Theorizing about software development practices
Tero Päivärinta a,*, Kari Smolander b

a Information Systems, Luleå University of Technology, Luleå, Sweden
b Software Engineering and Information Management, Lappeenranta University of Technology, Lappeenranta, Finland

1. Introduction

In IEEE Software, Johnson, Eksted and Jacobson [34] argue for “the General Theory for Software Engineering”. Especially, they call for theories which should provide predictive and prescriptive support for software engineering, instead of running costly design processes that are plainly based on trial and error. They mention the issue of choosing software development methods in development projects and organizations as an example of significant questions, which should be tackled by such theory. Especially, Johnson et al. state that “many proposed [...] methods, programming languages and requirements specification languages exist, but very few explicit theories explain why or predict that one method or language would be preferable to another under given conditions” [34, p. 94].

While it may be true that we lack a “general theory” of software engineering, we should be more accurate in our argumentation, however. What is this lack that we are talking about? Are we lacking theories about software artifacts or are we lacking theoretical knowledge on the work of software professionals, i.e. software development?

Theories can explain software engineering from many directions. They can explain a purely technical point of view, where the essence of software engineering is seen as a series of more or less formal transformations from a problem to its solution as a working software artifact (e.g. [25]). This approach is artifact-centric: the objective is to understand the development of...
the software artifact through transformations. Human actors and organizational contexts are reduced to requirements and domain models [25,49], which can be modeled formally.

Another approach has been the building of ontologies that give researchers and method-oriented practitioners a vocabulary for theoretical constructs. An example of this approach is the SEMAT kernel [32], which provides a set of concepts, or an ontology, for the essence of software engineering. Instead of creating theories based on scientific observation, SEMAT has chosen the standardization path [52], where the result is based on consensus, earlier experiences, or a best guess among a body of professionals. Another example is the encyclopedic SWEBOK [29,30], which is an attempt to cover all relevant areas of software engineering with commonly agreed knowledge. SWEBOK is not, however, centered on a single theory of the whole process; it describes areas defined by consensus or convention instead, and therefore cannot be considered as a scientific theory.

While mathematics and computer science provide a solid theoretical basis for understanding computational transformations and the artifact, the process and practices of building the artifact, i.e. software development, is less covered by scientific research. In our work we focus on the problem of theorizing about software development practices, which can create knowledge to support research, education and concrete development of practices in software development organizations. To build and especially validate theories about software development practices, we need to observe the work of software professionals in the real world context. This means that we must recognize software development as an organizational activity, where the context, the business environment and the human organization with its conflicts and other imperfections are at least as relevant for the success of software development as the computational transformations of the software artifact. In this view the observed software development work becomes the source of the theory, and the target of the theory is to understand, explain and predict the rationale and practices of software developers and the impact these have on the success of software development. The objective of this paper is to understand this theorizing process conceptually and to build a model that explains the essential concepts for theories of software development practices.

To understand software development and its practices, we must understand not only software but also software developers. In their work, software developers must continuously reflect on their understanding of software artifacts and artifact building, i.e. they, more or less actively and explicitly, use and build local theories of their own and their team’s actions in software development. Recognizing this, we develop our argumentation through the following steps. Firstly, we discuss two different modes of thinking through which we can appreciate development practices: technical rationality vs. reflection-in-action [60,51,46] and sketch briefly the previously established arguments on why efforts of understanding and theorizing about practice should focus on the latter. Secondly, by framing development practices taking place in software organizations (including projects) as “organizational practices”, we sketch previous practice research in organizations, which has profoundly influenced our work. Thirdly, we outline a model that is needed for theorizing about development practices, especially taking into account the intended rationale for the actual realization and resulting impacts of using particular practices. We use this model to evaluate individual studies published in the journal Science of Computer Programming that use real-world observations as the basis of their theorizing. Finally we discuss software development research in the light of our model and note some implications to current software engineering research approaches.

2. Background: practices and software development

In the core of our approach is the practice of software development, which is “concerned with creating descriptions of the purposes of the software, of its problem domain, of its structure and behavior, of the computations to be performed, of the interfaces between the software and its environment and its users […]” [31, p. 34]. Software development involves numerous practices including those of analysis, design, implementation and quality management. A practice means a more and less organized and a situated activity that is conducted recurrently by human agents [54] or, as Bourdieu [9] defines them, practices are “the recognizable patterned actions in which both individuals and groups engage. They are not a mechanical reaction to rules, norms or models, but a strategic, yet regulated improvisation responding to the dialectical relationship between a specific situation in a field and habitus” ([61, p. 204]; referring to [9, p. 67]). This kind of organic view to practices implies that in the software development context, practices may include both thoroughly organized use of predefined development methodologies and loosely organized and even emergent activities that may use individual tools or techniques at hand.

2.1. Technical rationality vs. reflection-in-action

The field of software engineering has over decades produced numerous methodologies that are normative models of how software development should take place [33]. However, already in the 1980s, [47] noted that a methodology does not really describe what actually happens in practice. The concrete actions in software development, the working practice in use, what is actually done in a situation may be quite different from what is described by a prescriptive methodology. In the field of information systems development, Larsen et al. [41] describe how a research stream has discussed the contextual nature of method/practice engineering and adoption by individual system and software developers, projects, and development organizations since the 1990s (see e.g. [40,63,18,20,19,45,4,66,55]). Recently, this phenomenon has gained increasing attention in the international software engineering community as well. For example, a best paper candidate in the International Conference on Software Engineering in 2013 [57] concluded that Unified Modeling Language, despite of its
status as the “de facto standard” of software modeling, was not at all universally adopted, and it was practically always “used selectively and often informally” [57, p. 731]. This distinction between the reflective actions of practitioners and normative instructions has its background on Schön’s seminal book on the reflective practitioner [60], where Schön identifies two different modes of thinking: technical rationality and reflection-in-action.

