2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion

Detecting and Quantifying Architectural Debt: Theory and Practice

Yuanfang Cai
Computer Science Department
Drexel University
Philadelphia, PA, USA
yfcai@csdrexel.edu

Rick Kazman
Shidler College of Business
University of Hawaii
Honolulu, HI, USA
kazman@hawaii.edu

I. Introduction

Software complexity accumulates slowly as part of normal development activities. Because it accumulates slowly, most developers are unaware that it is occurring: they are focused on their immediate task—such as adding a new feature or fixing a bug—and they typically try to accomplish this in the quickest way. However, these quick-and-dirty changes over time lead to problematic structures within the source code. The impact of such (typically incremental) design decisions on software maintenance and evolution have not been well studied or understood.

In our prior research we have shown that these flawed structures can propagate error-proneness and change-proneness among large numbers of files, and accumulate significant maintenance costs over time, similar to a debt that accumulates interest. We call this kind of technical debt *architectural debt* [7]. In this technical briefing we introduce the theory, practice and tool support for detecting and quantifying architectural debt.

We begin by introducing the theory behind our technologies and tool suite: the notion of the Design Rule Space (DRSpace) [6], an architectural model that we recently proposed. We have shown that a software system can and should be modeled as multiple, overlapping DRSpaces. Each DRSpace can capture one aspect of a complex software system, such as a feature, a pattern, or a crosscutting concern. Most importantly we have shown, over the more than 150 software systems that we have analyzed, that 50% to 95% of all the error-prone files are captured by just a few DRSpaces, typically fewer than 5. In addition, the files in these DRSpaces are frequently connected through flawed architectural relations. We call these DRSpaces *architectural roots* [6][3]. In this technical briefing, we will introduce these concepts and demonstrate how architectural roots can be automatically detected and visualized using the tools that we have developed. These tools integrate both structural and historical system information regarding the dependencies among files.

Second, we will introduce the definition and detection of architectural flaws—the flaws that lead to architectural debt—supported by our Titan tool chain [5] and our more recent tool, DV8. Based on our investigation of more than 150 open source and commercial software systems, we have shown that certain kinds of architectural flaws occur repetitively in virtually every software system. Based on this empirical foundation we have defined 6 types of recurring architectural flaws (what we call *hotspot patterns*), including *Unstable Interface, Modularity Violation*, *Unhealthy Inheritance*, *Cliques*, Crossings, and *Cross-package cycles*. These hotspot patterns have a significant and sustained impact on software quality and maintainability, and are, as we will demonstrate, the *root causes* of architectural debt. That is, you cannot get rid of (pay down) architecture debt without first removing these design flaws. We have shown, in our empirical studies, that these design flaws cause higher rates of bugs, higher rates of changes, and consume more and more project effort over time as the DRSpaces around them inevitably grown in size (number of files).

Even though we have proven that these architectural flaws are the locations of architectural debt, project managers still need to understand the costs and benefits of removing these debts before they can decide to invest in their removal, via refactoring. Project managers are, quite rightly, motivated to make decisions based on their return on investment. In most cases the "investment" needed to pay down architecture debt is refactoring. We will introduce our methodology of quantifying the key parameters of these architectural debts—such as the interest rate and accumulated penalty—so that the managers and architects can determine the return on their investment and hence make better informed decisions.

In summary, in this technical briefing, we will demonstrate how to use our tools to visualize various DRSpaces within a system, to detect architectural roots and hotspots, and to quantify architectural debts. We will also demonstrate how these techniques will provide insight into if, when, and how to refactor to remove such debt by refactoring. We have empirically verified our techniques in

978-1-5386-1589-8/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-C.2017.165

over 150 open source and industrial projects, written in Java, C++, C# and other major languages. And we have shown how our techniques help to identify flaws, identify debt, create refactoring plans, and build the business case to justify the refactoring.

II. IMPACT

The notion of architecture debt (its automated detection and the factors surrounding its removal) is relevant to software practitioners, researchers, and educators. For software practitioners, our extensive empirical studies have shown how the DRSpace approach is an effective way of locating the sources of bugginess in a project [1]. The methodologies and tools we have devised have already been adopted by several multinational companies. The analysis results have helped them identify major architectural risks, and the architectural debt quantification has convinced their top management to invest in refactoring.

For software researchers, even though it is well-accepted that finding and fixing bugs is one of the most time-consuming and costly activities in a software project's lifecycle, the majority of the work on, for example, bug detection and prediction, has been on how to find bugs at the source-code level. Our work, by contrast, identifies the root causes of bugs, which provides new ways to detect and remove them. For software educators, the concept of DRSpace can be used to teach how software architecture can be modeled and analyzed. In particular, we have shown that each design or architectural pattern can be modeled as a DRSpace [1][6], and our tool has been used to assess student submissions to check if their programs conform to a design, or if their application of design patterns is correct. We believe, therefore, that this topic is of broad interest and concern to the software engineering community.

This work moves us towards of our objective of explicitly linking software development practices and economic reasoning of architectural decisions. In doing so we are bridging the gap between software engineers and management, and enabling quantitative software analysis grounded on a sound theoretical and empirical basis.

REFERENCES

[1] Y. Cai, R. Kazman, C., Jaspan, J. Aldrich: Introducing tool-supported architecture review into software design education. CSEE&T 2013

[2] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka. A case study in locating the architectural roots of technical debt. In Proc. 37th International Conference on Software Engineering, May 2015.

[3] R. Mo, Y. Cai, R. Kazman, and L. Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In Proc. 12thWorking IEEE/IFIP International Conference on Software Architecture, May 2015.

[4] M. Ran, C. Yuanfang, K. Rick, X. Lu, and F. Qiong. Decoupling level: A new metric for architectural maintenance complexity. In Proc. 38thInternational Conference on Software Engineering, pages 499–10, 2016.

[5] L. Xiao, Y. Cai, and R. Kazman. Titan: A toolset that connects software architecture with quality analysis. In 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering, 2014.

[6] L. Xiao, Y. Cai, and R. Kazman. Design rule spaces: A new form of architecture insight. In Proc. 36rd International Conference on Software Engineering, 2014.

[7] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "dentifying and quantifying architectural debts," in *Proc. 38rd International Conference on Software Engineering*, 2016.

[8] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. Design rule hierarchies and parallelism in software development tasks. In Proc. 24th IEEE/ACM International Conference on Automated Software Engineering, pages 197–208, Nov. 2009.

[9] H. Cervantes, R. Kazman, Designing Software Architectures: A Practical Approach, Addison-Wesley, 2016.

[10] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, 3rd ed., Addison-Wesley, 2012 (2nd ed. 2003, 1st ed. 1998).