# Experiences Applying Automated Architecture Analysis Tool Suites

Ran Mo
Central China Normal University
Wuhan, China
moran@mail.ccnu.edu.cn

Will Snipes
ABB Corporate Research
Raleigh, NC, USA
will.snipes@us.abb.com

Yuanfang Cai
Drexel University
Philadelphia, PA, USA
yc349@drexel.edu

Srini Ramaswamy
ABB Inc.
Cleveland, OH, USA
srini@ieee.org

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

Martin Naedele
ABB Inc.
Baden, Switzerland
martin.naedele@ch.abb.com

## ABSTRACT

In this paper, we report our experiences of applying three complementary automated software architecture analysis techniques, supported by a tool suite, called DV8, to 8 industrial projects within a large company. DV8 includes two state-of-the-art architecture-level maintainability metrics—Decoupling Level and Propagation Cost, an architecture flaw detection tool, and an architecture root detection tool. We collected development process data from the project teams as input to these tools, reported the results back to the practitioners, and followed up with telephone conferences and interviews. Our experiences revealed that the metrics scores, quantitative debt analysis, and architecture flaw visualization can effectively bridge the gap between management and development, help them decide if, when, and where to refactor. In particular, the metrics scores, compared against industrial benchmarks, faithfully reflected the practitioners' intuitions about the maintainability of their projects, and enabled them to better understand the maintainability relative to other projects internal to their company, and to other industrial products. The automatically detected architecture flaws and roots enabled the practitioners to precisely pinpoint, visualize, and quantify the "*hotspots*" within the systems that are responsible for high maintenance costs. Except for the two smallest projects for which both architecture metrics indicated high maintainability, all other projects are planning or have already begun refactorings to address the problems detected by our analyses. We are working on further automating the tool chain, and transforming the analysis suite into deployable services accessible by all projects within the company.

## CCS CONCEPTS

• **Software and its engineering → Software architectures**;

## KEYWORDS

Software Architecture, Software Quality, Software Maintenance

**ACM Reference Format:**
Ran Mo, Will Snipes, Yuanfang Cai, Srini Ramaswamy, Rick Kazman, and Martin Naedele. 2018. Experiences Applying Automated Architecture Analysis Tool Suites. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3240467

## 1 INTRODUCTION

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. In this paper, we present our experience of applying three automated architecture analysis and quantification techniques, supported by a tool suite, called DV8[1], on eight large-scale projects within *ABB*.

The first technique is measuring architecture maintainability using a pair of architecture-level maintainability metrics: *decoupling level* (DL) [16] and *propagation cost* (PC) [13]. 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. Both metrics were proposed recently by different researchers. Applying both to the same projects could help us evaluate which metric is more effective and reliable, and if and how they can reveal different aspects of the same project.

The second technique is *architecture flaw* detection. Mo et al. [15] formally defined a set of architecture flaws that incur high maintenance costs. Files involved in such flaws suffer from one or more architecture design mistakes; these flaws have significant impact on the bug-proneness and change-proneness of the system. To make

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

*ASE '18, September 3–7, 2018, Montpellier, France*
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240467

the penalty incurred by these flaws explicit, we quantify the number of bugs and changes, as well as the bug churn and change churn, for each flaw using project history data. The users can also visualize each flaw as a *design structure matrix* (DSM) [1, 23].

The third technique we applied is *architecture root* analysis proposed by Xiao et al. [24]. They proposed the notion of a *design rule space* (DRSpace)—a set of architecturally connected files to implement a pattern, a feature, or other important system concerns. They also proposed the concept of *architecture roots*—design rule spaces that cluster together the most error-prone files in the system. As reported by Xiao et al. [24], five architecture roots in a project almost always cover 50% to 90% of the most error-prone files within a project.

In *ABB* we assembled and integrated these tools to create our automated architecture analysis framework. Using this framework, we analyzed eight projects within the company. These projects are developed at multiple locations (India, USA, and Switzerland) and differ in their ages, domains, and sizes. Our study had the following steps: first, the development teams of *ABB* granted us access to their code repository, from which we collected file dependency information, history data, and work items. Using these data as input, we ran DV8, which automatically generated metrics scores, and visualizable architecture flaws and roots along with supporting quantitative 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 as we will show. The architecture flaw detector can effectively pinpoint which files are suffering from which specific 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 detected flaws. Finally, the feedback we received regarding root analysis is divided: some teams found that it provided an effective way to detect architectural problems since they only needed to examine five groups of related files. But other teams found that a root can be misleading when there are highly influential utility files that may distort their results.

## 2 RESEARCH QUESTIONS

Our goal in employing these analysis techniques within *ABB* was to investigate the following research questions:

- RQ1: does DV8 help to close the gap between management and development? That is, does it help them to decide if, when, and where to refactor?
- RQ2: does DV8 help practitioners understand the maintainability of their systems relative to other projects internal to the company, and relative to a more broad-based benchmark suite?
- RQ3: does DV8 help developers pinpoint the *hotspots* of their systems—that is, the groups of files with severe design flaws?

We investigated these questions by interviewing the development teams, analyzing their experiences with the provided tools, and soliciting feedback. This interview process also allowed us to understand the limitations of DV8.

## 3 PROCEDURE

In this section, we present the projects where we applied the architecture analysis framework, the data needed to run the tool suite, and the overall structure of the analysis framework as shown in Figure 1.

**Subjects.** Table 1 presents the eight projects that we analyzed. The "*Lang.*" column shows the main language used. The "*#Files*" column shows the number of files in the project. For *Proj_CH* and *Proj_EC*, we measured multiple releases of each, so we listed the *range* in the number of files in these projects. The column "*#Com.*" presents the number of commits over the time period studied. The column "*Period*" shows the period of time we analyzed for each project. The column "*#P*" presents the number of people who made commits during this time period. Consider *Proj_BM* as an example: it contains 371 source files, and the history studied is from April 2015 to July 2017. It was maintained by 8 full-time developers, who made 536 commits. The real project and file names are anonymized in this paper.

**Table 1: Studied Projects**

| | Lang. | #Files | #Com. | Period | #P |
|---|---|---|---|---|---|
| Proj_EO | C# | 144 | 953 | 05/15-01/17 | 8 |
| Proj_BM | C/C++ | 371 | 536 | 04/15-07/17 | 8 |
| Proj_CH | C/C++ | 6,242-6,948 | 2,568 | 01/11-01/17 | 28 |
| Proj_EP | C# | 1,541 | 1,668 | 03/15-04/17 | 21 |
| Proj_SS | C/C++ | 15,333 | 6,400 | 02/13-10/16 | 30 |
| Proj_OP | C/C++ | 7,754 | 39,074 | 01/11-01/17 | 86 |
| Proj_CO | C# | 491 | 360 | 03/16-04/17 | 16 |
| Proj_EC | C/C++ | 2,493-4,125 | 1,043 | 08/13-02/16 | N/A |

**Data needed for the tool suite (DV8).** Once a development team agreed to participate in our study, they granted us access to their code repository, and the specific version(s) they wanted to analyze. From the repository, we extracted three types of data as the input to DV8, as shown in Figure 1:

(1) *The dependency information among source files.* Given a project source code, we first used *Understand*[2], a commercial static analysis tool, to generate a file-level dependency report in XML format for each project version to be analyzed.

(2) *The revision history of the project.* We extracted the revision history of a project from the *Team Foundation Server*[3] (TFS) system used in ABB. TFS records each commit as a changeset and provides console commands through which we can export the changeset into a plain text file. This file records, for a specified time period, which files were changed together in which commit, and how many lines of code were changed in each file.

