2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

A Case Study in Locating the Architectural Roots of Technical Debt

Rick Kazman*, Yuanfang Cai‡, Ran Mo‡, Qiong Feng‡, Lu Xiao‡,
Serge Haziyev†, Volodymyr Fedak†, Andriy Shapochka†
*SEU/CMU & U. of Hawaii, Honolulu, HI, USA. Email: kazman@hawaii.edu
†SoftServe Inc., Lviv, Ukraine. Email: {shaziyev,vfedak,ashopoch}@softserveinc.com
‡Drexel University, Philadelphia, PA, USA. Email: {yc349,rm859,lx52}@drexel.edu

***Abstract*—Our recent research has shown that, in large-scale software systems, defective files seldom exist alone. They are usually architecturally connected, and their architectural structures exhibit significant design flaws which propagate bugginess among files. We call these flawed structures the *architecture roots*, a type of technical debt that incurs high maintenance penalties. Removing the architecture roots of bugginess requires refactoring, but the benefits of refactoring have historically been difficult for architects to quantify or justify. In this paper, we present a case study of identifying and quantifying such architecture debts in a large-scale industrial software project. Our approach is to model and analyze software architecture as a set of design rule spaces (DRSpaces). Using data extracted from the project's development artifacts, we were able to identify the files implicated in architecture flaws and suggest refactorings based on removing these flaws. Then we built economic models of the before and (predicted) after states, which gave the organization confidence that doing the refactorings made business sense, in terms of a handsome return on investment.**

I. Introduction

Despite the many advances in architecture design and analysis over the past two decades, it still remains largely an art, based on experience and intuition. This is highly problematic for the state of the practice. In particular, it is problematic for practicing architects who need to justify their decisions—particularly those affecting cost, schedule, and quality—to managers who often lack the deep technical skills to properly evaluate those decisions. But project managers do understand cost and schedule, and they are motivated to maintain high quality. So it is in the architect's best interests to translate their technical concerns into economic concerns, so that they can properly justify those decisions.

In this paper we present a case study of a software development organization–SoftServe Inc.—that did just that: facing high and mounting problems with technical debt in a project, they were able to analyze their software architecture, pinpoint the *hotspots* within that architecture that were the principle causes of technical debt, propose refactoring solutions to fix the hotspots, and (perhaps most important) make a business case for the refactoring. In this paper, we will describe the architectural analysis that we did for one of the projects, and how we helped them build their business case.

The state of the practice today in technical debt identification is largely informal, experience-based, and intuition-based analysis. Our recent research has shown that, in large-scale software systems, defective files seldom exist alone. They are usually architecturally connected, and their architectural structures exhibit significant design flaws which propagate bugginess among numerous files. The popular but informal notions of "code smells" or "technical debts" are not sufficient to precisely locate the architecture problems that propagate errors among multiple files, nor to quantify their impact.

The consequence of this informality is that it is universally difficult for architects to convince project managers to allow them to refactor: the costs of refactoring are concrete and immediate whereas the benefits of refactoring are vague and long-term. Given this situation, it is no wonder that managers seldom give the green light to refactoring: it takes away resources from implementing features and fixing bugs and these are the activities that customers see and pay for.

To remedy this situation we have applied following strategy to identify and quantify architecture debts to a system that SoftServe was maintaining, and justified the refactoring of architecture problems with an economic analysis. We first used the Design Rule Space (DRSpace) analysis approach [30] to precisely locate architecture debts in a few clusters of files. After that, we visualized the architecture flaws among these files, pointing out to the architects how these flaws propagate errors. After these flaws, (architecture hotspots), were confirmed by the architects, we extracted data from the development process to quantify the penalty these debts were incurring, estimated the potential benefits of refactoring, and made a business case to justify refactoring.

When we started working together SoftServe had already been maintaining their system, which they inherited from another company, for almost two years. They were actively trying to improve the maintainability of the code base, remove dead and cloned code, and rationalize its architecture, and they had already made some progress in this direction. They had been working with commercial tools, such as SonarQube[1], Understand[2] and Structure 101[3], to help identify problematic areas in the system. What the DRSpace process offered them was, however, quite different than those commercial tools: we offered them explicit (and automated) identification of

[1]http://www.sonarqube.org
[2]https://scitools.com
[3]http://structure101.com

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.146

ICSE 2015, Florence, Italy
Software Engineering in Practice

problem areas in the architecture, along with explanations for *why* these areas were problematic. Unlike other tools that report a list of individual problematic files, We reported these architecture debts in the form of 3 to 6 groups of architecturally related files, and the architecture flaws among them can be visualized. This analysis revealed significant architecture issues not detectable by other tools, and allowed them to plan refactoring strategies to address these problems.

In the end, we convinced SoftServe to refactor and this was not a difficult argument. SoftServe was, in fact, happy to do the refactoring because: 1) they had specific advice about what to refactor, how, and why; 2) they had a framework for making economics-based decisions about refactoring that showed a clear and substantial predicted return on investment for this activity (nearly 300% ROI in the first year alone); and 3) they had more confidence in the results of the DRSpace analysis than in the outputs of the tools that they had been using because the visualization and quantification of architecture debts were intuitive and sensible to both architects and management. Furthermore, the proposed refactoring strategy was backed up by empirical evidence based on sound software engineering principles.

II. RESEARCH QUESTIONS

We conducted a case study as a means of achieving two objectives. First, we wanted to understand if our architecture hotspot analysis process could identify problems—architecture debts—that industrial practitioners consider to be real, significant, and worth fixing. Second, we wanted to understand if it is possible quantify these architecture debts, based on readily available project data, to help these practitioners make rational refactoring decisions.

Towards these objectives, we examined the following research questions.

RQ1: According to opinions of SoftServe's architects, are the set of architectural issues that we reported truly problematic issues—that is, architecture debts?

RQ2: How do the results returned by the Titan tool chain differ from the files reported as sources of technical debt by other tools SoftServe is using, such as SonarQube?

RQ3: Is it possible to quantify the return on investment of removing architecture debts? In other words, is it possible to determine the penalty incurred by the debts and the expected benefits if the debts are removed, and compare this with the costs of refactoring?

