Software Architecture Health Monitor

Yuanfang Cai
Drexel University
Philadelphia, PA, USA
yfcai@cs.drexel.edu

Rick Kazman
University of Hawaii & SEI/CMU
Honolulu, HI, USA
kazman@hawaii.edu

ABSTRACT

In this paper, we first discuss the concept of architecture debt and how this debt grows in virtually every software-intensive project. Next we propose a methodology for developers to monitor the health of a project's architecture through an analysis of the development artifacts that a project produces. Our objective is to monitor and manage architecture debt by leveraging an architecture model that we proposed recently, called the *Design Rule Space (DRSpace)*. We use DRSpaces to split a complex system into smaller subsystems based on features, patterns, refactoring targets, etc., so that we can monitor and analyze the evolution and interaction of each subsystem separately. We also employ a recently proposed architectural metric—Decoupling Level—to quantitatively monitor changes in a project's overall level of architecture maintainability. Using these tools, we describe our vision for a software architecture "health monitor", on analogy with a health monitor used in a hospital, to continuously monitor the health of the "patient" and alert staff to any potential danger signs.

1. INTRODUCTION

Software can be complex and this complexity is largely invisible, and intangible. Complexity accumulates slowly over time as part of normal development activities of bug fixing and adding features. Because it accumulates slowly, most developers are unaware that it is even occurring: they are focused on their specific task at hand, such as adding a new feature, and often accomplish their objective in the simplest, quickest way due to budget and schedule pressures. This accumulation of complexity, when it is unintended and when it contributes to making the system harder to maintain and evolve, as it often does, has been termed *technical debt* ([5], [3]). In this paper we are focused on a specific kind of technical debt—architecture debt [10]—and its cumulate effects on project health.

Architecture debt manifests in many ways. For example, it manifests itself as tangled, often cyclic, dependencies, or as modularity violations (where a set of files changes together frequently in the project's revision history, but those files have no structural dependency on each other) [16], or as unstable interfaces [13], where an important interface is buggy and changes frequently, and this affects the many files that use the interface, or as improper inheritance, where a parent class depends on its child class or where a client depends on both the parent and child class. Each of these is a violation of proper design principles and such violations (or *flaws*, as we call them) degrade the integrity of the architecture over time. While any individual flaw may not be burdensome for the project, the accumulation of such flaws gradually makes the architecture harder to maintain, harder to debug, and harder to evolve.

In this problem, we see an opportunity—a way to monitor such flaws, and the increasing amount of architecture debt in a project. In the remainder of this paper we will describe our approach to monitoring project architectural health, and our results to date.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
Bridge'16, May 17 2016, Austin, TX, USA
© 2016 ACM.
ISBN 978-1-4503-4153-0/16/05...$15.00
DOI:http://dx.doi.org/10.1145/2896935.2896940

2. ARCHITECTURE ANALYSIS

We have been analyzing software architectures for more than two decades, from the perspective of both design quality and costs/benefits (e.g. [9], [1], [3]). The techniques we had previously developed, such as the Architecture Tradeoff Analysis Method (ATAM), were manual and labor-intensive. Such techniques allowed us to broadly identify risks, but we wanted the ability to zoom in and try to find the specific architectural weaknesses or flaws in the code. In our more recent work we have introduced the concept of a Design Rule Space (DRSpace), based on Baldwin and Clark's *design rule theory* [2]. In a software system a design rule is an important abstraction, typically an interface or abstract class, that many other files depend upon.

We have shown how the DRSpace concept can be used to cluster and understand a software architecture's modular structure. And we have shown how an analysis of DRSpaces can provide insight into clusters of "bugginess" in a project [17]. But more than just identifying buggy areas of an architecture, the DRSpace approach has shown that it can help to determine the root causes for this bugginess. These root causes are architectural flaws (also called "hotspots"), that undermine the integrity of a design. These hotspots have been shown to be strongly correlated with high rates of bugs, changes, and churn [13] as well as high rates of security flaws [7].

3. BUILDING DRSPACES

We have developed the Titan tool chain to automate the collection of architectural information and the calculation of DRSpaces, as shown in Figure 1. On the left side of the tool chain are several project artifacts. In addition to source code, these artifacts include the revision control system (e.g., Subversion or Git) that manages commits and releases. Our approach relies on the fact that virtually every software development projects uses issue tracking tools, such as JIRA, to manage bugs and changes to the system. We realized that there is substantial information in these artifacts (issue tracking and revision control) that can provide valuable insights into the design of a system.

