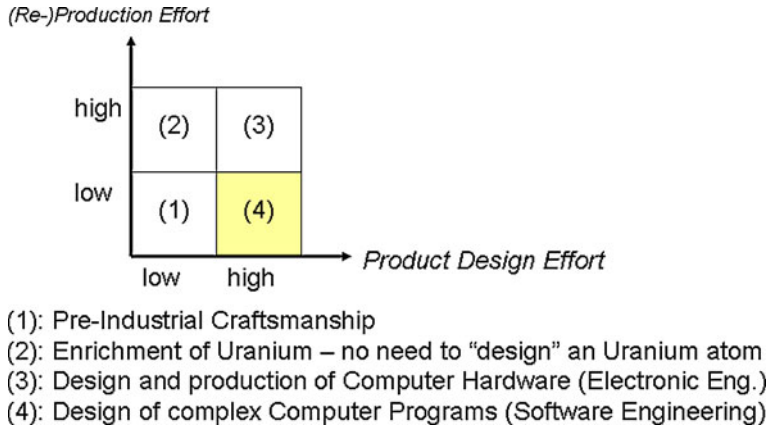

Fig. 4 Comparison of design and production efforts in different engineering areas

Towards a future philosophy of software engineering, this essay can only be regarded as yet another very small step. Thus this concluding section is actually far away from being a definitive conclusion of the actual matter. Counter critique to my critique is expected, and I especially hope that professional philosophers of science—which I am not—will find this topic interesting enough to be included into their discourses. This new branch in the tree of philosophy of science can only grow and bear fruits if philosophically minded software engineers (such as the ones mentioned and quoted throughout this essay) and technologically interested philosophers begin to communicate with each other in an interdisciplinary discourse.

Open issues and fundamental problems, which are still being debated inconclusively so far, are (amongst others): Where is the position of software engineering between 'science' and (classical) 'engineering'? Does the 'scientificness' of software engineering resemble 'physical' scientificness (nomothetic empiricism) or rather 'mathematical' scientificness (formalism)? In other words: is (or should be) software engineering rather an 'inductive' or rather a 'deductive' discipline?—or maybe rather an 'idiographic' and 'social' discipline, as it is claimed by the 'socio-humanists' in this field?

Moreover, as far as the above-mentioned 'mathematicalness' of software engineering is concerned: does (or should) this mean only 'fully axiomatised calculi' (position of formalism, logicism, logico-positivism) for mechanized (automated) deductions? Or can (or should) 'mathematicalness' in software engineering also include some 'weaker', more traditional forms of 'mathematicalness' (position of pre- or anti-logicism), as advocated, for example, by Raymond Boute or Hidetaka Kondoh (Kondoh 2000)? In this context it might be also be historically interesting to observe a kind of 're-incarnation' of the old 'Grundlagenstreit' of mathematics in the new application domain of software engineering.

Further new questions, issues and problems in the philosophy of software engineering will surely emerge as long as this discourse continues in 'co-evolution' with the newest results and developments in its object (or subject) discipline.

Acknowledgments Thanks to the students of my Software Engineering seminar in 2008 at the University of Pretoria for interesting discussions on the context of this article. Thanks also to Tom Maibaum for inspiring conversations during the ICFEM International Conference on Formal Engineering Methods in Kitakyushu, Japan, October 2008. Also the fruitful discussions with my colleagues Derrick Kourie and Morkel Theunissen are gratefully acknowledged. I also thank Markus Roggenbach for an example which I

have used in Sect. 3. Last but not least thanks to the editors and reviewers of this journal for their thoughtful feedback and helpful comments on the earlier drafts of this contribution, as well as to the production office for their professional typesetting of the manuscript.

Appendix: Further Reading[8] in the Wider Context of this Discourse

Armour, P.G. (2006). The Business of Software. *Communications of the ACM, 49/9*, 15–17.

Bergin, T.J. (2007). A History of the History of Programming Languages. *Communications of the ACM, 50/5*, 69–74.

da-Cunha, A.D. & Greathead, D. (2007). Does Personality matter?—An Analysis of Code-Review-Ability. *Communications of the ACM, 50/5*, 109–112.

Feitelson, D.G. (ed.) (2007). Experimental Computer Science. *Communications of the ACM, 50/11*, 24-59.

Floridi, L. (1999). *Philosophy and Computing: An Introduction*. Routledge.

