Proceedings

The Semat Workshop on a General Theory of Software Engineering 2012

November 8-9, 2012
KTH Royal Institute of Technology
Stockholm, Sweden

Ivar Jacobson
Michael Goedicke
Pontus Johnson

Program GTSE 2012

Thursday, November 8

09:00 – 10:00 Joint GTSE-PFSE introduction
Ivar Jacobson, Michael Goedicke, Arne Berre, Pontus Johnson

10:00 – 10:30 General theories: an exposé
Pontus Johnson

Coffee

11:00 – 12:00 Aspects of theories I
Per Runeson, Theory Building Attempts in Software Engineering
Tero Päivärinta and Kari Smolander, A Framework for Building Theories from Software Development Practice
Nada Bajnaid, Algirdas Pakštas and Shahram Salekzamankhani, Ontology-Based Modeling of the Software Quality Assurance Knowledge
Harlod Lawson, Software Engineering in the System Context

Lunch

13:15 – 14:00 Aspects of theories II
Jürgen Börstler, The Importance of Empirical Software Engineering
Howell Jordan and Rem Collier, Measuring Quality: A Cornerstone of Theory in Software Engineering
Pontus Johnson and Iaakov Exman, Requirements on theories of software engineering

14:00 – 15:00 Discussion: What are the objectives of a theory of software engineering? How can a theory be of use in practice?

Coffee

15:30 – 16:30 Discussion: What questions should a theory of software engineering answer?

16:30 – 17:00 Invited presentation
Capers Jones, Software Excellence

18:30 – Dinner

Friday, November 9

08:30 – 09:00 Invited presentation
Dines Bjørner, The Triptych Method and its Relations to SEMAT

09:00 – 09:30 Theory proposals I
Paul Ralph, Sensemaking-Coevolution-Implementation Theory
Ilia Bider, Knowledge Transformation in Software Development Processes

Coffee

10.00 – 10:45 Theory proposals II
Hannu-Matti Järvinen and Mikko Tiusanen, States and Transformations for Software Engineering Theory
Iaakov Exman, Linear Software Models
Ekkart Kindler, On the dimensions of software documents – An idea for framing the SE process

10:45 - 12:00 Discussion: What should the main elements of a general theory of software engineering be?

Lunch

13:15 - 14:30 Discussion: What are the most important qualities of a general theory of software engineering?

Coffee

15:00 – 15:30 GTSE Summary

15:30 - 16:00 Joint GTSE-PFSE summary

Sensemaking-Coevolution-Implementation Theory

A Model of the Software Engineering Process in Practice

Paul Ralph
Department of Management Science
Lancaster University
Lancaster, UK
paul@paulralph.name

***Abstract*—Sensemaking-Coevolution-Implementation Theory is a teleological process theory of the practice of designing complex software systems. It posits that an independent agent (design team) creates a software system by alternating between its three titular activities. Its veracity has been demonstrated using questionnaire and case-study methods. It has been used to evaluate software engineering curricula and highlight deficiencies in software engineering methods and practices.**

Keywords-SCI Theory; process theory; design; coevolution

I. SCI THEORY

Theories of the software engineering (SE) process have historically been dominated by stage-gate or lifecycle models, beginning with the Waterfall Model [1]. This was followed by a "methodology era", during which SE was usually conceptualized through a methods lens, and a "post-methodology era" where methods continued to dominate conceptualization of SE despite their decreasing relevance to practice [2]. These lifecycle models and the methods based on them are fundamentally misleading due to their empirically debunked assumptions [3], [4].

Sensemaking-Coevolution-Implementation Theory (SCI) was developed as an alternative to lifecycle models of SE [5]. It is based on Alexander's model of the "selfconscious" design process [6], reflection-in-action [7], and theorizing of coevolution by [8] among others. SCI (Figure 1, Table 1) posits that where a complex software system is developed by an independent, goal-oriented agent, that agent will engage in three basic processes – Sensemaking, Coevolution and Implementation – in a self-directed sequence.

The agent may be an individual or team. The arrows in Figure 1 indicate relationships between concepts and activities, not sequence – the agent may transition between activities in any order. In a typical project, Sensemaking may include interviewing stakeholders, writing notes, organizing notes, reading about the domain, reading about technologies that could be used in project, sharing insights among team members and acceptance testing (getting feedback from stakeholders on prototypes). Implementation may include coding, managing the codebase, writing documentation, automated testing, creating unit tests, running unit tests and debugging.

While Coevolution does not directly map to a variety of well-known software engineering activities, it is observable in real projects. For example, when a team stands around a whiteboard drawing informal models and discussing how to proceed, they often oscillate between ideas about the design object (e.g., 'how should we distribute features between the partner channel screen and the partner program screen?') and the context (e.g., 'you know what, I think channels and programs are just different names for the same thing.'). This mutual exploration of context and design object is Coevolution. Coevolution may occur in planning meetings and design meetings, following breakdowns or during an individual's internal reflection.

Evolution and coevolution are easily confused. In design literature, evolution, specifically evolutionary prototyping, denotes the gradual improvement of a software object. In contrast, coevolution refers to "developing and refining together both the formulation of a problem and ideas for a solution, with constant iteration of analysis, synthesis and evaluation processes between the … problem space and solution space" {Dorst:2001tq, p. 434}. SCI therefore distinguishes between two types of iteration – coevolution denotes simultaneously revising ideas of problem and solution within minutes or hours, while evolution denotes improving software artifacts over weeks and months.

SCI is a teleological process theory, intended to explain how software is developed in practice. Van de Ven [9] distinguishes two types of theories – variance theories explain the causes of consequences of something and often specify the relative contribution of multiple antecedents, while process theories explain *how and why* an entity changes and develops. Process theories come in at least four types [10]: lifecycle theories posit that an entity progresses through a series of stages in a predefined sequence; evolutionary theories posit a population of entities that changes as less fit entities expire and remaining entities change and recombine; dialectic theories posit that changes result from shifts in power among conflicting entities; teleological theories posit an agent who purposefully selects and takes actions to achieve a goal. SCI therefore takes a teleological approach to causality: software artifacts change as human beings (having free will) choose to change them. This differs from the probabilistic approach to causality adopted by many variance theories.

A survey [11] of over 1300 software development professionals found that SCI better described their processes than either Waterfall or an alternative SE process theory, the Function-Behavior-Structure Framework (FBS) [12]. Emerging evidence from an ethnographic study of an English software development team also supports SCI's core claims and the impossibility of understanding conventional SE through Waterfall or FBS. SCI has been used to analyze SE curricula [13]. It can also be used to analyze design

methods and practices, and to teach SE and project management.

REFERENCES

[1] W. Royce, "Managing the development of large software systems," presented at the Proceedings of WESCON, the Western Electronic Show and Convention, Los Angeles, USA, 1970.

[2] D. Avison and G. Fitzgerald, "Where Now for Development Methodologies," *Communications of the ACM*, vol. 46, no. 1, pp. 79–82, 2003.

[3] F. P. Brooks, *The Design of Design: Essays from a Computer Scientist*. Addison-Wesley Professional, 2010.

[4] P. Ralph, "Introducing an Empirical Model of Design," in Proceedings of The 6th Mediterranean Conference on Information Systems, Limassol, Cyprus, 2011.

[5] P. Ralph, "The Sensemaking-Coevolution-Implementation Theory of Software Design," *MIS Quarterly,* under review.

[6] C. W. Alexander, *Notes on the synthesis of form*. Harvard University Press, 1964.

[7] D. A. Schön, *The reflective practitioner: how professionals think in action*. USA: Basic Books, 1983.

[8] N. Cross, "Research in Design Thinking," in *Research in design thinking*, N. Cross, K. Dorst, and N. Roozenburg, Eds. Delft, Netherlands: Delft University Press, 1992.

[9] A. H. Van de Ven, *Engaged scholarship: a guide for organizational and social research*. Oxford, UK: Oxford University Press, 2007.

[10] A. H. Van de Ven and M. S. Poole, "Explaining development and change in organizations," *The Academy of Management Review*, vol. 20, no. 3, pp. 510–540, Jul. 1995.

[11] P. Ralph, "Comparing Two Software Design Process Theories," in Proceedings of the Fifth International Design Science Research in Information Systems and Technology Conference, St. Gallen, Switzerland, 2010, vol. 6105, pp. 139–153.

[12] J. S. Gero and U. Kannengiesser, "An ontological model of emergent design in software engineering," presented at the 16th International Conference on Engineering Design, Paris, France, 2007.

[13] P. Ralph, "Improving coverage of design in information systems education," in Proceedings of the 2012 International Conference on Information Systems, Orlando, FL, USA, 2012.

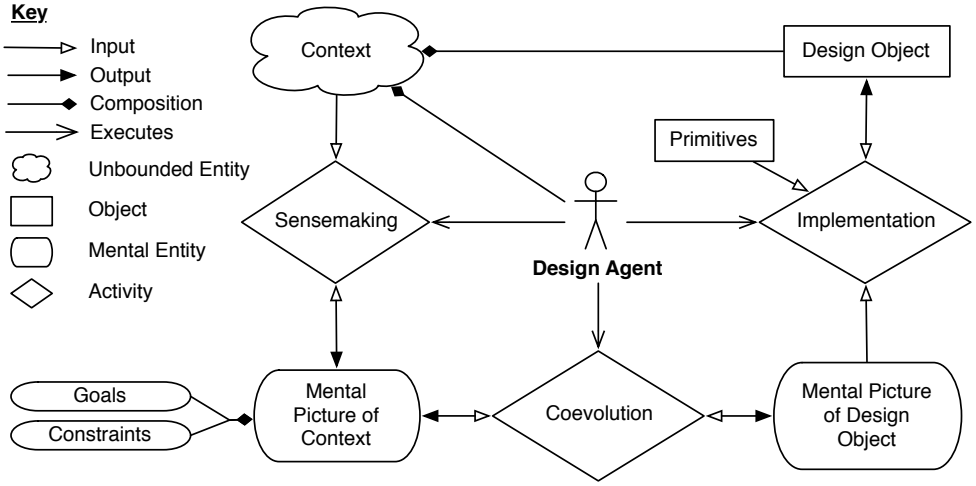

Figure 1. Example of a TWO-COLUMN figure caption: (a) this is the format for referencing parts of a figure.

