Faith, Hope, and Love

An essay on software science's neglect of human factors

Stefan Hanenberg

University Duisburg-Essen,
Institute for Computer Science and Business Information Systems
stefan.hanenberg@icb.uni-due.de

Abstract

Research in the area of programming languages has different facets – from formal reasoning about new programming language constructs (such as type soundness proofs for new type systems) over inventions of new abstractions, up to performance measurements of virtual machines. A closer look into the underlying research methods reveals a distressing characteristic of programming language research: developers, which are the main audience for new language constructs, are hardly considered in the research process. As a consequence, it is simply not possible to state whether a new construct that requires some kind of interaction with the developer has any positive impact on the construction of software. This paper argues for appropriate research methods in programming language research that rely on studies of developers – and argues that the introduction of corresponding empirical methods not only requires a new understanding of research but also a different view on how to teach software science to students.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Human Factors, Languages

Keywords Research methods, programming language research, software engineering, empirical research

1. Introduction

The term software crises [8] is often applied in the area of software engineering and programming language research in order to argue that these crises still exist, and in order to argue that new techniques are required in order to overcome the crises. And indeed, programming language and software engineering research seem to be inexhaustible fountains that produce over and over again new techniques that reduce the software crises: new development processes, modeling notations, programming language constructs, frameworks, etc. are invented and such techniques are claimed to overcome existing problems.

However, a closer look reveals that research in the area of programming languages and software engineering has a fundamental problem with how they reason on their newly invented artifacts. It turns out that many artifacts have an inadequate foundation of how they are justified because they do not consider that developers are part of the software construction process. This means that many new techniques are claimed as solutions for existing problems without any sufficient investigation.

Hence, it is adequate to ask whether software engineering and programming language research can be considered as serious scientific disciplines that construct new artifacts and examine them in an objective way. Spoken in a more practical way, it is valid to ask whether these artifacts contribute to a solution to today's problems – or whether they are the main cause for today's problems and consequently the main cause for the crises that they claim to reduce.

As a consequence, developers have to decide on their own whether a new artifact should be considered good or reasonable: "*for practitioners, it's hard to know what to read, what to believe, and how to put the pieces together*" [23, p. 67]. Hence, the assessment of new artifacts is the product of subjective experiences and sensations, which is an unacceptable situation - faith, hope, and love are a developer's dominant virtues to estimate the benefit of new artifacts.

This essay argues for the urgent need to consider human factors when reasoning about programming language and software engineering artifacts and emphasizes the need for appropriate empirical methods in order to provide valid and adequate rationales for such artifacts.

This paper analyzes current research approaches in software sciences and discusses their validity and adequacy. It shows that there is already a practice to use human factors to argue for certain artifacts - but a practice which is rather based on speculations instead of scientific methods.

Onward '10 October 17-21, 2010, Reno-Tahoe, Nevada, USA.

After comparing the consideration of human factors in other research disciplines, it is critically discussed why humans factors hardly play any role in programming language and software engineering research. Finally, the paper argues that a number of fundamental changes are necessary in research as well as in teaching in order to overcome the current inadequacies.

Notes: *This paper uses a number of prominent scientific works to illustrate the inappropriateness of current rationales used in software research. The aim is definitively not to discredit any author. Because of that, research papers are chosen in a way that they represent fundamental statements about known and popular topics in software research and rather no up-to-date papers. It is also necessary to note that this paper does not claim that the statements of these papers are wrong – it only argues about the missing evidence or the inappropriateness of the chosen research methods.*

This essay will use the term software science as a common term for software engineering research as well as programming language research in order to ease the reading[1].

2. Research Methods in Software Science

This section gives an overview of research methods and rationales which are currently applied in software science. The overview should not be considered as a complete description of all practiced and possible research approaches. The main intention is to show that there is already a variety of different approaches which differ with respect to the subject of research, the kinds of statements being promoted by them, and the techniques used to back up the corresponding statements. Then, the validity and adequateness of such approaches is discussed with a special focus on how the results can be used by a developer in order to determine whether the application of a certain artifact improves the development of software - with the result that human factors, which are essential to determine whether an artifact improves software development, are hardly (or inadequately) considered.

2.1 Classification of research approaches

The origin of software science is mathematics. Classical disciplines such as algorithms and data structures are based on the approach to examine a program according to some formal characteristics such as run-time behavior. Typical approaches in these disciplines are correctness proofs or run time estimations using the O-notation. The programs, which are understood as formal descriptions of a number of actions that take place in a certain ordering, are the focus of these approaches. Here, a program itself is the subject which is being studied. Mathematics is the underlying discipline which provides the research method. The aim is to construct theorems and to prove them. This approach considers programs as deterministic methods that transform input data into output data. Programs are the subjects of research. In the following, this approach will be describes as the *classical approach*.

Over time, new approaches were developed that differ from the classical approach. First, the assumption of determinism was softened: disciplines such as parallel computing do no longer assume that the ordering of statements during the execution of a program is known. Further disciplines concentrate on randomized algorithms where the result is permitted to depend on random distributions. Nevertheless, they still have in common with the classical approach that the subject being studied is the program itself. However, the research methods applied here differ from the classical approach. First, there is the *stochastic-mathematical approach* where stochastic statements are achieved by mathematical and analytical reasoning. Second, there is the *stochastic-experimental approach* where stochastic statements are achieved using statistical methods applied on measurements resulting from corresponding experiments. The main difference to the classical approach is that statements are no longer of kind *true* and *false*. Instead, the statements are stochastic statements based on probabilities.

The stochastic-mathematical approach as well as the stochastic-experimental approach depend on random distributions of certain variables contained in the programs to be analyzed. One characteristic of these randomized variables is that they are under the control of the researcher: researchers can control the input parameters, the underlying distributions, etc.[2].

There are further approaches that differ from the previous ones. For examples, approaches for improving the performance of software often have a characteristic that does not match the previous descriptions. There, different strategies or algorithms are examined in order to improve the performance of applications. The characteristic of the approach is, that software plays two different roles: first, there is a piece of software that is examined (such as a new run-time system), second, there are further pieces of software that are used as input parameters (at least, the approach permits to use software also as input parameters).

A noteworthy difference to the previous approaches is the second role of software: software plays the role as input parameters (instead of raw data such as integers, etc.). Hence, this approach considers software as existing (real-

[1] The term software science was already used in [16] for a different purpose - to describe a system of metrics. The author of this essay uses this term for a different purpose because he thinks that it describes best the here addressed topic and also meets best the common understanding of the topic. The author considers the risk of misinterpreting the term to be rather low.

[2] It is important to note that the stochastic-experimental approach does not describe all kinds of approaches that perform experiments. The approach describes those approaches where the subject of research and all other influencing variables can be formally described (and controlled). Hence, works that perform experiments on concrete machines (such as the measurement of time of a concrete algorithm on concrete CPUs) typically do not fall into this category, because they (typically) cannot control all influencing variables.

world) phenomena which are used to study the original subject (which is in this case the new run-time system). The software being used as input parameter is (typically) intended to give a representative sample from the reality. In order to gain such a sample (and to compare different research results), a typical approach is to use benchmarks, i.e. sets of upfront known pieces of software. We call this approach, where a predefined set of data is being used as input parameters for experiments, the *benchmark-based approach*.

