Sheffield Hallam University

Re-thinking software engineering approaches : a critical reflection on theory building.

BARN, Balbir S and CLARK, Tony <http://orcid.org/0000-0003-3167-0739>

Available from Sheffield Hallam University Research Archive (SHURA) at:

http://shura.shu.ac.uk/11989/

This document is the author deposited version. You are advised to consult the publisher's version if you wish to cite from it.

Published version

BARN, Balbir S and CLARK, Tony (2011). Re-thinking software engineering approaches : a critical reflection on theory building. In: ICSOFT 2011 : 6th International Conference on Software and Data Technologies, Seville, Spain, 18-21 July 2011. 59-64.

Copyright and re-use policy

See http://shura.shu.ac.uk/information.html

Sheffield Hallam University Research Archive
http://shura.shu.ac.uk

RE-THINKING SOFTWARE ENGINEERING APPROACHES: A CRITICAL REFLECTION ON THEORY BUILDING

Balbir S.Barn, Tony Clark
Middlesex University, The Burroughs, London, UK
b.barn@mdx.ac.uk, t.n.clark@mdx.ac.uk

Keywords: Theory Building, Modeling, Software Engineering

Abstract: This paper re-appraises Peter Naur's influential paper on Programming as Theory Building in the context of modern software engineering practice. The central argument is that such practice is focussed primarily on methods, notations, lifecycles and the description of artifacts such as models. Instead we propose that a theory building view is more appropriate, and that the concept of a theory should underpin a software design process which then calls for new tools and a new research agenda.

1 INTRODUCTION

This account sets out to re-appraise Peter Naur's influential paper on Programming as Theory Building (Naur, 1985) in the context of software engineering practice and in particular the model building focus of the last few years. We contend that software engineering in its efforts to establish itself as a science has focussed exclusively on methodology, lifecycles, processes and the description of artifacts. This we believe has lead to an number of emergent issues notably, there are too many methods, the processes of software design are bloated and a focus on the production of artifacts without evaluation of the goodness of artifacts. These points were first raised by Naur and we propose a return to a more fundamental re-think originating from a re-appraisal of Naur's work on theory building. In doing so, we suggest that making theory building the core unit of focus in the software process will create new insights into the design of software and open avenues of research that are rooted in the Philosophy of Science.

The starting point for this work has been triggered by the extent of activity that is currently surrounding Enterprise Architecture. As systems supporting business become increasingly more significant and complex an important approach to management and planning of systems that has gained prominence is model-based Enterprise Architecture (EA). EA has its origins in Zachman's original EA framework (Zachman, 1999) while other leading examples include the Open Group Architecture Framework (TOGAF) (Spencer et al., 2004) and the framework promulgated by the Department of Defense (DoDAF) (Wisnosky and Vogel, 2004). In addition to frameworks that describe the nature of models required for EA, modeling languages specifically designed for EA have also emerged. One leading architecture modelling language is Archimate (Lankhorst et al., 2010).

Central to enterprise architecture is the notion of development and presentation of models. Given the plethora of models available two concerns of note arise: Firstly, given the range of models available, it is difficult to ascertain why a particular model is relevant and preferable over others. This arises from a lack of clarity of the link between the contents and structure of a model on one hand and its purpose on the other (Johnson et al., 2004). If the general purpose of models is to answer questions then we need to have a clear understanding of why we use the models we do. Secondly, evaluation of quality of models in general, and therefore EA models, is relatively under researched. While there are international standards for software systems there is "little agreement among experts as to what makes a "good" model" (Moody, 2005). Partly this may be attributed to a relatively immature field, or more likely, the production of a model is a socio-technical event so evaluation is against a person's tacit

needs. Others assert a "good" model is not an inherent property rather it is how effective it is in supporting the analyses conducted against it. Each analysis or question of an EA model might require different perspectives or slices of information. Empirical measurements of the goodness of an EA model are generally lacking in the literature.

How these requirements are aligned to Naur's theory building view and whether the questioning the "why" of a model and its "goodness" is fundamental to assessing the efficacy of a model in representing knowledge of a domain. It is these aspects that we believe that a re-appraisal of Naur's ideas will provide a new insight.

