Information and Software Technology: Accepted Manuscript “in Print” version see
http://www.sciencedirect.com/science/article/pii/S095058491500124X

What Does it Mean to Use a Method?
Towards a Practice Theory for Software Engineering

Yvonne Dittrich

ydi@itu.dk
+45 7218 5177

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S, Denmark

Abstract

Context: Methods and processes, along with the tools to support them, are at the heart of Software Engineering as a discipline. However, as we all know, the use of the same method does not necessarily result in comparable impacts on software projects nor on their outcomes. What is lacking is an understanding of how methods affect software development.

Objective: The article develops a set of concepts based on the practice-concept in philosophy of sociology as a base to describe software development as social practice, and develop an understanding of methods and their application that explains the heterogeneity in the outcome. Practice here is not understood as opposed to theory, but as a commonly agreed upon way of acting that is acknowledged by the team.

Method: The article applies concepts from philosophy of sociology and social theory to describe software development and develops the concepts of method and method usage. The results and steps in the philosophical argumentation are exemplified using published empirical research.

Results: The article develops a conceptual base for understanding software development as social and epistemic practices, and defines methods as practice patterns that need to be related to, and integrated in, an existing development practice. The application of a method is conceptualised as a development of practice. This practice is in certain aspects aligned with the description of the method, but a method always under-defines practice. The implication for research, industrial software development and teaching are indicated.

Conclusion: The theoretical/philosophical concepts allow the explaining of heterogeneity in application of software engineering methods in line with empirical research results.

Keywords

Cooperative and Human Aspects of Software Engineering, Software Engineering Method and Theory

1 Introduction

Methods and processes that rationalize and support the development of quality software are at the heart of software engineering as a discipline. And the process models, modelling approaches, architecture patterns and programming techniques, together with the tools to support them are widely used to improve the practice of software engineering. It is, however, difficult to pinpoint and quantify effects in

practice: While effectiveness of specific techniques for isolated tasks (such as the effectiveness of reading vs. testing for finding bugs, respectively, different reading strategies) can be measured and compared [1], the comparison of project-level measurements, based even on a population of similar projects, is still not possible [2].
Qualitative empirical research indicates that software teams balance what is recommended by the method with the specific technical and organizational circumstances of the project. Button and Sharrock, for example, report a specific interpretation of Yourdon development methodology, CASE and C in reaction to the specific contingencies [5]. They argue that 'methods are *worked at* phenomena, that they are made to work in the circumstances of their deployment and that the details of that work are part and parcel of the development process' [5, p. 237, highlighting as in the original]. Early on, software engineers recognized that tools supporting software processes need to support exceptions and adaptations to allow developers to react to situated contingencies [4]. Martin et al. report a software team balancing optimal test design, computer resources and people, and organizational circumstances when testing software [6]. In some cases, the applicability of a method depends on how the business area is organized [7]. The 'work arounds' when applying SCRUM have in in the agile community gotten an own the name even: 'scrumbut' [8]. Based on surveys and interviews, Fitzgerald concludes that only 6% of all practitioners apply a formally defined method [55].

So what is responsible for the difference between the method as described and the method in use? Researchers take varying positions. Some argue that software practitioners are not educated and trained well enough, as e.g., Parnas takes this view in his debate article on empirical research, which acknowledges the situated rationale of observed non-compliance with methods and disciplinary norms [9]. In the article 'The rational Design Process – Why and how to Fake it' [53], Parnas and Clements acknowledge the impossibility of following a 'rational design process.' The authors, however, recommend following it as closely as possible and complementing the documentation with additional information regarding the design decision taken.
A second position is to argue that the methods do not fit the situated contingencies, as especially authors inspired by Computer Supported Cooperative Work (CSCW) propose; thus, practitioners need to work around what the method prescribes in order to get their tasks done [6, 7, 10]. Fitzgerald provides a more comprehensive and differentiated appraisal of both positions in [54].
A third position proposes that we might need to revise our understanding of methods and practice. Button and Sharrock, e.g., conclude their discussions with: "If we think that methods are procedural recipes to follow we might think that all we have to do is to develop or alight upon the best method for our purposes and our problem will be solved by cranking the methodological handle. … If instead of thinking of methods as procedural recipes to be used in the course of development, we think of them as tools in the organisation of development, then the artful and contingent *use* of those tools is as important as the character of the tools themselves" [5, p. 237]. Arguing along a

similar line, Fitzgerald et al. distinguishes between formalized methods and methods-in-action, and propose several levels of tailoring and appropriation of methods [56].

This article aims at taking the discussion a step further: It argues that we need to develop a theoretical base for understanding software development as a social practice in order to understand how methods and tools are appropriated in everyday software development. In other words, what is needed is a practice theory of software engineering. The purpose of such a theory should be to help explain the phenomena that we observe in empirical research in software engineering. The goal is to be able to address the question indicated in the title of this article “What does it mean to use a method?” not only empirically but also based on a set of concepts that allows the explaining of the observed phenomena.

In their article ‘Theorizing about Software Development Practice,’ Pävärinta and Smolander [11] propose developing a theory of software development practice based on empirical research. This current article proposes a specific way to conceptualise practices, their rationales and their relation to contextual contingencies. The aim is to encourage the discussion of software engineering method and theory (SEMAT) [52] that includes a theoretical underpinning of the social side of software engineering.

To develop a practice concept, the current article appropriates concepts from social theory. I use Schatzki’s concept of integrated practice to describe software development as shared social practices based on common understandings, rules and teleoaffective structures [12]. Based on Knorr Cetina’s concept, I argue that software development is an epistemic practice, one that unfolds its object as the team proceeds in the development [14]. Based on this foundation and referring to software engineering discussions of the character of methods, I define methods as practice patterns, explicitly formulated sets of (tool supported) understandings, rules and teleoaffective structures that need to be integrated in existing practices.

The genre of this article is thus philosophical argumentation: it develops concepts based on literature and shows that these concepts can be used to better explain empirical results. Examples of such argumentations are the articles by Knorr Cetina [13] and Schmidt [15] that are further discussed below. The quality criteria for philosophical argumentation are subject to philosophical sub-disciplines and in part also depending on the philosophical school the argument is contributing to. In the context of Software engineering, I propose to apply a) rigour of argumentation and b) relevance of results; for example, the theory should render results of empirical research as examples of the theoretical concepts and relations rather than idiosyncratic behaviour. The article requires its reader to adjust to an unusual style of argumentation. Philosophical texts tend to use longer citations in order to show how the original author defines a concept before it is adapted and applied in the new context. These citations are used as inline citations rather than formatted as an own paragraph.

The article is structured as follows: Section 2 introduces the Software Engineering discussion on methods and their usage. The discussion leads to an understanding of

software development as a social practice. Section 3 provides a contextualises the philosophical approaches used with respect to philosophy of sociology and CSCW. Section 4 'A practice concept for software engineering' develops the conceptual base; the concepts of social practice and epistemic practice are presented and software engineering is described as epistemic practice. Based on these concepts, section 5 'Methods and method usage' then addresses the research question regarding the usage of methods requiring integration in, and adaptation to, an existing practice with its specific setting and purpose, and further addresses the necessary substantial explicit adaptation of practice, which extends Schatzki's or Knorr Cetina's work. Section 6 'Practices are constantly maintained and developed' further explores the continuous adaptation and maintenance of practice referring to the concept of articulation work by Strauss. The results are summed up and discussed in section 7, and implications are proposed for research, industrial practice and teaching. Section 8 summarises the conclusions.

2 Methods and Practice in Software Engineering

The development and dissemination of methods in order to inform and improve practice are at the heart of software engineering. Nevertheless, surprisingly few researchers have discussed the character of methods and how they inform software development. This section begins with a discussion of methods in software engineering and then argues for a concept of software development as social practice in order to inform the development and usage of methods.