Using a commercial reverse engineering tool (we use Understand™, but in principle any reverse engineering tool could be used) we are able to extract the basic facts of a system. For

example, file A inherits from file B, or file A calls file B, or file A depends on file B also aggregates.

Using the information we obtained, we created a design structure matrix (DSM)—a square matrix where the rows and columns are the names of source files in the project, in the same order. Every cell in the matrix indicates a dependency between the file on the row and the file on the column (or is empty if no dependency exists). And the diagonal thus represents the self-dependency. This data structure allows us to view and manipulate a very compact representation of all the structural relationships that exist between the source files of a project.

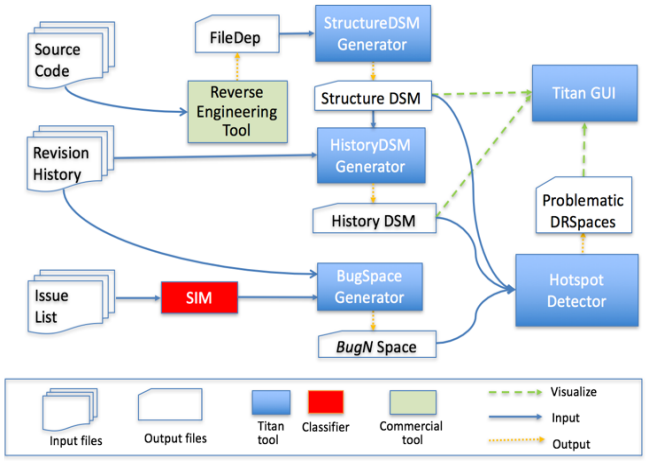

Figure 1: The Titan Tool Chain [17]

We next extract the revision history of the project. Every disciplined software project maintains its source code in a revision control system so that when changes are made to a file, an artifact is checked in for that change. Analyzing the revision history provides a historical record of how the project has evolved. Using that historical record, we build a separate "history DSM" that allows us to represent history dependencies between files. For example, if a developer makes a change to fix a bug or implement a feature, and this change involves modifications to both files A and B, the history DSM will indicate a history dependency in the cell at (row A, column B). Every time file A and file B are committed together, we annotate that in the history DSM, for all A and all B. Thus we have two sources of information about dependencies between a project's files: one representing structural dependencies, and the other representing historical dependencies.

Next, we clustered these DSMs using the design rule hierarchy (DRH) algorithm [6]. This algorithm rearranges the rows and columns of the DSM such that the lower layers depend on the higher layers and the modules in the same layers are independent of each other. The DRH algorithm reorders the rows and the columns so the most important files, the files that most others depend on, are at the top. Then, underneath those are files they depend on, and underneath those are the files they depend on, and so forth. For example, in Figure 2—a representation of the FileSystem class in Hadoop HDFS—you can see that the most important file in this small DRSpace is the parent class, org.apache.hadoop.fs.FileSystem. Beneath this parent are its 7 sub-classes, which form a coherent module. The cells in column 1 are annotated with "ih" to indicate that these classes inherent from the FileSystem class.

		1	2	3	4	5	6	7	8
1	org.apache.hadoop.fs.FileSystem	(1)							
2	org.apache.hadoop.fs.FilterFileSystem	ih	(2)						
3	org.apache.hadoop.fs.RawLocalFileSystem	ih		(3)					
4	org.apache.hadoop.fs.s3.S3FileSystem	ih			(4)				
5	org.apache.hadoop.fs.kfs.KosmosFileSystem	ih				(5)			
6	org.apache.hadoop.dfs.DistributedFileSystem	ih					(6)		
7	org.apache.hadoop.dfs.HftpFileSystem	ih						(7)	
8	org.apache.hadoop.fs.InMemoryFileSystem$RawInMemoryFileSystem	ih							(8)

Figure 2: A DRH-clustered DSM [17]

A DRSpace is thus formed by selecting a set of files, selecting their structural and evolutionary (history) dependency relations and then clustering them using the DRH algorithm.

According to design rule theory independent modules are decoupled from the rest of a system by design rules. These independent modules should only depend on design rules. As long as the design rules remain stable, a module can be improved, or even replaced, without affecting other parts of the system. But in reality this seldom occurs. In our cases studies of over 100 open source and industrial projects [11] we have seen that architectural complexity increases, and hence architecture debt increases, over time. Others have also noted this trend, calling it "software entropy" [8]. Unless energy is added to a closed system, entropy cannot be reduced. In a software project, the only way to reduce entropy and to pay down architecture debt, is via refactoring.