All previous approaches have a purely technical nature – software is being examined either in an analytical way (classical approach, stochastic-mathematical approach) or in an experimental way (stochastic-experimental approach, benchmark-based approach). However, the software developer or the user of a piece of software does not play any role in these approaches. Consequently, we call of these approaches *technical approaches* in the following.

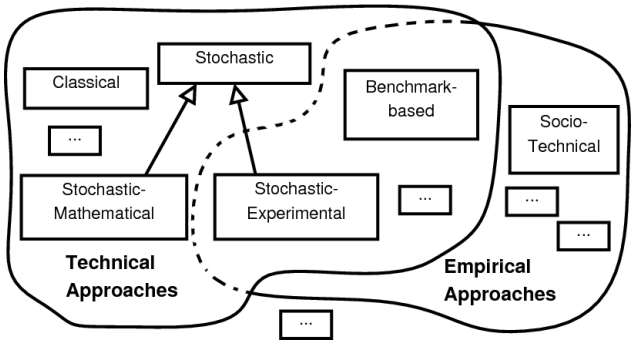

Figure 1. Categorization of research approaches

Apart from the technical approaches, further research directions have been followed that fundamentally differ from the technical ones - works that provide or invent new programming language constructs or new tools, such as the invention of object-oriented programming. The main argumentation in such directions is that a new construct or abstraction permits developers to write *better* software. *Better* typically means in this context that the piece of software to be written using the new abstraction or tool has fewer errors, is better maintainable or more reusable (corresponding qualitative criteria can be found in many text books such as [34]). The fundamental change to the previous technical approaches is that the subject being examined has changed. While in all previously described approaches a concrete piece of software (algorithm, run-time machine, etc.) was analyzed, this new approach studies the way developers construct pieces of software using a new artifact. Consequently, the developer becomes part of the argumentation for or against new techniques – a developer, a human being, is in addition to a new artifact in the focus of research. In the following we call this approach the *socio-technical approach* (see Figure 1).

Before considering the socio-technical approach we will consider the technical approaches from two perspectives. First, we consider to what extent the approaches are able to provide valid rationales. Next, we discuss to what extent the technical approaches are adequate to provide arguments in software science. Here, the main perspective is to ask to what extend the technical approaches provide adequate arguments for the decision whether the application of a new artifact is beneficial.

2.2 Validity of technical approaches

It is obviously not necessary to discuss the classical approach with respect to its validity: the subject of research can be formally described and the theorems can be proven. The same is true for the stochastic-mathematical approach, although "only" stochastic statements can be proven.

The stochastic-experimental approach already widely differs from the previous ones: the research method is no longer based on formal reasoning. Instead, the results of experiments are used as rationales for or against a certain technique (or piece of software). Although there are obvious parallels to empirical methods from other disciplines apart from computer science (such as medicine, experimental physics, etc.), it must be emphasized that there are also huge differences to them: the subject (the algorithm, etc.) as well as the result that is being examined (run-time benefit, exactness of result) can be formally described. As a consequence, all influencing variables that play a role in experiments can be (typically) formally described and are completely under the control of the researcher: the researcher can define upfront the distribution of input parameters, the random number generators being used, etc.. Consequently, there are no unknown factors that potentially influence the results of the experimentation. As a consequence, a repetition of an experiment using the stochastic-experimental approach leads to the same results.

Nevertheless, it seems clear that this way of reasoning on software leads to valid results, especially in situations where the subject cannot be analyzed using analytical methods. Nevertheless, the approach also has the characteristic that the experimenter decides the chosen distributions for input variables – and it is at least speculative how the results potentially differ if different distributions would have been chosen. For the same reason, it is potentially problematic to compare different pieces of research based on the stochastic-experimental approach, since the input parameters can be individually chosen by researchers.

The benchmark-based approach is quite similar to the stochastic-experimental approach with respect to its experimental character. Both perform experiments and apply statistical methods. However, in contrast to the stochastic-experimental approach, the experimenter cannot influence the data used within the experiment – the benchmark is typically an external factor. Consequently, the influence of an experimenter on the results is much more reduced in comparison to the stochastic-experimental approach, which improves the ability to compare different research works (since

the experiments are based on the same input data)[3]. However, in order to gain this benefit it is necessary that that there is a commonly accepted definition for such a benchmark. This situation is typically only given if the techniques under examination are already applied since a number of years. Furthermore, it requires some consensus in the (scientific or industrial) community about such benchmarks. The benchmark-based approach still has some subjective element: a benchmark is intended to be some representative sample over the set of all possible data which is constructed by human. For example, in [4] a benchmark suite is proposed which is "*a set of general purpose, realistic, freely available Java applications*" [4] which can be used to measure for example the performance of Java Virtual Machines. Although the authors of the benchmark suite argue for the quality of the suite it is at least questionable whether these applications are representative. Nevertheless, this is not the subjectivity of the researcher applying the benchmark - this is the subjectivity which is part of the benchmark itself.

The benefit of comparing different research works based on the benchmark-based approach lies in the application of new techniques to the same benchmark. Consequently, a benchmark is hardly able to evolve, because otherwise this benefit would no longer exist. But if the benchmark does not evolve, it cannot consider the continuous change in software development: new programming techniques, development environments, architectures, etc. frequently appear and have a direct impact on the resulting software (with respect to size, complexity, etc.) – but these changes are not part of the benchmark. Because of the above described potential problems, *"benchmark composition is always hotly debated*" [36, p. 36].

Although these problems are known, it still seems obvious to consider research statements or theories whose rationales are based on the benchmark-based approach to be valid, because it seems obvious that it is not possible to define benchmarks that evolve over time and that still permit to compare different pieces of work based on common data. The problem with the subjectivity of the benchmark remains, but since we have to accept that it is not possible to gather all current and possible future pieces of software in one single benchmark, we have to live with the remaining subjectivity.

2.3 Adequacy of technical approaches

It is important to consider the validity of research results based on the underlying research method. It is maybe even more important to consider whether the research methods are adequate to reason on statements about the techniques.

A general view on software science is that it provides (new) tools and techniques to build, maintain and execute software. The terms tools and techniques should be considered to be rather abstract. Examples for such tools are concrete software tools (such as development environments, software libraries, programming languages), as well as methods (such as development processes, or test techniques) up to models (such as formal languages, modeling notations, etc.). It is important to note that the construction of a new artifact itself does not represent a scientific activity. The scientific activity is the evaluation of statements about the technique, where the benefit of a certain technique is shown (or disproved). Other scientific activities are the construction and evaluation of theories that permit to predict certain phenomena that appear while a piece of software is constructed, maintained or run.

In the technical approaches the benefit of an artifact can be argued based on rationales on an artificial artifact. For example, the benefit of a JIT compiler in comparison to an interpreter can be argued by comparing the run-time using benchmarks (benchmark-based approach). The benefit of a certain type system that requires some additional type annotations can be argued by proving its type soundness (classical approach).

