Generating a Useful Theory of Software Engineering

Steve Adolph
Electrical and Computer Engineer
University of British Columbia
Vancouver, Canada
stevea@ece.ubc.ca

Philippe Kruchten
Electrical and Computer Engineer
University of British Columbia
Vancouver, Canada
pbk@ece.ubc.ca

***Abstract*—We argue a theory of software engineering must be useful to practitioners and explain the phenomena they are experiencing. Useful theories of software engineering can be generated empirically using methods such as grounded theory. We present our research and others as examples of how useful theory can be generated.**

***Index Terms*—theory, software engineering, grounded theory, social processes**

I. Introduction: Why a Theory of Software Engineering

Why should we be interested in creating a theory of software engineering? We should be interested because theory is the foundation of science and provides explanations of phenomena as well as testable predictions. Engineering originally emerged as a profession in response to the failure of constructed works such as collapsing bridges and exploding boilers [1]. Engineering disciplines apply theory to create theoretical models that predict the behaviour and performance of an engineered system before it is built. Our great grandparent engineers used such theoretical models to predict the load a bridge could support or the pressure a boiler could withstand before they were built.

Software engineering emerged forty years ago during a NATO scientific conference [2] in a response to the realization that large software systems were frequently delivered late – if delivered at all – were over budget with feature deficiencies, and had suspect quality. Mature academic disciplines have a well-established base of theories that codify and rationalize well understood knowledge, and provide a basis for identifying and interpreting new work [3]. The "Holy Grail" of software engineering is developing theories that enable us to predict the outcome of interventions on software engineering efforts.

In this short position paper, we state our agreement with the assertion that if software engineering is to be taken seriously as an engineering discipline then there must exist a set of theories that can explain software engineering phenomena. We argue that theories of software engineering must be useful to practitioners and useful theories will explain social phenomena. We present grounded theory as one approach for creating useful theories that are generate empirically from data. Finally, we present our research as an example of how a useful theory of software engineering can be generated.

II. Are Software Engineering Theories Useful?

Engineering is applied science and a theory of software engineering must be useful to practitioners. Practitioners will not use something if they do not perceive it as useful or if it does not provide a useful explanation of the phenomena they are experiencing. A case in point: our research was motivated by trying to understand why software engineers are reluctant to adopt software methodologies despite evidence software methodologies improve outcomes. Our conclusion was engineers do not find methodologies useful because the mechanistic reductionist models most software methodologies are based on limit, and even impede the way individuals actually create software [4].

Parnas and Clements [5] characterized the well intentioned search for a theoretically based rational software method as the search for the Philosopher's Stone of software engineering; a rational software process that would give us a good reason for doing what we're doing. Their vision for software methodology was to "*ideally derive our programs from a statement of requirements in the same sense that theorems are derived from axioms in a published proof*". Parnas and Clements also stated clearly that the Philosopher's Stone did not exist because the complexity of software makes an ideal rational process impossible. Despite this, they argued it is still beneficial to approximate or, as they put it, "fake" a rational process.

Most software methodologies try to approximate or "fake" a rational process by following a reductionist framework describing the successive transformation of work products as they flow from one stage to the next [6]. The classic waterfall embodies this reductionist framework and models software development as a linear series of processes where the outputs of the prior process flow down as inputs to the next [7].

While modern software engineering methodologies have moved towards more iterative and incremental models of software development, such as the Unified Process (UP) [8] and its variants, they still model software development as a technical process of reductionist transformation. Each process stage can be described in terms of an Input-Process-Output model (IPO), where inputs arrive and are then processed by transforming them into outputs. This model enables methodology designers to specify the inputs to a process, the outputs, the quality of those outputs, the steps for transforming the inputs into outputs, and the individuals responsible for

animating the process. The IPO model for software methodologies is standardized in the *Software Process Engineering Meta-model* (SPEM) [9]. Many of the modern UP software methodologies, including the Rational Unified Process (RUP[1]) and OpenUP, are built using SPEM as a meta-model.

The problem with software methodologies and the efforts to standardize their definition is it seems no one actually follows them unless coerced to do so [10-13] Time and time again, when we ask developers why they do not follow a corporate software development methodology, or adopt a "rational" methodology, the response we get is *"…because if we ever did it like that we would never get it done*".

In contrast to Parnas' idealized "faked" rational process, our observation of the methodology-in-use is that it is usually an ad hoc collection of practices agreed to on a situational basis by those immediately engaged in the software development process. The "faking" part is more often the arbitrary and coercive creation of work products (e.g. detailed documentation) that produces a veneer or fiction of a rational process that conceals the irrational process [12]. Furthermore, the strategy of "faking" a rational process is not serving development teams well in practice, and Nandhakumar and Avison concluded from their study that "*formal methodologies are too mechanistic to be of much use in the detailed, day-to-day organization of developers activities*". Further, they asserted the risk of faking such a process is that "*Seeking to impose methodologies to improve the productivity of the developers' task may be counter productive*".