2.1 Methods in Software Engineering

In ordinary language, the term 'method' describes a systematic way of addressing an endeavour. For example, Webster's dictionary defines method as the following

> **method**
> 1**:** a procedure or process for attaining an object: as
> *a (1)***:** a systematic procedure, technique, or mode of inquiry employed by or proper to a particular discipline or art *(2)***:** a systematic plan followed in presenting material for instruction
> *b (1)***:** a way, technique, or process of or for doing something *(2)* **:** a body of skills or techniques
> 2**:** a discipline that deals with the principles and techniques of scientific inquiry
> 3**:** *a*: orderly arrangement, development, or classification**:** plan *b***:** the habitual practice of orderliness and regularity
> …
>
> Webster's Dictionary [30]

Likewise, SWEBOK defines methods as: "Software engineering methods provide an organized and systematic approach to developing software for a target computer" [31]. The ISO24774 defines a meta-model of methods that is meant to provide a notation for the description and engineering of methods consisting of stages, work units, work products, producers and actions that allow the description of

recommended life cycles, processes and actions [32]. In other words, methods comprise both: specification of work products as well as the definition of processes in which they are included.

Mathiassen [33] presents methods as formalizations – that is, systematic descriptions of approaches to system development or parts of it. The author proposes the following characteristics that define a method:

> "An **area of application**: a type of software to be developed using it, a way to organize the development process;
> A **perspective** consisting of assumptions about the nature of the system, organizations, the surrounding society and the purpose of the local organization;
> **Principles** for organizing the development process, splitting it into partial tasks;
> **Techniques** of work used in the partial tasks;
> **Tools** used in the application of the technique (diagrams, notations, or computer support)" [33].

Based on empirical comparison of analysis and design methods, Floyd proposes extending this list with three additional characteristics:

> "**Theories:** mathematical theories on which, e.g., the notations are based, and the underlying theory describing what software development is;
> **Coherence** describes whether a method applies a connected set of techniques and tools or whether they are difficult to relate;
> **Coverage**: Which tasks of the development process are supported" [34].

Of these characteristics, the principles, techniques and tools are by and large covered by the method meta-model of the ISO 24774. Whereas coherence and coverage might also be formulated in the meta-modelling language, the area of application, perspectives and theories are information beyond what can be expressed by describing life cycles, procedures and activities. These characteristics are of a different nature: they specify the conditions of application and the goal of applying the method; this means they provide the information that supports the adaptation of the method in a specific context.

2.2 Methods and Practice

As discussions of the characteristics of methods are scarce, the relation between methods as descriptions and the practice that they are meant to inform and improve is by and large not problematized in software engineering. Mathiassen et al. [37] observe that experienced practitioners do not refer to the method description in their day-to-day practice; they also do not follow the method very closely, but competently select and implement relevant activities and notations. The authors propose that methods are learning vehicles rather than informing practice.

In line with these findings, Fitzgerald argues for distinguishing *formalized methods* and *methods-in-action*, where the method-in-action describes the structured way of development that is taking place in practice. He anchors the development and enactment of a 'method-in-action' with the individual developer: "[I]t is suggested that methodologies are never applied exactly as originally intended. Different

developers will not interpret and apply the same methodology in the same way; nor will the same developer apply the same methodology in the same way in different development situations. Therefore, on any development project, the methodology-in-action is uniquely *enacted* or *instantiated* by the developer" [54].

In the book "Information Systems Development: Method in Action," Fitzgerald and his co-authors further develop the notion of adaptation tailoring of methods to develop a method-in-action [56]. They, however, do not define the notion of practice, nor do they define methods in a way that allows a description of the interaction between formalized methods, methods-in-action and practice in a coherent manner. If we want to understand the relation between software engineering methods and practice, we need to have an understanding of what constitutes software engineering as a practice. As Kraft and Nørbjerg argue: "Software practice is social practice" [62].

Practice has been discussed in the philosophy of sociology as a central concept, constitutive for the establishment and tradition of shared norms and rules. This article shows how these concepts can be applied to software engineering to explain the development and application of software engineering methods. To motivate the choice of Schatzki's and Knorr-Cetina's work as a foundation and to provide a background for the reader to (critically) appraise the following argumentation, the next section contextualises the chosen approaches in the Philosophy of Sociology.

3 Practice Theories in the Philosophy of Sociology

In philosophy, the notion of practice has been discussed since the time of ancient Greek philosophy. A historical overview of the development of the practice concept found Schmidt's article on 'The concept of "practice": What is the point?' [15]. Schmidt relates the development of the modern practice concept to enlightenment: the purpose of science is not only the understanding of the (godly) organization of the universe but also the improvement of human affairs; craft practice therefore becomes an important object and target for science-based improvements.

This viewpoint is most prominently visible in Kant's text 'On the common saying: this may be true in theory but it does not apply in practice' [16]. Instead of describing theory, poesis and practice as being opposite, Kant develops practice and theory as mutually constitutive. "A collection of rules, even of practical rules is termed a *theory* if the rules concerned are envisaged as principles of a fairly general nature, and if they are abstracted from numerous conditions which, nonetheless, necessarily influence their practical application. Conversely, not all activities are called *practice*, but only those realisations of a particular purpose which are considered to comply with certain generally conceived principles of procedure" [16, highlighting according to original]. In other words, theory is developed as abstraction over (successful) practices; in turn, practices use theories as guiding principles and procedures.

Schmidt thus defines practice as 'normative regulated contingent activity' [15, p. 437], arguing that the modern concept of practice focuses 'on the ways in which the competent actor in his or her action is taking the particular conditions into account while committed to and guided by the appropriate general principles ('theory', 'rules')' [15, p. 436]. Practice in this sense is not the opposite of science or academia.

Theoretical argumentation and research can be understood as practices in their own rights, governed by common values and a shared understanding of what is good research and acceptable academic behaviour.

Even though the change in the perception and appreciation of practice was initiated earlier, the ‘practice turn in contemporary theory’ [13] is related to the publication and reception of Wittgenstein’s Philosophical Investigations. Wittgenstein’s late work develops a practice-based epistemology. Instead of basing the meaning of terms on their correspondence to the world [17, pp. 392ff], Wittgenstein in his Philosophical Investigations [18] argues that the meaning of words and utterances is founded in the way we use terms in our ‘language games.’ Understanding and participation in language games are based on a common way of living and an enculturation into patterns of interaction with the outside world and with each other using language. By anchoring the meaning of utterances in social practice, the Philosophical Investigations further explain other rule-based social activity referring to the notion of practice.

Wittgenstein’s Philosophical Investigations have inspired a number of contemporary schools in social science and philosophy. In his introduction to these schools, Nicolini talks about practice theories in plural [19]. To base our argumentation in a coherent and consistent manner, we refer to Schatzki’s ‘Social Practice: A Wittgensteinian approach to human activity and the social’ [12] as well as Knorr Cetina’s article ‘Objectual Practice’ [14].

Schatzki’s ambition is to develop a foundation for understanding sociality as a ‘nexus of integrative and dispersed practices’ [12, p. 173]. He writes that his ‘account of sociality (…) does not claim completeness as an account of social life. In laying out the basic nature of the social, it instead constitutes a framework through which to investigate social domains and phenomena and uncover their local details and complexities.” [12, p. 173] The current article aims at doing exactly this: it conceptualizes software engineering as a social practice, expanding Schatzki’s concepts in order to make sense of empirical observations that problematize the status of methods in software engineering.

The further argumentation below also refers to approaches and findings from sociology of work such as that found in Strauss’, Gerson and Star’s and Schmidt and Simone’s work [20, 21, 22], who ground their work in practice concepts compatible with Schatzki’s concept. Strauss, Gerson and Star relate to American Pragmatism as their theoretical underpinning. Although there is only a weak explicit link between Wittgenstein and the founders of American Pragmatism, Dewey, Mead and Peirce [23], the second generation of American Pragmatists embrace the later work of Wittgenstein together with other European thinkers [24].

The pragmatic schools on both sides of the Atlantic refer to social practice and interaction as the underpinning of meaning both in language and in action. Garfinkel, e.g., the founder of ethnomethodology, refers to Wittgenstein’s Philosophical Investigations [25], where the founding of social phenomena such as language and rules in social practice is formulated for the first time. Ethnomethodology aims at

understanding the social through the methods that a socio-cultural group, an ethne, deploys in its everyday interaction and collaboration.
The next section develops the central concepts. Based on Schatzki’s concepts, software development is described as shared social practices based on common understandings, rules and teleoaffective structures [12]. Knorr Cetina’s work is used to argue that software development is an epistemic practice that unfolds its object and, with it, its own practice as the team proceeds in the development [14]. This foundation is later used to define methods as practice patterns, explicitly formulated sets of (tool supported) understandings, rules and teleoaffective structures that need to be integrated in existing practices.

