Understanding Evolutionary Coupling by Fine-grained Co-change Relationship Analysis

***Abstract*—Frequent co-changes to multiple files, i.e., evolutionary coupling, can demonstrate active relations among files, explicit or implicit. Although evolutionary coupling has been used to analyze software quality, there is no systematic study of underlying reasons – which could be the root cause of quality problems – that drive frequent co-changes.**

In this paper, we report an empirical study on 27,087 co-change commits of 6 open-source systems with the purpose of understanding the observed evolutionary coupling. We extracted fine-grained change information from version control system to investigate whether two files exhibit particular kinds of co-change relationships. We consider code changes on 5 types of program entities (i.e., field, method, control statement, non-control statement, and class) and identified 6 types of dominating co-change relationships. Our manual analysis showed that each of the 6 types can be explained by structural coupling, semantic coupling, or implicit dependencies. Temporal analysis further shows that files may exhibit different co-change relationships at different phases in the evolution history. Finally, we investigated co-changes among multiple files by combining co-change relationships between related file pairs and showed with live examples that rich information embedded in the fine-grained co-change relationships may help developers to change code at multiple locations. Moreover, we analyzed how these co-change relationship types can be used to facilitate change impact analysis and to pinpoint design problems.

I. Introduction

Evolutionary coupling, also known as logical coupling [1] or change coupling [2], reflects how files change together in the evolution of a software system [3], [4], [5]. Evolutionary coupling indicates both active structural dependencies and, more importantly, *implicit* dependencies between program entities.

Analyzing evolutionary coupling between files contributes to multiple software comprehension and maintenance tasks such as predicting source code changes [3], [6], [7], [8], locating architectural design problems [9], [10], [11], [12], and identifying cross-cutting concerns [13], [14], [15].

Thanks to the wide usage of modern version control systems (VCS), evolutionary coupling can be easily measured by leveraging co-change information of files embedded in commits [1]. There also have been handy tools and algorithms that identifies fine-grained changes from evolution history [16], [17], [18], which simplifies the process on history data.

However, leveraging co-change information for software maintenance tasks is still challenging because there has not been comprehensive understanding of how and why files co-change. Some studies have shown substantial evidence that structurally coupled class pairs are usually evolutionary coupled [19], while some others have shown that most structural coupled program entities do not co-evolve [20], [21]. This chaos also exists the other way round: co-evolving entities may or may not be structurally coupled [20], [22]. There is not yet a categorization of co-evolutions that help to clarify the relationship between evolutionary coupling and other types of coupling.

Considering the recurring nature of evolutionary coupling, researchers have been trying a pattern-based perspective [23], [24], [25]. For example, Mondal et al. [23] took an investigation on a particular pattern of continuously co-changed methods. Silva et al. [24] looked at the topology of the co-changed file clusters and summarized three co-change patterns. However, there has not been a single study that constructs an in-depth understanding of the underlying reasons of file co-change relationships with consideration of frequently recurring changes applied to files. For example, one observes that, when Class A add a method, Class B usually add an invocation to the method. Then, is this co-change (e.g., addition of a method in A and addition of a method invocation in B) common enough in general? Under what circumstances this type of co-change is normal, or indicate design problems? How can we use the co-change information to guide developers to understand the logic of the program, find possible defects, and analyze possible impacts of changes? The phenomenon of co-changes and the rationale behind need to be analyzed and understood to support engineering tasks.

To address these questions, we conduct an empirical study on six open source systems. We extract co-change information from 27,087 commits in the evolution history of the six open-source systems. We developed co-change pixelmaps base on the co-change information and further identified 6 types of dominating co-change relationships. Each of the co-change relationship types indicates a representative reason that drive frequent co-changes and bring insights into how files co-change. Temporal analysis on evolution history revealed that co-change relationships may change and evolve at different periods of time. We also reveal how files co-change as a cluster and find evidences to show chained, spread, and convergent change propagations, which may help developers in their maintenance tasks.

The contribution of this paper include: 1) 6 co-change relationship types that categorize evolution couplings and reveal the underlying reasons that drive co-changes; 2) 3 typical evolutionary trends of co-change relationships between files, i.e., enhancing/weakening the same relationship, switching relationships, and mixing up relationships, that depict the dynamics of evolutionary couplings between files; 3) 3

change propagation cases that are valuable to provide guidance and suggestions on observed co-change relationship types for developers' maintenance action; and 4) a commit-based fine-grained co-change data set on six open source systems.

The rest of the paper is structured as follows. Section II presents basic concepts of evolutionary coupling and clarifies definitions of key concepts in our work. Section III details our research methodology, including the research questions, subject systems, and data collection and analysis processes. Section IV reports the identified co-change relationship types and the results of temporal and spacial analysis of the co-change relationships. Section V presents in-depth reflections based on the reported co-change relationship types. Section VI discusses the threats to validity of the empirical study. Section VII summarizes related research work. Section VIII concludes our work and presents possible research directions.

II. Concepts and Definitions

A. Evolutionary Coupling

Evolutionary coupling describes the relationship between parts of software systems which are frequently changed together [2], [26]. Evolutionary coupling is commonly expressed by association rules in the form of A $\Rightarrow$ B where A and B are two sets of program entities. A $\Rightarrow$ B means that when A is changed, B is also changed [3], [20]. The strength of the coupling can be measured by *support count* and *confidence* [3]. *Support count* of rule A $\Rightarrow$ B describes how many times A and B are changed together while *confidence* describes how many co-changes of A and B have occurred in all changes of A. For example, if A has changed 11 times among which 10 were observed as a co-change with B, the support count $count(A \Rightarrow B)$ is 10; the confidence $conf(A \Rightarrow B)$ is 10/11. According to this definition, $count(A \Rightarrow B)$ is equal to $count(B \Rightarrow A)$ since both of them denote the occurrence frequency [3] of the co-changes between A and B.

Rules mined from evolution history are helpful to answer the questions like "what else were changed when these were changed", and further "what else need to be changed if these are changed".

Although there have been studies on co-evolution between software artifacts other than source code [27], [28], [29], we only consider source code files in the target systems.

B. Terminology

We collect source code change information from Git, a state-of-the-art version control system (VCS), where source code changes are organized in *commit*s. A commit may touch multiple files. Any pair of files that are touched by one commit are considered *co-change*d once. We define $count(A \cup B) = count(A \Rightarrow B) = count(B \Rightarrow A)$ as the count of commits by which two files[1] A and B are touched; and $conf(A \Rightarrow B)$ as the confidence of A $\Rightarrow$ B.

[1]It is possible that more than three files are changed together. However, any co-changes involving a group of files can be regarded as combinations of co-changes between any two files in the file group. Therefore, considering co-changes between two files is a reasonable start point of the co-change analysis.

Each file in a commit may have one or more *change*s, such as *add a method* `m1`, *add a method invocation to* `C.mx`, and *add an IF statement in method* `m2`. Each change is of a specific *change type* [16], [30], such as *addition of a method*, *insertion of a method invocation*, and *insertion of an IF statement*.