Although the technical arguments seem to be quite strong, they still have weaknesses. The argumentation for the JIT compiler is problematic, because it is unclear whether the underlying benchmark represents a representative sample – but we already argued above that this argument is rather weak. However, there are more serious objections against the argumentation for the second example (type system). Although type soundness has been proven, it is not shown whether a developer is able to use the type system. It might be possible that the type system is too complicated that developers are overstrained when applying it. Consequently, the positive statement based on the technical approach would turn out to be rather misleading if used by developers in order to determine whether the type system should be applied: a type soundness proof does not say a word about whether the type system improves the construction of software. In this situation, the classical approach turns out to be inadequate for developers to decide whether the application of the new artifact is beneficial. For the JIT compiler the situation is different. The application of the JIT does not require additional actions by the developer. Consequently, a pure technical statement is adequate here.

The examples show that the technical approach is adequate in some situations – and in some situations it is inadequate. Although technical statements can be potentially proven, they prove a formal characteristic within a formal system. This does not permit one to reason about a possible benefit of the artifact that requires special user interactions because the possible behavior of users is outside the formal system.

If we assume that most techniques provided by software science require additional user interaction, it can be con-

[3] Of course, researchers still have the freedom to decide which benchmark they use - in case there are different alternatives available. But once a benchmark is chosen, the influence is reduced.

cluded that most of the time the application of a technical approach is rather inappropriate.

2.4 Further approaches

It should be noted that even other kinds of approaches are applied – a close look into international journals and conferences reveals that technical approaches do not represent the majority of research approaches[4].

Frequently, a common approach is to identify a problem by means of an example, to provide a new artifact and to show that the problem is solved by applying the artifact.

An example for such an approach can be found in [38][5]. There, it is argued that class-based object-oriented languages tend to be too complex, since they provide constructs such as classes, etc.. Then, a new language (the programming language *Self*) is introduced. Then, the benefit of Self is being argued by the absence of certain language constructs such as classes, etc..

From the scientific point of view the argumentation is problematic. First, it is unclear whether the addressed problem is really a problem. However, this (weak) argument is directed to the relevance of the work - which can be argued against any kind of research work. However, a really problematic question is, what exactly the research question is in the paper. If it was whether a programming language can be provided without the language construct class, then the answer already could have been given upfront (with a reference to procedural or functional programming languages). If the intention was to provide a language that is easier to use than a class-based language, then the paper failed to provide any rationales showing that the resulting language is easier. Hence, from the scientific point of view, it must be concluded that the paper does not give a scientific argument for the new language.

A different kind of approach that also can be frequently found is the transfer of artifacts from one discipline to another one. An example can be found in [18]. Here, the authors address the topic of how to document software frameworks. They provide a hint, that pattern languages have been used in architecture (not software architecture). Then, they transfer this idea to framework documentation. Then, the authors mention that a group of developers was satisfied with the pattern-based documentation after some iterations. Finally, they conclude that pattern languages are a good way to document software frameworks.

Again, the arguments are problematic. The transfer of the artifact (pattern language) from one discipline (architecture) to another one (software construction) is achieved more or less arbitrarily. Moreover, the paper just states that a group of developers was satisfied after some iterations. It does not state whether the developers were unsatisfied with the existing solution or whether they were more satisfied with the new solution. Finally, no scientific argument (based on a valid research approach) is given in order to conclude that patterns language are suited for documenting frameworks[6].

A characteristic of the examples is, that their argumentation does not follow any scientific approach. Consequently, the argumentation is not valid and the argued benefit of the proposed artifacts is purely speculative.

For readers of such works it is quite complicated how to handle such situations. Either they ignore the works because of the missing scientific approach, or they decide for themselves whether or not they consider the proposed artifacts to be beneficial. In the latter case the benefit of the artifacts lies only in the eye of the beholder.

Expressed in a more provocative way this means that it is up to the developer's faith to decide whether or not he believes in the proposed artifact; in case he decides to use the artifact in an industrial setting, he has to hope that the artifact will not have a bad impact on the software construction process. Finally, the developer has to decide on his own whether he loves the new artifact – since no objective rationales are given, the choice of a new artifact is a purely subjective and rather emotional process. Faith, hope, and love turn out to be the dominant factors for selecting and applying technical artifacts provided by software science.

2.5 Speculative considerations of human factors in software science

While the previously discussed approaches are often used in the scientific literature, the socio-technical approach is (still) controversially discussed. There is no answer to the question whether or not human factors should play any role in the scientific argumentation in software construction that is commonly accepted among all researchers. For example, Tichy reports in [36] about "*the fear that computer science will fall into the trap of soft science*" - where human subjects are typically considered to be the characteristic of soft science. Hence, it is even unclear whether a socio-technical approach really exists or whether this is rather one branch of popular and unscientific work.

However, a closer look into a number of commonly accepted research works reveals that human factors already play an essential role in software science – although they are typically not considered within a valid research approach. In order to exemplify this, two examples should be mentioned here.

[4] See e.g. [37, 42] for overviews of research methods and [3, 31] for overviews of experiments found in past conferences and journals.

[5] Once again, the author would like to emphasize that the intention is not to discredit any authors or any techniques (in fact, the author is an enthusiastic Self-programmer and an admirer of the works by David Ungar and Randy Smith). This essay also does not make any statement about the possible benefit of Self. The intention is only to argue that the approach in the Self-paper does not permit to draw the conclusions drawn in the paper.

[6] Again, the author (who likes software patterns and design patterns) would like to mention that he refers here only to the paper and discusses whether the argumentation within the paper follows a valid scientific approach. In fact, the area of software documentation using design patterns is already studied in other works (see for example [26]).

- Dijkstra wrote in [9, p. 210] that "*it is a characteristic of intelligent thinking to study in depth an aspect of one's subject matter in isolation*", a principle that he calls "*separation of concerns*". A large bunch of literature refers to the phrase separation of concerns in order to argue for the benefit of a certain technique (typically, the aspect-oriented literature, see [13], refers to this phrase in order to argue for aspect-oriented software composition).
- Another example can be found in [21, p. 8]: there, in order to argue for the benefit of object-oriented programming, it is stated that object-orientation "*is close to the natural perception of the real world: viewed as consisting of objects with properties and actions*".

For both (very prominent) arguments the human factors are essential. In both cases the reference to the human factors is the key argument – the first argument is the foundation for the aspect-oriented literature[7], the second argument intends to highlight the need for the new approach object-oriented programming. Both arguments refer to human characteristics – "human thinking" and "human perception". But from the scientific point of view the above examples must be considered to be problematic:

- Both works apply a characteristic from one discipline (psychology) more or less arbitrarily to a different one (software science).
- In both cases, the statements "come out of nowhere". Neither of the statements has any references to any neuronal science or psychology journal or something similar. Consequently, in the best case both sentences can only be considered as "a possible hypothesis that needs to be tested". In the worst case, one could say that both statements are just the result of the author's fantasy. In any case, it is not valid to conclude anything from them[8].
- Even under the assumption that the statements are right, it is unclear how the causal relationship between each statement and the argued technique (aspect-orientation, object-orientation) is. Even assuming the validity of a statement "people perceive the world as objects" it needs to be checked whether this has anything to do with software construction – and whether this has anything to do with our understanding of object-oriented programming.

