Assessing an Architecture's Ability to Support Feature Evolution

Ran Mo
Drexel University
Philadelphia, PA, USA
rm859@drexel.edu

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

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

Qiong Feng
Drexel University
Philadelphia, PA, USA
qf28@drexel.edu

ABSTRACT

Enabling rapid feature delivery is essential for product success and is therefore a goal of software architecture design. But how can we determine if and to what extent an architecture is "good enough" to support feature addition and evolution, or determine if a refactoring effort is successful in that features can be added more easily? In this paper, we contribute a concept called the *Feature Space*, and a formal definition of *Feature Dependency*, derived from a software project's revision history. We capture the dependency relations among the features of a system in a *feature dependency structure matrix* (FDSM), using features as first-class design elements. We also propose a *Feature Decoupling Level* (FDL) metric that can be used to measure the level of independence among features. Our investigation of 17 open source projects shows that files within each feature space are much more likely to be changed together, hence each feature space forms a meaningful maintainable unit that should be treated separately. The data also show that the history-based FDL is highly correlated a structure-based maintainability metric: *Decoupling Level* (DL). When we examine a project's evolution history, we see that if a system is well-modularized, it is more likely that features can be added independently. For shorter periods of time, however, FDL and DL may not be consistent, e.g., when the addition of new features deviates from the designed architecture or does not involve parts of the system that have architecture flaws. In such cases, FDL and FDSM can be used to monitor potential architecture degradation caused by improper feature addition.

CCS CONCEPTS

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

KEYWORDS

Software Architecture, Software Maintenance, Software Evolution

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.
ICPC '18, May 27–28, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196346

ACM Reference Format:
Ran Mo, Yuanfang Cai, Rick Kazman, and Qiong Feng. 2018. Assessing an Architecture's Ability to Support Feature Evolution. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196321.3196346

1 INTRODUCTION

Being able to add features quickly to a software project is essential for companies and is one of the most important objectives of software architectural design. Meyer proposed the now-famous open-closed principle [7]: *"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."* A highly extensible and maintainable software architecture should allow features to be added without extensive ripple effects. The problem is, how can we tell if and to what extent features are actually independent? Without a quantitative measure, it is hard to tell if an architecture is "good enough" to support feature delivery, if a refactoring effort is needed, or whether a refactoring was successful (so that adding features became easier).

There has been significant research on feature identification [1, 2] and localization [6, 15, 16, 22, 27] at the source-code level, but not on the dependencies among features as evidenced by a project's revision history, which reflects the actual difficulty of maintaining features. Feature delivery velocity is increasingly used as a productivity measure, but without quantitative measurement and deeper understanding of the interaction between architecture and features, it is difficult to manage and compare feature delivery velocity over time and across projects.

Given the lack of a widely accepted definition of *feature*, we propose a concept called a *Feature Space*, and define *Feature Dependency* based on evolution history. We visualize the dependency relations among a system's features using a *Feature Dependency Structure Matrix* (FDSM), in which features are the first-class elements. We also propose a *Feature Decoupling Level* (FDL) metric to measure the level of independence among features during a given period of time, as evidenced in revision history.

Our definition of *feature space* is based on revision history, and quite different from existing feature definitions [16, 22]. To evaluate if the feature spaces we defined truly model semantically cohesive, maintainable units, we investigated 17 well-known open source projects, and showed that files within a feature space are the most likely to change together, indicating that these files should be maintained together. For feature spaces with *feature dependency*, we

demonstrated that changes to files in one feature space are likely to cause changes to the dependent feature spaces.

Since FDL measures the architecture's ability of add new features, we further investigated the correlation between FDL and a newly proposed maintainability metric, *Decoupling Level* (DL) [30], that is based on syntactical relations among source files. The DL metric measures the level of modularity, that is, how well a system is decoupled into small and independent modules. It is widely accepted that a well-modularized system can better support the addition and modification of features. If FDL is a valid metric, it should quantitatively reflect this relation.

We demonstrate that, over a mature project's history, FDL scores are strongly correlated with DL: the higher the DL, the higher the FDL. For shorter periods of revision history, however, these two metrics may deviate from each other; even for well-modularized systems, certain kinds of features may be difficult to add. On the other hand, a system may have a low DL due to the existence of architecture flaws, but if the flawed part is not active, or if the addition of new features does not involve the flawed structure, it is still possible to have a high FDL, indicating that not all structural flaws need to be fixed. In these cases, we illustrate how one can use these models and metrics to monitor and explore the interaction between architecture and features, and to understand under what circumstances an architecture can better support which types of features.

2 FEATURE RELATED DEFINITIONS

In this section, we introduce formal definitions for the following concepts: *Feature Space*, *Feature Dependency*, *Feature Dependency Structure Matrix (FDSM)*, and *Feature Decoupling Level (FDL)*.

2.1 Feature Space

We represent each feature using a *feature space* ("*feature*" for short), defined as a 3-element tuple:

$$FeatureSpace = \langle TimeSpan, FileSet, DSM_r \rangle \quad (1)$$

$TimeSpan$ is a pair of timestamps recording the dates of the first and last commit of the feature: $TimeSpan = (startDate, endDate)$.

$FileSet$ is the set of all files added or modified for the feature in a period of history: $FileSet = \{f_1, f_2, ..., f_m\}$, where m is the total number of distinct files.

r is a release number, and DSM_r is the snapshot of the feature in release r, represented by a design structure matrix (DSM). In this paper, we use Design Structure Matrix (DSM) to model the files involved in a feature and the structural dependencies among these files. Please note that $FileSet$ contains all the files ever changed or added for the feature in a give time period, but some files maybe changed or deleted during software evolution. As a result, at a given release, the snapshot of the feature may only contain a subset of $FileSet$. Accordingly, the DSM of the same feature at different releases may be different.

A DSM is a square matrix, its rows and columns are labeled with the same set of design elements in the same order. Annotations in a cell reflect dependencies between the elements on the row and the element on the column. For example, in the DSM in Figure 1a, the elements are files, and the "x" in cell (9,8) (circled) indicates that *file* `SelectStatement` depends on *file* `Selection`.

Files in a DSM are clustered into a design rule hierarchy (DRH) [10, 11, 35], which identifies the *independent modules* and *design rules* [4]. A DRH has two important characteristics: 1) elements in a layer only depend on files in the upper layers; 2) elements within the same layer are grouped into *mutually independent* modules. The DSM in Figure 1a shows a DRH with 4 layers: L1: (rc1-rc5), L2: (rc6-rc7), L3: (rc8-16), L4: (rc17-26). We can see that files in L2 only depend on files in L1, and files in L3 depend on files in L2 and L1. Each layer is grouped into mutually *independent modules*. Taking L4 as an example, it is decoupled into 9 independent modules: (rc17), (rc18-19), (rc20), (rc21), etc.