4 A Practice Concept for Software Engineering

This section develops the central concepts as foundation for the following argumentation. The concepts presented here have the same function that axioms and proven theories have in a mathematical proof. This is also the reason for the rather long quotes: Instead of only providing an own interpretation, the original voice of the author cited is used. The subsection 4.1 presents Schatzki’s notion of practice and applies it to software engineering. In 4.2, also based on Schatzki’s ‘Social Practice,’ the role of tools, equipment and settings in social practice is developed.
As a starting point for discussing diversity and evolution of software development practices, e.g., when introducing methods, subsection 4.3 discusses the way Schatzki conceptualizes change and flexibility of practices. Comparing his concepts to results of empirical research in software engineering leads to the recognition of the need to further develop the practice concept to be able to grasp the heterogeneity and diversity of software development practices. In section 4.4, Knorr Cetina’s notion of epistemic practice is used to explain the diversity of software engineering practices. As software engineering is a practice geared towards design and development of a thus-far unknown piece of software, it, at the same time, unfolds its goal and the practice, tools and techniques deployed to get there. These concepts then are used to define methods as practice patterns in section 5.

4.1 Software engineering as social practice

The aim of Schatzki’s book: ‘Social Practices. A Wittgensteinian approach to human activity and the social’ is to explicate and further develop the notion of practice as a basis for conceptualizing the social. Practice thus is the central concept Schatzki discusses. Schatzki defines, based on Wittgenstein, practice as a “… temporally unfolding and spatially dispersed nexus of doings and sayings. (…) to say that the doings and sayings forming a nexus is to say that they are linked in certain ways. Three major avenues of linkage are involved: 1) through understandings, for example, of what to say and do; (2) through explicit rules, principles, precepts, and instructions; and (3) through what I will call teleoaffective structures embracing ends, projects, tasks, purposes, beliefs, emotions, and moods.” [12, p. 89]
Schatzki uses this concept of practice to explain the coming about of everyday activity that allows us to relate meaningfully to each other and to pursue relevant

goals in specific contexts: When acting in a certain situation, as part of a certain practice, what one says and what one does is understood by others in relation to the commonly shared practice and with respect to the shared goals and purposes of that practice.

Practice should thus not be confused with ad hoc behaviour, even if the rules and principles guiding the acting are not expressed explicitly. People sharing a practice would easily recognize action as ‘making sense’ in the specific context. Ad hoc behaviour can be seen when exceptions occur that require action to bring the state of affairs back to normal. Even in such cases, the actions make sense, given the understanding of underpinning practice. The article ‘When plans do not work out’ [26] analyses a situation that both shows the work of practice and of ad-hoc behaviour: a project steering group is confronted with a re-scoping of a project, leading to a subproject having to rework central documents. They develop a future line of action that allows the project to progress orderly by partly violating the company wide project model. The exception is formulated in terms of this project model, making the ‘ad hoc’- behaviour accountable with respect to the organization’s software development practice, which is supported by the terminology and rules provided by the project model.

Schatzki distinguishes between dispersed practices and integrated practices. Dispersed practices are widely shared practices, such as “asking for and giving explanations, describing, ordering that occur in different sectors of social life” [12, p. 91]. The term integrative practice is used for “more complex practices found in and constitutive of particular domains of social life” [12, p. 98]. The author gives business practices, voting, teaching, celebration, cooking, recreational, industrial, religious and banking practices as examples [12, p. 98].

Dispersed practices can become part of integrative practices and develop a special flavour as part of them. As an example, Schatzki cites the specific meaning of orders in the military; though ordering is a dispersed practice, “when someone enters, say, military life, the understanding of ordering he has previously acquired becomes sensitized to the particular way the activity runs on there…” [12, p. 100].

Integrative practices are linked together by: (1) understanding of the actions involved, including the ‘transfigured forms of the dispersed practices’; ‘(2) explicit rules, principles, precepts, and instructions; and (3) teleoaffective structures comprising hierarchies of ends, tasks, projects, beliefs, emotions, moods and the like’ [12, p. 98-99]. Schatzki defines the notion of teleoaffective structures as ‘hierarchized orders of ends, purposes, projects, actions, beliefs, and emotions’ [12, p. 100] that are expressed in the behaviours constituting a practice. ‘Unlike explicit rules, the orders constituting a teleoaffective structure need not be spelled out and explicitly enjoined in formulations, although formulation does sometimes occur, especially (but not only) in learning situations, in the face of nonstandard doings and sayings, and on the occasions when the flow of reactions suffers … “breakdowns” in continuous absorbed coping’ [12, p. 99, referring to Dreyfus’ discussion of Heidegger, 27].

Software engineering as social practice falls into the category of integrated practices. Dispersed practices still exist, but they change character; for example, the practice of

asking and giving of explanations will also be observed in software engineering, but it changes character and meaning. An example can be found in the dialogs between long-term project members and newcomers in the pair programming sessions analysed by Plonka et al. [58], where the explanations are the base for both knowledge sharing and improvement of the existing code.

To say that software engineering is a social practice suggests that it consists of coordinated activities that are connected through a.) shared teleoaffective structures – here, the intention to design and develop a piece of software, b.) shared understandings of what it entails to develop software both in general as well as in a specific domain, and as part of a specific organization and c.) explicit rules and principles – examples would be company-wide methods and hand books, but also knowledge about methods, techniques and tools as taught in courses or in tutorials. An example for the latter can be found in a steering group meeting analysed in [26]. The company wide project model underpins the whole discussion; the rules stipulated by it are explicitly used as criteria for an acceptable and organizationally accountable action on a major re-scoping of the project. At the same time, the analysis shows an implicitly shared understanding that the purpose of the meeting is to allow the unaffected sub-teams to proceed with their work in a meaningful way.

However, as mentioned above, the notion of social practice here relates not only to industrial practice. Open source software development can be regarded as a practice as well. Open source software development is guided by shared understanding of what it requires to be a member of an open source project, by explicit rules (e.g., in the form of frequently asked questions), and by sharing teleoaffective, structural hierarchies of goals and endings that connect individual actions. (See for example the discussion of requirements in open source software by Scacchi [29].)

4.2 Tools, equipment and settings

As section 2 has shown, methods come with notations, techniques and tools. Schatzki discusses the relation between practices and tools. Practices help us to make sense not only of each other's behaviour but also of both the objects we manipulate and transform as part of the practices and the tools we use. The jointly enacted software development practice allows the team not only to understand each other's actions as being meaningful, but it also renders terms and tools meaningful. The terms steering group, sub-project and product owner in Rönkkö's case [26], cited above, acquire their meaning based on the shared practice of the observed team. As Schatzki puts it:

> "Not only people, but objects (and events) as well acquire meaning within practices. This occurs, most importantly, whenever objects are used in the performance of constituent actions. Teaching, for instance, encompasses writing on blackboards and other surfaces with certain entities, which therewith receive the meaning: things with which to write. … Like understanding generally, the understanding of equipment is expressed not only in doings (i.e., uses) but also in sayings. People give names to equipment and say of them that they have such and such practical meanings, for instance, that chalk and magic markers are

> things with which to write … An integrative practice, consequently, carries interwoven understandings of interrelated equipment” [12, p. 113-114].

The set of computer based tools, documents, common repositories, as well as environments automating builds and tests are examples from software engineering. A tool like Eclipse is intrinsically connected with software engineering and does not make sense outside the practice of software engineering. The dependency of the meaning of equipment on the specific practice it supports becomes more visible when we consider the way story cards are handled in Scrum. Although other kinds of practices might seem to have similar equipment (e.g. qualitative analysis of field material might use similar cards), the specific meaning of story cards in Scrum cannot be explained without referring to the way a Scrum team uses it to coordinate their development.

Scrum cards and boards are also examples where not only the meaning of the individual piece of equipment is dependent on the practice but where the setting in which the equipment is arranged and the arrangement itself is intrinsically interwoven with the practice it supports. Interrelated sets of equipment are organized in settings that are maintained as part of the practice.