Technical rationality refers to instrumental problem solving via application of scientific theory and technique. In technical rationality, knowledge is a result of science and it must be separated from practice. According to Schön [60, p. 26], “researchers are supposed to provide the basic and applied science from which to derive techniques for diagnosing and solving the problems of practice. Practitioners are supposed to furnish researchers with problems for study and with tests of the utility of research results. The researcher’s role is distinct from, and usually considered superior to, the role of the practitioner”. Technical rationality therefore sees practice as fundamentally different from research. Researchers provide theory that practitioners can use and apply, and practitioners supply researchers with real-world problems to solve [51]. In technical rationality, the skills of practitioners are ambiguous and secondary to knowledge, because real knowledge lies in the theories of basic and applied science. Technical rationality also supposes that situations in practice are known beforehand: they fall into categories defined by scientific research.

The other mode of thinking, reflection-in-action, sees situations as unique, complex, uncertain and value-conflictual. As Schön puts it, “our knowing is in our action” [60, p. 49], which is also recognized in modern knowledge management research (e.g. [11,64]) that considers knowledge as inseparable from action. This conception includes the idea that practitioners conduct their practices through similar previous experiences and build a repertoire of skills and outcomes. They also refine their techniques to become masters and highly specialized in their actions. Skills are not secondary to knowledge, but their equal or even superior role follows from knowledge being inseparable from action. This also implies that practice is not separable from research: theorizing needs contribution of practice and the tasks in practice include also research. Researchers do not have exclusive rights to research and theories: also practitioners need to reflect their actions and create their own theories to succeed in their actions. Table 1 describes the most significant differences between these two modes of thinking [51].

Research on software engineering methodologies and techniques has traditionally adopted mostly the mode of technical rationality and focused on documenting prescribed procedures, techniques, tools and notations for software development [35,1]. The purpose of research is then to describe the essential phenomena to practitioners, so that they can take the right theories, tools, and techniques into use to succeed in software development. However, by adhering to a research tradition on actual development practices in systems and software development organizations (summarized e.g. in [41]), we believe that practices in a software development organization do not always follow technical rationality. Instead, they are contextual and evolve over time when professionals reflect in action [41]. In the next subsection we will briefly look at how organizational research has approached practices. We think that the software engineering theory could benefit and receive important ingredients from existing organizational research and its view on organizational practices.

### 2.2. Practice research in organizations

We assume that a software development practice may become an organizational practice, which can be defined as the organization’s routine use of knowledge. Organizational practices can exist at multiple levels, such as in a software development project or in an organization taking part in several projects or development processes (e.g., software development teams, groups, or companies). Organizational practices often have tacit components embedded partly in individual skills and partly in collaborative social arrangements [64,38,50].

Although the software engineering literature comprises many important studies on software organizations and their practices (for example [42,14,26]), organizational practices have not yet been thoroughly studied and theorized in the software development context. Many essential features of software development practices, such as their forming and evolution, require more attention from research and theory to be fully understood. The general management literature, however, theorizes with many viewpoints on how organizational practices take shape. For example, Szulanski [64] argues that a “best practice” represents organizational knowledge which can be transferred between a source and a recipient unit inside an organization as a replication of an organizational routine. Kostova and Roth [39, p. 216] suggest that an organizational practice “evolves over time under the influence of the organization’s history, people, interests, and actions” and that it comes “to reflect the shared knowledge of the organization and tend[s] to be accepted and approved by the organizational members”.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Most significant differences between the two modes of thinking [51, p. 56].</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technical rationality</td>
<td>Reflection-in-action</td>
</tr>
<tr>
<td>Situations (Development contexts)</td>
<td>Fall into scientifically defined categories</td>
</tr>
<tr>
<td>Knowledge (of how to develop software)</td>
<td>Is a result of science and must be separated from practice</td>
</tr>
<tr>
<td>Relationship between research and practice</td>
<td>Practice is fundamentally different from research (science); practice is application of theory and research is production of theory</td>
</tr>
</tbody>
</table>
That is, a practice may be rationally adopted or may emerge in an evolutionary manner in an organization. Within the latter view, any identifiable practice can be phrased to be meaningful to the extent to which it is regarded as useful in a contextual, situated organizational activity [54].

All existing descriptions of a practice imply that at least one stakeholder in the organization has intended that it should be enacted. However, ethnographic studies have shown that such canonical, pre-described practices often deviate from the actual actions taken [10]. Moreover, software development organizations may also follow undocumented, habitual practices. Pentland and Feldman [56] highlight a distinction between the performative and ostensive aspects of organizational routines. The performative aspect represents “the specific actions taken by specific people at specific times when they are engaged in what they think of as an organizational routine” [56, p. 796]. The ostensive aspect is “the abstract or generalized pattern of the routine” [56, p. 796]. The ostensive aspect is not necessarily a formal written description of the practice; it can also be the rationalized response given by a developer when asked how he or she is carrying out his or her work.

