Decoupling Level: A New Metric for Architectural Maintenance Complexity

Ran Mo, Yuanfang Cai
Drexel University
Philadelphia, PA, USA
{rm859,yc349}@drexel.edu

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

Lu Xiao, Qiong Feng
Drexel University
Philadelphia, PA, USA
{lx52,qf28}@drexel.edu

ABSTRACT

Despite decades of research on software metrics, we still cannot reliably measure if one design is more maintainable than another. Software managers and architects need to understand whether their software architecture is "good enough", whether it is decaying over time and, if so, by how much. In this paper, we contribute a new architecture maintainability metric—*Decoupling Level* (DL)—derived from Baldwin and Clark's option theory. Instead of measuring how coupled an architecture is, we measure how well the software can be *decoupled* into small and independently replaceable modules. We measured the DL for 108 open source projects and 21 industrial projects, each of which has multiple releases. Our main result shows that the larger the DL, the better the architecture. By "better" we mean: the more likely bugs and changes can be localized and separated, and the more likely that developers can make changes independently. The DL metric also opens the possibility of quantifying canonical principles of single responsibility and separation of concerns, aiding cross-project comparison and architecture decay monitoring.

CCS Concepts

•Software and its engineering → Software architectures;

Keywords

Software Architecture, Software Quality, Software Metrics

1. INTRODUCTION

Despite decades of research on software metrics, we still cannot reliably measure if one design is more maintainable than another. Software managers and architects need to understand whether their software architecture is "good enough", whether it is decaying and, if so, by how much. Most well-known metrics focus on measuring the complexity and quality of *code*, such as McCabe's Cyclomatic Complexity [23] and Chidamber & Kemerer's metrics [8]. These metrics have been used for defect prediction and localization, but not for comparing design alternatives, or indicating architecture decay. MacCormack et al. proposed Propagation Cost (PC) [22] to measure how tightly source files are coupled. PC has been used by multiple companies to monitor coupling variation in a system. But, as we will show, PC is sensitive to the number of source files, and does not always capture architecture-level variations.

In this paper, we contribute a new architecture maintainability metric: *Decoupling Level* (DL). Instead of measuring the level of *coupling*, we measure how well the software is *decoupled* into small and independently replaceable modules. This metric is derived from Baldwin and Clark's *design rule theory* [1], which suggests that modularity creates value in the form of *options*: each module creates an opportunity to be replaced with a better version, hence improving the value of the system. Accordingly, small, independently changeable modules are most valuable, and the more such modules there are, the higher the system's value.

The implication of option theory in software design is significant: an independent module in software implies that its bugs can be fixed locally, and changing it will not cause any ripple effects. The smaller the module, the easier it is to improve, and the more such small, independent modules there are, the more developers can contribute to the system independently and in parallel. In our prior work, we created a *Design Rule Hierarchy* (DRH) clustering algorithm [5, 36, 6] to identify independent modules. We now calculate the *Decoupling Level* of a system from its DRH.

We measured the DL for 108 open source and 21 industrial projects, each having multiple releases. We observed that 60% of them have DLs between 46% and 75%. To evaluate if DL can be used as a real *metric*, we use it both to compare multiple snapshots of the same project, and to compare DLs collected across multiple projects. The results showed that the DL values extracted from consecutive, non-refactoring snapshots are very stable, and non-trivial variation of DL indicates major architectural degradation or improvement. To evaluate if projects with higher DL have higher maintainability, we contribute a suite of maintainability measures that can be extracted from the revision history of a project. These measures indicate the extent to which files were changed separately, and to what extent committers have to change the same set of files. Our results show that the larger the DL, the more likely bugs and changes can be localized and separated, and the more likely that developers can make changes independently.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE '16, May 14-22, 2016, Austin, TX, USA

© 2016 ACM. ISBN 978-1-4503-3900-1/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2884781.2884825

With intellectual roots in design rule theory, DL quantitatively measures canonical principles of single responsibility and separation of concerns. Although the ability to support parallelized development and localized changes is just one of many aspects of software maintainability, it is a critical one that needs to be measured and monitored continuously.

2. BACKGROUND

In this section, we introduce the fundamental concepts behind our new metric, including *Design Rule Theory*, *Design Structure Matrix*, and *Design Rule Hierarchy* (DRH).

Design rule theory. Baldwin and Clark proposed *design rule theory* [1] that explains how modularity adds value to a system in the form of options. Their theory suggests that independent *Modules* are decoupled from a system by the creation of *Design Rules* (DRs). The independent *Modules* should only depend on DRs. As long as the DRs remain stable, a module can be improved, or even replaced, without affecting other parts of the system.

Since Sullivan et al. [33] introduced design rule theory to software design, we have observed that design rules are usually manifested as interfaces or abstract classes. For example, in an Observer Pattern [11], the observer interface decouples the subject and concrete observers into independent modules. As long as the interface is stable, the subject shouldn't be affected by the addition, removal, or changes to concrete observers. In this case, the observer interface is considered to be a *design rule*, decoupling the subject and concrete observers into two independent *modules*.

Design Rule Hierarchy (DRH). To detect design rules and independent modules within a software system, our prior work defined a clustering algorithm, *Design Rule Hierarchy* (DRH) [5, 36, 6], which clusters a system's files into a hierarchical structure with n layers, where layer 1 contains the most influential files, typically interfaces or abstract classes that have many dependents. Files in layer i should only depend on files in higher layers, i.e., layer j, where $j < i$, but not depend on files in lower layers.

The unique feature of a DRH clustering is that files in the same layer are decoupled into *modules* that are mutually independent from each other. Independence here means that changing or replacing one module will not affect other modules in the same layer. The modules in layer n, that is, the bottom layer of the DRH are truly *Independent Modules* because each can be improved or replaced without influencing any other parts of the system.

Design Structure Matrix (DSM). *Design rules* and *modules*, as well as the structure of the *design rule hierarchy*, can be visualized using a *Design Structure Matrix* (DSM). A DSM is a square matrix; its rows and columns are labeled with the same set of files in the same order. Take the DSM in Figure 1a as an example. The columns and rows are labeled with the names of Java classes reverse-engineered from the source code of a student project submission. A marked cell in row x, column y, $cell(x, y)$ means that the file in row x *depends* on the file in column y.

The marks in the cell are refined to indicate different *types* of dependencies. For example, in Figure 1a, $cell(9, 3)$ is labeled with "*Ex,Cl*", which is short for "*Extend, Call*". This cell indicates that `Question` is an abstract class, and `Match`—a question matching class—extends it and calls one of its methods. The cells along the diagonal represent self-dependency. In this paper, we use "x" to denote an unspecified dependency type in a DSM.

The DSM in Figure 1a is an automatically generated DRH structure with 4 layers. Layer 1 has one class, `UI` (File 1), which is the interface that decouples the module with two user interface classes, `TextFileUI` (File 6) and `CommandLineUI` (File 7), from their clients, that is, various question and answer classes (File 9 to File 14). Layer 2 has one module with two files, the abstract `Question` (File 3) and `Answer` (File 2) classes. The third layer contains one module with one file, `Survey` (File 4), that aggregates a collection of objects of type `Question`. These four files in the topmost three layers completely decoupled the rest of the system into 6 independent modules in the last layer, Layer 4 (from File 5 to File 16). Consider the module containing `Choice` (File 12) and `ChoiceAnswer` (File 11). The designer could choose better data structures for these multiple-choice question classes without influencing any other classes.