Concept / Activity	Meaning
Constraints	the set of restrictions on the design object's properties
Design Agent	an entity or group of entities capable of forming intentions and goals and taking actions to achieve those goals and that specifies the structural properties of the design object
Context	the totality of the surroundings of the design object and agent, including the object's intended domain of deployment
Design Object	the thing being designed
Goals	optative statements about the effects the design object should have on its environment
Mental Picture of Context	the collection of all of the design agent's beliefs about its and the design object's environments
Mental Picture of Design Object	the collection of all of the design agent's beliefs about the design object
Primitives	the set of entities from which the design object may be composed
Sensemaking	the process where the design agent organizes and assigns meaning to its perception of the context, creating and refining the mental picture of context
Coevolution	the process where the design agent simultaneously refines its mental picture of the design object, based on its mental picture of context, and the inverse
Implementation	the process where the design agent generates or updates the design object using its mental picture of the design object

TABLE I. CONCEPTS AND RELATIONSHIPS OF SCI THEORY, DEFINED

Measuring Quality: A Cornerstone of Theory in Software Engineering

Howell Jordan
Lero @ University College Dublin
Ireland
howell.jordan@lero.ie

Rem Collier
University College Dublin
Ireland
rem.collier@ucd.ie

***Abstract*—In any engineering domain, a detailed understanding of what constitutes a 'good' product is vital for the development of theories that are both general and useful. However, software engineering researchers' understanding of desirable product qualities is not yet fully mature, especially for continuously-evolving software systems. Inspired by two historical examples, this paper calls for a discipline-wide effort to precisely define the attributes and variables of software product quality in a measurable way. We expect this effort will lead to two major contributions. Firstly, the defined attributes and variables should act as units in any general theory of software engineering. Secondly, once instruments to measure these attributes and variables are developed, systematic large-scale empirical studies of software product quality will become much easier, eventually yielding a rich corpus of data which should prove fertile for further theory building.**

***Keywords*-Software quality; Software measurement**

I. Introduction

It seems certain that the scope of any general theory of software engineering must include not only software developers and development methods, but also the software product itself. The task of building a general theory of software engineering then consists of identifying relationships among the attributes and variables - or *units* - of these objects [1]. This paper aims to spark a discussion about which product attributes and variables should be included in such a theory, and how they should be measured. Since the utility of a theory is determined by the hypotheses that can be derived from it, we begin by asking: what predictions should a universal theory of software engineering be capable of making?

II. Prediction and Software Evolution

In mature engineering disciplines, theories from the mathematical and physical sciences are used extensively to ensure that a proposed product will meet the customer's requirements. Similarly, many software engineers assert that the correctness of programs should be assured through mathematical proof [2], or more generally that "To build something good ... you have to predict in the design stage the qualities of the end product" [3]. In this section we will argue that a broad definition of quality is required, if theory is to help developers achieve long-term customer satisfaction.

A limitation of the proof-of-correctness view is that a definitive 'end product' does not always exist. Many software systems continue to evolve long after deployment, often leading to variants and enhancements far in excess of their original scope. Therefore it is crucial that software is not only correct, but also easy to modify.

Despite advances in requirements elicitation and specification techniques, it is likely that software evolution will become increasingly common for three reasons. Firstly, customers don't always know what they want. A customer - afflicted by the I'll Know It When I See It (IKIWISI) syndrome - may be unable to provide a system's requirements until a prototype has been delivered, and this is especially common for graphical user interfaces [4]. Secondly, for any product operated by humans there is no such thing as a perfect design; "the problems of multiple users or changing fashion or new aesthetics will always be lying in wait" [5, ch.3]. Even for software with no human user interfaces, as interconnection between systems grows, updates to keep pace with external changes become routine [4]. Finally, future variant systems, if any, are likely to be accommodated in software where possible, due to the relatively malleable nature of software as an engineering medium. For example, "most car manufacturers now offer engines with different characteristics ... frequently these engines ... differ only in the software of the car engine controller" [6]. In summary, a practical unifying theory of software engineering should be capable of predicting all qualities that are important to a long-lived evolving family of software products.

III. Defining Product Quality

Software product quality is often modelled as a hierarchy of attributes [7]. While some attributes such as reliability are well defined, there is no overall agreement on the content or structure of this hierarchy; in particular, many attributes relating to the nonfunctional properties of software and its evolution are poorly understood. We argue that a major research effort is required to clearly define them.

From a scientific perspective, it is crucial that such definitions are not predicated on specific software technologies. For example, several authors have proposed object-oriented coupling and cohesion (e.g. [8]) as measures of maintainability, and this view is supported by empirical evidence

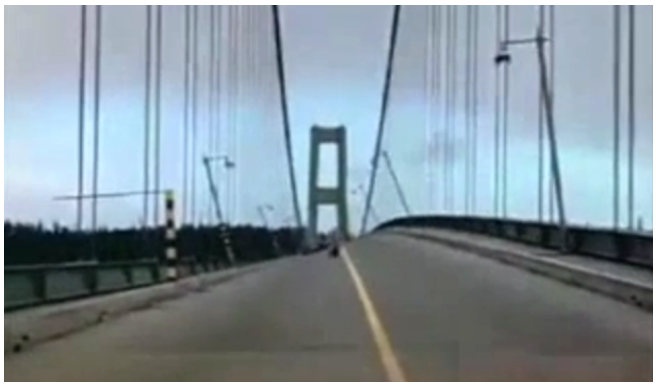

Figure 1. Empirical engineering in action: the first Tacoma Narrows bridge, oscillating torsionally just prior to its collapse. A simple optical instrument to measure the vertical displacement of the bridge deck can be seen on the left of the picture. The blurred figure near the centre line of the roadway is almost certainly that of Professor Burt Farquharson.

in certain contexts (e.g. [9]). However, these definitions are rooted in the object-oriented paradigm, so they cannot be used to compare, for example, object-oriented and functional programming approaches. It follows that any theories built on these definitions cannot predict which method or language would be preferable under given conditions - and these are currently some of the most significant questions in software engineering research [3].

IV. Examples from Engineering History

The history of software engineering is relatively short, so for illustrative examples of product quality definition problems we must turn to other engineering disciplines.

A. Tacoma Narrows Bridge Collapse, 1940

The first Tacoma Narrows bridge, which opened in July 1940 and collapsed five months later, is a well-known example of design failure due to incomplete theoretical knowledge. The collapse was caused by violent torsional oscillations in the bridge deck, induced by aeroelastic fluttering. The design, while ambitious in its scope and daring in its economy of materials, did not apparently violate any contemporary theories of good bridge building [10, ch.9]. However it is surprising to note that, by 1939, flutter in aircraft wings was theoretically quite well understood, and had been widely reported for at least two decades [11]. The Tacoma Narrows disaster can thus be viewed as a collective failure to understand that aeroelasticity is an important attribute of bridge design quality.

An investigation into the disaster was led by Burt Farquharson of the University of Washington, who began his study of the bridge soon after its opening, and was present at the time of its collapse. Figure 1 shows a still image of the bridge on that day, taken from 16mm Kodachrome video footage which was fortunately recorded by the owners of a nearby camera shop. To the left of the picture, an optical gauge constructed by Farquharson's team can be seen; this allowed the bridge's vertical oscillations to be precisely measured and recorded using a film camera positioned on the shore[1]. These observations and measurements were crucial to the disaster investigation, and ultimately led to the integration of aeroelastic flutter into mainstream bridge engineering theory during the 1950s [12]. In the intervening period, the Bronx-Whitestone bridge, which is of similar design, was strengthened against the *symptoms* of aeroelastic fluttering by adding extra material [10, ch.9]; however such quick-fix solutions to quality issues are rarely available in software engineering.

It is likely that the first Tacoma Narrows bridge was not the first bridge to be damaged or destroyed by aeroelastic fluttering [11]; and some authors consider that the 1940 collapse might have been prevented if these earlier incidents had been observed, measured, and studied in more detail [10, ch.9]. Collectively, the history of long-span suspension bridge design illustrates that a rich set of observations and measurements may be an essential prerequisite to successful theory building. The lessons learned from Tacoma Narrows also suggest that software engineering may have much to gain from detailed studies of project failure.

B. Langley Field Aircraft Experiments, 1919-1941

Once recognised, the problem of a poorly-understood product quality attribute can apparently be overcome by a concerted research effort.

Before 1920, the maneuvrability of aircraft could not be predicted at design time, and was mostly a matter of trial and experience. Prototype aircraft sometimes exhibited dangerous handling characteristics, and often these flaws could only be corrected by costly and time-consuming modifications [13, ch.4]. At Langley Field aeronautical laboratory, Virginia, from 1919 to 1923, this problem began to receive serious research attention. A lengthy series of flight tests successfully moved the focus of investigation from qualitative judgements by pilots to quantitative measurements of the control forces required to perform various maneuvres. Key to this progress was the development at Langley of experimental procedures based on new measuring instruments and data recorders, such as the three-axis accelerometer [14] and synchronizing chronometer [15]. These devices relieved pilots from having to pause during and between maneuvres to record measurements manually, and thus allowed a far greater quantity of more accurate product data to be collected [16, ch.3].

By the mid 1930s, the growth of commercial air travel and the problem of pilot fatigue over longer journeys led to

[1] At the instant shown in figure 1, the striped vertical pole and the markers attached to the street lamps behind it are clearly not aligned, due to the extreme twisting motion of the bridge. Clearly, the displacement gauge was not designed to measure torsional oscillations; the bridge's final mode of collapse apparently came as a surprise even to Professor Farquharson.

renewed interest in flying qualities[2]. Under the leadership of Robert Rowe Gilruth, comprehensive flight tests of at least 18 aircraft and ground experiments to discover the forces exerted by pilots on the controls led to the publication in 1941 of the first full flying qualities specifications[3]. These specifications included a beautifully simple measure of maneuvrability - the stick force per g - that is equally applicable to all types of aircraft and is still in use [16, ch.3]. The experiments also yielded the large body of data necessary to build theories of maneuvrability [13, p.32], that today allow flying qualities to be accurately predicted from a given aircraft design.

V. Conclusions

Any holistic view of software product quality should include attributes that are relevant to continuously-evolving systems; however many such attributes are currently only poorly understood. Our first historical example, taken from the pioneering era of long-span suspension bridge design, illustrates the desirability of a complete theoretical understanding of product quality in any engineering endeavour. This theoretical understanding might take many decades to arise, unless a concerted effort is made to identify the phenomena of interest and study them by observation and measurement of product instances.

Our second example, taken from the early years of long-range aircraft design, shows how instrument development can enable large-scale product observations, which in turn may yield concise and useful theoretical knowledge. Vincenti identifies in this example seven phases of product quality research: familiarization with problem; identification of variables, concepts, and criteria; development of instruments and techniques for measurement; growth of opinion regarding desirable qualities; scheme for (empirical) research; measurement of qualities for a cross-section of products; and assessment of results to arrive at general conclusions [16, p.102]. While these phases do not represent a strict ordering, in our example the important theoretical results only began to emerge during the empirical phase. Noting the apparent shortage of widely-accepted theories in software engineering [3], we suggest that a lack of interest in measurements and instruments may be holding back empirical research in our discipline. We hope that the removal of this barrier will eventually lead to significant theoretical progress.

[2]In the period 1923 to 1935, researchers at Langley had mostly been preoccupied with aircraft performance. Walter Vincenti remarks that "control of an airplane is, in a sense, secondary to its speed, range, ceiling, or carrying capacity ... only when the performance gains had been at least in part realized did concentration on problems of stability and control become advantageous" [16, p.78]. It is interesting to consider whether a similar set of implicit research priorities currently exists in software engineering.

[3]Instrument development continued at Langley throughout the 1930s and 1940s, however the details were not published. William Hewitt Phillips, who by 1945 was head of Stability and Control at Langley, recalls that the restriction was imposed "so that industry would have to come to us to get some of the more advanced research done" [16, p.277]. Aircraft measurement and instrumentation had evidently evolved from a routine engineering activity to a specialist scientific discipline producing knowledge of potential economic importance.

