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*

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, 2, 3, 4, 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 '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? To-

wards 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.

The fourth and final article in this special section is by Paul Ralph, entitled "Software Engineering Process Theory: A Multi-Method Comparison of Sensemaking-Coevolution-Implementation Theory and Function-Behavior-Structure Theory." This article does not propose a new theory, but instead presents a comparison of two software engineering process theories. Whereas the Tarpit theory (by Johnson and Ekstedt) and the Theory of Distances (Bjarnason et al.) are *variance* theories, Ralph discusses and compares two *process* theories. As Ralph points out, the former tend to focus on *why* events occur, whereas the latter tend to focus on *how* events occur. The two theories that Ralph compares are the Sensemaking-Coevolution-Implementation (SCI) theory on the one hand, and the Function-Behavior-Structure (FBS) theory on the other hand. The SCI was developed by Ralph himself [14], whereas the FBS theory was developed by Gero [15]. Ralph employed a multi-methodological approach, using a multiple case study and a questionnaire study to compare the two theories. The results suggest that SCI better explains how developers create software than the rival theory FBS. Ralph emphasizes that his study does not *prove* SCI or *falsify* FBS, arguing that verificationism and Karl Popper's falsification are defunct epistemologies. Furthermore, he also points out a number of implications of these results. For example, the article argues that problem framing and design are tightly-coupled activities in practice, and therefore separating them as conceptually separate activities (as is done in the waterfall model) can be misleading.

3. But what is it good for?

The four articles included in this special section differ in level of abstraction (concrete vs. philosophical), research approach (empirical vs. theoretical), and scope (general theory vs. middle range theory). Consequently, readers may find some articles more accessible than others due to the different styles of presentation. However, together these four articles offer a number of useful contributions which can be classified into three dimensions: the substantive dimension, representing the phenomenon of interest; the conceptual dimension, representing conceptualizations and theoretical contributions; and the methodological dimension, representing contributions in terms of research approaches [5]. Table 1 summarizes these domains and provides examples.

3.1. Advances in the substantive domain

Each of the four articles is positioned in a certain topic of interest, which is the real-world phenomenon that a researcher may be interested in. This is the *substantive* area of interest. Given the nature of this special section with a specific focus on theory of software engineering, the substantive contributions are limited in all four articles.

Johnson and Ekstedt offer their view on what they believe software engineering is about and define what they consider to be the core concepts in our discipline. In their own words, the Tarpit theory revolves around *"the communicative difficulty between the architecture of human cognition and the architecture of computing systems."*

The substantive topic in the article by Bjarnason and colleagues is in the area of requirements engineering and testing. Their article offers an empirically-grounded model that emerged from data gathered through five case studies. The substantive contribution of their article is an understanding of the relationships between so-called alignment practices and a variety of distances that they identified. Consequently, given the strong empirical foundation and the insights that are derived from their analysis, the substantive contribution of this article is considerable and offers sound advice to practitioners.

The substantive topic of the article by Dittrich is software development methods. Dittrich focuses on the question what it means to use a methods in practice as it is tailored to the context within which the method is applied. Dittrich illustrates her argumentation with various examples from the literature.

The substantive element in Ralph's study is that of software development as an activity. The contribution in this study is similar to those of Johnson and Ekstedt, and Dittrich, namely, in the description of software development as an activity.

As mentioned, the substantive contribution is limited in all four articles – however, this should not be considered a limitation of these studies. The substantive element in these studies is the 'background' or area within which the authors have positioned their main contribution, which lies in the conceptual domain. We discuss these contributions next.

3.2. Advances in the conceptual domain

The emphasis of the contributions of the articles in this special section lies in the conceptual domain, which is the domain of theories, analytical frameworks, and new lenses to look through when considering topics of study [5]. Table 2 lists the theoretical contributions of the four articles.

Table 1. Three domains of elements of research studies.

Domain	Description	Example
Substantive	Software engineering phenomena or systems. These are the objects of study that a researcher is interested in.	Open Source software development, Linux, crowdsourcing, software architecture, distributed software development, developer motivation
Conceptual	Constructs, relations, frameworks, theories to describe, compare or explain phenomena.	Analytical and comparative frameworks, micro theories, middle-range theories, hypotheses, propositions, concepts, abstractions, mathematical models, Lehman's Laws
Methodological	Methods or techniques used to gather data	Case study, survey, experiment, ethnography, repertory grid technique, comparative analysis, instruments, techniques, content analysis, MOOD metrics

Johnson and Ekstedt's article contributes the Tarpit theory, which is a general theory of software engineering. The Tarpit theory is a *variance* theory as opposed to a *process* theory. The Tarpit theory defines the relationships between the key concepts that Johnson and Ekstedt identified. In their article, Johnson and Ekstedt define three major inhibitors in software development: (1) making informed design decisions, (2) correctly translating between languages (specification, implementation), and (3) coordination. The Tarpit theory, then, is presented as a theory that can explain and predict a variety of software engineering phenomena.