Independence Level. Based on design rule theory, the more independent modules there are in a system, the higher its option value. In our prior work [31], we proposed a metric called *Independence Level* (IL) to measure the portion of a system that can be decoupled into independent modules within the last layer of its DRH. For example, the IL in the DSM of Figure 1a is 0.75 because 12 out of the 16 files are in the last layer. The *Decoupling Level* metric we propose here improves on the IL metric.

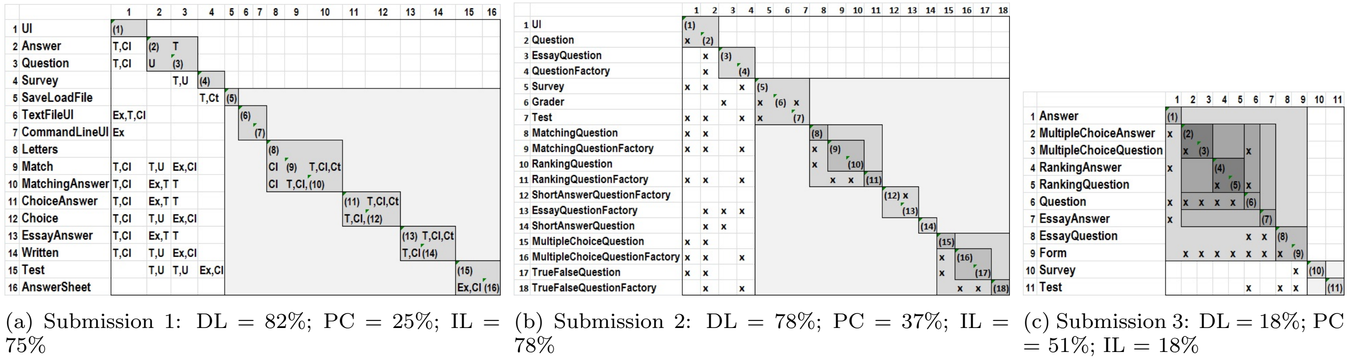
Propagation Cost. MacCormack et al.'s *Propagation Cost* metric—also calculated based on a DSM—aims to measure how tightly coupled a system is. Given a DSM, they first calculate its transitive closure to add indirect dependencies to the DSM until no more can be added. Given the final DSM with all direct and indirect dependencies, PC is calculated as the number of non-empty cells divided by the total number of cells. For example, the PCs of the three DSMs in Figure 1 are 25%, 37%, and 51% respectively. The lower the PC, the less coupled the system.

The problem with IL is that it doesn't consider the modules in the top layers of a DRH, nor does it consider the size of a module. Working with our industrial partners, we observed cases where the lowest layer contained very large modules. In these cases, even through the IL appeared to be high, the system was *not* well modularized. In other cases, we observed that even though the number of files decoupled in the last layer were not large, the modules in upper layers had few dependents. In this case, a system may not experience maintenance problems, despite its low IL.

The problem with PC is that it is sensitive to the size of the DSM: the greater the number of files, the smaller the PC. For example, from the 46 open source projects with more than 1000 files, 70% of them have PCs lower than 20%. For the other 62 projects with less than 1000 files, however, about 48% of them have PCs lowers than 20%. More importantly, as we will see later, sometimes an architecture can change drastically without significantly changing its PC.

3. DECOUPLING LEVEL

In this section, we introduce the rationale and formal definition of *Decoupling Level* (DL) using several examples. Baldwin and Clark's theory suggests that a module with high option value should be small (easy to change), with high *technical potential* (active), and with few dependents. The DRH structure allows us to assess a software architecture in terms of its potential to generate option values because it



(a) Submission 1: DL = 82%; PC = 25%; IL = 75%

(b) Submission 2: DL = 78%; PC = 37%; IL = 78%

(c) Submission 3: DL = 18%; PC = 51%; IL = 18%

Figure 1: Design Rule Hierarchy Samples. T:*Typed*; Cl:*Call*; Ex:*Extend*; Ct:*Cast*, U:*Use*, x: any dependency

explicitly identifies modules, their sizes, and how decoupled are they from each other. We thus propose *Decoupling Level* (DL) to measure *how well* an architecture is decoupled into modules. Concretely, since DRH separates modules into layers, we calculate the DL of each layer. Because the modules in the last layer do not have any dependents, we treat them differently. Note that we cannot quantify technical potential, as DL only measures source code. Next we introduce the formal definitions of DL based on the above rationale.

3.1 Formal Definitions

We define $\#AllFiles$ as the total number of files in the system, and $\#Files(M_j)$ as the number of files within a DRH module, M_j. Given a DRH with n layers, its DL is equal to the sum of the DLs of all the layers:

$$DL = \sum_{Li=1}^{n} DL_{L_i} \qquad (1)$$

Since the last layer of a DRH is special (in that it contains truly independent modules that can be replaced or changed without influencing any other parts of the system) we calculate the DL for the last layer differently. For an upper layer $L_i, (i < n)$ with k modules, we calculate its DL as follows:

$$DL_{L_i} = \sum_{j=1}^{k} [\frac{\#Files(M_j)}{\#AllFiles} \times (1 - \frac{\#Deps(M_j)}{\#LowerLayerFiles})] \qquad (2)$$

where, $\#Deps(M_j)$ is the number of files within lower layer modules that directly or indirectly depend on M_j. If a module influences all other files directly or indirectly in lower layers, its DL is 0; the more files it influences in lower layers, the lower its DL; the larger a module, the more likely it will influence more files in the lower layer, and hence the lower its DL. Based on the definition of DRH, a module in upper layers must influence some files in lower layers.

We calculate the DL of the last layer, L_n, based on the following rationale: the more modules in last layer, and the smaller each module, the better the system can support module-wise evolution. Our earlier work of *Independence level* (IL) [31] only considers the proportion of files within the last layer: the better modularized a system is, the larger the proportion of files in the last layer. However, we have seen some very large modules in the last layer of some projects, which can skew the IL metric.

Ideally, we want modules to be small. But how small? From an analysis of 41 projects, we calculated the average number of files in last layer modules and in all DRH modules. The results are 2.11 and 3.27 files respectively, meaning that the average DRH module has just a few files. Prior work on cognitive complexity [12] also shows that people can comfortably process approximately 5 "chunks" of information at a time. Accordingly, we consider a DRH module with 5 files or fewer to be a *small* module.