Acknowledgments

We thank Klaas-Jan Stol, and the participants of the SEMAT General Theory of Software Engineering 2012 workshop, for many useful comments and reading suggestions. This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855.

References

[1] R. Dubin, *Theory building*. Free Press, 1978.

[2] E. Dijkstra, "Programming as a discipline of mathematical nature," *Am. Math. Mon.*, vol. 81, no. 6, pp. 608–612, 1974.

[3] P. Johnson, M. Ekstedt, and I. Jacobson, "Where's the theory for software engineering?" *IEEE Softw.*, vol. 29, no. 5, pp. 96–96, 2012.

[4] B. Boehm, "A view of 20th and 21st century software engineering," in *Proc.28th Int. Conf. on Software Engineering*. ACM, 2006, pp. 12–29.

[5] H. Petroski, *Small things considered: Why there is no perfect design*. Random House, 2003.

[6] M. Svahnberg, J. Van Gurp, and J. Bosch, "A taxonomy of variability realization techniques," *Software: Practice and Experience*, vol. 35, no. 8, pp. 705–754, 2005.

[7] B. Kitchenham and S. Pfleeger, "Software quality: The elusive target," *IEEE Softw.*, vol. 13, no. 1, pp. 12–21, 1996.

[8] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[9] D. Darcy, C. Kemerer, S. Slaughter, and J. Tomayko, "The structural complexity of software: An experimental test," *IEEE Trans. Softw. Eng.*, vol. 31, no. 11, pp. 982–995, 2005.

[10] H. Petroski, *Design paradigms: Case histories of error and judgment in engineering*. Cambridge University Press, 1994.

[11] I. Garrick and W. Reed III, "Historical development of aircraft flutter," *J. Aircraft*, vol. 18, no. 11, pp. 897–912, 1981.

[12] K. Bülah and R. Scanlan, "Resonance, Tacoma Narrows bridge failure, and undergraduate physics textbooks," *Am. J. Phys.*, vol. 59, p. 2, 1991.

[13] W. Phillips, *Journey in aeronautical research: A career at NASA Langley Research Center*, ser. Monographs in Aerospace History. NASA, 1998, no. 12.

[14] H. Reid, "The NACA three-component accelerometer," National Advisory Committee for Aeronautics, Tech. Rep., 1922.

[15] W. Brown, "The synchronization of NACA flight records," National Advisory Committee for Aeronautics, Tech. Rep., 1922.

[16] W. Vincenti, *What engineers know and how they know it: Analytical studies from aeronautical history*. Johns Hopkins University Press, 1990.

A Framework for Building Theories from Software Development Practice

Tero Päivärinta
Computer and Systems Science
Luleå University of Technology, Sweden
tero.paivarinta@ltu.se

Kari Smolander
Department of Information Technology
Lappeenranta University of Technology, Finland
kari.smolander@lut.fi

***Abstract*—The paper presents a framework for building theory from development practices. The framework locates practices in a learning loop that is situated in a development context. The framework recognizes that practices are related to their learned rationale that may come from previous experiences, i.e. observed impacts of practices, or from existing theory.**

I. INTRODUCTION

Software engineering is a practice-oriented field and the work of the software developer is in the core of its research. Lack of theory in software engineering has been recognized by researchers [1] and attempts have been made to establish a theory base [2]. In this paper we present a framework for building theories' from software development practice using six concepts and their relationships. The first part of this paper gives a brief definition of these concepts. The second parts ties them together as a framework through which theories of ISD practices can be built by learning from practice. An earlier version of the framework has been published in [3].

II. ESSENTIAL CONCEPTS FOR CREATING THEORIES OF SOFTWARE DEVELOPMENT

Our framework builds on six main concepts to be distinguished in order to learn from software development practice and to build theories of it: *learning, a practice, development context, rationale, impact,* and *theory*.

Learning is based on Argyris & Schön [4], to the idea of "theories-in-use". To learn from practice requires that we identify or assume causal relationships between actions taken during software development and desired outcomes. 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.

A central concept in our framework is the concept of *a practice*. One dictionary definition of a practice is "something people do regularly" [5]. In context of a development project or an organization, a development practice may become an *organizational practice* or *routine*, which can be defined as the organization's routine use of knowledge, especially "know-how" [6]. 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 through them between organizational contexts and over time. However, organizational practices often have tacit components embedded partly in individual skills and partly in collaborative social arrangements. If we compare a software development method and a practice, a method adopted in an organization always embodies a predefined practice or a set of them, whereas a practice is not always defined at the detailed and explicit level, at least with regard to all potential elements of method knowledge.

A software development effort takes place in a *development context*. For example, Orlikowski [7] identifies that the role of the system, 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 changes in development practices. A recent study [8] identified 170 different situational factors that affect the software development process.

The concept of *rationale* is 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 provides justifications for the creation, use and modification of the practice or set of practices.

Learning requires analysis and identification of *impacts* of the practices to the software, project, or to the development context in general. Such impacts may be desired already according to the explicit method or practice rationale(s), or they may be unexpected, sometimes even unwanted.

Finally, these concepts are needed for creating and evaluating *theories* of software development practices. That is, we pursue theories which can analyze, describe, and explain contextual practices, ultimately aiming at a level of prediction. That is, 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.

III. FRAMEWORK DEFINITION

In the following, let us relate these concepts to each other to form a framework to guide research on development practices. Figure 1 relates these concepts together and shows their relationships we need to understand in order to build theory from software development practice.

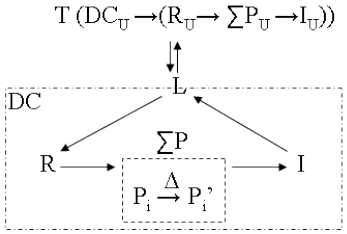

Figure 1 A framework for building theories from development practices.

Learning (L) is a boundary-spanning mechanism which need to exist, on the one hand, in a ***development context*** (DC) so that previous ***theories*** (T, including previous, more or less well-grounded, methodological recommendations) of

development can inform local ***rationale*** (R) for new practices ($\boldsymbol{T \rightarrow L \rightarrow R}$) and that observed ***impacts*** (I) of the target organization's previous ***practices*** (P) can inform further local rationality to adjust the practices ($\boldsymbol{I \rightarrow L \rightarrow R}$). 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 ($\boldsymbol{L \rightarrow T}$) (Figure 1).

Development context (DC) involves all the issues which have impact on how practices in the target organization or project are socially constructed and how the software development organization can learn from its practices. The context has impact on rationale (R) to implement new practices and to motivate change, on the actual construction of practices (P) themselves, on the impacts (I) reached from the desired change, and on the learning process and lessons learned. That is, practices, their impacts, and learning may not be purely based on the identified rationale alone, but can be affected by contextual issues (Figure 1). If contextual issues are explicitly identified before implementing a new set of practices, it becomes a part of the rationale. However, some contextual issues may have a more implicit effect on enacted practices and their impacts, recognized only after new practices have been tried out.

Learning from local and contextual development practices requires good understanding of how practices are implemented and used in any target context of development. The contextual rationale(s) for particular practices and their improvements should cause meaningful change(s) in a practice or a set of practices, which are, again, often a part of a larger, interrelated set of practices ($\boldsymbol{R \rightarrow \sum P,\ \Delta(P_i \rightarrow P_i')}$) in the context. Moreover, contextual impact(s) after a practice has been introduced or changed need to be studied ($\boldsymbol{\Delta(P_i \rightarrow P_i') \rightarrow I}$), and lessons learned from the observed impacts need to be distilled ($\boldsymbol{I \rightarrow L}$) (Figure 1.)

If observed changes and improvements in local practices are used to contribute to a theory (**T**) of a selected set of general-level development practices (beyond the context in question) through a learning process ($\boldsymbol{L \rightarrow T}$) , then we need also to recognize ideas of more generic or universal rationales giving reasons to implement certain types of practices ($\boldsymbol{(R_U \rightarrow \sum P_U \rightarrow I_U)}$). As well, generic ideas to categorize development contexts, which may have impact on rationales, enactment of particular practices, and impacts resulting from particular practices, may be theorized. Through learning from the target context(s), software development research may theorize further on more universal issues of the development context ($\boldsymbol{(DC_U)}$), their impact on rationales for practices, actual practice domains of interest, and the generalized ideas of impacts from choosing particular practices ($\boldsymbol{DC_U \rightarrow (R_U \rightarrow \sum P_U \rightarrow I_U)}$) (Figure 1). Here, it is important to denote that the descriptions of development contexts, rationales, practices, and their impacts at the level of a theory should be distinguished from the observed practices (or local interpretations of practices) in the context.

We believe that theories of software development practices should pursue to promote understanding of reasons why to consider implementation of particular idealized practices and impact of those practices, discussed in the light of theoretical categories of contextual issues and contingencies. Such theories (T) would be able to answer to three research questions, which we believe to be of interest for scholars, educators, and practitioners (Figure 1.):

- Why are particular practices followed (or not) in software development? ($\boldsymbol{R_U \rightarrow \sum P_U}$)
- What are the expected impacts (both desired and undesired) from adhering to a set of certain pre-described practices? ($\boldsymbol{\sum P_U \rightarrow I_U}$)
- How are certain types of development contexts expected to affect on the rationale for, the impact on, and the implementation of certain pre-described practices? (**DCU →(RU→∑ PU→IU)**)

IV. Final Remarks

The biggest differences between our framework and the Essence by the SEMAT initiative [2] are:

1. The Essence attempts to build ontology of software development by identifying pertinent 'alphas' and 'activity spaces'. Our framework is more research oriented and uses a very limited set of concepts and leaves the ontology building to research through observation and *learning*.
2. In our framework all theory is used and produced through the lens of *learning*. In the Essence learning is not so much a part of theory creation.

Although we have presented the framework rather formally, we believe that the actual theories of software development practice are usually informal by nature. Software development is a human activity that is done in social organizations. The development context is therefore often unique and exact repeating of studies is hard. Software itself is often social and linguistic by nature. Therefore results of many empirical studies will remain descriptive. Hence, advanced theorizing will require synthesizing over the existing base of rather idiographic empirical results from literature. Any theorizing effort would require a knowledge base including thorough descriptions of lessons learned from contextual software development cases, to enable establishment of theoretical patterns among similar types of cases. Theoretical efforts to integrate already existing reports and lessons learned into more generic theoretical models are also required.

V. References

[1] P. Johnson, M. Ekstedt, and I. Jacobson, "Where's the Theory for Software Engineering?," *Software, IEEE*, vol. 29, no. 5, pp. 96–96, 2012.

[2] SEMAT, "Essence - Kernel and Language for Software Engineering Methods (OMG ad/2012-08-15, Revised submission)," *http://semat.org/wp-content/uploads/2012/02/12-08-15.pdf*, 01-Oct-2012. .

[3] T. Päivärinta, K. Smolander, and E. Å. Larsen, "Towards a Framework for Building Theory from ISD Practices," in *Information Systems Development*, J. Pokorny, V. Repa, K. Richta, W. Wojtkowski, H. Linger, C. Barry, and M. Lang, Eds. Springer New York, 2011, pp. 611–622.

[4] Argyris, Chris and Schön, D. A., *Organizational Learning II*. Reading, MA: Addison-Wesley, 1996.

