Problems for a Philosophy of Software Engineering

Stefan Gruner
Department of Computer Science
University of Pretoria
South Africa
sg@cs.up.ac.za

Introduction and Motivation

The call for philosophical meta theory of software science and engineering is partly due to the success and partly due the problems, shortcomings and failures experienced by this still rather young discipline so far. Had software engineering had no success at all up to now, then it might (probably) have vanished already as rapidly as it had emerged, and thus it could have become at best a footnote in future's books on the history of science, but not a topic of this essay. For example, the global social (r)evolution induced by the internet and mobile telephony, which has profoundly changed some social habits even in deepest Africa, would not have been possible without the achievements of software engineering as 'enabling technology'. On the other hand, had software engineering been only successful and nothing but successful until today, then we would most probably still indulge complacently in the sunshine of our success and would not feel any necessity for critical reflection and self-reflection. The crash of the Ariane5 space flight #501 on the 4th of June 1996 is probably the most widely known example of the dire consequences of software defects. As it is common knowledge today, the *causa proxima* of that costly accident was a software defect, or, to be more precise: a numeric type error after the device's control software had been wrongly ported from the hardware system of Ariane4 to the slightly different hardware system of Ariane5. From such kind of suffering, philosophy emerges.

Philosophy of computer science, which is the general theme of this special issue in this journal, deals predominantly with problems and questions around the nature of computation as a process in time, the physicality or non-physicality of information [Lan1961], or with the question whether or not computer science belongs to the group of mathematical or natural sciences [Den2007]. Thus the question arises why an essay on the philosophy of software engineering should find its place in a special issue on the philosophy of computer science? This could be justified with the answer that software engineering is a sub-field of the field of computer science, as some people would argue, or that software engineering is based on computer science (as an auxiliary science), as other people would argue. The question, whether software engineering is a sub-field of computer science or if computer science is only an auxiliary science to an independent field of software engineering, is a science-philosophical question with relevance also to the philosophy of computer science. Anyway –whatever answer to that question the reader might prefer– we can state with a high degree of credibility that software engineering has not yet been sufficiently taken into account in our attempts towards a philosophical understanding of computer science. Though there already exist several science-philosophical reflections by various authors on various topics in the computer science context of software engineering –see, for example, the contribution by Smith on the ontology of *objects* [Smi1998] (which are also relevant concepts in software engineering), or the works by Fetzer on the philosophy of program verification [Fet1988,1998] and the concept of models in computer science [Fet1999]– the field of software science and software engineering as a whole is surely not yet exhaustively explored and philosophically reflected. For these reasons, this essay aims at making another step into this direction.

This step shall be made by reviewing and discussing some recent issues in the philosophy of software engineering, and –consequently– by pointing to some open problems which deserve our attention in future work. The essay as a whole is motivated by a recent verdict by Rombach and Seelisch: *"up to now, a concise, practicable theory of software engineering does not exist"* [RSe2008]. However, most of the critical and meta-theoretic remarks about software engineering have come from philosophically minded software scientists and engineers within this discipline so far, thus we are still hoping for a more interdisciplinary discourse together with professional philosophers of science (and also: historians of science, sociologists of science, etc.) in this regard.

In a recent article by Amnon Eden in this journal [Ede2007] one can find an interesting discussion of three different 'paradigms' of computer science, whereby software engineering –the topic of this essay– was associated by Eden with the 'technocratic' one of those three paradigms.[1] For each of those three paradigms of computer science, Eden had identified different philosophical foundations and presuppositions in three philosophical areas, namely *ontology* (i.e.: what is there), *epistemology* (i.e.: what can be known), and *methodology* (i.e.: how can knowledge be reached). My argument, in this essay, mimics Eden's argument formally (or structurally) in the sense that we shall in the following identify and discuss three different 'paradigms' of software engineering itself; (and thereby implicitly utter a modest critique of Eden's somewhat too one-sided association of software engineering, as a whole, with the 'technocratic paradigm' of computer science). [Ede2007] is thus the most important point of reference in the discourse of this essay.

Last but not least –after its central argument– this essay also suggests in its outlook section some possibly promising themes for future research in the philosophy of software engineering – themes which could not have been treated in this essay due to lack of space and for the sake of topical cohesion.

Throughout the remainder of this essay it is assumed that the readers already have some basic understanding of software engineering as an academic and industrial discipline; non-expert readers are referred to the explanations provided in an earlier essay about this topic [NKB+2008].

[1] One of the anonymous reviewers of the pre-print manuscript to this article suggested that Eden would have "conjectured" the technocratic paradigm mainly for methodological reasons, so-to-say as an ideal methodological entity without much of a basis in reality. On the contrary, Eden replied recently in a private communication that he still believes that "the technocratic paradigm is not only live and kicking, but it also has all but taken over computer science, at least at the level of funding and other forms of decision making, which singularity affects the direction that this field is taking" (18 Oct. 2010, via e-Mail).

‘Paradigms’ and Quarrels about the Foundations of Sciences

As Eden has pointed out in [Ede2007], there are often deeper science-philosophical issues behind the facades of obvious ‘paradigmatic’ differences. This shall be shown also for the case of software engineering in the following parts of this essay – though the now rather modish term ‘paradigm’ should not be used inflationary, or otherwise it would lose its particularly important Kuhnian attributes of historic dominance and systematic incommensurability – Masterman’s critique of the lack of univocity of the term ‘paradigm’ throughout Kuhn’s writings notwithstanding [Mas1970].

Anyway, the science-philosophical issue behind a ‘paradigm’ is in many cases something which the constructivists have called the ‘problem of origin’[2], i.e., the problem about how to establish the conceptual foundations of a particular science consistently and without *petitio principii*. Even if one is not an ardent supporter of methodological constructivism,[3] one has to credit the protagonists of that philosophical school for their sincere enquiries in the context of the ‘problem of origin’. Whilst the methodological constructivists approached this problem rather pragmatically, not un-similar to Heidegger’s thoughts about ‘world’[4] and ‘equipment’[5], the problem of origin also has a terminological aspect which reveals itself in the *words* (terms, notions, concepts) which are axiomatically *used* in the terminological system of a scientific theory but are not essentially explained by its according particular science. Metaphorically speaking, those un-explained fundamental terms and notions are the ‘doors’ through which we can proceed from the domain of a particular science into the domain of philosophy. For a number of long-established sciences, the following examples of ‘door concepts’ are well known to every student of philosophy:

- In *biology*, ‘life’ is comprehensively described but not essentially explained in its deep nature.[6]
- In *physics*, ‘energy’ and ‘force’ remain two aptly described meta-physical mysteries.
- In *stochastics* and *mathematical statistics*, we quantify but do not essentially clarify an imported idea of ‘probability’ or ‘likelihood’.
- In formal *logics*, we use but do not deeply explain the notion of ‘truth’.
- In *jurisprudence*, the notion of ‘law’ is based on the idea of justice, but an ultimate definition of ‘justice’ cannot be given within the legal framework itself (and history has indeed seen many examples of unjust laws).
- In *informatics* (a.k.a. computer science) as well as in *information theory* (i.e. the science about message transmissions via channels following the works by Claude

[2] Anfangsproblem

[3] Hugo Dingler, Peter Janich, etc. Their *methodological* constructivism must not be confused with *epistemological* constructivism (truth as social construct), and also not with Brouwer’s *mathematical* constructivism, a.k.a. intuitionism (in which, amongst others, the classical tertium-non-datur axiom, namely “$\neg\neg A \equiv A$”, is not accepted).