The remainder of the paper is structured as follows: Section 2 outlines the main hypotheses posed by Naur and presents a more detailed analysis of aspects of some of the key issues raised by current software engineering practice (programs as models and methods). Section 3 presents our hypothesis of making theory building central to the software engineering process. Section 4 presents the main conclusions and an indication of future research in this area.

2 REFLECTIONS ON NAUR

Peter Naur wrote "Programming as Theory Building" in 1985, it was reprinted later in his collection of works, Computing: A Human Activity in 1992 (Naur, 1992). The paper presents a discussion that contributes to what programming is. While it is tempting to assume from the title of the paper that Naur is focused on the minutiae of programming, he is specific in that programming denotes the "whole activity of design and implementation" and thus his theory applies to the field of software engineering. The fundamental premise asserts that programming should be regarded as an activity by which programmers achieve a certain insight or theory of some aspect of the domain that they are addressing. The knowledge, insight or theory that the programmer has come into possession of is a theory in the sense of Ryle (Ryle, 1949). That is, a person who has a theory knows how to do certain things and can support the actual doing with explanations, justifications and responses to queries. That insight or theory is primarily one of building up a certain kind of knowledge that is intrinsic to the programmer whilst any auxiliary documentation remains a secondary product. Of particular interest, is how Naur explains the life-cycle of a program. Programs are created by the establishment of a theory, the maintenance of a program is dependent on the theory being transferred between programmers; and the program dies when the theory has decayed. Program revival is described as re-establishing the theory behind the program which cannot be done merely from documentation and should only be considered in exceptional circumstances as the cost of theory revival is prohibitive and the resulting theory may be different from that originally conceived.

In addition to the theory view of a program life cycle, he also directed criticism at the then significant emphasis of methods for program development. He claimed that the methods based on sequences of actions of certain kinds cannot lead to the development of a theory of the program because the intrinsic knowledge held by a human has no inherent division into parts nor an inherent ordering. Instead the person possessing the knowledge is able to present multiple viewpoints as responses to requests. Where methods were supplemented with notations or formalizations then these were treated as secondary items as the theory of a program is intrinsic and cannot be expressed. Thus: "...there can be no right method".

2.1 Programs as Models

Naur was concerned with programs, but Enterprise Architecture is concerned with the production of models of interconnected systems or components. Thus we need to explore the relationship between programs and models and use that as a basis for analysing the applicability of Naur's hypothesis in the context of Enterprise Architecture.

A major activity in software engineering and computer science in general is modelling and as Fetzer has noted "the role of models in computer science appears to be even more pervasive than has been generally acknowledged.."(Fetzer, 1999). A key feature of modelling is the existence of an isomorphic relationship between the parts of the model and the parts of the thing modelled at some level of abstraction. Smith whilst noting these different types of models emphasizes the nature and importance of "representation"(Smith, 1991) :

> "To build a model is to conceive of the world in a certain delimited way... Computers have a special dependence on these models: you write an explicit description of the model down inside the computer...".

Smith suggests this feature distinguishes computers from other machines because they run by manipulating representations. "Thus there is no computation without representation" (Smith, 1991, p, 360) If we pursue this analysis further: From Naur we can state that the program is a theory; from general computation principles, we can state: the program is a model.

This leads to the notion that there is an equivalence between program = theory = model. We might moderate this further by noting that a program is a representation of a slice of "the" theory. In general though, this blurring between programs, theories and models is confusing and inaccurate. While models may exhibit an isomorphic relationship with their subject matter, this relationship may not reveal the theoretical connections that allow the theory to be defended in the form of Ryle's definition of a theory. Ideally, then, the theory must be statable independent of the computer model. In an essay that predates Naur's paper but still based on a prevailing view of the time that programs are theories, James Moor notes:

> "My claim is that this is rarely, if ever, the correct response. Even if there is some theory behind a model, it cannot be obtained by simply examining the computer program. The program will be a collection of instructions which are not true or false, but the theory will be a collection of statements which are true or false. Thus, the program must be interpreted in order to generate a theory. Abstracting a theory from the program is not a simple matter for different groupings of the program can generate different theories. Therefore, to the extent that a program, understood as a model, embodies one theory it may well embody many theories."(Moor, 1978, p.221)

From this analysis arises two key concerns. Firstly, programs and models may have multiple theories and a program or model may not refer to the same theory. Secondly these theories must be state-able independent of the program or the model. Then, there is an additional dichotomy: Is a program a representation of one view of an aggregate theory or is the program a representation of a component theory of the aggregate theory? These complexities, in the case of Naur's Programming as theory perspective have implications, because if the program is the only vehicle through which a theory can demonstrate that requirements of the intended system have been met, then, that theory testing process comes too late in the system life cycle.