The IPO model behind most software methodologies simply does not address the issues that are relevant to the practitioner or explain the phenomena they are experiencing. Despite their derivation from theoretical input-process-output models, software methodologies are not used because they do not address the main concerns of practitioners. The lesson is if we do not address the main concerns of practitioners then any theory of software engineering will not be useful and remain just an academic curiosity.

III. Software Development As a social Process

It is immediately clear to an observer that software development is a highly social process. Other researchers have made the same observation: "*Software Engineering is primarily a social and creative process, where the creativity, skill, and co-operation of developers, users, and procurers determine the quality and effectiveness of the developed software*" [14]. The social processes are not a side effect of the software development process; the social processes are the control mechanism – as was highlighted by Nandhakumar and Avison's *"The evidence suggests that social controls, such as norms promoting collaboration with colleagues, professional design practices and established routines appeared to be a more significant influence on developers' work practice at LMC than the requirements of a methodology"* [12]. Put in modern agile shorthand: "*people trump process*" [15].

Over thirty years ago Boehm [16] reported the influence of social factors on the outcome of software engineering efforts: "*Personnel attributes and human relations activities provide by far the largest source of opportunity for improving software productivity*". If social factors are the biggest cost drivers and explain variances in the productivity of software development teams, then theories that helps us identify and understand social processes in software engineering should be useful and yield significant benefit to the industry.

IV. Generating Useful Software Engineering Theory

While a grand theory of software engineering may elude us because of the complexity of software and the dominant influence of social processes on software engineering, we can still generate useful theory that practitioners can use to understand and explain their situation. Grounded theory is one tool available to us to generate substantive theories that explain what is going on. A Grounded Theory is a set of integrated conceptual hypotheses systematically generated to produce a theory about a substantive area [17]. Glaser and Strauss [18] proposed the method as having application for both qualitative and quantitative data. When working with qualitative data, and when compared to other qualitative research methods such as narrative or ethnography, Grounded Theory generates a substantive theory that explains participants' behavior as a set of integrated hypotheses.

Co-discoverers Barney Glaser and Anselm Strauss called the method "grounded" because a theory is systematically obtained from a broad array of data through a rigorous process of constant comparison – it is 'grounded' in the data [18, 19]. Grounded Theory differs from logico-deductive methods of inquiry because, rather than developing a theory without relying on data and then systematically seeking out evidence to verify the theoretical constructs, researchers using Grounded Theory set out to gather data and then systematically develop a mid-level substantive theory derived directly from the data [18]. The goal of Grounded Theory is to generate concepts and categories that account for a pattern of behavior which is relevant and problematic for those involved [20].

We used Grounded Theory in our field study of practitioners to generate a substantive theory of how people manage the process of software development. We learned the main concern of people involved in the process of software development is getting the job done and different points of view and expectations create impediments. People use a four stage process of Reconciling Perspectives [21] to remove these impediments. When a perspective mismatch is discovered, people converge their mismatched perspectives by reaching out and negotiating a consensual perspective. Constructing the job work products and evaluating them validates the consensual perspective. The process may yield accepted work products, providing objective evidence the perspective mismatch was reconciled, or the process may result in waste.

Reconciling Perspectives is a social process moderated by social dynamics, and creating accepted work products depends

[1] RUP is a registered Trademark of IBM Corporation

on individuals' abilities to reach out and engage in negotiations while also sheltering themselves from distracting interruptions. This creates a tension in the process that must be managed because to discover and converge perspective mismatches, people must be open to interruptions while also remaining focused on getting the job done.

What Reconciling Perspectives highlights is the need for managed communications and the roles leaders (or "critical individuals") must play to manage communications. When we explained the theory to practitioners a common comment was "*yeah, that's my life!*" From Reconciling Perspectives we were able to explain the role of communications in software projects and that more is not necessarily better. We were able to generate a set of recommendations for practitioners to help them better understand their communications patterns and appropriate interventions leaders could take to avoid the creation of waste. Furthermore, Reconciling Perspectives is generated from field data and therefore gains an element of "street credibility" with practitioners that a logico-deductive approach may not enjoy.

Other researchers have previously taken a similar approach of using grounded theory to explain software engineering phenomena. Ferreira and colleagues' [22] study explains how two different communities of developers user interaction designers and agile software developers integrate their work and describes a process similar to Reconciling Perspectives. Whitworth's and Biddle's study of XP teams explains some of the social processes that contribute to the team's success [23]. Their findings suggest agile practices were seen by the study participants to increase the ability for team members to work together with others in an environment of constant feedback and progress. Hoda, Noble, and Marshall's [24] investigation of self-organizing teams explains the influence of individuals in the roles they play facilitating the organizing of self organizing teams.

This empirical approach to theory generation opens the opportunity to create a collection of mid-level theories that can explain the main concern of individuals and teams and how they are resolving their main concern. While a collection of mid-level theories that explains what is going on may not be as viscerally satisfying as an elegant logically derived theory, such a collection will nevertheless be useful and perhaps better capture the messiness of the socio-technical systems that characterizes software development. Empirically generated theories capture what people are really doing rather than what experts believe they should be doing.