[5] Collins CoBUILD, *English Dictionary*. 1989.

[6] B. Kogut and U. Zander, "Knowledge of the Firm, Combinative Capabilities, and the Replication of Technology," *Organization Science*, vol. 3, no. 3, pp. 383–397, Aug. 1992.

[7] W. J. Orlikowski, "CASE Tools as Organizational Change: Investigating Incremental and Radical Changes in Systems Development," *MIS Quarterly*, vol. 17, no. 3, pp. 309–340, 1993.

[8] P. Clarke and R. V. O'Connor, "The situational factors that affect the software development process: Towards a comprehensive reference framework," *Information and Software Technology*, vol. 54, no. 5, pp. 433–447, 2012.

Theory Building Attempts in Software Engineering

Per Runeson
Software Engineering Research Group
Lund University, Sweden
per.runeson@cs.lth.se

***Abstract*—The lack of theory in software engineering is acknowledged and empirically shown. Still there exist attempts to build theories in the literature. This position paper briefly introduces the published works on theory building in software engineering and outlines key characteristics of software engineering theories to be applied for future theory building and use.**

***Index Terms*—software engineering; theory; empirical studies;**

I. Introduction

Theory building is an activity aimed at abstracting and generalizing knowledge within a field of research. It is assumed to improve communication among researchers and support building new research upon existing knowledge. According to Hannay et al. [6] *"[a] theory provides explanations and understanding in terms of basic concepts and underlying mechanisms, which constitute an important counterpart to knowledge of passing trends and their manifestation"*.

Hannay et al. [6] conducted a systematic literature review of software engineering experiments, 1993–2002. They found 40 theories in 23 articles, out of the 113 articles in the review. However, only two of the theories were used in more than one article – hence theories were not used for communication and building upon existing research. The use of theory is hence concluded being scarce in software engineering, in the observed time period. Although there is no systematic literature review conducted for the sub-sequent decade, we have not seen any clear indications in the major software engineering journals of substantial activity in the theory building and use.

This position paper is based on a chapter by Wohlin et al. [12, p. 21-22]. We first summarize existing work on theory building and the discuss where to go from there.

II. Related Work

Endres and Rombach [4] identified a list of 50 findings which they referred to as 'laws', which is a notion for a description of a repeatable phenomenon in a natural sciences context. They applied this notion to software engineering. Many of the listed 'laws' are rather general management theory than software engineering, for example, *"it takes 5000 hours to turn a novice into an expert"*. In their notion, *theories* explain the 'laws', *hypotheses* propose a tentative explanation for why the phenomenon behaves as observed, while a *conjecture* is a guess about the phenomenon. Endres and Rombach listed 25 hypotheses and 12 conjectures appearing in the software engineering literature.

Zendler [13] took another approach, defining a "preliminary software engineering theory", composed of three fundamental hypotheses, six central hypotheses, and four elementary hypotheses. He defined a hierarchical relation between the hypotheses, the fundamental being the most abstract, and elementary the most concrete ones, originating from outcomes of experimental studies. An example theory according to Zendler is *"object-oriented programming techniques have advantages against structured programming techniques"*. He surveys software engineering experiments, and structure his findings in fundamental, central and elementary hypotheses. However, the chain of evidence from the experiments to the hypotheses are not clearly reported.

Gregor [5] described five general types of theory, which may be adapted to the software engineering context according to Hannay et al. [6]:

1) *Analysis:* Theories of this type describe the object of study, and include, for example, taxonomies, classifications and ontologies.
2) *Explanation:* This type of theories explains something, for example, why something happens.
3) *Prediction:* These theories aim at predicting what will happen, for example, in terms of mathematical or probabilistic models.
4) *Explanation and prediction:* These theories combine types 2 and 3, and is typically what is denoted an "empirically-based theory".
5) *Design and action:* Theories that describe how to do things, typically prescriptive in the form of design science It is debated whether this category should be denoted theory at all.

Sjøberg et al. [11] propose a framework for software engineering theories, comprising of four main parts: (i) Constructs, (ii) Propositions, (iii) Explanations, (iv) Scope, The *constructs* are the entities in which the theory are expressed, and to which the theory offers a description, explanation or prediction, depending on the type of theory as defined above. *Propositions* are made up from proposed relationships between the constructs. The *explanations* originate from logical reasoning or empirical observations of the propositions, that is, the relationship between the constructs. The *scope* of the theory defines the circumstances, under which the theory is assumed to be applicable. Sjøberg et al. [11] suggest the scope being expressed in terms of four archetype classes: actor, technology, activity and software system, see Table I.

TABLE I
FRAMEWORK FOR SOFTWARE ENGINEERING THEORIES, AS PROPOSED BY SJØBERG ET AL. [11].

Archetype class	Subclasses
Actor	Individual, team, project, organisation or industry
Technology	Process model, method, technique, tool or language
Activity	Plan, create, modify or analyze (a software system)
Software system	Software systems may be classified along many dimensions, such as size, complexity, application domain, business/scientific/student project or administrative/embedded/real time, etc.

The scope description is a kind of minimal ontology. *Ontology engineering* is a field originating from knowledge representation, and there are exist ambitious initiatives to explore this, both to develop principles, methods, tools and languages for ontologies in software engineering, e.g. Calero et al. [1] as well as defining the ontologies themselves, e.g. Jacobson et al. [7].

III. FUTURE OF SOFTWARE ENGINEERING THEORIES

Based on the attempts to develop and use theories in software engineering, we derive three criteria that should apply to any software engineering theory. A theory should, (i) be empirically founded, (ii) have a defined scope, and (iii) be relevant.

A theory should be *empirically founded*, based on a systematic collection of empirical evidence, e.g. through systematic literature reviews [8]. Whether the evidence is based on experiments [12] or case studies [10] is a secondary issue, but the systematic *collection* [8], and *synthesis* [2] of the evidence is crucial. The above cited theory proposals are based on empirical studies [4, 13], but they fail with respect to the systematic collection of empirical studies, as well as in an transparent synthesis, leading to the theory.

A theory in software engineering should have a *defined scope*. Firstly, the scope should be software engineering, and not interfere with general management theory, as in one of the early attempts to software engineering theory [4]. Secondly, also within software engineering, the scope of the empirical findings may be very specific to the scope, as recently illustrated by Dybå et al. [3], who observed, as an example, that pair programming worked very differently for novices and experienced programmers. Consequently, the scope of a theory on pair programming must this into account.

Finally, a theory must *be relevant* for the practitioner. Knowing that *"object-oriented programming techniques have advantages against structured programming techniques"* [13] does not help neither a practitioner nor a researcher, unless it defines which advantages is has, and under which circumstances these advantages are observed.

There is a conflict between the ambitions of researchers to provide rigorous answers and practitioners' expectations of short and clear answers. We cannot avoid this dilemma, but have to face it and derive empirically founded, scoped and relevant theories to communicate software engineering knowledge. Hopefully we will embrace Lewin's famous quotation: "There is nothing so practical as a good theory" [9, p. 169].

IV. ACKNOWLEDGMENT

Thanks to Professor Claes Wohlin for fruitful discussions during the update work on the Experimentation book [12] and feedback on this position paper.

REFERENCES

[1] C. Calero, F. Ruiz, and M. Piattini, editors. *Ontologies for Software Engineering and Software Technology*. Springer Berlin Heidelberg, 2006.

[2] D. S. Cruzes, T. Dybå, P. Runeson, and M. Höst. Case studies synthesis: Brief experience and challenges for the future. In *Proc. 5th Int. Symp. on Empirical Software Engineering and Measurement*, Banff, Canada, 2011.

[3] T. Dybå, D. I. K. Sjøberg, and D. S. Cruzes. What works for whom, where, when, and why? On the role of context in empirical software engineeringn the role of context in empirical software engineering. In *Proc. 6th Int. Symp. on Empirical Software Engineering and Measurement*, pages 19–28, Lund, Sweden, 2012.

[4] A. Endres and H. D. Rombach. *A Handbook of Software and Systems Engineering – Empirical Observations, Laws and Theories*. Pearson Addison-Wesley, 2003.

[5] S. Gregor. The nature of theory in information systems. *MIS Quarterly*, 30(3):491–506, 2006.

[6] J. E. Hannay, D. I. Sjøberg, and T. Dybå. A systematic review of theory use in software engineering experiments. *IEEE Transactions on Software Engineering*, 33(2):87–107, 2007.

[7] I. Jacobson, P.-W. Ng, P. E. McMahon, I. Spence, and S. Lidman. *The Essence of Software Engineering – Applying the SEMAT Kernel*. Pearson Education, Inc., 2012.

[8] B. A. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering (version 2.3). Technical Report Technical Report EBSE-2007-01, Keele University and Durham University, July 2007.

[9] K. Lewin. In D. Cartwright, editor, *Field theory in social science; selected theoretical papers*. Harper & Row, New York, 1951.

[10] P. Runeson, M. Höst, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering – Guidelines and Examples*. Wiley, 2012.

[11] D. I. K. Sjøberg, T. Dybå, B. Anda, and J. E. Hannay. Building theories in software engineering. In F. Shull, J. Singer, and D. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*. Springer-Verlag, London, 2008.

[12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer, 2012.

[13] A. Zendler. A preliminary software engineering theory as investigated by published experiments. *Empirical Software Engineering*, 6:161–180, June 2001.

Knowledge Transformation in Software Development Processes

Ilia Bider
Department of Computer and Systems Sciences (DSV), Stockholm University (SU)
IbisSoft AB
Stockholm, Sweden
ilia@{dsv.su.se | ibissoft.se}

This position paper discusses the theoretical foundation of Software Engineering (SE) and argues that it should consist of two branches, the first branch deals with the generic properties of software products, the second one deals with the generic properties of SE processes. The paper argues that one of the important properties of the software development process is its model of knowledge transformation, and it suggests to adapt Nonaka's SECI model for investigating and comparing various software development methodologies.

Software Engineering; knowledge transformation; SECI model; software development; agile

I. INTRODUCTION

Software Engineering (SE) is an engineering discipline that deals with manufacturing of certain kind of products – software. Therefore, the SE theoretical foundation should concern both:

- the properties of the product (software), and
- the properties of the processes of product development, maintenance and retiring/disposing

This means that the theoretical foundation of SE is naturally split into two main areas, which while being interconnected have different underlying concepts.

As we deal with the theoretical foundation, the properties of both products, and processes should be considered on the high abstract level, so that the theory can be applied to any kind of software, and to any kind of methods of software development and maintenance.

A typical example of product properties on the high abstract level is Software Quality, the subject addressed in numerous research works. Less popular subject that, in our view, is of great theoretical interest is multilayered architecture of software products. We believe that the theoretical foundation for creating a product-independent platform to address this subject can be found in the works of Michael Polanyi [1]. In it, he introduces the principle of boundary control according to which each layer has two sets of laws to obey. One set is the laws of its own layer, the other set is the laws of boundary control from the layer above which uses this layer for attending its own goals/properties. In addition, he states that the laws of the upper layer are not reducible to the laws of the lower one.