Florman, S.C. (1996). *The Introspective Engineer*. St. Martin's Griffin.

Fujita, H. & Pisanelli, D. (eds.) (2007). New Trends in Software Methodologies, Tools and Techniques. *Proceedings of the 6th SOMET Conference*. IOS Press.

Glass, R.L. (2007). One Man's Quest for the State of Software Engineering's Practice. *Communications of the ACM, 50/5*, 21-23.

Jeffries, R. & Melnik, G. (Eds.) (2007). Test-Driven Development. *IEEE Software, 24/3*, 24– 83.

McBride, M.R. (2007). The Software Architect. *Communications of the ACM, 50/5*, 75–82.

Rajlich, V. (2006). Changing the Paradigm of Software Engineering. *Communications of the ACM, 49/8*, 67–70.

Sugumaran, V. & Park, S. & Kang, K.C. (eds.) (2006). Software Product Line Engineering. *Communications of the ACM, 49/12*, 28–89.

Turski, W.M. (2000). Essay on Software Engineering at the Turn of the Century. *Lecture Notes in Computer Science, 1783*, 1–20.

Vincenti, W.G. (1993). *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*. Johns Hopkins University Press.

References

Arageorgis, A., & Baltas, A. (1989). Demarcating technology from science: Problems and problem solving in technology. *Zeitschrift für allgemeine Wissenschaftstheorie, 20*(2), 212–229.

Broy, M., & Rombach, D. (2002). Software engineering: Wurzeln, Stand und Perspektiven. *Informatik Spektrum, 16*, 438–451.

DeMarco, T. (1982). *Controlling software projects: Management measurement and estimation*. Yourdon Press: Prentice Hall.

DeMarco, T. (2009). Software engineering: An idea whose time has come and gone? *IEEE Software, 26*(4), 95–96.

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.

[8] Most of the papers listed in this appendix are written in a rather non-technical style and should thus be understandable also for readers not familiar with the field of software engineering. Note that I do *not* claim this to be a complete list of all the relevant literature in the context of this essay-review; this list should rather be taken as an "interesting selection" of recommendable and easily accessible titles, which are suitable for pointing the non-expert reader to some key issues in the wider area of software engineering.

Hernandez-Orallo, J., & Ramirez-Quintana, M. J. (2000). Software as learning-quality factors and life-cycle revised. *Lecture Notes in Computer Science, 1783*, 147–162.

Hoare, C. A. R., & He, J. (1998). *Unifying theories of programming*. London: Prentice Hall.

Kondoh, H. (2000). What is 'Mathematicalness' in Software Engineering?—Towards precision software engineering. *Lecture Notes in Computer Science, 1783*, 163–177.

Logrippo, L. (2007). Normative systems: The meeting point between jurisprudence and information technology? In H. Fujita & D. Pisanelli (Eds.), *New trends in software methodologies, tools and techniques* (pp. 343–354). Amsterdam: IOS Press.

Maibaum, T. (2008). *Formal methods versus engineering*. Proceedings of the First International Workshop on Formal Methods in Education and Training, at the ICFEM International Conference on Formal Engineering Methods, Kitakyushu, Japan.

Northover, M., Kourie, D. G., Boake, A., Gruner, S., & Northover, A. (2008). Towards a philosophy of software development: 40 years after the birth of software engineering. *Zeitschrift für allgemeine Wissenschaftstheorie, 39*(1), 85–113.

Rogers, G. F. C. (1983). *The nature of engineering*. Palgrave: Macmillan.

Rombach, D., & Seelisch, F. (2008). Formalisms in software engineering: Myths versus empirical facts. *Lecture Notes in Computer Science, 5082*, 13–25.

Snelting, G. (1998a). Paul Feyerabend und die Softwaretechnologie. *Informatik Spektrum, 21*(5), 273–276.

Snelting, G. (1998b). Paul Feyerabend and software technology. *Software Tools for Technology Transfer, 2*(1), 1–5.

Tichy, W. F. (2007). Empirical methods in software engineering research. Proceedings 4th IFIP WG 2.4 Summer School on Software Technology and Engineering, Gordon's Bay, South Africa.

Zhirnov, V., Cavin, R., Leeming, G., & Galatsis, K. (2008). An assessment of integrated digital cellular automata architectures. *Computer, 41*(1), 38–44.