III. CASE STUDY PROCEDURE

We were fortunate to work with SoftServe, a leading software outsourcing company with more than 3,500 employees, distributed over 200 active projects, with locations in 8 countries. SoftServe has always prided itself on being a disciplined software engineering organization, having reached a CMMI level 3 and adopting many best practices in architecture, testing, agile development, and project management. Each project at SoftServe is managed using a suite of version control and issue tracking tools.

Moreover, SoftServe has made a significant, long-standing commitment towards maintaining software quality by both investing in ongoing education and by employing many commercial tools to identify technical debt, including Understand, Structure101, and SonarCube. Prior to our case study with the subject project—a web portal system which we will refer to as SS1 in this paper—SoftServe architects compiled a list of technical debts in SS1. These technical debts were of multiple types, and were detected by various tools and methods, such as multiple code violations detected by Sonar, numbers of Todo and FIXME tags reported by Eclipse, lack of reusability detected by code reusability scenarios, etc. We were interested to understand if the architecture debt areas we identified overlapped with the ones identified by the tools that SoftServe employed.

The most recent version of the project that we analyzed contains 797 source files. The revision history that we studied covers from July 2012 to May 2014. The project was maintained by 6 full-time developers, but with sporadic contributions from several dozen more developers. Over this nearly two-year time period, there were 3262 commits as recorded by their version control system. There were 2756 issues recorded in their JIRA issue tracking system. Of these issues, 1079 of them were bug issues, and 1677 were about epics, improvements, stories, technical tasks, etc. Given the choice of SS1 as our subject, our case study prosecuted the following steps:

First, we collected various data sets from the project, as shown in Figure 1. We processed the following inputs from SoftServe's project:

- A set of dependencies between all of the project's source files, output by Understand
- The project's revision history, from its Git repository
- The project's issue history, from its JIRA repository

Second, we used our tool, Titan [30], to calculate architecture hotspots within the source code of SS1, and to summarize all the architecture issues within these hotspots into a few high-priority areas of architectural technical debt.

Third, we output these architecture debt areas, represented as *Design Structure Matrices (DSM)*. We exported these as Excel spreadsheets and shared them with the project architects. We asked them a series of questions aimed at answering the first research questions proposed in the previous section. Our purpose was to understand if the problems that we identified were real, significant, and worth treating, and if we could identify significant problems that were not detectable by other tools they are using.

Finally, to quantify the architecture debts verified by the architects, we requested additional project data: about the lines of code committed to address issues, and estimates of the effort required to refactor the architecture to address the architecture technical debts that we identified. Using this information we were able to form a business case to help the architects decide if it was worthwhile to refactor, as we will discuss in section VI.

Fourth, to answer the second research question, we compared the sets of files identified by our tool chain as architec-

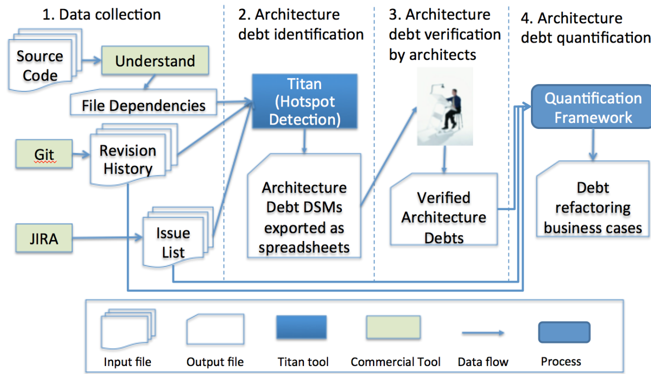

Fig. 1. Architecture Debt Identification, Verification, and Quantification Process

ture hotspots with the files reported by Sonar as containing "technical debt" to assess the degree to which our results differed from this *de facto* industrial standard tool. We also compared the debts we identified with the debt list that the architects had already compiled, to assess what our tool chain could and could not detect.

IV. Architecture Debt Identification

In this section, we describe how we identified architecture debts from dependency information output by Understand, as well as the project's revision history and issue tracking systems. We start by introducing the background concepts needed in this procedure.

A. Background

In our recent work, we proposed the concept of *Design Rule Space* (DRSpace) [30], [32]. Instead of viewing the modular structure of software architecture just as files and their relations, we consider software as a set of overlapping design spaces, each of them having its own modular structure formed by a suite of *design rules* and *independent modules* [1]. Reflected in source code, design rules are usually key interfaces or abstract classes that decouple other files into independent modules. Intuitively, if multiple design patterns are applied in a system, then each pattern has its own design space that overlaps with others. Since most patterns feature one, or a few, key interfaces as design rules, the design space of each pattern forms a DRSpace.

In general, a DRSpace can be seen as a selected set of files and a selected set of relations, such as inheritance, aggregation, or dependency. These files are clustered into a special form called a *design rule hierarchy* (DRH) [3], [4], [29] which identifies design rules and independent modules. The first layer of a DRH contains the files that have significant influence on the rest of system, but are not influenced by other files in lower layers. These files are usually important base classes, key interfaces, etc., and we call them the *leading files*.

We model a DRSpace using a *Design Structure Matrix (DSM)*, a square matrix with rows and columns labeled with the same set of files in the same order. A marked cell in row x, column y, c:(rx,cy) means that file x is related to file y, either through some kind of structural relation, or through evolutionary coupling (i.e., they have changed together, as recorded in the project's commit logs). The cells along the diagonal means self-dependency. A DSM, clustered as a DRH, can be viewed and manipulated using our tool Titan [31].

The DSM in Figure 2 presents a DRSpace generated from our case study with fake file names. This DRSpace is led by path1.Bean.java, and is clustered into 4 layers: l1: (rc1), l2: (rc2-rc19), l3: (rc20-rc25), l4: (rc26-rc27). As an example, the cell in row 5, column 1, cell(r5,c1) contains: "Create,10". It means that path1.ThirdFruit.java creates an instance of path1.Bean.java, and these two files changed together 10 times in the project's revision history. Consider another example, the cell in row 15, column 7. Cell(r15, c7) only contains ",16", which means that the file on row 15, path5.TenthFruit.java, and the file on column 7, path5.FifthFruit.java, do not have any structural dependencies, but they changed together 16 times as recorded in the revision history.