For each pair of co-changed files A and B, we define the support count $count(ct_1, ct_2)$ of a pair of change types (ct_1, ct_2) on the file pair as the number of commits in which ct_1 is a change type on File A and ct_2 is a change type on File B. We also define the support (percentage) $supp(ct_1, ct_2)$[2] of the co-occurrence of change types ct_1 and ct_2 as

$$supp(ct_1, ct_2) = \frac{count(ct_1, ct_2)}{count(A \cup B)}. \quad (1)$$

The support of a pair of change types is symmetric, i.e., $supp(ct_1, ct_2) = supp(ct_2, ct_1)$.

The *co-change relationship* between a co-changed file pair (A, B) describes the most frequent types of changes applied to files A and B when they are co-changed. The *co-change relationship* between files A and B is represented by a tuple (cts_A, cts_B), where cts_A is an ordered set (i.e., a sequence) of the most frequent change types on File A that appear in co-changes with File B, and cts_B is an ordered set of the most frequent change types on File B that appear in co-changes with File A. Whether a change type is "frequent enough in co-changes" is determined by thresholds of support count and support percentage which are set experimentally.

Figure 1 gives an example of co-change relationship. We consider three co-changing files A, B, and C, and observe 5 commits. A commit may touch all three files (as in Commit 1 and 3) or only some of them (as in Commit 2, 4, and 5). Change types on each file are denoted by various geometric icons if the file is changed in the commit. In this example, Files A and B co-changed three times, while Files B and C co-changed four times and Files A and C only co-changed twice. Whenever File A has "circle" and "diamond" changes, File B has a "trapezoid" change, and vice versa. The support values of ("circle", "trapezoid") and ("diamond", "trapezoid") are both 1.0, but the support of ("triangle", "pentagon") is $1/3$. By applying data mining approaches with certain parameter settings (e.g., support need to be larger than 0.5), one may claim that File A and File B have a co-change relationship as depicted on the upper-right part of the figure. Files B and C also has a co-change relationship as depicted on the lower-right part of the figure. However, Files A and C only co-changed twice out of total five commits, which is less likely that they have a co-change relationship at all.

[2]Zimmermann et al. [3] also mentioned *support*, which in our case would be defined as support count over the number of all commits, but did not use it in their work. We choose support because we use the number of all co-change commits instead of the number of all commits, which makes the percentage more meaningful. Moreover, we choose *support* rather than *confidence* because the number of all occurrences of a single change type (such as ct_1) could be very large and may bring statistical noise whereas counting all co-changes between A and B is much more specific to the co-changes between the two files and thus not likely to be disturbed by independent changes.

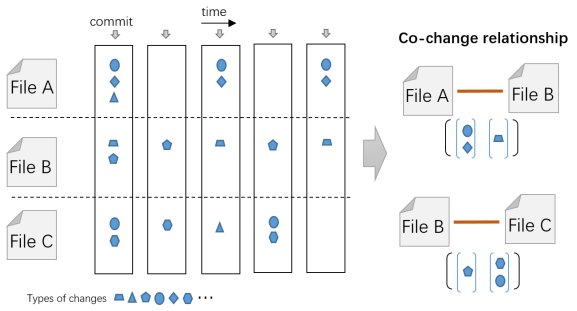


Fig. 1. An Example of File Co-change Relationship

In this example, we also notice that the change types in co-change relationship A-B and those in co-change relationship B-C are quite different in terms of the participating change types. We define *co-change relationship type* to describe the difference. So Files A and B are of co-change relation type ((“circle”, “diamond”),(“trapezoid”)) and Files B and C are of co-change relation type ((“pentagon”), (“hexagon”, “circle”)).

The motivation we define co-change relationship is as follows. 1) Files showing different types of dependencies have the tendency to co-change differently. Co-change relationship may characterize the difference of various dependencies. 2) Different co-change relationships may reveal possible reason of the evolutionary coupling, without the need to parse source code structure.

III. Methodology

A. Research goals and questions

This section discusses the research questions to be answered in this paper.

RQ1: Are there dominating co-change relationships among files that frequently change together? What are the typical co-change relationships that contribute to the co-changes? This RQ aims at revealing the underlying causes of high co-changes among files. We conjecture that there exist dominating co-change relationships between two files that cause them to frequently change together. In addition, we categorize the discovered relationships based on why they cause the co-changes. The contribution of this RQ is to fill the gap in current literature that an in-depth understanding of the fundamental causes of co-changes is lacking.

RQ2: Whether and how do the co-change relationships among files dynamically change with time? *“The only thing that does not change is change itself.”*. This very much applies to software systems. Therefore, we assume that the dominating co-change relationships (i.e., the underlying causes of the co-changes) among files will also change along with time, especially when major maintenance events, like refactoring, take place. This RQ investigates how the co-change relationships among files change with the evolution of the system. For example, if the developers refactor the code, it is possible that the co-change relationship among the refactored files will also change. To answer this question, we will conduct the temporal analysis of the co-changes among files.

TABLE I
Subject Systems

System	Period	#Cmts.	#Rels.	#Files	#LOC
Camel	03/2007-06/2017	28,881	113	6,697	1.17M
Cassandra	03/2009-06/2017	23,082	225	1,593	323K
CXF	04/2008-06/2017	13,180	123	3,371	713K
Hadoop	05/2009-06/2017	16,119	247	5,162	2.26M
HBase	04/2007-06/2017	13,577	551	2,218	1.25M
Wicket	09/2004-06/2017	19,989	244	1,822	246K

Note: Cmts.=Commits Rels.=Releases

RQ3: How can the co-change relationships be helpful in software maintenance? The ultimate goal of co-change relationship analysis to support maintenance. In this RQ, we will present three typical change clusters formed by the co-change relationships. Since each co-change relationship has very specific meaning. The change clusters based on co-change relationship can provide insights to developers in maintenance.

B. Subject Systems

Our empirical study uses six open-source Java projects of the Apache Software Foundation as the subject systems. They are: 1) Camel [31], an integration framework based on known enterprise integration patterns; 2) Cassandra [32], a distributed database system; 3) CXF [33], a services framework which helps users build and develop services using frontend programming APIs like JAX-WS and JAX-RS; 4) Hadoop [34], a framework that allows for the distributed processing of large data sets across clusters of computers; 5) HBase [35], the Hadoop database, a distributed, scalable, big data storage infrastructure; 6) Wicket [36], a component-oriented Java web application framework.

The information of these systems is shown in Table I, including the analyzed time period, number of commits, number of releases, number of files, and approximate size in LOC.

We choose these projects for the following reasons. First, these systems have been used as subject systems in previous studies on co-change analysis [12], [28], [37], [38], [39], thus we have a basis for comparison. Second, each of these projects has a long evolution history and is still under maintenance at the time of data collection. A long and active change history is suitable for analyzing source code co-changes. Third, these projects are of industry-scale and being collaboratively developed within the open-source community so they are more likely to convey common practices in software development, which would be valuable for other projects. Fourth, they have well maintained issue databases, which is useful for answering our research questions.

C. Data Preparation

The goal of data preparation is two-fold: 1) sanitize the commits to retrieve qualified changes; and 2) prepare co-changed file pairs.