We derived a FeatureSpace for each feature using the project's revision history and issue tracking systems. As an example, Figure 2 depicts a Git commit for the Apache Cassandra project,[1] showing that this commit: 1) was made on `2011-06-15`, and 2) was to implement issue `CASSANDRA-2617`, which is labeled as a new feature in the project's Jira issue tracking record. Six java files in the rectangle were added/changed for this commit, and the two numbers in front of each line indicate the number of LOC added and removed respectively. This was the only commit related to the feature recorded in the revision history. Using the above information, we can model the FeatureSpace of `CASSANDRA-2617` as follows:

$TimeSpan$: (`2011-06-15`, `2011-06-15`).

$FileSet$: {`CFMetaData`, `QueryProcessor`, `StatementType`, `CassandraServer`, `ThriftValidation`, `SchemaLoader`}.

DSM_r: the DSM of this feature space in the *latest release* of Cassandra can be found in Figure 1b.

If multiple commits were made for the same feature, then the $FileSet$ will contain the union of all involved files, and its $endDate$ will be the date of the last commit.

2.2 Feature Dependency

We defined the dependency relation, $FSDep$, among features as a binary relation on a set of FeatureSpaces, FS. If a feature, fs_y depends on fs_x, i.e. $(fs_y, fs_x) \in FSDep$, this means that some files involved in fs_y already exist in fs_x. In other words, when fs_y is added or modified, there may be a change to existing files in fs_x, which was created before fs_y.

For two features fs_a and fs_b, there are thus four possibilities:

If $fs_a.FileSet \bigcap fs_b.FileSet = \varnothing$, it means these two features are mutually independent;

If $fs_a.FileSet \bigcap fs_b.FileSet \neq \varnothing$, then:

1) if fs_a was commited before fs_b, then fs_b depends on fs_a;

2) if fs_b was commited before fs_a, then fs_a depends on fs_b;

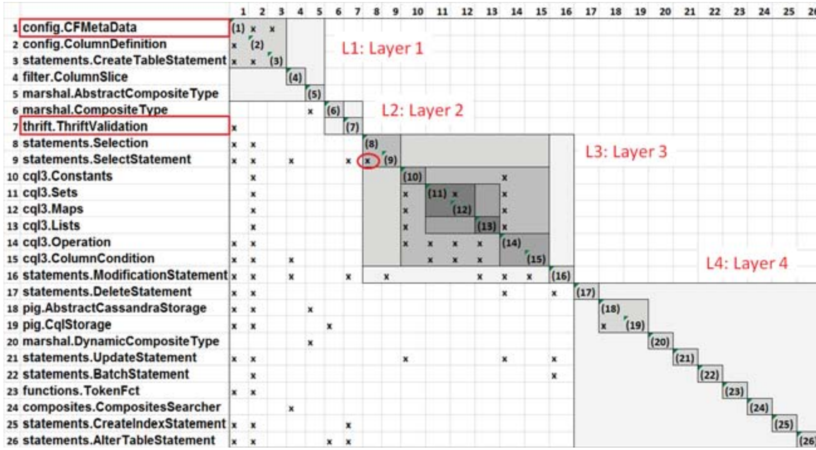
3) if fs_b and fs_a were committed together, then both (fs_a, fs_b) and (fs_b, fs_a) belong to $FSDep$.

Figure 1a depicts the feature space DSM of `CASSANDRA-6561`, a feature that was committed in 2014. Its feature space has two files (in red boxes) that were created or changed for `CASSANDRA-2617` in 2011. According to our definition, feature `CASSANDRA-6561` depends on feature `CASSANDRA-2617`.

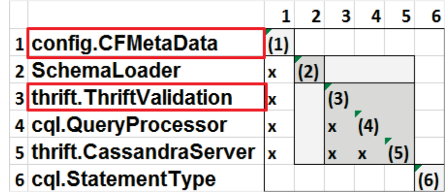
2.3 Feature Dependency Structure Matrix

Similar to the Design Structure Matrix (DSM) in Section 2.1, we define a *Feature Dependency Structure Matrix* (**FDSM**) as a tuple:

[1] http://cassandra.apache.org/



(a) DSM of Cassandra-6561 FeatureSpace.



(b) DSM of Cassandra-2617 FeatureSpace.

Figure 1: DSMs of two Feature Spaces with structural dependencies
x: structural dependency between files

```
commit 8b24d813b586cbc16783d7b7a384ebfaef4ca3ec
Author: Jonathan Ellis <jbellis@apache.org>
Date:   2011-06-05 13:41:30 +0000

    add cql drop index
    patch by pyaskevich and jbellis for CASSANDRA-2617

    git-svn-id: https://svn.apache.org/repos/asf/cassandra/branches/cassandra

1   0   CHANGES.txt
9   0   doc/cql/CQL.textile
37  0   src/java/org/apache/cassandra/config/CFMetaData.java
10  0   src/java/org/apache/cassandra/cql/Cql.g
47  27  src/java/org/apache/cassandra/cql/QueryProcessor.java
1   1   src/java/org/apache/cassandra/cql/StatementType.java
7   6   src/java/org/apache/cassandra/thrift/CassandraServer.java
51  13  src/java/org/apache/cassandra/thrift/ThriftValidation.java
31  4   test/system/test_cql.py
1   1   test/system/test_thrift_server.py
2   1   test/unit/org/apache/cassandra/SchemaLoader.java
```

Figure 2: An Example Commit for Cassandra

$< FS, FSDep >$, where FS is a set of feature spaces and $FSDep$ is the *feature dependency*. A FDSM is also visualized in a square matrix, whose columns and rows are labeled with the same set of features in the same order. If a cell in row x, column y—i.e. $cell(x, y)$—is not empty, it means feature x depends on feature y. For example, Figure 3 shows a FDSM formed by a subset of features in the *Cassandra* project. The $cell(4, 2)$ (circled) means feature CASSANDRA-6561 depends on Cassandra-2617, as we have seen above.

This FDSM is clustered using the DRH algorithm [11]. The feature-level DRH (FDRH) in Figure 3 has four layers: L1: (rc1-rc4), L2: (rc5-rc9), L3: (rc10-rc18) and L4: (rc19-rc22). Features in L2 only depend on the features in L1. Features in L3 depend on the features in its upper layers: L2 and L1, but are not depended on by any other features. L4 contains isolated features that do not have any dependencies with any other features. According to this DSM, features in L4 may be changed or replaced without influencing other features. Feature modules in L3 may also change together freely. In this paper, we call layers like L1 and L2 to be *"upper layers"*, a layer like L3 to be *"dependent-free layer"* and a layer like L4 to be *"isolated layer"*. The FDRH of a project could contain many *upper layers* and one *dependent-free* layer, but at most one *isolated layer*.