[4] Welt

[5] Zeug

[6] Unless, of course, we would be willing to amputate the semantics of ‘explanation’ deliberately and ad-hoc to such a crippled extent that it becomes, by decree, completely equivalent to the semantics of “comprehensive description” – the *Wiener Kreis* (Circle of Vienna) is often associated (rightly or wrongly) with such kind of epistemological surgery.

Shannon) we use the idea of 'information' but we cannot essentially explain its deeper meaning within the theoretical framework of these sciences.

- In classical *mathematics*, a pre-understanding of the notion of 'proof' beyond the symbolic operations is taken for granted, and, last but not least in this list of examples,
- in *psychology*, the concepts of 'soul' and 'mind' lead straight into one of the oldest problems of philosophical thought.

Historians of science will notice that a 'foundation dispute'[7], i.e.: a methodological and meta scientific dispute as exemplified above, has a tendency to emerge particularly at the 'thresholds' of those 'doors' between science and philosophy. In all those foundational disputes there was ultimately the problem of metaphysics at stake: Shall metaphysics be acknowledged and admitted at all, yes or no? – if yes, then how much of it? – etc. The quarreling parties in most of those examples listed above were usually some sorts of 'positivists' / 'empiricists' / 'formalists' / 'behaviourists' / 'reductionists' / 'materialists' (etc.) on the one side, versus some sorts of 'platonists' / 'holists' / 'rationalists' / 'idealists' (etc.) on the other side. Take, for example, the notorious debate about *vitalism* in meta-biology and philosophy of nature (e.g.: Hans Driesch), or the flurry of modern truth-theories behind the scenes of sociolinguistics, textual hermeneutics, and mathematical logics, or the foundation dispute of meta mathematics about the notion of 'proof', which lead to the development of proof-theory, Brouwer's and Heyting's intuitionism, etc. For the field of informatics or computer science, empiricism versus rationalism was discussed in [Ede2007].

Whereas Eden has delivered a foundations analysis for the domain of computer science, and thereby identified (or at least strongly associated) software engineering as (or with) one particular 'paradigms' in (or of) computer science [Ede2007], this essay goes further to discuss different philosophical 'streams' *within* software engineering itself (which Eden had still treated more or less like one monolithic entity without any inner frictions and factions). Some kind of foundation dispute in software engineering we can observe very clearly in these days, namely the one between followers of the human-centered 'agile' versus the followers process-centered 'engineering' methodologies [NKB+2008], or, in other terms: 'humanists' versus the 'formalists'. This quarrel between 'humanists' and 'formalists' in software engineering is also connected with the questions whether or not software engineering is a sub-science of computer science, and –even more problematic– whether or not software engineering is an instance of 'science' at all. This problem, which was also a theme in [Ede2007], will be further discussed in the remaining sections of this essay.

However, before this discussion about the philosophical problems in software engineering can continue, a few clarifying remarks need to be made about '*what is?*' software engineering, on the basis of some clarification about '*what is?*' software itself.

The Ontological Status of Software

Much truth has already been said and written about the ontological status of computer programs or software as a non-material entity, see for example [Ols1997], [Cle2001], [BRo2002], [Ede2007], [NKB+2008], and many others. To date, and in those literature

[7] Grundlagenstreit

examples, there is still some ambiguity about the notions of ‘software’ and ‘computer programs’ which would deserve some further clarification (i.e.: are ‘software’ and ‘computer programs’ extensionally equivalent concepts, or are these concepts in some unidirectional inclusion relation with each other, etc.?) but this question is not the main question of this essay.

For the understanding of the remainder of this essay it is sufficient to understand the immaterial nature of software in terms of this often-told anecdote from the early days of computing: There was computer scientist traveling by ship to some conference overseas, and the quarter-master of the vessel complained that the ‘software’ in the travelling scientist’s luggage, full of computer-readable punch cards, was ‘too heavy’. Answered the scientist to the ship’s quarter-master: “My software weighs nothing! Do you see the holes in these punch cards? *These holes are the software*!”

For the sake of argument in this essay (and notwithstanding the above-mentioned literature references on the ontological status of software) let us first of all understand ‘software’ as *text* – however *not* the paper (or whatever material substrate) on which the text is written. Like a poem, software has thus also *aesthetic* qualities (which are often forgotten in the literature on software ontology), such as: *form* (and even beauty in its form), *legibility*, etc. What distinguishes a module of software from a poem is its interpretability and executability by some kind of computing machinery, or, in layman’s language: unlike a poem, software can ‘tell’ a computer ‘what to do’. This features of software reminds us, for example, of a baking recipe, which is also text, however text in such a way that it can tell a baker in the bakery how to bake a cake. Thus, in this very broad sense of the term, even a baker’s recipe for cake-baking could be regarded as ‘software’, whereby the specific differences between this kind of ‘software’ and actually executable computer programs are mainly found in the degree of detail and precision as far as the algorithmic description (or prescription) of the executable computational steps are concerned [Cle2001] – see for comparison [Ede2007] with his terminological distinctions of ‘program script’, ‘program process’, etc.

By the way: My categorisation of software code as ‘text’ should not be mistaken as just a fashionable ‘postmodernist’ hermeneutical gimmick; it has its justification in the very construction principle of the von-Neumann/Zuse hardware architecture itself, in which both ‘instructions’ and ‘data’ are stored as bit-patterns –i.e.: ‘text’– indistinguishable from each other in the same storage section (RAM); whether a bit-pattern in some RAM cell *c* is *interpreted* as ‘instruction’ or as ‘data’ depends mainly on the operations of the von-Neumann/Zuse machine at runtime – see the ‘technocratic ontology’ of [Ede2007] for comparison.

Anyway, on the basis of the simple notion of ‘software’ as sketched above (which is sufficient for the understanding of the remainder of this essay) it is fair to say that ‘software engineering’ is both the theory (science) of ‘grasping’ software, as well as the practice (industry) of ‘making’ it in the best possible way – whereby the qualifier ‘best possible’ refers both to the *quality* of the production method (*process*) and to the quality of the software as deliverable outcome (*product*) of that production process. For clarification in terms of the bakery analogy of above, note that the software engineer is *not* the baker, and software is *not* the cake! Rather: the software engineer is like somebody who creates cake-recipes *for* the baker, such that the baker (i.e.: the computer) can bake cake (i.e.: computation output, calculation results, etc.) on the basis of a given recipe. This simple analogy is all we need to keep in mind about software and software engineering for understanding the subsequent sections of this essay. (Whether or not such a textual software recipe corresponds to a pre-existing ‘form’ or ‘idea’ in a Platonic realm of super-reality –in other words: whether software recipes, in their role as technical problem solutions, are ‘discovered’ or ‘invented’– is *not* a question for this essay.)

Software Engineering between Rationalism and Empiricism

Ten years after the software engineer Gregor Snelting had sharply attacked especially the academic (not so much the industrial) branch of software engineering for a wide-spread attitude of ‘Feyerabendianism’ [Sne1998] (as far as the flood of their out-of-the-blue-sky concepts and rather unsound publications was concerned) –see again [Ede2007] for comparison– the computer scientists and software engineers Rombach and Seelisch have continued this dispute with their statement 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* – i.e. that the knowledge gap between software engineering research and practice is widening [RSe2008]. They further 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”. Those problem statements lead consequently to the 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” [RSe2008]. In that short paragraph one can already detect two relevant science-philosophical problems (which Rombach and Seelisch did not explicitly address), namely:

- how to bridge the category gap from ontology (“discovering *facts*”) to deontology (“how projects *should* be run best”) without committing the notorious naturalist fallacy?, and
- what types of investigations may be methodologically admitted as ‘empirical studies’ if software engineering at large does not (and cannot) happen in the closed environment of a well-controlled chemical laboratory? – see for comparison [Tic2007] with a classification of various empirical methods in software engineering.

