DV8: Automated Architecture Analysis Tool Suites

Yuanfang Cai
Department of Computer Science
Drexel University
Philadelphia, USA
yfcai@cs.drexel.edu

Rick Kazman
Dept. of Information Technology Management
University of Hawaii
Honolulu, USA
kazman@hawaii.edu

***Index Terms*—Software Architecture, Software Quality, Software Maintenance**

I. Purpose of the Tool

Although software measurement and source code analysis techniques have been researched for decades, making project decisions that have significant economic impact—especially decisions about technical debt and refactoring—is still a challenge for management and development teams. Development teams feel the increasing challenges of maintenance as the architecture degrades, and often have intuitions about where the problems are, but have difficulty pinpointing which files are problematic and why. It is still a challenge for the development teams to quantify their projects' maintenance problems—their debts—as a way of justifying the investment in refactoring.

Here we present our tool suite called *DV8*[1]. The objective of DV8 is to measure software modularity, detect architecture anti-patterns as technical debts, quantify the maintenance cost of each instance of an anti-pattern, and enable return on investment analyses of architectural debts. Different from other tools, DV8 integrates data from both source code and revision history. We now elaborate on each of DV8's capabilities.

A. Maintainability Measurement

The first function of DV8 is architecture maintainability assessment using a pair of architecture-level maintainability metrics: *decoupling level* (DL) [7] and *propagation cost* (PC) [4]. DL measures how well a software system is *decoupled* into small and independent modules that can be developed and maintained in parallel. PC measures how tightly the files in a software system are *coupled*, which indicates the probability that changes to one file propagate to other files. These metrics were developed independently by different research groups. Applying both to the same project helps us evaluate which metric is more effective and reliable, and if and how they can reveal different aspects of the same project.

B. Architecture Anti-pattern Detection

The second major function of DV8 is *architecture anti-pattern* detection. Mo et al. [6] formally defined a set of recurring architecture problems that incur high maintenance costs. Files involved in such anti-patterns suffer from one or more architecture design mistakes, and have a significant impact on the bug-proneness and change-proneness of a software system. To make the penalty incurred by these anti-patterns explicit, we quantify the number of bugs and changes, as well as the bug churn and change churn, for each instances of anti-patterns using project history data. The users can also visualize each instance as a *design structure matrix* (DSM) [1], [13]. The anti-patterns that can be detected by DV8 are as follows:

1) Unstable Interface. According to design rule theory [1] and well-known design principles [5], an influential interface with a large number of dependents should remain stable. In practice, if influential files have high change rates, then multiple files depending on them have to be changed as a consequence.

2) Modularity Violation Groups. According to design rule theory [1], truly independent modules should evolve independently. Our prior work [6] proposed the concept of *Implicit Cross-module Dependency*, which is a variant of *modularity violation* [12] to identify modules without structural relationships but which have changed together frequently, as recorded in revision history. In DV8, the user can detect *Modularity Violation Groups*, which contain a minimal number of files involved in a mutual modularity violation.

3) Unhealthy Inheritance Hierarchy. This anti-pattern was defined to detect violations of the Liskov Substitution principle [2], [5] or Dependency Inversion Principle [5], where a parent class depends on one or more of its subclasses, or where a client of an inheritance hierarchy depends on both the parent class and its children. These cases make it impossible for an inheritance hierarchy to enable polymorphism, and could propagate bugs and changes to files that depend on this inheritance hierarchy.

4) Crossing. In DV8, a Crossing is defined as: a file that has both high fan-in and high fan-out, and changes often together with both the files it depends on and its dependents. Such a file is often at the center of maintenance activities. Since such files form a cross shape in a Design Structure Matrix [1], hence it is called *Crossing*.

5) Clique. Cyclic dependencies are a well-know design problem. Instead of detecting just pair-wise cycles [6], DV8 detects Cliques—sets of files that form strongly connected graphs.

6) Package Cycle. Dependency cycles between packages should also be avoided because this violates the basic design

[1] https://www.archdia.net/products-and-services/

978-1-7281-3371-3/19/$31.00 ©2019 IEEE
DOI 10.1109/TechDebt.2019.00015

principle of forming a hierarchical structure [10]. Changes to a file in one package often cause unexpected changes to files in other packages due to the cycle of dependencies among them.

Given the history record of a project, DV8 can also quantify the *maintenance costs* incurred by each anti-pattern instance, in terms of the number of bugs, changes, bug churn and change churn incurred by the files involved in each anti-pattern instance.