If the last layer, L_n, has k modules, then its DL is:

$$DL_{L_i=n} = \sum_{j=1}^{k} SizeFactor(M_j) \qquad (3)$$

If a module, M_j, has 5 files or fewer, we calculate its *SizeFactor* based on its relative size:

$$SizeFactor(M_j) = \frac{\#Files(M_j)}{\#AllFiles} \qquad (4)$$

If M_j has more than 5 files, we add a penalty to reflect the limits of human cognitive capabilities:

$$SizeFactor(M_j) = \frac{\#Files(M_j)}{\#AllFiles} \times (log_5(\#Files(M_j)))^{-1} \qquad (5)$$

Figure 2 illustrates how the size of modules influences DL. Suppose a system only has 1 layer with 100 files. If they form 25 modules, each having 4 files, its DL is 100% (Figure 2a), meaning that each module can improve its value by being replaced with a better version and thus increase the value of the whole system easily. As the size of each module grows, it becomes harder for them to change. If each module has 25 files (Figure 2b), then its DL decreases to 50%, and then to 41% if each module has 50 files (Figure 2c). If the 100 files only form one module, then it only has 35% DL (Figure 2d). If the system only has 1 layer containing 1 module with multiple files, then its DL decreases with the number of files.

If all the modules in the last layer have fewer than 5 files, then the DL of the last layer equals the proportion of last-layer files to the total number of files, which is equivalent to the *Independence Level* metric proposed in our prior work [31]. DL is, however, different from IL: first because DL considers modules in *all* the layers, and second because it takes the size of a module into consideration.

3.2 An Example

Now we use the example shown in Figure 1 to illustrate how DL can manifest design quality. The three DSMs were reverse-engineered from student submissions for the same class project used in a software design class offered at Drexel

University. The students were given 10 weeks to create a questionnaire management system, so that a user can create a questionnaire that can be graded, and a respondent can complete a given questionnaire.

The basic types of questions supported include multiple choice, true/false, matching, ranking, short answer, and essay. The software was to be designed for easy extension, such as adding new types of questions, adding a GUI in addition to a console UI, and supporting different display and printing formats. The students were supposed to achieve this objective by properly designing question and answer classes, and applying appropriate design patterns.

The three DSMs revealed drastically different designs for the same project. All three students designed an abstract `Question` class, to abstract the commonality among question classes. Since a true/false question can be seen as a special type of multiple-choice question, ranking is a type of matching, and short answer is a type of essay, both Submission 1 and Submission 3 have 3 types of question and corresponding answer classes. But Submission 1 has the highest DL, 82%, and Submission 3 has the lowest DL, 18%.

There are several major differences among these designs. First, both `Survey.java` in Submission 1 and 2, as well as `Form.java` in Submission 3 aggregate a collection of questions. In the first two designs, however, both `Survey` classes only interact with the `Question` base class. That is, as a design rule, `Question` decouples `Survey` from concrete question and answer classes, and each type of question and its answer classes form a module that can be changed independently. In Submission 3, however, `Form` depends on every question and answer class, leaving only two classes that can change independently: `Survey` and `Test`.

In addition, Submission 1 applied a bridge pattern correctly, so that the user can choose to use a `TextFileUI` or `CommandLineUI` at runtime, creating more independent modules. Submission 2 also attempted to apply a bridge pattern, but didn't do it correctly. In addition, although Submission 2 has 6 types of question and answer classes, they are not independent of each other. For example, the `Ranking` class extends the `Matching` class. Consequently, the DL of Submission 2 is lower than that of Submission 1.

3.3 Tool Support

We have created a program to calculate *Decoupling Level*, which we are integrating into Titan [38]. Our DL algorithm takes the following inputs: 1) A DSM file that contains the dependency relations among project source files. Currently Titan creates DSM files from XML files generated by a commercial reverse-engineering tool called Understand^{TM}. 2) A clustering file that contains the DRH clustering information for the source files. We generate this using the DRH clustering function of Titan. Given these inputs, our tool calculates the *Decoupling Level* of a software system.

4. EVALUATION

Now we evaluate DL based on how a *metric* should behave, using the concept of a centimeter—a commonly used metric—as an analogy. Concretely, we investigate the following questions:

RQ1: If Alice–a young girl–is measured two weeks in a row, her height measures should be the same, or very close to each other. Our analogous research question is: if a project is being revised for a limited period of time, say, through two releases, but doesn't go through major refactoring (for example, reorganizing the code base by applying some new architecture patterns), are the DL measures of these releases close to each other?

RQ2: If Alice measured 130 cm. and is subsequently measured one year later, we expect her height to be significantly more than 130 cm. By analogy, our research question is: if a project is successfully refactored and its modularity has truly improved, will its DL reflect the improvement? Or if the architecture of a project has degraded over time, does its DL reflect this degradation?

Positive answers to the above two questions imply that it is possible to quantitatively monitor architecture degradation or assess the quality of refactoring using the DL metric.

RQ3: If Alice measures 130 cm. tall, and Bella measures 140 cm., we can definitively state that Bella is taller than Alice. Our analogous question is: if project A has a higher DL than project B, is project A more maintainable than project B?

A positive answer to this question means that it is possible to quantitatively compare the maintainability of different projects or different designs for the same project. If 130 cm. is above the 50th percentile for a girl of Alice's age, her mom would know that her height is "above average". Similarly, if we measure the DL for a large number of projects, perhaps considering project characteristics, a manager could consult this dataset and determine the relative "health" of a specific project, from a maintainability perspective.

Since both propagation cost (PC) and independence level (IL) purport to measure software architecture's maintainability, we would like to know whether DL is more *reliable* than PC and IL. Therefore we will investigate the first three questions using DL, PC and IL comparatively.

Because we need to analyze multiple versions of the same project to answer RQ1 and RQ2, we call this *vertical evaluation* (Section 4.2). For RQ3 we need to compare different projects with various domains, sizes, and ages, and so we call this *horizontal Evaluation* (Section 4.3). Section 4.1 presents the measures we extracted from 129 projects, and Section 4.4 summarizes the results.

4.1 Subjects and their metrics

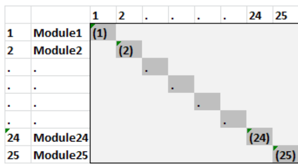
To minimize the possible bias caused by specific project characteristics, such as domain or size, we randomly chose 108 open source projects from Open Hub[1], and collected 21 industrial projects from 6 of our collaborators. Due to space limitations, we placed all the data on our website[2]. A brief inspection of this data shows that their domains, sizes, and implementation languages vary drastically.

To calculate the DL, PC, and IL values of these projects, we chose the latest version of each project, downloaded its source code, and then reverse-engineered this code using Understand^{TM}, which can output an XML file containing all the file-level dependency information for a project. Given these XML files, we used our tool Titan to calculate the DSM files and DRH clustering files. Finally, for each project, the DL, PC, and IL values are calculated, based on the project's DSM and DRH clustering files.

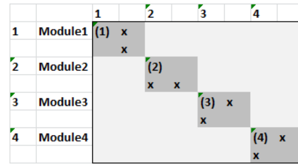
Table 1 reports the statistics of these metric values obtained from these subjects. This table shows that the average DL of all open source projects and industrial projects

[1]https://www.openhub.net/

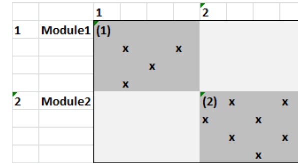
[2]https://www.cs.drexel.edu/~rm859/DL.html



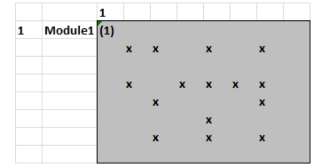
(a) 100 files are decoupled into 25 modules, each having 4 files: DL = 100%



(b) 100 files are decoupled into 4 modules, each having 25 files: DL = 50%



(c) 100 files are decoupled into 2 modules, each having 25 files: DL = 41%



(d) 100 files are decoupled into 1 module with 100 files: DL = 35%

Figure 2: One-layer DRH with different modular structure

are 60% and 54% respectively. Less than 20% of open source projects have a DL lower than 47% or higher than 75%, and these numbers are slightly lower for commercial projects. It also shows that the project with the best DL, 93%, is a commercial project, even though commercial projects have lower DL values in general. We will discuss the characteristics of PC and IL later.