In summary, organizational research considers practices as organic and consequential to the organization itself [17]. The development and adoption of practices thus require reflection and understanding of their rationales and impacts, including the initially implicit ones. As Schulze and Boland Jr. [61] note, there is often a difference between what people say they do, what they ought to do, and what they actually do. In addition, the performative aspects of practices, the effects of practices, what doing the practice does, including all its implicit effects, are often hidden from the participants and observers at first. An inquiry into organizational practices requires a reflection-in-action mode to knowledge. Our view is that studies that reflect and theorize software development practices in the real world context, in the field, can provide important knowledge for improving the practice. This kind of studies and theorizing should be situated in actual software development contexts, similarly as in the general practice research [22], and they should lead to reflective learning that can be generalized in relation to clearly expressed theories. Following this line of thought we have developed a model for theorizing about practices for software and systems development, presented in the following section.

3. Coat Hanger model for theorizing about practices

We have created a model for a special purpose, for theorizing about software development practices. We also recognize that there are general principles and models for empirical research and experimentation in software engineering (such as [67,59,37]). These sources define how empirical research in software engineering should be conducted and reported, whereas we want to concentrate on theorizing only. Our purpose is not to replace these but to provide a structured way to investigate and create theories about software development practices or sets of practices with a reflection-in-action mode to research and practice. Our model builds on six main concepts that need to be distinguished in order to learn from software development practice and to build theories of development practices: learning, a practice, development context, rationale, impact, and theory. We use dual naming for each concept, for example rationale/rationalizing and impact/evaluating (Fig. 1). Practice theory [17,8] sees situated actions as consequential in producing organizational structures. This means that actions themselves create and reproduce those organizational structures that they are constituents of. Therefore the double naming – the concepts are not only objects or entities in organizational structures but also actions or activities that reproduce the organization and themselves.

This section contains first a brief definition of these concepts separately, based on previous literature. At the level of separate concepts, there is nothing new in each particular concept per se. However, our model as a whole ties these previously rather separately discussed concepts together, which we argue to facilitate the analysis and theory-building of software development practices.
Lyytinen and Robey [44] relate learning, based on Argyris and Schön [2], to the idea of “theories-in-use”. To learn from practice requires that we identify or assume causal relationships between actions taken during the development and its desired outcomes [44]. Learning from a particular set of development actions requires that we treat development projects and actions as “experiments” from which we generate evidence to test selected theories-in-use with regard to selected ideas of development practices [44]. Our further construction of the Coat Hanger model is based on this general-level idea, while it suggests a few additional fundamental concepts which need to be clarified further in order to reflect theories-in-use to the more generally suggested development practices.

A central concept in our model is the concept of a practice. One dictionary definition of a practice is “something people do regularly” [13]. In the context of a development project or an organization, a systems development practice may become an organizational practice or routine, which can be defined as the organization’s routine use of knowledge, especially “know-how” [38]. The concept of “best practices” illustrates an assumption that abstractions of such know-how can be usefully analyzed, and lessons learned from practice can be transferred between organizational contexts and over time (e.g. [64]). However, organizational practices often have tacit components embedded partly in individual skills and partly in collaborative social arrangements [64,38,50]. If we compare a development method and a practice, a method adopted in an organization always embodies a predefined practice or a set of practices, whereas a practice is not always defined at the detailed level, at least with regard to all potential elements [65] of method knowledge [41]. However, while being cautious on the concept of “best practices”, we share the belief [20] that practice descriptions and definitions may appear as useful for analyzing recurrent development actions in context, as a basis of learning from them.

A software development effort takes place in a development context, which includes a large number of issues and factors [48,12,53]. For example, Clarke and O’Connor [12] identified 140 situated factors from previous literature that may affect the software development process. Similarly, in the field of information systems development, Orlikowski [53] claims that the role of the system, systems development structure and operations, development policies and practices, development staff, corporate strategies, organizational structure and culture, customers, competitors, and available technologies represent contextual categories of issues which may influence the success or suitability of development practices.

Rossi et al. [58] discuss the concept of method rationale as an important part of evolutionary method engineering to support software and systems development. While they regard good understanding of the method rationale as necessary for continuing the “modification and augmentation” of an organization’s methods, we will widen the concept of rationale to be equally useful for understanding the reasons for an organization’s development practices in general (i.e. also those practices in use, which do not necessarily fulfill the characteristics of a thorough method). A rationale for a development practice thus provides justifications for the creation, use and modification of the practice (or a set of practices).

Lyytinen and Robey [44] emphasize the importance of learning from the organization’s own development experience. This evidently requires analysis and identification of the impacts of the development practices to software engineering success (including such dimensions as project schedule, cost, scope, software properties, time, market performance, and other success expectations held by the stakeholders). Such impacts may be desired already according to the explicit method rationale(s), or they may be unexpected, sometimes even unwanted.

Finally, the above concepts are needed for creating and evaluating theories of development practices. That is, we pursue theories which can analyze, describe, and explain contextual software development practices, ultimately aiming at a level of prediction [23]. We believe that it is useful to analyze the practice and aim at predictive theories of certain types of development practices, with regard to their impacts on the development products, projects and processes, and contexts.