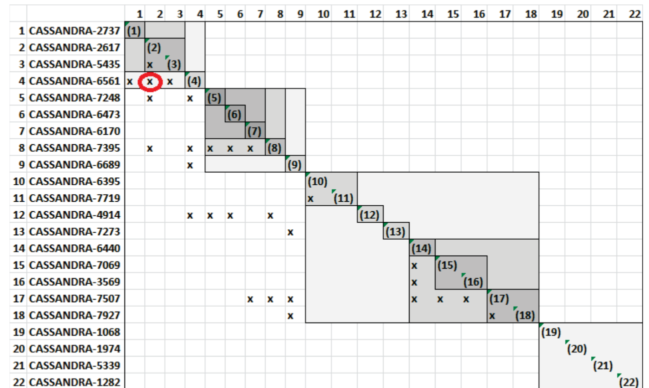


Figure 3: An Example of *Feature Design Structure Matrix*
x: Feature Dependency

2.4 Feature Decoupling level (FDL)

DL [30] has been shown to be an effective measure of independence among modules at the file level. It this research we modified the algorithm slightly so that it can be applied to a feature-level DRH. For each project, we calculated its FDL as follows:

Basic definition. $\#AllFts$: the total number of detected features of a project; $\#Fts(M_j)$: the number of features in a FDRH module, M_j .

Calculation. Given a FDRH with n layers, its FDL is equal to the sum of the FDL of all the layers:

$$FDL = \sum_{Li=1}^{n} FDL_{L_i} \tag{2}$$

For an *upper layer* with k modules, we calculate its FDL as follows:

$$FDL_{L_i} = \sum_{j=1}^{k} [\frac{\#Fts(M_j)}{\#AllFts} \times (1 - \frac{\#Deps(M_j)}{\#LowerLayerFts})] \tag{3}$$

where, $\#Deps(M_j)$ is the number of features within lower layer modules that directly or indirectly depend on M_j. For a feature-level module in upper layers, the number of other features it influences are taken into consideration: the more feature it influences in lower layers, the lower its FDL. Based on the definition of FDRH, a module in upper layers must influence some features in lower layers.

For a module in the *dependent-free layer*, the number of features *within* this module is considered. A module in this layer can be changed without affecting other modules, but changes to the module itself could be difficult if this module includes too many features. Prior research in cognitive complexity [19] has shown that people can comfortably process approximately 5 "chunks" of information at a time. Accordingly, we consider that a module with more than 5 features would be difficult to process. Thus, for the *dependent-free layer* with k modules, if $k > 5$, we calculated its FDL as follows:

$$FDL_{L_i} = \sum_{j=1}^{k} \frac{\#Fts(M_j)}{\#AllFts} \times (log_5(\#Fts(M_j)))^{-1} \tag{4}$$

if $k <= 5$, we calculated its FDL as follows:

$$FDL_{L_i} = \sum_{j=1}^{k} \frac{\#Fts(M_j)}{\#AllFts} \tag{5}$$

For an *isolated layer* with k modules, we calculated its FDL as equation 5.

Figure 4 presents 3 different FDSMs with different FDLs. The features in Figure 4a are completely isolated from each other, hence its FDL is 100%. Figure 4b shows a FDSM where the features are highly coupled except for Feature 10 that could be changed easily, and its FDL is just 10%. Figure 4c shows a FDSM with 5 layers, and its FDL is 50%, meaning that 50% of all the features were independent from each other.

Please note that despite their similarities FDSM and *structural* DSM (SDSM) are very different. An FDSM does not represent syntactic relations among files as a SDSM does. Instead, an FDSM *only* captures evolutionary dependencies and time sequence. For example, consider feature fs_a containing files $\{f_1, f_2, f_3\}$, and feature fs_b that was added after fs_a, containing files $\{f_4, f_5\}$. Since these two features do not share files, they appear to be mutually independent in a FDSM. However, it is possible that f_4 calls f_1, a syntactic relation that links fs_a with fs_b, but is not reflected in the FDSM. It is possible that when f_1 in feature fs_a is changed, f_4 in fs_b also has to be changed. However, as we demonstrate later, even if a FDSM does not represent syntactic relationships among features, it represents their *fundamental* (semantic) relationship: files within the same feature indeed change together far more often than files not within the same feature, showing that a FDSM reveals meaningful feature-level modules.

3 EVALUATION

To evaluate these concepts, we studied the following research questions, using a corpus of 17 open source projects as our empirical base. We have created a set of tools to support our evaluation, including programs that take a system's revision history, issue tickets, and a snapshot of its source code as inputs, and generate FDSMs and DSMs for each feature space of each project. We use Titan [36] to visualize and cluster FDSMs.

RQ1: Do files contained in a feature space form a semantically meaningful group in that they are changed together more often?

To measure the ease of adding new features into a system, we define a feature space to contain files that were committed together to resolve issues labeled as "new feature". But we need to understand if this simple definition is adequate. For example, it is possible that, as software evolves, files are added to a feature, but such commits are not labeled as "new features". In this case, the newly added files will be missing from our feature space.

The question is, can our (relatively simple) feature space model capture file groups that are meaningful and useful to developers? If the answer to this question is yes, then each feature space can form a maintainable unit and should be treated separately. For example, maintenance tasks for each feature can be assigned to different teams, and if there is a bug in a feature, developers can examine the corresponding feature space—identifying potentially implicated files—to aid in debugging.

RQ2: Are feature spaces that depend on each other more likely to change together?

Similarly, if the answer to this question is positive, it means that FDSMs have the potential to help developers better maintain and debug features: if two features depend on each other and are frequently changed together, when one feature changes, the developers can use FDSMs to figure out which other features (and hence files) are also likely to be changed.

RQ3: Is FDL consistent with architectural maintainability metrics?

It is widely accepted that a better-modularized system should be easier to maintain. And, more importantly, a better-modularized system should make it easier to add or modify features. Mo et al. [30] has shown that DL can reliably reflect the modularity level of a software system. If FDL faithfully reflects the ease of adding features then it should align with architectural quality measures. We investigate this question by exploring the correlation of FDL—a *history* measure—and DL—a *structural* measure.

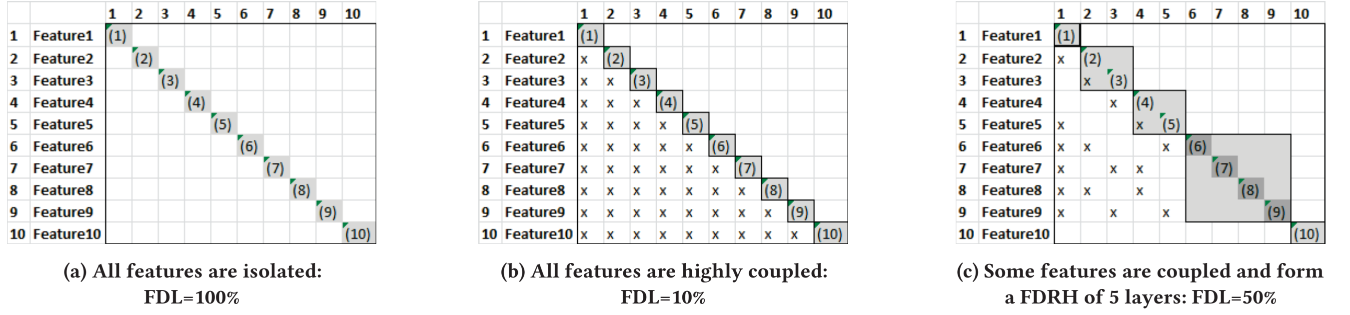
RQ4: Is it possible for a well-modularized system to have a low FDL, or for a seemingly poorly modularized system to have a high FDL?