Rombach and Seelisch further identified two key 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” [RSe2008] which lead 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 [BRo2002] – see below. About this relation between software engineering and computer science 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" [RSe2008]. Thus, in contrast to the viewpoint of [Ede2007] wherein software engineering appeared like one particular 'paradigm' *of* computer science (i.e.: a particular way of 'doing' computer science), software engineering appeared in [RSe2008] as an autonomous engineering discipline *on the basis* of computer science.

It is not the purpose of this essay to discuss comprehensively the extent of structural and methodological similarity between computer science and physics which Rombach and Seelisch have asserted in their essay – such analogy was also mentioned by [Ede2007] (in its section on the 'scientific paradigm'). At this moment it seems to me that such an analogy between computer science and physics, as asserted in [RSe2008], is –at least in our time– more wishful thinking than observable reality, but anyway that is a topic of discussion for the 'classical' philosophy of computer science about which some volumes of publications already exist.[8] This essay is mainly concerned with software engineering and its relation to other disciplines, not the meta-scientific disputes about those other disciplines themselves as such, though it should also be clear that there must be some topical overlap between the philosophy of computer science and the philosophy of software engineering, in correspondence with the topical relationships between computer science and software engineering themselves – see the further discussions below.

Anyway, Rombach and Seelisch continued their discussion with the topic of 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 should be better called a 'maxim', such as not to confuse ontology and deontology, world and method. However, pedantic terminology aside, there must surely arise the question about the *limits* of 'principles' such as 'divide and conquer' themselves: Classically –and apparently also in [Ede2007]– one had almost always tacitly presumed rather simple hardware structures, such as the *Zuse/von-Neumann* computer architecture (or rather simple networks composed of such devices), as the material basis for 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' (such as our methodical reductionism) might possibly falter. In the words of Victor Zhirnov and his co-authors this reads as follows: "When we consider the use of these systems" (i.e. cellular automata) "to implement computation for general applications, a vexing set of software challenges arise (...) and we are aware of little work

[8] About the philosophy of computer science, at least two special editions of the journal *Minds & Machines* (Springer-Verlag, 2007) and the *Journal of Applied Logic* (Elsevier, 2008) have already appeared in print (see http://pcs.essex.ac.uk/). Moreover there exist textbooks such as [Flor1999] and [Col2000].

in this area" [ZCL+2008]. In other words: At stake is thus, from a science-philosophical perspective, the principle-ness of those software engineering concepts which had been so far regarded as 'principles' under un-reflected, accidental historical circumstances (such as the technical and technological dominance of the Zuse/von-Neumann machine) which had been simply been taken for granted during several decades of our times. In this context, software engineering philosophy must thus ask the question: What is the characteristic feature of a 'principle', and are we really confronted with (genuine) principles when practical software engineers speak about such? Such a philosophical concept analysis –for example on the notion of 'principle'– can then lead to further 'paradigmatic' insights, similar to what has been shown in [Ede2007].

Regarding the demanded *empirical evidence* in software engineering –see for comparison the 'scientific paradigm' in [Ede2007]– Rombach and Seelisch 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" [RSe2008]. At this point I can see [RSe2008] going a step further than [Ede2007] in which one cannot find sufficient mentioning of any kind of 'meta-method' for an effective 'cultural' transition from the (rejected) 'technocratic' to the (desired) 'scientific paradigm' [Ede2007]. Moreover, Here I can also see a 'door' into the domains of philosophical ethics and philosophical anthropology with the question whether or not any change of our 'working culture' is arbitrarily at our disposal, or if there exists anything like a 'human nature' on which our 'working culture' might depend in a non-arbitrary manner; the general presumption of [RSe2008] seems to be that this *can* be done.

Another, namely normative, question is then: *should* such a work-cultural transition (under the assumption of its possibility) be made (at all)? In this context it is interesting to note that the software engineer Tom DeMarco, previously known as a strong supporter of rigorous metrics and quantitative measurements in the software engineering process as advocated by [RSe2008], has recently dissociated himself from his earlier positions and is now strongly emphasising the importance of ethical concepts such as 'value' and 'purpose' beyond the borderlines of quantitative control and controllability [DeM2009]. Contrary to Rombach and Seelisch's remarks regarding physics as the role-model discipline for computer science and software engineering, DeMarco claimed recently 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" [DeM2009].

Here we have arrived at a fundamental science-philosophical and methodological point of issue in software engineering again (between the three major parties which like to I call the 'formalists', the 'engineers' and the 'humanists'), at which DeMarco confessed: "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, (etc.) All these strive for consistency of practice and predictability. Consistency and predictability are still desirable, but they haven't ever been the most important things. 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" [DeM2009].

In terms of classical philosophy, DeMarco's statement sounds very much like American Pragmatism; yet it remains to be demonstrated whether or not such pragmatism will be able pull the discipline of software creation by its own hair out of swamp in which it is notoriously sitting (like in the story of Münchhausen). DeMarco's is clearly an optimistic point of view based on the 'common sense' experience that we are not daily confronted with catastrophic Ariane-5 incidents and that our daily interaction with mundane household software products (e.g.: eMail, internet, telephony, computer games, etc.) is –in spite of the occasional 'hickup'– reasonably pleasant and successful. Thus, DeMarco simply refused to accept the notorious 'software crisis' –by which much software philosophy, including [RSe2008], is motivated– as a crisis at all, which points –in the end– to related questions in social philosophy and the philosophy of systems about the concept and the essence of what we want to call a '*crisis*'. For comparison: we would (at this point in time) also not easily want to assert that the entire discipline of civil engineering (as a whole) would be 'in crisis', though many streets and roads are still rather poorly built, some hardware structures still collapse catastrophically every now and then, and also many civil engineering projects –not only software engineering projects– are running late and over-budget, as many town mayors and district managers can tell. From the perspective of [Ede2007], however, the viewpoint of [DeM2009] is also interesting, because his cannot be captured adequately by any of Eden's three 'paradigms' alone: DeMarco's viewpoint is not in contradiction to any of those three; it is located somewhat 'orthogonal' to (or even beyond) all of them, and single-handedly adds another dimension to our software-philosophical problems under consideration.

Anyway, the issue of 'experimental-ness' of software engineering, vaguely mentioned by DeMarco in his quote of above, shall now lead us back to the discussion of the issues emphasised in [RSe2008] in which it was stipulated that "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"

[RSe2008], which seems very much compatible with the 'scientific paradigm' of computer science in [Ede2007].

The lengthy quote of above contains the core of Rombach's and Seelisch basically empiricist software engineering philosophy. 'Formally' we can immediately recognize their adherence to the ideal of physics as the role-model science, 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, Rombach and Seelisch did not clarify (and did not even attempt to clarify) their notion of 'experiment', especially (not) as far as the crucial classical criterion of *repeatability* is concerned. How are software engineering 'experiments' *controlled* and *isolated* from their environment, which is the classical precondition of their repeatability? Has any 'software engineering experiment' in the history of science ever been de-facto repeated? And if repeatability cannot be granted, what is then the degree of validity of the 'laws' which are supposed to emerge from such an 'experimental' procedure? These are the kind of questions which philosophical software engineers like Rombach and Seelisch must try to answer seriously – otherwise they would immediately run into the same kind of difficulties as *Auguste Comte* with his empiricist conception of sociology as the 'physics of society' more than 150 years ago. I would like to add that repeatable experiments (in the classical sense of the term) in software engineering are possible if a computer itself is the well-controlled 'laboratory' and a computer program is the subject of experimentation –see [Ede2007] for comparison– but then we are back in the comparatively narrow field of computer science and programming, which does not exhaust the wider field of software engineering in which we have to deal with larger projects, various human or corporate stake-holders, legal and financial constraints, etc. – how can all those be subject to 'experiments' in the classical, physics- or chemistry-oriented sense of the term? About Rombach's and Seelisch's methodological request regarding the 'verification' of hypotheses in the domain of software engineering, see Karl Popper's notion of falsifiability and the related discussions in [NKB+2008].