[2]https://scitools.com/
[3]https://www.visualstudio.com/tfs/

(3) *Bug records.* TFS also has a *work item* tracking system where the developers can record bugs or other tasks, such as adding a new feature. When a developer makes a commit, he/she can link the commit (changeset) to a work item. From the *work item* tracking system, we can download all the bug reports and their associated changesets in XML format.

The XML files recording dependency information among source files, the plain text file recording all changesets in the revision history, and the XML file recording bug fixes were the input needed by the automated tool suite, as shown in Figure 1.

**Automated architecture analysis framework**. This framework contains the following components:

(1) *SDSM generator.* The representation used by DV8 is a *design structure matrix* (DSM), first proposed by Baldwin and Clark [1]. We will elaborate the model using examples in Section V. The SDSM (short for *structure DSM*) generator transforms a XML file containing file dependency information into a DSM.

(2) *HDSM generator.* This component transfers the plain text file recording changeset information into a DSM format, which we call a HDSM (short for *history DSM*), so that both structural and history information can be processed simultaneously.

(3) *DL & PC calculators.* These two calculators take a SDSM as input and output the decoupling level and propagation cost scores respectively, calculated using the file dependencies output by *Understand.*

(4) *Flaw detector.* This component takes both a SDSM and a HDSM as input, and generates a set of new DSM files, each of which contains an architecture flaw (which we will describe in Section V), as defined in Mo et al.'s work [15]. Each flaw DSM can also be exported into a spreadsheet for further analysis and broad dissemination.

(5) *Root detector.* This component takes a SDSM, a HDSM, and the changeset file as input, and generates a set of DSMs, each containing an architecture root [24] capturing the most change-prone or bug-prone files in the system. The change-proneness or bug-proneness of files can be ranked by the number of changes/bugs to any given file, which can be calculated from the project's changeset file. A DSM containing the root can also be exported into a spreadsheet.

(6) *Flaw cost calculator.* This component extends the *flaw detector* component by calculating the number of LOC, the number of changes, and the number of bug fixes incurred by each flaw. It takes the output of the *flaw detector* as input, and uses the changeset file and bug records to calculate the maintenance costs related to each flaw. It outputs the results into a spreadsheet, as we will elaborate later.

(7) *Debt calculator.* Inspired by the experiences reported in [12], this component calculates the expected number of additional bugs, changes, and churn incurred by a root or a flaw, as compared to system averages. The output of this component is also a spreadsheet.

(8) *Report generator.* This component automatically puts together a report by summarizing the output of the other components, including the metrics scores, the summarization of flaws and roots, and the costs and debts. We can manually add project-specific prose, as needed, to the report before sending it back to the development team, along with supporting data. For each project, the report contains its basic facts, DL and PC values with the corresponding percentile rankings, detected flaws and roots with the involved files, and the costs and debts. We then present the report and the associated data to the development team to help them properly interpret the results.

Of all the 8 components in DV8, the first five were obtained from the researchers who originally created these technologies. The final three components are extensions newly developed for ABB. All these 8 components have been integrated into our framework to automate the architecture analysis.

After the development team had time (at least one week) to fully digest the report, we set up a follow-up interview to evaluate the effectiveness of the tools and their results. We asked the architects and project managers a set of pre-defined interview questions, as explained in section 7, so that we could confidently answer the research questions posed in section 2.

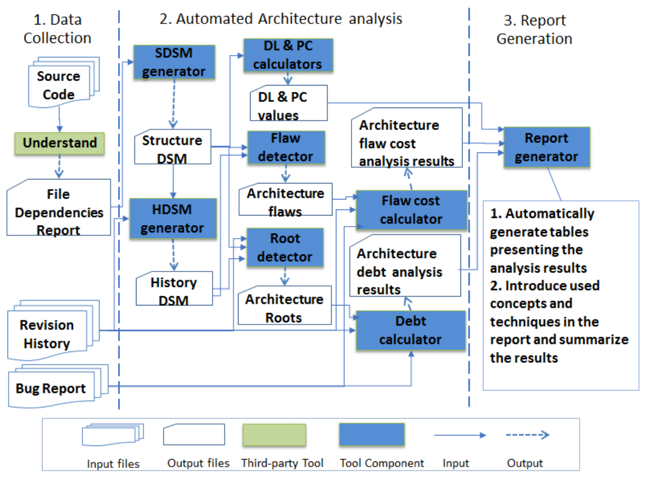


Figure 1: Analysis Framework

## 4 ARCHITECTURE MEASUREMENT

We first measured the maintainability of each project using the *DL and PC calculators* as shown in Figure 1, and compared each project's scores with an industrial benchmark suite, so that both the management and development team can understand how their project compares to others in industry. For projects where multiple releases were available to us, we also calculated the variation of DL and PC over time to see if the trend matched the practitioners' intuitions. Next we first introduce these two metrics, and then present the results.

### 4.1 Two Architecture-level Metrics

*Decoupling Level (DL)* was proposed by Mo et al. [16] to measure how well a software system is *decoupled* into independent modules. The theoretical foundation behind DL is Baldwin and Clark's design rule theory [1]: the more independent, small, and active modules there are, the higher option values the system can produce. Based on this rationale, their algorithm first clusters all source files into a design rule hierarchy (DRH) [5, 23], a hierarchical structure with two features: (1) files in lower layers only depend on files in higher layers; (2) files within the same layer are grouped into mutually independent modules. Based on DRH, DL is calculated as follows: the more independent modules there are, the higher the DL; the larger a module is, the lower its DL. For a module that influences others, the more dependents it has, the lower its DL.

*Propagation Cost (PC)* was proposed by MacCormack et al [13] to measure how tightly *coupled* a system is. The calculation of a system's PC value is based on a design rule matrix [1], whose rows and columns are labeled with the files in the same order. Based on this matrix, their algorithm first represents the direct dependency relations among files in a system, and then calculates its transitive closure to add indirect dependencies to the matrix until no more dependencies can be added. A nonempty cell in the matrix indicates an indirect or direct dependency between the file on the row and the file on the column. Given the final matrix containing all direct and indirect dependencies, PC is calculated as the number of nonempty cells divided by the total number of cells. PC has been used to analyze large projects with similar domains and sizes [13].

Mo et al. calculated the PC and DL of 129 projects [16], and published the data as an industrial benchmark. Figure 2 depicts the cumulative distributions of benchmark DL and PC values. The red solid line represents the probability that a random DL value is less than or equal to a specific value. For example, the mark "80*th*, 74.9%" on the red solid line indicates that 80% of the projects in the benchmark data have DL scores less than or equal to 74.9%. The blud dashed line represents the probability that a random PC value is larger than or equal to a specific value. For example, the mark"80*th*, 7.7%" at the blue dashed line indicates that 80% of the projects in the benchmark data have PC scores larger than or equal to 7.7%. The figure shows that DL and PC complement with each other: a ***high*** DL and a ***low*** PC indicate ***better*** modularity and maintainability, and a ***low*** DL is usually associated with ***high*** PC, indicating ***lower*** level of modularity and maintainability.