Several of our industrial collaborators have complained that sometimes a system appears to have "good quality" as measured by, for example, coupling and cohesion metrics, but can still be difficult to understand and maintain. It would be interesting to see if there are similar phenomena in open source projects.

Since the ultimate purpose of architectural design and refactoring is to ease feature delivery (rather than improving metrics) if this divergence does occur, it is important to understand the reasons so that the architecture can be further improved. We demonstrate how to use FeatureSpace, FDL and FDSM to explore feature/architecture evolution and interaction.

3.1 Subjects

To answer the above research questions, we selected 17 open source projects as subjects.We chose these projects because they have different sizes, lengths of history and domains, and have evolved for significant periods of time. Moreover, these projects are well



(a) All features are isolated: FDL=100%

(b) All features are highly coupled: FDL=10%

(c) Some features are coupled and form a FDRH of 5 layers: FDL=50%

Figure 4: Three different FDSMs
x: Feature Dependency

managed; for example, features are labeled separately from other types of issues. Table 1 presents our subject projects. Table 2 reports the statistics of the feature spaces for each project, and the last row shows the average for each column.

Table 1: Basic facts of the subjects

Subjects	Rel.	#Files	#Mons	#Com.	LOC
Activemq	5.13.3	1235-4120	124	9241	138-404k
Avro	1.7.7	156-444	52	1383	41-178k
Camel	2.12.4	1838-9866	84	16715	111-764k
Cassandra	2.1.2	311-1337	66	15330	45-249k
CXF	3.0.9	2861-6011	93	8937	315-709K
Flink	1.0.2	1728-3210	22	8799	190-444k
GeoTools	13.6	6328-7748	59	4906	768-1013k
Hibernate	4.3.11	4780-7335	73	5675	394-568k
Hive	1.2.1	511-3897	79	6889	142-753k
Mahout	0.11.1	455-1213	80	3424	33-125k
Nutch	2.3.1	376-438	101	2034	58-147k
OpenJPA	2.4.0	1266-4457	100	4729	195-494k
PDFBox	1.8.9	447-799	73	3664	49-121k
Pig	0.15.0	594-1638	74	2754	95-369K
Spring	3.2.16	3253-4920	73	11522	278-493k
Tika	1.8	131-653	94	2497	10-79k
Wicket	6.19.0	1879-3081	98	18373	127-279k

Rel.: latest release number; #Files: range of file count; #Mons: evolution time in months; #Com.: total number of commits; LOC: range of LOC between the first and last releases.

3.2 Feature Space as Maintainable Unit

To answer the first research question, we quantify how frequently two files were changed together in the revision history using *cochange*: when two files A and B were changed together in one commit, we consider that their $cochange(f_a, f_b)$ is 1. The higher the value of $cochange(f_a, f_b)$, the more frequently these two files were changed together. For files within a feature space, FS_k, we calculated how frequently each pair changed together and the average:

$avg_CC_FS_k = 1/n \times \sum_i^n \sum_j^n cochange(f_i, f_j)/n$, where f_i, f_j are files involved in FS_k, $i \neq j$, n is the number of files involved in FS_k and $n >= 2$.

For each file in FS_k, we also calculated how frequently it was changed with files in the *other* feature spaces:

Table 2: Feature Space Summary for each project

Subjects	#FS	#Files.Dist	FS.Size	#InCom.
ActiveMQ	195	1450	11.5	2.1
Avro	83	301	7.2	1.3
Camel	557	4396	11.4	2.8
Cassandra	232	578	7.3	1.6
CXF	134	1150	10.8	3.5
Flink	74	880	14.9	3.2
GeoTools	66	744	14.7	1.8
Hibernate	106	3230	38.1	2.6
Hive	318	1548	10.7	1.1
Mahout	82	579	8.9	2.1
Nutch	40	201	9.3	1.7
OpenJPA	78	804	16.2	3.4
PDFBox	40	201	5.9	2.5
Pig	112	784	10.5	1.3
Spring	214	1120	8.8	2.4
Tika	101	294	6.2	3.5
Wicket	100	1458	17.6	2.1
Average	149	1160	12.3	2.3

#FS: the number of feature spaces;
#Files.Dist: the number of distinct files of all feature spaces;
FS.Size: average number of files in a feature space;
#InCom.: average number of commits made for each feature

$avg_CC_nFS_k = 1/n \times \sum_i^n \sum_j^m cochange(f_i, f_j)/m$, where f_i is inside FS_k, f_j is a file involved in *other* feature spaces, n is the number of files in FS_k, m is the total number of files in *other* feature spaces.

Table 3 reports the average of all $avg_CC_FS_k$ values (*aaCC*) and the average of all $avg_CC_nFS_k$ (*aaCCn*) values for each project. These results show that *aaCC* is much higher than *aaCCn* over all projects, indicating files within a feature space change together more frequently.

To be more rigorous, we used the *Wilcoxon signed-rank test*, a non-parametric statistical hypothesis test for comparing two related samples, to compare whether the $avg_CC_FS_k$ is significantly larger than the $avg_CC_nFS_k$ over all features for each project. The tests on *all* of the subject projects have P-values less than 0.01. We can thus conclude that each feature space captures a group of highly evolutionarily coupled files, which form a maintainable unit. The implication is that files in a feature space have a high likelihood

Table 3: *aaCC* **vs** *aaCCn* **for each project**

Project	$aaCC$	$aaCCn$	Project	$aaCC$	$aaCCn$
ActiveMQ	3.221	0.039	Mahout	2.515	0.069
Avro	2.707	0.165	Nutch	1.654	0.079
Camel	1.969	0.013	OpenJPA	2.038	0.024
Cassandra	11.104	0.467	PDFBox	2.340	0.145
CXF	2.347	0.016	Pig	1.725	0.059
Flink	1.503	0.023	Tika	1.815	0.081
GeoTools	1.229	0.004	Spring	2.102	0.013
Hibernate	1.223	0.011	Wicket	1.726	0.036
Hive	5.928	0.132	-	-	-

of being changed together in the future. When a file in a feature space is changed, developers can effectively identify other potential changes by examining the feature space.

As an example, figure 5 depicts a DSM with both structure and history relations among files in the `Cassandra-6561` feature space. The number in a cell, which we call "co-change", indicates how many times the file on the row and the file on the column have changed together in a period of history. From the large co-change numbers in the cells we can tell that files within this feature space have changed together very often in the project's history.