In this paper, we, however, leave the properties of the software products outside the scope of our consideration and concentrate on theoretical foundation for the software development process. As this is an intellectual process which practically does not deal with the physical world, the transformation of knowledge in this process is of major importance. There are numerous methods of software development; therefore creating a theoretical foundation for knowledge transformation that is applicable to any of them is a challenging task. We propose to use ideas from Nonaka's SECI model [2] as a foundation for the theory of knowledge transformation in the software development process. In this short paper, we will give an outline of how ideas from SECI could be applied to software development.

II. SECI MODEL

SECI stays for Socialization – Externalization – Combination – Internalization, see Fig. 1, was developed in [2] to explain the ways how knowledge is created in an organization while being transformed from the tacit form to explicit and back.

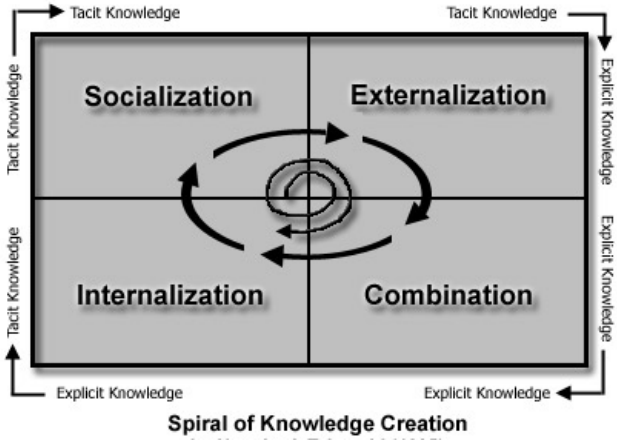

Figure 1. SECI diagram, adapted from http://gramconsulting.com

In the sections that follow, we demonstrate how the ideas from SECI can be applied to software development process.

III. KNOWLEDGE TRANSFORMATION IN THE TRADITIONAL SOFTWARE DEVELOPMENT

Knowledge transformation during the traditional cycle of software development is represented in Fig 2. It starts with tacit knowledge on the needs that exists in the heads of the stakeholders. Then, it is transformed into explicit knowledge of requirements specifications which correspond to

Externalization in Fig. 1. After that, this explicit knowledge is converted into another explicit form of design specifications. This transformation corresponds to *Combination* in Fig. 1, as the transformation is done with the help of software design principle of the appropriate SE domain. The next step, *Coding*, consists of transforming explicit knowledge of design specifications into the knowledge *embedded* into a software system. Though parallel to *Internalization* from Fig. 1, this step does not correspond to the latter exactly. We call this step *Embedment*. The next step, *Learning to use*, consists of transforming the knowledge embedded in the software system into the tacit knowledge of its users that use the system in their practice. Though parallel to *Socialization* from Fig. 1, this step does not correspond to the latter exactly. We call this step *Adoption*.

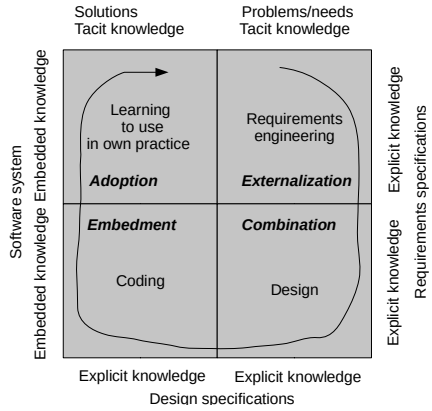

Figure 2. ECEA - Knowledge transformation in the traditional software development

The resulting model, which abbreviates to ECEA, can be useful for analyzing risks inherent to the traditional software development:

1. Requirements does not catch the needs properly
2. Requirements are not converted into a proper design
3. Coding does not follow the design exactly
4. The new software is not properly understood by its users, and it is rejected or used in the wrong fashion

The risks above can be minimized by employing qualified requirements engineers, developers, programmers, and trainers. However, they can never be totally eliminated. The biggest risk of all, however, in today's highly dynamic environment is that:

5. While a new system is under development, the problems/needs are continuing to evolve. As the result, a wrong/outdated system is delivered to the stakeholders.

IV. KNOWLEDGE TRANSFORMATION IN THE AGILE SOFTWARE DEVELOPMENT

One way to minimize the risks of the traditional software development is to use the agile principles [3]. Knowledge transformation in the agile development in the idealized form is represented in Fig. 3. In it, the *Design* phase is removed, and one big cycle is substituted by many small ones. This corresponds to the main idea of agile development to avoid, as much as possible, transforming knowledge into explicit form. Minimum requirements and design documents, e.g., notes, emails, or black- whiteboard diagrams, however, still exist, but they do not represent legally binding requirements of the traditional approach.

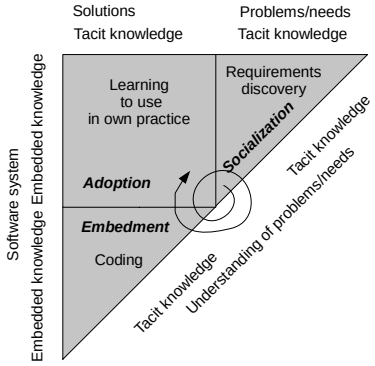

Figure 3. Knowledge transformation inthe the agile softwre development

As we see from Fig. 3, the nature of the requirements engineering phase is also changed. It consists in transferring tacit knowledge of problems/needs from the stakeholders to the design team, and thus corresponds to *Socialization* in Fig. 1. As the result we get the repeating cycle of *Socialization- Embedment- Adoption* – SEA model.

V. CONCLUSION

As we discussed in Section I, the theoretical foundation of SE should have two branches, one concerns the generic properties of software, the other concerns the properties of SE processes. As there are multiple different methods of developing software, the theory of the second brunch should be able to model different sides of these methodologies, so that they could be compared, and the one that suits best a given context of development could be be chosen.

We believe that SE theory should include an approach to describing and modeling knowledge transformation in software development processes. As we have shown in Section II-IV, SECI model from [2] could be adapted for this kind of modeling.

The ideas presented in this position paper are based on the analysis of the authors own practice of developing software systems (Agile as well as not agile) and introducing them into organizational practice, see for example [5].

REFERENCES

[1] M.S. Polanyi, Knowing and Being. Chicago, University of Chicago Press, 1969.

[2] I. Nonaka, "A dynamic theory of organizational knowledge creation," Organ. Sci., 5(1), 1994, pp. 14–37.

[3] Agile manifesto. http://agilemanifesto.org/.

[4] T. Andersson, I. Bider and R. Svensson R. "Aligning people to business processes experience report", Software Process Improvement and Practice, 10(4), 2005, pp. 403.

Requirements on General Theories of Software Engineering

An Unusually Dense Position Paper

Pontus Johnson
KTH Royal Institute of Technology
Stockholm, Sweden
e-mail: pontus@ics.kth.se

Iaakov Exman
The Jerusalem College of Engineering
Jerusalem, Israel
e-mail: iaakov@jce.ac.il

***Abstract*— In this paper, we propose a set of quality criteria of general theories of software engineering: The quality of a general theory of software engineering depends on (i) the universality and precision with which it predicts the influence of the software decision makers' actions on the software development goals, (ii) its degree of corroboration, (iii) its degree of formalization, (iv) the unambiguousness of its measurement procedures. The argument for these quality criteria is based on (a) the oft-quoted adage by Kurt Lewin, "There is nothing so practical as a good theory", (b) Karl Popper's *The Logic of Scientific Discovery*, and (c) the Essence of the SEMAT Initiative.**

I. INTRODUCTION

Quality criteria for scientific theories have been extensively examined within the field of philosophy of science [1]. In this paper, we propose a slightly modified set of criteria suitable for general theories of software engineering. The purpose of the proposal is to guide the search for a general theory of software engineering, as well as to contribute to an increased awareness of the importance in practice of software engineering theories.

II. PREDICTING DECISIONS' EFFECTS ON GOALS

Kurt Lewin famously proposed that "There is nothing so practical as a good theory" [2]. In the context of software engineering, the word "practical" arguably means that the theory is of benefit to the engineering effort, i.e. that it somehow aids the practitioner attaining her design goals. How then can a theory provide such aid? A theory can itself not perform actions in the real world. Its only manner of influencing the world is through the actions of decision makers; a person's behavior is contingent on the theory she subscribes to. The benefits of theory thus must come through its decision-guiding capacity.

Given that decision-makers strive for certain goals with a limited set of means at their disposal, their problem is to determine which of the available actions achieve what results; which knobs influence what gauges. This is a problem that a theory can solve; a theory can be used to predict the effects of actions on goals. This leads to a first version of quality criteria (i): *The quality of a general theory of software engineering depends on the extent to which it predicts the influence of the software decision makers' actions on the software development goals.*

III. UNIVERSALITY AND PRECISION

In a subsequent section, the software decision makers' actions and goals are elaborated on. Another critical part of the formulation of the first criteria is "the *extent* to which [the theory] predicts […]". The iconic philosopher of science Karl Popper [3] employs the term *empirical content* to denote that "extent". According to Popper, empirical content is increased either by increasing the *universality* of the theory or by increasing its *precision*. To explain these concepts, Popper exemplifies with the following four theories:

p: All heavenly bodies which move in closed orbits move in circles: or more briefly: All orbits of heavenly bodies are circles.

q: All orbits of planets are circles.

r: All orbits of heavenly bodies are ellipses.

s: All orbits of planets are ellipses.

Moving from p to q, the degree of universality decreases; and q says less than p because the orbits of planets form a proper subclass of the orbits of heavenly bodies. Moving from p to r, the degree of precision decreases: circles are a proper subclass of ellipses. Corresponding remarks apply to the other moves. To a higher degree of universality or precision corresponds a greater empirical content. In summary, higher universality means that more things can be predicted, and higher precision means that those predictions become more exact.

Universality and precision are implied in quality criteria (i). A theory that only predicts the effect of one kind of action, e.g. choice of development method, on one kind of goal, e.g. development effort, is less universal than a theory that also can predict the effect on software quality. A theory that only vaguely predicts that a big application will be more costly to develop than a small one is less precise than a theory that predicts the cost in dollars and cents based on function points. A reformulation of criteria (i) is thus: *The quality of a general theory of software engineering depends on the universality and precision with which it predicts the influence of the software decision makers' actions on the software development goals.*

IV. DEGREE OF CORROBORATION

According to Popper, "Theories are not verifiable, but they can be 'corroborated'. The attempt has often been made

to describe theories as being neither true nor false, but instead more or less probable. […] Instead of discussing the 'probability' of a hypothesis we should try to assess what tests, what trials, it has withstood; that is, we should try to assess how far it has been able to prove its fitness to survive by standing up to tests. In brief, we should try to assess how far it has been 'corroborated'."

Software engineering theories do not differ from other theories in this respect; corroboration is a quality criteria. A theory that has passed many difficult empirical trials is better than one that has not. This leads to quality criteria (ii): *The quality of a general theory of software engineering depends on its degree of corroboration.*

V. Degree of formalization

Popper continues: "The requirement of consistency […] can be regarded as the first of the requirements to be satisfied by every theoretical system, be it empirical or non-empirical. In order to show the fundamental importance of this requirement it is not enough to mention the obvious fact that a self-contradictory system must be rejected because it is 'false'. […] But the importance of the requirement of consistency will be appreciated if one realizes that a self-contradictory system is uninformative. It is so because any conclusion we please can be derived from it. Thus no statement is singled out, either as incompatible or as derivable, since all are derivable."