2.2 On Methods and Theory Building

Earlier we noted that Naur had reserved considerable criticism for methods. We develop this discussion further in this section. The tendency of methods research in the IS discipline is to propose algorithmic steps to analysing and designing solutions to problems. As Naur notes: "A method implies a claim that program development can and should proceed as a sequence of actions leading to a particular kind of documented result". In contrast, a theory building view holds that a theory "held by a person has no inherent division into parts and no inherent ordering". At large, IS/SE research is embarked on a journey based on epistemological foundations and as a consequence has mostly neglected *techne* (the technical know how of getting things done) and *phronosis* (wisdom derived from socialised practices) (Wyssusek, 2007). In a more generalised form, this has correspondence to the distinctions between explicit and tacit knowledge (Nonaka and Takeuchi, 1995) and Naur would seem to be arguing the case for methods research that suggests more attunement with the effects that methods may have in the education of programmers. That is, the creation and embedding of tacit knowledge rather than the production of artifacts representing explicit knowledge through an algorithmic process.

Naur cites a study of five different methods by Floyd et al (cited here for completeness (Floyd, 1984))where the key result that a system of rules will lead to good solutions is an illusion, what remains is *the effect of methods on the education of programmers*. Thus the use of methods may themselves not lead to a good design but the practice of the method may lead to a better innate ability for theory building. Much research in IS literature relates to methods for IS development but this point is absent until we consider the advent of what we might term as micro methods - such as the literature surrounding Design Patterns (Gamma et al., 1995). Here Naur seems to have predicted the advent of design patterns approaches to theory building: *"the quality of the theory built by a programmer will depend to a large extent on the programmers familiarity with model solutions of typical problems with techniques of description and verification and with principles of structuring systems consisting of many parts in complicated interactions."*.

If the quality of the theory that a programmer has developed will depend on the familiarity of the programmer with a specific class of problem and perhaps the clarity of the question to be asked then methods that create a familiarity with model solutions of typical problems will be more effective. This of course resonates with the shift towards design patterns, analysis patterns and even enterprise architecture patterns where there has been a focus on capturing explicit knowledge as pattern solutions to specific problems. The class of problem under consideration is a key issue. It is clearly true that some classes of programs are easier than others (clerical versus algorithmic). Clerical programs that require little or no algorithmic complexity are more amenable to automation and have less need for theory building. A good example

where program = model and methods to arrive at a theory of a program occurred when Texas Instruments and (later Sterling Software Ltd) developed the Information Engineering Facility (IEF) - a model driven environment that used an integrated set of diagrams and modeling notation to generate relatively low complexity applications such as air-line frequent flyer programs (Texas-Instruments, 1988) .

Another aspect of methods is the use of notations and languages. Here, Naur is quite damning. For him notations or formalizations are treated as secondary items as the theory is intrinsic and cannot be expressed. Thus: "...there can be no right method".

Given that the theory is intrinsic to the human much like tacit knowledge there may be more than one way of "knowing" how to ride a bicycle and as we can't explicate that knowledge, then these multiple theories may exist. This is consistent with the Popperian understanding that questions the notion of "the" theory. Naur however, in his discussions around program revival suggests that a program cannot be revived (i.e. its theory re-constituted) from axillary documentation alone. The programmers that originally conceived the theory must be present. Naur would seem to preclude the existence of the multiple views of theories. For us then something interesting emerges: the notion of theory evolution from the initial theory construction to the various viewpoints of theories. If such collections of theories are to be managed, then methods and emerging tools that support domain specific modelling are critical. One potential benefit from theory slicing using viewpoints, is that a viewpoint constructed for a model and a viewpoint constructed for a program can be compared as there is a common language underpinning all viewpoints. Such comparison and reasoning may provide a route to addressing the issue of the 'quality' or 'goodness' of a model. Figure 1 presents a conceptual model of emerging concepts and relationships around how multiple theories, programs and models relate.