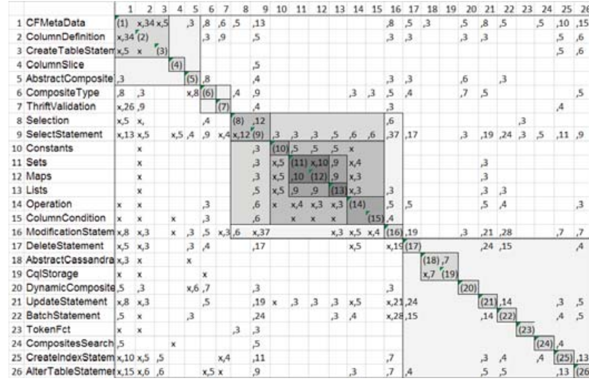


Figure 5: DSM of Cassandra-6561 with structural and evolutionary dependencies
x: structural dependency between files, number: co-changes

3.3 Dependent Features

To answer the second research question, we first quantify how frequently two features, FS_a and FS_b, were changed together. We calculated the average *cochange* between their involved files as follows:

$aCC(FS_a, FS_b) = 1/n \times \sum_i^n \sum_j^m cochange(f_i, f_j)/m$, where f_i and f_j are the file involved in FS_a and FS_b, *n* and *m* are the number of files in FS_a and FS_b respectively.

For each of the dependent features, FS_k, we calculated the average *cochange* among their files as follows:

$FS_k_Dep_aCC = 1/m \times \sum_i^m aCC(FS_k, FS_i)$, where *m* is the number of features depended by FS_k, FS_i is a feature depended by FS_k. There are shared files among dependent features, to avoid biased results from the co-changes between the shared files and other files, we excluded the co-changes involving shared files in our calculation.

For independent features:

$FS_k_NDep_aCC = 1/n \times \sum_i^n aCC(FS_k, FS_i)$, where *n* is the number of features independent to FS_k.

Table 4 reports the average of all $FS_k_Dep_aCC$ values (*daaCC*) and the average of all $FS_k_NDep_aCC$ (*daaCCn*) values for each project. These results demonstrate that *daaCC* is much higher than *daaCCn* over all projects, which indicates that a feature's files were changed together with its dependent feature's files more often.

Table 4: *daaCC* **vs** *daaCCn* **for each project**

Project	$daaCC$	$daaCCn$	Project	$daaCC$	$daaCCn$
ActiveMQ	1.40	0.44	Mahout	1.60	0.19
Avro	1.43	0.51	Nutch	0.43	0.22
Camel	0.68	0.10	OpenJPA	0.48	0.21
Cassandra	7.13	5.04	PDFBox	1.12	0.56
CXF	0.92	0.08	Pig	0.77	0.30
Flink	0.60	0.13	Tika	0.78	0.34
GeoTools	0.32	0.04	Spring	0.80	0.10
Hibernate	0.35	0.17	Wicket	1.00	0.40
Hive	3.23	2.56	-	-	-

Again, for each project, we conducted a *Wilcoxon signed-rank test* to explore whether $FS_k_Dep_aCC$ is significantly larger than $FS_k_NDep_aCC$. We observed that *all* these tests show P-values less than 0.01, indicating that files in dependent features are more likely to change together, compared to files in mutually independent features. The implication is that if one feature changes, the developer can use a FDSM to check what other features might also need to be changed.

3.4 Feature Independence and Structural Independence

The results from the above two research questions show that files in a feature space or in dependent features are more likely to change together. A maintainable architecture should allow features to be added and changed relatively independently. And it is widely accepted that a well-modularized system should be easy to maintain. Thus, if FDL is a valid metric it should align with architectural modularity measures.

In Mo et. al.'s work [30], the authors demonstrated that DL can be used to compare different projects, and can indicate significant architectural changes, such as refactoring or degradation. They also have shown that DL could reliably reflect the level of modularity of a software system. To evaluate if FDL is a reliable indicator of the ease of adding and modifying features, we explored the correlation between FDL—a pure *history* measure—and DL—a pure *structural* measure.

For this purpose, we calculated the DL and FDL values for each of our subject projects, as reported in Table 5. Both DL and FDL change during the evolution of a project, so we chose to calculate these metrics from the latest version of each project. We made this choice because DL [30] values usually remain stable in consecutive

releases of a project, unless there was substantial refactoring or degradation. Based on our detailed analysis of these projects, we did not observe any significant changes in their architectures during the latest few releases, so the DL from the last release was determined to be sufficiently representative in each case.

Table 5: Metric values for each project

Project	*DL*	*FDL*	Project	*DL*	*FDL*
ActiveMQ	78%	52%	Mahout	92%	87%
Avro	82%	63%	Nutch	77%	60%
Camel	84%	68%	OpenJPA	70%	53%
Cassandra	39%	16%	PDFBox	47%	86%
CXF	88%	91%	Pig	55%	49%
Flink	79%	65%	Tika	84%	59%
GeoTools	88%	87%	Spring	86%	72%
Hibernate	65%	42%	Wicket	72%	65%
Hive	55%	21%	-	-	-

For FDL values, since features can be quite different from each other, it makes sense that FDL could fluctuate more than DL, as we will show. In particular, it is natural that at the beginning of a project when the architecture is not stabilized, FDL values may change more rapidly. We deliberately chose projects that have evolved for a long time (more than 6.5 years on average) under the assumption that the architecture should be stable and so should the project's ability to support feature evolution. As a result, the FDL of the latest version should be representative.

We conducted a *Pearson Correlation Analysis* between DL and FDL. The results show the Pearson values is 0.65 and p-value is 0.004, meaning the correlated relationship is statistically significant. The correlation of 0.65 is considered strong. However, from Figure 6, we observed a prominent outlier: PDFBox, whose DL is only 47%, but its FDL is as high as 86%. If we remove this outlier, the correlation among the remaining 16 projects increases to 0.91, indicating that, within these projects, those with higher DL (i.e., those that are well-modularized) also have very high FDL (features can be added/modified more independently). In particular, all 3 projects with highest DL values (larger than 87%) also have FDL of at least 87%, meaning that more than 87% of their features can be maintained independently.

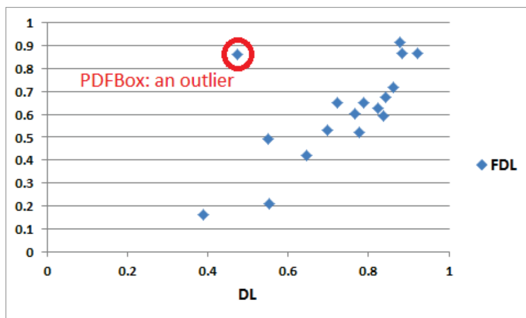


Figure 6: The Correlation between DL and FDL

This study shows that in general, FDL and DL are strongly correlated but we did find one exception. We will examine PDFBox in detail in the next subsection to understand why this project is special, and to understand the meaning of a divergence between FDL and DL.

