Information and Software Technology 70 (2016) 176–180

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Introduction to the special section—General Theories of Software Engineering: New advances and implications for research

Klaas-Jan Stol [a,*], Michael Goedicke [b], Ivar Jacobson [c]

[a] Lero—the Irish Software Research Centre, University of Limerick, Ireland
[b] University of Duisburg-Essen, Essen, Germany
[c] Ivar Jacobson International, Verbier, Switzerland

ARTICLE INFO

Article history:
Received 17 July 2015
Revised 31 July 2015
Accepted 31 July 2015
Available online 17 August 2015

Keywords:
General Theory of Software Engineering
Special section
Editorial

ABSTRACT

In recent years, software engineering researchers have recognized the importance of the role of theory or SE research, resulting in the emergence of the General Theories of Software Engineering (GTSE) community. This editorial introduces a special section that contains four articles, and reflects on the advances made by the contributing authors.

We discuss the different approaches taken in each of the four papers and outline a number of avenues for future research.

1. Introduction

In the last decade or so, the software engineering research community has increasingly started to pay attention to the topic of theory in software engineering [1–5]. In addition to several workshops on the theme of General Theories of Software Engineering (GTSE), a first special issue was published in the journal *Science of Computer Programming* [6]. Given the momentum of the emerging community around this theme, we published a call for papers, and this special section is the result.

Many other disciplines have general theories—for example, physics has the Standard Model of particle physics [7]. General theories are useful for several reasons, and one important reason in particular is that it helps to identify important questions and as such helps to set out a research agenda for a discipline as a whole. A recent example of this is a long-time missing component of the Standard Model in physics. The Standard Model suggested the existence of a specific type of particle (a *boson*). By 2013, physicists announced that they believed they had found the Higgs boson. Thus, the Standard Model provided an overall framework that suggested to researchers what to look for.

In software engineering, such an overall framework is missing. The SEMAT (Software Engineering Methods and Theory) initiative, founded in 2009 by Ivar Jacobson, Bertrand Meyer and Richard Soley, has argued that software engineering needs to identify a common ground. To that end, the SEMAT initiative has defined the 'Essence' language and kernel [4] which has been accepted as an OMG standard [8].

Most studies in software engineering pay little or no attention to theory development, and very few studies are based on existing theories, although exceptions do exist [9]. The explanation for this may lie in the tradition of how software engineering studies have been conducted thus far. Software engineering studies can be roughly organized into two categories. The first category is what we call *solution-seeking* studies. These studies observe a certain technical problem and 'engineer' a solution that addresses the problem. Wieringa would call these 'practical problems' [10]. In most cases such engineering studies also contain an experimental, quantitative evaluation to demonstrate how well the formulated solution addresses the problem.

The second category is what we call *knowledge-seeking* studies. These are studies that investigate software engineering practice by studying, for example, what software professionals do, what their challenges are, and what processes they use, addressing questions such as "how are things done" and "what's going on here." Wieringa would call these 'knowledge problems' [10]. This type of study has become more common over the last decade, and researchers conducting this type of study have adopted a variety of research methods from other disciplines most notably from the social sciences, such as case studies, surveys, grounded theory and ethnography. The use of qualitative data is quite common in knowledge-seeking studies

Solution-seeking studies tend to focus on very specific and detailed software engineering problems. Often, solutions are composed to analyze or change a system's source code. In such studies, the

* Corresponding author.
E-mail address: klaas-jan.stol@lero.ie (K.-J. Stol).

http://dx.doi.org/10.1016/j.infsof.2015.07.010
0950-5849/

'theory' tends to be in the form of a hypothesis that the proposed solution works better than existing solutions. Exemplar constructs in such studies are program size (which can be measured as lines of code or object code size) and performance. We call such 'theories' (the sets of hypotheses put forth around a specific tool or technique) *micro-theories*. While these studies offer direct value in that they provide a solution to a software engineering problem, it is often not immediately clear how they contribute to the larger issues in software engineering.

Knowledge-seeking studies, on the other hand, can be conducted at many different levels of detail. Some studies are case studies to investigate a new phenomenon. A classic example of this is the study by Mockus et al. investigating open source software development [11]. Based on the results of that study, the authors proposed a set of hypotheses that they suggest could explain how open source software development works 'in general,' and form the basis for a *middle-range theory*. Such middle-range theories are very useful as they facilitate the integration and linking of several studies with one another and construct a body of knowledge on software engineering phenomena.

Despite an active community that publishes hundreds of research papers every year, many researchers in our field agree that our research is not making a significant impact on industry. Perhaps we are not asking the right questions. Software engineering researchers are studying a wide variety of topics, and the boundaries of software engineering as a discipline are still expanding. Partly this is due to the fact that new trends are continuously emerging that are relevant for software practitioners—for example, the use of social media in software engineering practice [12]. However, the 'big picture' of software engineering research remains unclear—a General Theory of Software Engineering is missing. A GTSE is needed to position all those micro and middle-range theories.