In the following, we relate these concepts to each other to form a model to guide the research on development practices. Fig. 1 relates these concepts together, whereas their relationships, which we need to understand in order to build theory from practices, are discussed further below.

Learning is a boundary-spanning mechanism which needs to exist, on one hand, in a development context so that previous theories, including previous, more or less well-grounded, methodological recommendations of development can inform local rationale for new practices and that the observed impacts of the target organization’s previous practices can inform further local rationality to adjust the practices. On the other hand, learning is needed between development organizations and the theory builders, who observe development actions (and local interpretations of such actions) in practice and try to abstract lessons to be learned from the particular practices in question (Fig. 1). The process of theorizing is circular by nature. The local and global theories of practices will affect the rationalizing of practices and practice change in the development context. This in turn will lead to new learning that will create new or extended local or global theories.

The development context involves all the issues which have impact on how practices in the target organization or project are socially constructed and how the development organization can learn from its practices. The context may have an impact on the rationale to implement new practices and how the implemented practices are twisted during actual development activities, on the impacts reached from the practices-in-use, and even whether a learning process takes place in a context at all and whether lessons learned from previous experience are made explicit. That is, practices, their impacts, and learning may not be purely based on the identified rationale alone, but can be affected by contextual issues (Fig. 1). If contextual issues are explicitly identified before implementing a new set of practices, they may, in turn, become an explicitly recognized part of the rationale. However, some contextual issues may have a more implicit effect on practices in use and their impacts, recognized only after new practices have been tried out.

Learning is the intermediary that enables accumulation of local and global theories and their utilization in the development context. The circularity of the model means that learning from local and contextual development practices follows
Table 2
Selection of articles.

<table>
<thead>
<tr>
<th>Journal:</th>
<th>Science of Computer Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Years:</td>
<td>2010–2013</td>
</tr>
<tr>
<td>Total article count:</td>
<td>371</td>
</tr>
<tr>
<td>Inclusion criteria:</td>
<td>1. The study gathers knowledge from professional software development contexts/practitioners and includes some form of direct observation.</td>
</tr>
<tr>
<td></td>
<td>2. The study focuses on the work of software practitioners.</td>
</tr>
<tr>
<td>Examples of excluded article classes:</td>
<td>1. Pure formal or technical studies that observe or describe software artifacts only.</td>
</tr>
<tr>
<td></td>
<td>2. Artificial experiments in laboratories or with students.</td>
</tr>
<tr>
<td></td>
<td>3. Pure metrics studies using open source repositories, version management databases, project management databases or similar.</td>
</tr>
<tr>
<td>Articles that meet the criteria:</td>
<td>4 (13%):</td>
</tr>
<tr>
<td></td>
<td>- Arias et al. [3]: A top-down strategy to reverse architeciting execution views for a large and complex software-intensive system: An experience report.</td>
</tr>
<tr>
<td></td>
<td>- Eklund and Gustavsson [16]: Architecting automotive product lines: Industrial Practice.</td>
</tr>
<tr>
<td></td>
<td>- Ganesan et al. [21]: An analysis of unit tests of a flight software product line.</td>
</tr>
<tr>
<td></td>
<td>- Hadar [24]: When intuition and logic clash: The case of the object-oriented paradigm.</td>
</tr>
</tbody>
</table>

from understanding of how practices are implemented and used in any target context of development. The contextual rationale(s) for particular practices and their improvements may lead to change(s) in a practice or a set of practices, which are often a part of the practice repertoire in the context. Learning from the contextual impact(s) after a practice has been introduced or changed can be used for theorizing about the practice locally or globally (Fig. 1).

If observed changes and improvements in local practices are used to contribute to a theory of software development beyond the context in question, then we also need to recognize ideas of more generic rationales giving reasons to implement certain types of practices, the very ideas and descriptions of those practices and their interrelation, and ideas of impacts realized from adhering to the practices in question. Also generic ideas to categorize development contexts which may have an impact on the rationales, enactment of particular practices, whether they are only espoused but not really in use, and impacts resulting from particular practices, may be theorized. Through learning from the target context(s), development practice research may thus theorize further on more universal issues of the development context, their impact on rationales for practices, actual practice domains of interest, and the generalized ideas of impacts from choosing particular practices (Fig. 1). Here it is important to note that a set of pre-defined practices or methods “in theory”, like our theoretical (and often normative) assumptions of where and why they should be adopted, needs to be distinguished from the contextual practice descriptions of what actually happens. Although the idea may sound self-evident, this distinction has not been always very clear in the traditional studies of software methods and their use – as noted e.g. by Fitzgerald et al. [20]. Our model thus suggests that any research effort on software development practices should discuss its results beyond the context and in light of a more general theory which aims to say something about development contexts, rationales, a set of practices and their impacts beyond the particular development context(s) in question. However, learning in the local context is, of course, also useful for improving the software practices in the organization in question.

We argue that theories of development practices, when constructed according to the Coat Hanger model, help to promote understanding of the reasons why and under which conditions it is possible to apply particular ideallized practices and whether the organizations could predictably expect to realize the positive (and to identify other possible) impacts under given conditions (as requested by Johnson et al. [34]). For researchers of software development practice, we thus argue that the Coat Hanger model can serve as an analytical tool for formulating and discussing clear research hypotheses of software development practices. The model suggests that a meaningful general-level format for a theoretical hypothesis of a (set of) software practice(s) is as follows: Under identifiable contextual conditions (C), software developers have a rationale (R) to choose a set of practices (P) so that the measurable impacts (I) can be expected to be realized (C → R → P → I).