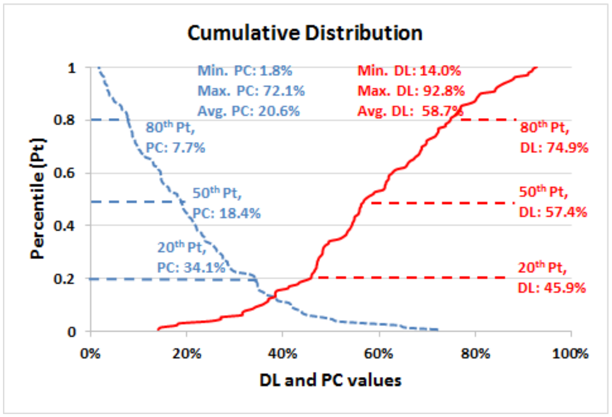


**Figure 2: Cumulative probability distribution of DL and PC**

## 4.2 Measuring and Ranking of Maintainability

For each project, we first calculated its DL and PC scores, then calculated its percentile ranking as compared with the benchmark data. For example, the DL and PC of Proj_EO are 78% and 6% respectively, ranked the 85th among the 129 projects for both metrics, meaning that its modular structure is better than 85% of the benchmark projects. Table 2 summarizes the DL and PC scores for the latest version of each project, showing their metric values, percentile rankings, and the number of files.

The row for *Proj_EP* shows that its DL is 72%, better than 74% of all benchmark projects. The PC of this project is 7%, lower (better) than 83% of the benchmark projects. As we can see from the table, in general, a higher DL is associated with a lower PC and their percentile rankings are largely consistent, differing by less than 10 percentile points. There are exceptions, however, such as Proj_CH: its DL is 76% (ranking 81st of all projects), but its PC is 16%, only ranking the 54th.

Considering that this project has 6,948 files, changes to one file may affect—directly or indirectly—approximately 1,000 files, suggesting that this part of the system will suffer from considerable maintenance difficulty, even though the majority of the system is reasonably decoupled. This observation was confirmed by the development team. As we discuss later, using architecture *flaw* and *root* analyses, we were able to pinpoint the file groups, and their architecture flaws, that are responsible for this maintenance difficulty.

For two of the eight projects the product organization requested that we calculate the DL and PC values of multiple snapshots to assess whether the architecture is degrading or not, and if the assessment is consistent with the practitioners' intuitions. For *Proj_CH*, the DL of its latest release was 76%. Since then, the system has been changed considerably. The development team asked us to measure a more recent working version and obtained a DL of 64%, showing that the architecture has degraded since the last release. Its architect confirmed that the degradation was expected as the release was still in alpha testing after implementing a major technology improvement. The project team plans to continue working on refactoring the code to improve the architecture.

We also calculated the scores for six releases of *Proj_EC*. All these releases have very low DL scores (26% - 30%) and high PC scores (60% - 68%). The practitioners confirmed that this project has always been difficult to maintain: a seemingly simple change often caused a large number of unexpected changes involving multiple files. For the latest version of Proj_EC, its DL is only 28%, ranking in the 5th percentile, and is the worst of all eight projects. One practitioner commented: *"... revising even two lines of code would require double digit man-months"*, indicating that the project has been extremely difficult to understand and maintain.

We reported our results to the architect of each project and asked whether DL and PC analyses faithfully reflect the maintainability of their projects. All the architects confirmed that the DL and PC scores indeed reflected their knowledge of the software architecture, and further helped them to present the architecture quality issues to management in a quantitative way. All four projects whose DL values ranked lower than the 50th percentile reported that they were experiencing severe maintenance difficulty. Of the eight projects, other than Proj_EO and Proj_BM that are small and have high PC and DL values, the other six project teams have decided, or have already begun refactoring to address the problems presented in our reports. The architects also expressed their willingness to leverage DL and PC metrics to continuously measure (quarterly or even at every release) the architecture of their projects. By tracking the variations of DL and PC values regularly, they believe that they could monitor whether their architecture is improving or decaying.

**Table 2: Data of each project's DL and PC**

| | DL | Percentile | PC | Percentile | #Files |
|---|---|---|---|---|---|
| Proj_EO | 78% | 85th | 6% | 85th | 144 |
| Proj_BM | 77% | 85th | 2% | 98th | 371 |
| Proj_CH | 76% | 81st | 16% | 54th | 6,948 |
| Proj_EP | 72% | 74th | 7% | 83th | 1,541 |
| Proj_SS | 57% | 49th | 20% | 45th | 15,333 |
| Proj_OP | 57% | 49th | 21% | 41th | 7,754 |
| Proj_CO | 55% | 43rd | 17% | 52th | 491 |
| Proj_EC | 28% | 5th | 62% | 2nd | 4,125 |

## 5 ARCHITECTURE FLAW ANALYSIS

The DL and PC scores only provide a coarse assessment of a project's overall modularity. But a development team needs to precisely determine where and how the system should be improved. In [15], Mo et al. formally defined a set of architecture design *flaws*[4], shown to be highly correlated with error-proneness and change-proneness. We applied their *flaw detector* to the eight *ABB* projects, and created an extension of the tool to calculate the maintenance costs of each flaw, which we call the *flaw cost calculator*. We reported the results to practitioners to see if this analysis could help them pinpoint the file groups responsible for maintenance problems, to visualize architecture flaws, and to make refactoring decisions. In this section, we introduce the concept of architecture flaws, and then report the flaws detected in *ABB* projects.

### 5.1 Six Types of Architecture Design Flaws

Mo et al. [15] first defined five types of architecture design flaws that were repeatedly observed from many software systems, including: 1) *Unstable Interface*, where an influential file changes frequently with its dependents as recorded in the revision history; 2) *Modularity Violation*, where structurally decoupled modules frequently change together; 3) *Unhealthy Inheritance*, where a base class depends on its subclasses or a client class depends on both the base class and one or more of its subclasses; 4) *Cyclic Dependency* or *Clique*, where a group of files form a strongly connected graph; and 5) *Package Cycle*, where two packages depend on each other (rather than forming a hierarchical structure, as it should). Their tool was recently extended to detect a 6th type of flaw: 6) *Crossing*, where a file with both high fan-in and high fan-out changed frequently with its dependents and the files it depends on.

A system may have multiple instances of any flaw, and each can be visualized as a DSM using existing tools, such as Titan [25]. As exemplified in Figure 3, a DSM is a square matrix in which columns and rows are labeled using the same set of files in the same order. The annotations in each cell indicate the structural and evolutionary relations between the file on the row and the file on the column. For example, the cell in row 3, column 1, cell(r3,c1) contains "*d*, 14", which means that *path1.File3_cpp* depends on *path1.File1_cpp*, and they have changed together 14 times as recorded in the revision history. Cells with numbers only indicate that there are no structural dependencies between the files, but they were changed together. Cells with letters only indicate that the file on the row syntactically depends on the file on the column, and they were not changed together. The cells along the diagonal indicate self-dependency. Figure 3 depicts two DSMs, each representing a flaw instance from a *ABB* project. Figure 3a presents the DSM of an instance of Clique (the actual files names are anonymized), which shows that there are 16 files in the Clique; changes to any one of them could influence the other 15.

[4]Which are also called as "*hotspots*" or "*issues*"

### 5.2 Architecture Flaws Detected in Practice

The *flaw detector* component, as depicted in Figure 1, automatically detected the flaws within each project, and output a DSM file for each flaw instance, which became the input of the *flaw cost calculator* component that quantified the maintenance costs of each flaw, so that the users could better prioritize and rank them. In this component, four measures extracted from revision history were used to quantify maintenance costs: 1) change frequency (CF)—the number of commits a file is involved in; 2) change churn (CC)—the number of lines of code (LOC) committed to make changes; 3) bug frequency (BF)—the number of bug fixes a file is involved in; 4) bug churn (BC)—the number of LOC committed for bug fixes. Not all projects have all the data needed. If the commits of a project do not link to issues, or issues were not categorized into bugs, features, etc., this component will only calculate CF and CC. Some legacy systems do not keep records about the LOC changed in each commit. In these cases, this component only calculates BF or CF. The output generated by these two components has three parts:

(1) Flaw summary. As an example, Table 3 summarizes the flaws detected in *Proj_EP*, their scopes and maintenance costs. The first line shows that there are 322 files (21% of all the files) involved in 26 Clique instances. These files were changed 1,790 times involving 26,294 LOC, 41% of all the LOC changed in the revision history. 643 of the changes are for bug fixing, involving 16,557 LOC, which is 45% of all the LOC spent for bug fixing. This table shows that Cliques are very expensive to maintain in this project.