The best way to avoid inconsistency is by formalization. A sufficiently formalized system can be subjected to automated methods to detect inconsistencies. But benefits of formalization appear before a system has been mathematically formalized. A theory presented in a structured form is often less ambiguous than a theory presented casually.

This leads to quality criteria (iii): *The quality of a general theory of software engineering depends on its degree of formalization.*

VI. Ambiguity of measurement procedures

As stated in the second section, a precise theory is preferable to an imprecise theory. Imprecision can appear in two relations. The first relation, discussed in Section II, is between the constructs of the theory. For instance, the proposition "System X is larger than System Y" is less precise than "System X is twice as large as System Y".

The second relation that can cause imprecision is between the constructs of the theory on the one hand and the real world on the other. In other words, the theory's measurement procedures may be imprecise, or stated differently, the theory's definitions may refer to the real world in imprecise ways. A theory that represents system response time on an ordinal scale of "high", "medium" and "low" is less precise than a theory that represents response time in seconds.

This leads to quality criteria (iv): *The quality of a general theory of software engineering depends on the ambiguity of its measurement procedures.*

VII. Software Engineering Actions and Goals

Having formulated the four quality criteria, we may ask:

-Which specific features distinguish a software engineering theory from other theories?

To this end we elaborate additionally on the first quality criteria by detailing the actions and goals of the software decision maker. We believe that it is fruitful to relate these to *the Essence* developed by the SEMAT initiative [4]. In the Essence, a key concept is the *Alphas*. Alphas are "representations of the essential things to work with" [4]. Examples of alphas are *Software System, Requirements, Work, Team, and Way of working.* We propose that these alphas constitute the base for the *actions* of the software decision maker in quality criteria (i). For instance, an action might be to change the requirements, or to modify the way of working by replacing one practice with another.

Considering the software development *goals*, SEMAT describes these as *better software, faster and with happier customers*. Thus, criteria (i) can be further elaborated to the following: *The quality of a general theory of software engineering depends on the universality and precision with which it predicts how actions on the SEMAT alphas lead to (or away from) better software faster and with happier customers.*

VIII. Summary

We have presented four quality criteria for general theories of software engineering that we believe should guide theory development within the field: (i) the universality and precision with which it predicts the influence of the software decision makers' actions on the software development goals, (ii) its degree of corroboration, (iii) its degree of formalization, (iv) the unambiguousness of its measurement procedures.

We believe that good SE theories – obeying the above quality criteria – will not appear like fashionable meteors just to fall in disregard a short time after. Each good theory will be tested in the laboratory and in the field against real software systems and endeavors – its predictions gradually gaining confidence by corroboration. This will impart relatively long-term resilience, until a better theory appears and replaces it, like in any other field of science and engineering.

References

[1] P. Godfrey-Smith, *Theory and reality: An introduction to the philosophy of science*, University of Chicago Press, 2003.

[2] K. Lewin, *Field theory in social science; selected theoretical papers.* D. Cartwright (ed.). Harper & Row. 1951.

[3] K. Popper, The Logic of Scientific Discovery, Hutchinson & Co, 1959.

[4] I. Jacobson et al., *The Essence of Software Engineering: Applying the SEMAT Kernel*, Addison-Wesley Professional, 2013.

States and Transformations for Software Engineering Theory

(Position Paper)

Hannu-Matti Järvinen and Mikko Tiusanen
Department of Software Systems
Tampere University of Technology
P.O. Box 553 (Korkeakoulunkatu 1)
FI–33101 Tampere, FINLAND
Email: {hannu-matti.jarvinen,mikko.tiusanen}@tut.fi

***Abstract*—Our position is that an engineering discipline needs a theory that can be used to predict the properties of its artefacts. If one follows the example of Maxwell's equations, software engineering needs to be based on a theory of computation, like electrical engineering is based on a theory of electromagnetism. Such a theory of computation is likely to fundamentally be about states and state transformations.**

***Index Terms*—Software engineering, Theory, State, Transition.**

I. Introduction

Writing about what is the theory of one's subject domain is an exercise in finding the limits of one's thinking. There need to be limits, since otherwise the subject is all-encompassing: without limits nothing is excluded. Nevertheless, limits on thinking also limit solutions that get considered.

Maxwell's equations describe limitations on the values of measurable quantities of electromagnetic phenomena. They are not exact—at limit, since they ignore quantization—but bulk descriptions. They do, however, allow a limiting process of more accurate approximations until the quantization effects break this down. As such they are eminently applicable to engineering, as witnessed by their success.

Engineering is the art of constructing artefacts to provide services to society. Specifically, it is an art, not a science! A science would be more concerned about understanding its underlying phenomena.

Engineering does not need understanding as much as it needs an ability to make predictions, hopefully quantifiably reliable. Engineering has used many phenomena long time before the physicists, chemists, or others have come up with explanations of how these come about. For the engineer being able to achieve an effect reasonably reliably is more important than why the effect comes about. (The same is true of the magician: it is not without cause that Thomas Alva Edison was called the "wizard of Menlo Park".)

Software is unique in that its construction is to *verbalize* it in a form that spawns a behaviour when interpreted. One of the subliminal attractions of writing software is this—here rather trivial—godlike ability to say "Let there be light" and make light appear. Before software, you needed to at least connect together components to make something, say, a radio receiver by soldering. With software, you can make a radio by writing a program to process radio frequency samples and to procuce sound—well, some can... Small change in the program can can also radically alter its behaviour, making programs chaotic systems. Moreover, a program can spawn a behaviour that can be *partially* parameterized by the conditions under which it is interpreted, only partially, since it can display also at least seemingly random traits. All such behaviours should be predictable, however. So, constructing software is quick and easy, but getting it *predictably* to behave as is desired can be even harder than with other branches of engineering.

II. Discussion

We shall briefly treat each of the questions posed by the call-for-pariticipation, in order:

What are the objectives of a theory of software engineering: These are the same as for any engineering discipline: to be able to predict the behaviour of its artefacts.

The artefacts of software engineering are computations or descriptions of such. Computations are processes that transform state. Therefore, processes that produce computations are also such artefacts (computations), as are processes that produce such processes, recursively, *ad infinitum*, along with descriptions of these. Obviously, the quality of the prediction may deteriorate in each recursion due to approximations made.

Another obvious observation is that the above, if acceptable as a definition of computation, covers many physical, chemical, psychological, and other processes, continuous as well as discrete. Is this of concequence, however? The theory can be restricted to cover only those aspects of current interest. Restricting the domain too much, however, makes for rigidity: Perhaps quantum computations are to be realized by quantum mechanical processes? Would these then fall outside the scope of software engineering simply because the definition of software engineering does not allow for such processes? Some of the rather commonly accepted parts of the theory of software engineering are indeed rather close to social psychology, say, Conway's law.

How can a theory be of use in practice: In ways, exemplified by any theory used by any engineering discipline, to predict the behaviour of its artefacts.

What is a useful definition of theory: Logically, a theory is a collection of true statements about a domain of discourse. Some statements will be statistical or stochastic ones. Determination of the truth of a statement is not a part of logic which only assumes that this can be done if necessary: it is simpler for some domains of discourse. Indeed, it may be impossible, *re* Gödel and Turing, e.g.

The true statements will by necessity be conditional in that they will need to describe the conditions under which they hold.They will also name concepts of relevance. What these concepts would be for a theory of software engineering, is unclear to us. What the practitioner will use will be something that is based on such true statements, most likely quite distinct in appearance from these.

What questions should a theory of software engineering address: This needs to be a theory of computations, rather, as engineering will need to use it to predict the properties of its artefacts. Specifically, the construction of software in a software project is a computation by the above definition (it is a process to produce a relevant artefact), the predictability of which and its results is the urgent problem for software engineering. The question to be answered is how to provide predictability to a level at least somehow comparable to other engineering disciplines.

How foundational/universal should it be: As fundamental as is needed. It is rather fundamental, if the above definition of a computation is accepted.

What should the main concepts of the theory be: Our suggestion is to use local state and local transformation of state. These produce a computation, a flow of control potentially distributed and concurrent. This age-old dichotomy of states and state transformations characterize processes across scientific disciplines, e.g., [1]. These can also be used to describe also human behaviour to a degree at least. SEMAT uses these concepts when talking about alphas.

This is only a partial answer, obviously: States and transitions are very low-level concepts. If following the lead of Maxwell's equations—not necessary but perhaps plausible—the issue would be, what are the measurable bulk quantities that would have to be related by the theory, those that are relevant for increased predictability of sofware projects. If software projects are computations, trajectories of interest, what, by analogy to Maxwell's equations, are the concepts corresponding to fields (or potential functions) that affect these trajectories? These are not clear to us.

It is most likely not wise to restrict these concepts even to realisable states, transformations, and computations, since unrealisable ones may be relevant for the theory, although less likely to the practice. Obviously, it is definitely not wise to restrict to serial computations, since these only cover a portion of the interesting ones: even a software project tends to have concurrent parts even if actually being executed serially by one person!

Should a theory of software engineering be expressed formally: Indeed, it should. If it can be expressed with the conciseness of Maxwell's equations, so much the better. Whether this can be done is not clear.

If formalized, what is a suitable language for a theory of software engineering: Mathematics and logic are the best language that we as human beings have invented in order to describe and then predict the behaviour of processes, among other things. They are obviously incomplete: to quote Eugene O'Neill, "A poor thing but mine own." [2]

It is not clear if the mathematics needed to achieve such a formalization exists or even can be created, although the human mind has been in the past been able to achieve similar breakthroughs when put under the pressure of necessity.

III. Conclusion

As is obvious, there are significant bodies of knowledge that can be drawn upon that have explored related issues. The ones closest to our current interests are action systems, as exemplified by Unity (of Chandy and Misra [3]) and DisCo (of Kurki-Suonio [5]), Temporal Logic of Actions (TLA, of Lamport [4]), and Petri nets [1]. The weight of a body of knowledge is, however, of little consequence unless it is relevant to increasing the predictability.

References

[1] C.A. Petri, *State-transition Structures in Physics and in Computation*, Int. J. of Theor. Physics 21(1982)12, 979–992.

[2] E.G. O'Neill, *Long Day's Journey Into Night* (First edition). New Haven 1956. ISBN 250174075.

[3] K.M. Chandy, J. Misra, *Parallel Program Design: A Foundation*. Addison-Wesley 1988. ISBN 0201058669.

[4] L. Lamport, *Temporal Logic of Actions*. Research report 71, Digital systems research center, 1991.

[5] R. Kurki-Suonio, *A Practical Theory of Reactive System*, Springer Berlin Heidelberg New York, 2005. ISBN 3-540-23343-3.