Last but not least an academic question re-arises from Rombach's and Seelisch's statement, namely: whether software engineering should be categorized as 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; this classification problem was also mentioned in [Ede2007]. Rombach and Seelisch seem to oscillate between these two classification alternatives and did not commit themselves to a final decision in this regard. Rightly, however, it was pointed out in [RSe2008] that the *immaturity* of software engineering as an 'engineering' discipline is closely related to the following practical shortcomings and methodological 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"

[RSe2008], which leads the authors to conclude that *"up to now, a concise, practical theory of software engineering does not exist"* [RSe2008]. The remainder of their paper deals with particular examples of popular software engineering myths, and the suggestion of some concrete research questions to stimulate research programmes –thereby obviously intending: '*progressive*' (not: 'degenerating') research programmes in the terminology of [Lak1978]– with the aim of eventually being able to replace those myths: In that part of their paper we can find much support for what Eden had called the 'scientific paradigm', in spite of his suggestion that software engineering would largely be dominated by the 'technocratic paradigm' [Ede2007].

Now, since Rombach's and Seelisch's manifesto [RSe2008] is directly related to an earlier contribution by Broy and Rombach [BRo2002], it makes sense to look at that contribution as well, for the sake of a more comprehensive understanding of our topic. As an initial definition of software engineering we can find there: "Purpose of the industrial engineering of software is the development of large-scale software systems under consideration of the aspects costs, deadlines, and quality" [BRo2002]. Already this initial definition by Broy and Rombach could lead us further into a discussion of general philosophy of technology, namely: what constitutes an 'industry'? Is 'industry' a large number of people and how they organise their work in a Taylorist or Fordian manner, or is it the application of accumulated 'capital' (i.e.: machinery and automated tools) in the production process? Is the use of 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 surely be interesting new questions for a more general philosophy of engineering, technics and technology [Rap1974]. Though these questions cannot be discussed any further within the scope of this essay, they clearly tell us that a comprehensive philosophy of software engineering must reach beyond the scope of our classical philosophy of computer science in which such questions and problems (e.g.: industry, organisation of human work, etc.) do not find their suitable place; see **Figure 1** for the sketch of a topic map, which shows the embedding of the philosophy of software engineering in a wider philosophical context.

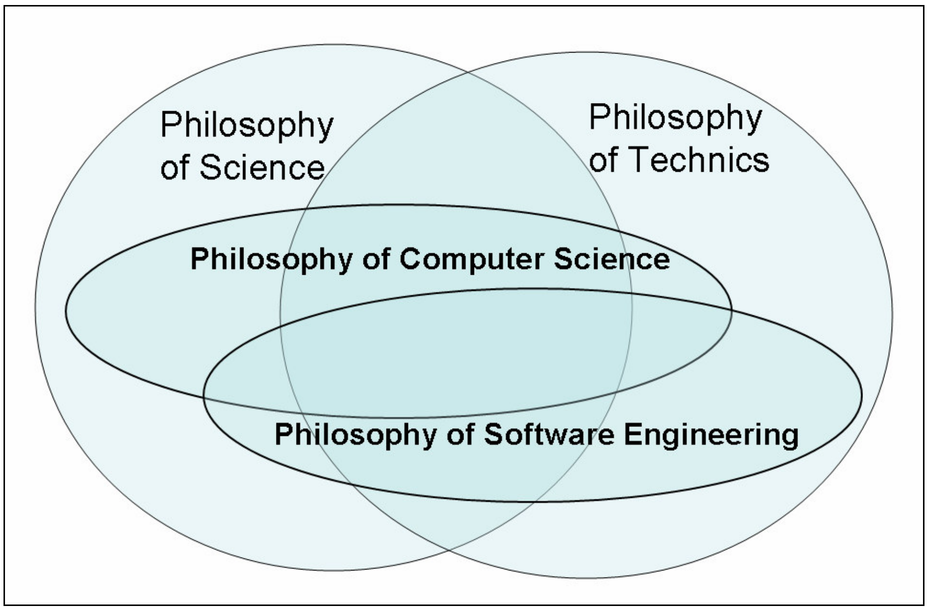

Philosophy of Science
Philosophy of Technics
Philosophy of Computer Science
Philosophy of Software Engineering

***Figure 1:** Philosophies of Computer Science and Software Engineering as special cases of Philosophies of Science and Technics / Technology.*

However the main theme of [BRo2002] is the degree of difference and similarity between software engineering and other engineering disciplines, on the basis of the immateriality of software itself. Particularly they mentioned:

- 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 –see again [Ede2007] for comparison– and, last but not least,
- the absence of natural physical constraints as protectors against weird forms of design and construction.

On these premises [BRo2002] 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", and
- that "foundations and methods are partially still missing".

The most interesting part of [BRo2002] in the context [Ede2007] is their attempt to classify software engineering in a category of related disciplines, with the purpose of contributing to a philosophical self-understanding (the lack of which they had previously identified) of the software engineering discipline. The key question (which also came up in [RSe2008] again) was, whether software engineering is *included* as a *sub*-field of computer science (as it is currently enshrined the academic curricula at many universities, with software engineering courses being lectured as part of the computer science degree), or whether software engineering is a field on its own with computer science as its separate basis. Also [BRo2002] did not reach a decisive conclusion in this regard, though they apparently tended towards the latter solution with the analogy argument: "Imagine that physicists with a specialisation in mechanics would be employed as mechanical engineers!". Here we could ask if that was 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 – but anyway, the basic classification by [BRo2002] (as well as [RSe2008]) looks as depicted in **Figure 2**.

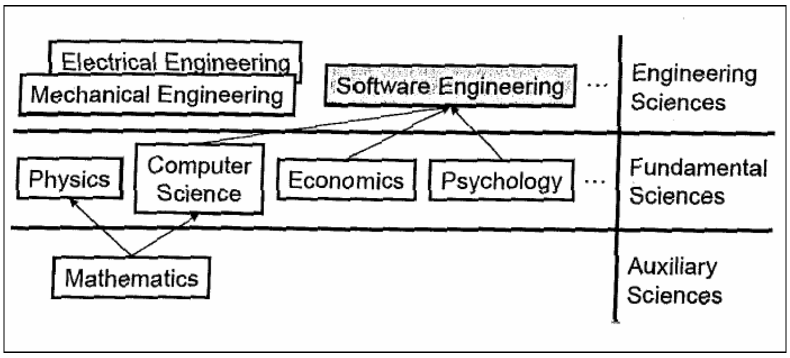


***Figure 2**: Classification of Software Engineering according to [BRo2002]*

In Figure 2 we can see three categories of sciences, namely 'auxiliary' sciences, 'fundamental' sciences, and 'engineering' sciences. Software engineering appears here in the third category, together with electrical and mechanical engineering as (some examples of) sister sciences. Sciences so different from each other as physics, computer science, and psychology appear here all in the category of 'fundamental' sciences (middle layer of Figure 2), whereas mathematics appears in the bottom layer of Figure 2 only as an 'auxiliary' science.