(2) Flaw costs. As an example, Table 4 shows that Clique1 involves 99 files and incurred the most maintenance costs, definitely worth attention. Clique5, although it contains just 16 files, also appears to be very costly. Using this table, the development team can prioritize which flaws need to be addressed in which order. By comparing with system average bug and change rates, we can see that files involved in these flaws are causing high maintenance difficulty.

(3) Flaw DSM. For each instance of each flaw, the tool generated a DSM which we exported as a spreadsheet so that the development team could visualize the relevant structure. Figures 3a-3b present the DSMs of several typical flaws that we reported to the architects, together with their maintenance effort.

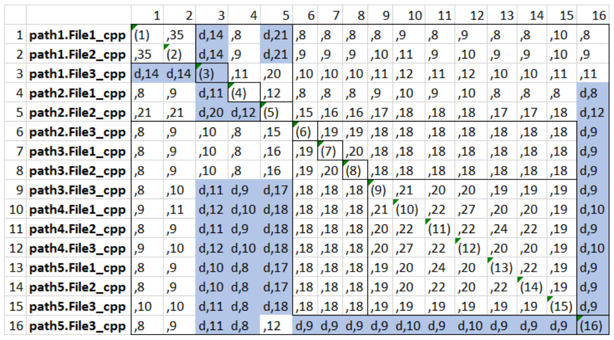
- Figure 3a shows a Clique where files are highly coupled by cyclic dependencies. For example, *path1.File1_cpp*, *path1.File2_cpp*, and *path1.File3_cpp* form two dependency cycles, and these files changed together frequently.
- Figure 3b shows a Crossing, centered on *path2.File2_h*, which depends on four files and influences eight other files. This file changes frequently with all these 12 files, and ranked the 8th most error-prone (e.g., changed for bug fixes 11 times) and change-prone among all 6,984 files in Proj_CH.

**Table 3: Architecture Flaws in *Proj_EP***
***Pt. : Percentage; Flaw CF - BC : maintenance costs, quantified by CF, CC, BF and BC, of the files in each flaw***

| | #Instances | #Files | Pt. | Flaw CF | Pt. | Flaw CC | Pt. | Flaw BF | Pt. | Flaw BC | Pt. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Clique | 26 | 322 | 21% | 1,790 | 28% | 26,294 | 41% | 643 | 34% | 16,557 | 45% |
| Crossing | 91 | 368 | 24% | 3,146 | 50% | 40,247 | 63% | 1,051 | 55% | 25,177 | 68% |
| ModularityViolation | 667 | 588 | 38% | 4,538 | 72% | 46,224 | 72% | 1,438 | 75% | 27,648 | 74% |
| PackageCycle | 175 | 499 | 32% | 2,417 | 38% | 29,906 | 47% | 778 | 41% | 18,889 | 51% |
| UnstableInterface | 6 | 316 | 21% | 1,669 | 26% | 19,898 | 31% | 388 | 20% | 11,457 | 31% |
| UnhealthyInheritance | 72 | 257 | 17% | 1,528 | 24% | 22,007 | 34% | 480 | 25% | 13,481 | 36% |

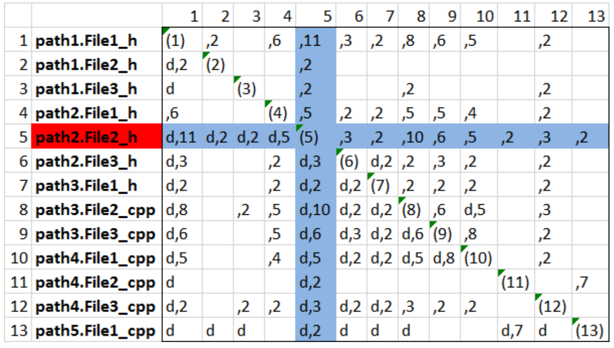
**Table 4: Maintenance Costs of Clique instances**
***Tot. CF - BC: the total CF - BC of all files in each clique instance***

| Instance Name | Size | Tot. CF | Tot. CC | Tot. BF | Tot. BC |
|---|---|---|---|---|---|
| Clique1 | 99 | 226 | 7,847 | 112 | 4,673 |
| Clique2 | 78 | 181 | 431 | 7 | 212 |
| Clique3 | 28 | 181 | 1,686 | 49 | 897 |
| Clique4 | 18 | 246 | 3,130 | 39 | 1,427 |
| Clique5 | 16 | 168 | 3,553 | 89 | 2,662 |



| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | path1.File1_cpp | (1) | ,35 | d,14 | ,8 | d,21 | ,8 | ,8 | ,8 | ,8 | ,9 | ,8 | ,9 | ,8 | ,8 | ,10 | ,8 |
| 2 | path1.File2_cpp | ,35 | (2) | d,14 | ,9 | d,21 | ,9 | ,9 | ,9 | ,10 | ,11 | ,9 | ,10 | ,9 | ,9 | ,10 | ,9 |
| 3 | path1.File3_cpp | d,14 | d,14 | (3) | ,11 | ,20 | ,10 | ,10 | ,10 | ,11 | ,12 | ,11 | ,12 | ,10 | ,10 | ,11 | ,11 |
| 4 | path2.File1_cpp | ,8 | ,9 | d,11 | (4) | ,12 | ,8 | ,8 | ,8 | ,9 | ,10 | ,9 | ,10 | ,8 | ,8 | ,8 | d,8 |
| 5 | path2.File2_cpp | ,21 | ,21 | d,20 | d,12 | (5) | ,15 | ,16 | ,16 | ,17 | ,18 | ,18 | ,18 | ,17 | ,17 | ,18 | d,12 |
| 6 | path2.File3_cpp | ,8 | ,9 | ,10 | ,8 | ,15 | (6) | ,19 | ,19 | ,18 | ,18 | ,18 | ,18 | ,18 | ,18 | ,18 | d,9 |
| 7 | path3.File1_cpp | ,8 | ,9 | ,10 | ,8 | ,16 | ,19 | (7) | ,20 | ,18 | ,18 | ,18 | ,18 | ,18 | ,18 | ,18 | d,9 |
| 8 | path3.File2_cpp | ,8 | ,9 | ,10 | ,8 | ,16 | ,19 | ,20 | (8) | ,18 | ,18 | ,18 | ,18 | ,18 | ,18 | ,18 | d,9 |
| 9 | path3.File3_cpp | ,8 | ,10 | d,11 | d,9 | d,17 | ,18 | ,18 | ,18 | (9) | ,21 | ,20 | ,20 | ,19 | ,19 | ,19 | d,9 |
| 10 | path4.File1_cpp | ,9 | ,11 | d,12 | d,10 | d,18 | ,18 | ,18 | ,18 | ,21 | (10) | ,22 | ,27 | ,20 | ,20 | ,19 | d,10 |
| 11 | path4.File2_cpp | ,8 | ,9 | d,11 | d,9 | d,18 | ,18 | ,18 | ,18 | ,20 | ,22 | (11) | ,22 | ,24 | ,22 | ,19 | d,9 |
| 12 | path4.File3_cpp | ,9 | ,10 | d,12 | d,10 | d,18 | ,18 | ,18 | ,18 | ,20 | ,27 | ,22 | (12) | ,20 | ,20 | ,19 | d,10 |
| 13 | path5.File1_cpp | ,8 | ,9 | d,10 | d,8 | d,17 | ,18 | ,18 | ,18 | ,19 | ,20 | ,24 | ,20 | (13) | ,22 | ,19 | d,9 |
| 14 | path5.File2_cpp | ,8 | ,9 | d,10 | d,8 | d,17 | ,18 | ,18 | ,18 | ,19 | ,20 | ,22 | ,20 | ,22 | (14) | ,19 | d,9 |
| 15 | path5.File3_cpp | ,10 | ,10 | d,11 | d,8 | d,18 | ,18 | ,18 | ,18 | ,19 | ,19 | ,19 | ,19 | ,19 | ,19 | (15) | d,9 |
| 16 | path5.File3_cpp | ,8 | ,9 | d,11 | d,8 | ,12 | d,9 | d,9 | d,9 | d,9 | d,10 | d,9 | d,10 | d,9 | d,9 | d,9 | (16) |

