The Journal of Systems and Software 000 (2016) 1–13

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Technical debt and system architecture: The impact of coupling on defect-related activity

Alan MacCormack*, Daniel J. Sturtevant

Harvard Business School, Soldiers Field, Boston, MA 02163, United States

ARTICLE INFO

Article history:
Received 31 May 2015
Revised 28 May 2016
Accepted 4 June 2016
Available online xxx

Keywords:
Technical debt
Software architecture
Software maintenance
Modularity
Complexity

ABSTRACT

Technical Debt is created when design decisions that are expedient in the short term increase the costs of maintaining and adapting this system in future. An important component of technical debt relates to decisions about system architecture. As systems grow and evolve, their architectures can degrade, increasing maintenance costs and reducing developer productivity. This raises the question if and when it might be appropriate to redesign ("refactor") a system, to reduce what has been called "architectural debt". Unfortunately, we lack robust data by which to evaluate the relationship between architectural design choices and system maintenance costs, and hence to predict the value that might be released through such refactoring efforts.

We address this gap by analyzing the relationship between system architecture and maintenance costs for two software systems of similar size, but with very different structures; one has a "Hierarchical" design, the other has a "Core-Periphery" design. We measure the level of system coupling for the 20,000+ components in each system, and use these measures to predict maintenance efforts, or "defect-related activity." We show that in both systems, the tightly-coupled Core or Central components cost significantly more to maintain then loosely-coupled Peripheral components. In essence, a small number of components generate a large proportion of system costs. However, we find major differences in the potential benefits available from refactoring these systems, related to their differing designs. Our results generate insight into how architectural debt can be assessed by understanding patterns of coupling among components in a system.

1. Introduction

How do system design decisions affect the long-term costs of system maintenance? A wealth of studies has examined the topic of system design, developing insights into how decisions should be made during the development of new technological systems (Banker et al., 1993; Banker and Slaughter, 2000). This work reveals the critical impact of architectural choices in creating a design that can meet requirements along multiple, sometimes competing, dimensions of performance (e.g., functionality, speed, ease of use, reliability, upgradeability etc.). Fewer studies however, have explored how system design decisions affect performance in the *mature* stage of a system's life, where maintenance and adaptation costs are relatively more important. Given prior work argues that these costs can represent up to 90% of the total expenditures over a system's lifetime, this represents a significant gap in our knowledge (Brooks, 1975).

This topic is especially relevant to the software industry, given the dynamics of how software is developed. In particular, software systems rarely die. Instead, each new version forms a platform upon which subsequent versions are built. With this approach, today's developers bear the consequences of all design decisions made in the past (MacCormack et al., 2007). However, the early designers of a system may have different objectives from those that follow, especially if the system is successful and long lasting (something that may be uncertain at the time of its birth). For example, if early designers favor approaches that are expedient in the short term (say, to speed up time to market), later designers will bear the consequences of those decisions. Furthermore, as the external context for a system evolves over time, even design decisions that were made correctly may become obsolete and require revisiting (Kruchten et al., 2012a).

These dynamics raise an interesting question, in that for many mature systems, significant potential value might be released through design changes to reduce a system's complexity while maintaining its functionality (known as "refactoring"). Unfortunately, decision makers have little empirical data by which to evaluate the value that might be generated by such efforts

* Corresponding author. Tel.: +1 617-495-6856.
E-mail addresses: amaccormack@hbs.edu (A. MacCormack), dsturtevant@hbs.edu (D.J. Sturtevant).

http://dx.doi.org/10.1016/j.jss.2016.06.007
0164-1212/

(MacCormack et al., 2006). While a software architect might intuitively recognize the potential benefits of architectural change, senior managers typically require a robust assessment of the financial consequences of change, before funding such efforts. This need, to link software design decisions with their financial consequences, has given rise to a new metaphor, Technical Debt. It captures the *extent to which design decisions that are expedient in the short-term can lead to increased system costs in future* (Brown et al., 2010; Kruchten et al., 2012a).

In this paper, we attempt to bridge the worlds of software architecture and finance. In particular, we evaluate the relationship between system design decisions and the costs of maintenance for two software systems that represent different design "Archetypes" – one possesses a Core-Periphery design, the other possesses a Hierarchical design. We characterize system design using a network analysis technique called Design Structure Matrices (DSMs) (Steward, 1981; Eppinger et al., 1994). Our analysis allows us to calculate the level of coupling for components in each system, and thereby to identify which are more central to the design, and which are peripheral. We then analyze the extent to which components with different levels of network coupling generate different maintenance costs (i.e., in terms of the activity required to fix defects) in these systems. Our results allow us to speculate on the potential value that could be released by a refactoring effort, and to assess whether this differs between different system types.