4. Cases

To illustrate the value of the Coat Hanger model we made first a survey of articles in the journal Science of Computer Programming in order to identify the latest practice-focused research on software engineering that has been regarded as interesting in the journal in question. We went through four full years of articles (2010–2013) and selected, with the principle of reflection-in-action, articles that researched real-life professional software development and/or used practitioners as sources of information. In total the journal had published 371 articles during these four years (see Table 2). We included those articles that studied real software development practices or practitioners in real-life organizations, where the researchers were able to directly observe how professional software developers reflect in action. Because we required both real-life software development and software developers as informants, we excluded purely formal or technical studies, laboratory experiments, student experiments, and pure metrics analyses, e.g., from open source databases. After reading the
abstracts of the 371 articles we found seven candidates that could meet the criteria. Of these seven, four articles met all the criteria, i.e. they were executed in professional software development organizations and observed/studied the actual work of professional software developers. Therefore only 1% of the journal articles during 2010–2013 collected first-hand evidence from professional software development organizations and took reflection-in-action into account. In the following we will evaluate these four articles in light of the Coat Hanger model. By necessity, we give rather general-level description of our analysis below, whereas our interpretations can be easily tested and replicated by anyone who has access to the four articles in question.

4.1. Arias et al. [3]

Arias et al. [3] focus on the practice area of reconstructing software architectures for existing systems, of which architectures have not been previously documented. They report experience from the case of Philips magnetic resonance image (MRI) scanner software. At the level of theory/theorizing (see Fig. 1), Arias et al. touch upon most of the issues. The general-level context comprises reverse-engineering and reconstruction of architectures of large-scale and complex software systems. Architecture reconstruction as a relevant field of practice is rationalized with the need to “improve software maintenance”. Arias et al. identify a few previously suggested practices in the literature, such as methods, techniques and tools of architecture reconstruction. However, the expected general-level impacts of and measures for architecture reconstruction remain implicit.

The Philips case convinces the reader that the development context (see Fig. 1) of MRI software in Philips is large and complex. The article assumes implicitly that the contextual rationale to adopt architecture reconstruction practices in Philips coincides with the generic motivation for improved software maintenance. Moreover, the study describes how the top-down strategy and tool support for reverse architecting are usually applied as the espoused practice in the case organization. It remains less clear whether the actual practices of reverse engineering and reconstruction work in the organization would have deviated from the espoused ones.

With regard to measures whether any contextual impacts (Fig. 1) from practicing architecture reconstruction is reached, the paper highlights explicitly that the architecture reconstruction work on the MRI scanner resulted in better software with 30% decreased system startup time. Hence, the measured impact in context focuses on the system startup time, while it remains unclear how the startup time relates to the initial rationale of improved software maintenance. Arias et al. [3] discuss their contextual experiences and lessons learned in light of previous methods, techniques and tools of architecture reconstruction and explain how their “three aspects of practice to support reverse architecting” differ from previous literature.

4.2. Eklund and Gustavsson [16]

Eklund and Gustavsson [16] focus on architects’ work on maintenance of product line architectures in automotive companies, where complex software embedded in electronic control units of vehicles shares characteristics that are claimed to be common to the automotive domain in general. Eklund and Gustavsson motivate their work at the level of theory/theorizing (Fig. 1) by stating that the generic rationale for adopting and using practices for architecture work is to manage change in software when new features or functionalities and product configurations are introduced to the product lines. As a general-level impact, the increased architecture understanding in itself results in a decreased number of architecture-level changes when new changes to software or product configurations are introduced. The authors relate their work to a theory-level model of software architecture design [27] and on a few other observations from previous literature. In the model, architecting tasks are divided into five generic categories of practices concerning need, impact analysis, solution, decision, and validation.

The case study describes the development contexts (see Fig. 1) of two automotive product lines, Volvo and Scania. The rationale for architecture work in the target organizations varies: Volvo focuses on cost reduction whereas Scania adapts the product line architecture to unavoidable hardware changes over time. The study reports actual practices-in-use of software architects. The actual work practices of architects are described under the five generic task categories identified in the theory part. Some pre-defined or commonly espoused practices of architecture work are identified in the organizations. Such practices include, however, a rarely used “method similar to the Pugh evaluation matrix” for architecture analysis in Volvo, network topology descriptions, and UML-based models of logical architectures. The study mentions that the time needed to understand the impact of a change on the architecture is rather similar in the organizations. However, no explicit contextual evaluation of any impacts of the reported practices in relation to the time measure or the theory-level rationale is given. As an anecdote, the informants are mentioned to be generally happy with their own work practice.

The lessons learned in the practice description include the observation that the target organizations actually involve tasks of architecting work related to every category in the generic model. This observation is stated to “confirm the validity”
[16, p. 2358] of the general model [27] of architecture processes. As the main lesson learned, Eklund and Gustavsson (ibid.) propose that a company's core values may influence and shape the architectural decisions. Their observation in these contexts also opposes the previous literature-based claim of architects working mainly alone [28], and highlights the importance of communication and teamwork instead.