But refactoring is expensive, consumes project schedule, and produces no new features. The costs of refactoring are immediate and concrete, whereas the benefits are vague and long-term. So projects seldom invest in it. Our research aims to make architecture debt continuously and automatically visible, as a measure of project health, so that it can be monitored and, when it reaches critical levels, fixed.

4. MONITORING AN ARCHITECTURE VIA DRSPACES

By running the Titan tool chain on a regular basis, as part of the nightly build of a project, an architect or project manager can quickly assess the health of the software architecture. This health measurement can be applied to a single DRSpace or to the entire project. In this section we describe how we monitor the health of a single project's architecture. In the next section we describe how we can monitor the health of any DRSpace in the project.

We have three distinct ways to assess project health:

1. by calculating and tracking a project's "architecture roots" [17][16]. Architecture roots are DRSpaces that, collectively, cover the majority of a project's buggy files. We have shown that, for the vast majority of projects, at least 80% of the project's buggy files can be found in just 5 DRSpaces, these DRSpaces are typically led by one of the buggiest files in the project, and these DRSpaces grow over time, accumulating more files and more bugs [18]. Thus these clusters are "roots" of bugginess and should be tracked as they grow and evolve over time.
2. by calculating and tracking a project's architecture flaws. The flaws that we detect include: (a) unstable interface, (b) modularity violation, (c) improper inheritance, (d) cross-module cycle, (e) and cross-package cycle [13]. The first four hotspot patterns capture flawed relations among files while the last one captures cycles among packages. As mentioned above, the rates of these flaws have been shown to be strongly correlated with bugs, changes, and churn ([13], [7]). That is to say, the more flaws that any given file participates in, the greater the probability that the file will experience high rates of bugs, high rates of changes, and a large quantity of churn

(committed lines of code) to fix those bugs and to make those changes. Thus flaws also need to be monitored in a project, as they are signs of degrading patient health.

3. by calculating a project's *decoupling level* (DL) [12]. DL is a new metric for measuring the maintainability of a software architecture. It measures the degree to which an architecture is decoupled into (small, independent) modules that can be independently modified by developers. Higher values of DL are better, indicating a more modular architecture. By tracking DL, a project can not only determine whether this measure is going up or down, but can compare the project to industry averages as a way of assessing whether some mitigations (refactorings) need to be put in place to address growing architecture debt.

Consider, for example, Figure 3, which tracks the DL measures for 29 snapshots of a commercial project, taken over a 6-year period. Initially the DL starts out at approximately 0.45, which is well under the mean of 0.6 for the 129 projects that we studied [12]. However, this was a period during which the project was evolving from a prototype and, as the architecture matured and design rules were implemented, the DL reached a peak of nearly 0.8 before settling at a still-healthy level of nearly 0.7. By snapshots 17 and 18, however, the DL had dropped precipitously. The cause of this drop was a refactoring that was improperly implemented.

This refactoring introduced 5 major new interfaces—that is, design rules. 4 of the 5 DRSpaces led by these new design rules were successful (and limited in scope), but the fifth was not: It affected many other files in the project. By examining the DRSpace led by this new design rule, the reason for the difficulties became obvious: there were many secondary dependencies among the files that were intended to be decoupled by the new design rule. These unhealthy dependencies were eventually removed, resulting in the DL returning to a healthy state by snapshot 29.

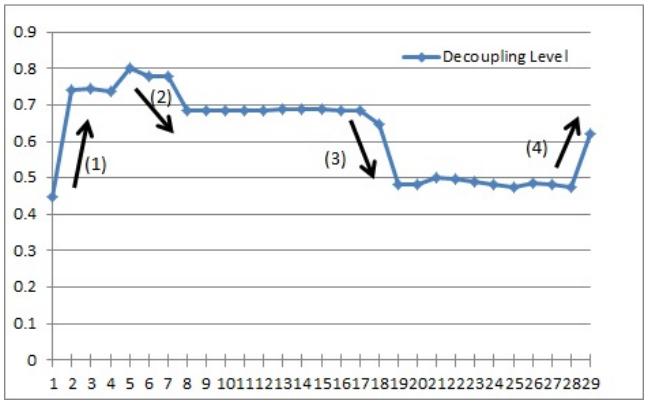

Figure 3: Tracking DL in a Commercial Project [12]