1) Selecting commits: While VCS captures all commits made to a software project, but not all of them are relevant to the co-change analysis.We sanitize the commits based on three criteria and rationale.

First, we discard commits simply for merging prior changes to branches because these commits do not actually introduce

TABLE II
STATISTICS ON SELECTED COMMITS

System	#Commits touching 2-30 files (%)	#Files involved(%)
Camel	8,278 (29%)	1,361 (20.3%)
Cassandra	4,920 (21%)	822 (51.6%)
CXF	4,045 (31%)	608 (18.0%)
Hadoop	5,590 (35%)	921 (17.8%)
HBase	4,230 (31%)	778 (38.1%)
Wicket	4,663 (23%)	786 (43.1%)

new changes to the source code. Second, we discard commits that only contain non-source-code files (e.g., configuration files or read-me files) and test files. Although there is other work considering artefacts other than source code [27], [28], [40], our study focuses on co-changes among the functional source code only. Last, we only consider commits touching 2 to 30 source files. Commits touching only a single file do not introduce file co-changes so the change information in these commits does not add value to co-change analysis. Commits touching a massive number (>30) of files are usually considered as house-keeping the repository [3], [24], [41]. These commits are usually excluded from studies. The numbers of selected co-change commits are show in Table II. Files that do not show in these commits are not considered co-changed with other files. The percentage of files that ever co-changed with other files are also shown in the table. Note that Cassandra has the most co-changed files, involving more than half of all files. Hadoop and CXF have only less than 20% files involved.

2) Selecting co-change file pairs: Next, we calculated all the co-changed file paris from the selected commits. In a commit that changed n files, we generate C_n^2 pairs of files. However, not all file pairs are qualified for co-change analysis. Some of the file pairs may be changed in the same commits coincidentally. We filter out potential coincidences using two heuristics: 1) discard file pairs change in less than 10 commits, as such the co-change pairs all change together more than 10 times, and 2) discard the file pairs whose confidence values from both sides (i.e., both $conf(File1 \Rightarrow File2)$ and $conf(File2 \Rightarrow File1)$) are less than 0.5, as such when one file in a co-change file pair changes, there is more than 50% chance that the other file changes together.

Among all the involved files, we finally have 1,752 co-changed file pairs under consideration. Hadoop contributed 562 file pairs, the highest in the six systems; while CXF contributed 102 file pairs, the least of all six systems.

D. Co-change Relationship Analysis

Our co-change relationship analysis consists of five main steps. First, we extract fine grained file changes from all selected commits. Then, we pair up the change types according to the prepared co-change file pairs and generate co-change information. After that, we develop a visualization-based approach to summarize co-change relationship types to answer how files co-change. Fourth, we conduct temporal analysis to find how the co-change relationships between files evolve. Finally, we group the co-change file pairs that share files in common into co-change clusters in order to reveal potential benefits on maintenance tasks.

TABLE III
GROUPED STATEMENT TYPES REFINED FROM INTERNAL *Statement* TYPE IN CHANGEDISTILLER

GST	Internal Statement Types	Comments/Examples
1	Assert Statement	`assert(a==b);`
2	Assignment	`i = 0;`
3	Method Invocation	`cypher.encode();`
4	For Statement * Foreach Statement * While Statement * Do Statement *	Loop structure
5	Continue Statement	`continue;` in a loop
6	Break Statement	`break;` in a loop
7	Try Statement Catch Clause Throw Statement	Exception handling structure
8	Switch Statement Switch Case Clause	Switch-Case structure
9	If Statement *	`if` statement, including `else`
10	Return Statement	`return` from a method
10	Class Instance Creation	`new Person();`
12	Constructor Invocation	`inherited;`
13	Synchronized Statement	Starting with `synchronized`
14	Variable Declaration Statement	`int i;`
15	Labeled Statement	A label followed by a ":"
16	Postfix Expression Prefix Expression	`++i` or `i++`

*Internal statement types that are related to *Condition Expression Change*

1) Extract fine-grained source code changes: We consider 68 types of changes that have been applied on 16 types of source code statements. The 68 change types are derived from the 47 change types generated by ChangeDistiller [16], a widely-used general-purpose code change extraction tool working on object-oriented languages. ChangeDistiller originally produces types of fine grained source code changes such as *Statement Insert/Delete/Update/ParentChange/OrderingChange*, *Control Structure Condition Expression Change*, *Additional/Removed Functionality* [30]. However, we notice that these change types are not sufficient for our co-change analysis. We made two improvements:

First, ChangeDistiller differentiates five change operation types (i.e., insertion/deletion/update/ordering-change/parent-change) to a *Statement*, without distinguishing the type of the *Statement*. Capturing the different types of *Statements* is important for understanding how a co-change happens. In fact, ChangeDistiller's internal data structures already differentiate 23 types of statements, such as method invocation, assert statement, and assignment, but they are not reported. Therefore, we extract the internal statement types. Furthermore, we find that some statement types have similar semantics. For example, loops are recognized as four statement types: For/Foreach/While/Do. We group the statement types that have similar semantics and finally expand a single *Statement* type into 16 grouped statement types (GST), as listed in Table III.

Second, we notice that the original change type *Condition Expression Change* reported by ChangeDistiller is related to both Loop-structures (GST 4) and If-structure (GST 9) (5 internal statement types marked with a star (*) in Table III). Updating a statement on the condition expression can also

provide information about how the code is related to other changes. We then elaborate *Condition Expression Change* into 2 change types for GST 4 and 9, respectively.

Finally, we determine the set of change types that we are interested. We first discard the changes types only related to comments and inline documentations. Then we group part of the change types to avoid over-categorizing. For example, Ordering-change and Parent-change on any statements may both indicate moving the statement; Updating a Break statement may be similar to moving the statement; *Parameter Insert/Delete/Type-change* are all *method parameter updates*. The grouping allows for integrating essential changes and prevents data fragmentation. Due to limited space, we put the full list of the 68 change types online for your reference. The change types cover changes on 5 program entities, including field, method, control statement, non-control statement, and class. [3]

2) Generate detailed co-change information: Instead of just counting how many times a two files change together, we generate the detailed co-change information based on the fine-grained changes on each file.

We first record the detail of each co-change in the format (X, Y, ct_X, ct_Y, $count_{changetypes}$, $count_{files}$, $support_{changetypes}$), where X, Y are co-changed files; ct_X and ct_Y are change types on files X and Y, respectively; $count_{changetypes} = count(ct_X, ct_Y)$; $count_{files} = count(X \cup Y)$; and $support_{changetypes} = supp(ct_X, ct_Y)$. For example, a record `(NodeCmd.java, NodeProbe.java, METHOD_INVOCATION_INSERT, ADDITIONAL_FUNCTIONALITY, 65, 128, 0.5078)` means that `NodeCmd.java` and `NodeProbe.java` change together when the former has change type: `METHOD_INVOCATION_INSERT` and the latter has change type: `ADDITIONAL_FUNCTIONALITY`, and this pair of change types happened in 65 commits out of total 128 commits where the two files co-change, showing 50.78% support rate. Since two files often exhibit multiple co-change types, the support values of all co-change types between two files may depict how two files co-change. We then use the distribution of the support values of all co-change types for summarizing co-change relationships.