Based these concepts, we first identified the DRSpaces that capture the most error-prone and change-prone files. We call these DRSpaces architecture *roots*. From these root DRSpace, we further diagnosed architecture issues and extracted architecture debts. We now elaborate these steps in the next two subsections.

B. Architecture Root Calculation

We first identify the DRSpaces that cover the most error-prone and change-prone files, following the procedure as described in [30] and [21]. The rationale is that, if most error-prone files are architecturally connected, as we have observed from numerous open source projects, then Titan should identify just a few DRSpaces that contain most of these error or change prone files. We call these DRSpaces the *roots* of error and/or change proneness. The fewer the number of roots, the more closely these high-maintenance files are architecturally connected.

We consider that the set of files that change most frequently or have the most bug fixes as *change spaces* and *bug spaces*. For example, we consider all the files that were changed 10 times or more as a *Change10* space, and all the files that have more than 2 bug fixes as a *Bug2* space. In this case study, we calculated the root DRSpaces that cover the Change10 (63 files) and Bug2 (55 files) spaces. The data for root spaces that cover Change10 and Bug2 are reported in Table I and Table II. The file names in the first column of the table are the leading file of the DRSpace.

Taking the DRSpace led by Pear.java in Table I as an example, we can see that this DRSpace has 139 files (DRSsize). Although it contains about 17% of all the files in the project, it has 36 (Bug2Files) of all the 55 files with more than 2 big fixes, about 65% (Bug2%) of the Bug2 space. The column "Dist size" shows the cumulative number of distinct files. For example, Pear.java, Apple.java, and Strawberry.java together contain 306 distinct files, covering 80% of Bug2 space.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	path1.Bean_java	(1)	,	,10	,	,10	,	,	,	,	,	,	,	,	,	,8	,8	,	,	,	,16	,	,	,	,8	,	,	,
2	path2.Pear_java	,	(2)	,	,	,	,	,	,	,	,	,	,	,	,	,8	,	,	,	,	,	,	,	,	,	,	,	,
3	path3.FirstFruit_java	,10	,	(3)	,	,	,	,8	,	,	,	,	,8	,8	,	,10	,10	,8	,	,8	,8	,	,	,	,	,	,	,
4	path4.SecondFruit_java	Create,	,	,	(4)	,10	,8	,	,	,	,	,	,	,	,	,8	,8	,	,	,	,10	,	,	,	,	,	,	,
5	path4.ThirdFruit_java	Create,10	,	,	,10	(5)	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,14	,	,	,	,	,	,	,
6	path4.FourthFruit_java	Create,	,	,	,8	,	(6)	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,
7	path5.FifthFruit_java	,	,	,8	,	,	,	(7)	,	,8	,	,	,12	,16	,14	,16	,10	,10	,8	,10	,	,	,	,	,	,	,	,
8	path5.Pear_java	,	,	,	,	,	,	,	(8)	,	,	,	,	,	,	,	,10	,	,	,	,	,	,	,	,	,	,	,
9	path5.FirstJuice_java	,	,	,	,	,	,	,8	,	(9)	,8	,	,10	,10	,	,12	,8	,10	,12	,	,	,	,	,	,	,	,	,
10	path5.SecondJuice_java	,	,	,	,	,	,	,	,	,8	(10)	,	,	,	,	,10	,8	,	,	,	,	,	,	,	,	,	,	,
11	path5.SixthFruit_java	,	,	,	,	,	,	,	,	,	,	(11)	,	,	,	,10	,	,	,8	,	,	,	,	,	,	,	,	,
12	path5.SeventhFruit_java	,	,	,8	,	,	,	,12	,	,10	,	,	(12)	,14	,10	,12	,10	,10	,10	,8	,	,	,	,	,	,	,	,
13	path5.EighthFruit_java	,	,	,8	,	,	,	,16	,	,10	,	,	,14	(13)	,14	,16	,10	,10	,10	,10	,	,	,	,	,	,	,	,
14	path5.NinethFruit_java	,	,	,	,	,	,	,14	,	,	,	,	,10	,14	(14)	,14	,10	,8	,	,10	,	,	,	,	,	,	,	,
15	path5.TenthFruit_java	,8	,8	,10	,8	,	,	,16	,	,12	,10	,10	,12	,16	,14	(15)	,14	,14	,10	,14	,10	,	,	,	,	,	,	,
16	path5.EleventhFruit_java	,8	,	,10	,8	,	,	,10	,10	,8	,8	,	,10	,10	,10	,14	(16)	,12	,	,8	,10	,	,	,	,	,8	,	,
17	path5.TwelvethFruit_java	,	,	,8	,	,	,	,10	,	,10	,	,	,10	,10	,8	,14	,12	(17)	,	,	,	,	,	,	,	,	,	,
18	path5.ThirteenthFruit_java	,	,	,	,	,	,	,8	,	,12	,	Use,8	,10	,10	,	,10	,	,	(18)	,	,	,	,	,	,	,	,	,
19	path5.FourteenthFruit_java	,	,	,8	,	,	,	,10	,	,	,	,	,8	,10	,10	,14	,8	,	,	(19)	,	,	,	,	,	,	,	,
20	path6.FirstFood_java	,16	,	,8	,10	,14	,	,	,	,	,	,	,	,	,	,10	,10	,	,	,	(20)	,	,8	,10	,8	,8	,	,
21	path7.SecondFood_java	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	Use,	(21)	,	,14	,	,	,	,
22	path8.ThirdFood_java	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	Use,8	,	(22)	,	,	,	,	,
23	path7.Apple_java	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	Use,10	,14	,	(23)	,	,	,	,
24	path9.FourthFood_java	Create,8	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	Use,8	,	,	,	(24)	,10	,	,
25	path9.FifthFood_java	Create,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,8	,	,	,	Use,8	,	,	,	,10	(25)	,	,
26	path10.FirstFig_java	Cast,Use,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	(26)	,8
27	path2.SecondFig_java	Cast,Use,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,	,8	(27)

Fig. 2. DRSpace clustered into DRH Structure with only structure relations

TABLE I
THE MOST ERROR-PRONE DRSPACES

Leading Class	DRSsize	Dist size	Cover	Size%	Bug2Files	Bug2%
Pear.java	139	139	65%	17%	36	65%
Apple.java	158	277	76%	20%	12	22%
Strawberry.java	33	306	80%	4%	3	5%
Grape.java	5	311	84%	1%	2	4%
Blackberry.java	13	315	87%	2%	9	16%
Peach.java	36	351	89%	5%	1	2%

From Table II, we can see that Pear.java and Apple.java together cover 79% of the most frequently changed files, i.e., the Change10 space. 41 out of 63 most change-prone files can be found in the DRSpace led by Pear.java alone!