There are some obvious omissions in that diagram which do not need to be discussed any further. For example: mathematics is obviously also a helper-science to economics, and economics must certainly be taken into account not only in commercial software engineering (as shown in Figure 2) but also in commercial mechanical engineering (not depicted in Figure 2) – ditto for psychology which must obviously be taken into account also for the design of useful and intuitive user-interfaces in the domain of electrical and hardware engineering.

More interesting about Figure 2 is the question why mathematics does *not* point *directly* also to the engineering sciences (only *in*directly via the foundation sciences)? Thus, the diagram in this form seems to suggest that [BRo2002] seem to believe that whenever a software engineer is applying mathematics, then he is actually doing computer science, not software engineering. Such an opinion would be consistent with Eden's assertion that the 'rationalist paradigm' (in theoretical computer science) and the 'technocratic paradigm' (in software engineering) would have little to do with each other [Ede2007]. However this is in contrast to other schools of software engineering according to which mathematical methods are indeed *genuine* software engineering methods, and not only computer-science-supporting methods at the basis of software engineering. As far as this mathematical-ness of engineering in general and software engineering in particular is concerned, Tom Maibaum has recently pointed out two further relevant issues:

- "Engineers *calculate*, mathematicians *prove*"; this is a somewhat bold expression which basically means that engineers are only applying "distilled handbook-mathematics" the rules of which had been developed outside the realm of engineering [Mai2008].
- 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 [Maib2008].

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 appears to be consistent and feasible before the according artifact is produced. But that was not Maibaum's point in the discussion. The issue 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 yet nowhere to be seen. This I regard as the deeper meaning of the speaking about "immaturity" and "lack of foundations" in [BRo2002], and this still notorious lack of 'handbook mathematics' in the sense of [Mai2008] for software engineering might also have been a reason for Eden's classification of software engineering as mainly ‘technocratic’ [Ede2007]. But on the other hand it is also true that more and more mathematical ‘tools’ are getting applied directly in the domain of software engineering: see for example the application of graph theory for the purpose of software testing, whereby graph theory is helping to *design* the test experiments which are then carried out in a practical experimental way [AOf2008]. If software testing, within the realm of software engineering, is conducted in such a way, i.e.: when the experimental practice is theory-guided, then we are indeed on the way towards what Eden has called the ‘scientific paradigm’, not only in computer science but also in the wider field of software engineering.

Going back to Figure 2, computer science –there regarded as foundation science to software engineering– is surely an issue in and by itself. As it was rightly remarked in [BRo2002], computer science itself is not a monolithic science. Instead, computer science has various parts and aspects, such that it “structures itself (further) into (computer science) as *foundation science and* (computer science) as *engineering science*” [BRo2002]. Let me give two simple examples: A formalised theory of Chomsky *grammars* and a large volume of empirical, practical experience about the design and development of *operating systems* are both included in the domain of computer science at large, whereby the formal grammars are ‘mathematics’ whereas the operating systems are ‘engineering’ (in this somewhat simplified picture). In [Ede2007] this ‘inner diversity’ of (or within) computer science was also not fully taken into account.

Academically, this diversity within the field of computer science is reflected by the placement of computer science departments into different faculties at different universities –typically (with some exceptions) either in faculties of *mathematics and natural sciences* (example: University of Aachen), or in faculties of *engineering and technology* (example: University of Pretoria); sometimes even as a faculty (of computational sciences) in its own right (example: University of Bremen)– which was rightly mentioned also in [Ede2007]. But in reality the situation is even more complicated. Take for example the University of Aachen again, where some computer science chairs even belong to different faculties, though they are all ‘computer science’: there, for example, the chair of operating systems belongs (for historic reasons) to the faculty of electro-engineering, whereas the chair of compiler construction belongs to the faculty of natural sciences and informatics, though both operating systems and compiler construction clearly fall into the category of ‘computer science’. As far as the classification of [Ede2007] is concerned, we can thus say that different ‘paradigms’ might be predominant in different *sub*-areas of computer science (and software engineering), but it would –in my opinion– be wrong to say that there are different ‘paradigms’ *of* computer science *as such* (as a whole), especially if we also take into account that Eden’s whole paradigm scheme is very much based on the ontology of ‘what is a computer program’ [Ede2007], whereas there also exist branches and sub-areas of computer science (for example: database systems design) in which computer programming and computer programs, as such, are not in the centre of attention.

Anyway, the main problem with the classification by [BRo2002], as depicted in Figure 2, is, as far as I can see, that it treats computer science too simplistically and too ‘monolithically’ as mathematics-based foundation science (of software engineering), thereby ignoring other engineering-related sub-sciences of computer science, such as the above-mentioned operating systems. Consequently, a string of subsequent problems arises:

- If software engineering has been 'lifted out' of the domain of computer science into the domain of engineering (Figure 2), should then not –by analogy– also the field of operating systems be lifted out of computer science into the domain of engineering? – and so on, until 'computer science', stripped bare of all its practical aspects would be nothing more than formalised Chomsky grammars and some discrete algebra? – then we would indeed arrive at a very narrow understanding of 'computer science', which would correspond quite accurately with the 'rationalist paradigm' of [Ede2007].
- On the other hand, if we would leave our operating systems where they are, namely in computer science, would then not also the operating systems, according to Figure 2, belong to the foundations of software engineering? Whilst this is certainly not wrong, it is only half of the picture: In fact, operating systems *are* nothing else than large software systems, which means that software engineering should now also be listed, vice versa, as a 'foundation science' for operating systems [NKB+2008] within the domain of computer science. In Figure 2, however, the link between computer science and software engineering is only unidirectional, not bidirectional.
- In this context, last but not least, it is also interesting to note that the mutual dependency between computer science and software engineering, or at least parts thereof (though not depicted in Figure 2), corresponds quite well to the well-known *Constructivist* argument about the mutual dependency between physics and the engineering of technical artefacts which need to be used for physical measurements; the Constructivist viewpoint is thus in contrast to the classical interpretation of physics as the foundation of engineering as it was expressed by [BRo2002] (Figure 2).

As an intermediate summary of the discussion as it has been conducted so far it seems fair to say that *neither* is software engineering only a sub-area (nor 'paradigm') of computer science (because it essentially entails activities such as project management about which genuine computer science is not concerned), *nor* is software engineering simply 'based on' computer science in an unidirectional relation (because several subjects of computer science, such as operating systems or compilers, *are also* large-scale software systems and would not exist without the existence of successful software engineering methods; the same is even true for computer hardware design which is becoming more and more dependent on software-based modelling tools).

Going back to Figure 2, let us now dig somewhat deeper and ask the question: *What is it, that 'lifts' software engineering up onto the level of engineering*, above computer science? The answer of [BRo2002] is: "*experience*", such that, for example: "A new method is perhaps a remarkable result of computer science, but without robust empirical experiences about its effectivity and limits of applicability it is not a contribution to software engineering" [BRo2002]. This science-philosophical position is schematically depicted in **Figure 3**, which is also taken from [RSe2008] with reference to [BRo2002].