3) Categorizing Co-change Relationships: In order to make the categorization of the co-change relationship types, we developed a pixelmap view for co-change file pairs to visualize the co-changes between files. The pixelmap is similar to the one used by Zimmermann [42] except that the vertical and horizonal axis are the fine-grained change types of the two files. Each pixel in the pixelmap represents a pair of change types. The weight (value), shown in various darkness, represents the support of the change type pair between the two files. The darker a pixel is, the higher support value the pair of change types have.

To categorize the co-change relationships into types, we first consider the co-change pixelmaps that show similar mosaic layouts to be of the same co-change relationship type. Then we manually read the source code and check the dominant changes to the code and see whether they are driven by similar reasons. If all file pairs of the same co-change relationship type share a common underlying reason, we finally accept the co-change relationship type; otherwise, we reject the co-change relationship type and all related file pairs remain undecided.

The types of co-change relationships consist of one or more dominant change types on each files in the file pair. However, it is possible that files co-change for multiple reasons and thus the co-changes will show a mixture of different co-change relationships. In this case, we may not identify which co-change relationship type the two files have. In order to get a confident understanding of each co-change relationship, we do not count the two files in any relationship types but set them as open for further analysis.

4) Temporal Analysis: Co-change relationship between files may evolve along with time. To capture the evolution of co-change relationship among files, we split the history of each subject system into consecutive time windows. We applied two strategies to separate the windows. One is base on the count of commits: each time window contains the same number of commits. The other is based on the span of time: each time window covers the same length of time.

We identify the change of the mosaic patterns in different time windows, since the drastic change in the pixelmap indicates the change of the co-change relationship between two files. In particular, we are interested in cases where structural dependencies between two files remain stable, but the pixelmap displays drastic change. Such cases indicates implicit triggers instead of the explicit structural dependencies that cause the co-change relationship to happen.

5) Clustering Co-changed Files for Fine-grained Manual Analysis: In order to further understand how files co-change with other files, we connect co-changed file pairs on the shared file and get a co-changed file graph. In this graph, we look for the files that exhibit high connection with other files and then conduct fine-grained manual analysis on them in order to identify which program entities are changed most together with which other files. This helps to understand what changes have been made in the evolution of the file and may provide insight into how it will be changed in the future.

IV. Results

In this section, we report data analysis results regarding to the research questions.

A. Reasons of Evolutionary Couplings (RQ1): Co-change Relationship Types

We identified 6 co-change relationship types from the co-change data collected from the six open source systems. Table IV lists the types and corresponding number of co-change file pairs that belong to the type.

We found that 749 (43% of total 1,752) co-changed file pairs belong to one of the co-change relationship types. Others

[3]The full list of the change types is available temporarily in the "Additional materials". An open link will be provided here if the paper gets accepted.

do not meet our criteria that we set in our study. It is still possible that some co-change file pairs match one or more types during a different time period. In this case, the file pairs may not seem to belong to a single type when we consider the co-change history as a whole. We did not count these file pairs but leave them in temporal analysis and discussions in Subsection IV-B. In the following discussions, we use classes instead of files since our target is object-oriented and most files consist of one class.

1) Type1: Public-Field-Driven Relationship: This co-change relationship type describes when additional fields in one class (A) are added, many types of changes happen in the other class (B). This is usually because class B depends on public fields in class A. Figures 2(a) and 2(b) depict examples of this type. The two examples are similar except that the latter example exhibits more types of changes on class $DatabaseDescriptor$ (B), indicating that more complex couplings possibly exists in $DatabaseDescriptor$.

2) Type2: Interface-Driven Relationship: This co-change relationship type describes additional/removal/updating methods happened in one class (A), many types of changes happen on the other class (B). Usually A is an abstract class or an interface and class B inherits or implements A. Figure 2(c) shows an example, in which class $StorageService$ implements the interface $StorageServiceMBean$.

3) Type3: Method-Invocation-Targeted Relationship: This co-change relationship type features recurring and dominant changes in method invocations but few changes of other types in class A while class B suffers multiple types of changes at many places. This is usually caused by a contract change between A and B where A calls a method of B. The change of the contract is initiated inside B with many internal logic changes, resulting changing the signature of a method implementing the contract. For A, it is a small change on the invocation of B's method and A itself is stable compared to B. But for B, many changes have to be made before upgrading the contract to a new version. Figure 2(d) shows an example with $FormEncodingProvider$(A) and $FormUtils$(B). $FormUtils$ changes multiple places to add new method and change existing method. $FormEncodingProvider$ primarily update method invocation according to the changes in $FormUtils$. This indicates that the coupling is mainly on the change of the signature of the methods.

4) Type4: Benign Private-Field-Driven Relationship: This co-change relationship type describes that dominant changes in class A are method invocation, while many different types of changes happen on class B, most of which are additional fields and additional methods. Figure 2(e) shows an example with $CamelNamespaceHandler$(A) and $CamelContextFactoryBean$(B). In this case, field additions in $CamelContextFactoryBean$ are primarily private fields; the method additions are usually getters/setters.

We consider this relationship type benign because this is usually the case of A visiting B's fields through getter/setter methods. But developers still need to keep an eye how B's field to avoid a *data class* smell [43].

5) Type5: Private-Field-Driven Relationship Affecting Internal Logic: This co-change relationship type describes when many additional fields and methods were added in class A, many change types including method invocation changes and internal logic changes happen in class B. Figure 2(f) depicts an example of $S3Configuration$(A) and $S3Endpoint$(B). $S3Configuration$ primarily add methods and add fields, while $S3Endpoint$ insert new method invocations and insert *IF* statements correspondingly. $S3Endpoint$ encapsulates an object typed $S3Configuration$ which provides configuration for the Endpoint. There are additional *IF*-statement changes involved in $S3Endpoint$. Usually this indicate more complex logic has been introduced in the system, which may need developers attentions.

This is usually a bad case that need developers' attention.

6) Type6: Sibling-Evolution Relationship: This co-change relationship type features symmetric change types from either side of the two files. This is usually because both classes inherit the same super class or implement the same interface. Therefore, the two files may share certain assumptions from their common ancestor. Figure 2(g) shows an example with $Check$(A) and $Radio$(B) from Wicket. The two files are inherited from the same ancestor class $WebMarkupContainer$. They implement two UI components correspondingly. While this case seems to be ok regarding the design, we need to be alert of implicit dependencies indicated by the same changes. In some cases, co-changed parts may be pulled up to the ancestor class if possible.

There are still cases that have not been clearly categorized. Figure 2(h) and 2(i) are examples. These cases reflect complicated internal logic changes. The circled pixels reveal many method body co-changes (mainly on method invocations, variable declarations, and control statements) in both Classes A and B. Widely-spread changes of different change types show that there can be implicit dependencies between them or either class burdens from the complicated internal logic structure. Further maintenance attentions should be put on these files.