3.5 The Interaction between Architecture and Feature Addition

The fact that PDFBox has a relatively low DL but very high FDL triggered our curiosity to dig more into its architecture and evolution history. Moreover, the discrepancy between architectural measurement and actually maintainability happens in reality. Several of our industrial collaborators have complained that sometimes a system has really good quality measures as measured by, for example, coupling and cohesion metrics, but still can be difficult to maintain. It would be interesting to see if there are similar phenomena in open source projects, and most importantly to explore when, where, and why these discrepancies happen and how to explore the interaction between architecture and feature addition so that architects can make decisions about if and when the architecture should be improved.

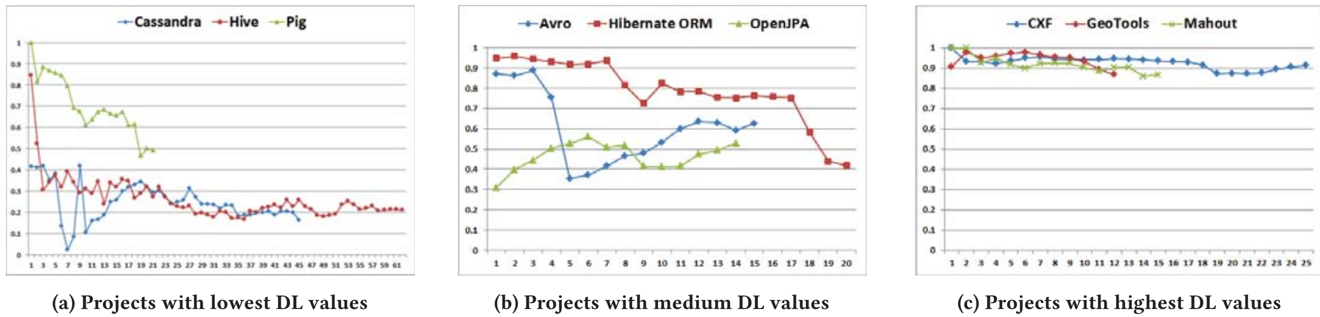
Next we explore why PDFBox was an outlier, and the variation of FDL of each project to see how the ability of adding new features changes over time. Finally, we qualitatively explain how DL, FDL, DSM and FDSM can be used to explore the interaction between architecture and features.

3.5.1 PDFBox. We first note that even though PDFBox has been evolving for about 6 years, it has only 40 features recorded in its history, whereas the average number of features for all 17 of our subject projects are 149. In addition, the average number of files within each feature is the smallest among the 17 projects: 5.9 vs. the average of 12.3. We then calculated the number of files within all PDFBox feature spaces and found that 26 out of 40 features (65%) have just 1 or 2 files.

To further understand the architecture and features of PDFBox, we generated the FDSM for the latest version. Based on the FDSM, we find that the 4 largest features—with more than 10 files each—are among the earliest features added to the system, as part of the infrastructure. For example, the largest feature of this project, PDFBox-572 (73 files), was added as the 5th feature. According to its revision history: the purpose of this feature was "*Upgrading PDFBox (incl. JempBox and FontBox) to use Java 5 specifications*".

Next we studied the structural DSM of PDFBox and observed several big "blobs" that explain its relatively low DL: for example, there is a parser in the system (which is naturally complex), and it employed a visitor pattern to handle exceptions (which contains a dependency cycle). Fortunately, most features added later to this system do not involve this infrastructural code. Also, in PDFBox's own documentation, it states the following: "*Optional dependencies: Some features in PDFBox depend on optional external libraries. You can enable these features simply by including the required libraries in the classpath of your application.*" This optionality may explain why the number of files of most features, extracted from co-commits, is small.

In summary, PDFBox exemplifies the case where some parts of the architecture are not well-modularized, but they are stable and not involved in new features. In such a case, it is still possible to



(a) Projects with lowest DL values

(b) Projects with medium DL values

(c) Projects with highest DL values

Figure 7: FDL variations

achieve a high FDL. This is consistent with the notion of architectural debt [23]: even if a part of the system is not well modularized or has a lot of code smells, as long as this part is not generating interest, it is not a debt.

3.5.2 The Variation of FDL. The exploration of PDFBox triggered another question: are there cases where the DL is very high, but the FDL is low? Also, does FDL remain stable if the architecture remain stable? To answer these questions, we first ordered the features added to each project according to their commit time, and then calculated the FDL values for every 5 features added. That is, we calculated the FDL after the first 5 features were added, the first 10 features, 15 features, etc.. Finally, we presented the variations of FDL for different projects. Figure 7 depicts several representative cases: the FDL variation for projects with lowest DL values (Figure 7a), highest DL values (Figure 7c), and the projects in the middle (Figure 7b).

These charts reveal the following interesting phenomena: for projects that are extremely well modularized (high DL values), their features are also highly independent from each other (high FDL values), over all or most of their feature addition history. In the projects with low DL values, by contrast, their FDL values fluctuated when the first set of features were added, and then stabilized at a low level (less than 40%), or showed a steady downward trend.

The projects whose DLs are in the middle, however, experienced ups and downs during their feature addition history. Take Hibernate ORM as an example. This project has evolved for over 6 years, and its FDL chart showed that its features are less and less independent over time. We calculated the DL values of its previous snapshots, and found that its DL decreased from 69% to 65%. For Avro and OpenJPA, their DLs increased from 72% to 82% in 4 years, and 55% to 77% in 8 years respectively. We can see fluctuating but upward trends for both projects. The FDL of AVRO is higher than that of OpenJPA most of the time, which is consistent with the fact that AVRO has a higher DL in general. These charts, again, illustrate that FDL and DL are highly correlated: they tend to increase or decrease together.

However, even for a project with high DL, its FDL can be low from time to time. For example, the FDL measured after the 33th feature of AVRO is the lowest of all its history. If the measures were taken at that time, how does one tell if the low FDL is caused by a degrading architecture, or it is just because the system hasn't evolved long enough yet? Next we explore this question.