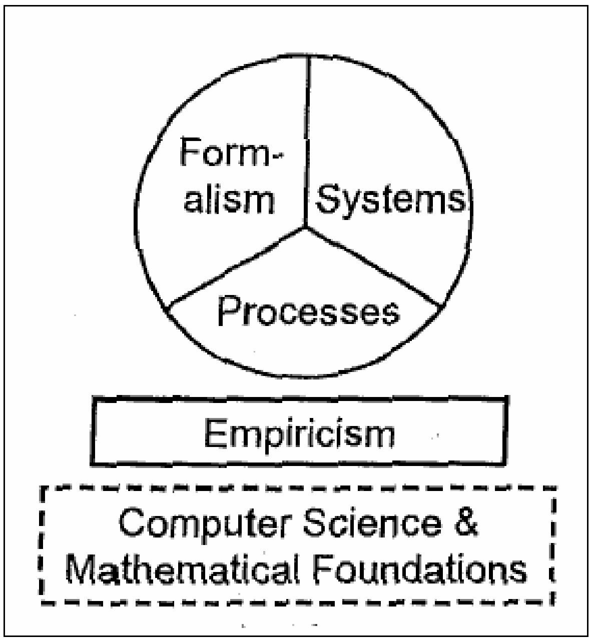
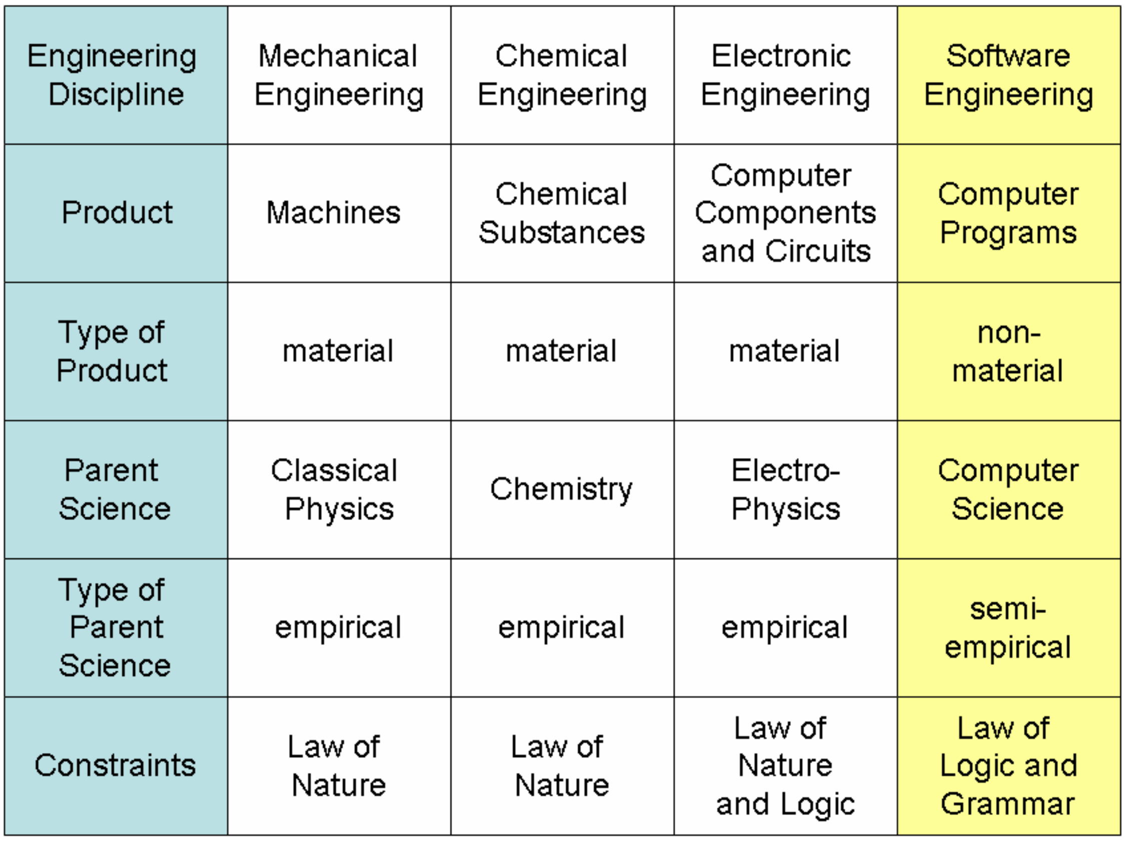


Figure 3*: Elements of Software Engineering according to [BRo2002] and [RSe2008]*

The problem with the above-mentioned quote, as far as I can see, is the implicit equation "Software Engineering = Computer Science + Empiricism" (see Figure 3) which tacitly reduces computer science to purely rationalist, non-empirical science, similar to the 'rationalist paradigm' of [Ede2007], in contrast to what we have already discussed above. On the other hand, computer science was likened by the same authors to physics according to Figure 2 of [BRo2002] – would they then, by analogy, also assert the equation "Engineering = Physics + Empiricism" and thereby reduce physics to pure speculative scholastics (as it has been historically the case throughout the Latin middle ages)? On the basis of everything what I have discussed above, I cannot conclude that empiricism would enter software engineering by addition to a purely rationalist computer science: both computer science and software engineering have both rationalist and empiricist elements in it, and neither can computer science be fully reduced to (or subsumed by) software engineering, nor can software engineering be fully reduced to (or subsumed by) computer science – unless we would aim at a redefinition of our historically grown terminology in a merely stipulative way, purely ad-libitum and ad-hoc.

Nevertheless: on the ontological status of software (which has been more comprehensively discussed elsewhere) it was rightly pointed out by [BRo2002] that software is an enhancer of the *intellectual* abilities of its users, (an observation which did not play an important role for the discussions in [Ede2007]), whereas material hardware is an enhancer of the *bodily* abilities of its users. The latter thought leads straight back to the classical machine theory, formulated already in 1877 by the philosopher of technics and engineering, *Kapp*, in his '*Grundlinien einer Philosophie der Technik*'. Moreover, related to *Heidegger*'s notion of '*Zeug*' (equipment), a software-plus-computer system has elsewhere been dubbed as 'Denkzeug' (think-equipment), in contrast to the 'Werkzeug' (work-equipment, tool) of the material world, and the same thought, though not through the same words, was thus expressed in the philosophy of [BRo2002]. Here we are actually at yet another interface between philosophy of computer science, philosophy of software engineering and a general philosophy of technics and technology; see Figure 1 again for a graphical sketch of the topical situation.

Summary and Conclusion
What have we reached? Starting from [Ede2007] and his interesting discussion of three ‘paradigms’ of computer science, this essay has asked the question if (and, if “yes”, to what extent) the arguments of [Ede2007], in the domain of philosophy of computer science (including software engineering at the margin), could be mapped directly into a more general philosophy of software engineering (which is related to –though not identical with– the more specific philosophy of computer science)? It has been found that not all of the arguments from [Ede2007] are directly applicable in this case, and several reasons therefore have been shown. More than in [Ede2007] the question about the ‘engineering-ness’ of software engineering has been an important theme of this essay which should also be regarded as a continuation of [NKB+2008].



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

***Figure 4**: Classification and Characterisation of several Engineering Disciplines.*

In **Figure 4** I have summarily listed four examples of engineering disciplines, including some related ‘parent’ sciences plus some important features and characteristics. (As discussed above, the relation between ‘parent’ science and ‘child’ science in the table must be taken ‘with a grain of salt’ and is not necessarily unidirectional.) The table shows clearly that software engineering is less constrained by the laws of nature than other engineering disciplines; this allows for more freedom of human ingenuity and leads consequently also a greater potential of errors and mistakes. Note, however, the grammar constraint: programming languages must always have a well-defined formal syntax.

Moreover we can summarize that 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 production 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.