> “When a practice, as is usually the case, is carried out in specific settings, the settings are set up to facilitate the efficient and coordinated performance of its constituent actions. The layouts of the settings, as a result, reflect the interwoven meanings that the entities used in these actions possess by virtue of being so used (and talked about). Settings in other words, are often set up as sites where a given practice or set thereof is to be carried out. When this occurs, the setups are derived from the understandings, rules and teleoaffective structure organizing the practice. The disclosure and layout of equipment within practices also consequently, exhibits normativity, meaning that things are usually so arranged that they can be easily used in the correct and acceptable ways” [12, p. 114].

The story cards on a Scrum board - their colour, annotations, and their placing - make sense, given the practices developed and established as part of the team’s common development [28]. Similarly, the layout of a website of an open source project with documentation, access to forum lists and bug report software, frequently asked question, and the software repository can be seen as such a (virtual) setting related to the practice(s) developed by the open source community. Scacchi has coined the notion of ‘informalism’ for specific pieces of equipment and settings of open source projects together with the way an open source project uses them [29]. These ‘informalisms’ can be seen as nexuses of understandings, tools and rules that support joint software engineering that is not related to an organizational and employment set up.

The representations, tools and communication channels make sense with respect to this specific way of organizing software development and derive their meaning from the shared practices. The examples cited above show that objects in such a setting are not only tools and equipment: objects and their states serve as well as indicators, such as the number of open issues, burn down charts, or other measurements. Similar to

equipment, such objects and indicators gain meaning through the practice. With Schatzki’s words: “How people talk about and act toward objects is not exhausted by how they use them. People also observe objects, examine them, measure them, admire them, draw them, and talk about them in numerous ways that do not pertain to use. The meanings that objects thereby come to bear are still established within practices to the extent that the ways of talking and acting in question are components of practices” [12, p. 114].

The whole purpose of the introduction of new methods and tools into a practice is the change of this practice. The next section develops how Schatzki conceptualizes change and relates it to software engineering practice change. This in turn provides the motivation to include Knorr-Cetina’s concept of epistemic practice in the conceptual base of the article.

4.3 Flexibility and change

Schatzki emphasizes that practices and the related understanding of behaviour, objects and equipment are open to new ways of doing things. The normative regulation of practices through the implicit understandings and explicit rules does not mean that practices are unambiguously defined:

> “A practice’s organization establishes not only that certain actions are correct (in certain situations), but also that other actions are acceptable, even if they are not how one should proceed. … I note that among the acceptable actions (and live condition orders) constitutive of a practice’s teleoaffective structures are some that have not yet been carried out (or instantiated). Practices found possible novelty in that people happen upon new ways of proceeding, and others deem these ways acceptable, on the background of their participation in practices and familiarity with teleoaffective structures. The understandings that organize an integrative practice likewise, though more weakly, open ranges of acceptable doings and sayings broader than the behavior already performed in the practice” [12, p. 102].

This flexibility provides not only the base for evolving practices, but also allows adapting the joint practice to situated contingencies. Software practices can still be recognized as such by software engineers across different projects and different ways to organize software engineering. An example would be: an issue tracker with its usage to report, analyse, triage and correct bugs is well known across many software organisations as well as in open source communities. The specific way how the triage results are communicated will differ, however, from organization to organization, from open source project to open source project. In other words, the specific practices around bug fixing and with it the usage of an issue tracker will differ, depending on the local adaptation of the more generic practices and understandings.

To emphasize this diversification of practices with respect to the local contingencies, I discuss below local practices that have been adapted in specific ways to the individual organizations, even in specific projects. As practices in the sense developed here do not exist but in the interlinked doings and sayings constituting it (Schatzki p. 90), one could argue that integrative practices are always also local practices.

The different local contingencies though might not be the only factor that explains the diversity of software engineering practices. The following subsection further explores the specificities of design and development practices, where the outcome is not fully defined ex ante. This is very much the case in software engineering. To this end, the next section introduces the notion of epistemic practice.

4.4 Software engineering as epistemic practice

To understand the heterogeneity of software engineering practices, the introduction of another concept further developing Schatzki’s notion of practice is necessary. Schatzki et al. [13] emphasize the foundations of social structures in the regularity of social practices. In her article, ‘Objectual Practice,’ Knorr Cetina [14] develops Schatzki’s [12] notion of practice, further emphasising their teleoaffactive dimension. She describes science and design as epistemic practices. Based on fieldwork on the development of a particle accelerator, she argues that research ‘seems to be particular in that the definition of things, the consciousness of problems, etc., is deliberately looped through objects and the reaction granted by them’ [14, p. 175]. She later includes design and other knowledge intensive practices in that definition.

Epistemic practices aim at the developing and unfolding of only partially existing and known objects, characterized by their ‘lack in completeness of being’ [14, p. 181]. “[O]bjects of knowledge in many fields have material instantiations, but they must simultaneously be conceived of as unfolding structures of absences: as things that continually ‘explode’ and ‘mutate’ into something else, and that are as much defined by what they are not (but will, at some point have become) than by what they are” [14, p. 182]. In other words, epistemic practices are practices underpinning creative activities that bring something that is not yet there into being.

In her conclusion, Knorr Cetina proposes that ‘Knowledge-centred work shifts back and forth between performance of ‘packaged’ routine procedures and differentiated [epistemic] practices’ [14, p. 187]. Defining software engineering as epistemic practice suggests that software engineering, on the one hand, is based on established ways of doing design, guided by explicit and implicit understandings of how things are done and explicit rules concerning methods and techniques, but, on the other hand, the teleoaffective structure guiding the development is the idea of a *piece of software that does not yet exist*. While its object - the software under development - unfolds, software development as practice unfolds itself, thus completing the developing understanding of what the goal is of its activities. The creative activity that brings about the object is itself an object of design as part of this process; in other words, it is the object of a creative activity itself.

As different software engineering projects have different objects of design and as the unfolding of the epistemic objects affords a corresponding unfolding of the design and development practices, software engineering practices, though clearly perceivable as practices of a similar kind, will also always differ, even in the same company with the same contextual characteristics. This means that the locality of software engineering practices is not only due to different settings and different contextual contingencies but also needs to be attributed to the specificity of its epistemic or design object.

The central concepts discussed thus far have been based on Schatzki’s concept of social practice and Knorr Cetina’s concept of epistemic practice. Software development as social practice according to Schatzki [12] can be understood as a nexus of doings and sayings that are kept together by common understandings, common explicit rules and principles, and common teleoaffective structures. It is supported by a setting and a set of tools and techniques that are rendered meaningful in the context of this practice. By proposing to regard software development as epistemic practice [14], the creative side of software development is highlighted: the teleoaffective structure or objective of software development is to bring a piece of software into being that is not yet there but needs to be defined, designed and developed through the practice. This also means that in order to bring this epistemic object about, the practice itself needs to develop.
To understand how methods nonetheless are useful to support widely heterogeneous design and development practices, the next section develops a concept of methods that can then be used to grasp the adaption and adoption of methods in a concrete practice.

5 Methods and method usage

Based on the conceptual base developed above, this section develops the concepts of method and method usage in a way that allows us to understand the interaction between methods and practices. Sub-section 5.1 proposes understanding methods as practice patterns consisting of explicitly formulated understandings, explicit rules, explicitly stated teleoaffective structures that need to be adopted and adapted to fit with the specific epistemic practice. This concept is in line with, and provides a theoretical underpinning for, previous research results on methods and their use in software engineering. Subsection 5.2 then further explores how methods inform practice, referring to Wittgenstein’s Philosophical Investigations. Subsection 5.3 shows how the thus-far developed concepts can be used to explain and develop empirically observed phenomena around the adaptation and use of methods in software engineering.