So far, we can conclude that the human factors are already considered even in programming language research. However, the way how they are used does not follow any valid scientific approach. Consequently, it is necessary to understand how the human factors can be integrated into a valid socio-technical approach for software science. Therefore, it is reasonable to have a look into sciences that do not rely on formal methods, i.e. sciences that do not purely rely on the technical approaches.

[7] It should be noted that the strong emphasis on the phrase "separation of concerns" was probably not intended by Dijkstra: the phrase hardly plays any role in [9].

[8] Both statements are different from those ones that are traditionally analyzed by computer scientists. Consequently, it is highly probable that computer scientists are not able to determine whether these statements have any scientific background - or whether those statements represent some common sense in these disciplines.

3. Beyond Formal Systems: Consideration of Human Factors in other Disciplines

Disciplines that do not rely on formal systems typically designate the critical rationalism as the foundation of their research method. The core of the critical rationalism according to Karl Popper [24] is that all scientific statements must by falsifiable, i.e. it must be possible to test the statement in order to determine whether it is false. Furthermore, a scientific statement must be universally quantified: an existentially quantified statement is not considered to be scientific.

The tasks of a scientist are twofold. First, his task is to construct sets of (hopefully consistent) scientific statements which are called theories. Second, he needs to try to falsify them using empirical methods. It is important to note that the critical rationalism does not assume that the correctness of a theory can be proven. Instead, it can only be shown whether a theory is false. The more often falsification trials of a theory fail, the more stable is a theory considered. The most extreme interpretation of the critical rationalism demands only one falsification of a theory in order to reject it. In order to explain the critical rationalism, Popper uses a number of examples, mainly from physics.

However, if human behavior is observed using empirical methods the idea of falsification slightly differs. For physical objects a number of "objective" metrics such as "mass" exist and the physical object is not able to change this metric "because of the absences of a free will". Humans on the other hand can reflect on themselves and change in that way their behavior. Hence, humans are even able to "behave in a way they usually would do not". As a consequence, a singular observation in disciplines that depend on human factors cannot be used to falsify a theory (or a single statement, see for example [6], p. 11). Instead, it is necessary to make observations on multiple subjects. This implies that a corresponding statistical analysis is necessary on those subjects. And a consequence of this is that the result of observations can only be expressed using probabilistic methods. This is probably the main reason why computer scientists, whose educational background is mainly influenced by mathematics, are rather reluctant to consider this as a hard science: they use rather the devalued term soft science.

Disciplines that use humans as subjects are for example medicine, psychology, pedagogic, or social sciences. However, each discipline typically has a different view on humans with a different focus. All of these disciplines have in common that they require statistical methods in order to interpret their measurements.

In the German computer science literature, medicine (especially drug research) is sometimes named as a discipline whose research method should be applied in software sciences (see for example [32]). The parallel to software science seems obvious – neither a human nor a certain medicine or therapy is in the main focus of research. Instead, it is the influence of a certain artifact (medicine or therapy) on a set of individuals. This seems to be very similar to software science where a certain artifact has some effect on a developer (or a development team) and the resulting software. However, a closer look reveals that this parallel does not match. In drug research, some objective (at least well studied) metrics such as blood pressure are being used to evaluate the effect of a medicine; metric whose causal effect on subjects is known (such as the effect of high blood pressure on heart diseases, or the probability of suffering from an apoplexy)[9]. For such metrics, differences between subjects caused by different educational or cultural background typically do not play any role – because it is assumed that the subjects cannot attentively influence the results of the experiments. An exception to this are experiments where the effects of placebos are also measured or where no objective metrics are available. Here, the experimental design explicitly assumes a potential influence of the subjects on the results and which also influences the analysis of the measurements.

Applying this approach to software science would imply that the effect of an artifact on a developer would hardly be influenced by the developer's background. For obvious reasons, this cannot be considered to be serious. It needs to be emphasized that software artifacts require an intellectual action of the developer: instead of considering the developer as some kind of "physical machine" he is considered as an intellectual individual who performs some creative actions while developing software. Although the developer (who corresponds to the patient in medical research) used a new technique (which corresponds to a medicine), the effect of this technique cannot be predominantly measured on the subject itself (which would correspond to the measurement of blood pressure): the effect can predominantly measured on the artificial product (the software) resulting from the creative actions.

Here, it seems more likely to consider other empirical disciplines that also have a focus on human factors. Psychology seems closely related to software science. In psychology, human beings are in the focus of or at least part of the subject being examined. Different facets such as cognition, perception, learning, memory, thinking, problem solving, knowledge, etc. are being studied – facets which are for software science of obvious importance, too. Furthermore, psychology has a long experience in empirical methods which means that there is already a knowledge-base for experimental design, experiment execution and analysis techniques. Furthermore, there are standards (cf. e.g. [40]) that need to be considered in order to gain valid scientific results and in order to get such works published. From that perspective, psychology seems to be an adequate discipline that could be used as an example for software science.

[9] It should be clear that such objective metrics from medicine have hardly something in common with metrics such as lines of code where causal effects of this metric (other than the length of a program) are rather unknown.

4. The Socio-Technical Approach: Empirical Software Engineering Today

The demand for applying empirical methods with special focus on human factors is far from being new. Especially, in the 70's and 80's there were a number of works that emphasized the need for empirical methods (taken from psychology) in software science (see for example [27, 28, 39]). Probably the most drastic criticism of current practice in software science can be found in [27], which describes "*the study of programming as an unholy mixture of mathematics, literature criticism and folklore*". Also, there are a number of German authors that argue for the need of corresponding empirical methods from (see for example [25, 32, 33, 35, 36])[10].

It should be mentioned that the terms "empirical methods" or "empirical software engineering" in the area of software science typically describes the socio-technical approach, i.e. the explicit consideration of the human factors within an empirical approach[11].

Meanwhile, there are a number of teaching books about empirical methods available for software science (see for example [19, 25, 41]). An interesting characteristic is, that the content of these books is quite heterogeneous: for example sometimes the necessary statistical methods are the main content of the books (see for example [19]) or fundamental considerations for the experimental design and experiment execution are the main content (see for example [25, 41]), sometimes in combination with descriptions of experiments done so far (see for example [25]).

What is slightly irritating about the literature on empirical software science is the fact that the teaching books describe concrete results of performed experiments, but they hardly provide any special knowledge for software science. This means, concrete, domain-specific knowledge required to perform the socio-technical approach in software science is hardly provided.

The following example should explain this in more detail. For example Prechelt gives in [25] the hint, that subjects participating in an experiment, should have homogeneous capabilities (see [25], p. 112). In fact, many empirical studies ex-

[10] It also needs to be emphasized that there is a number of authors that argue against the use of empirical methods and the consideration of human factors in computer science. For example [15] states that when comparing approaches that have an empirical character and those ones that are primarily speculative, none should be considered to me more worthy.

[11] Of course, this use of the term "empirical software engineering" is quite misleading, because it ignores that even technical approaches such as the stochastic-experimental as well as the benchmark-based approach are empirical approaches.

hibit the need to distinguish between professional software developers and beginners and to analyze both kinds of developers separate from each other. However, if someone tries to perform an experiment, one question remains: How to do that?