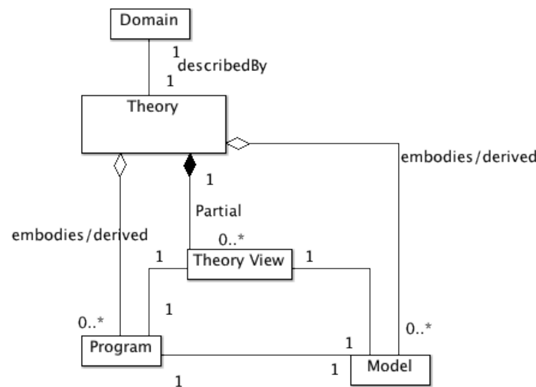

Figure 1: Domain theories, programs and models

3 THE PROPOSED APPROACH

The need for requirements or domain level understanding is critical as that is the start of all software engineering activity. A key aim of domain specific approaches is to provide abstractions that are meaningful to a domain expert and which shield the developer from having to work in terms of implementation concepts. However, technologies for performing domain specific modelling often focus more on the syntactic or structural features of the domain rather than their meaning. Following Naur we argue that the act of domain specific modelling involves the construction of theories that represent features from the application domain.

A theory is a set of statements about a domain. There is no limitation to the level of detail expressed by a theory, and in particular it is not limited to expressing functional features about an application. A theory should capture a description of a system from a particular stakeholder viewpoint and there may be many different theories about the same system. For example, a theory may describe the responses for a set of user operations. Another theory may describe features of the user interface in terms of usability and ergonomics. A further theory may describe non-functional requirements such as quality of service, resource cost, and operational resilience.

A theory must be able to determine which statements about the system are true and which are false. A theory should allow stakeholders to express theorems: statements about the system that may or may not be true, and be able to determine whether these statements are true or not.

An important feature of a theory is that it provides a language for expressing theorems and thereby allows stakeholders to specify both desirable and undesirable system properties. It must be possible to deduce whether a given theorem is true using the theory, although there is no requirement that the deduction is automatic.

Theories should be transformable so that one theory can be translated into another theory in a way that preserves the truth or otherwise of the theorems. This is important to be able to work at an appropriate level of abstraction (a domain specific theory) and then to embed the theory into a theory about an implemen-

tation technology (semantics preserving code generation).

Theories should be compositional so that different stakeholder views can be merged into a single theory. It is important that the composition mechanisms can determine whether or not the resulting theory is consistent, i.e. that no theorems in the component theories are contradictory.

Our proposal is for theory building to be placed at the centre of system development and for the theory to be the foundational artifact. Theories can then be used to unify different phases of system development including specification, design, implementation and maintenance. Each phase is characterized by properties of the theories, however, in principle, the same meta-technology can be used throughout:

Specification is characterized by multiple theories that capture different functional and non-functional properties of the required system.

Design is characterized by theories that describe the system in terms of an idealized engine. Specification is linked to design using theory mappings that embed the required properties into idealized executions.

Implementation is characterized by theories in terms of a technology platform and design theories are translated in much the same way as advocated by Model Driven Architecture approaches.

Maintenance is characterized by understanding a provided system in terms of implementation theories and then abstracting execution details through design and into specification-level theories.

The following diagram shows aspects of the process:

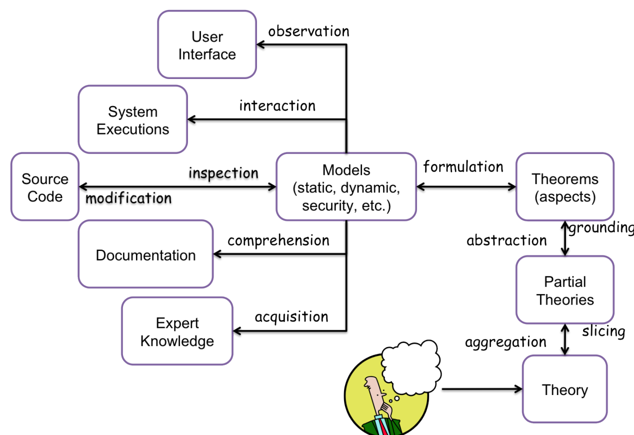

Figure 2: Theory Building Process

The diagram above shows how a developer interacts with the system development process based on theories and is not intended to prescribe any particular method. For example, starting with the conceptualization of a system, the developer constructs a theory in terms of its desirable properties. Slices of the theory give aspects of the system that can be made increasingly concrete to produce executable models of the system. MDA principles can be applied to the executable models in order to produce the source code and other artifacts. The developer above may be interpreted as a new project member whose task is to quickly become a productive contributor. The project member is introduced to various abstract theories that capture aspects of the system and which are linked to the executable code via mappings. The project member must understand the theory languages (far simpler than the implementation code) and can test their understanding by formulating new theorems, checking their validity and tracing their deduction through to code. Interpreting the diagram in reverse allows us to use theories to address the issue of program maintenance and re-engineering. In this case, an existing system and its associated artifacts are analyzed in order to produce partial models that can be formulated as theorems about aspects of the system's execution. These descriptions can be constructed using code inspection, existing reverse engineering tools, discussions with human experts, reading documentation, and most likely by a mixture of all of these. By abstracting the concrete theorems it is possible to produce partial theories that can be composed to produce one or more complete abstract descriptions. By maintaining the system in terms of theories, our claim is that the overhead of understanding and maintaining the system (the knowledge management problem) is reduced.

4 CONCLUSIONS

This paper has proposed that the current focus on software engineering from the perspectives of methods, lifecycles, and descriptions of artifacts has led to a number of problems notably the a large number of methods that have little or no research evidence that provides data on a method's efficacy; bloated lifecycle processes such as those exemplified by TOGAF; over-engineered artifact design arising from concepts and notations that have little semantic integrity such that the models produced are difficult to assess as to their “goodness”. Radically we have proposed that theory building and the core features of the notion of a theory may address these issues. We have not proposed how tools may be built , instead we suggest that

there is the potential for a new emerging agenda on theory building.

References

Fetzer, J. (1999). The role of models in computer science. *The Monist*, 82(1):20–36.

Floyd, C. (1984). Eine untersuchung von software-entwicklings-methoden. In Morgenbrod, H., Sammer, W., and Tagung, I., editors, *Programmierumgebugnen und Compiler*. Tuebner Verlag.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*, volume 206. Addison-wesley Reading, MA.

Johnson, P., Ekstedt, M., Silva, E., and Plazaola, L. (2004). Using enterprise architecture for cio decision-making: On the importance of theory. In *the Proceedings of the 2nd Annual Conference on Systems Engineering Research (CSER)*.

Lankhorst, M. M., H.A. Proper, H., and Jonkers, J. (2010). The Anatomy of the ArchiMate Language. *International Journal of Information System Modeling and Design*, 1(1).

Moody, D. (2005). Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. *Data & Knowledge Engineering*, 55(3):243–276.

Moor, J. (1978). Three myths of computer science. *British Journal for the Philosophy of Science*, 29(3):213–222.

Naur, P. (1985). Programming as theory building. *Microprocessing and Microprogramming*, 15(5):253 – 261.

Naur, P. (1992). *Computing: a human activity*. ACM New York, NY, USA.

Nonaka, I. and Takeuchi, H. (1995). The knowledge-creating company. *New York*, 1.

Ryle, G. (1949). The concept ofmind. *London, Hutchinson*.

Smith, B. (1991). *Limits of correctness in computers*. Academic Press Professional, Inc.

Spencer, J. et al. (2004). *TOGAF Enterprise Edition Version 8.1*.

Texas-Instruments (1988). *A Guide to Information Engineering Using the IEF: Computer-Aided Planning, Analysis, and Design*. Texas Instruments Inc.

Wisnosky, D. and Vogel, J. (2004). DoDAF Wizdom: A Practical Guide to Planning, Managing and Executing Projects to Build Enterprise Architectures Using the Department of Defense Architecture Framework (DoDAF).

Wyssusek, B. (2007). A philosophical re-appraisal of peter naur's notion of "programming as theory building". In *European Conference on Information Systems (ECIS)*.

Zachman, J. (1999). A framework for information systems architecture. *IBM systems journal*, 38(2/3):454–470.