C. Architecture Debt Analysis

DV8 also provides *architecture root* analysis, proposed by Xiao et al. [15], in which the authors proposed the notion of a *design rule space* (DRSpace)—a set of architecturally connected files that implement a pattern, a feature, or other important system concern. This paper also proposed the concept of *architecture roots*—DRSpaces that cluster together the most error-prone files in the system. As Xiao et al. [15] reported, five architecture roots in a project typically cover 50% to 90% of the most error-prone files within a project. Using DV8, the user can also estimate the return-on-investment of refactoring each root, which can be used as a basis to make refactoring decisions.

II. Validation Experiences with Practitioners

DV8 has been used by multiple companies and we have analyzed dozens of industrial projects. As we reported recently [8] with ABB[2], we integrated DV8 into their development process. In that paper, we reported our analysis of eight projects in the company. These projects were developed at multiple locations (India, USA, and Switzerland) and differ in their ages, domains, and sizes. Our study had the following steps: first, ABB's development teams granted us access to their code repository from which we collected file dependency information, history data, and work items (commits). Using this data as input, we ran DV8, which automatically generated metrics, and visualizable architecture anti-patterns and roots, along with supporting data. Finally, we combined the output from these tools into a report for each project, and presented these to the development teams. After we ensured that the development teams understood the reports, we conducted a phone or email interview with each team to collect their feedback and, most importantly, to see if these techniques helped them to determine if, when, and where to refactor.

Our experiences have shown that the two metrics—PC and DL—can faithfully reflect the extent to which a project is experiencing maintenance difficulty. The complementary nature of PC and DL can provide useful insights. The architecture anti-pattern detector can highlight which files are suffering from which design problems; this visualization and quantification has effectively bridged the gap between management and development teams. Except for the two smallest projects, containing just a few hundreds of files each (and the highest metrics scores), all other projects are now undergoing major refactorings to address the problems that DV8 detected.

[2]http://www.abb.com

III. Relevance to Technical Debt

The architecture problems identified by DV8 are one type of technical debt. DV8 provides two ways to quantify architecture debt. First, DV8 detects Architecture Roots [15], which capture how bug-prone files are structurally connected and clustered together, and how they evolve over time. Considering each root as a debt [3], DV8 can calculate the maintenance costs of files involved in each root and the benefits achievable through refactoring. Second, DV8 detects architecture anti-patterns [6], that is, recurring architecture problems among files with significant impacts on bug-proneness and change-proneness. DV8 not only identifies these anti-patterns, but also quantifies the severity of each instance. DV8 has been repeatedly validated in industrial settings and greatly valued by practitioners [3], [8], [9], [11], [14]

References

[1] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.

[2] L. Barbara. Keynote address - data abstraction and hierarchy. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, pages 17–34, 1987.

[3] 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.

[4] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, July 2006.

[5] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, 2003.

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

[7] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng. Decoupling level: A new metric for architectural maintenance complexity. In *Proc. 38th International Conference on Software Engineering*, pages 499–510, 2016.

[8] R. Mo, W. Snipes, Y. C. S. Ramaswamy, R. Kazman, and M. Naedele. Experiences applying automated architecture analysis tool suites. In *Proc. 33rd IEEE/ACM International Conference on Automated Software Engineering*, pages 779–789, 2018.

[9] M. Nayebi, Y. Cai, R. Kazman, G. Ruhe, Q. Feng, C. Carlson, and F. Chew. A longitudinal study of identifying and paying down architectural debt. In *Proc. 41st International Conference on Software Engineering*, May 2019.

[10] D. L. Parnas. Software fundamentals. chapter On a Buzzword: Hierarchical Structure, pages 161–170. 2001.

[11] R. Schwanke, L. Xiao, and Y. Cai. Measuring architecture quality by structure plus history analysis. In *Proc. 35rd International Conference on Software Engineering*, pages 891–900, May 2013.

[12] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *Proc. 33rd International Conference on Software Engineering*, pages 411–420, May 2011.

[13] 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.

[14] W. Wu, Y. Cai, R. Kazman, R. Mo, Z. Liu, R. Chen, Y. Ge, W. Liu, and J. Zhang. Software architecture measurement—experiences from a multinational company. In *Proc. 12th European Conference on Software Architecture*, Sept. 2018.

[15] 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.