It is obvious that the data in Table 1 will vary if we examine a wide variety of subjects; this is the expected behavior with all metrics. For example child growth charts differ for girls and boys, and vary with other factors, such as diet and region. We will continue collecting project data in the future and observe how these values vary and cluster.

Table 1: Metric Summary for 129 Projects
Pt: Percentile

Stats	Open Source (%)			Commercial (%)			All Projects (%)		
	DL	PC	IL	DL	PC	IL	DL	PC	IL
Avg	60	20	43	54	21	35	59	21	42
Median	58	18	41	56	20	28	57	18	40
Max	92	72	100	93	50	83	93	72	100
Min	14	2	12	15	2	9	14	2	9
20th Pt	47	8	28	36	6	24	46	8	26
40th Pt	55	14	37	46	17	26	54	15	37
60th Pt	66	21	45	59	24	38	63	22	45
80th Pt	75	34	55	65	35	46	75	34	54

4.2 Vertical Evaluation

We first evaluate if these metrics are stable for multiple consecutive, non-refactoring releases. We expect to observe little variation if a metric is reliable. After that, we apply these metrics to a commercial project that experienced serious architectural problems over a long period of evolution, which resulted in a significant refactoring to restore modularity. We are interested to know if the variation of these metrics can reflect the variations in the architecture.

4.2.1 The Stability of DL

To evaluate the stability of these metrics, we selected 16 out of the 129 projects, and a series of releases for each of them. Our selection was based on the following criteria: a) each project should have at least 4 sprints, meaning that it is revised during the chosen time period; b) these multiple snapshots are consecutive, e.g., multiple sprints from the same release, and c) a major refactoring among these snapshots was unlikely, either based on our prior study or on our communication with the architects.

We chose the 3 out of the 21 commercial projects (which we call *Comm_1*, *Comm_2*, and *Comm_3*) because we know that there was no refactoring during these selected snapshots based on our communication with their architects, and they all have sufficiently long revision histories. We chose 13 out of 108 open source projects because we had analyzed their structure before [37, 25], and have prior knowledge that these selected snapshots do not contain a major refactoring.

Table 2 reports the statistics of DL, PC and IL respectively. Take OpenJPA for example: even though the number of files increased from 2296 to 4406 during the 9 snapshots, its DL increased from 67% to 71%. The standard deviation and coefficient of variation (CV) of DL are only 1% and 2% respectively, meaning that even though it went through significant changes, its architecture, as reflected by DL, does not vary drastically. By contrast, the CVs of PC and IL are 26% and 7% respectively, meaning that these two metrics are more unstable. Over the 16 projects, we employ *Paired t-test* to test whether the CV of DL is significantly different with the CVs of PC and IL. The results are as follows: DL-PC, p-value=0.005; DL-IL, p-value=0.003. The results indicate that CV of DL is significantly different from the CVs of PC and IL, showing much higher stability.

Table 2: Coefficient of Variation(CV) of DL, PC and IL

	#Rls	Rls Range	#Fl Range	Avg DL	CV (%) DL	PC	IL
Avro	10	1.4.0-1.7.6	227-500	80%	4	18	5
Camel	9	2.2.0-2.11	4097-9011	83%	1	8	2
Cassandra	7	0.7.1-2.0.8	513-1105	35%	3	7	6
CXF	12	2.1.1-2.6.9	3108-5268	86%	1	5	1
Derby	13	10.4-10.10	2412-2796	64%	3	11	5
Hadoop	5	1.0.0-1.2.1	1990-2353	73%	1	2	2
Httpd	4	2.2.0-2.4.6	236-370	62%	1	8	2
Mahout	7	0.3-0.9	990-1376	92%	1	12	2
OpenJPA	9	1.2.0-2.2.2	2296-4406	69%	2	26	7
PDFBox	10	1.7.0-1.8.7	901-997	53%	1	2	1
Pig	6	0.6.0-0.9.1	997-1585	54%	2	16	6
Tika	8	1.0-1.7	392-550	81%	1	3	4
Wicket	10	1.3.0-1.5.6	1896-2612	71%	1	3	1
Comm_1	6	1.01-1.06	1455-1523	76%	3	5	5
Comm_2	8	1.0.1-1.0.9	287-494	73%	5	30	11
Comm_3	20	12.2-14.8	5239-6743	80%	3	52	13
Avg(CV)					2	12	5

4.2.2 The Variation of DL

To evaluate if non-trivial variation in a metric can faithfully indicate architecture variation, we need to understand what happens to an architecture when the metric value increases or decreases considerably. Ideally, we would want to collect the "inflection" points from multiple projects, and talk to their architects to verify what happened between these snapshots where the metric value changes noticeably. Given time and resource constraints, finding such candidates from open source projects was unrealistic. For commercial projects, 17 out of the 21 have 3 snapshots or fewer.

For the remaining 4 commercial projects, our analysis for

two of them were previously published [30, 18]. One of them was just refactored and we don't have the latest data yet, and the other was sold and we no longer have access to it. Comm_1 and Comm_3 are the remaining commercial projects we can explore. Comm_3 is one of the best modularized projects of all the projects we have analyzed, and their architect confirmed that no major architectural degradation or refactoring had occurred.

Project Comm_1, by contrast, exhibits a more typical evolution path. Table 3 displays the metric values for all 29 snapshots collected since 2009, and Figure 3 depicts the trends of the three metrics over these snapshots. The DL value indicates 4 major inflection points where its value increased or decreased more than 10 points: (1) from version 0.10 to 1.01, the DL increased from 45% to 74%, (2) from 1.06 to 2.01, DL decreased from 78% to 68%, (3) from 2.11 to 2.12, DL decreased from 65% to 48%, and (4) from 2.21 to 3.01, its DL increased from 48% to 62%.

By contrast, although both PC and IL reflect the first inflection point, the PC doesn't change in the other 3 points. That is, if we only consider PC as an architecture metric, it will indicate that the architecture didn't change at the 3 later points. IL missed the 3rd and 4th points, but indicates an architecture degradation (5) from 2.18 to 2.19 where it dropped 10 points, while PC also increased 2%.

We presented this data to the architects, and asked them what happened during these transitions. In other words, did the architecture actually improve when the DL increased, and did it degrade when the DL decreased? Did DL miss the 5th point, where IL indicates a degradation? Next we discuss these inflection points.

Transition 1: From 0.10 to 1.01, all three metrics indicate a significant improvement. According to the architect, when version 0.10 was released, it had been evolving for a year as a prototype. From 0.10 to 1.01 (released in April 2010), the product was refactored significantly and multiple design patterns were applied for the purpose of transforming it into a commercial product. The architecture indeed improved significantly. The transferring and refactoring process was accomplished by Sept 2010 when the commercial project was released as version 1.06.