4.3. Ganesan et al. [21]

Ganesan et al. [21] focus on unit testing of software product lines. The theory-level (Fig. 1) rationale for unit testing is a previously reported workshop result [5], which states that the sooner the bugs are found by unit testing, the less cost is incurred for bug fixing. Thus, the article gives a general-level theoretical motivation for good software unit testing practices and suggests even rather clear and measurable general-level impacts that could be expected from them. The article recognizes that little previous research has been done to report experience of actually using stubbing frameworks and related practices for this purpose, which motivates this experience report in general.

The case study focuses on more than 10 years of experience gained from unit testing in the development context (see Fig. 1) of the Core Flight Software System (CFS) at the NASA Goddard Space Flight Center. As no deviating contextual rationales for adopting the reported bug fixing practices in the case study are declared, we assume that the cost reduction has been the initial rationale also in this context. The recommendations and 19 espoused practices are reported “as is”, based on the authors’ experience. It remains unclear how the 19 practices have been shaped through actual practices over the ten-year period. The practice description in this paper gives no explicit report of contextual impacts (such as actually realized cost reductions) resulting from the adoption of the 19 suggested practices. Relating the lessons learned back to the previous literature, the paper claims a contribution through clarifying a set of characteristics of architectural design to facilitate unit testing and through suggesting 19 “good practices” for unit testing and architecting.

4.4. Hadar [24]

Hadar [24] focuses on whether and how professional software developers experience problems to follow good object-oriented design practices. In theory (see Fig. 1), the article identifies a few universally recommended object-oriented design practices, such as how to identify and define objects and object classes, how to assign functionality and decompose a problem into objects, and how to deal with inheritance. The rationale for following these practices is to cope with software complexity. Mastering these practices can be regarded as a part of the personal skills and knowledge of individual developers. The article is less explicit about the expected impact or concrete measures for coping better with software complexity.

Issues describing the development context (see Fig. 1) of the participant practitioners include their individual pre-knowledge and length of experience in object-oriented software development. The participants are from six software development companies, which remain unidentified in the study. The paper provides no espoused practices of the practitioners’ own organizational contexts. Instead, actual individual practices of experienced practitioners are studied through a standardized experiment, during which their actual decisions and the results of designing a system specification as a response to standardized development problems are recorded and analyzed. With regard to the espoused practice, the article thus assumes that every developer actually shares the theory-level ideas of the rationale and knowledge of universal object-oriented practices. The actual practice is revealed by observing how developers solved software development problems during the experiment. No contextual impacts resulting from the actual practices are reported, as they are not the specific target of this study. The lessons learned are compared to previous research by observing that even experienced object-oriented programmers experience problems in adhering to the ideal practices, despite the fact that they know them, and suggesting a theoretical explanation for this phenomenon and implications for improving the situation in the future.

4.5. Summary of the analysis

Table 3 summarizes our analysis of the four selected articles and the areas to which we can suggest improvements for reporting of the above-mentioned practice studies and theorizing from them on the basis of the Coat Hanger model.

While the articles were in general rather clearly motivated and the problem areas explicitly related to previous research, the analysis showed that three of the articles remained slightly unclear about which impacts in theory (see Fig. 1) could be expected (and how they could be measured) according to the previous literature if the suggested practices were taken into use. The rationale for a practice alone does not necessarily tell us directly how we can verify whether the rationale has been actually reached, let alone which other impacts may occur after the practice is adopted. In this regard, the theoretically most complete article was Ganesan et al. [21] that explicitly both rationalized the studied practice and expressed a clear statement about which impact could be expected and measured if the suggested practices were to be adopted. Even if the previous literature were vague on the expected impacts and how to measure them, the use of the Coat Hanger model would make the focus on expected impacts and their relations to actual practice descriptions more explicit.

While three of the articles [3,16,21] described the development contexts (see Fig. 1) of their target organizations in great detail, the articles remained less explicit on how the contexts in question would be related to the rationale, espoused/actual practices, and the realized impacts of the practices in use (Table 3). The most developed article in this regard was Eklund and Gustavsson [16], which discussed how organizational values were visible in architecting practices, and how, against their
Table 3
Theorizing and practice description elements in the analyzed articles.

<table>
<thead>
<tr>
<th>Paper</th>
<th>Review of theory</th>
<th>Context description</th>
<th>Contextual rationale</th>
<th>Espoused practices</th>
<th>Actual practices</th>
<th>Contextual impacts</th>
<th>Lessons learned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arias et al. [3]: architecture reconstruction of complex software</td>
<td>Vague on expected impacts.</td>
<td>Shows only that the case context matches with the research problem.</td>
<td>Implicitly assumed to match with the generic one.</td>
<td>Focus on describing the contextually espoused practice as a result of the study.</td>
<td>Not reflected on.</td>
<td>One measure demonstrates software improvement. Unclear alignment to rationale/expected impact.</td>
<td>Aspects of practice.</td>
</tr>
<tr>
<td>Eklund and Gustavsson [16]: architecting work for SW product line maintenance</td>
<td>Generic model/categories of tasks; slightly vague on expected impacts.</td>
<td>Observed org. values related to some of observed practices. (Slightly unclear how many context aspects are otherwise related to the research.)</td>
<td>Given, not linked back explicitly to the general rationale.</td>
<td>Some pre-defined practices identified, few used regularly.</td>
<td>Actual work described in more detail than espoused.</td>
<td>No explicit evaluation, anecdotes unaligned to the rationale/expected impact.</td>
<td>Observation that organizational values shape actual practices.</td>
</tr>
<tr>
<td>Ganesan et al. [21]: unit testing of SW product lines</td>
<td>Clear on rationale and expected impact; few previous practices reported to exist.</td>
<td>Shows only that the case context matches with the research problem.</td>
<td>Implicitly assumed to match with the generic one.</td>
<td>Focus on abstracting contextual espoused practice as a result of the study.</td>
<td>Not reflected on; assumed to match with the espoused.</td>
<td>No contextual evaluation.</td>
<td>19 &quot;good practices&quot;, suggestions for architecture design to enhance unit testing.</td>
</tr>
<tr>
<td>Hadar [24]: object-oriented design practices</td>
<td>Vague on expected impacts of &quot;good practices&quot;.</td>
<td>Experiment participants chosen by the length of their experience and assumed pre-knowledge.</td>
<td>Implicitly assumed to match with the generic one.</td>
<td>Assumed to match with the universal &quot;good practices&quot;.</td>
<td>Focus on actual deviations from the espoused, universal &quot;good practices&quot;.</td>
<td>No contextual evaluation.</td>
<td>Observed phenomenon and a suggested theoretical explanation for it.</td>
</tr>
</tbody>
</table>