**(a) A Clique: highlighted cells form the dependency cycles.**
***Effort: Tot. CF: 456; Tot. CC:15,162; Tot. BF:116; Tot. BC:2,676.***



| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | path1.File1_h | (1) | ,2 | | ,6 | ,11 | ,3 | ,2 | ,8 | ,6 | ,5 | | ,2 | |
| 2 | path1.File2_h | d,2 | (2) | | | ,2 | | | | | | | | |
| 3 | path1.File3_h | d | | (3) | | ,2 | | | ,2 | | | | ,2 | |
| 4 | path2.File1_h | ,6 | | | (4) | ,5 | ,2 | ,2 | ,5 | ,5 | ,4 | | ,2 | |
| 5 | path2.File2_h | d,11 | d,2 | d,2 | d,5 | (5) | ,3 | ,2 | ,10 | ,6 | ,5 | ,2 | ,3 | ,2 |
| 6 | path2.File3_h | d,3 | | | ,2 | d,3 | (6) | d,2 | ,2 | ,3 | ,2 | | ,2 | |
| 7 | path3.File1_h | d,2 | | | ,2 | d,2 | d,2 | (7) | ,2 | ,2 | ,2 | | ,2 | |
| 8 | path3.File2_cpp | d,8 | | ,2 | ,5 | d,10 | d,2 | d,2 | (8) | ,6 | d,5 | | ,3 | |
| 9 | path3.File3_cpp | d,6 | | | ,5 | d,6 | d,3 | d,2 | d,6 | (9) | ,8 | | ,2 | |
| 10 | path4.File1_cpp | d,5 | | | ,4 | d,5 | d,2 | d,2 | d,5 | d,8 | (10) | | ,2 | |
| 11 | path4.File2_cpp | d | | | | d,2 | | | | | | (11) | | ,7 |
| 12 | path4.File3_cpp | d,2 | | ,2 | ,2 | d,3 | d,2 | d,2 | ,3 | ,2 | ,2 | | (12) | |
| 13 | path5.File1_cpp | d | d | d | | d,2 | d | d | d | | | d,7 | d | (13) |

**(b) A Crossing: the cell in red is the Crossing file; Blue cells show dependencies and their co-changes.**
***Effort: Tot. CF: 183; Tot. CC:13,179; Tot. BF:66; Tot. BC:2,936.***

**Figure 3: Example DSMs of Architecture Flaws**
**d: depend; number: co-changes**

After submitting the report to the development teams, we also conducted a telephone conference to review the results in case the developers were not familiar with DSMs. During the presentation and interaction, the development teams all commented that these architecture flaws revealed key problems that they had suspected but had no way to specify or quantify before.

## 6 ARCHITECTURE ROOT ANALYSIS

Unlike the flaw detectors that identify many file groups, the *root detector*, as shown in Figure 1, generates 10 (or fewer) file groups that typically capture the majority of a system’s most error-prone files. Our objective is to assess whether root analysis can help development teams pinpoint architectural problems more effectively.

### 6.1 Architecture Roots

Xiao et al. [24] proposed that software architecture can be modeled as multiple overlapping *design rule spaces* (DRSpaces), each containing an architecturally connected group of files. They also define the top few DRSpaces that capture most error-prone files as *architecture roots* (or *roots* for short). They have shown that five roots can typically cover 50% to 90% of all the error-prone files in a system, an observation validated over dozens of industrial and open source software systems. The implication is that most error-prone files are architecturally connected, and that the more error-prone the files are, the more likely that they are architecturally connected and that errors propagate through the connections.

A root can also be modeled using a DSM. Figure 4 presents a root detected from one of the projects. This DSM reveals multiple architecture design flaws. For example: 1) $p1.F1$, an unstable interface, is depended upon by most of the files, and most of these dependents changed together with it frequently; 2) Multiple dependency cycles are identified, such as, $p1.F5 \leftrightarrow p2.F2$, and $p2.F2 \rightarrow p2.F1 \rightarrow p1.F6 \rightarrow p1.F5 \rightarrow p2.F2$; 3) $p1.F1$ depends on its child, which is Unhealthy Inheritance; 4) Many modularity violations are highlighted in red: structurally independent modules that have changed together frequently. We also showed the change rate and ranking of each file in the first two columns of the DSM. For example, the file “$p4.F3$" in row 26 was changed 361 times, and ranked the most change-prone among all 2,403 changed files in Proj_SS. In this root, 84% of the files ranked within top 10 percentile most change-prone, and 6 out of the 31 files ranked within top 1 percentile, which indicates that this root is a real maintenance hotspot.

### 6.2 Root Analysis in Practice

We use *Proj_EP* as an example to illustrate the roots detected in these eight projects. In *Proj_EP*, the *root detector* detected 4 roots, and generated the following data:

(1) The scope and cost of each root, as shown in Table 5. For example, the first row shows that the first root involves 147 files. These files were changed 1,109 times, consuming 13,487 LOC. Of these changes, 414 were bug fixes involving 9,347 LOC. As we can see from the table, even though a Root only covers a small portion

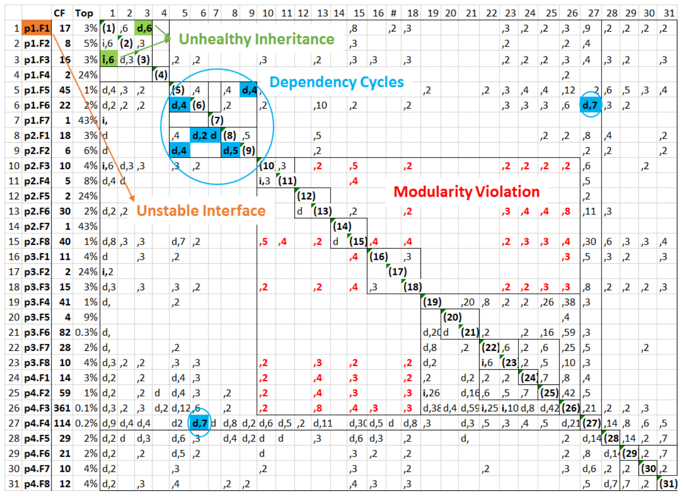


**Figure 4: DRH-Clustered Architecture Root**
***d: depend; i: inherit; CF: Change Frequency; Top: percentile rank***

of the system, it is a hotspot where much maintenance effort was spent.