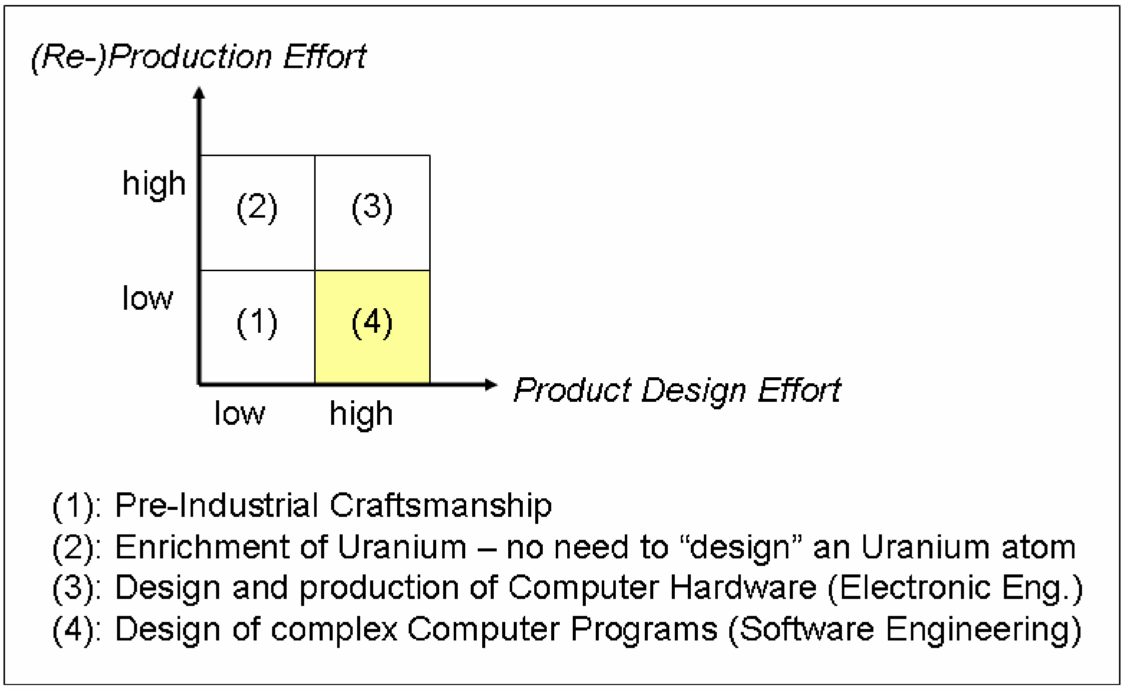

(Re-)Production Effort
high
low
(2)
(3)
(1)
(4)
low
high
Product Design Effort
(1): Pre-Industrial Craftsmanship
(2): Enrichment of Uranium – no need to "design" an Uranium atom
(3): Design and production of Computer Hardware (Electronic Eng.)
(4): Design of complex Computer Programs (Software Engineering)

***Figure 5**: Comparison of Design and Production Efforts in different Engineering Areas*

In software engineering, on the contrary, all the 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. **Figure 5** shows various engineering disciplines, including software engineering, in a two-dimensional matrix with respect to their efforts in design and (re-) production. This is also one of the points where a philosophy of software engineering must interface with a general philosophy of technics and technology, as it was sketched above in Figure 1.

Outlook: Open Questions and Future Work

Since philosophy of software engineering is relatively new in comparison to philosophy of computer science, there is a long list of interesting questions for future work and further discussions. Though the main contribution of this essay –as discussed in the previous sections– was an analysis of some 'paradigms' of software engineering in comparison with related 'paradigms' of computer science from [Ede2007], this essay would be too incomplete if it would not point at some other issues for further investigations in a new philosophy of software engineering. This shall be done very briefly in these remaining few sections at the end of this essay.

'*System*', '*model*' and '*process*' are three fundamental concepts not only in computer science [Fet1999] but also in software engineering. Those three terms can be found in almost every software engineering publication, research paper or textbook for students. Typically there will be a software development 'process' in a software engineering project, during which a software 'system' is being produced in accordance with a corresponding 'model'. However, all those three concepts already have a longer history in the terminology of various sciences, and software engineering has simply 'inherited' these terms and continued their usage without much reflection about their historical semantics. Here I can also see opportunity for interesting philosophical work in the future, such as to find out which aspects of the historical semantics of 'system', 'process' and 'model' have been preserved in the terminology of software engineering, and which aspects of their semantics have been modified (or even lost)? As mentioned above, such an investigation cannot be done within the scope of this essay any more, but at least the following hints shall be given:

- A 'model' in the classical, physical sciences (from which mathematics and formal logics are here excluded, because they use yet another notion of 'model') is usual constructed *by abstraction from* something that already exists; for example the globe on the desk of a foreign minister in politics is a model of our planet Earth. In software engineering, on the contrary, we typically construct a 'model' of a software system *before* that software system itself comes into existence. In other words: we can see here a *switch* of direction between domain and range of the model relation.
- According to [Roe1983], our modern notion of 'process' –which seems to be an observable modification of 'something' during the passage of time– was strongly informed by the science of chemistry. Here we could ask if Roettger's explanations can be losslessly transferred into the domain of the activity of software development, or if there is anything specific about a software development 'process' which is not sufficiently covered by the historic semantics of that technical term. From there we could go even further into the philosophy of processes (since the Pre-Socratics: Hegel, Nietzsche, Whitehead, etc.) and ask whether or not a 'thing'-based metaphysics (e.g.: Strawson) would be sufficient at all to capture the essence of software engineering?
- As a software 'system', when not being processed by a running computer, we usually regard something rather static, namely a large set of program files and the program-call relations within and between those files. As far as I can see, such a static notion of software 'system' is well compatible with the classical notion of 'system' provided by *Johann Heinrich Lambert* [Lam1782/1787], but future investigations would be needed to substantiate such a claim.

The often-mentioned material 'no-thing-ness' of software does not only have implications for our ability of understanding it [BRo2002]; it also has implications for the semantics of the term 'software maintenance',[9] which is a standard word in every software engineer's technical vocabulary. Something that is not material cannot physically decay or wear out; in what sense is it then possible to speak of 'software maintenance'? Maintenance of a car means typically, for example, to replace lost engine oil, or to replace a worn-down tyre by a new one, such that a previous state of newness is re-established. In the domain of building architecture, such a fix would be called 'restoration', for example of an old villa from the previous century, at which some broken roof tiles could be replaced by tiles of *identical type*. In the usual software engineering terminology, on the contrary, 'maintenance' typically means either:

- the replacement of a program file F, which is (and had always been since its creation) *wrong* in relation to some requirements specification S, by a new (different) program file F' which now (hopefully) fulfills the requirements stipulated by S, or:
- the replacement of a program file F, though not wrong with regard to S, by a new program file G which adds additional functionality and features to a software system which had previously not been there, because they had not even been mentioned by its initial requirements specification S. In other words, the replacement F/G corresponds to an a-posteriori requirements specification modification S/S'.

In the analogy of our old villa from the previous century, the latter modification would be called a 'renovation' (rather than a 'restoration'), whereby the villa could get, for example, an additional door or window at a place were there was previously only a wall. I have mentioned this little peculiarity of 'software maintenance' –more recently even more problematic: software 'evolution'!– as only one example of a software engineering terminology which is full of 'home-grown' and often un-reflected *metaphors*. Language philosophers might perhaps find it interesting to delve somewhat deeper into this techno-linguistic domain.