Transition 2: From 1.06 to 2.01, since the architecture was stable, the management was eager to add new features, which was the main objective in the next 3 years. We were told that during these 3 years the developers were aware that, to meet deadlines, architecture debts were introduced and the project became harder and harder to maintain. We can see that the DL decreased from 77% to 68% and remained around 68% till 2.11 when the DL dropped to 65%. The PC values, by contrast, decreased only slightly.

Transition 3: From 2.11 to 2.12, we observed a significant drop of DL from 65% to 48%. Referring to the dataset of all 129 projects, the maintainability of this product decreased from the 80th percentile to around the 20th percentile. When we presented the data to the architect, unlike the previous two points when the DL changes were expected, this transition point was a small surprise: indeed there was a significant refactoring in 2.12 in which 5 new interfaces were introduced to decouple several highly coupled parts. This was the time when the developers were unable to tolerate the technical debt anymore, and decided to clean up while continuing to add new features.

After adding the 5 new interfaces, the architect expected a significant improvement in 2.12, but the DL showed the opposite. We then examined the DRSpaces [37] led by these 5 new design rules and tried to understand what happened. It turns out that 4 out of the 5 new design rules had very minor impact, only influencing a few other files. The other new design rule, however, was very influential, influencing 133 other files. Examining the DRSpace [37] formed by the 133 files, we observed that these files were not as decoupled as the architect expected. There existed several large dependency cycles that should have been decoupled. The architect confirmed that, since this was a significant refactoring combined with the addition of new features, the refactoring wasn't completely finished by 2.12. Instead, the cycles we observed in 2.12 were removed gradually from 2.12 to 2.21.

Thus the surprising decrease of DL was caused by a significant, but incomplete refactoring: the new design rule introduced many more dependents, but many modules were inadequately decoupled, hence the significant decrease of DL.

Transition 4: From 2.21 to 3.01, we observed a significant increase in DL but still not as high as the DL for 1.06 when the product was first refactored. The architect told us that the objective of 3.01 was to conform to a new third party library, and to improve unit testing. To do so, the major activities in 2.21 were "clean up", that is, reducing technical debt. Several big cycles we observed in 2.12 were completely decoupled in this process, which explains the increase of DL.

Transition 5: From 2.18 to 2.19, we observed that hundreds of files were added to the project, and both PC and IL indicate a degradation. The PC increased because of the extra dependencies to these new files, and IL decreased because the new files are all at the higher levels of the DRH. We examined the DRSpace formed by the new files, and found these files to be well decoupled. We also examined DRSpace led by each of new files. All these DRSpaces are small: the mean and median of their sizes are 5.3 and 4 files respectively, meaning that these newly added files are well-decoupled and have little influence on the other files. We presented the results to the architect and asked if the architecture degraded due to these newly added files. The architect explained that these new files were added because a new spreadsheet component was integrated to the system. This component interacts with the rest of the system through APIs, but it was designed to be architecturally isolated, and didn't change the structure of the rest of the system. The system didn't experience any degradation due to the integration either. Therefore, we believed that IL and PC reported false positives.

The architect confirmed that since the project has been accumulating debt for 4 years, not all architecture issues were completely resolved. And they are again facing the ubiquitous dilemma: *"Shall we refactor or keep adding new features?"*. The architect told us that the DSM analysis made it very clear where the debts are located and hence what should be done to remove them.

In summary, we observed that, in this project, the variation in DL measures not only indicated successful refactoring and architecture degradation, but also revealed an *unsuccessful* refactoring. Neither of the other metrics could provide such insights. Using DRSpace analysis, we were able to identify why the modules were not decoupled as expected.

4.3 Horizontal Evaluation

Our horizontal evaluation aims to investigate if software

Table 3: 29 Snapshots of Comm_1

Index	release	DL(%)	PC(%)	IL(%)	#Files
1	0.10	45	25	11	1063
2	1.01	74	12	22	1455
3	1.02	74	12	22	1465
4	1.03	74	12	22	1474
5	1.04	80	13	20	1411
6	1.05	78	11	22	1519
7	1.06	78	11	22	1523
8	2.01	68	8	31	1086
9	2.02	69	9	31	1097
10	2.03	69	9	31	1097
11	2.04	69	9	30	1096
12	2.05	69	9	30	1096
13	2.06	69	9	30	1099
14	2.07	69	9	30	1099
15	2.08	69	9	30	1099
16	2.09	69	9	30	1107
17	2.10	69	9	30	1109
18	2.11	65	8	33	1140
19	2.12	48	9	32	1197
20	2.13	48	9	33	1216
21	2.14	50	9	33	1196
22	2.15	50	9	33	1209
23	2.16	49	9	34	1233
24	2.17	48	8	34	1248
25	2.18	47	8	34	1273
26	2.19	48	10	25	1575
27	2.20	48	10	25	1589
28	2.21	48	10	26	1556
29	3.01	62	6	28	1852

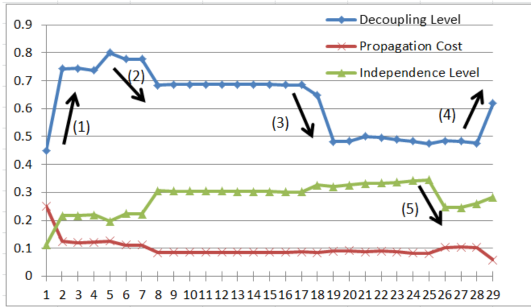


Figure 3: The DL variation for Comm_1

systems with higher DLs are easier to maintain. The root question is, how to measure *maintainability*? In this section, we first propose a suite of maintainability measures that can be extracted from the revision history of a software system. After that, we introduce the subjects we used in this evaluation, and the correlation between DL, PC, IL and these maintainability measures.

4.3.1 Maintainability Measures

Ideally, maintainability should be measured by the effort, in terms of hours, spent on each task. But such data is virtually impossible to obtain. We thus propose a suite of maintainability measures that can be extracted from software revision history. We illustrate the rationale using the 4 cases depicted in Figure 4.

Suppose there are three committers for a project, and each of them committed one revision, each changing a set of files. If each of the committers changed completely different sets of files (Figure 4a), it means that these files can be changed independently and in parallel, and it is unlikely that the committers need to expend effort communicating with each other. On the other extreme, if all three committers changed exactly the same set of files (Figure 4d), these files cannot be changed in parallel, and it is highly likely that the committers have to communicate to resolve conflicts. If these are bug-fixing changes, then the first case implies that the bugs were localized and separated, while the second case implies that all the bugs are in the same set of files. It is obvious that, in the first case, these files have the best maintainability, and in the second case the worst maintainability. We thus propose the following measures to quantify maintainability from the revision history of a project:

1. Commit Overlap Ratio (COR): measures to what extent changes made to files are independent from each other. That is, the total number of files revised in all commits, divided by the number of distinct files:

$$CommitOverlapRatio = \frac{\sum_1^m |FC_i|}{|FC_1 \bigcup FC_2 \bigcup ... \bigcup FC_m|} \quad (6)$$

Where m is the total number of commits, $|FC_i|$, $i = 1, 2, ..., m$, is the number of files changed in a specific $Commit_i$, and $|FC_1 \bigcup FC_2 \bigcup ... \bigcup FC_m|$ is the total number of distinct files involved in all commits. In Figure 4, the COR is 1 in Case (a)—the best case, and is 3 in Case (d)—the worst case. If these are bug-fixing commits, a larger COR means that more bugs are fixed by changing the same set of files, indicating that these files are harder to maintain. We further distinguish COR for bug-fixing only commits and all commits using BCOR and CCOR respectively.