5.1 Methods as practice patterns

In section 2.1, methods were defined as explicit descriptions of approaches to system development or parts of it. According to the ISO 24774, such descriptions define stages, work units, work products, producers and actions. According to Mathiassen [33] and Floyd [34], methods are further defined by the conditions of application and the goal of applying the method underpinning the notations, work products, processes and activities. But how can we understand methods based on the conceptual base developed above, and how can we relate them to social practices based on Schatzki’s and Knorr Cetina’s terms?
Holding Schatzki’s definition of practice up against the constituting elements of methods, as defined by Mathiassen [33] and Floyd [34], methods can be defined as description can be seen as complex related sets of explicitly formulated

understandings (notations, modelling languages, concepts, area of application, coverage, the mathematical side of the theories), explicit rules (processes, task descriptions, techniques etc.), and more or less explicitly stated teleoaffective structures (principles, perspectives, theories in the sense of what software engineering is about). Such complex sets of (tool supported) explicitly formulated understandings, rules and teleoaffective structures can be seen to be similar to the architecture and design patterns by Gamma et al. [35], respectively, Alexander [36]. They are proven solutions for specific practical problems. In the following, I call these related sets of understandings, rules and teleoaffective structures practice patterns.

As with design patterns, the description of a practice pattern is not the usage of the pattern. In line with Alexander, Gamma et al. claim that a pattern is never used in exactly the same way. This indicates that the use of a design pattern is not the same as using a cookie cutter, but requires adoption and adaptation. In other words, it does not replace design activity but supports it. Likewise, methods understood as practice patterns need to be adopted and adapted to the situated contingencies of the specific project. Above, I have argued that software engineering is an epistemic practice [14] that is geared to bringing about its object. The central rationale for adopting and adapting a method is therefore whether it helps to bring about the software a project is meant to develop under the given circumstances.

As with the architecture and design patterns, applying a method or a practice pattern therefore does not mean doing exactly the same thing as the neighbour project applying the same method. The participants in the practice of software engineering embed the practice patterns the method consists of with heed to their knowledge of this complex and related set of (tool supported) understandings, rules and teleoaffective structures. The adoption and adaptation of the method is part of the creative activity that is the dimension of an epistemic practice that evolves that practice in order to bring the epistemic object into being.

This is in line with, and explains, Fitzgerald’s distinction of ‘formalized methods’ and ‘methods-in-action’ [54]. Practice patterns can be seen as formalized methods, whereas the adopted and adapted practice pattern then is the method-in-action. In other words, the philosophical argument developed here provides a theoretical foundation and explanation to the phenomena Fitzgerald observed. But how do we get from a formalized method to a method in action? Or – using the terminology developed here – what does it mean to use a practice pattern? Below I further explore what it means to embed formalized methods in concrete practices and in that way, create a method-in-action.

5.2 Embedding Methods in Concrete Practices.

In order to inform software development practices, the methods – understood as practice patterns – need to be integrated into the practice of the project team that decides to use the methods. To do so, the participants of the practice need to make sense of the practice patterns. As was argued in section 4, no explicit formulation – and that includes the explicit formulation of rules, understandings, or of teleoaffective structures – has a meaning per se; rather, the participants in a practice make sense of

them in relation to the established practice. The same needs to hold for methods as practice patterns: The description of activities, work products, producers, procedures, and life cycles do not have any meaning per se. They acquire meaning only in relation to the practice at hand. This implies that the understandings, rules and teleoaffective structures, together with the objects and tools that exist prior to the usage of a method, influence the meaning and application of the method understood as practice patterns.

Practice patterns can be understood in line with Wittgenstein’s discussion of explicit rules. Wittgenstein discusses the issue of rules and rule following at length, as the understanding of rules underpins the pragmatic theory of language and the discussion of logic and mathematics in the philosophical investigations. Rules, like signpost, do not determine what it means to act according to them [18, Philosophical Investigation (PI) 201]. To develop an understanding of meaning rooted in practice, Wittgenstein uses the metaphor of a game: the meaning of words is defined by the way we use them in everyday or ordinary language rather than by a correspondence to some aspect of the world. These ways of usage are commonly shared ‘ways of doings and sayings’ [12]. The meaning of explicit rules are similar to the meaning of other words and sentences, defined by the way they are used and followed in specific language and behaviour. Accordingly, Wittgenstein states that ‘following a rule is a practice’ [18, PI202]. This means that even if explicit rules influence the behaviour, what it means to follow them is defined by the community whose practice adopts the rules.

Applied to software engineering, when making sense of the description of principles, guidelines, processes, work products, tools, techniques, and so on in the context of their prior existing practices, these descriptions become meaningful. The meaning of the descriptions though is dependent on the practice embracing them and is a result of both conscious and unconscious adaptation of the method.

As architectural patterns [36] or design patterns [35], method understood as practice patterns can be applied again and again without ever doing the same thing twice. Similar ideas have been voiced earlier, often based on research or experience from industrial practice.

Based on industrial as well as research experience, Floyd proposes: “We do not apply predefined methods, but construct them to suit the situation in hand. There is no such thing as methods per se – what we are invariably concerned with are processes of situative method development and application. We select methods and adapt them. What we are ultimately doing in the course of design is developing our own methods” [38, p. 95]. In the project at hand, the members need to develop a common understanding of what it means to apply the method; in other words, a local practice defines what it means to apply the method. Without such a practice, the method is meaningless. This means that each team defines its *method-in-action* [54], whether the team intends to do so or not. Such adoption and adaptation might again change the explicit description of how software is to be developed, leading to a local version of the method. Expressed in Wittgenstein’s terms, they develop the rules of the game while playing [18, PI 83].

Based on an ethnomethodological underpinning, Rönkkö [39] argues in dialogue with Mathiassen [33] that the application of methods is always an interpretation that relates

the abstract descriptions, respectively, formalizations to the situation at hand. He further argues that this interpretation is a continuously ongoing achievement. Through the action in line with the commonly agreed on interpretation as well as explicit demarcations of exceptions as such, the agreed upon interpretation underpinning the common practice is maintained [26].
The next subsection shows that method adaptation as method design has been observed also in empirical research.

5.3 Empirical evidence for the local adaptation and design of methods-in-use
The local project-specific adaptation of methods normally does not pose a serious problem. Of course, the designer of the methods might argue that the full potential of the methods is not deployed. It can become problematic though if different sub-teams of the same project embed the common methods and tools in different ways. This seems to happen especially in distributed software development. Damian et al. [40] report on a case of a team distributed between USA and Canada, using the same method and tooling. One issue was the use of the Code Versioning System. Whereas in one locality, the team members relied on the commit comments distributed to the whole team, the team members in the other locality used extra mails distributed via a mailing list to highlight changes affecting other members of the team. Damian et al. attributed the resulting breakdowns to cultural differences. In line with the theoretical underpinning above, culture can be understood as a connected set of both dispersed and integrated practices. This is in line with an ethnographical understanding of culture: “But ethnography isn't *just* about shared knowledge; rather, it's about the *practices of everyday life,* the way those practices are built out of shared knowledge, *plus* all the other things that are relevant to the moment.” [59, p. 9, highlighted as in original] In other words, the local practices embedded and interpreted the tool in different ways.
In his PhD thesis on “The purposeful adaptation of practice: an empirical study of distributed software development” [44], Sigfridsson poses that both industrial project teams and open source development communities discuss and adapt their practices to address changes in the environment or as problematic recognized aspects of the current practice. Draxler et al. discuss how teams organize the keeping up-to-date with developments of their tools, and assign members to take on the tailoring and adaptation of new features for the whole project [41].
Giuffrida and Dittrich [42, 43] show how student project teams when initiating the project and on occasions of a breakdown will discuss and introduce new ways of cooperating and coordinating. Further, successful collaboration, especially in distributed projects, seems to coincide with the establishment of such explicit negotiations and agreement on how to do things together.

This section has developed the concept of methods as practice patterns, consisting of explicitly formulated understandings, explicit rules, explicitly stated teleoaffective structures that need to be adopted and adapted to fit with the specific and situated epistemic practice. The empirical research cited indicates that the adoption and

adaptation of methods, tools, and processes is part of the routine software development practices. The empirical research cited above indicates that the adaptation and adoption of methods, processes and tools is a normal part of software development. This aspect is, however, hardly discussed in the theories of sociology. I can neither be subsumed under what Schatzki describes as “people happen upon new ways of proceeding” [12, p. 102]. The reason might be, as Knorr Cetina also states [14], that sociological theorists are interested in understanding the establishment of social order before addressing its change. In her article she points to science and design as epistemic practices that unfold both the object they are about to bring into being and the practices necessary for this purpose [14]. The next section draws on findings from the sociology of work to further explore the design of the design practices.