TABLE II
THE MOST CHANGE-PRONE DRSPACES

Leading Class	DRSsize	Dist Size	Cover	Size%	Chg10Files	Chg10%
Pear.java	139	139	65%	17%	41	65%
Apple.java	158	277	79%	20%	17	27%
Berry.java	58	305	86%	7%	16	25%
Whitepeach.java	60	319	87%	8%	20	32%
Greengrape.java	1	320	89%	0%	1	2%
Redgrape.java	2	322	90%	0%	1	2%
Yellowpeach.java	62	328	92%	8%	25	40%

These tables show that only 6 root spaces are needed to cover 89% of the Bug2 space, and 7 root spaces can cover 92% of the Change10 space, meaning that both error-prone and change-prone files are highly architecturally connected. In particular, we observe that the first two DRSpaces are the same for both Bug2 and Change10, and cover up to 76% and 79% of the most error and change prone files respectively.

C. Architecture Issues and Debt Extraction

There are many overlaps in the root spaces. This is because a file may participate in many relationships with other files. And there are many architecture issues within these root spaces. To return meaningful results to the SoftServe architects we identified instances of following architecture issues within these DRSpaces:

- *Unstable Interface* - A leading file with large number of dependents but changes frequently with many of them.
- *Implicit Cross-module Dependency* - Files belong to different independent modules in the DRH clustering, but are changed together frequently. This phenomenon is also called *modularity violations* [28].
- *Unhealthy Inheritance Hierarchy* - A parent class depends on one of its subclasses, or a client of the inheritance hierarchy depends on both the parent class and all the subclasses.

We identified file groups with these issues using the definitions and techniques defined in [21]. We did not report the most commonly found architecture problems, such cyclical dependencies among files or packages because those issues can be easily detected by the commercial tools SoftServe is already using, such as Sonar or Structure101.

We distinguish these issues because they already suggest corresponding refactoring strategies. For example, a DSM with an Unhealthy Inheritance Hierarchy instance only contains the files involved in the hierarchy, so their relations can be easily seen. If the problem is that a parent class depends on one of its subclasses, then it is obvious that this illegal dependency should be removed. A modularity violation instance suggests the need to discover the hidden relations among these files, while an unstable interface points to a better designed base class or interface.

Since each root DRSpace may contain multiple instances of architecture issues, and some files may be involved in multiple issues in multiple root spaces, we extracted 6 groups of files

(95 distinct files in total) with the least overlaps and the most prominent type of architecture issues, and returned these to SoftServe as possible *architecture debts*. The 6 file groups contained 3 instances of improper inheritance with 6, 3, and 7 files respectively. The small sizes of these issue instances indicate that inheritance is not a major architecture problem.

The other 3 instances included one modularity violation group with 27 files, and two instances of Unstable Interface issues, involving 26 and 52 files respectively. The DSM of the modularity violation instance is shown in Figure 2. This DSM reveals that although these files have very few structure dependencies (only 13 out of 729 pairs of files have structural relations), they changed together very often, indicating the existence of strong implicit dependencies among these files.

V. Architecture Debt Verification

To investigate if the 6 file groups reveal true architecture issues that are considered significant and worth treating, we exported their 6 DSMs into spreadsheets and returned them to the SoftServe architects as possible architecture hotspots (potential architecture debts) for them to verify. We also compared the files within these 6 DSMs with technical debt files reported by Sonar for further analysis.

A. Debt Verification by SoftServe Architects

We returned these architecture debt candidates to our collaborators along with the following questions:

Q1: For each instance, is this a real design/architecture problem with significant maintenance costs? If yes, do you plan to refactor and fix the issues in them?

Except for one improper inheritance instance with 7 files, the architect confirmed that all other instances were real architecture issues. They agreed that two of the improper inheritance instances indeed revealed that these files were over-designed. Since these instances have small numbers of files, and thus limited maintenance costs, we focused on the other three bigger architecture issue instances, and refactoring these three instances has been planned.

Q2: Are there any issues we identified but which were not revealed by other tools in use?

The architect pointed out that the modularity violation instances we identified (Figure 2) revealed deep problems that were not detectable by other tools. It revealed a poor design decision that caused large number of co-changes among large number of seemingly unrelated files.

When one of the Unstable Interface instances was reported, our collaborators realized that the interface file leading this DRSpace, Pear.java was overly complex, turning into a God interface, and recognized that Divide-and-conquer would be the proper strategy to refactor this DRSpace.

The feedback from the architect is extremely encouraging. Since we have reported these architecture issues, they have spent much effort devising strategies to address these debts. As we will show, we were also able to extract more detailed data to quantify the cost and benefits, making it possible to make a business case targeting at the refactoring of these localized hotspots.

B. Debt Comparison

The most significant difference between our approach and that of other technical debt detection tools, such as Sonar, is that, we identify debts as architecturally related groups (in our current case study, we reported 3 major file groups). We use DSMs to visualize the architecture problems linking these files together, indicating how defects may propagate between them. Sonar reports a list of files, without showing the relations among them. Although the architecture issue instances we identified have been confirmed by SoftServe architects, we were curious to know whether the files comprising these architecture issue instances could be found by Sonar.

We only compared with Sonar, since it is the *de facto* industrial standard for detecting technical debt. SoftServe also used other methods to identify other types of technical debt, like checking the “Todo” comments in source code, but we don’t consider those as comparable to architecture debts. We chose the three most common metrics used by Sonar to find files in debt: lines of code, McCabe complexity, and duplicated lines. These metrics were also used by SoftServe to identify technical debts, prior to this case study. In a technology assessment report created before our interaction with SoftServe, they listed their 21 “fattest” (most complex) files. These files, reported by Sonar, were “considered as refactoring priority candidates” by SoftServe. But their report did not show the relations among these fattest files, nor their impact scope.