**Table 5: Data of each detected Architecture Root**
***%: percentage; Rt. CF - BC: the total CF - BC of all files in each root***

| | Size (%) | Rt. CF (%) | Rt. CC (%) | Rt. BF (%) | Rt. BC (%) |
|---|---|---|---|---|---|
| root1 | 147 (10%) | 1,109 (18%) | 13,487 (21%) | 414 (22%) | 9,347 (25%) |
| root2 | 93 (6%) | 1,050 (17%) | 11,486 (18%) | 452 (24%) | 6,696 (18%) |
| root3 | 79 (5%) | 601 (10%) | 5,453 (9%) | 183 (10%) | 3,821 (10%) |
| root4 | 104 (7%) | 486 (8%) | 10,794 (17%) | 166 (9%) | 6,236 (17%) |

(2) The cumulative data for *all* Roots. A file may participate in more than one architecture root; that is, roots overlap with each other. DV8 also calculates their cumulative data, as shown in Table 6. In this table, "*Size*" means the number of distinct files in the first $n$ roots, where, $n = 1, 2, \ldots, 4$. The "*%Size*" column presents the percentage of the root size compared with the total number of files in the project. For example, "222" in the second row means that *root*1 and *root*2 (the first 2 Roots) contain 222 distinct files, which covers 14% of all files in the project. The "*Coverage*" column presents the cumulative coverage of change-prone or bug-prone files by these roots. The fourth row of this table indicates all these 4 roots contain only 24% of all the files in this project, but cover 55% of all change-prone files and 65% of all bug-prone files. Files in each root are architecturally connected, hence it appears that change-proneness or bug-proneness may be propagated among these files.

Moreover, following the experience reported in [12], and considering each architecture root as a *debt*, we created the *debt calculator* to compute the penalty incurred by these roots as the difference between the actual maintenance effort spent and the *expected* maintenance effort spent on them. We use the average change/bug rate of all the files in each project as the *expected* maintenance effort, following [12]. The *expected* effort columns "*ExtraCF*"- "*ExtraBC*" represent the cumulative maintenance penalty from the roots. For example, "615" in the second row of "*ExtraBF*" column indicates that the 222 files in *root*1 and *root*2 are involved in bug fixes 615 times more often than average files. The "*Percentage*" row presents the percentage of the extra maintenance effort to whole project. The last row indicates that, 28% of all the changes, 41% of all the LOC spent, 40% of bug-fixing changes, and 47% of bug-fixing LOC spent on the entire project are incurred (or penalties) by these roots.

**Table 6: Cumulative Data of Architecture Roots (*Proj_EP*)**

| | | | Coverage | |
|---|---|---|---|---|
| Root | Size | % Size | Change | Bug |
| root1 | 147 | 10% | 24% | 29% |
| root2 | 222 | 14% | 38% | 52% |
| root3 | 263 | 17% | 47% | 57% |
| root4 | 364 | 24% | 55% | 65% |
| | Penalty of Architecture Roots | | | |
| | Extra CF | Extra CC | Extra BF | Extra BC |
| root1 | 612 | 8,450 | 263 | 6,418 |
| root2 | 1,332 | 16,601 | 615 | 11,175 |
| root3 | 1,687 | 19,570 | 724 | 13,552 |
| root4 | 1,754 | 26,110 | 763 | 17,314 |
| Percentage | 28% | 41% | 40% | 47% |

We have observed consistent results from all eight projects, as summarized in Table 7. Column "*All Roots Tot. Size%*" shows that, for all the projects, their detected architecture roots contain 2% - 24% of all files in each project, but cover a much larger portion of the project's change-prone (37% - 68%) or bug-prone files (47% - 81%) as shown in column "*Tot. Coverage*". Due to the lack of traceability of the bugs, we did not conduct the bug-related analysis on the last three projects. The "*Penalty*" columns show that, for all projects, a large portion of maintenance effort spent on a project is generated from the detected architecture roots.

As we will elaborate later, the feedback regarding the detected roots was divided. Some teams found that roots can capture hotspots more effectively than the flaw analysis since they only need to examine a few file groups, but other teams found that the detected roots can be distorted.

**Table 7: Architecture Root Analysis results of all projects**
***Tot. Ext CF - BC %: showing the ratio of penalty from all roots in each project to the total maintenance effort spent on the project***

| Project | #Root | All Roots | Tot. Coverage | |
|---|---|---|---|---|
| | | Tot. Size % | Change | Bug |
| Proj_CH | 4 | 8% | 53% | 47% |
| Proj_CO | 1 | 18% | 59% | 56% |
| Proj_EP | 4 | 24% | 55% | 65% |
| Proj_OP | 8 | 14% | 51% | 47% |
| Proj_EC | 7 | 13% | 46% | 81% |
| Proj_SS | 5 | 2% | 63% | - |
| Proj_BM | 1 | 11% | 37% | - |
| Proj_EO | 2 | 19% | 68% | - |
| | Penalty Percentage (%) of all Architecture Roots | | | |
| | Tot. Ext CF % | Tot. Ext CC % | Tot. Ext BF % | Tot. Ext. BC % |
| Proj_CH | 53% | 83% | 64% | 89% |
| Proj_CO | 39% | 56% | 46% | 62% |
| Proj_EP | 28% | 41% | 40% | 47% |
| Proj_OP | 28% | 43% | 35% | 45% |
| Proj_EC | 9% | 21% | 11% | 3% |
| Proj_SS | 30% | 59% | - | - |
| Proj_BM | 35% | 56% | - | - |
| Proj_EO | 47% | 68% | - | - |

## 7 INTERVIEWS AND FEEDBACK

To better understand how the information presented by the analysis framework was understood and used, we formulated a set of questions for the pilot participants to answer after they had reviewed the results. Participants were provided with the questions in advance of a telephone conference conducted to record their answers. The questions were designed to follow the key deliverables of the report, the metrics, architecture flaws, and architecture roots. We posed the questions to five participants who represented one or more of the 8 projects we analyzed. Three of the participants were software architects and two were R&D managers. Our objective was to explore how the development teams intended to react to and use the report to improve their architecture quality. Next are the questions and the summary of their responses:

*Q1: What did the report reveal that you didn't know about your software?* Prior to having this report, the participants had intuitive understanding of their architectures. One of them commented: "*We understand intuitively how our code is structured.*" The DL and PC scores were surprising for the architects of Proj_OP and Proj_SS, who thought the report indicated their code was better than their opinion of it. For Proj_OP, their maintenance effort was higher than they thought the metrics indicated it should be. For Proj_SS, some programming languages used in the project were not fully supported by Understand, therefore the scores were better than they should be due to missing elements.

*Q2: Are the metrics useful for reflecting the architecture of your software?* All the participants commented that the report provided them quantifiable results and actionable items to improve their architecture, as well as a way to discuss the importance of refactoring their architecture with managers. For Proj_CH, the trend of metrics between releases was useful to understand whether the architecture was improving or degrading. For Proj_EC and Proj_SS, the participant thought the scores were useful and would like to run daily analyses to compare the variations of the scores over time. The need for updated reports was also expressed by projects BM, CO, and EP. Proj_EO had a very high DL and a low PC score, but they still had more rework effort than expected as they attempted to evolve their product, which indicates that metrics derived from syntactical relations only may not be sufficient.

*Q3: What did the architecture design flaws reveal about your software?* Proj_EO was surprised by their architecture design flaws and found a few false-positives. All other projects reported that the detected flaws confirmed what they intuitively knew about the structure of their software, and made their intuitive knowledge visualizable and quantifiable.