2. Commit Fileset Overlap Ratio (CFOR): measures to what extend the filesets managed by different committers overlap. Suppose that a committer, C_i, makes multiple commits, and FS_i is the set of all the files C_i revised, then we calculate CFOR for all the committers as follows:

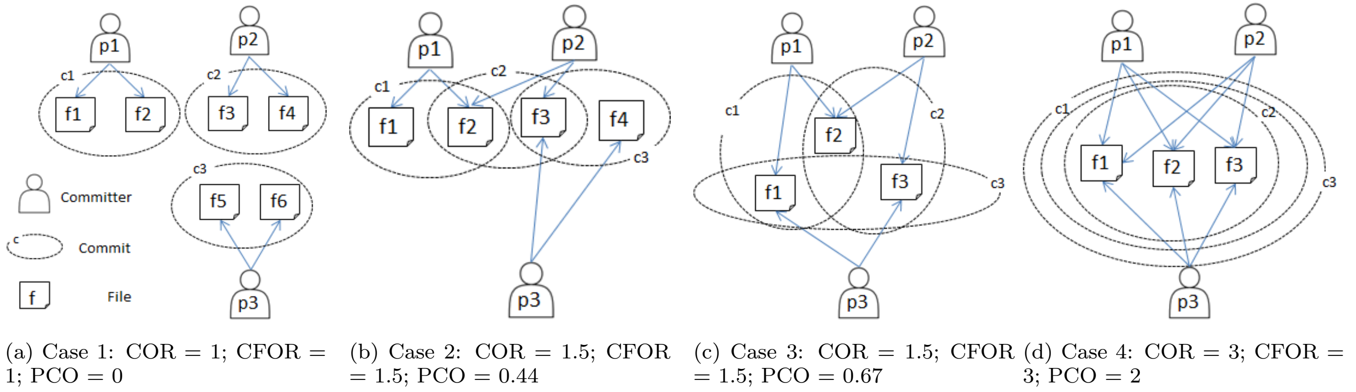
$$CFOR = \frac{\sum_1^m |FS_i|}{|FS_1 \bigcup FS_2 \bigcup ... \bigcup FS_m|} \quad (7)$$

Where m is the number of committers, and the denominator is the total number of distinct files committed by all committers. The larger the CFOR, the fewer files can be changed in parallel by different committers, indicating lower maintainability. We similarly distinguish CFOR for bug-fixing and all commits using BCFOR and CCFOR respectively.

3. Pairwise Committer Overlap (PCO): measures the likelihood that two committers have to communicate with each other to resolve conflicts. Suppose committer C_a changed fileset, FS_a, and committer C_b changed fileset, FS_b. We measure their communication need as the number of files they *both* changed, divided by the total number of files changed by *either* of them. For each committer C_i, we thus use Jaccard index [17] to calculate her potential interaction with all other committers as:

$$CommitterOverlap_i = \sum_j^m \frac{|FS_i \bigcap FS_j|}{|FS_i \bigcup FS_j|} \quad (8)$$

where $i \neq j$ and m is the number of committers. Then we define *Pairwise Committer Overlap (PCO)* as the average of *CommitterOverlap* among all committers. The higher the number, the more likely the committers have to communicate. In Figure 4, the PCO of Case (a) is 0, meaning that there is no need to communicate at all. Case (b) and Case (c) have the same COR and CFOR, but in Case (c), p3 may need to talk to both p1 and p2, hence Case (c) has higher PCO. In Case (d), each committer may have to talk



(a) Case 1: COR = 1; CFOR = 1; PCO = 0

(b) Case 2: COR = 1.5; CFOR = 1.5; PCO = 0.44

(c) Case 3: COR = 1.5; CFOR = 1.5; PCO = 0.67

(d) Case 4: COR = 3; CFOR = 3; PCO = 2

Figure 4: Layers with different modular structure

to 2 other committers (PCO = 2). We similarly distinguish PCO into BPCO (for bug-fixing commits) and CPCO (for all commits).

4.3.2 Evaluation Strategy

Ideally, to fairly evaluate whether DL can be used to indicate maintainability differences, we should first chose a set of projects of various sizes and domains, each having multiple maintainers, being well managed using proper version control and issue tracking systems, where most commits are explicitly linked with issues. More importantly, none of these projects should go through significant architecture or design level refactoring. Given these projects, we should measure DL values from their earlier versions that reflect a stable design, then monitor and collect their maintenance measures for a long enough period of time until we can get a statistically valid sample set. Ideally, these projects should have similar numbers of changes and bug fixes of similar levels of complexity, so that we can compare if higher DL values indeed correlate with lower maintenance effort.

In reality, finding such projects is not realistic. Even though there are large numbers of open source projects, it is unrealistic for us to interview each open source team to determine at which release the architecture was stable for measurement purposes. Moreover, given the frequent addition and removal of features, even though there is no significant refactoring, it is possible that the architecture may change during the project's evolution, intentionally or unintentionally, which will in turn influence its subsequent maintainability. Maintainability will also be affected by other factors, such as the popularity of a project, which will influence the number of bugs that are found and fixed. Consequently, our idealized evaluation is not feasible.

As a result, we decided to investigate as many projects as we could and include all the history of each to minimize the differences caused by the number of revisions. Since we don't know from which release the DL, PC, and IL should be measured to indicate a stable design, we measure multiple snapshots of each project and calculate average values. We are confident that average DL should faithfully reflect the architecture of a project, according to the stability analysis, and from the fact that large scale refactorings in open source projects are extremely rare [20]. We are less confident with using average PC since we have observed that PC varies drastically even for versions known to be stable, making it a less-qualified architecture measure. To make a fair comparison, however, we still calculate the average PC for each project, and calculate the correlation between average DL, PC, and DL on the one hand, and project maintainability measures on the other hand. Next we introduce the subjects we selected and the analysis results.

4.3.3 Horizontal Evaluation Subjects

We chose 38 out of the 108 open source projects and 3 out of the 21 commercial projects as the subjects for horizontal evaluation. As shown in Table 4, these projects are implemented using different languages, and have different sizes, ages, and domains. We chose those projects, rather than using all 129 projects mainly because their revision histories are well managed using well-known tools from which we can extract data. More importantly, in these projects, we are able to extract the linkage between commits and issues reliably. In other words, most committers in these projects labeled their commits with the ID of the issue that each commit addresses, so that we can distinguish bug-fixing commits from other changes.