It turns out that a number of different studies try to address this question in many different ways. Diverse kinds of questionnaires were applied in the past, different kinds of pretests for developers, etc.. This means that "every experimenter has his own view on how to classify subjects". But it is important to note that "the experience on developers" is domain-specific knowledge for software science. While information about different kinds of experimental designs, statistical analysis, etc. can be directly gathered from fundamental teaching books about psychology or social sciences, information about "how to classify subjects" cannot.

As a consequence, teaching books about empirical software science are typically summaries of knowledge gathered from other disciplines about experimental design and analysis – but they hardly provide special, domain-specific knowledge required in the area of software science in order to perform experiments. Literature that provides domain-specific knowledge (such as concrete "laws" for software science which can be found in [11] or concrete lessons learned from a long lists of experiments which can be found for example in [3]) is rather the exception.

If we take a look into concrete experiments that were performed in the past, we see that a typical approach of these experiments is to compare two techniques. For example, it is measured how long a group of subjects requires to solve a given programming problem using technique A or B, or it is measured how many failures were performed by the subjects using technique A or B (an example for such an approach can be found in [1]). A characteristic of these studies is, that finally some insights are collected – because it was possible to measure a difference between technique A and B – but it is hard to get any more insights from these studies that could be used in different experimental settings. For example, it is unclear whether the same or similar results would have been measured under slightly different experimental settings. The reason for this problem is, that most of the studies compare technique A and B in a concrete setting, but this comparison is not meant to back up any underlying theory such as "technique A requires 10% more time to solve a problem" or "for problems that require only 200 lines of code technique A requires less time, if more than 200 lines of code are required to solve the problem technique B requires less time"[12]. In fact, there is no underlying theory (set of scientific statements) but typically only one single statement be tested. Hence, these "theories" have hardly any means to predict any possible situation.

[12] Of course, a theory that states that a technique A requires less time for all problems than a technique B is some kind of prediction. But for obvious reasons, such theories cannot be often found.

Coming back to the original idea of the critical rationalism, this means that the current situation in empirical software science (using the socio-technical approach) is rather frustrating: the original idea of constructing and testing theories is hardly followed (such theories are not even constructed). Instead, only singular statements are tested.

It should be mentioned that there are prediction models in software science, i.e. models that represent some kind of theory in order to predict future situations, like for example the one proposed in [5]) for estimating the effort for software projects[13]. From the perspective of programming language research, these models are far away from the topic of programming languages, because the influencing factor programming language is hardly considered in these models. Hence, from the programming language research perspective these prediction models rather do not represent domain knowledge that could be used to design and perform experiments.

However, beyond the area of programming languages, modeling notations, etc. there are disciplines that could be considered as part of software science and which intensively make use of the socio-technical approach and which already use theories with corresponding empirical evidence: the area of Human-Computer-Interaction (HCI, see for example [29]). The overlap with the area of software science is (typically) in the area of design of graphical user interfaces. An interesting characteristic of HCI is, that psychology does not only provide the dominant research method (the socio-technical approach), but that psychology also provides a number of theories that are applied there. Examples for such theories are models for the cognitive capabilities (see [17]) or the reaction time of users (see [14]). Both kinds of models represent domain-specific knowledge and theories; fundamental knowledge that researchers in these areas need to know. Such corresponding theories and common knowledge (with empirical evidence) does not exist in the area of programming language.

A first conclusion is that the socio-technical approach is hardly applied in software science, but that there are already studies that consider the socio-technical approach. We have already discussed potential problems of the approach (which was the missing domain knowledge). But there are more reasons why the socio-technical approach is not frequently applied which need to be discussed.

5. Why is the Socio-Technical Approach Hardly Applied?

If the socio-technical approach is a valid approach to validate statements about software artifacts, why isn't it simply applied – and why should it be necessary to write this essay?

First, we need to accept that empirical works are hardly applied (see for example [36, 42]) – and that the socio-

[13] However, some authors rather doubt about the empirical evidence of current prediction models (see for example [12], p. 445).

technical approach is only a subset of all empirical studies[14]. Hence, only a very small part of all empirical works (which are already just a small part of current scientific publications) follows the socio-technical approach.

One could argue that this small number is just a matter of time – considering that software science is still a very young discipline, it is no wonder that no adequate research method has been established and applied so far. Although it is obviously true that software science is relatively young, this argument is still hard to follow, since the area of software science was established in the 20th century – and it should be assumed that researchers in the 20th century are already familiar with different kinds or research methods and are able to distinguish between valid and adequate research methods, unscientific reasoning and pure speculations.

However, there is also a number of different arguments why the socio-technical approach plays a minor role in software science.

5.1 Problems in education

Foundations for empirical studies, which are also the foundation for the socio-technical approach, are typically not taught in the area of software science. Furthermore, empirical education requires knowledge from a number of different disciplines. First, knowledge from the area of stochastics is required which provides knowledge about different distributions and their characteristics. Furthermore, knowledge in the area of statistics is necessary which provides knowledge about descriptive and inductive statistics, the differences between both, and knowledge about significance tests which are required in order to interpret measurements. Furthermore, knowledge from the area of experimental design is required – in order to understand how an experiment can be designed, what the pros and cons of a certain experimental design are, what implications a certain design has on the resulting analysis techniques, etc.[15].

However, looking into the current curricula at universities that teach different facets of software science reveals that hardly any stochastics, statistics, and experimental design is being taught. As a consequence, students are simply not aware of "what this empirical thing" is. Even if students should (by luck) see some empirical works during their studies, their knowledge is simply not sufficient to test, whether these works potentially suffer from errors in the experimental design or analysis. Consequently, students are not able to verify, whether the results of these empirical studies are to be trusted and whether the conclusions drawn from the studies by the corresponding authors are valid and adequate. Hence, the original motivation for empirical studies becomes absurd. The motivation is to gather objective, empirical knowledge by backing up or falsifying theories. If no one is able to understand these empirical works, no one is able to come to an objective conclusion about the subject of the study. Consequently, students are doomed to perceive empirical works as strange collections of huge, arbitrary data sets, without any obvious relationship to the topic software science.

The problem is not only, that the research method is not explicitly taught. The method is typically also not implicitly taught, too. In mathematics for example, the underlying research method (formal reasoning) is also typically not explicitly taught. Nevertheless, the research method is practices in all courses. A course in mathematics does not only teach theorems, it also contains the proofs for such theorems. Students always come in touch with the research method. They are implicitly educated in the research method without explicit courses about it. In psychology, the research method is typically explicitly taught. Furthermore, the research method is part of most courses. There, students do not only learn theories. They learn the experiments that back up these theories. They learn how the experiments were built up, what data has been measured, how the data has been analyzed, and what conclusions were drawn from it.

In software science, education practice is completely different. While in theoretical computer science the research method is also typically implicitly taught, in the area of software science research methods hardly play any role. New (or old) techniques such as programming languages, software development processes, modeling notations, etc. are taught without any scientific argumentation for or against them[16]. In the best case, students learn "examples" where a certain technique "does not seem to be adequate" or "possible scenarios where a certain technique possibly dominates another one". However, how such examples or possible scenarios could be evaluated, i.e. how the objective reasoning about these examples works is typically not part of the education. Due to the missing educational background, students are not able to distinguish between singular observations, speculative reasoning on measurements and valid applications of empirical research.