Our purpose is to understand (1) whether files reported by Sonar also suffer from architecture issues, and (2) whether high-maintenance files have been missed by just detecting files involved in architecture hotspots. For Sonar, we took the top 10 percentile most complex files (LOC or McCabe), as well as the top 10 percentile files with the most duplicated lines, and took the union of these sets to form a final set of 98 files as the debts identified by Sonar. We compared these 98 files (which we call *SonarDebts*) with the 95 files (*TitanDebts*) that we reported to SoftServe as being directly implicated in architecture issues.

We first compare the precision and recall of both of these file sets against the Bug2 (55 files) and Change10 (63 files) spaces. The Bug2 and Change10 file sets served as the “oracle” for this study, since these are the ground truth set of files that are causing problems in the project. Our reasoning is that, the more files detected by a technique that are truly error-prone or change-prone (high precision), and the more high-maintenance files that are detected (high recall), the more effective the technique is. The result of this study showed that the precision of TitanDebts is 31% vs. 18% for SonarDebts using Bug2 as the oracle, and 40% for vs. 27% using change10 as the oracle. The recall of TitanDebts vs. SonarDebts is 53% vs. 33% for Bug2, and 60% vs. 41% for Change10. These data indicate that Titan consistently performs better, in terms of capturing the most error-prone and change-prone files.

The precision of any single technical identification technique is likely to be low, because files might be buggy or

change-prone for a number of reasons: because of architectural complexity, because of code complexity, or because of inherent domain complexity. For TitanDebt, not all files involved in architecture issues have high maintenance costs; for SonarDebt, not all files that are complex are necessarily error-prone or bug-prone. In addition, the precision numbers reported here are low because of the small sizes of Bug2 and Change10. Precision is a measure of what fraction of the retrieved results are relevant. Since the Bug2 and Change10 sets are smaller than SonarDebts and TitanDebts (due to the relatively short project history that we were considering) the highest possible precision value for Bug2 would be about 57% and the highest precision value for Change10 would be 66%.

Next we examined the overlap between TitanDebts and SonarDebts. There are 25 files found in the intersection of TitanDebts and SonarDebts. These 25 files are undoubtedly the most problematic ones in the project: they are both complex and architecturally problematic. The fact that the intersections of these two sets only have about 1/4 of their total number of files indicates that Sonar and Titan detect substantially different, and complementary, sets of files.

In summary, from this comparative analysis we can observe that the architecture instances we detected capture file groups with higher error-proneness and change-proneness than what Sonar captured. In addition, a significant portion of the files with severe, high-maintenance architecture issues, detected by our tool, are missed by Sonar. We are not aware of any other tools that can detect those files, together with their visualizable architecture issues.

VI. Architecture Debt Quantification

Now that the 3 instances of architecture issues are verified to be true architecture debts, the architect at Softserve needs to estimate the economic consequences of these debts, to make decisions regarding to whether it is worthwhile to refactor. We first need to determine the scope of the debts. That is, how many files are influenced by these architecture flaws? Since each DRSpace contains all the files that are directly and indirectly impacted by the leading files, the scope of the 3 architecture debts should be the DRSpaces led by the leading files of these instances. In this case study, the scope should be all the 291 distinct files contained in 3 DRSpaces led by Pear.java, Apple.java, and Bean.java.

Next we need to quantify the unit of "effort" or "cost". Here we needed to make some assumptions and collect project data based on those assumptions. Like most industrial projects, effort data was not carefully collected, and was not associated with file-level work. So while our DRSpaces technique was based on the file as the most basic unit of analysis, we could not collect true effort data on a per-file basis.[4] For this reason we chose to collect other types of file-level data that *were* available:

- number of resolved defects per file
- number of completed changes per file
- number of modified/added/deleted lines of code per file, to fix defects and make changes

These measures have a number of advantages in terms of supporting an economic analysis: they are objective, they are easily gathered and counted in a fully automated fashion, and they are broadly available in most industrial and open source projects. Given this background we were able to collect data associated with the three most problematic DRSpaces in the SoftServe project, led by Apple.java, Bean.java, and Pear.java. The data that we collected, and our analysis of this data, is shown in Figure 3. We will refer to this figure repeatedly in explaining how we calculate technical debt and how we supported the project's architect and manager in making refactoring decisions.

To begin, we needed to calculate the size of each DRSpace. As explained in [30], DRSpaces are overlapping. The Design Rule Hierarchy clustering algorithm simply clusters files that "follow" the leading file or files—the design rules. Since any given file (class) may participate in many relationships with other files, this may result in the same file appearing in multiple DRSpaces. As a consequence, we calculated two measures of the size of each DRSpace: the raw size, in terms of number of files, and a normalized size. The raw size for the DRSpaces led by Apple.java, Bean.java, and Pear.java are presented in cells B2-B4 of Figure 3. To normalize the size, we considered that any file that is included in two DRSpaces should be counted as $1/2$ a file for each DRSpace. If a file participates in three DRSpaces, it is counted as $1/3$ in each one. In this way the impact of a file—its set of defects and changes—is shared among the DRSpaces. This normalization is necessary because if we were not to do this we would be double (or triple) counting the impact of files that participate in more than one DRSpace. The normalized size figures are shown in cells C2-C4 of Figure 3. These three DRSpaces represent a total of 291 files, out of a total of 797 files in the entire project (as shown in cells C6 and B7 respectively).

Now we are ready to calculate the penalty of the debt within each of the DRSpaces. We first count the number of defects associated with each DRSpace that we actually fixed during the prior year. This information is easily retrieved from the project's revision control and defect-tracking systems. These values are shown in cells D2-D4. However, since we do not want to double-count any defect associated with a file that appears in more than one DRSpace, we normalize the number of defects by multiplying the raw defect count associated with each DRSpace by the fraction of normalized DRSpace size divided by actual DRSpace size. The normalized defect counts are given in cells E2-E4 of Figure 3, and their total is shown in cell E6. Note that the normalized number of defects associated with these three DRSpaces is 89% of the project's total number

[4]We have assumed, for the purposes of our analysis, that the ratio of files to classes is 1:1. This has broadly held true for the 30 or so systems that we have analyzed to this point. If, for some reason, a project deviated from this convention, we could simply normalize the counts of defects, changes, and lines of code, to account for a different ratio.