We also notice that the left strip of mosaics in Figure 2(i) (shown by an orange rectangle) appears similar to those of Type2. After code investigation, we believe it is because the class $StorageService$ is the bridge that connects $StorageServiceMBean$ and $NodeProbe$ and thus the co-changes between $StorageServices$ and $StorageServiceMBean$ can be seen in the co-changes between $StorageService$ and $NodeProbe$. It is a sign for developers to consider the co-changed files as a cluster, as will be discussed in Subsection IV-C.

Reasons behind co-changes. The co-change relationship types indicate that two files usually change together for the following four reasons.

1)*Increasing/Decreasing Existing Structural Dependency*: In this case, the same type of dependency is increased/decreased by adding/deleting fields/methods that are directly coupled. Types 1,2,5, as well as additional fields/methods in Type 4 belong to this case. Fields and

TABLE IV
STATISTICS ON CO-CHANGE RELATIONSHIP TYPES

ID	Co-change Relshp. Type	Reason of co-change	#FilePairs	#File Pairs in Subject Systems					
				Camel	Cassandra	CXF	Hadoop	HBase	Wicket
1	Public-Field-Driven	Class B depends on public fields in Class A.	41	7	6	3	13	10	2
2	Interface-Driven	Class B depends on public methods in Abstract Class/Interface A.	251	31	39	2	105	56	18
3	Method-Invocation-Targeted	Class B depends on private methods in Class A.	59	7	12	2	25	8	5
4	Benign Private-Field-Driven	Class B depends on private fields in Class A via getter/setter methods, but the dependency does not affect B's internal logic.	73	9	14	2	32	10	6
5	Private-Field-Driven Affecting Internal Logic	Class B's internal logic depends on private fields in Class A via getter/setter methods	79	19	10	5	25	15	5
6	Sibling-Evolution	Class A and Class B are inherited from the same ancestor class or implement the same interface.	246	34	42	32	29	61	48
Total			749	107	123	46	229	160	84

methods are added/deleted during the evolution because of different responsibilities assigned to the classes.

2)*Propagating Interface Changes via Existing Dependency*: In this case, two files co-change because of changes to method signatures via method call dependencies. Co-change relationship Type 3 and part of Type 4 support this case in that method invocations are often changed (in class A as in Figure 2(d) and 2(d)) due to the changes of method definitions in the other file (class B as in the figures).

3)*Propagating Internal Changes via Existing Dependency*: In this case, two files co-change due to internal changes in one file. The changes may propagate via method call dependencies. This case is supported by part of co-change relationship types 4 and 5 and many co-change cases that we have not assign a type to. Take Type 5 for example, class B suffers a lot of internal changes not because of directly calling methods/visiting fields of A but because of other logic changes, such as a new configuration setting. The new setting changed the business logic and class B have to change accordingly.

4)*Parallel Changes due to Implicit Dependency*: In this case, two files are not directly structurally coupled. Their co-changes are caused by certain implicit dependencies or dependencies to a third entity. Co-change relationship Type 6 supports this case. The classes may share the same ancestor class or have dependencies to the same interface/class. There are also other case such as both files depend on database schema or SQL statements, which is out of the scope of this paper.

Although the identified co-change relationship types may not be complete, they show a way to understand the evolutionary coupling.

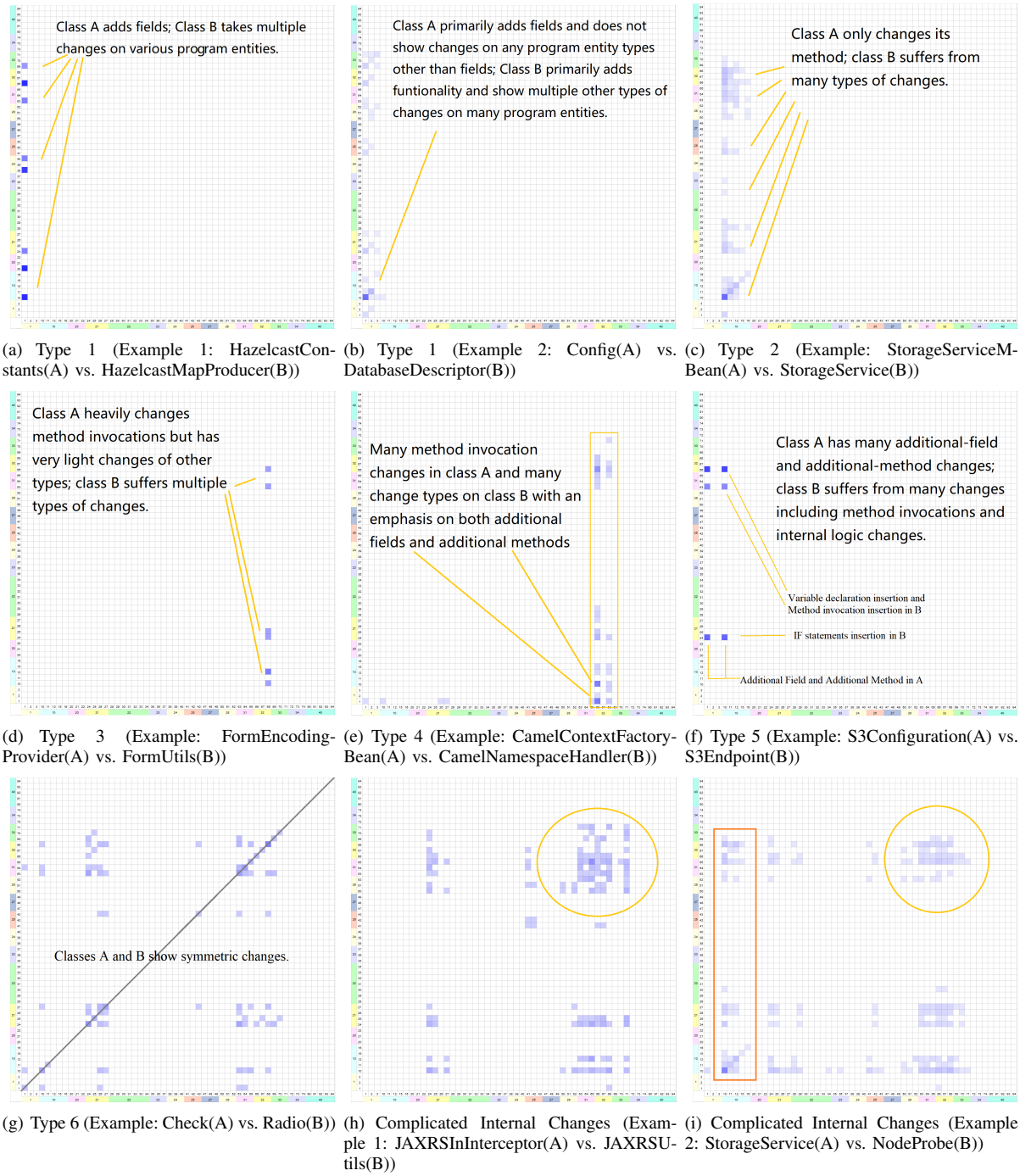
B. Evolution of Co-change Relationships (RQ2)

After analyzing the mined co-change relationship types, we investigate how the file pairs show a "*live*" coupling as they co-evolve. Based on temporal analysis, we find three basic scenarios that co-change relationships show themselves differently.