This has tragic consequences. Students typically do not learn how to investigate a certain technique – they do no learn how to doubt about the benefit of a certain technique. One of the main academic properties – the objective reasoning on certain topics – is hidden to the students. This also typically means that hardly any bachelor or master thesis is done using the socio-technical approach, because the nec-

[14] In [7], Denning reports about a panel in 2004 that discussed the accomplishments of computer science [22] that hardly said a word about experimentation.

[15] An interesting (or maybe even alarming) observation is, that such a knowledge is not only required in order to understand and apply the socio-technical approach: it is required in order to understand and apply empirical approaches (see figure 1) in general.

[16] A small test for the reader: Most of us believe in object-oriented programming. What experiment are you aware of that measured a positive impact of object-oriented programming over procedural programming? In case you rather believe in e.g. function programming: what experiment are you aware of that measured a positive impact of functional programming over procedural programming? Try to answer this question without using additional literature.

essary background is unknown to students and the time for such theses is not sufficient to train students in theses topics. In case a student is still willing to apply the socio-technical approach, the problems of designing and performing an experiment (which will be discussed in sections 5.2 and 5.3) become relevant - problems that make it even more improbable that students get a chance to apply the socio-technical approach.

If any kind of socio-technical approach was hidden to the students, what should be their motivation to search for them (and to apply them) when they are scientists?

5.2 Problems in designing experiments

As already mentioned in section 4, a fundamental problem of the socio-technical approach is, that domain-specific knowledge from the area of software science is missing. As a consequence, an experimenter is forced to make a large number of assumptions on his own. Following the argumentation from section 4, an experimenter decides on his own how to distinguish "good developers from bad ones". Consequently, the results highly depend on the experimenter's personal decision about the distinction of good and bad developers - which is a subjective decision. One could argue here in a malicious way: from the current perspective, the socio-technical approach provides, because of the missing domain knowledge, means to misuse the original idea of gathering objective knowledge. Instead of "objective knowledge" this approach provides "subjective number generators" which are highly influenced by the researcher.

Although this is not satisfying, there are from the current point of view no means to get rid of this problem: if no empirical domain-specific knowledge is available, there is no way to get it somehow by magic. The probability that current socio-technical studies rely on wrong assumptions is very high. The only way to reduce this problem (and to get still meaningful data that can be potentially used and interpreted in the future) is to document experiments as detailed as possible. Additionally, there are already some (preliminary) guidelines available that state how to perform empirical research in the area of software science (see for example [20]).

For the researcher applying the socio-technical approach the missing domain-knowledge has the tragic consequence that he is aware that possibly a number of experiments he performs might turn out to be in vain (maybe already in the near future), because they start from wrong assumptions - a problem that researchers applying for example the classical approach typically do not suffer from.

Although this problem is real, it should be emphasized that this fear is rather the result of the researchers' mathematical background: students and researchers which are mainly by the classical approach tend to believe that a research statement, once it is proven, is eternally valid. However, it is rather naïve to assume that research results (outside formal systems) are stable over decades or centuries. And the reason for today's problems (the absence of missing domain-knowledge among many others) is that the socio-technical approach (and empirical methods in general) has hardly been applied in the last decades - and still plays only a minor role in the software science today.

5.3 Problems in performing experiments

Even if someone ignores the problems described above, another problem is: "What chances do I have to apply the socio-technical approach?". Assuming that the socio-technical approach typically relies on experiments with humans, the question can be refined: "What chances do I have to perform an experiment with humans"?

The main problem here is that universities and other institutions hardly provide a corresponding required infrastructure. A researcher has to find subjects, who typically voluntary participate in an experiment. Paying these subjects is typically not possible because such a payment assumes some kind of funding and a grant for getting such a funding typically requires a bunch of publications in conferences and journals with high reputation, which is typically quite problematic (which will be discussed in section 5.4). However, even if the number of publications would not play a major role for getting a grant, the problem of the socio-technical approach is, that it competes with research proposals that rely on technical approaches – and which do not require any funding for paying subjects. For the organization providing a grant the question is, whether a grant should be given for a proposal where the chances for publications are rather low (see section 5.4) and which costs a lot, or whether the grant should be given to a proposal that does not suffer from these problems.

Even ignoring the payment issue, the researcher has still a problem to find subjects. Since students hardly come in touch with the socio-technical approach during their studies, a socio-technical experiment rather looks strange from their point of view – what should be their motivation to participate in experiments?

In empirical sciences such as psychology this problem of finding subjects was already identified and addressed. It is a typical scenario that the participation in experiments is a necessary prerequisite for psychology students to pass their studies – the execution of experiments becomes in that way part of the daily business at universities[17]. In these domains, it is also quite normal that a bachelor thesis or a master thesis performs an experiment where other students are used as subjects. Such a situation in the area of software science is from the current point of view far from being realistic.

[17] Although it should be mentioned that there are different practical problems that need to be addressed. While in psychology an experiment about for example eye movement control in reading requires maybe just one hour time from each subject, experiments in software science, which require subject to build software, typically require much more time.

5.4 Problems for the academical career

The socio-technical approach plays a minor role in the literature about software science. In conferences and journals with a good reputation it is hard to find works that apply the socio-technical approach. Although there are some means to publish socio-technical studies in conferences and journals that explicitly call for socio-technical studies (such as the International Symposium on Empirical Software Engineering and Measurement or the Journal on Empirical Software Engineering), these conferences and journals play rather a minor role. However, scientists need to publish their works – and they need to publish at conferences and journals with a good reputation. Consequently, doing research using the socio-technical approach drastically reduces a young researcher's career opportunities.

6. Demands to Research and Teaching

In order to overcome the current frustrating situation with software science's carelessness with regard to human factors, there is a need for a number of fundamental changes in software science not only with respect to research but also with respect to teaching. Hence, this section describes a number of demands directed to different audiences: to people that teach students, to students, to researchers, and to people participating in the publication process.

6.1 Demands to teaching

In teaching, research methods need to be taught. This implies an explicit as well as an implicit teaching of those methods. Here, the socio-technical approach needs to be communicated as a different approach among the possible research approaches. Valid and adequate rationales must be made explicit in teaching, i.e. teaching must not only consist of the introduction of new artifacts. Instead, for every artifact corresponding valid and adequate rationales must be given. Consequently, this means that curricula in software science require an orientation toward curricula of those sciences that have already identified a strong impact of human factors. Of course, this orientation should not mean that socio-technical approaches should become the dominant approaches. Instead, they should at least be considered as of equals status or level.

For those artifacts for which such rationales do not exist, at least possible studies must be proposed – and additional effort should be spent on performing such studies as soon as possible. Furthermore, in these situations it must be communicated to students that currently objective knowledge is missing for these artifacts in order to preserve them from the (potentially misleading) faith about artifacts taught at universities.

Finally, universities should provide some kind of infrastructure to students that permits them to perform socio-technical experiments using subjects. This could be done by pushing a duty to students to participate for a number of hours (or days) in experiments.