For each snapshot of each project, we downloaded the source code, reverse-engineered the code, transformed the file dependencies into a DSM file using Titan, and generated its DRH clustering file. Using the DSM and DRH clustering files, we computed each DL, PC and IL, and calculated their averages over multiple snapshots. We used the revision history of each project to extract maintainability measures as introduced in Section 4.3.1.

Table 4: Horizontal Evaluation Subjects

#Projects:	41	Languages:	Java, C, C++, C#
#Files:	39-11130	CLOC:	10K-2.7M
#Committers:	18-915	#Commits:	346-74269

4.3.4 Analysis

Given the following measures of maintainability—Change Commit Overlap Ratio ($CCOR$), Bug Commit Overlap Ratio ($BCOR$), Bug Commit Fileset Overlap Ratio ($BCFOR$), Change Commit Fileset Overlap Ratio ($CCFOR$), Pairwise Committer Overlap, based on both all changes ($CPCO$) and bug-fixing commits ($BPCO$), we conducted a *Pearson Correlation Analysis* between these measures and DL, PC and IL respectively. We report the Pearson values (pv) and p-values in Table 5. All the p-values are less than 0.01, meaning these correlated relationships are statistically significant.

This table shows that these measures have the highest negative correlation with DL, meaning the higher the DL, the better the maintainability. IL similarly showed negative correlation but the correlation was much weaker than with DL. PC displays positive correlation with these maintenance measures, meaning that the more tightly coupled a system is, the harder it is to maintain. Although PC has relatively high correlations with CCOR, BCOR, CCFOR, and BCFOR, its correlations with CPCO and BPCO are much lower, meaning that this coupling measure is less correlated with how well people can make changes independently from each other.

Table 5: Pearson Correlation Analysis

	DL		PC		IL	
	pv	p-value	*pv*	p-value	*pv*	p-value
CCOR	-0.74	2.7E-8	0.68	8.7E-7	-0.48	0.0016
BCOR	-0.77	3.1E-9	0.65	4.1E-6	-0.48	0.0014
CCFOR	-0.67	2.1E-6	0.62	1.6E-5	-0.49	0.0015
BCFOR	-0.70	5.6E-7	0.59	5.2E-5	-0.52	0.0006
CPCO	-0.61	2.6E-5	0.54	0.0004	-0.42	0.0063
BPCO	-0.63	1.2E-5	0.53	0.0005	-0.45	0.0033

4.4 Evaluation Summary

So far we have answered all four questions positively. Compared with PC and IL, DL appears to be a more reliable metric in that it remains stable over subsequent releases, reveals architecture degradation and major refactorings, and has significant correlations with maintenance measures.

5. DISCUSSION

In this section, we discuss the threats to validity of the research, the interpretation of DL, and future work.

Threats to validity. Although our evaluation showed that DL is a promising metric, we understand that the evaluation suffers from several threats. First, even though we measured 129 projects, these projects use C, C++, C# and Java only. We thus cannot claim that DL can be applied to projects written in other languages, such as Perl or Javascript. Second, since all the DLs are calculated based on the static information extracted by Understand™, for projects in which major dependencies are not statically detectable, such as service-oriented architecture, we may not be able to calculate their DLs accurately. Third, the maintenance measures we proposed in Section 4 may not reflect the true *maintenance effort*. We will keep looking for better approximations and for actual effort data. Fourth, as we explained at the beginning of Section 4, to evaluate how well DL can predict maintainability, we should have used the DL collected at the beginning of a release, and then measured the subsequent maintenance effort until the DL changed significantly. We tried this strategy for a few projects, but the history data between releases is too small for us to form statistically meaningful results. Luckily we have several industrial collaborators who have refactored their code based on our previous analyses. Now that we have observed increased DLs, we will follow the newly refactored projects for at least a year and track how maintenance effort changes.

The Interpretation of DL. It is possible that variance in DL may be due to reasons other than the inherent architectural complexity. The domain and technology in used, for example, can both influence DL values. Mahout[3] has the highest DL, 92%, among all open source projects we have studied. Mahout is a machine-learning library including techniques for classification, recommendation, and clustering. The techniques and algorithms in Mahout's library are inherently separate from each other. So its domain leads naturally to a higher DL. However, the other two projects that have very high DLs—90% and 93%—are actually commercial projects with complex domains. One of them has a size similar to Mahout, while the other is 5 times larger, meaning that even a system with a complex domain can have a well-modularized design. In addition, we have shown in Section 3 that, even in student projects, different developers may create dramatically different designs.

A low DL may also have more than one explanation. One possibility is that the system is stable and seldom needs to change. Or, there could be very few developers maintaining it, so that they can work closely without the need to program in parallel. For example, for the three open source projects with the lowest DLs—GNU Screen[4] (DL = 14%), rsync[5] (DL = 18%), and ImageMagick[6] (DL = 18%)—Open Hub[7] reported that in the past 12 months, they all have just 3 developers, and their activities were either described as "*has not seen any change in activity*" or "*has seen a substantial decrease in development activity*". The fact is that these projects currently do not support large numbers of developers or parallel development. The reason could be either that their low DLs make changes extremely difficult, or the projects have been stabilized and there is no need to change. Our industrial experience shows that a low DL usually indicates significant maintenance difficulty caused by decayed architecture. In fact, several industrial projects with low DLs that we are engaged with are currently being refactored.

A high DL, however, doesn't always mean that the architecture is high quality. Of all the industrial projects we have analyzed, two of them have DLs higher than the 80th percentile. But they still experienced considerable maintenance difficulties. Our detailed analysis using Titan revealed that the major issues in these projects are caused by unstable interfaces and modularity violations [25]. In other words, implicit dependencies are the main causes of their maintenance problems [30, 18]. As a result, even though a project may have excellent scores as measured by static analysis tools, it may still suffer from maintenance difficulties. Thus an architecture should be analyzed using both structure and history information.

Also it should be noted that, although we intended to create a metric, and DL has demonstrated crucial properties of a metric, it is certainly not as precise and sensitive as, for example, a centimeter. Our experience has shown that, if the DL varies 10 points or more, it is almost certain that some architecturally significant changes have occurred. A smaller variation may, however, just be noise. As we have reported, a software project that has been refactored does *not* necessarily have a high DL. A complex refactoring may take a while to complete and the DL may actually decrease

[3]http://mahout.apache.org/
[4]http://www.gnu.org/software/screen/
[5]https://rsync.samba.org/
[6]http://www.imagemagick.org/
[7]https://www.openhub.net/

while the refactoring is in progress.

Future work. One of our ongoing tasks is to keep collecting more data from more open source and industrial projects, and form a broader architecture "Health chart". We plan to make DL calculation a *service*, so that anyone can measure their own project and add their data to our dataset.

We will also explore how to make the implicit (run-time) dependencies among services explicit, and explore the applicability of DL for these systems. It would also be interesting to investigate the three research questions using traditional code-based metrics. Also, thus far we have only investigated the *correlation* between DL and software maintainability. One of our future tasks, therefore, is to explore their causal relationships. Finally, doing a sensitivity analysis of the value at which modules are considered *small* is also part of our future work.

6. RELATED WORK

In the past decades, researchers have proposed various "metrics" to measure software quality. We now discuss three categories of work on software metrics.