The shaded cells represent areas to which improvements in theorizing on the practices and practice descriptions can be suggested in the respective articles.

Initial expectations, some differences between their two case organizations actually did not seem to have significance on how the actual practices had been shaped. However, even Eklund and Gustavsson documented several other observations of their context, which remained rather unconnected to the observations of the actual practices. The report could have benefited from more explicit discussion about which contextual issues in total might have significance for the other elements of the practice description. Hadar [24] experimented with professional developers from six organizations, whereas the paper involved no discussion about the potential differences between the developers’ organizational contexts. To summarize, we argue that a more explicit focus on aligning discussions about contexts to the other elements of the Coat Hanger model could have helped the authors of these reports to come up with more structured and context-aware theorizing.

Three articles assumed without problematizing that the theoretical rationale to adopt the practices in question would correspond to the contextual rationale (Table 3). The only article which mentioned the contextual rationale at all was that of Eklund and Gustavsson [16], while also their paper would have benefited from more explicit alignment of the contextual rationales and theory-level motivation.

Two articles [3,21] focused solely on describing ex post rationalized, espoused practices in their target organizations. The articles thus assumed that the actual work in organizations equaled to the described, espoused, practice. Reflection-in-action to reveal the history of how actual practices have been adopted or emerged would have made the grounds to recommend the resulting abstract practices more explicit and convincing. Eklund and Gustavsson [16] mentioned a couple of pre-defined (espoused) practices in their target organization, but focused mainly on describing the actual practice. Hadar [24] assumed that the espoused practices of object-oriented design are universally shared and studied how the designers deviated from those during their actual design tasks in action.

According to our analysis (Table 3), the articles described and evaluated the actual impacts of the practices-in-use in the development contexts rather minimally or not at all. The only paper that pointed explicitly to the impacts of the ap-
plied practice, was Arias et al. [3]. However, even they did not align their actual evaluation of improved software efficiency (decreased system startup time) explicitly to their initial rationale of improved software maintenance. Eklund and Gustavsson [16] mentioned that the developers were satisfied with their contextual practices which was an additional qualitative measure unaligned with the initial rationale for the practices, i.e., decreased number of architectural changes in connection to new software changes. While Ganesan et al. [21] were most explicit on the expected impacts in their theory part, they reported no contextual evaluation of whether the recommended practices actually resulted in a decreased number of bugs and decreased costs. Hadar [24] did not evaluate the contextual impacts, which is understandable due to the nature of his experimental research.

All the articles were able to conceptualize valuable findings and relate them back to the literature (Table 3). However, the summary of our analysis (Table 3) suggests that three articles remained, in general, vague of some aspects of theory/theorizing and that all the articles could benefit from our model with regard to describing aspects of the development context, contextual rationales and the actual impacts of the analyzed practices (as aligned to the initial rationales) more systematically. Two of the articles involved no analysis on the actually realized practices while focusing plainly on the espoused practices.

5. Discussion

Our analysis of the four practice descriptions above illustrates how the Coat Hanger model can reveal shortcomings in current practice-oriented research and suggests some improvements for theorizing from practice. Although the literature contains writings about theories in software engineering in general (e.g. [62]), as a model for a specific purpose – for theorizing about software development practices – our results have clear values of novelty and utility. Whereas Sjöberg et al. [62] describe the creation and evaluation of theories in general, our interest is more specific in theorizing about software development practices, and our purpose is to make the constituents of such theories explicit. The Coat Hanger model makes these constituents explicit and offers a structured view on theorizing about practices.

The Coat Hanger model can be used for several purposes. When analyzing existing literature, researchers can identify existing theoretical issues of practices more explicitly already during their literature reviews and recognize relationships between contextual factors, contextual rationale, espoused and actually realized work practices, and rigorously evaluated and verified impacts of the realized practices in context. The model can help researchers to discuss the shortcomings and further potential of existing recommendations for software practices in a more structured way. It is clear that the four practice descriptions above focus more on giving plainly descriptive narratives of issues which are not very explicitly related to each other. An important contribution of the Coat Hanger model is its focus on defining and theorizing about relationships between the suggested concept categories (contextual factors, rationale, espoused vs. actual practices and impacts).