6.2 Demands to students

While the previous demands were mainly directed to teachers, there are also demands to students. Students needs to be encouraged to doubt about artifacts taught at universities. Whenever new artifacts (or statements) are being taught, students should actively ask for valid and adequate rationales for such artifacts. Most importantly, students should ask for such rationales directly at the beginning of a course in order to determine whether the course provides insights into a topic with sufficient evidence: students should consider evidence as a desirable aim for their studies and should make explicit that missing evidence is a reason for disapproving certain courses and topics. This implies that students should try to establish a new attitude with regard to their studies: instead of trying to come in touch with new technologies over and over again, students should consider stable knowledge about fewer topics to be worth more than speculations about many topics.

Students should become aware that many artifacts that they are being taught are not as well studied as they are probably being told. Instead of considering this situation to be frustrating, it should be considered as a great challenge that needs to be addressed - and that a thesis could be a good mean to address such a challenge. This, of course, requires also a cooperative attitude of students, because if a fellow student requires subjects for experiments, they should try to provide help.

6.3 Demands to researchers

The demands for researchers are manifold. As a first step, it is necessary to guarantee that all research works and publications at least contain a falsifiable statement (which permits at least other researchers to test the proposed statement). For all scientific papers it must be made explicit what the chosen research method is with a short discussion, why the research method is valid and adequate in order to follow the corresponding research question. Hence, researchers should actively work on discussions about chosen research methods and possible alternative research methods. It would probably also help if researchers would discuss why certain research methods are considered to be inadequate for a certain research question.

When referring to related work in scientific publications, researchers should make explicit which of these works provide evident knowledge and which ones do not. This does not mean that literature without sufficient evidence should be ignored but such literature should be used more carefully. At least it is very problematic if literature whose statements are not provided with sufficient evidence becomes part of the argumentation for or against a certain artifact.

Finally, the current trend of researchers to become mainly experts for a certain topic (object-oriented languages design,

etc.) should be abandoned. Instead, the main focus of researchers should be to become an expert in a certain research method - the more mature the applied research methods are, the better are the scientific results.

6.4 Demands to editors, pc chairs, and reviewers

One demand to editors, pc chairs and reviewers of journals and conferences is that a consciousness must be established that research methods are the foundation for every kind of research, that a research method must be valid and adequate to evaluate a certain artifact and that empirical research (especially the socio-technical approach) is one possible way to evaluate statements about artifacts. The persons responsible for the publication process must become aware that technical approaches are rather inadequate to back up arguments that focus on human factors.

The persons responsible for the publication process should also explicitly ask authors to make the research method explicit and to argue about the appropriateness of the chosen research method. Editors and pc chairs should make sure that reviewers are familiar with the research methods of those papers they are responsible for. Reviewers should also make sure on their own that they are familiar with the research method of paper they are asked to review. In case they are not familiar with the method, they should reject to review the paper. This also requires an awareness that there is a difference between a familiarity of the subject of research and a familiarity of a research method: while familiarity with the subject helps to identify problems in a related work section of a paper, this does not necessarily mean that it helps to understand (and evaluate) the value of a paper.

Reviewers of socio-technical papers must be quite disciplined to distinguish between a plausible, probable, subjective, valid, or invalid argumentation for and against a paper. Using the example from the previous sections, it is plausible that reviewers of a paper about a socio-technical programming experiment ask for a classification of subjects within an experiment. However, reviewers must also be aware that currently no objective (or at least commonly accepted) distinction between good and bad programmers exists. Consequently, reviewers must be aware that in such a situation the missing classification is not a scientific objection against a paper - it is a speculative objection where the author (currently) has no objective chance to fulfill this requirement.

Finally, editors and pc chairs should be aware that a lot of knowledge about fundamental questions in software science is missing. Instead of only asking for papers whose focus is mainly in up-to-date topics in software science, they should ask for works that address more fundamental questions in order to reduce the problem of missing domain knowledge that we currently have - and that we will still have in 20 years if this kind of research will not be promoted.

7. Summary and Conclusion

Software science is a domain which frequently provides new artifacts which claim to solve current problems. This essay addresses the question what research methods are valid and adequate to argue for or against statements about such artifacts and emphasizes the need to consider human factors by using appropriate scientific methods.

This essay started with a short introduction of research methods and identifies, that although a number of research works address human factors, typically no valid and adequate research methods are applied – a socio-technical approach, which considers technical artifacts as well as humans which apply them, is more or less missing in software science. Furthermore, the paper shortly discussed possible reasons why the socio-technical approach is hardly applied.

In order to establish the application of such a research approach, this paper argued that software science should take sciences that address the human factors as an example – psychology might be the right choice here. The essay argued that this does not only imply how research should be performed, this also implies how teaching should be done in software science - in order to establish an environment where the socio-technical approach can be practices and taught.

In general, this paper advocates the need for empirical methods in software science considering the human factors. In fact, this idea is far from being new. This paper is just another one in a long history of papers that try to raise this issue (see for example [27, 28, 35, 36, 39]).

It should be noted that there is literature available where authors already argue about the future of empirical software science (see for example [2, 30]). The author of this essay shares the idea that there is a need to discuss the future of empirical software engineering in general but the author considers fundamental changes in teaching and research to be necessary as a first step.

While this paper speaks about the socio-technical approach, it should be clear that there is no single socio-technical approach which considers human factors. Instead there are a number of different research methods which can be considered as socio-technical approaches (see for example [10] for an introduction into different kinds of empirical methods which focus on the human factor). It would be desirable and necessary to have more literature that discusses the validity and adequacy of these approaches to back up different kinds of research statements. In general software science rather suffers from the problem that the topic of research methods rather plays a minor role in literature, research, and teaching. A situation which the author of this essay considers to be tragic.

The author would like to emphasize that this essay does not not try to claim that the socio-technical approach (as well as empirical approaches in general) should be the only research approaches to be taught and practiced. Especially, the author does not try to argue that non-empirical approaches

(such as the classical approach with its focus on mathematical reasoning) should be abandoned by the research community or ignored in teaching. This essay argues that researchers and students need to become aware that there are different research methods for different purposes. And researchers and students should at least become skeptical whenever a software artifact appears which claims to have a positive impact on software development and which provides rationales where the underlying research method does take the existence of software developers into account.

An implicit demand of this essay is to distinguish clearly between scientific and non-scientific works in the scientific literature. This does not imply that unscientific work is considered to be useless - but it is very dangerous if unscientific literature becomes part of an argumentation for or against certain artifacts.

It is important to note that this paper mainly addresses the need to apply the socio-technical approach in order to provide evidence for scientific statements that consider not only a piece of software but also the developer or a user of a piece of software. This essay does not say a word about how scientific statements appear - the question of how scientific statements appear needs to be discussed completely separate from the question how scientific statements should be evaluated.

This essay can be considered as a catalyst to start the discussion about a topic which is rarely addressed in software science although this topic plays a major role in other sciences: research methods. It emphasizes the need to address one obvious fundamental problem in software science: the carelessness with regard to human factors. And it argues that without considering human factors in software science, the construction of software and the application of new software artifacts will still be based on the developer's subjectivity which relies mainly on three principles – faith, hope and love.