*Q4: What actions have you planned as a result of the architecture design flaws report?* Six of the projects said they planned to perform refactoring to address the detected architecture design flaws. Proj_BM and Proj_EO do not plan to refactor, and indeed their DL and PC scores were high. The architect of Proj_CH commented that the architecture flaw report provided them quantitative data to help prioritize where to refactor the code. Proj_EC and Proj_SS, which are already actively refactoring, desired more specific guidance towards how they should refactor.

*Q5: What did the architecture roots reveal about your software?* Proj_OP reported that roots in a particular component were unexpected, and probably indicated a higher amount of development activity in normally stable code. The participant commented: "*We may have underestimated the risk of changing this part of the code.*" For Proj_CH and Proj_OP, the architecture roots pointed to utility files that should not be considered as roots of the structure. These modules were used by many other components thus the analysis considered them as roots. Proj_EC and Proj_SS found the files reported as roots were expected because they contain definitions that change frequently.

*Q6: What actions do you plan to take to address architecture roots?* Proj_OP had a plan to componentize the architecture, which is expected to address the roots and improve the metrics scores. They commented: "*We plan to monitor our progress in architecture decoupling using these metrics over time.*" Proj_EC and Proj_SS reported being unsure about how to proceed with improving their architecture roots and felt they lacked an accurate mental model of what an architecture root is. Proj_CH expressed a similar sentiment because utility files were identified as roots and they did not fully understand the purpose of root identification. By contrast, Proj_EP confirmed that the detected roots are composed of defect-prone files and they were planning to refactor them to improve quality.

## 8 ANSWERS TO RESEARCH QUESTIONS

We summarize the feedback provided by the development teams to answer each research question below.

*RQ1: does DV8 help to close the gap between management and development? That is, does it help them to decide if, when, and where to refactor?* Three of the participants in charge of 5 projects (Proj_BM, Proj_CO, Proj_EP, Proj_CH, and Proj_OP) verified that the information provided was useful in closing the understanding gap with management. Even though the other two participants didn't explicitly comment on this aspect, the fact that six of eight projects planned or had already begun refactoring their code to address the flaws and roots suggests that our report played a role in reaching these refactoring decisions.

*RQ2: does DV8 help practitioners understand the maintainability of their systems relative to other projects internal to the company, and relative to a more broad-based benchmark suite?* All the participants said the report gave them quantifiable results with which to judge their code base. Two were surprised on how good their products were rated, that is, close to the 50th percentile within the industrial benchmark, given their intuitive understanding of the maintenance effort involved for their products. The comparison with industrial benchmark makes it clear that maintenance difficulty caused by degrading architecture is very common.

*RQ3: does DV8 help developers pinpoint the* hotspots *of their systems—that is, the groups of files with severe design flaws?* Based on the feedback from all 5 participants, the answer to this question is clearly *yes*. Six of the eight projects planned to or already started refactoring to address the detected flaws. The project with the lowest DL score is undergoing a major rewrite. One practitioner expressed the need for more detailed guidance on how to refactor the detected flaws.

In summary, we can answer all three questions positively.

## 9 LESSONS LEARNED

In this section, we discuss the lessons learned in terms: the effectiveness of these techniques, the data quality, and the limitations that lead to future work.

**Using DL and PC.** The practitioners adopted DL and PC easily, and expressed the need to compare multiple projects and analyze multiple releases of the same project using just a few metrics so that they can monitor the quality of the architecture. We summarized the following lessons regarding how DL and PC should be used in a complementary way:

(1) If a system has a low DL and high PC score, it means that maintenance difficulty is inevitable, and this conclusion is consistent with the experience of practitioners: so far we have seen no exceptions.

(2) If the DL and PC scores are both highly ranked, it means the system is likely not experiencing severe problems. If the development team is experiencing maintenance difficulty, this suggests that the system has a large number of implicit dependencies. In this case, architecture flaw detector should be executed to pinpoint the problematic file groups.

(3) If the system has a highly ranked DL, but a much lower ranked PC, such as Proj_CH where DL is ranked the 81st percentile, but PC is only ranked the 54th percentile, it means that there could be a small portion of the system that is highly coupled, which is confirmed by the development team. This result implies that an overall DL score only may not be able to reflect the existence of a high-maintenance subsystem.

(4) If both scores are ranked medium, e.g. the DL of Proj_CO ranked the 43rd and its PC ranked the 52nd, then the project is likely experiencing maintenance difficulties already, as confirmed by the development teams.

We have not observed a case where the system has a highly ranked PC score, but its DL ranking is very low. Since PC is very sensitive to the size of the project, as reported in [16], we suggest that these two scores should be used together. Another lesson is that a good DL and PC score does not necessarily mean that a system is healthy. If the development team is experiencing difficulty despite good scores, it is worthwhile to use flaw detectors to further pinpoint where the difficulty comes from.

**Using flaws and roots.** In addition to quantifying flaws, visualization of each flaw augments the intuition of developers, bridging the gap between development and management. Currently we export flaw DSMs into spreadsheets, and mark the files involved in each flaw manually. In the future, we will further automate this process.

The experiences of root analysis are divided. Some teams like the fact that using roots, they only need to inspect a few file groups, and observe how most error-prone files are connected. Other teams found that some roots are caused by high-impact utility files. Because these files have a large number of dependents, they can form a large DRSpace that captures a large number of files, including both error-prone and healthy ones, thus distorting the result. The lesson here is that the high-impact utility files should be excluded from this analysis.

We also learned that the pilot participants required comparison reports which showed the evolution of a product between releases. These reports were useful because the team could understand the trend of whether their architecture was degrading or improving. The report contained comparisons of the key metrics plus comparisons of the architecture flaws and which classes were involved in architecture roots. We found some roots persisted between releases, which means that they were a consistent source of development effort across releases.

One comment: "*Since the guidance of the report is towards refactoring, it makes sense that a release to release comparison would be a useful way to integrate it into the software development life-cycle... I would love to have this integrated into the daily build, with automatic production of guidance to architects to help them with day to day architectural governance.*"

**Data quality.** Another lesson was to consider multiple branches when collecting history data. What the participants provided as a known good source code branch frequently was a branch with very few actual developer commits (they were mostly merges) thus little history of each file could be collected. By considering multiple branches for historical changes, we expanded the set of commits in the analysis and had a more complete view of the history of each file.

Of all the three types of analysis provided by DV8, the DL and PC are structure metrics, calculated based on syntactic relations only. Three architecture flaws require structural relations only: Clique, Improper Inheritance, and Package Cycles. The other three flaws, as well as root analysis rely on revision history data. If the issue tracking data is available, issues can be categorized, e.g., into bug-fixing, feature addition, etc., and each commit is linked with an issue, then we can calculate both change related costs (change frequency and change churn) and bug-related costs (bug churn and bug frequency). If the issue tracking data is not available, or not linked with commits, then we can only calculate change-related costs.

Consistent with our observations with other industrial projects, most of the comany's projects do not have the data needed for all analyses. Some projects have a long history, back to the time when modern version control systems were not available. Other projects used their own issue tracking systems that do not support issue categorization. There are also projects in which the commits were not linked to issues.

After talking to the development teams, we realized that in some projects, the developers were not required to link commits with issues since they didn't envision the possible usage of the data. Now that they have seen the benefits of these analyses, a more rigorous management process was being discussed.