The article by Bjarnason et al. offers a theory that can explain why certain practices support alignment and coordination of software development projects. Their Theory of Distances is based on what they refer to as the Gap Model, which is an empirically grounded model of how a number of alignment practices can help to overcome 'distance' in software engineering, which they defined as a multi-faceted concept.

Dittrich offers a new conceptual lens, derived from concepts in the philosophy of sociology, and through which she explains the heterogeneity in the outcome of using software development methods. Earlier empirical research had long established that software development methods are almost never adopted as-defined, but virtually always in an *a la carte* fashion [16]. The variety with which practitioners adopt methods has important implications for comparing different projects that claim to be using a certain method, as differences in project success cannot easily be attributed to the use of a method as they are implemented in different ways. However, thus far very few researchers have tried to provide an understanding of this phenomenon. Dittrich's article aims to develop such an understanding.

Finally, rather than contributing a *novel* theory, the theoretical element in Ralph's study are two rival theories, the Sensemaking-Coevolution-Implementation theory and Function-Behavior-Structure theory. The FBS theory was not developed specifically for the software engineering field, and had not been evaluated for the software engineering domain. Ralph's article presents the first empirical evaluation of the SCI theory.

3.3. Advances in the methodological domain

Besides drawing attention to the potential benefits of a theory-oriented approach [5], one of our goals in this special section is to demonstrate how activities such as theory development and comparison could be done. Table 2 summarizes the methodological approaches taken in the four articles.

As is the case for all research studies, the studies included in this special section have limitations. The research approaches taken vary and therefore so do the limitations. Readers may be left unconvinced as to the results of the studies or may disagree with the theories that are proposed. However, this does not diminish the insights that we may glean from the methodological approaches which the authors have employed, and we believe the various approaches used in these four articles can be used as exemplary 'models' or templates.

Johnson and Ekstedt take a theoretical approach, and base their Tarpit theory on a set of constructs that they identify in four different theoretical fields. To demonstrate the utility of their theory, they presented three test cases. A different approach to theory development is taken by Bjarnason et al. who based their theory on a number of empirical case studies. From the analysis of the empirical data they developed the "Gap Model." Their Theory of Distances was based on this empirically-grounded model. Dittrich follows a theoretical approach by presenting a philosophical argumentation that is based on the philosophy of sociology. Throughout her article she draws on empirical results published in the literature to illustrate her arguments. Finally, Ralph presents an example of how one could empirically compare two different process theories.

4. The Future of Theory-Oriented Software Engineering

The four articles in this special section offer a variety of different approaches to theory development and evaluation. We hope these articles inspire others to consider how their future studies can benefit and contribute to a theory-oriented software engineering. Besides the theories proposed in these articles, several other theories or theoretical frameworks have been published in the first special issue on the GTSE theme [17, 18, 19]. Together, these proposed theories offer various opportunities to explore how the extant software engineering literature can be integrated, for example:

- Development of novel native theories for software engineering that define a distinguishable theoretical core for the field and integrate the numerous micro theories that have resulted from both solution-seeking and knowledge-seeking studies;

Table 2. Overview of the articles in this special section

Article	Scope	Approach	Contribution
Johnson, Ekstedt: The Tarpit – A general theory of software engineering	Software engineering	Theoretical: theory development based on four theoretical fields.	A general theory of software engineering: The Tarpit theory
Bjarnason, Smolander, Engström, Runeson: A theory of distances in software engineering	Requirements engineering and testing	Empirical: multiple case study, followed by theory development.	A theory of alignment of requirements engineering and testing: the Theory of Distances, and the Gap Model that underpins it.
Dittrich: What does it mean to use a method? Towards a practice theory for software engineering	Software development methods	Theoretical: conceptualization of the use of software development methods, grounded in the philosophy of sociology.	A conceptual base for understanding software development as social and epistemic practices.
Ralph: Software engineering process theory: A multi-method comparison of SCI theory and FBS theory.	Software development and design	Empirical: survey and multiple case study to evaluate and compare two theories.	Empirical comparison of two process theories of software design.