In addition to an already existing discussion about the comparability of the ontological status of software and the ontological status of *art* [NKB+2008] I shall remark only briefly at this point that the usual discussion about whether software engineering is 'engineering' or 'science' is still confronted with yet another opinion which claims that software engineering is *neither* 'science', *nor* 'engineering', but simply '*art*' [Edm2007]. Indeed, such claims seem to be at least partly consistent with the analyses by arts theorists such as Goodman or Burnham, in the context of which Edmonds has rightly pointed out that the *quality* of software systems is *also* measured in *aesthetic* categories [Edm2007], in quite a similar way in which many theoretical physicists (example: Einstein) have insisted on an aesthetical-theoretical position according to which a mathematical formulation of a physical law cannot be true if it does not also have 'beauty' – here we can perhaps find a modern remainder of the ancient Greek notion of ***κάλόσ*** (kalos) in which the concepts of 'beautiful' and 'good' were not to be separated. In analogy, also a skillful software engineer would intuitively reject an 'ugly' software design plan in almost the same way in which Einstein would have

[9] Thanks Morkel Theunissen for this hint – see acknowledgments at the end of this essay.

intuitively rejected an ‘ugly’ formulation of a physical theory, without being able to reason analytically and with full logical rigor about such an issue of ‘ugliness’. In other words: a future philosophy of software engineering could also include some philosophy of aesthetics.

Finally I want to mention yet another epistemological issue, namely in the context of software and knowledge. Many software products seem to be inappropriate or do not fulfill their intended purpose simply because in many cases we just do not know ‘*how* to do things’; we lack procedural knowledge in many domains and circumstances. Vice versa one could even assume a very radical epistemological position and say: We do not have any knowledge about something unless it is procedural (algorithmic) knowledge about how to create it.[10] This is related to the problem of what is ‘creativity’. For example, it is fair to say that we have very good procedural knowledge about how to create a compiler – in short: we ‘know’ compilers very well. On the contrary we do not have procedural knowledge about how to create stunning original pieces of art – therefore, in radical terms, we do not know art, not in this strong sense of ‘knowing’ with which we know compilers (because we can produce them easily following standardized handbook procedures). Also in software engineering in general we still have very little (procedural) knowledge about how to create a software system which adequately fulfills some arbitrarily given purpose P. This epistemological problem is related especially to the ongoing efforts in the sub-field of ‘automated software engineering’ (ASE), wherein we could say: the more software creation processes we can automate (algorithmically) the better we ‘know’ software engineering, and vice versa. Not everybody would want to assume such a radical epistemological position –which says: *only* procedural production knowledge is counted as ‘knowledge’ at all– but this to discuss is yet another task or problem for an upcoming philosophy of software engineering.

Acknowledgements
For inspiring discussions I thanks the students of my software engineering seminar at the University of Pretoria, as well as *Tom Maibaum*, *Derrick Kourie*, and *Morkel Theunissen*.

References

Amman, P. & Offut, J. Introduction to Software Testing. Cambridge University Press, 2008.

Broy, M. & Rombach, D. Software Engineering: Wurzeln, Stand und Perspektiven. Informatik Spektrum, Vol. 16, pp. 438-451, Springer-Verlag, 2002.

Cleland, C. Recipes, Algorithms, and Programs. Minds and Machines, Vol. 11, No. 2, pp. 219-237, Kluwer Acad. Publ. / Springer-Verlag, 2001.

Colburn, T. Philosophy and Computer Science. Series ‘Explorations in Philosophy’, M.E. Sharpe Publ., 2000.

DeMarco, T. Software Engineering: An Idea whose time has come and gone? IEEE Software, Vol. 26, No. 4, pp. 95-96, IEEE Computer Society Press, 2009.

Denning, P. Computing is a Natural Science. Communications of the ACM, Vol. 50, No. 7, pp. 13-18, ACM Press, 2007.

Eden, A. Three Paradigms of Computer Science. Minds and Machines, Vol. 17, No. 2, pp. 135-167, Springer-Verlag, 2007.

Edmonds, E. The Art of Programming or Programs as Art. Proceedings ‘New Trends in Software Methodologies, Tools and Techniques’, pp. 119-125, IOS Press, 2007.

[10] I have heard about such a position in a lecture presented by Manfred Nagl in the late 1990s.

Fetzer, J. Program Verification: The very Idea. Communications of the ACM, Vol. 31, No. 9, pp. 1048-1063, ACM Press, 1988.

Fetzer, J. Philosophy and Computer Science: Reflections on the Program Verification Debate, pp. 253-273 in Bynum, T. & Moor, J.H. (eds.), The Digital Phoenix: How Computers are changing Philosophy, Basil Blackwell Publ., 1998.

Fetzer, J. The Role of Models in Computer Science. The Monist, Vol. 82, pp. 20-36, Hegeler Institute Publ., 1999.

Floridi, L. Philosophy and Computing: An Introduction. Routledge Publ., 1999.

Lakatos, I. The Methodology of Scientific Research Programmes – Philosophical Papers, Vol. 1, Cambridge University Press, 1978.

Lambert, J. Drei Abhandlungen zum Systembegriff (1782/1787). Re-published in Diemer, A. (ed.), System und Klassifikation, 1968.

Landauer, R. Irreversibility and Heat Generation in the Computing Process. IBM Journal of Research and Development, Vol. 5, No. 3. Reprinted in IBM Journal of Research and Development, Vol. 44, No. 1/2, pp. 261-269, 2000, IBM Press, 1961.

Maibaum, T. 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, 2008.

Masterman, M. The Nature of a Paradigm, pp. 59-89 in Lakatos, I. & Musgrave, A. (eds.), Criticism and the Growth of Knowledge (Proceedings of the 1965 International Colloquium in the Philosophy of Science ay Bedford College), Vol. 4, Cambridge University Press, 1970.

Northover, M. & Kourie, D. & Boake, A. & Gruner, S. & Northover, A. Towards a Philosophy of Software Development: 40 Years after the Birth of Software Engineering. Zeitschrift für allgemeine Wissenschaftstheorie, Vol. 39, No.1, pp. 85-113, Springer-Verlag, 2008.

Rapp, F. (ed.), Contributions to a Philosophy of Technology: Studies in the Structure of Thinking in the Technological Sciences. Reidel Publ., 1974.

Roettgers, K. Der Ursprung der Prozeßidee aus dem Geiste der Chemie. Archiv für Begriffs-geschichte, Vol. 27, pp. 93-157, Meiner-Verlag, 1983.

Rombach, D. & Seelisch, F. Formalisms in Software Engineering: Myths versus Empirical Facts. Lecture Notes in Computer Science, Vol. 5082, pp. 13-25, Springer-Verlag, 2008.

Smith, B. On the Origin of Objects. MIT Press, 1998.

Snelting, G. Paul Feyerabend und die Softwaretechnologie. Informatik Spektrum, Vol. 21, No. 5, pp. 273-276, Springer-Verlag. English translation: Paul Feyerabend and Software Technology. Software Tools for Technology Transfer, Vol. 2, No. 1, pp. 1-5, Springer-Verlag, 1998.

Tichy, W. Empirical Methods in Software Engineering Research. Invited Lecture; Proceedings 4th IFIP WG 2.4 Summer School on Software Technology and Engineering, Gordon's Bay, South Africa, 2007.

Zhirnov, V. & Cavin, R. & Leeming, G. & Galatsis, K. An Assessment of Integrated Digital Cellular Automata Architectures. Computer Vol. 41, No. 1, pp. 38-44, IEEE Computer Society Press, 2008.