Ontology-Based Modeling of the Software Quality Assurance Knowledge

Nada Bajnaid
King Abdulaziz University, SA
London Metropolitan University
United Kingdom
nab0378@londonmet.ac.uk

Algirdas Pakštas, SM IEEE
London Metropolitan University
United Kingdom
a.pakstas@londonmet.ac.uk

Shahram Salekzamankhani
London Metropolitan University
United Kingdom
s.salekzamankhani@londonmet.ac.uk

Abstract

Software is a key element of the modern computing systems (from mobile phones to supercomputers) and there is a need for high standards in educating people who are involved in its development. It becomes especially critical when there are special requirements for high quality software. This paper presents an ontological model to describe and define the Software Quality Assurance SQA knowledge area. International standards (SWEBOK, IEEE, and ISO) were the main sources of the terminology and semantic relations of the developed SQA model. The Application-Based ontology evaluation is used to measure practical aspects of ontology deployment. An ontology-based e-learning prototype was designed and implemented to guide students and practitioners about a process of development of the SQA compliant software.

1. Introduction

In software engineering, people with different necessities and viewpoints need to communicate and share knowledge during all the stages of the software life cycle. Information sharing helps to prevent inconsistency among teams that are geographically dispersed and are participated in the development process.

Using ontology to model the SE knowledge shorts the development time, improves productivity, decreases cost, and increases product quality. Ontologies provide better understanding of the required changes and the system to be maintained [Calero, 2006, p. 57-62; Mendes, 2004; Wille, 2004]. Software engineering domain ontologies are very useful in developing high quality, reusable software by providing an unambiguous terminology that can be shared through the development processes. Ontologies also help in eliminating ambiguity, increasing consistency and integrating distinct user viewpoints [Uschold, 1996; Zhao, 2009].

In addition, software engineering ontologies can be used to support the translation between different languages when different users/agents need to exchange data. Designers with different backgrounds and viewpoints working on the same project can be helped by ontologies in the requirement specification process by offering a declarative specification of the system, its components and the relationship between the components [Calero, 2006].

2. Building the SQA Ontology

There was an effort by different bodies to develop Software Engineering standards followed by the forming of the ISO/IEC Joint Technical Committee 1 (JTC1) workgroup in order to guarantee consistency and coherency among standards. The IEEE Computer Society and the ISOJTC1-SC7 agreed to harmonize terminology among their standards. However, there is still no single standard which embraces the whole SQA knowledge. Because of that, there are various vocabularies to describe the SQA knowledge in learning context including textbooks. In addition, Software Engineering teachers have different backgrounds, use different languages and/or jargons which motivate additional research related to SQA teaching.

Our research aims to investigate, design and evaluate a model of the SQA knowledge area that would facilitate automated retrieval of the domain knowledge using ontologies. After a thorough review of the software engineering field and software quality knowledge area in particular, ontologies (development methodologies, tools, and languages), and previous work in literature of developed ontologies in the field of software engineering and technologies, we built a software quality ontology model that represents the main software quality concepts and relations among them. The primary source of the developed SQA ontology is the SWEBOK guide (2004), in addition to that, ISO and

IEEE standards (ISO 9126, IEEE 12207, IEEE 610.12, IEEE 00100, SWEBOK 2004, PMBOK 2008) were used and from them relevant terminology was extracted with help of domain specialists. The developed ontology was implemented using the Web Ontology Language OWL. Protégé was selected as the ontology editing and knowledge acquisition tool [Bajnaid et al., 2011].

3. Evaluating the Developed SQA Ontology

Higher quality ontologies can be easier reused and shared with confidence among applications and domains. Additionally in case of re-use, the ontology may help to decrease maintenance costs [Vrandečić, 2010]. Thus, ontology evaluation is an important step followed its development which includes assessing the usefulness of the ontology for the purpose it was built for and evaluating the quality of the ontology (its conceptual coverage, clearness, etc.). Evaluating ontology is not an evidence of the absence of problems, but it will make its use safer.

Our ontology evaluation is limited to the criteria identified by Gómez-Pérez [2001] such as: completeness, consistency, conciseness, and expandability.

Completeness: all knowledge that is expected to be in the ontology is either explicitly stated in it or can be inferred.

Consistency: refers to weather a contradictory knowledge can be inferred from a valid input definition.

Conciseness: if the ontology is free from any unnecessary, useless, or redundant definitions.

Expandability: refers to the ability to add new definitions without altering the already stated semantic.

Different ontology evaluation approaches have been considered in literature depending on the purpose of the evaluation and the type of the ontology being evaluated. Brank and colleagues [2005] classify ontology evaluation approaches as following:

1. Those based on comparing the ontology to a "golden standard" which might be an ontology itself;
2. Those based on using the ontology in an application and evaluating the results or application-based ontology evaluation;
3. Those involving comparison with a source of data (e.g. a collection of documents) about the domain to be modeled by the ontology;
4. Those where evaluation is done by humans who try to assess how well the ontology meets a set of predefined criteria, standards, requirements, etc.

The first approach is not applicable due to the lack of a "golden standard" or upper Software Engineering ontology.

The second approach has been adopted and an application-based ontology evaluation was conducted using a prototype system which was implemented for this purpose [Bajnaid et al., 2012].

The third approach was held during development of the ontology when the evolving conceptual model [Bajnaid et al., 2012] was compared to the sources of knowledge.

The fourth approach included usage of the ontology assessment questionnaire which was distributed among some Specialist Groups of well-known communities to validate the quality of the ontology.

4. Conclusion

A well-defined, complete and disciplined SQA process can be helpful to improve communication and collaboration among project participants and can serve as a standard when there is a disagreement. To our knowledge, there is no software quality ontology available for teaching and learning purposes. Having the opportunity to build dynamic ontology reasoning rules will provide a unique insight in teaching software quality in an e-learning environment. The quality of the ontology was validated against several criteria. The consistency and conciseness of the developed ontology were automatically validated during the implementation process using the Protégé consistency checker tool. A proof of concept e-learning prototype was built to evaluate the SQA ontology deployment. [Bajnaid et al., 2012].

5. References

[1] Calero, C., Ruiz, F. and Piattini, M., 2006. *Ontolgies in Software Engineering and Software Technology*, Springer

[2] Mendes, O., and Abran, A., 2004. Software Engineering Ontology: A Development Methodology, Position Paper, *Metrics News* 9:1, August, pp. 68-76.

[3] Wille, C., Dumke R., Abran A. and Desharnais J.M., 2004. E-learning Infrastructure for Software Engineering Educations: Steps on Ontology Modeling for SWEBOK, Proceedings of the IASTED International Conference on Software Engineering, 2004, pp. 520-525.

[4] Uschold, M., and Gruninger, M., 1996. Ontologies: Principles, Methods, and Applications, *Knowledge Engineering Review*, Volume 11 number 2, June 1996.

[5] Zhao Yajing, Dong Jing, Peng Tu, 2009. Ontology Classification for Semantic-Web-Based Software Engineering, *IEEE Transactions on Services Computing*, v.2 n.4(2009), 303-317.

[6] SWEBOK, 2004. Guide to the Software Engineering Body of Knowledge, ed. Bourque P., and Dupuis R. IEEE Computer Society Press, 2004. Available at: http://www.swebok.org

[7] Bajnaid N., Benlamri R. and Cogan B., 2011. Context-Aware SQA E-learning System, Proc. of the *Sixth International Conference on Digital Information Management ICDIM 2011*, Melbourne, Australia, 26-28 Sept., 2011.
[8] Vrandečić, D., 2010. *Ontology Evaluation*, PhD Thesis
[9] Gómez-Pérez A (2001) Evaluation of Ontologies. International Journal of Intelligent Systems 16(3), p.391–409
[10] Brank, J., Grobelnik, M. and Mladenic, D. 2005 A survey of ontology evaluation techniques. In Proceedings of the Conference *on Data Mining and Data Warehouses (SiKDD 2005)*, Ljubljana, Slovenia.
[11] Bajnaid N., Benlamri R. and Cogan B. (2012), "An SQA e-Learning System for Agile Software Development", Proc. of the Fourth International Conference on Networked Digital Technologies, Dubai, UAE, April 24-26, 2012. Communications in Computer and Information Science(CCIS 7899) Series of Springer LNCS – in press.

On the dimensions of software documents – An idea for framing the software engineering process

E. Kindler, H. Baumeister, A. Haxthausen, and J. Kiniry
DTU Informatics
DK-2800 Lyngby, Denmark
{*eki,hub,ah,jkin*}*@imm.dtu.dk*

***Keywords*-software engineering documents; software development process; software engineering theory;**

I. Introduction

In a call for action [1], Jacobson, Meyer, and Soley, together with many other signatories, encourage the software engineering discipline to "re-found" software engineering based on a "solid theory" [2]: The SEMAT initiative. In a soon to be published book [3], the "The Essence of Software Engineering" is presented by "Applying the SEMAT Kernel". This SEMAT kernel identifies the essential concepts or "things" that need to be kept track of in order to successfully develop software, the so-called *alphas* (α). This way, SEMAT conceptualizes the "things" going on in the software development process, independently from a specific software development approach, methodology or philosophy. The alphas allow us to talk about what things need to done and monitored, discussed and taught in software engineering independently from how they are done in a specific development approach. This agnostics when identifying the alphas is one of the strength of SEMAT's conceptualisation.

Surprisingly enough, the artefacts that are used for software development seem not to be of primary concern in SEMAT: documents describing the software in some form or the other. In this paper, we understand *software documents* in the broadest possible sense, which would subsume single paragraphs with the product objective, product definitions, systems specifications, source code, binary code, tests (executable and not), all kinds of UML and non-UML diagrams, formal models, user stories, GUI definitions, and handbooks; in short, any written or graphical artefact we encounter during the software development process (be it on paper or in electronic form).

We can only guess as to why software documents do not play a more prominent role in the SEMAT kernel; one reason might be that discussing any of these software documents specifically, would introduce a bias towards some specific development approaches – SEMAT would not be agnostic anymore. When discussing specific documents – and in particular when defining specific structures and how they should be written – we might introduce a bias towards how things should be done, and this way towards a specific software development philosophy.

Still, we believe that software documents are way too important not to be a primary concept of a theory of software engineering. In this paper, we will have a first glance at the space of software documents and their characteristics – independently from a specific software development philosophy. In order to understand this space, we identify some first dimensions that span the space of all software documents with their different characteristics; we give a glimpse of how these dimensions could be used to better understand what should be done during the software development process, which, in particular, would help teaching software engineering. Moreover, the way and order in which different software development approaches create documents with their specific characteristics in this space – i. e. the project's software document trajectory in this space – might characterise specific software development approaches and provide insights into the way they work.

In this paper, we will discuss some ideas of how this could look like. This paper, however, does not provide the answer yet – we do not even dare to fix the most essential dimensions yet. The dimensions and examples discussed in this paper, should demonstrate that it is worthwhile investigating the dimensions, and that, eventually, these dimensions could be an ingredient to the theory of software engineering.

II. Dimensions and their purpose

Next, we discuss some first candidates for some of the dimensions of software documents, and how they reflect on the development process.

A. Some dimensions

Figure 1 depicts three dimensions, which – from our teaching experience – seem to be important for software development. For lack of a better name[1], we call the first one the *"What-How" dimension*; the idea of this dimension is that in the early phases it should be defined "what" the final software product should do, in contrast to "how" this

[1] A Sofa Seminar discussion of the Software Engineering Section of DTU Informatics resulted in a proposal to call this dimension the Abbott-Costello dimension after the famous "Who's on first?" performance from 1945.