1) Sticking to the same co-change relationship type but the coupling can be strengthened or weakened: In this scenario, co-change relationship type remains stable but the coupling between the two co-changed files may increase or decrease. This is reflected in the change types on each of the files. For example, class $DatabaseDescriptor$ and $Config$ in Cassandra is a typical case. This example reflects the increasing field access dependencies from $DatabaseDescriptor$ to $Config$. When $Config$ adds a new field, $DatabaseDescriptor$ adds corresponding getter/setter methods for it. Along time, as more fields have been added, the coupling is strengthened. The increasing dependency may be an indicator of improper distribution of responsibilities among different classes. In the previous example, class $Config$ sends out a "*data class*" smell and attracts "*feature envy*" from $DatabaseDescriptor$.

2) Co-change relationships disappear: Co-change relationships between files may disappear. The co-evolution of classes $DatabaseDescriptor$ and $CassandraDaemon$ is an example. The history from April 2009 to March 2014 shows that, when $DatabaseDescriptor$ adds a field and a corresponding getter method, $CassandraDaemon$ adds a method call. However, after March 2014, although $DatabaseDescriptor$ still added some fields and getter methods, $CassandraDaemon$ no longer added corresponding method calls. This trend shows that some evolutionary dependencies are no longer active. There are two main causes behind the trend. One is that the corresponding requirements and design became stable and similar change operations no longer occur. The other is that refactoring or design modification changes the evolutional dependencies of the co-change group. In the latter case, some classes in a co-change group undergo similar changes but other classes in the group are no longer impacted by these changes.

3) Interleaving co-change relationship types: There are files that show a pure and benign co-change relationship at the early phase of the history but the design gradually deteriorated and showed a mixture of multiple co-change relationships. In extreme cases, multiple co-change relationship types may interleave, which results in a vague and non-categorized co-change relationship overall. Classes $StorageService$ and $NodeProbe$ in Cassandra, for example, show a mixture of Type2 and Type6. For some time, the two classes acted just like a Java interface and the implementation class; for some other time, they seemed like two classes derived from the same ancestor. The reason is that both classes are



(a) Type 1 (Example 1: HazelcastConstants(A) vs. HazelcastMapProducer(B))

(b) Type 1 (Example 2: Config(A) vs. DatabaseDescriptor(B))

(c) Type 2 (Example: StorageServiceMBean(A) vs. StorageService(B))

(d) Type 3 (Example: FormEncodingProvider(A) vs. FormUtils(B))

(e) Type 4 (Example: CamelContextFactoryBean(A) vs. CamelNamespaceHandler(B))

(f) Type 5 (Example: S3Configuration(A) vs. S3Endpoint(B))

(g) Type 6 (Example: Check(A) vs. Radio(B))

(h) Complicated Internal Changes (Example 1: JAXRSInInterceptor(A) vs. JAXRSUtils(B))

(i) Complicated Internal Changes (Example 2: StorageService(A) vs. NodeProbe(B))

Fig. 2. Examples of Co-change Relationship Types. In all figures, horizontal scale represents change types on file A while vertical scale represents change types on file B.

related to $StorageServiceMBean$: $NodeProbe$ integrates a $StorageServiceMBean$ and calls its methods whereas $StorageSerivce$ implements $StorageServiceMBean$.

Interleaving or mixture of multiple co-change relationship types may indicate possible structural or logical mixture. In fact, most co-changed file pairs that exhibit multiple co-change relationship types embody complicated logical couplings that cannot be easily understood. Analysis on a cluster of files may help to distinguish such mixtures.

C. Guidance for Maintenance Tasks (RQ3)

In order to answer the third research question, we focus on co-change file clusters and the files that show the most coupling with other files. Maintaining co-changed files consumes extra effort because changes can propagate via various co-change relationships. Therefore, we investigate how fine-grained change operations may be inferred and what potential actions are needed to improve the design. In other words, what actions are probably suitable for which part of the code?

1) Chained Propagation: Files in the co-change file cluster form a chain of change propagation, passing changes from one file to another. Take $StorageService$ from Cassandra for example. It is a class implementing Interface $StorageServiceMBean$, showing a Type2 co-change relationship. It is also implicitly referenced by class $NodeProbe$. Any changes in either $StorageService$ or $NodeProbe$ may transitively affect all files in the chain. This type of structural relationships reflects the propagation of change impact along a series of evolutionary dependencies.

2) Spread Propagation : This co-change file cluster consists a center class that propagate changes to other classes in the cluster. An example from Cassandra involves three classes: $ColumnFamilyStore$, $NodeCmd$, and $Memtable$. After analyzing the changes on source code, we found that some of the changes on $ColumnFamilyStore$ spread to $NodeCmd$, some others spread to $Memtable$, and the two kinds of changes are on different parts of class $ColumnFamilyStore$. Therefore, maintainers have to be careful to make changes on the appropriate part.

3) Convergent Changes: This co-change file cluster has a class that may be affected by other classes. Classes $NodeCmd$, $ColumnFamilyStore$ and $NodeProbe$ from Cassandra forms such a co-change file cluster. $NodeCmd$ have to follow the changes in $ColumnFamilyStore$ and $NodeProbe$ by adding method calls, for example, to the added methods. Therefore maintainers need to be aware that parts of $NodeCmd$ are impacted by the changes of various classes.

V. Discussion

In this section, we present in-depth reflections based on the results reported in the previous section.

A. Investigating Fine-grained Architectural Debt Formation

Architectural flaws, such as modularity violations and unstable interfaces, can form expensive architectural debts [12]. The existing tools only identifies the classes/files involved in these flaws [9], [10], [12]. But there is no systematic approach to understand the actual causes of the debt, nor its occurrence, accumulation, and disappearance over time.

In comparison, in this paper, we investigated *how* classes change together in a fine granularity, interpreted as the 6 types of co-change relationships. These types of relationships have the potential to provide insights to understand the formation and evolution of the design problems that contribute to increased maintenance effort in a project.

For example, we observed that the "unstable concept" could be in various forms.

1) The interface responsibilities are unstable and usually increasing. In other words, the interface is declaring more methods, and thus the concrete classes add the respective method implementations. For example, in Cassandra, $StorageServiceMBean$ and $StorageService$ of Type2 co-change relationship exhibits more and more responsibilities in the development history because the Java interface $StorageServiceMBean$ often adds method declarations.

2) The method signatures of the interface are unstable (e.g., adding/removing parameters) and thus cause the client classes to change accordingly. For example, in Cassandra, $ColumnFamilyStore$ frequently changes its method signatures, adding new or removing existing parameters. Whenever this happens, four client classes of $ColumnFamilyStore$: $MemTable$, $Table$, $SSTableWriter$, and $StorageService$, update the respective method calls(i.e., Add/Remove Parameter-Change Method Call).