6 Practices are constantly maintained and developed

In the discourse of sociology of work, the structuration and articulation of the organization of distributed activities from within have been the subject of discussion and empirical research. Below, first Strauss’ notion of articulation work [20] and Gerson’s notion of meta-work [57] are used to further explore the conscious adaptation of practice; thereafter, empirical research is used to exemplify such practices of changing practices.

6.1 Articulation work and the practice of changing practices

The organization of distributed work certainly requires planning and management, but it is also an achievement of the cooperating workers who need to structure and articulate their individual contribution in such a way that allows their co-workers to relate to the same. Strauss introduced the notion of articulation work in order to conceptualize the organization of work from the inside of the joint endeavour. Such joint projects require not only individual tasks, but an ‘arc of work.’ “Articulation work amounts to the following. First, the meshing of the often numerous tasks, clusters of tasks, and segments of the total arc. Second, the meshing of efforts of various unit-workers (individuals, departments, etc.). Third, the meshing of actors with their various types of work and implicated tasks.” [20]

With the term articulation work, Strauss [20] refers not only to the establishment and planning but also to the standardization of cooperation procedures, the articulation of tasks in order for others to relate to it, and the handling of contingencies that lead to exceptions from the standard. Based on the development of the concepts and demonstration using work-studies in hospitals, Strauss discusses – partly as proposals for future work – that different aspects influence how stable the division of labour is and with that the character of the articulation work involved. Referring to him, Gerson [57] distinguishes between: a) situated articulation, the articulation of tasks in order for others to relate to it and the handling of contingencies that lead to exceptions from the standard, and b) meta-work, establishment, planning and the standardization of cooperation procedures.

6.2 Articulation and meta-work in Software Engineering

The concepts of articulation work and meta-work are widely used in CSCW literature in the analysis of cooperative practices and the difficulty of supporting them. The renegotiation of the project plan analysed by Rönkkö et al. [26] provides an example of local articulation. Giuffrida and Dittrich’s [43] analysis of communication via social software shows the importance of (successful) meta-work for the establishment of the coordination procedures and mechanisms in distributed software development. Situated or local articulation is then used as part of deploying the coordination mechanism in the specific context and when breakdowns occur.

Sigfridsson’s PhD thesis [44] and the article by Draxler et al. [41] that motivated the exploration of meta-work both report concrete examples of meta-work practices. Note that in these cases, established ways of discussing the use of methods and tools, and the current practices existed. In other words, meta-work in itself is a practice that depends on common understandings, explicit and implicit rules, and teleoaffective structures, and is therefore itself subject to meta-work. An example of such meta-meta-work can be found in Sigfridsson’s PhD thesis [44], where an industrial team asks a sub team to explore how to embed the new tools that were mandated to support the distributed work.

Both the conceptual and the empirical argumentations are in line with Knorr Cetina’s concept of epistemic practice [14]: As the object of software development is unfolded, so do the practices geared to bring about the software need to unfold. Strauss’ concepts further elaborate both on how practice is coordinated and how it evolves. With these concepts, the last piece in the theoretical work that the article set out to achieve is provided. The next section discusses implications for research, practice and teaching before the conclusion summarises the path of the argumentation and revisits the discussion on ‘bad methods’ or ‘bad practice’ that motivated the article.

7 Implications for research, practice, and teaching

The article set out to provide a conceptual base to discuss the deployment of methods that developed around empirical research results, thus challenging the relationship between methods and their use. In the above sections, we have developed a set of theoretical-philosophical concepts that allow us to make sense of results of empirical software engineering research. This section - standing in lieu of a discussion - deals with the implications of the practice concept for software engineering and its application to elaborate on what it means to use a method.

7.1 Implications for research

One of the central goals of software engineering research is the development of tools, techniques and, last but not least, methods to support software engineering practice. Understanding software engineering as epistemic practice does not challenge this; however, it results in a different relation between research-based method development and industrial practice.

Empirically, we can identify two main ways in which methods come about: a) as abstractions of existing practice in order to communicate useful practice patterns with

newcomers and fellow practitioners and b) as output of a special practice called research. Examples of abstractions from practice are the early software process models and the agile methods. The waterfall model presented and criticised by [45], Boehm’s spiral model [46] and Lehman’s ‘Laws of Software Evolution’ [47] are examples of methods as abstractions from practice. Likewise, the agile manifesto states the legitimation for its propositions and the methods developed and promoted by its authors: “We are uncovering better ways of developing software by doing it and helping others do it” as its first sentence [48].

Software engineering research, on the other hand, develops principles, guidelines, (mathematical) tools and techniques to be applied by software engineers. Programming language technologies, algorithms, methods for formal specification and also for architecture documentation are examples of this way of method development. Although software engineering research and software engineering practice both are epistemic practices, they, however, are two different practices, each adhering to its own understandings, rules and teleoaffective structures. This implies that software engineering researchers developing software engineering methods cannot use their own practice as paradigmatic for software development practices as such; rather, they need to actively search to understand the practices they are to support.

Here, empirical research of software practices becomes a vital input. Empirical research is not only about the experimentation with new methods in more or less controlled (quasi) experiments, but it is also about exploring the changing conditions under which software development takes place and providing knowledge to support the development and use of methods as practice patterns in industrial practices. The goal of this empirical research is, however, not to blindly appraise a practice which might or might not be problematic, but to understand the rationale of the observed practices in order to understand in which contexts the methods are to be applied, in which practices the practice patterns need to be instantiated and how these contexts influence the applicability of the practice patterns proposed.

Divergence from what method developers and research community recommend and diversity of practices should not by default be regarded as a problem but as source of understanding the rationalities of practice that then can inform method development and appropriation. Based on empirical research and an interview study, Unphon and Dittrich [49] propose that software engineering practices, which researchers may consider to be problematic, might be kept for good reasons. The interviewed software architects and lead-developers found it problematic to rely on software architecture documents: the access of developers to the written architecture would replace face-to-face meetings, thus cutting the software architects from an important source of information about problematic aspects of the current architecture. Proposing different documentation techniques would not help this problem. Together with the tools and techniques, new social protocols needed to be established to support the architect. The only architect who maintained a wiki documenting the architecture reported the necessity of daily checking the wiki and the commit notifications in order to keep up-to-date with the changes to the structure of the software. These results provide an

indication that in the development of software architecture documentation tools important practice requirements were not addressed.

It is not a co-incidence that the majority of the empirical research cited in this article is using qualitative research and is often inspired by ethnography. Ethnographic studies aim at understanding practice from a member’s point of view [60]. The focus on how things are worked out rather than on the shortcomings of practice allows understanding the rationalities of practice.

Understanding the rationalities of practice is important to understand how new technologies, business models and innovations in the use context are accommodated and change of the contexts in which methods are used and to which they need to be adopted. As new techniques and new forms of using and deploying software arise, the teleoaffective structures, the tools and with these, the rules and understandings evolve. An example here is the development of software products like ERP systems, which substantially differs from software practices in contract development: whereas contract development aims at delivering a more or less specified piece of software, software products are supposed to be used by many customers over a long time and need to evolve constantly to support the development in the market [50]. This in turn impacts the development processes, the architectural practices and the way the software development relates to the users and customers. By focusing on the project as the way software development takes place, the challenges of software product development have only recently been addressed in research [64]. Similarly, the current focus on continuous software engineering [63] and the DevOps development model [65] in industry is announcing a change that so far is mainly discussed in conferences and journals close to industry.

However, empirical research can provide more than a better understanding of the rationales behind different practices. It also can inform the adaptation and development of methods, tools and techniques. To this end, empirical research aiming at the understanding of rationales of practice can be combined with action research [60]. It thus can help not only to design and devise methods that are more useful, but also to develop knowledge to support the adaptation of practices and the integration of methods. That way empirical research can contribute to develop orientational knowledge that allows ‘designing design’ [38].