The data collected and analyzed by the Titan tool chain provides the necessary inputs to a rational, facts-based refactoring decision. Once a flawed DRSpace is identified, we can quickly assemble the data needed to make a business case for refactoring. We can compare bug, change, and churn rates in a flawed DRSpace to project averages. Using the difference between these rates, we can calculate an estimated benefit for the refactoring activity, in terms of bugs avoided or lines of code not written, per unit of time. And, given that the flaws in a DRSpace are explicitly identified by Titan, we can create—and create a cost estimate for—a refactoring plan based on removing those flaws. Given these estimated costs and benefits, we can finally calculate a return-on-investment for the refactoring strategy [10].

5. MEASURING ARCHITECTURE DEGRADATION

The techniques presented above can also be applied to any sub-part of the architecture. Consider, for example, the two DSMs in Figures 4 and 5, which represent the DRSpaces for an interpreter pattern, used in a commercial system. This DRSpace is led by a class named *Expression* (the file enclosed in the red circle in Figure 4). These two DSMs come from successive versions of the project. In the initial version, in Figure 4, the interpreter pattern was poorly implemented. As shown in this DSM, the leading file, *Expression,* depends on 12 other classes, while 41 other files depend on it. This DSM has a Propagation Cost (PC; an architecture-level measure of coupling [11]) of 50%, and a DL of 46%, which is well below industry averages.

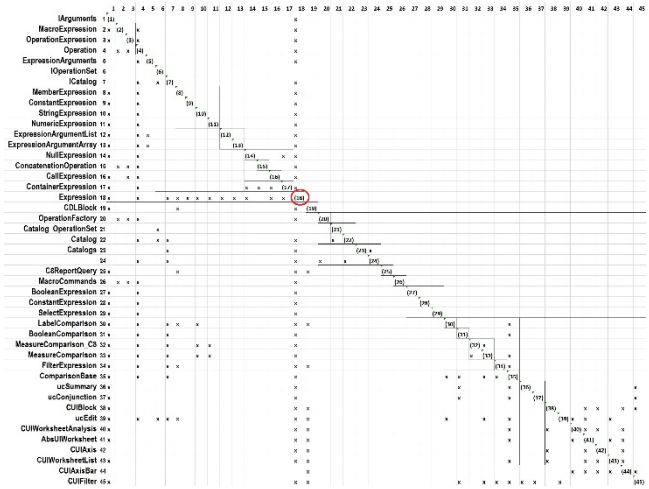

Figure 4: An Incorrect Interpreter Pattern DRSpace

In the next version of the system, the team made an attempt to refactor the interpreter pattern by injecting a visitor pattern to better decouple the algorithm from the data structure on which it operates. At the same time, the team needed to add new features to the system, to parse new expressions.

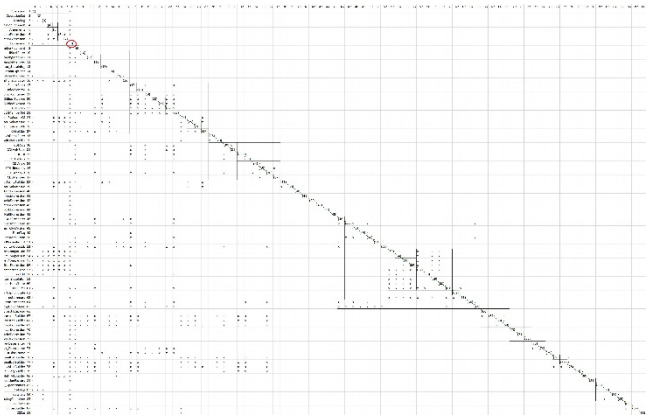

Figure 5: The Refactored Version of the Interpreter Pattern DRSpace

To check the effects of the refactoring, we extracted the same DRSpace, led by *Expression,* from the next version of the project, as shown in Figure 5.

Even though the DSM is too large to be presented clearly here, we can easily observe that the *Expression* class—the key design rule of this system—now depends on only 6 other classes in the DRSpace. The PC of this DRSpace has decreased to 21%, while the DL of the DRSpace has increased to 61%. These two measures indicate that the refactoring was successful and that the project has restored health to this portion of the architecture.