Acknowledgments

The author would like to thank David Ungar, Randy Smith, Robert Hirschfeld, and Michael Haupt for discussions about the paper, and Walter Tichy, whose keynote at the German Software Engineering Conference largely influenced this essay (and who made the author aware of the conflict of the term software science). Mark Mahony shepherded this paper and helped to significantly improve the paper. The author would like to thank Holger Wiese from the department of general psychology from the University of Jena for his patience and guidance into the research methods of psychology. Finally, the author would like to thank Erik Ernst for giving very detailed, critical feedback which substantially improved this essay.

References

[1] BARTSCH, M., AND HARRISON, R. An exploratory study of the effect of aspect-oriented programming on maintainability. *Software Quality Control 16*, 1 (2008), 23–44.

[2] BASILI, V. R. The role of experimentation in software engineering: past, current, and future. In *ICSE '96: Proceedings of the 18th international conference on Software engineering* (Washington, DC, USA, 1996), IEEE Computer Society, pp. 442–449.

[3] BASILI, V. R., SELBY, R. W., AND HUTCHENS, D. H. Experimentation in software engineering. *IEEE Trans. Software Eng. 12*, 7 (1986), 733–743.

[4] BLACKBURN, S. M., GARNER, R., HOFFMANN, C., KHANG, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., MOSS, B., PHANSALKAR, A., STEFANOVI, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), ACM, pp. 169–190.

[5] BOEHM, B., BOEHM, B., CLARK, B., HOROWITZ, E., WESTL, C., MADACHY, R., AND SELBY, R. Cost models for future software life cycle processes:. In *Annals of Software Engineering* (1995), pp. 57–94.

[6] BORTZ, J., AND DÖRING, N. *Forschungsmethoden und Evaluation: für Human- und Sozialwissenschaftler*, 4. ed. Springer, Heidelberg, 2006.

[7] DENNING, P. J. Is computer science science? *Commun. ACM 48*, 4 (2005), 27–31.

[8] DIJKSTRA, E. W. The humble programmer. *Commun. ACM 15*, 10 (1972), 859–866.

[9] DIJKSTRA, E. W. *A Discipline of Programming*. Prentice Hall, Inc., October 1976.

[10] EASTERBROOK, S. M., SINGER, J., STOREY, M., AND DAMIAN, D. Selecting empirical methods for software engineering research. In *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. Sjøberg, Eds. Springer, 2007.

[11] ENDRES, A., AND ROMBACH, D. *A Handbook of Software and Systems Engineering*. Pearson Addison-Wesley, 2003.

[12] FENTON, N. E., AND PFLEEGER, S. L. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

[13] FILMAN, R. E., ELRAD, T., CLARKE, S., AND AKŞIT, M., Eds. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[14] FITTS, P. M. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology 47*, 6 (June 1954), 262–269.

[15] GÉNOVA, G. Is computer science truly scientific? *Commun. ACM 53*, 7 (2010), 37–39.

[16] HALSTEAD, M. H. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[17] HICK, W. E. On the rate of gain of information. *The Quarterly Journal of Experimental Psychology 4*, 1 (1952), 11–26.

[18] JOHNSON, R. E. Documenting frameworks using patterns. In *OOPSLA '92: Conference proceedings on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 1992), ACM, pp. 63–76.

[19] JURISTO, N., AND MORENO, A. M. *Basics of Software Engineering Experimentation*. Springer, 2001.

[20] KITCHENHAM, B., AL-KHILIDAR, H., BABAR, M. A., BERRY, M., COX, K., KEUNG, J., KURNIAWATI, F., STAPLES, M., ZHANG, H., AND ZHU, L. Evaluating guidelines for empirical software engineering studies. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering* (New York, NY, USA, 2006), ACM, pp. 38–47.

[21] MADSEN, O. L., AND MØLLER-PEDERSEN, B. What object-oriented programming may be - and what it does not have to be. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'88)* (1988), Springer, pp. 1–20.

[22] NATIONAL RESEARCH COUNCIL. *Computer Science: Reflections on the Field, Reflections from the Field*. National Academy Press, 2004.

[23] PFLEEGER, S. L. Soup or art? the role of evidential force in empirical software engineering. *IEEE Software 22* (2005), 66–73.

[24] POPPER, K. In *Logik der Forschung* (2007), H. Keuth, Ed., Akademie Verlag GmbH.

[25] PRECHELT, L. *Kontrollierte Experimente in der Softwaretechnik*. Springer, Berlin, March 2001.

[26] PRECHELT, L., UNGER, B., AND TICHY, W. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Transactions on Software Engineering 28* (2002), 595–606.

[27] SHEIL, B. A. The psychological study of programming. *ACM Comput. Surv. 13*, 1 (1981), 101–120.

[28] SHNEIDERMAN, B. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, August 1980.

[29] SHNEIDERMAN, B., AND PLAISANT, C. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 5. ed. Pearson Addison-Wesley, Upper Saddle River, NJ, 2009.

[30] SJØBERG, D. I. K., DYBA, T., AND JØRGENSEN, M. The future of empirical methods in software engineering research. In *FOSE '07: 2007 Future of Software Engineering* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 358–378.

[31] SJØBERG, D. I. K., HANNAY, J. E., HANSEN, O., BY KAMPENES, V., KARAHASANOVIĆ, A., LIBORG, N.-K., AND C. REKDAL, A. A survey of controlled experiments in software engineering. *IEEE Trans. Softw. Eng. 31*, 9 (2005), 733–753.

[32] SNELTING, G. Paul Feyerabend und die Softwaretechnologie. *Informatik-Spektrum 21*, 5 (October 1998), 273–276.

[33] SNELTING, G. Feyerabend - zwei Jahre später. *Softwaretechnik-Trends 21*, 1 (February 2001), 40–43.

[34] SOMMERVILLE, I. *Software Engineering*, 9. ed. Addison-Wesley, Harlow, England, 2010.

[35] TICHY, W. F. Die Bedeutung der Empirie für die Softwaretechnik. Keynote at German Conference on Software Engineering, Essen, March, 8-11, 2005.

[36] TICHY, W. F. Should computer scientists experiment more? *IEEE Computer 31* (1998), 32–40.

[37] TICHY, W. F., LUKOWICZ, P., PRECHELT, L., AND HEINZ, E. A. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software 28*, 1 (1995), 9–18.

[38] UNGAR, D., AND SMITH, R. B. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented Programming Systems, Languages, and Applications* (December 1987), ACM, pp. 227–242.

[39] WEINBERG, G. M. *The Psychology of Computer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1985.

[40] WILKINSON, L., AND THE TASK FORCE ON STATISTICAL INFERENCE. Statistical methods in psychology journals: Guidelines and explanations. *American Psychologist 54* (1999), 594–604.

[41] WOHLIN, C., RUNESON, P., HÖST, M., OHLSSON, M. C., REGNELL, B., AND WESSLÉN, A. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[42] ZELKOWITZ, M. V., AND WALLACE, D. R. Experimental models for validating technology. *Computer 31* (1998), 23–31.