V. The Search for a Useful Holy Grail

The effort to find a theory of software is much like the legendary search for the Holy Grail: a doomed and yet honourable test of courage. The search for the Grail was in reality a quest to develop and demonstrate moral leadership in a time when people needed a way to teach moral principles. We believe the search for a theory of software engineering is an honourable one because we need to be able to make decisions and take interventions on the basis of some useful theoretical model rather than "gut feelings" and hearsay.

We certainly do not believe that attempting to incorporate social practices into a methodology following a formal meta-model such as SPEM (will work, or even that it is desirable. Our failure with OpenUP [25] is a clear demonstration of this. Instead, we recommend employing an empirical approach, following the principles of validated learning, and what has widely become known as "Lean Start-up" [26], to design and deploy a socio-technical methodology. A good start would be using Reconciling Perspectives to explain and highlight the social aspects of existing software methodologies such as Scrum. Feedback from practitioners can then be used to design socio-technical methodologies with greater scope.

References

[1] D. Parnas, "Licensing software engineers in Canada", *Commun. ACM,* vol. 45, pp. 96-98, 2002.

[2] P. Naur and B. Randell, Eds., *Software Engineering: Report of a conference sponsored by the NATO Science Committee* Garmisch, Germany: Brussels, Scientific Affairs Division, NATO (1969), 1968

[3] T. Hall, N. Baddoo, S. Beecham, H. Robinson, and H. Sharp, "A systematic review of theory use in studies investigating the motivations of software engineers," *ACM Trans. Softw. Eng. Methodol.,* vol. 18, pp. 1-29, 2009.

[4] S. Adolph, "Reconciling Perspectives: How People Manage the Process of Software Development," PhD, Electrical and Computer Engineering, UBC, Vancouver, 2012.

[5] D. Parnas and P. Clements, "A rational design process: How and why to fake it Formal Methods and Software Development." vol. 186, H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, Eds., ed: Springer Berlin / Heidelberg, 1985, pp. 80-100.

[6] D. G. Wastell, "The fetish of technique: methodology as a social defence," *Information Systems Journal,* vol. 6, pp. 25-40, 1996.

[7] W. Royce, "Managing the Development of Large Software Systems," in *Proceedings of the International Conference on Software Engineering*, 1970.

[8] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*: Addison-Wesley, 1998.

[9] OMG. (2005, Software Process Engineering Metamodel Specification. Available: www.omg.org/docs/formal/05-01-06.pdf

[10] B. Fitzgerald, "The use of systems development methodologies in practice: a field study," *Information Systems Journal,* vol. 7, pp. 201-212, 1997.

[11] M. Huisman and J. Iivari, "Deployment of systems development methodologies: Perceptual congruence between IS managers and systems developers," *Information & Management,* vol. 43, pp. 29-49, 2006.

[12] J. Nandhakumar and D. E. Avison "The fiction of methodological development: a field study of information systems development," *Information Technology & People.,* vol. 12, p. 176, 1999 1999.

[13] J. Ward, P. Taylor, and P. Bond, "Evaluation and realisation of IS/IT benefits: an empirical study of current practice," *Eur J Inf Syst,* vol. 4, 1996.

[14] A. Fuggetta, "Rethinking the modes of software engineering research," *Journal of Systems and Software,* vol. 47, pp. 133-138, 1999.

[15] A. Cockburn and J. Highsmith, "Agile software development, the people factor," *Computer,* vol. 34, pp. 131-133, 2001.

[16] B. W. Boehm, *Software engineering economics*: Prentice-Hall, 1981.

[17] B. Glaser and J. Holton. (2004, Remodeling Grounded Theory. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research in Nursing & Health 5(2)*. Available: http://www.qualitative-research.net/fqstexte/2-04/2-04glaser-e.htm

[18] B. Glaser and A. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Chicago Illinois: Aldine, 1967.

[19] B. G. Glaser, "The Constant Comparative Method of Qualitative Analysis," *Social Problems,* vol. 12, pp. 436-445, Spring 1965.

[20] B. G. Glaser, *Theoretical Sensitivity*. Mill Valley, California: Sociology Press, 1978.

[21] S. Adolph, P. Kruchten, and W. Hall, "Reconciling perspectives: A grounded theory of how people manage the process of software development," *Journal of Systems and Software,* vol. 85, pp. 1269-1286, 2012.

[22] J. Ferreira, H. Sharp, and H. Robinson, "Agile Development and User Experience Design Integration as an Ongoing Achievement in Practice," presented at the Agile 2012, Grapevine, Texas, 2012.

[23] E. Whitworth and R. Biddle, "The Social Nature of Agile Teams," in *AGILE 2007*, 2007, pp. 26-36.

[24] R. Hoda, J. Noble, and S. Marshall, "Organizing self-organizing teams," presented at the Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, Cape Town, South Africa, 2010.

[25] P. Kroll and B. MacIsaac, *Agility and Discipline Made Easy: Practices from OpenUP and RUP*: Addison Wesley, 2006.

[26] E. Ries, *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*: Crown Publishing Group, 2011