6. CONCLUSIONS AND FUTURE DIRECTIONS

In our earlier work we argued that the software industry needs to adopt ideas and principles from "Manufacturing Execution Systems" into software development practices ([14], [15]). In this work we claimed that an MES-style approach, with integrated, continuously-collected data about process activities can aid in planning and estimation accuracy, process improvement, and product quality.

The DRSpace approach provides one piece of this vision—a means by which a project manager or architect can continuously monitor product quality, comparing it to project, organization, or industry norms. And, using the continuously collected information, a manager can make reasoned, economics-based decisions on if and when to refactor. This allows the project to manage and control a form of technical debt: architectural debt.

We liken this process to that of a health monitor, that continuously measures a patient's vital signs, providing timely and critical feedback to health-care professionals. Such feedback is essential to determine whether to stage an intervention and to assess the success of any such intervention.

Interventions, in the form of refactorings, are the principle means by which an architecture's health can be improved. A DRSpace-based analysis not only can aid in determining whether such a refactoring is required, but can aid in assessing the return-on-investment of such an intervention. In this way project stakeholders can make such decisions in confidence, based on hard empirical data.

In this paper we reviewed the three main ways, based on our Titan tool chain that we monitor and manage architecture debt: by tracking the architecture roots, by tracking architecture flaws, and by tracking DL. These techniques have now been applied on over 150 projects, including 30 large-scale industrial projects, and have provided valuable insight to project stakeholders.

7. ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Science Foundation under grants CCF-1065189, CCF-1514315 and CCF-1514561.

8. REFERENCES

[1] J. Asundi, R. Kazman, M. Klein, "Using Economic Considerations to Choose Amongst Architecture Design Alternatives", CMU/SEI-2001-TR-035, Software Engineering Institute, Carnegie Mellon University, 2001.

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

[3] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, N. Zazworka, "Managing Technical Debt in Software-Reliant Systems", *FSE/SDP Workshop on the Future of Software Engineering Research at ACM SIGSOFT FSE-18*, 2010.

[4] P. Clements, R. Kazman, M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison- Wesley, 2001.

[5] W. Cunningham, "The WyCash Portfolio Management System", *Proceedings of OOPSLA '92*, 1992.

[6] Y. Cai, H. Wang, S. Wong, L. Wang, "Leveraging design rules to improve software architecture recovery", *Proceedings of the 9th international ACM SIGSOFT conference on Quality of Software Architectures*, 2013

[7] Q. Feng, R. Kazman, Y. Cai, R. Mo, L. Xiao, "An Architecture-centric Approach to Security Analysis", *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture*, 2016.

[8] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison–Wesley, 1992.

[9] R. Kazman, G. Abowd, L. Bass, M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architectures," *Proceedings of the 16th International Conference on Software Engineering*, 1994.

[10] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, A. Shapochka, "A Case Study in Locating the Architectural Roots of Technical Debt", *Proceedings of the 37th International Conference on Software Engineering,* 2015.

[11] A. MacCormack, J. Rusnak, C. 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.

[12] R. Mo, Y. Cai, R. Kazman, L. Xiao, Q. Feng, "Decoupling Level: A New Metric for Architectural Maintenance Complexity", *Proceedings of the 38th International Conference on Software Engineering,* 2016

[13] R. Mo, Y. Cai, R. Kazman, L. Xiao, "Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells", *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture*, 2015.

[14] M. Naedele, H-M Chen, R. Kazman, Y. Cai, L. Xiao, C. Silva, "Manufacturing Execution Systems: A Vision for Managing Software Development", *Journal of Systems and Software*, March 2015, 101, 59-68.

[15] M. Naedele, R. Kazman. Y. Cai, "Making the Case for a "Manufacturing Execution System" for Software Development", *Communications of the ACM*, December 2014, 57:12, 33-36.

[16] S. Wong, Y.Cai, M. Kim, M. Dalton. "Detecting software modularity violations", *Proceedings of the 33th International Conference on Software Engineering,* 2011.

[17] L. Xiao, Y. Cai, R. Kazman, "Design Rule Spaces: A New Form of Architecture Insight", *Proceedings of the 36th International Conference on Software Engineering,* 2014.

[18] L. Xiao, Y. Cai, R. Kazman, R. Mo, Q. Feng, "Identifying and Quantifying Architectural Debts", *Proceedings of the 38th International Conference on Software Engineering,* 2016.