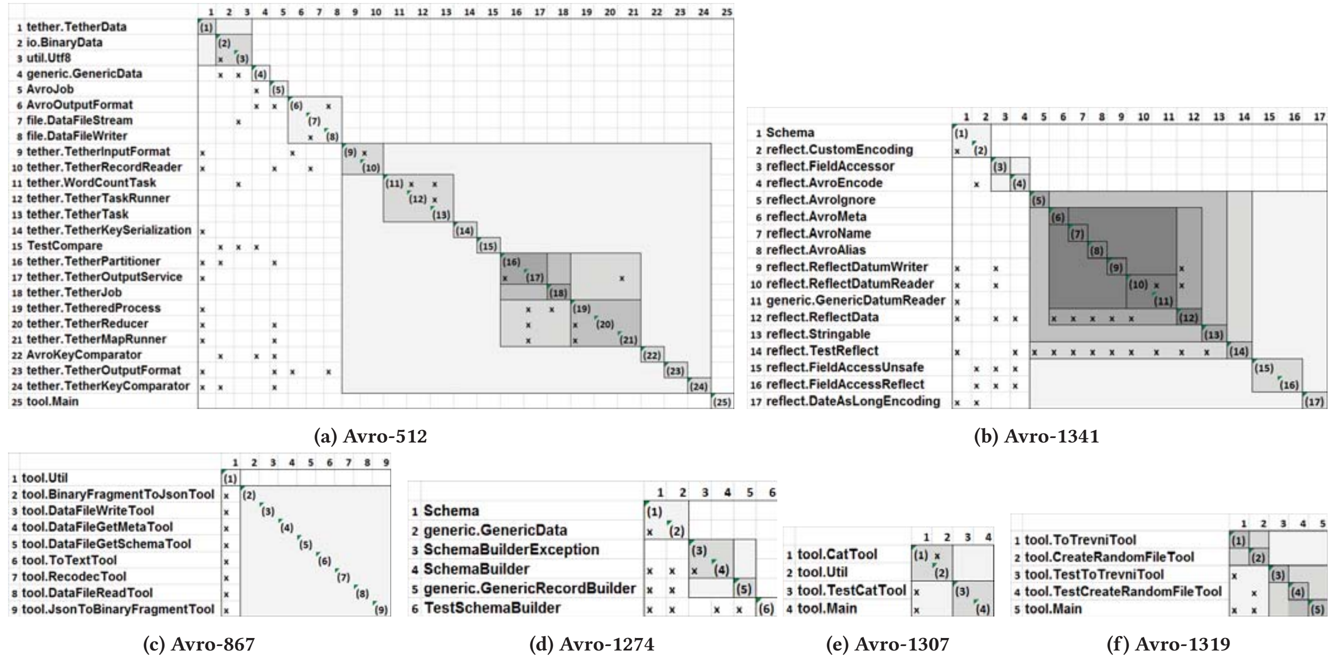
3.5.3 The Interaction of Architecture and Features. Now we demonstrate how to use DSM and FDSM to explore the interaction between architecture and features, using AVRO as an example. Figure 8a is the feature space DSM of AVRO-512, the 29th feature added to the system; after this feature was added, the FDL dropped to 36%. From the DSM, we can see that this feature has 25 files, involving multiple namespaces.

The revision log of AVRO made it clear that this feature is also an infrastructure feature, described as "*Define And Implement Mapreduce Connector Protocol*". This feature was added in 2010, the second year after AVRO became an Apache project. By contrast, the features added more recently were much smaller and more independent. Figures 8b-8e depict the feature space DSMs of 5 features added in 2013-2014. We can see that these features are all small, and the files were all added or changed within the same namespace. According to their revision log, these are not infrastructure features, and the architecture supports their addition easily.

For an architect, feature space DSMs can be used to analyze why a FDL is decreasing, and to analyze each feature: if a feature is naturally complex and will be part of the system's infrastructure, it is normal that its addition may temporarily lower the DL and FDL. As long as the architect envisions that most new features can be easily added in the future, this is fine. If the value of FDL indeed increases over time, this means that the architect's vision is correct. By contrast, if an architect observed continuously decreasing FDL and DL values, and most features, infrastructural or not, could not be added independently, then this is an alarming sign that the features are not aligned with the architecture, and the architecture should be reevaluated.

3.6 Answers to Research Questions

Now we can answer our four research questions. For RQ1 and RQ2, we can answer for both questions from the data: indeed the files contained in a feature space form a meaningful group that should be maintained together, and dependent features are likely to change together. This result implies that FDSM could be used to monitor the maintenance, evolution, and interaction of features. We can also answer RQ3 and RQ4 positively and their answers complement each other: for RQ3, the data shows that FDL is positively correlated with DL if we examine enough evolution history: consistently high DL scores are associated with consistently high FDL scores over time and vice versa. On the other hand, if we examine shorter time



(a) Avro-512 (b) Avro-1341

(c) Avro-867 (d) Avro-1274 (e) Avro-1307 (f) Avro-1319

Figure 8: Feature Space DSMs of AVRO – *x: structural dependency between files*

periods, even for a project with high DL, its FDL can be low from time to time if the features added during that period happen to be infrastructural. It is also possible for a project with low DL to have high FDL, if the complex part of the system is not active or changeable, or the addition of features does not involve the part of the system that has architecture flaws. Our analysis shows that FDL and FDSM can be used to assess the impact of adding new features to the architecture, to examine the impact of each feature, and to monitor if the overall architecture is degrading because of feature additions, which can be reflected as a decreasing DL in the long run.

4 DISCUSSION

4.1 Limitations

First, we assume that independence among features implies ease of adding features. Of course, the actual speed of adding features may be affected by other factors unrelated to interdependencies such as the experience and skills of developers, time-to-market, and the availability of resources. Thus we are not claiming that, in all cases, the more independence among features, the faster features can be delivered.

For similar reasons, we did not use feature delivery velocity—the length of time between a feature being added to the issue tracking system and when it is committed and released—of these 17 subjects as a way of evaluating FDL. The time spent to add features is highly influenced by the nature of project, the number of users, the number of contributors, the maturity of project, and so forth. For example, a project that is mature may not have many new feature commits, and a project with relatively few contributors may take longer to add new features. But this, by itself, does not mean that such projects do not have the *i*nherent capability to add features quickly. Thus a comparison of these measures among different projects can be meaningless, and thus these "*measurements*" are not qualified as proper "*metrics*".

Based on our research, we believe that FDL is independent of other factors such as user experience and project popularity, and can be used to compare different projects, and to monitor the congruence between architecture and feature addition.

4.2 Threats to Validity and Future work

First, even though we made an effort to choose projects of different sizes, domains, and ages, we only studied 17 projects, most of which were from the Apache open source community. Due to the limitations of our evolving tool set, we can only process projects that use SVN or Git, and all of the chosen projects were implemented using Java. Hence we can not claim that our results are generalizable to other projects implemented using different programming languages or managed by other version control tools. Extending the experiments to a broader set of projects is future work.

Second, most of the projects we analyzed have moderate or high DL measures: thus we know that they have moderate or high modularity. We tried to select more projects with lower DLs, but they either did not have a sufficiently long evolution history, were not active, or did not reliably distinguish features from bugs in their revision histories. We intend to analyze more projects with various levels of DL values in the future. However it is possible that, because of the poor modular structure of low-DL systems, they are not capable of supporting long-term feature addition, which could be the reason we can not find them and collect their feature data. Poor project management, such as not linking commits to issues,

could be another reason. These are hypotheses that we plan to test in the future.

Third, since we used issue tracking records to find feature IDs, and then mined revision records to find the mapping between commits and features, our results rely on the accuracy of the revision history and issue tracking records. This is similar to many other studies that depend on evolution history data. We thus can not claim that our results apply to projects that are relatively new or that have a short history record. Addressing these issues is also part of our future work.