Code Quality Measures. Numerous metrics are well-known measures to software quality, such as McCabe Cyclomatic complexity [23], Halstead metrics [14], Lines of Code. Various metrics were proposed to measure OO programs, such as CK metrics [8], LK Metrics [21], and MOOD Metrics [9]. These metrics have been used to predict quality issues, or to locate error-prone files [24, 15, 19, 35]. Combining code quality metrics to measure software maintainability has also been widely studied [27, 28, 16, 2]. Oman and Hagemeister [28] proposed Maintainability Index (MI), a composite number based on multiple metrics, to determine software maintainability. Bijlsma et al. [2] and Heitlager et al. [16] also used combined metrics to rate software maintainability. Similarly, commercial tools, such as SonarQube[8] and Infusion[9], also provide a single composite number—SonarQube's technical debt and Infusion's Quality Deficit Index (QDI), for management to monitor software quality.

The problem is, as Nagappan et al. reported [26], even though these complexity metrics have demonstrated to be useful for defect prediction, the most predictive metrics are different in different projects. Menzies [34] also pointed out the difficulty of using the measures collected from one set of projects to predict issues for another set of projects.

Different from these existing metrics, our DL is the only metric that measures how a software system is *decoupled*, quantifying canonical single responsibility and separation of concern principles. Our objective is not defect prediction, but architecture comparison and degradation monitoring. To the best of our knowledge, we are not aware of an existing metric, or a metric suite that has been successfully used to compare software architecture among large number of different projects, or demonstrates similar behavior as a real *metric*, such as a centimeter.

History Measures and Quality. Similar with structural metrics above, most measures extracted from history were proposed for bug location and error prediction. There have been numerous studies of the relationship between evolutionary coupling and error-proneness [10, 13, 7]. For example, Cataldo et al.'s [7] work reported a strong correlation between density of change coupling and failure proneness. Ostrand et al. [29]'s study demonstrated file size and file change information were very useful to predict defects.

Again, we are not aware of a history measure that has been used to compare software architecture like a real metric. Even though history can most faithfully reflect architecture quality, we prefer to access architecture/design quality before history and penalty accumulate. Using DL, the designer can judge if the maintainability is below average at early stages of software development.

Architecture Metrics. Although by far less well-known than the metrics mentioned above, MacCormack's *Propagation Cost (PC)* [22] has been used by most of our industrial collaborators. *Independence Level (IL)* [31], as a simplified, option-based metric, also has been accepted by one of our industrial collaborators to compare hundreds of projects, using IL values of open source projects as benchmarks. The most comprehensive formula is the *Option valuation* [1], proposed by Baldwin and Clark, which we first used to quantitatively compare the two variations of KWIC [32]. As we explained earlier in the paper, option valuation requires the estimation of the technical potential of each module, and the problems with PC and IL have been discussed.

Bouwers et al. [4, 3] reported that three of twelve system-level architecture metrics for encapsulation are correlated with the ratio of local change-sets [39]: higher ratio of local change-sets indicates better encapsulation. By contrast, our DL measures how well a software architecture can support parallelized, distributed development, and localized changes.

In summary, to the best of our knowledge, *Decoupling Level* is the only software maintainability metric that bears similarity with other metrics we use in everyday life, such as the centimeter, in that it allows managers to monitor, evaluate, and compare software projects and their evolution.

7. CONCLUSION

In this paper, we proposed a new metric, *Decoupling Level*, to measure software architecture maintainability. Based on Baldwin and Clark's design rule theory, this metric aims to measure to what extent an architecture is decoupled into small, independent modules that can be changed separately and in parallel by multiple developers.

Our evaluation has revealed that DL values remain stable through multiple, non-refactoring releases of projects, and its non-trivial variation indicates severe architecture decay or the occurrence of significant refactoring activities. We also extracted a suite of measures from a project's revision history to indicate maintainability, and analyzed how the DL, PC, and IL correlated with these measures using data from 41 projects. The results showed that DL has a much stronger negative correlation with project maintainability than the other architecture metrics. Our investigation suggests that DL has the potential to be a valuable metric for measuring, comparing, and monitoring software maintainability.

Acknowledgments

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

[8]http://www.sonarqube.org/

[9]http://www.intooitus.com/products/infusion

8. REFERENCES

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

[2] D. Bijlsma, M. A. Ferreira, B. Luijten, and J. Visser. Faster issue resolution with higher technical quality of software. *j-sqj*, 20(2):265–285, June 2012.

[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. Quantifying the encapsulation of implemented software architectures. In *IEEE International Conference on Software Maintenance and Evolution*, pages 211–220, 2014.

[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] 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.

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

[10] 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.

[11] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] F. Gobet and G. Clarkson. Chunks in expert memory: evidence for the magical number four ... or is it two? *Memory*, 12(6):732–47, Nov. 2004.

[13] 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.

[14] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., 1977.

[15] R. Harrison, S. J. Counsell, and R. V. Nithi. An investigation into the applicability and validity of object-oriented design metrics. *Empirical Software Engineering*, 3(3):255–273, Sept. 1998.

[16] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *Proc. 6th International Conference on Quality of Information and Communications Technology*, pages 30–39, 2007.

[17] P. Jaccard. The distribution of the flora in the alpine zone. *New Phytologist*, 11(2):37–50, Feb. 1912.

[18] 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.

[19] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, Nov. 1993.

[20] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao. Refactoring navigator: Interactive and guided refactoring with search-based recommendation. In *Submission*, 2015.

[21] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice Hall, 1994.

[22] 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.

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

[24] S. C. Misra. Modeling design/coding factors that drive maintainability of software systems. *Software Quality Control*, 13(3):297–320, Sept. 2005.

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

[26] 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.

[27] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *IEEE International Conference on Software Maintenance*, pages 337–344, 1992.

[28] P. Oman and J. Hagemeister. Construction and testing of polynomials predicting software maintainability. *j-ss*, 24(3):251–266, Mar. 1994.

[29] 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.

[30] 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.

[31] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna. From retrospect to prospect: Assessing modularity and stability from software architecture. In *Proc. 8thWorking IEEE/IFIP International Conference on Software Architecture*, Sept. 2009.

[32] K. J. Sullivan, Y. Cai, B. Hallen, and W. G. Griswold. The structure and value of modularity in design. *ACM SIGSOFT Software Engineering Notes*, 26(5):99–108, Sept. 2001.

[33] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *Proc. Joint 8th European Conference on Software Engineering and 9th ACM SIGSOFT International Symposium on the Foundations of*

Software Engineering, pages 99–108, Sept. 2001.

[34] M. Tim, B. Andrew, M. Andrian, Z. Thomas, and C. David. Local vs. global models for effort estimation and defect prediction. In *Proc. 26thIEEE/ACM International Conference on Automated Software Engineering*, pages 343–351, 2011.

[35] M. P. Ware, F. G. Wilkie, and M. Shapcott. The application of product measures in directing software maintenance activity. *Journal of Software Maintenance*, 19(2):133–154, Mar. 2007.

[36] 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.

[37] 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.

[38] 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.

[39] L. Yu, A. Mishra, and S. Ramaswamy. Component co-evolution and component dependency: speculations and verifications. *IET Software*, 4(4):252–267, Aug. 2010.