- Evaluate the theories through further empirical studies to gauge whether they capture the key concerns of software engineering;

- Systematic literature reviews that use these theories as a foundation for synthesis. Rather than grouping studies based on topic (most studies that are called 'systematic reviews' are, in fact, mapping studies), theories can be used to position studies in relation to one another.

- Comparison of these theories, to evaluate which of them fits best to explain software engineering phenomena. As is the case for the articles in this special section, theories may aim at capturing software engineering discipline as a whole (e.g. the Tarpit theory), or a specific area (e.g. the Theory of Distances).

- Re-evaluate research on software development methods and tools using new conceptual lenses, such as that provided in Dittrich's article in this special section. Software Engineering researchers continuously develop new methods and tools, but such proposals often ignore the context within which these methods and tools will be used. In many cases, methods and tools are not used as initially intended by the method or tool designer.

We believe a further investigation of how the various theories and theory development approaches can contribute to theory-oriented software engineering research is warranted.

Acknowledgments

We wish to thank all authors who have submitted to the special section. We are also grateful to all anonymous reviewers who have contributed their time and expertise, without whom this special section would not have been possible. Furthermore, we are grateful to Prof. Claes Wohlin, the Editor-in-Chief of *Information and Software Technology*, for providing assistance and guidance throughout the process of this special issue. This work was supported in part by Science Foundation Ireland grant 13/RC/2094 to Lero—the Irish Software Research Centre.

References

[1] P. Johnson, M. Ekstedt, In search of a unified theory of software engineering, in: International Conference on Software Engineering Advances, IEEE Computer Society, 2007.
[2] D. I. K. Sjøberg, T. Dybå, B. C. D. Anda, J. E. Hannay, Building theories in software engineering, in: F. Shull, J. Singer, D. I. K. Sjøberg (Eds.), Guide to Advanced Empirical Software Engineering, Springer Verlag, 2008, pp. 312–336.
[3] P. Johnson, P. Ralph, M. Goedicke, P.-W. Ng, K. J. Stol, K. Smolander, I. Exman, D. E. Perry, Report on the second SEMAT workshop on general theory of software engineering (gtse 2013), ACM SIGSOFT Software Engineering Notes 38 (5) (2013) 47–50.
[4] I. Jacobson, N. Pan-Wei, P. E. McMahon, I. Spence, S. Lidman, The Essence of Software Engineering, Addison-Wesley, 2013.
[5] K. Stol, B. Fitzgerald, Theory-oriented software engineering, Science of Computer Programming 101 (2015) 79–98.
[6] P. Johnson, M. Ekstedt, M. Goedicke, I. Jacobson, Editorial: Towards general theories of software engineering, Science of Computer Programming 101 (2015) 1–5.
[7] D. Griffiths, Introduction to Elementary Particles, Wiley & Sons, 2008.
[8] Object Management Group, Kernel and language for software engineering methods (essence), available at: `http://www.omg.org/spec/Essence/1.0/PDF/` (2014).
[9] T. Hall, N. Baddoo, S. Beecham, H. Robinson, H. Sharp, A systematic review of theory use in studies investigating the motivations of software engineers, ACM Trans Softw Eng Methodol 18 (3).
[10] R. Wieringa, Design science as nested problem solving, in: Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology (DESRIST'09), ACM, 2009.
[11] A. Mockus, R. Fielding, J. Herbsleb, A case study of open source software development: The apache server, in: Proceedings of the 22nd international conference on Software engineering, 2000, pp. 263–272.
[12] M.-A. Storey, L. Singer, B. Cleary, F. Filho, A. Zagalsky, The (r) evolution of social media in software engineering, in: Proceedings of the Future of Software Engineering (FOSE), 2014, pp. 100–116.
[13] B. Fitzgerald, N. Russo, E. Stolterman, Information systems development: methods in action, McGraw-Hill Education, 2002.
[14] P. Ralph, The sensemaking-coevolution-implementation theory of software design, Science of Computer Programming 101 (2015) 21–41.
[15] J. S. Gero, U. Kannengiesser, The situated function-behaviour-structure framework, Design Studies 25 (2004) 373–391.
[16] B. Fitzgerald, An empirical investigation into the adoption of systems development methodologies, Information and Management 34 (6).
[17] C. Erbas, B. C. Erbas, Modules and transactions: Building blocks for a theory of software engineering, Science of Computer Programming 101 (2015) 6–20.
[18] T. Päivärinta, K. Smolander, Theorizing about software development practices, Science of Computer Programming 101 (2015) 124–135.
[19] A.-J. Stoica, K. Pelckmans, W. Rowe, System components of a general theory of software engineering, Science of Computer Programming 101 (2015) 42–65.