	A	B	C	D	E	F	G	I	J	K	L	M	N
1	***DRSpace Leading File***	**DRSpace Size**	**Norm Size**	**Current Defects/Yr**	**Norm Defects**	**Current Changes/Yr**	**Norm Changes/Yr**	**Tot LOC Changed**	**Norm LOC Changed**	**Refactor Cost (PM)**	**Norm Exp Defects/Yr**	**Norm Exp Changes/Yr**	**Norm Exp LOC Changed**
2	*Pear.java*	139	119.33	166	142.5	1068	839.2	49,171	42,213	5.5	39	346	20,281
3	*Apple.java*	158	133.83	63	53.4	607	451.7	25,603	21,686	7	44	388	22,745
4	*Bean.java*	65	37.83	72	41.9	429	207.2	17,807	10,364	1.5	12	110	6,429
5													
6	**DRSpace Total**		**290.99**		**237.8**		**1498**		**74,263**		**96.0**	**843.871**	**49,455**
7	**Project Total**	**797**		**265**		**2332**		**135,453**		**14**			
8	**Savings**										**142**	**654**	**24,808**
9													
10													
11	Base defect rates	0.33											
12	Base change rates	2.9										**Exp PM saved**	**41.35**
13	Base LOC/file	169.95											
14	LOC/PM	600											

Fig. 3. Technical Debt Calculation Framework

of defects (which is 265), even though the normalized size of these DRSpaces—291—is just under 37% of the entire project.

Similarly we count the total number of changes affecting the files in the three DRSpaces over the past year, and we normalize these as we normalized the numbers of file and defects. The raw numbers of changes are shown in cells F2-F4, and the normalized values are given in G2-G4. Note that the total normalized number of changes affecting Apple.java, Bean.java, and Pear.java is 1498, or about 2/3 of the project total of 2332. This is consistent with our prior research, where complex, problematic DRSpaces account for far more than their share of defects and changes, and require many more lines of code to modify and fix than average project files [21], [30].

Finally we show the number of lines of code committed to fix the defects and to make the changes for the files in these three DRSpaces over the past year. The raw numbers of lines of code are given in I2-I4 and the normalized values are given in J2-J4.

Another key parameter needed to make refactoring decisions is the cost of refactoring. The chief architect of the SoftServe system agreed that not only are these DRSpaces problematic, but also agreed that the architectural flaws that we identified were indeed design problems, violating standard rules of good design such as the Law of Demeter, the Open/Closed principle, and so forth. Using the architectural flaws as a guide, a set of refactorings were determined and the chief architect made effort estimates for each of these refactoring efforts. These effort estimates, in person months (PM) are shown in cells K2-K4 of Figure 3 (highlighted in orange). The total effort for the refactorings is given in cell K7.

We have thus far calculated the "penalty" being incurred by these three DRSpaces, as a result of their architectural flaws, and the chief architect has estimated the cost to refactor these DRSpaces. Now we turn to the issue of estimating the benefit that we expect to accrue from this refactoring.

We used, as a basis for the estimate, existing project averages, shown in cells B11-B13. An *average* file in this project is subjected to 0.33 defects annually (i.e., there were 265 defects affecting a total of 797 files) and 2.9 changes annually (i.e. 2332 changes over 797 files), requiring 169.95 lines of code to resolve (there were a total of 135,453 lines of code committed for the project's 797 files).

Our assumption is that the refactoring of the three DRSpaces will bring them down to project averages. This is, we feel, a very conservative assumption for two reasons: 1) the current project averages already include these flawed DRSpaces, which inflates the averages, and 2) it is likely that the refactoring could result in much better structure than the project average, since the average project file has *not* been refactored. Thus the refactoring could conceivably result in lower defects, changes, and committed lines of code for these three DRSpaces. For these reasons we feel that using existing project averages as our "target" for improvement is very conservative. Our follow-on longitudinal study will allow us to test the validity of this assumption.

Based on this assumption, we can now calculate the expected benefit from the refactoring these three DRSpaces. Cells L2-L4 list the expected numbers of defects that would affect each of the problematic DRSpaces—34, 34, and 10—assuming that the refactoring brought them down to project averages. Similarly cells M2-M4 and N2-N4 show the normalized, expected numbers of changes and committed lines of code under the same assumption—that the refactored DRSpaces exhibit project average behaviors. The totals for the expected numbers of defects, changes, and lines of code are presented in cells L6, M6, and N6.

Now we are in a position to calculate the expected benefit from these refactorings. The benefit is the difference between the actual annual numbers of defects, changes, and committed lines of code and the expected numbers of defects, changes, and committed lines of code. These expected "savings" are given in cells L8, M8, and N8. Let us ignore the loss of time and reputation due to bugs that are avoided (L8) and focus purely on the lines of code that we expect the project will not have to commit, due to the refactoring (N8). The project can conservatively expect to save 24,808 lines of code by refactoring the three problematic DRSpaces. Now we take company average productivity numbers and using this to calculate the expected person months of effort avoided as a

result of the refactorings. This savings is shown in cell N12. The project can expect to save 41.35 person months of effort per year due to the proposed refactorings. Given that these refactorings are estimated to cost just 14 person months of effort, the investment in refactoring is paid off in less than 1/2 year and the project experiences a net benefit thereafter. Or, to put it in financial terms, the project can expect a 295% return on investment in the first year alone.

This is, to our knowledge, the first time that the penalty associated with technical debt, the cost of refactoring to remove that debt, and the expected benefits of removing the debt have been quantified based on hard data—project-specific empirical evidence. Of course, there are assumptions wrapped up in the estimates, but this is true of any financial estimates in any field. These assumptions are supported by our prior research, but they are assumptions nonetheless. As we collect more data we will be able to report on the validity and stability of these assumptions.

VII. Results and Lessons Learned

Now we are in a position to answer the research questions that we posed in section II.