**Limitations and Future Work.** The process of analyzing the 8 projects also revealed several limitations of DV8 and the underlying technologies. First of all, the definition of certain architecture flaws should be further refined. For example, the tool may detect a large number of modularity violations (MV) that overlap with each other. Sometimes the number of files involved in MV is so large that the instances were just ignored all together. We are exploring the possible ways to further refine the definition of MV and its detection algorithm to obtain more focused results. Second, based on the feedback we received about root analysis, we plan to further refine the root detection methods so that utility files can be excluded. Third, DV8 is limited to C, C++, C# and Java, the mainstream programming languages that can be accurately processed by $Understand$. Processing multi-language systems is a future direction of the tool suite. Finally, we will further increase the number of projects in the benchmark database so that the users can compare projects with similar sizes and domains. Even though we have not observed that DL is affected by these characteristics, PC can be significantly affected by the number of files in the project.

Findings with respect to the research questions have a threat to external validity in that they are based on the opinions of only 5 practitioners and the set of practitioners is focused on architects and managers not developers.

During this process, we have observed the need to further automate the process to enable the analysis of more complex code bases; for example, a system may contain multiple components. Each component should be analyzed individually, and the system should be analyzed as a whole. We have worked to automate the analysis by adding configuration management to the interim data and applying scripts to automate the generation of report data. The practitioners also suggested that they would like to know whether a product's DL and PC values have gotten worse over night. If so, they would like to know how to locate the specific problem area. They envision a visual diff of the DSM. Then they would need automated guidance on what to change (which file and which refactoring) to repair the situation before problems accumulate. So far, our analysis is at the file level, one of the practitioners suggested that we could dig more details at the method level. If so, they would like to know which methods or attributes are responsible to the specific flaws or roots.

## 10 RELATED WORK

In this section, we compared with the related work in the areas of software metrics, defect prediction and technical debt.

*Software Metrics.* Numerous research has been conducted to measure software systems. McCabe [14] measures code complexity by calculating the number of linearly independent paths in the source code. Various metrics were proposed to measure OO projects, such as C&K metrics[8] and MOOD Metrics [11]. Yu at. el's [26–28] proposed multiple coupling metrics and reported that they were correlated to history changes, reuse effort and software performance respectively. Bouwers et. al. [4] showed that the measuring results from two architecture metrics [2, 3] matched practitioner's intuitions and could help in the decision-making process. There is no substantial evidence showing that these metrics can be used to effectively compare and contrast different projects or multiple versions of the same project, and thus form an effective benchmark to bridge the gap between management and development teams. Sahraoui et. al. [20] presented a measurement program, MQL. Their results showed that using MQL could significantly impact the quality of software systems, such as, maintainability, evolvability, code complexity, etc. But they didn't have a benchmark to follow and didn't clearly present how to guide the development teams for further refactorings.

*Defect prediction.* Defect prediction has also been widely studied. Code metrics, history measures or both were used for defect prediction. Nagappan et al. [17] presented a combination of code metrics used for defect prediction. However, they also reported that the best combination of metrics varies in different projects. Selby and Basili [22] presented that dependency structure is a good indicator of software defects. Cataldo et al.'s [7] showed that the density of change coupling is strongly correlated with failure proneness. Ostrand et al. [19] demonstrated that a combination of file metrics and file change history can be used to effectively predict defects. All the above studies treat files individually in the analysis, not taking architectural connections among files into consideration. By contrast, the root and flaw detection we applied can reveal architecture problems that propagate errors among multiple files. Schwanke et. at. [21] reported their experience of using structure dependency and history measures to predict defects and detect architecture issues, but their experience was based on one industrial case and focused on the detection of molecularity violation. By contrast, we report our experiences of applying software measurement, flaw and root detection comprehensively.

*Technical Debt Analysis.* In the past decade, a number of heuristics have been proposed to analyze technical debt [9] of software systems. Kazman et. al. [12] presented their experience of using economic models to assess the costs and benefits of refactoring software architecture debts, in which they only reported the experience from one system, without the application of DL/PC benchmark, flaw detection, or the automated calculation of maintenance costs of each flaw. Carriere et. al. [6] used a cost-benefit model to estimate the effort and benefits of applying refactoring to decouple the architecture. Their study only considered the coupling level of architecture in one case, and it did not provide information about when and where to refactor. Curtis et. al. [10] presented a model to estimate technical debt principal in terms of cost which is determined by static analysis of source code. Nord et. al. [18] developed a formula to assess the impact of technical debt in architecture, and presented that their approach could be used to optimize the long-term evolution of a product. These studies only reported information for assessing technical debt, but didn't report information about where to refactor.

## 11 CONCLUSIONS

In this paper, we reported our experiences of applying three architecture analysis techniques, supported by an automated tool suite with 8 components, to 8 projects in *ABB*. Our experiences demonstrated that: 1) DL and PC could effectively reflect the maintainability of a software project by comparing with a published industrial benchmark; 2) architecture flaw analysis enables practitioners to pinpoint and visualize severe design flaws, as well as to quantify their maintenance costs, so that developers can target refactoring actions towards the most severe architecture flaws; and 3) architecture root analysis could reveal how bug-prone and change-prone files are connected more effectively. These techniques and our tools have been adopted within *ABB*, we are now working on integrating the three techniques into a deployable service that can be used by all projects in the company. We will also create a command line version of the tools, so that the key analyses, such as DL and PC calculations, can be more easily integrated into existing software quality control tools, such as SonarQube[5]. Our objective is to measure projects with each build so that any quality degradation can be detected immediately.

## ACKNOWLEDGMENTS

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

[5]https://www.sonarqube.org/

## REFERENCES

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

[2] E. Bouwers, J. P. Correia, A. van Deursen, and J. Visser. Quantifying the analyzability of software architectures. In *Proc. 12thWorking IEEE/IFIP International Conference on Software Architecture*, pages 83–92, 2011.

[3] E. Bouwers, A. van Deursen, and J. Visser. Dependency profiles for software architecture evaluations. In *Proc. 27thIEEE International Conference on Software Maintenance*, pages 540–543, 2011.

[4] E. Bouwers, A. van Deursen, and J. Visser. Evaluating usefulness of software metrics: An industrial experience report. In *Proc. 35thInternational Conference on Software Engineering*, pages 921–930, 2013.

[5] 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.

[6] J. Carriere, R. Kazman, and I. Ozkaya. A cost-benefit framework for making architectural decisions in a business context. In *Proc. 32ndInternational Conference on Software Engineering*, pages 149–157, 2010.

[7] 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.

[8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[9] W. Cunningham. The WyCash portfolio management system. In *Addendum to Proc. 7th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 29–30, Oct. 1992.

[10] B. Curtis, J. Sappidi, and A. Szynkarski. Estimating the principal of an application's technical debt. *IEEE Software*, 29(6):34–42, 2012.

[11] F. B. e Abreu. The mood metrics set. In *Proc. ECOOP'95 Workshop on Metrics*, 1995.

[12] 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.

[13] 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.

[14] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

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

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

[17] 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.

[18] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas. In search of a metric for managing architectural technical debt. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pages 91–100, 2012.

[19] 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.

[20] H. Sahraoui, L. C. Briand, Y.-G. Gueheneuc, and O. Beaurepaire. Investigating the impact of a measurement program on software quality. *Information and Software Technology*, 52(9):923–933, 2010.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] L. Yu, K. Chen, and R. Ramaswamy. Multiple-parameter coupling metrics for layered component-based software. *Software Quality Journal*, 17(1):5–24, 2009.

[27] L. Yu and R. Ramaswamy. Component dependency in object-oriented software. *J. Comput. Sci. Technol.*, 22(3):379–386, May 2007.

[28] L. Yu and R. Ramaswamy. Examining the relationships between software coupling and software performance: a cross-platform experiment. *Journal of Computing and Information Technology*, 2011.