Metaphorically, our model is a “hanger” which remains empty without concrete “coats” of well-declared hypotheses of what should result from adopting certain practices and clearly reported practice descriptions in development contexts, lessons learned, and their comparisons back to relevant theory level issues. The model will help to create conceptual clarity and to capture variance on selected contextual issues, rationale and impact that can be related to particular types of software development practices. Researchers interested in issues affecting the implementation of particular types or sets of practices can now be more specific under which contextual conditions and rationales particular practices can be expected to produce the desired (and undesired) impacts with relevant measures. While the rationale to adopt a certain practice may seem “self-evident” or common-sense at the first glance, previous research has shown that the contextual rationales to adopt a practice may vary from the initial rationale suggested e.g. by researchers, method engineers, or the “common wisdom” of the field (cf. [55,7]). Moreover, the Coat Hanger model suggests detailed understanding and description of when and why actual uses of practices may deviate from the espoused practices in context. As a stream of recent research on systems and software practices (summarized in [41]) shows, organizations, teams and even individuals often deviate from the espoused practices. If the Coat Hanger model is followed to report on and discuss practice, it would suggest more explicit descriptions about the role of espoused practice descriptions in relation to the actual work practices, enabling continuous learning and reflection-in-action from the practice. The model also provides a conceptual basis to observe and to form theories of how variance in selected elements of the model would impact development performance and resulting software systems, and align the impact analyses explicitly to the initial rationales, without forgetting to observe the emerging impacts. Hence, our work responds to the continuing calls for better empirical theorizing and education of software and systems development practice to increase professionalism in the field (e.g. [68,36,34]). Another use of the Coat Hanger model for researchers would be to use it as an analytical tool for understanding and integrating theorizing of particular areas of development practices. The use of the Coat Hanger model as guidance to plan for and report research on software practices would thus improve theorizing about software development.

While we consider the Coat Hanger model as such as the main contribution of this study, the survey we made on the journal Science of Computer Programming produced a rather surprising result. Only 1% of the papers in the journal during the last four years both gathered knowledge directly from professional software development contexts or practitioners and focused on the work of software practitioners. Our preliminary additional observations on other journals suggest that the situation is somewhat similar elsewhere as well. Some journals publish more practice-oriented papers, but in most cases computation, artifact transformations and controlled experiments get more attention among software engineering researchers than the work of software professionals in real-life contexts. This is an indication of the prevalence of technical
rationality in software engineering research, as opposed to reflection-in-action, and it might at least partially explain the lack of theory of software engineering [34] in the professional work context. Theories in software engineering tend to center around phenomena in the artificial world that are in a way certain mathematically or statistically. Computation, artifact transformations and controlled experiments produce knowledge that is indisputable, but only in the context of the particular artifact or the particular experiment. The capability of this knowledge to explain the actions of software professionals in real-life contexts may be limited, and therefore such theory base alone may be seen as inadequate. Current theories can explain computation and transformations robustly, but they have trouble in explaining the problems and issues met in real software development organizations theoretically. This situation is very analogous to the extensive “rigor vs. relevance” discussion in the field of information systems (see e.g. [6,15,43]), which was very active 15 years ago.

Software development seems to be rarely considered as an organizational activity in software engineering research. Other organizational research is seldom referenced, while it could provide important theoretical contribution to the field of software engineering as well. Our interpretation is that most software engineering research represents largely the technical rationality [60]. It presupposes that there can be universal, context-independent methods, tools, theories, and best practices that can be applied to clearly defined situations. It makes a relatively clear distinction between research and practice and premises that research can produce universal solutions to software development practice. Much of software development research is more or less related to human action and human organizations. In the sense of reflection-in-action, the informants for scientific inquiry are practitioners who also must build their own theories-in-use for their own actions in software development. This kind of scientific inquiry requires more attention to be paid on appropriate research methodologies and epistemologies. Therefore the reference sciences of software engineering research include not only computer science and mathematics but also organization sciences, psychology, management and sociology, and although software development has its own idiosyncrasies, much of the methodology and epistemology may come from these reference sciences as well as from computer science and mathematics.

6. Conclusion

We have discussed the need in software engineering research to observe the work of software professionals in real environments and presented a model that includes essential concepts for the theorizing of software development practices. The Coat Hanger model and its concepts – learning, a practice (espoused and actual), development context, rationale, impact, and theory – can be used as a tool by research planners and evaluators. To illustrate the value of the model we also made a survey of articles in Science of Computer Programming and went through four full years of articles (2010–2013). Of 371 articles, we found only four articles that studied and observed directly professional software development using practitioners as informants. Our analysis in light of the model could reveal areas of improvement with regard to the theoretical and empirical parts of the four studies. Finally we pointed out that software engineering research tends to value technical rationality instead of reflection-in-action and emphasized that theorizing of software development practices requires novel approaches that recognize also other reference sciences than computer science and mathematics and pays more attention to scientific methodology. In the future we aim to continue our theory building on software development practices. When combined with our previous model on practice changes [41], the coat hanger model is expected to serve as a conceptually clear and useful instrument for such theorizing. One alternative for continuing the study could be to build a catalogue of well-known or otherwise interesting software development practices, their rationales, and impacts on known contexts.

References


Please cite this article in press as: T. Päivärinta, K. Smolander, Theorizing about software development practices, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2014.11.012