3) The business logic within a method is unstable and thus causes the calling condition in the client to change accordingly. For example, in Cassandra, when the internal implementation of the *getKeyRange* method from class $StorageProxy$ is changed, the corresponding method call in $CassandraServer$ is moved into a *try/catch* block, showing complicated internal logic changes.

Similarly, for modularity violations, existing tools [9], [10], [12] only identify which classes are involved, without directly answering what are the "shared secrets" among classes that underly the violations. In this paper, we provided fine-grained analysis of the "secret" complying to which two classes frequently co-change without direct structural dependencies. For example, the "secret" between $AlterTableStatement$ and $CreateColumnFamilyStatement$ is that they both call the same methods from $ClusteringComparator$, $CFPropDefs$, and $CFMetaData$.

In summary, the co-change relationship analysis provides a systematic way to investigate the fine-grained design concepts that may cause architectural flaws. Co-change relationships also help to understand the over-time evolution of the identified design concepts.

B. Early Indicating Potential Architectural Debt

Existing architectural debt identification approach usually works in a retrospective manner, and require significant amount of revision history to identify a debt [12].

Co-change relationship analysis has the potential to provide early warning of a potential architectural debt. For example, we are able to identify $StorageServiceMBean$ as an unstable interface as soon as it co-changes with $StorageService$ on specific change types for the third time within a short period of time. This, again, is because we identify not only which classes are involved, but also the underlying "unstable" concept: $StorageServiceMBean$ is providing an increasing number of method signatures.

We have discovered many similar cases in our analysis. We believe that some co-change relationship types can be used as an early indicator for architectural debts.

C. Suggesting Refactoring for Cross-cutting Design Problems

In our study, we observed classes that are involved in multiple co-change relationships. Usually, these classes are called hotspot classes [10], which either (in some cases, both) dominantly propagate changes to or (and) submissively change with a large number of other classes. They usually

contain cross-cutting design problems that cause high-impact and expensive co-changes.

For example, co-change relationships involving $StorageService$ show co-changes with 15 classes. These 15 classes are the clients calling the methods implemented in $StorageService$. Actually, whenever $StorageService$ implements a new method, the other class adds a call to the newly added method. Some of the methods in $StorageService$ are implementing the declarations in $StorageServiceMBean$ while the majority others are irrelevant to $StorageServiceMBean$. $NodeProbe$ is the only class in the 15 classes that calls the methods declared in $StorageServiceMBean$. Meanwhile, the other 14 clients of $StorageService$ call the implemented methods that are irrelevant to $StorageServiceMBean$. Hence, we believe that $StorageService$ is a refactoring point to decouple the cross-cutting (involving 15 classes) design problem. A possible solution would be splitting $StorageService$ into multiple interfaces of separate responsibilities by providing independent methods for different clients.

We have observed multiple classes with cross-cutting design problems and we hypothesize that each of these classes presents refactoring opportunity to alleviate cross-cutting maintenance difficulties.

VI. Threats to Validity

Our results and findings in the empirical study suffer from several internal and external threats.

A major internal threat to validity lies in both the definition of change types and the categorization of the change types. Using various differencing tools may get a different set of change types and thus affects the result of co-change types. However, our selection of the differencing tool *ChangeDistiller* is a widely-used mature one and the results are widely accepted, which mitigates the threat to validity. Nevertheless, if other tools should be applied, our methodology still apply.

A major external threat to validity lies in the fact that we only considered six open source systems in our study and all of them were developed in Java. Our findings may not apply to much larger software systems, systems that are developed in other languages, or enterprise software systems. However, the six systems are carefully selected and may represent a large portion of the open source system. So our reported results and findings reflect certain generality.

VII. Related Work

Co-change analysis is widely used in multiple software engineering tasks, such as change impacts analysis [27], [44], [45], [46], defects prediction [47], [48], [2], [49], revealing architectural weakness and design problems [50], [39], [9], [51], [11], and identifying crosscutting concerns [13], [15], [14].

Understanding why and how parts of software co-change has always been a problem due to the complexity of software. Evolution history embeds rich information about change and numerous studies have been conducted. Yu [22] conducted a study on Linux evolution history and found a linear correlation between evolutionary coupling and structural coupling. Wong et al. [7] formalized evolutionary coupling as a stochastic process using a Markov chain model. Ajienka and Capiluppi [19] reported a large scale study on open source system and concluded that structural coupling usually causes evolutionary coupling but not all co-changed classes has structural dependencies. These empirical studies reveal complicate relations between evolutionary coupling and other types of software dependencies.

Various visualization tools have been proposed to help analyzing co-changes. D'Ambros et al. [52] presented a radar-like visualization to display logically coupled entities at various levels of granularity. Zimmermann et al. [42] used pixelmaps to depict the spread and strength of file-level co-changes. However, there has not been visualization work for refined change types that have happened on files.

There has been work that focuses on the developers' view. Beck et al. [39] investigate developers' opinions on the congruence between coupling concepts such as evolutionary coupling and the modularization of the system in practice. Silva et al. [24] discovered co-change patterns in terms of how co-changed program entities spread within a subsystem or across multiple systems and evaluated them with developers. Bavota et al. [53] empirically investigate how class coupling captured by structural, dynamic, semantic, and logical coupling measures aligns with developers' perception of coupling. Robbes et al. [54] took a look into the development process and analyzed the fine-grained changes made during a development session and extract changes from programmers' interactions with the development environment[55]. Although programmers help a lot in identifying the reason for evolutionary coupling, it is still difficult and effort-consuming to explain an specific evolutionary coupling.

There are also researches on the co-change analysis between code and build configuration [28], [40], [27] or between code and documents [29], which may construct our future work.

VIII. Conclusion

In this paper, we conducted an empirical study on six open source systems to understand evolutionary coupling by fine-grained co-change relationship analysis. We identified 6 co-change relationship types based on change operations extracted from co-change commits. The co-change relationship types are visualized in pixelmaps and can be used for understanding the reason behind the co-evolution of file pairs. The empirical study also shows that the co-change relationships themselves change along time and points out which classes are to be refactored and possible ways for refactoring. Our findings indicate that fine-grained co-change analysis has the potential to support more precise change impact analysis and early architectural flaw detection. Our future work includes developing code change and refactoring recommendation techniques based on the identified co-change relationship types.

REFERENCES

[1] H. C. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, 1998, pp. 190–197.

[2] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, 2009, pp. 135–144.

[3] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[4] T. Ball, J. Kim, A. A. Porter, and H. P. Siy, "If your version control system could talk..." 1997. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.87.137

[5] I. S. Wiese, R. T. Kuroda, R. Ré, G. A. Oliva, and M. A. Gerosa, "An empirical study of the relation between strong change coupling and defects using history and social metrics in the apache aries project," in *Open Source Systems: Adoption and Impact - 11th IFIP WG 2.13 International Conference, OSS 2015, Florence, Italy, May 16-17, 2015, Proceedings*, 2015, pp. 3–12.

[6] H. H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, 2010, pp. 119–128.

[7] S. Wong and Y. Cai, "Generalizing evolutionary coupling with stochastic dependencies," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, 2011, pp. 293–302.