The goal of this special section, as well as the workshop series on this theme organized by other members of the GTSE community, is to draw attention to this issue, to explore community members' ideas, and to encourage others to think about how their research could benefit from a theory-oriented approach to software engineering research. The scope of this special section was not limited to *general* theories. Instead, we welcomed middle-range theories, evaluations of theories, and proposals for how to use theories from other disciplines to explain software engineering phenomena.

2. The articles in this special section

Following the call for papers on the theme of General Theories of Software Engineering, we received 11 submissions. Of those, one was desk-rejected as it did not fall within the scope of the original call. The remaining ten articles were each reviewed by two reviewers as well as by the guest editors. Of these ten, four articles were accepted for publication.

In their article "The Tarpit – A General Theory of Software Engineering," Pontus Johnson and Mathias Ekstedt propose a general theory of software engineering. Johnson and Ekstedt developed this theory (the 'Tarpit') based on their argument that communication breakdowns are at the heart of the challenges in software engineering. The Tarpit is based on four theoretical fields that are of central importance to software engineering: languages and automata, cognitive architecture, problem solving, and organizational structure. These four different fields also reflect the socio-technical nature of the software engineering field. To illustrate the utility of the Tarpit as a theory, Johnson and Ekstedt demonstrate how it can be used to explain and predict three well-known phenomena in software engineering: Brooks's Law (a principle), domain-specific languages (an artifact), and continuous integration (a practice). The Tarpit theory can be seen as a common framework that offers explanations and allows predictions for a variety of phenomena. One current limitation of the Tarpit theory, as acknowledged by Johnson and Ekstedt is that its presentation is qualitative and not formalized. We believe that the Tarpit theory can be further explored in a number of ways. As the authors suggest, further work may focus on formalization, such as the definition of an explicit set of propositions. Another venue is the use of the Tarpit theory as a framework for integrating an existing body of literature in a particular area, for example, coordination in global software development. By doing so, the Tarpit can be used as 'theoretical glue' to integrate an existing body of empirical research.

The second article, "A Theory of Distances in Software Development" by Elizabeth Bjarnason, Kari Smolander, Emelie Engström and Per Runeson also presents a theory. In contrast with the Tarpit theory by Johnson and Ekstedt which is based on existing theoretical constructs, the Theory of Distances was inductively developed and grounded in empirical data. The Theory of Distances is based on an empirically-based model, which the authors named the "Gap model," that consists of three parts. The first part of the Gap Model is the definition of eight different types of *distances*. These include the well-known geographical and temporal distances, but new types of distances are psychological and cognitive distances which affect an individual's perceptions, communication skills and competence levels. The second part is the definition of eight so-called *alignment practices* which help to link requirement engineering on the one hand and testing on the other hand. One such alignment practice is *cross-role collaboration*, which involves roles from different disciplines in software engineering activities; for example, testers who participate in the reviewing of requirements documents. The third part of the Gap Model provides the link between the former two parts and explains how alignment practices help to reduce the various types of distances. Effectively, this third part in which Bjarnason and colleagues outline how the various alignment practices affect distances is a set of implicit propositions. The Gap Model is based on empirical findings, and offers practical insights that can be of immediate use to software professionals. At the same time, we also believe that the Gap Model invites further studies that empirically test the various implicit propositions. To do so, these propositions should be instantiated as hypotheses through the operationalization of the various constructs, i.e., the various types of distances and alignment practices. For example, geographical distance is not sufficiently operationalized as *longitudinal* geographical distance will be affected differently than *latitudinal* distance. In the former, time zone differences will play a role, whereas in the latter no time differences are present.

The third article, "What does it mean to use method? Towards a Practice Theory for Software Engineering" by Yvonne Dittrich presents a conceptual foundation for understanding software development as a social practice. In particular, Dittrich aims to develop an understanding of why the use of software development methods varies by project. The issue addressed here is that each organization, project, or team adopts methods (or practices) in their own specific way that fits within a specific context. Earlier researchers named this 'method-in-action' [13]. Following an in-depth philosophical argumentation that draws from several insights from other disciplines, Dittrich outlines a number of very important implications for research, practice and education. Dittrich argues that methods emerge in one of two ways: either as abstracted practice patterns to communicate to colleagues, or as output of software engineering research. The impact of the latter is very small. In both cases, empirical research is concerned with evaluating those methods and techniques, but also with understanding the context in which these methods and techniques are used. As each software development endeavor takes place in a unique context with specific challenges and constraints, the methods used may or may not work as expected. Furthermore, Dittrich also argues that the tailoring and adoption of methods needs to be carefully deliberated and that the suitability of methods should be evaluated after adoption so as to ensure that their intended goals are achieved.