Science of Computer Programming 101 (2015) 6–20

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Modules and transactions: Building blocks for a theory of software engineering[☆]

Cengiz Erbas[a,*], Bahar Celikkol Erbas[b]

[a] *ASELSAN Inc., Ankara, Turkey*
[b] *Department of Economics, TOBB University of Economics and Technology, Ankara, Turkey*

ARTICLE INFO

Article history:
Received 1 October 2013
Received in revised form 24 February 2014
Accepted 25 June 2014
Available online 24 November 2014

Keywords:
Transaction costs
Top–down governance
Bottom–up governance
Module decomposition
Module construction

ABSTRACT

This article leverages the findings of the transaction cost economics field, and proposes a simple theory and associated vocabulary to serve as a foundation for a unified theory of software engineering. The theory characterizes software engineering as a set of transactions organized under a spectrum of three governance structures (top–down, bottom–up and reuse), and explains the strengths and weaknesses of these governance structures in relation to asset specificity and uncertainty. It takes into account the recursive nature of the notions in software engineering, and applies uniformly to various contexts at different levels of granularity. It sheds light both on the technical and on the human aspects of software engineering through a unified explanatory framework, without requiring a need to assemble different approaches to address each. The theory not only explains some of the propositions given in the software engineering literature but also reveals the boundaries of their applicability.

1. Introduction

We have a chicken-and-egg problem in software engineering research. We have neither a common terminology nor a recognized theory for software engineering. There are numerous fragmented hypotheses regarding one observed phenomenon or other without having a shared explanatory framework [19]. A unified theory can be built only by means of a common terminology; however, a common terminology can emerge only in the presence of a theory [24]. The development of a theory and consensus building on terminology will therefore have to happen concurrently.

A unified theory should say something about the choice of "software development methods," however, this term by itself means different things to different individuals. There is no consensus, for example, on what "agile" constitutes. Numerous practices are mentioned in the literature for agile development methods and variants. The differences, nevertheless, are little understood and substantiated [18]. The problem is not something limited to the agile community, however. We can find references which assert that "modular programming" and "structured programming" refer to different methods, as well [47]. Other than serving as historical footnotes, it is difficult to judge the significance of such distinctions in practice in the absence of predictive and prescriptive support of a unified theory.

[☆] This is an extended version of the paper titled "On a Theory of Software Engineering: A Proposal Based on Transaction Cost Economics", which was published at the 2nd SEMAT Workshop on a General Theory of Software Engineering (GTSE 2013), San Francisco, California, May 18–26, 2013, pp. 15–18, IEEE Conference Publications, http://dx.doi.org/10.1109/GTSE.2013.6613864.

[*] Corresponding author.
E-mail address: cerbas@aselsan.com.tr (C. Erbas).

http://dx.doi.org/10.1016/j.scico.2014.11.006
0167-6423/© 2014 Elsevier B.V. All rights reserved.

A unified theory should not only be empirically testable but should provide good explanations for software engineering phenomena at multiple levels, including human aspects and technical artifacts [21]. It should go beyond displaying statistical relationships among bunch of variables, and should explain observations by linking with the existing scientific knowledge. Good explanations are often strikingly simple, elegant and hard to vary. They do not contain superfluous features or arbitrariness, and cannot be altered without changing their predictions [38].

The task of developing a unified theory could probably be easier, if we begin with the micro-foundations of software engineering. The success of a software development method for a given project is a function of many variables, a few of which are somewhat intricate. Rather than trying to explain the aggregate behavior of a set of complex variables, we can initially try to identify basic "units of analysis" and formalize their relationships to serve as a foundation for further theory development. If we can reach a consensus for a firm foundation, then it could be less complicated to build on.

It is also important to keep terminological consistency with other established scientific fields. Physicists and chemists, for example, use the same term "electron" to refer to the same phenomena. A chemist and a biologist can agree quite easily about the meaning of "protein." "Synapse" means the same thing to a biologist, a neuroscientist, and a physician. We should try to achieve similar consistency across relevant disciplines. Ensuring such consistency not only facilitates consensus building within the software engineering community but also enables us to leverage the existing findings of these other related disciplines in theory formation.

In this article, we make one such attempt. We demonstrate the relationship of software engineering with another established discipline, microeconomics, and propose a simple theory of *software decomposition*, *construction* and *reuse*. Relying on "modules" and "transactions" as the units of analysis, we leverage a few concepts, such as transaction costs, asset specificity and governance structures from microeconomics; introduce new ones, such as bottom up governance and top–down governance; and propose a theory which sheds light both on the technical and on the human aspects of software engineering through a unified explanatory framework. We consider the proposed theory, however, not as a unified theory of software engineering per se, but as a building block towards one. The theory places a central role to the way software development is governed (top–down or bottom–up) when seeking explanations to the observed phenomena in the field. The proposed theory is capable of not only explaining some of the propositions given in the software engineering literature, such as Parnas [35], Brooks [7], and Boehm and Turner [6], but also disclosing the limits of their applicability.

Seeking the foundation of software engineering in the economics discipline may appear counter intuitive. We argue the contrary. It is relevant not only because economical considerations, such as cost, value and productivity are important for software engineering, but more importantly, because the best software designs are quite frequently the most economical ones. It is not the complexity, but the simplicity of the design of UNIX operating system which makes it remarkable [45]. We argue that this relationship between "good" design and "economical" design is deeper than it may first seem. Through this relationship, we will show that economics may lay the common ground between the technical and the human aspects of software engineering.

The paper is organized as follows: Section 2 presents a review of the related work, specifically in relation to the applications of transaction cost economics in software engineering. Section 3 reviews the underlying assumptions of a spectrum of microeconomic theories, and argues that transaction-cost economics best represents the dynamics of software engineering. Section 4 introduces "modules" and "transactions" as the basic units of analysis for software engineering. Section 5 illustrates the concepts of asset specificity and uncertainty in software engineering through examples, and proposes three governance structures, namely reuse, top–down and bottom–up, as a foundation for a unified theory. Section 6 broadens the analysis from the level of modules to the level of module hierarchy, and analyzes the strength and the weaknesses of different development methods, from waterfall to agile, through this unified view. Section 7 demonstrates that proposed theory successfully explains many propositions cited in the literature. Section 8 concludes the paper.

2. Related work

Scientific disciplines arise on theoretical foundations; and software engineering is no exception. Even though the software engineering discipline has not yet succeeded to build one, the search for a unified theory was there, now and then, since the initial days of the discipline. We may recall the efforts of Halstead [14] as one early example to build a quantitative basis for a software science. The interest to build theoretical foundations to software engineering has been revitalized recently by the *Software Engineering Method and Theory* (SEMAT) initiative [20,17]. Among the papers presented in SEMAT workshops, two are particularly relevant from the perspective of this article, mainly because they make comparable predictions for certain phenomena associated with software development methods.

The first paper [39] proposed formulating a multi-level approach, and reviewed five theories that may explain software engineering phenomena at different levels, including individual (cognitive biases), team (transactive memory), artifact (boundary objects), process (sensemaking-coevolution-implementation) and project (complexity theory). The paper asserted that these theories provide consistent explanations and can be used in parallel to grasp the same software engineering phenomena. The second paper [42] framed development practices as organizational practices, and separated two conflicting approaches, namely, technical rationality, which views software development as a methodical and plan-centered process of building an artifact for a given set of requirements, and reflection-in-action, which views software development as an evolving and improvisational process of simultaneously understanding the problem and building the artifact. The paper ar-