Forth, we consider each feature space to only contain files which were changed in a commit labeled as "new feature". But as software evolves, a feature space may miss some source files which committed to implement the respective feature but not label as "new feature". However, as we describe earlier, each of our feature spaces could effectively model files which forms a meaningful maintainable unit, meaning that files in a feature space are more likely to be changed together.

Finally, this research is motivated by our industrial collaborators' question: "*How do we know if a refactoring is successful?*" All of our collaborators have used multiple tools to measure source code structure, but they do not have an effective way to measure if the ability of adding new features has increased, which is what they care most in the end. Accordingly, applying FDL to industrial projects, and evaluating it in longitudinal studies, is our final future goal.

5 RELATED WORK

Feature-related Research. Various feature location techniques have been described in [16, 22]. These papers proposed different approaches [2, 14, 29, 31] for feature location. For example, [2] collected and analyzed execution traces of target features in software based on probabilistic ranking; [29] extracted and ranked program elements to investigate by analyzing the topology of structural dependencies; [14] presented a tool, Hipikat, using information retrieval techniques to identify the artifacts related to a feature. More recently, [18] presented predicted failure-prone configurations by using feature locality and historical data. Based on diff set, [21] used two heuristics to improve the accuracy for locating distinguishing features. Using a data fusion model, [15] combined information retrieval, execution and link analysis algorithms to improve feature location. Feature interaction [3, 5, 33] is where the behavior of one feature may be affected by another feature. Feature-based specifications were used in [3] to detect interactions and showed that the detection approach is effective in most of the studied systems. Feature interaction algebra was used by [5] and they demonstrated that this approach could improve the precision of formalizations for concepts used in software product lines.

None of these works considers the historical dependency between features. Our FDSM explicitly presents the historical dependencies between features and the depended featured are more evolutionarily coupled. Besides, our FDL has the potential to reflect the quality of software architecture.

Metrics and Quality. Metrics have been widely studied to measure software quality. McCabe Cyclomatic complexity [25] was proposed in 1970s. CK metrics [13] and MOOD Metrics [17] are two suites of metrics used for analyzing object-oriented design. These are code-based metrics. In addition, many other measures have been proposed, such as LOC, that can be extracted from history. These metrics or measures have demonstrated their effectiveness in bug prediction and localization [12, 20, 26, 28, 34]. For example, [20] used the ranking information of each statement to assist fault location. [26] examined different code metrics and observed that a combination of these metrics is useful to predict defects in systems. However, they couldn't find a unique combination of code metrics which is useful in all different projects.

Unlike these metrics, FDL has shown the potential to measure how well a system can support feature addition during software evolution. High correlation between FDL and DL suggests that the ability to add features is rooted in architecture, not individual code files.

Architecture Metrics. Many architectural metrics have also been proposed to measure software quality. Propagation Cost (PC) [24] was proposed to measure how tightly a system's files are coupled to each other. Independence Level (IL) [32] was proposed to measure how many files could be decoupled into structurally independent modules. [8, 9] have investigated system-level architecture metrics and their correlations with the ratio of local change-sets. [37, 38] proposed multiple coupling metrics which presented correlations with history changes and reuse effort. Decoupling Level (DL) [30] was proposed to measure how well a system is decoupled. However, all these metrics only consider structural relations. FDL is derived from revision history and reflects how well a system can support feature addition in reality.

6 CONCLUSION

To assess, compare, and monitor the ease of adding new features to a software system we have proposed and formally defined *Feature Space*, *Feature Dependency*, and *Feature Decoupling Level*. After investigating thousands of features within 17 open source projects, we reported that files within the same feature space, as well as files in dependent features are much more likely to be changed together, indicating that these feature spaces form meaningful "modules" that should be identified so that they can be analyzed and maintained. We also showed that in, most cases, the FDL metric (a pure *history* measure) and DL (a pure *structural* measure) are highly correlated, indicating that FDL could reliably reflect architecture maintainability.

Our study also showed that, in some rare cases a system with a low DL can also have a high FDL. Furthermore even for systems with high DLs, adding features may be difficult from time to time, depending on the nature of the features, the maturity of the architecture, and their interactions. Using real examples, we demonstrated how DSMs, FDSMs, DL and FDL can be combined to analyze the interaction between architectural structure and features, so that the architecture can be evolved in an informed way, to better support feature addition.

ACKNOWLEDGMENTS

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

REFERENCES

[1] G. Antoniol and Y.-G. Gueheneuc. Feature identification: a novel approach and a case study. In *Proc. 21stIEEE International Conference on Software Maintenance*, 2005.

[2] G. Antoniol and Y.-G. Gueheneuc. Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering*, 32(9):627–641, 2006.

[3] S. Apela, A. von Rheina, T. ThÍźmb, and C. Kastnerc. Feature-interaction detection based on feature-based specifications. *Computer Networks*, 57(12):2399 – 2409, 2013.

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

[5] D. Batory, P. Hofner, B. Moller, and A. Zelend. Features, modularity, and variation points. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, pages 9–16, 2013.

[6] F. Beck, B. Dit, V.-M. Jaleo, W. Daniel, and P. Denys. Rethinking user interfaces for feature location. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 151–162, 2015.

[7] M. Bertrand. *Object-oriented Software Construction (2Nd Ed.)*. Prentice-Hall, Inc., 1997.

[8] 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.

[9] 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.

[10] 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.

[11] Y. Cai, H. Wang, 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.

[12] 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.

[13] 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.

[14] C. Davor, M. G. C., S. Janice, and B. K. S. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, June 2005.

[15] B. Dit, R. Meghan, and P. Denys. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.

[16] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: A taxonomy and survey. In *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

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

[18] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Using feature locality: Can we leverage history to avoid failures during reconfiguration? In *Proceedings of the 8th Workshop on Assurances for Self-adaptive Systems*, pages 24–33, 2011.

[19] 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.

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

[21] R. Julia and C. Marsha. Locating distinguishing features using diff sets. In *Proc. 27thIEEE/ACM International Conference on Automated Software Engineering*, pages 242–245, 2012.

[22] R. Julia and C. Marsha. *A Survey of Feature Location Techniques*. Springer Berlin Heidelberg, 2013.

[23] 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.

[24] 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.

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

[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] A. Nouh, D. Natalia, C. M. L., and M. J. I. Improving feature location by enhancing source code with stereotypes. In *Proc. 29thIEEE International Conference on Software Maintenance*, pages 300–309, 2013.

[28] 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.

[29] R. M. P. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology*, 17(4), 2008.

[30] M. Ran, C. Yuanfang, K. Rick, X. Lu, and F. Qiong. Decoupling level: A new metric for architectural maintenance complexity. In *Proc. 38th International Conference on Software Engineering*, pages 499–510, 2016.

[31] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *Proc. 14th IEEE International Conference on Program Comprehension*, 2006.

[32] 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.

[33] A. Sven, K. Sergiy, S. Norbert, K. Christian, and G. Brady. Exploring feature interactions in the wild: The new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, pages 1–8, 2013.

[34] 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.

[35] 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.

[36] 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.

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

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