The paper proceeds as follows. In the next section, we review the prior literature on Technical Debt and system design, focusing on work that explores how measures of system design predict the costs of maintenance. We then describe our methods, which make use of Design Structure Matrices (DSMs) to understand system structure, and measure the level of coupling between components. Next, we introduce the context for our study and describe the two systems that we analyze. Finally, we report our empirical results and discuss their potential implications for both practitioners and academia.

2. Literature review

2.1. Technical debt in software systems

In a software system, design decisions that systematically favor short-term gains over long-term costs create what is called "technical debt" (Cunningham, 1992; McConnell, 2007). These debts arise from, among other things, poor design practices, inadequate testing procedures, missing documentation, or excessively interdependent architectures (Brown et al., 2010; Seaman and Guo, 2011; Kruchten et al., 2012a, 2012b; Li et al., 2015). The interest on these debts comes in the form of increased costs for maintenance and adaptation in future. For smaller software systems, these costs may not be significant, hence not worth addressing. But as a system grows and evolves, these costs can become substantial and an increasing burden on development teams (Eick et al., 1999). Evolutions in the external context may also render past design choices outdated, creating a "technological gap" between an existing design and current requirements (Kruchten et al., 2012a). Where such technical debts exist, opportunities to create value through redesign may exist, assuming the value released exceeds the cost of taking action (Sarker et al., 2009; Schmid, 2013).

Early work in the field of technical debt focused on describing the phenomenon, and developing typologies for the different types of debt that can affect a system (Guo and Seaman, 2011; Kruchten et al., 2012a; Tom et al., 2013). For example, Kruchten et al., (2012a) propose a technical debt "landscape," which divides software improvements from a given state along two dimensions: whether they are visible or invisible; and whether they focus on maintainability or evolvability. Early empirical studies emphasized understanding the debts associated with lower-level decisions surrounding code complexity, coding style, code "smells" and poor system documentation (Brown et al., 2010). Tools based upon these approaches focus on the degree to which a software system follows, or departs from, common coding "rules" (e.g., sonarcube.org). To make the concept of Technical Debt operational, significant efforts have been made to define metrics that capture both the totality of the debt in a system, as well as the drivers of different components of this debt (Seaman and Guo, 2011; Nughoro et al., 2011).

More recently, attention has been given to how higher-level design decisions associated with a system's architecture impact technical debt (Nord et al., 2012; Kazman et al., 2015; Xiao et al., 2016). As Kruchten et al., (2012a) argue, "*more often than not*, technical debt isn't related to code and its intrinsic qualities, but to structural or architectural choices…." Of particular interest to this study, several authors have developed metrics to capture structural properties of software systems, to be used to evaluate architectural debt (MacCormack et al., 2006; 2012; Nord et al., 2012; Kruchten et al., 2012b; Kazman et al., 2015). While the specifics of these metrics differ by study, they retain a common theme in that they focus on measuring the *coupling* in a system. This is achieved by examining direct and indirect dependencies between system components.

2.2. The design of complex technological systems

A large number of studies have contributed to our understanding of the design of complex systems (Holland, 1992; Kaufman, 1993; Rivkin, 2000; Rivkin and Siggelkow, 2007). Many of these studies are situated in the field of technology management, exploring factors that influence the design of physical or information-based products (Braha et al., 2006). Products are complex systems in that they comprise a large number of components with many interactions between them. The scheme by which a product's functions are allocated to components is called its "architecture" (Ulrich, 1995; Whitney et al., 2004). Understanding how architectures are chosen, how they perform and how they can be changed are critical topics in the study of complex systems.

Modularity is a concept that helps us to characterize architecture. It refers to the way that a product's design is decomposed into parts. While there are many definitions of modularity, authors tend to agree on the concepts that lie at its heart: The notion of interdependence within modules and independence between modules (Ulrich, 1995). The latter concept is referred to as "loose-coupling." Modular architectures are loosely-coupled in that changes made to one component have little impact on others. The costs and benefits of modularity have been discussed in a stream of research that has examined its impact on complexity (Simon, 1962), production (Ulrich, 1995), platform design (Sanderson and Uzumeri, 1995), process design (MacCormack et al., 2001) process improvement (Spear and Bowen, 1999) and industry structure (Baldwin and Clark, 2000).

Studies that seek to measure the modularity of technical systems typically focus on capturing the level of coupling that exists between different parts of a design. In this respect, the most prominent technique comes from the field of engineering, in the form of the Design Structure Matrix (DSM). A DSM highlights the inherent structure of a design by examining the dependencies that exist between its constituent elements in a square matrix (Steward, 1981; Eppinger et al., 1994). These elements can represent design tasks, design parameters or the components that comprise the system. DSMs have also been used to explore the degree of alignment between task dependencies and project team communications (Sosa et al., 2004). Recent work has extended this methodology to show how dependencies can be extracted automatically from software source code and used to understand system design

Please cite this article as: A. MacCormack, D.J. Sturtevant, Technical debt and system architecture: The impact of coupling on defect-related activity, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.007