7.2 Implications for industrial practice

Although the empirical research of industrial software development cited above shows that software teams consciously work with the adaptation of practices, many software engineering practitioners still refer to methods as if they were programs to run on a computer; practitioners also see the adaptation of methods as a problematic practice. Here, the argumentation presented in this article proposes a different understanding of how methods inform practices. Methods are devised to address certain problems and support specific goals. In order to get the best out of methods, the integration of methods in the existing practice needs to be carefully deliberated and prepared: Do the proposed patterns of notations, guidelines, principles and rules, as well as the goals proposed by a method fit with the problem at hand? How will the

adoption of a method change the existing practice of the team? What aspects of the current practice need adjustment to harvest the intended benefits? And – when the method has been in use for a while – how does it work and what else is needed to support the team considering the new practices?

The adoption of methods should be monitored in order not only to understand whether the intended goals are actually achieved but also to understand and be able to act on the challenges the new practices create for the team, as well as to catch and address negative impacts of the adoption of the method.

The above can best be achieved when maintenance and evolution of practices are part of everyday software development, both on a team level and – if applicable – on a company level. The empirical research cited above indicates that this is already the case in many software development teams. Such practices should be encouraged and appreciated by management. Introducing it on the project level can take the form of regular reflective meetings similar to those the retrospectives recommended for Scrum [49]. On a company level, many big software providers have teams that work with methods, processes and tool support and advise the software teams. The role of these teams, however, should be both to encourage individual teams to adapt centrally decided methods and to gather feedback from the local development in order to improve central methods, processes and tools.

7.3 Implications for teaching

When teaching software engineering, we need to teach methods as practice patterns. This means that, parallel to design patterns, the presentation of methods needs to explicitly address the purpose of the method. There is a need to discuss not only how notations, principles and guidelines, but also how tools are relevant to achieve this goal. There is also a need to discuss the implications of the usage of this method; e.g., many formal mathematical methods implicitly take a waterfall process for granted. Finally, advantages and disadvantages of applying the method proposed need to be discussed.

Many of us have already tried to implement this kind of reflective teaching. It is, however, not the easiest way for students to learn. They need to acquire the skills to implement the central disciplines, adapt the textbook method(s) to a case and reflect on the choices that are part of this process. The effort might though be worth it: in this way, we provide our students – and future practitioners – with knowledge that not only allows them to apply the method, but also to adjust and develop their practices to address software development for innovative products, thereby changing social and technical contexts.

8 Conclusions

The article set out to answer the question ‘What does it mean to use a method?’. The answer can be summarized as follows: To anchor the discussion of software development as a social practice in a sound philosophical argumentation, I have used Schatzki’s concept of integrated practice to describe software development as shared social practices based on common understandings, rules and teleoaffective structures.

Based on Knorr Cetina’s concept, it was argued that *software development is an epistemic practice*, one that unfolds its object and its own practice as the team proceeds in the development. This foundation then was used to define *methods as practice patterns, explicitly formulated sets of (tool supported) understandings, rules and teleoaffective structures that need to be integrated into existing practices*.

Referring once again to Wittgenstein’s notion of rule following, the relation between explicit formulations of methods and the heterogeneity of method deployment is further explored. To better understand these processes, Strauss’ concepts of articulation work and meta-work have been used. Articulation and *meta-work are normal parts of software development practices that also comprise the integration of new methods and tools*.

Following Knorr Cetina, I argued that the unfolding of practice that is motivated by the unfolding of the epistemic object is part of the creative activity leading to a new piece of software. As much as the resulting practice depends on the previously existing practice, on the goal of the development and the anticipated contribution of the method, it also depends on the descriptions of understandings, rules and teleoaffective structures of which the method consists.

Throughout the argumentation, empirical research was cited to illustrate the theoretical concepts and to ensure that the developed concepts would allow making sense of observed practices. Moreover, based on the theoretical foundation, the empirical findings no longer appear to be idiosyncratic behaviour, but rather as examples of unfolding of epistemic practices guided by the goal to bring about a piece of software.

Looking back to the beginning, the article does not take sides on the argument whether the seemingly lacking implementation of methods is due to the methods or due to the (incompetence of the) practice [10]; rather, the article explains the observed heterogeneity and the differences between ‘formalized methods’ and ‘methods-in-action’ [54]. How to address the divergence between methods and practice depends on whether the method to be deployed is helping to bring about, and unfold, the understanding and implementation of the software to be developed. In other words, it depends on how practitioners, based on their previous experience and existing practice, are able to make sense of the method at hand, and it also depends on whether the method to be integrated actually fits with the situated contingencies of the practice it should support.

Whereas the first two of these potential causes can be addressed with better education – both about the intention and rational of the method and about the usage of the notations, guidelines and tools – the latter points to the need to better understand the software development practices and their rationalities in order to devise better methods.

In line with these results, it is argued that there is a need to integrate method development and empirical research of industrial software development practices. With respect to industrial practice, I recommend organisational acknowledgment and support of reflective practices similar to retrospectives in agile development geared towards maintaining and evolving practices. Further, it is argued that methods should

be taught as practice patterns, including explicit discussions of the rationalities, conditions and implications of applying a method.

The intention of the article, however, is not to define software engineering as practice and method as practice patterns once and for all. It is rather intended as an initiation of a discussion of software engineering methods and theory [52], one that includes a theoretical conceptualization of the social side of software engineering.

Acknowledgements

Thanks to the PhD students who challenged me to explicate my understanding of practices and how methods influence practices. Further, thanks to all the colleagues who have discussed Wittgenstein and the practice concepts with me.

References

[1] V.R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, M. V. Zelkowitz, The empirical investigation of perspective-based reading. *Empirical Software Engineering*, *1*(1996), 133-164.

[2] C. Johansson, P. Hall, M. Coquard, “Talk to paula and peter—They are experienced.” The experience engine in a nutshell. In *Learning Software Organizations.* Springer Berlin Heidelberg 2000, pp. 171-185.

[3] M. Jørgensen, B. Boehm, S. Rifkin. Software Development Effort Estimation: Formal Models or Expert Judgment?. *IEEE Softw.* 26 (2009) 14-19. DOI=10.1109/MS.2009.47

[4] P. Mi, W. Scacchi, Modelling Articulation Work in Software Engineering Processes. Proceeding of the 1st International Conference on the Software Process 1991, pp. 188– 201.

[5] G. Button, W. Sharrock, Occasioned Practices in the Work of Software Engineers. In M. Jirotka and J. Goguen (eds.): Requirements Engineering: Social and Technical Issues. London: Academic Press 1994, pp. 217–240.

[6] D. Martin, J. Rooksby, M. Rouncefield, I. Sommerville, 'Good' Organisational Reasons for 'Bad' Software Testing: An Ethnographic Study of Testing in a Small Software Company. *ICSE 2007. 29th International Conference on Software Engineering, IEEE*, pp. 602-611.

[7] K. Rönkkö, B. Kilander, M. Hellman, Y. Dittrich, Personas is not Applicable: Local Remedies Interpreted in a Wider Context. Proceedings of the Participatory Design Conference PDC, Toronto, Juli 27 – 31, ACM 2004, pp. 112 – 120.

[8] Scrum-But (http://agileatlas.org/articles/item/fractional-scrum-or-scrum-but)

[9] D.L. Parnas, B. Curtis, Point/counterpoint. *Software, IEEE*, *26*(2009), 56-59.

[10] K. Rönkkö, O. Lindeberg, Y. Dittrich, Bad Practice or Bad Methods: Are Software Engineering and Ethnographic Discourses Incompatible?. Proceedings of The International Symposium on Empirical Software Engineering, 3–4 October 2002, Nara, Japan, pp. 204–210.

[11] T. Päivärinta, K. Smolander, Theorizing about Software Development Practices, *Sci. Comput. Program.* (2015), http://dx.doi.org/10.1016/j.scico.2014.11.012

[12] T.R. Schatzki, *Social practices: A Wittgensteinian approach to human activity and the social*. Cambridge University Press 1996.

[13] T. R. Schatzki, K. Knorr Cetina, E. von Savigny (eds.), *The practice turn in contemporary theory*. London: Routledge 2001.

[14] K. Knorr Cetina, Objectual practice. In: *The practice turn in contemporary theory,* T. Schatzki, K. Knorr Cetina, and E. von Savigny (eds.), London: Routledge 2001, pp. 175-188.

[15] K. Schmidt, The Concept of 'Practice': What's the Point?. In *COOP 2014-Proceedings of the 11th International Conference on the Design of Cooperative Systems, 27-30 May 2014, Nice (France),* Springer International Publishing, pp. 427-444.