Regarding RQ1: According to the opinions of the SoftServe's architects, the set of architectural issues that we reported to SoftServe were truly problematic. They often had a vague idea that a region of the architecture was "troublesome" or "hard to maintain", but they were unable to precisely identify the problems and their scope.

Regarding RQ2, the results returned by the Titan tool chain did differ significantly from the files reported as sources of technical debt by SonarQube. The precision and recall of Titan outperformed that of Sonar by 50% or more, when compared with Bug2 and Change10.

Finally, we feel that the answer to RQ3 was perhaps the most important outcome of this case study. It is indeed possible to quantify the return on investment of removing architecture debts. We were able to mine project data to estimate the penalty incurred by the debts (hotspots) identified by Titan, and to calculate the expected benefits if the debts are removed. When we compared this with the costs of refactoring it made a compelling argument for SoftServe's management, who immediately chose to refactor the system in the areas we identified.

What have we learned, having worked through this process with an industrial partner? We have gathered several important lessons.

The first, and perhaps most important lesson is that the analysis we did here was not remarkable; it is easily repeatable. It does not depend on the skills of the analyst; it simply depends on having the appropriate input data. The good news is that most projects have enough data to make this determination: that is, they have source code that can be reverse engineered to extract file dependencies, they have revisions control systems that show which files were committed and how many lines of code were modified, and they have issue tracking systems that show and classify the reported project defects and change or feature requests. What not all projects have is the ability to trace among these project records. If the project does not have the discipline to always associate a commit with an issue number from the issue-tracking system then we can not trace from file to commit to bug or change. Thus, one of our lessons learned is that we can influence projects to improve their record-keeping practices. We can influence them because we can show them how such tiny and inexpensive changes in their processes can result in greater insight into the sources of project technical debt.

The second important lesson that we learned is that technical debt can arise from a variety of sources, and no single tool or approach is going to find all of them. Code-based approaches will tend to find one class of problems, dealing with (not surprisingly) code-level issues—poor code structure, repeated lines, lack of comments, and so forth. But another important source of technical debt comes from architectural problems and the code-based analysis tools do not find this debt.

The third (related) lesson that we learned from the project, and also from many other interviews with practicing architects is that architectural technical debt is extremely common. Like rust, it never sleeps; it just accumulates in projects, unless some conscious refactoring effort is made. This is because architectural debt is extremely easy to introduce, and extremely difficult for a programmer to discern. A programmer typically wants to fix a bug or introduce some new feature or function. In doing so they create new classes, modify existing classes, add relationships between existing classes, and so forth. Some of these changes inevitably undermine the architectural structure, even if this structure was not consciously described. The structure slowly becomes more complex, more highly coupled, less cohesive. Unfortunately, refactorings to fix these debts are seldom made because the architects typically do not know: 1) how to locate the debts, and 2) how to create a business case that presents compelling evidence for the *value* of refactoring. By arming SoftServe's architects with such information they were able to make a compelling business case which was immediately accepted and acted upon.

VIII. Related work

Our work is related to the work of technical debt detection, architecture analysis, and defect localization and prediction.

a) Technical Debt Detection: To locate technical debt in code, a number of heuristics have been proposed. These heuristics attempt to identify characteristic problems in code—such as clones, long methods, and god classes—that can be detected by code analysis tools such as SonarQube. But not all of these code problems are certain to cause maintenance or quality problems. In fact, no existing work has been able to accurately locate the sources and estimate the magnitude of technical debt. For example, Zazworka et al. [33], compared four different technical debt detection approaches and found that only a subset of the debt detected by the four approaches were strongly correlated with software changes and defect proneness.

The concept of a "bad smell" was first proposed in 1999 as a heuristic for identifying redesign and refactoring opportunities [7]. Code clones and feature envy were examples of smells proposed in this work. Others [9] have extended this notion to include architecture-level bad smells. But to detect debt efficiently, the approach must be automatable. For example, Moha et al. [22] created the Decor tool to automate the creation of design defect detection algorithms. In addition, some research has proposed automatically detecting bad smells that suggest refactorings. For example, Tsantalis and Chatzigeorgiou's static slicing approach [27] aims to detect extract method refactoring opportunities. In addition, some common smells, such as code clones, have been extensively studied, such as Higo et al. [12]'s Aries tool to identify code clones as candidates for refactoring.

Our architecture debt detection approach, however, is different. First, our approach focuses on the structure among files, rather than the internal problems within a file. Not all files involved in architecture issues have bad smells. Second, existing research on bad smells has always focused on analyzing a single version of the software, while our approach examines the project's evolution history. We can thus focus on the most recent and most frequently occurring architecture problems, and detect architecture issues that can only be exposed during evolution, such as Implicit Cross-module Dependency and Unstable Interfaces. Neither can be detected by examining a single version of a code base.

b) Architecture Representation and Analysis: The ground truth of the architecture of a software project is usually difficult to acquire; architecture documentation is rarely up-to-date or accurate. A software system contains multiple architectural structures that may be documented as views [2], [6], [18]. But the views proposed in prior work are general-purpose. To locate and diagnose specific modularity debt, we need to focus on just a single architecture view—the module view. Within the module view DRSpaces are organized based on design rules and independent modules.

Methods supporting the analysis of architecture have been widely studied. The majority of architecture analysis methods created to date have either focused on questionnaires [20] or scenarios [14], [16]. For example, Kazman et al. [16] created the Architecture Tradeoff Analysis Method (ATAM) for analyzing architectures. This was extended with the Cost Benefit Analysis Method (CBAM) [15], [24] so that the technical analysis of an ATAM could be informed by the costs and benefits of proposed architectural strategies, as a means of determining an optimal project evolution path. Andrew [17] proposed anti-patterns to represent recurring problems that are harmful to software systems. These methods are manual, and depend heavily on the skills of highly trained and experienced architecture analysts. Our approach, by contrast, detects architecture issues automatically and can guide the user, helping them to locate and diagnose software quality problems. Furthermore, we assist the user in analyzing the economic consequences of these problems and their repairs. And this analysis requires only project data that is easily available.

c) Defect Localization and Prediction: Numerous work has been proposed to locate and predict software defects using dependency relation, history, or metrics [11], [13], [19]. Selby and Basili [26] first explored the relation between dependency structures and software defects. The relation between evolutionary coupling and error-proneness has also been extensively studied [5], [8], [10]. Cataldo et al.'s [5] reported a strong correlation between change coupling density and failure proneness. Ostrand et al. [25] demonstrated that file change history can be used to effectively predict defects. Nagappan et al. [23] used complexity metrics to predict defects, but admitted that in different projects, the best metrics for prediction can be different. Different from these prior work that all focus on individual files as the unit of analysis, our approach reveals architecture flaws that propagate errors among files.