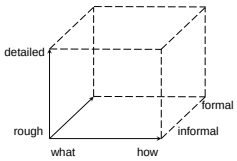

Figure 1. Three dimensions of software documents

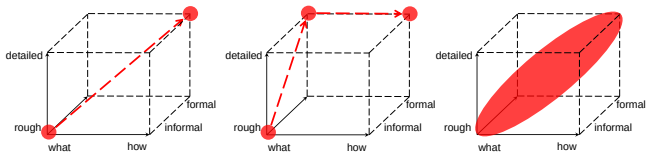

Figure 2. Process trajectories

is finally technically realized and implemented. The second dimension is *level of detail*, which runs from "rough" to "detailed" – we will see later, that the *level of detail*, probably, can be decomposed into two independent dimensions. The third and, for now, last dimension is *formality*, which runs from "informal" to "formal". Also for *formality*, it appears that it can be decomposed – at least its is entangled with another dimension, *executability* (see Sect. II-C).

Of course, there are more dimensions; which ones are the most relevant and helpful ones, is still an open issue. For getting a grip on the issue, we will start some form of wiki or open document, where all interested people could contribute their perspective. A reasonable schema for defining a dimension could consist of a *name*, an (informal) *definition* or characterisation, and a "*litmus test*" for identifying on which side of a dimension a software document would be located; in some cases, there could be even some *metrics* for measuring documents with respect to the dimension; most importantly, there should be a set of *examples* that show which kind of document would be at which end of the resp. dimension. For example, the "product objective", which typically is a single sentence or paragraph of what should be achieved with the product, would be about the "what", "informal", and "rough"; by contrast, the handbook would be about the "what", more or less "formal", but "detailed". The result of an object oriented analysis would still be about the "what", would be more "formal", and more "detailed". The code – remember that we also consider code as a document – would be about the "how", would be "formal", and "detailed". It is a worthwhile exercise to place more kinds of software documents in this space.

B. Dimensions and development process

Now, let us have a brief look at how software engineers would navigate through the space of software documents. Figure 2 shows three cases. The left one, is where there might be a *rough product idea* or *objective* initially, and an *implementation* finally. The middle once shows one iteration of agile development: it goes from an initial *user story*, via an (automated) *test*, to the final *impelementation*. The right one, shows a more waterfall-like process, which more systematically covers all stages. Being agnostic about the process, we do not prefer one over the other – it certainly depends on the kind and size of software what is better. Anyway, Fig. 2 suggest that the trajectory of a process in the space of software documents tells something about the underlying process – for the left one, there almost certainly is no handbook, since this would imply that the "detailed" "what" area is covered.

As mentioned, the middle trajectory shows one iteration of a agile development process only. When going through different iterations the collected user stories and implementation would cover more and more features of the final software. This observation, actually gives rise to another dimension: *coverage*, which is not yet shown in Fig. 2.

C. More dimensions and entanglement

The *coverage* mentioned above seems to introduce another dimension of software documents (or in the case of agile a collection of documents). Somehow this is related to the *level of detail* – just organized according to the product's features or functions. The *level of detail* seems to have two independent components: *coverage* and *abstraction*, which however needs more investigation.

Likewise, there are other dimensions like *non-executable/executable*, which, however, is entangled with (i. e. is not full independent of) *formality*, since *executability* implies some form of *formality*. And there are more dimensions, that should be discussed before ultimately deciding on *the dimensions* of software documents: "textual/graphical", "imprecise/precise", etc.

III. Conclusion

In this paper, we gave a glimpse of the dimensions of software documents – barely enough to see that it might be a worthwhile endeavour to better understand these dimensions, which then could be a part of software engineering theory. In this endeavour, existing characterisations of kinds of software documents such as the one discussed by Bjørner [4] should be taken into account.

References

[1] I. Jacobsen, B. Meyer, and R. Soley, "The SEMAT initiative: A call for action," Dr. Dobb's Journal, Dec. 2009, http://www.ddj.com/architect/222001342.

[2] I. Jacobsen and I. Spence, "Why we need a theory for software engineering," Dr. Dobb's Journal, Oct. 2009, http://www.ddj.com/architect/220300840.

[3] I. Jacobsen, P.-W. Ng, P. E. McMahon, I. Spence, and S. Lidman, *The Essence of Software Engineering*. Addison Wesley, 2013, pre-publication draft.

[4] D. Bjørner, *Software Engineering 1*. Springer, 2006.

Linear Software Models

Extended Abstract

Iaakov Exman
Software Engineering Department
The Jerusalem College of Engineering
Jerusalem, Israel
e-mail: iaakov@jce.ac.il

***Abstract*— Modularity is essential for automatic composition of software systems from COTS (Commercial Off-The-Shelf) components. But COTS components do not correspond exactly to the units and functionality of the designed software system architecture modules. One needs a precise composition procedure that assures the necessary and sufficient components to provide the required units. Linear Software Models are rigorous theoretical standards subsuming modularity. The theory uses a Modularity Matrix which links independent software structors to composable software functionals in a Linear Model.**

Keywords: Software Composition, Linear Software Models, Well-composed Systems, Modularity Matrix.

I. INTRODUCTION

The software composition problem, dealt with in this work, is how to build a well-designed modular software system from available COTS components that were not designed specifically for our particular system.

This work describes Linear Software Models as a theory of software composition. In this theory, the architecture of a software system is expressed by two kinds of entities: structors and functionals.

Structors are architectural units, from the structural point of view. *Functionals* are architectural system units from a behavioral point of view. Functionals can be, but are not necessarily invoked.

II. THE THEORETICAL MODEL

A Linear Software Model contains a list of software structors and another list of software functionals. Its Modularity Matrix is defined as a Boolean matrix which asserts links (1-valued elements) between software functionals (rows) and software structors (columns).

Assuming linear independence of structors and of the corresponding functionals, one can prove the following results about a system's Modularity Matrix:

- it is Square;
- it is Reducible, i.e. it can be put in block-diagonal form.

The software system modules at a certain level of the software hierarchy are represented by the blocks of the block-diagonal matrix.

The theory allows formalization of commonly used concepts of Software Engineering. For instance, *coupling* just means lacking linear independence.

One can express modularity quantitatively by the diagonality of a Modularity-Matrix *M*. It tells us how close the matrix *1-valued* elements are to the main diagonal. Our proposed definition of diagonality is the difference between the matrix Trace and *offdiag*, a new term dealing with off-diagonal elements:

$$Diagonality(M) = Trace(M) - offdiag(M)$$

Proofs of the theory results and more detailed expressions, say for the *offdiag* terms, are found in our longer paper [1].

III. CANONICAL CASE STUDIES

In order to corroborate the theory, we tested it by applying it first to small canonical systems and then to larger real software systems.

The first canonical system was the KWIC system described in the classical paper by Parnas [2]. Parnas suggested two different modularizations of such a system and showed by informal argumentation that one of the modularizations is better than the other one.

We have taken the data from Parnas' paper and formulated the Modularity matrices for Parnas' modularizations. Using the above definition of diagonality, its value was calculated for both modularizations. We obtained the same results of Parnas' paper by formal means.

Another canonical system was the *Observer* design pattern taken from the Design Patterns' GoF book [3]. The *Observer* Design Pattern abstracts one-to-many interactions among objects, such that when a "subject" changes, its attached "observers" are notified and updated.

The Observer Modularity Matrix was obtained from the sample code in the GoF book, referring to an analog and a digital clock, the "concrete observers", following an internal clock, the "concrete subject".

Row/column reordering of a quite arbitrary initial matrix causes modules to emerge in a strictly Linear-Reducible

matrix. Meaningful subject, observer and clock application modules emerged from basic structors.

The Observer analysis illustrates that, despite arbitrary initial order, automatic reordering brings about a matrix accurately reflecting the pattern functionality.

IV. Larger Software Systems

We further tested the theory by applying it to larger real software systems found in the literature.

A typical example is the NEESgrid "Network Earthquake Engineering Simulation" project. It enables network access to allow participation in earthquake tele-operation experiments. The system was designed by the NCSA at University of Illinois.

Modularity Matrix functionals for NEESgrid were extracted from a report [4] with exactly 10 upper-level structors. An initial modularity matrix was obtained with sparse scattered non-zero elements typical of initial matrices in this kind of analysis.

Pure algebraic row/column reordering, without prior semantic knowledge, brought about an almost block-diagonal Matrix, easily amenable to meaningful interpretation.

The significant result, common to large case studies, is that there are few outliers, and all of them are in columns/rows adjacent to the Linear Model blocks. This is what we call bordered Linear-Reducible.

These outliers point out to possible improvements of the software system design.

V. Discussion

The main contribution of this work is the Linear Software Models. These are theoretical standards against which to compare actual software systems. The models stand upon well-established linear algebra, as a broad basis for a solid theory of composition – beyond currently accepted principles and practices.

One can assert, from the Modularity Matrix properties of a system, which structors are independent and which functionals are independently composable. One can then infer which design improvements are desirable.

This view is very different from *design* models, such as UML, whose purpose is not to serve as theoretical standards. Design models freely evolve with design and system development. Design models have indefinite modifiability to adapt to any system, in response to tests of system compliance to design.

VI. Related Work

Matrices have been proposed and used to deal with modularity. A prominent example is DSM (Design Structure Matrix) proposed by Steward [5], and developed by Eppinger and collaborators, see e.g. [6].

Linearity, not found in DSM, is the outstanding feature of our standard models. Moreover, our modularity matrices display structor to functional links, while both DSM matrix dimensions are labeled by the same structures.

In practice, our modularity matrices may be much more compact than DSM.

Although diagonality has seldom been calculated within the context of modularity, such formulas have appeared in other contexts. Our *offdiag* definition is better suited to modularity than alternative definitions found in the literature.

VII. Conclusion

Software engineering has been perceived as essentially different from other engineering fields, due to software's intrinsic variability, implied by the *soft* prefix. This versatility is seen as an advantage to be preserved, even though software composition has largely resisted theoretical formalization.

We have shown that Linear Software Models can be formulated, without giving up variability. Thus, software systems of disparate size, function and purpose, may have *Linearity* in common.

References

[1] I. Exman, "Linear Software Models for Well-Composed Systems", in S. Hammoudi, M. van Sinderen and J. Cordeiro (eds.), Proc. 7th ICSOFT'2012 Conference, pp. 92-101, Rome, Italy, July 2012.

[2] D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", Comm. ACM, Vol. **15**, pp. 1053-1058, 1972.

[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, USA, 1995.

[4] Finholt, T.A., Horn, D., and Thome, S., "NEESgrid Requirements Traceability Matrix", Technical Report NEESgrid-2003-13, School of Information, University of Michigan, USA, 2004.

[5] Steward, D., "The Design Structure System: A Method for Managing the Design of Complex Systems", IEEE Trans. Eng. Manag., **EM-29** (3), pp. 71-74, 1981.

[6] Sosa, M.E., Eppinger, S.D., and Rowles, C.M., "A Network Approach to Define Modularity of Components in Complex Products", ASME Journal of Mechanical Design, **129**, 1118, 2007.