[16] I. Kant, On the Common Saying: 'This May be True in Theory, but it does not Apply in Practice'. In: I. Kant, *Political Writings*. Cambridge University Press 1970, 1991, pp. 61-92.

[17] W. Stegmüller, Hauptströmungen der Gegenwartsphilosophie. Vol.1. Stuttgart 1978.

[18] L. Wittgenstein, Philosophical Investigations. The German text, with an English translation by G.E.M. Anscombe, P.M.S. Hacker and Joachim Schulte. Revised 4th edition by P.M.S. Hacker and Joachim Schulte. Wileay Blackwell 2009.

[19] D. Nicolini, Practice theory, work, and organization: an introduction. Oxford University Press, Oxford 2012.

[20] A.L. Strauss, Work and the division of labor. Sociol Q, 26(1985) 1–19

[21] E.M. Gerson, S.L. Star, Analyzing due process in the workplace. ACM Trans Office Inf Syst, 4(1986) 257–270

[22] K. Schmidt, C. Simone, Coordination mechanisms: Towards a conceptual foundation of cscw systems design, Computer Supported Cooperative Work (CSCW) 5 (1996) 155–200.

[23] J. Nubiola, Scholarship on the relations between Ludwig Wittgenstein and Charles S. Peirce. *Proceedings of the III Symposium on History of Logic 1996*. Berlin: Walter de Gruyter GmbH and Co.

[24] T.J. Barnes, American pragmatism: towards a geographical introduction. *Geoforum*, *39*(2008) 1542-1554.

[25] H. Garfinkel, Studies in Ethnomethodology. Englewood Cliffs, NJ: Prentice-Hall, 1967.

[26] K. Rönkkö, Y. Dittrich, D. Randall, When plans do not work out: How plans are used in software development projects. *Computer Supported Cooperative Work (CSCW)*, *14*(2005), 433-468.

[27] Dreyfus H. (1991). Being-in-the-World: a commentary on Heidegger's Being and Time, Division I. Cambridge, Mass., MIT Press, 1991.

[28] H. Sharp, H. Robinson, M. Petre, The role of physical artefacts in agile software development: Two complementary perspectives. *Interacting with Computers*, *21*(2009) 108-116.

[29] W. Scacchi, Understanding the requirements for developing open source software systems. In *Software, IEE Proceedings-,* 149 (2002) 24-39.

[30] http://www.merriam-webster.com/dictionary/method

[31] IEEE Computer Society, 2004. Guide to the Software Engineering Body of Knowledge (SWEBOK). Version 3, http://www.computer.org/portal/web/swebok/home. Last accessed November 3rd, 2013.

[32] B. Henderson-Sellers, C. Gonzalez-Perez, *Standardizing methodology metamodelling and notation: an ISO exemplar.* R. Kaschek, C. Kop, C. Steinberger, G. Fliedl (eds.), In *Proceedings of UNISCON 2008*, , Springer, Berlin/Heidelberg, pp. 1-12.

[33] L. Mathiassen, System Development and System Development Method. PhD thesis, Oslo University 1981. (In Danish)

[34] C. Floyd, A comparative evaluation of system development methods. In: T.W. Olle, H.G. Sol, A.A. Verrijn-Stuart (eds.), *Proc. of the IFIP WG 8.1 working conference on Information systems design methodologies: improving the practice*, 1979, pp. 19-54.

[35] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education 1994.

[36] C. Alexander, *The timeless way of building* (Vol. 1). Oxford University Press 1979.

[37] L. Mathiassen, A. Munk-Madsen, P.A. Nielsen, J. Stage, Method Engineering: Who's the Customer? In *Method Engineering.* Springer US 1996 pp. 232-245.

[38] C. Floyd, Software Development as Reality Construction, in: C. Floyd, H. Züllighoven, R. Budde, R. Keil-Slawik, eds., Software Development and Reality Construction, (Springer Verlag, Berlin, 1992).

[39] K. Rönkkö, Making Methods Work in Software Engineering: Method Deployment – as a social Achievement, Blekinge Institute of Technology, School of Engineering, Doctoral Dissertation Series No. 2005:04.

[40] D. Damian, L. Izquierdo, J. Singer, I. Kwan, Awareness in the wild: Why communication breakdowns occur. In International Conference on Global Software Engineering (ICGSE) 2007.

[41] S. Draxler, G. Stevens, A. Boden, Keeping the development environment up to date-A Study of the Situated Practices of Appropriating the Eclipse IDE. IEEE Transaction on Software Engineering, 40 (2014) 1061-1074

[42] R. Giuffrida, Y. Dittrich, How social software supports cooperative practices in a globally distributed software project. Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering, 2014, pp. 24-31.

[43] R. Giuffrida, Y. Dittrich, A conceptual framework to study the role of communication through social software for coordination in globally-

distributed software teams. *Information and Software Technology*, 63 (2015) 10-30.

[44] A. Sigfridsson, The purposeful adaptation of practice: an empirical study of distributed software development. PhD Thesis. University of Limerick 2010.

[45] W. W. Royce, Managing the Development of Large Software Systems In: *Proceedings of IEEE WESCON, 1970*, pp. 1-9.

[46] B.W. Boehm, A Spiral Model of Software Development and Enhancement, IEEE Computer, 21 (1980) 61–72.

[47] M. Lehman, Programs, Life Cycles, and Laws of Software Evolution, Proceedings of the IEEE, 68 (1980) 1060–1076.

[48] The Agile manifesto: Agile Alliance web-site http://www.agilealliance.com

[49] H. Unphon, Y. Dittrich, Architecture Awareness in Industrial Software Development, The Journal of Systems and Software 83 (2010) 2211–2226.

[50] Y. Dittrich, Software engineering beyond the project–Sustaining software ecosystems. Information and Software Technology, 56 (2014) 1436-1456.

[51] K. Schwaber, Agile Project Management with SCRUM. Microsoft Press 2004.

[52] Software Engineering Method and Theory, SEMAT, http://semat.org/.

[53] D.L. Parnas, P.C. Clements, A rational design process: How and why to fake it. *Software Engineering, IEEE Transactions on*, 1986 (2), 251-257.

[54] B. Fitzgerald, Formalized systems development methodologies: a critical perspective. *Information systems journal* 6.1 (1996): 3-23.

[55] B. Fitzgerald, An empirical investigation into the adoption of systems development methodologies, *Information and Management*, 34, pp. 317-328, 1998.

[56] B. Fitzgerald, N.L. Russo, E. Stolterman, *Information systems development: methods in action*. McGraw-Hill Education 2002.

[57] E. M. Gerson, Reach, bracket, and the limits of rationalized coordination: Some challenges for CSCW. In M. S. Ackerman & C. A. Halverson & T. Erickson & W. A. Kellogg (Eds.), Resources, Co-Evolution and Artifacts. London, UK: Springer 2008, pp.193-220.

[58] L. Plonka, H. Sharp, J. Van der Linden, Y. Dittrich, Knowledge transfer in pair programming: An in-depth analysis. *International Journal of Human-Computer Studies*, *73(2015)*, 66-78.

[59] M. H. Agar, *The professional stranger: An informal introduction to ethnography*. San Diego, CA: Academic Press, 1996.

[60] Y. Dittrich, K. Rönkkö, J. Eriksson, C. Hansson, & O. Lindeberg, Cooperative method development. *Empirical Software Engineering, 13 (2008),* 231-260.

[61] H. Sharp, Y. Dittrich, C. de Souza, The Role of Ethnographic studies in Empirical Software Engineering. Work in Progress.

[62] J. Nørbjerg, & P. Kraft, Software Practice Is Social Practice. In: Y. Dittrich, C. Floyd, R. Klischewski (eds) *Social Thinking-Software Practice,* MIT Press 2002, pp. 205-222.

[63] J. Bosch (ed.) *Continuous Software Engineering*. Springer International Publishing AG, 2014.

[64] G. K. Hanssen, C. F. Alves, J. Bosch (eds.) Special issue on Understanding software ecosystems. *Information and Software Technology*, 56 (2014), 1421 ff.

[65] L. Bass, I. Weber, L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.