IX. Conclusions and Future Work

Our case study with SoftServe has confirmed our research hypotheses: we are able to locate the architectural sources of technical debt, quantify them, and quantify the expected payback for refactoring these debts. We did this based solely on data that was already available within SoftServe. The evidence that we produced and the arguments that we made based on this evidence were compelling to SoftServe's management, who immediately decided to invest in the proposed refactorings. One might object that these estimates are just that–estimates. However, all decision-making in business involves investment under uncertainty. And even if our ROI estimate is off by an order of magnitude—that is, if it was merely a 30% ROI—it still represents an excellent choice for the company, which presumably can not earn such a high ROI through any traditional means.

Our future work consists of a longitudinal study wherein we do four things: First, we will track the architectural integrity of this system on a regular basis. That is, we plan to analyze periodic snapshots of SoftServe's system, to see whether the refactoring is being done correctly, and whether it is eroding over time. Second, we plan to continue to track the frequency of reported defects, and their connection to the files in SS1. Third we plan to continue to track the frequency of changes to the files of SS1. Finally, we plan to track the lines of code committed to fix defects and to make changes. This longitudinal data capture and analysis will allow us to validate the expectations and opinions collected in the present study, and to build better predictive models for SoftServe in the future. We are also in the process of conducting other industrial case studies, to show the repeatability of our methods in different industrial contexts.

In addition, we would like to examine the background trends of the data in future work. For example, are bug rates, change rates, and churn level, going up, or going down in the project, irrespective of any intervention?

For now, SoftServe is very happy with the outcomes and is taking all necessary steps to refactor their architecture to fix

the defects that our Titan tool has highlighted. The SoftServe architects felt that Titan provided insights, supporting data and, (most important) *explanations* that no other analysis tool had hitherto provided. These insights accorded with their experience of the system, and supported their intuitions about the problems with its architecture. But, more importantly, the combination of project-data-driven economic arguments and evidence-based identification of technical debts was compelling for SoftServe's architects and they plan to pursue this strategy with other systems right away.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation of the US under grants CCF-0916891, CCF-1065189, CCF-1116980 and DUE-0837665.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0002092

REFERENCES

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

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 3rd edition, 2012.

[3] Y. Cai and K. J. Sullivan. Modularity analysis of logical design models. In *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 91–102, Sept. 2006.

[4] Y. Cai, H. Wong, S. Wong, and L. Wang. Leveraging design rules to improve software architecture recovery. In *Proc. 9th International ACM Sigsoft Conference on the Quality of Software Architectures*, pages 133–142, June 2013.

[5] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, July 2009.

[6] D. Falessi, G. Cantone, R. Kazman, and P. Kruchten. Decision-making techniques for software architecture design: A comparative survey. *ACM Computing Surveys*, 43(4):1–28, Oct. 2011.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, July 1999.

[8] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. 14th IEEE International Conference on Software Maintenance*, pages 190–197, Nov. 1998.

[9] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *Proc. 13th European Conference on Software Maintenance and Reengineering*, pages 255–258, Mar. 2009.

[10] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[11] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272, 2005.

[12] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. In *Proc. 5th International Conference on Product Focused Software Development and Process Improvement*, pages 220–233, Apr. 2004.

[13] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. 24thInternational Conference on Software Engineering*, 2002.

[14] R. Kazman, G. Abowd, L. Bass, and M. Webb. Saam: A method for analyzing the properties of software architectures. In *Proc. 16th International Conference on Software Engineering*, pages 81–90, May 1994.

[15] R. Kazman, J. Asundi, and M. Klein. Quantifying the costs and benefits of architectural decisions. In *Proc. 23rd International Conference on Software Engineering*, pages 297–306, May 2001.

[16] R. Kazman, M. Barbacci, M. Klein, S. J. Carriere, and S. G. Woods. Experience with performing architecture tradeoff analysis. In *Proc. 16th International Conference on Software Engineering*, pages 54–64, May 1999.

[17] A. Koenig. *Patterns and antipatterns*. The patterns handbooks, 1998.

[18] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12:42–50, 1995.

[19] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *13rd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2005.

[20] J. Maranzano, S. Rozsypal, G. Zimmerman, G. Warnken, P. Wirth, and D. Weiss. Architecture reviews: Practice and experience. *IEEE Software*, 22:34–43, 2005.

[21] R. Mo, Y. Cai, R. Kazman, and L. Xiao. Hotspot patterns: The formal definition and automatic detection of recurring high-maintenance architecture issues. In *Submission*, 2014.

[22] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, and L. Duchien. A domain analysis to specify design defects and generate detection algorithms. In *Proc. 11th International Conference on Fundamental Approaches to Software Engineering*, pages 276–291, Mar. 2008.

[23] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. 28th International Conference on Software Engineering*, pages 452–461, 2006.

[24] R. Nord, M. Barbacci, P. Clements, R. Kazman, M. Klein, L. O'Brien, and J. Tomayko. Integrating the archi-tecture tradeoff analysis method (atam) with the cost benefit analysis method (cbam). Technical Report CMU/SEI-2003-TN-038, Carnegie Mellon University/SEI, 2003.

[25] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[26] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, Feb. 1991.

[27] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, May 2009.

[28] S. Wong and Y. Cai. Improving the efficiency of dependency analysis in logical models. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 173–184, Nov. 2009.

[29] 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.

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

[31] 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.

[32] L. Xiao, Y. Cai, R. Kazman, and R. Mo. Investigating the evolutionary consequences of architecture roots of error-proneness. In *Submission*, 2014.

[33] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull. Comparing four approaches for technical debt identification. *Software Quality Journal*, pages 1–24, 2013.