[8] T. Rolfsnes, S. Di Alesio, R. Behjati, L. Moonen, and D. W. Binkley, "Generalizing the analysis of evolutionary coupling for software change impact analysis," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, 2016, pp. 201–212.

[9] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *International Conference on Software Engineering*, Honolulu, HI, USA, 2011, pp. 411–420.

[10] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *Working Conference on Software Architecture*, Montreal, QC, Canada, 2015, pp. 51–60.

[11] F. Palomba, G. Bavota, M. D. Penta, and R. Oliveto, "Detecting bad smells in source code using change history information," in *International Conference on Automated Software Engineering*, Silicon Valley, CA, USA, 2013, pp. 268–278.

[12] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *International Conference on Software Engineering*, Austin, TX, USA, 2016, pp. 488–498.

[13] G. Canfora, L. Cerulo, and M. D. Penta, "On the use of line co-change for identifying crosscutting concern code," in *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, 2006, pp. 213–222.

[14] S. Breu and T. Zimmermann, "Mining aspects from version history," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, 2006, pp. 221–230.

[15] B. Adams, Z. M. Jiang, and A. E. Hassan, "Identifying crosscutting concerns using historical code changes," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 305–314.

[16] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Software Eng.*, vol. 33, no. 11, pp. 725–743, 2007.

[17] M. Pawlik and N. Augsten, "RTED: A robust algorithm for the tree edit distance," *PVLDB*, vol. 5, no. 4, pp. 334–345, 2011.

[18] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324.

[19] N. Ajienka and A. Capiluppi, "Understanding the interplay between the logical and structural coupling of software classes," *Journal of Systems and Software*, vol. 134, pp. 120–137, 2017.

[20] G. A. Oliva and M. A. Gerosa, "On the interplay between structural and logical dependencies in open-source software," in *25th Brazilian Symposium on Software Engineering, SBES 2011, Sao Paulo, Brazil, September 28-30, 2011*, 2011, pp. 144–153.

[21] ——, "Experience report: How do structural dependencies influence change propagation? an empirical study," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015, pp. 250–260.

[22] L. Yu, "Understanding component co-evolution with a study on linux," *Empirical Software Engineering*, vol. 12, no. 2, pp. 123–141, 2007.

[23] M. Mondal, C. K. Roy, and K. A. Schneider, "Insight into a method co-change pattern to identify highly coupled methods: An empirical study," in *International Conference on Program Comprehension*, San Francisco, CA, USA, 2013, pp. 103–112.

[24] L. L. Silva, M. T. Valente, M. de A. Maia, and N. Anquetil, "Developers' perception of co-change patterns: An empirical study," in *International Conference on Software Maintenance and Evolution*, Bremen, Germany, 2015, pp. 21–30.

[25] Y. Wang, N. Meng, and H. Zhong, "An empirical study of multi-entity changes in real bug fixes," in *International Conference on Software Maintenance and Evolution*, 2018, pp. 316–327.

[26] S. Kirbas, T. Hall, and A. Sen, "Evolutionary coupling measurement: Making sense of the current chaos," *Sci. Comput. Program.*, vol. 135, pp. 4–19, 2017.

[27] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan, "Cross-project build co-change prediction," in *International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015*, Montreal, QC, Canada, 2015, pp. 311–320.

[28] C. Macho, S. McIntosh, and M. Pinzger, "Predicting build co-changes with source code change and commit categories," in *International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016*, Suita, Osaka, Japan, 2016, pp. 541–551.

[29] Z. Xing and E. Stroulia, "Analyzing the evolutionary history of the logical design of object-oriented software," *IEEE Trans. Software Eng.*, vol. 31, no. 10, pp. 850–868, 2005.

[30] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, 2006, pp. 35–45.

[31] "Camel," http://camel.apache.org/.

[32] "Cassandra," http://cassandra.apache.org/.

[33] "Cxf," http://cxf.apache.org/.

[34] "Hadoop," http://hadoop.apache.org/.

[35] "Hbase," http://hbase.apache.org/.

[36] "Wicket," http://wicket.apache.org/.

[37] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: a new metric for architectural maintenance complexity," in *International Conference on Software Engineering*, Austin, TX, USA, 2016, pp. 499–510.

[38] E. Kouroshfar, "Studying the effect of co-change dispersion on software quality," in *International Conference on Software Engineering*, San Francisco, CA, USA, 2013, pp. 1450–1452.

[39] F. Beck and S. Diehl, "On the congruence of modularity and code coupling," in *International Symposium on the Foundations of Software Engineering*, Szeged, Hungary, 2011, pp. 354–364.

[40] S. Mcintosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *International Conference on Software Maintenance and Evolution*, Victoria, BC, Canada, 2014, pp. 241–250.

[41] L. Moonen, S. Di Alesio, D. Binkley, and T. Rolfsnes, "Practical guidelines for change recommendation using association rule mining," in *International Conference on Automated Software Engineering*, Singapore, Singapore, 2016, pp. 732–743.

[42] T. Zimmermann, S. Diehl, and A. Zeller, "How history justifies system architecture (or not)," in *6th International Workshop on Principles of Software Evolution (IWPSE 2003), 1-2 September 2003, Helsinki, Finland*, 2003, pp. 73–83.

[43] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the design of existing programs," 1999.

[44] S. Hassaine, F. Boughanmi, Y.-G. Gueheneuc, S. Hamel, and G. Antoniol, "A seismology-inspired approach to study change propagation," in

International Conference on Software Maintenance, Williamsburg, VA, USA, 2011, pp. 53–62.

[45] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *International Conference on Software Engineering*, Zurich, Switzerland, 2012, pp. 430–440.

[46] H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Using topic model to suggest fine-grained source code changes," in *International Conference on Software Maintenance and Evolution*, Raleigh, NC, USA, 2016, pp. 200–210.

[47] M. Steff and B. Russo, "Co-evolution of logical couplings and commits for defect estimation," in *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, 2012, pp. 213–216.

[48] S. Kirbas, B. Caglayan, T. Hall, S. Counsell, D. Bowes, A. Sen, and A. Bener, "The relationship between evolutionary coupling and defects in large industrial software," *Journal of Software: Evolution and Process*, vol. 29, no. 4, 2017.

[49] S. Kirbas, A. Sen, B. Caglayan, A. Bener, and R. Mahmutogullari, "The effect of evolutionary coupling on software defects: an industrial case study on a legacy system," in *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, 2014, pp. 6:1–6:7.

[50] H. C. Gall, M. Jazayeri, and J. Krajewski, "CVS release history data for detecting logical couplings," in *6th International Workshop on Principles of Software Evolution (IWPSE 2003), 1-2 September 2003, Helsinki, Finland*, 2003, pp. 13–23.

[51] R. Schwanke, L. Xiao, and Y. Cai, "Measuring architecture quality by structure plus history analysis," in *International Conference on Software Engineering*, San Francisco, CA, USA, 2013, pp. 891–900.

[52] M. D'Ambros, M. Lanza, and M. Lungu, "Visualizing co-change information with the evolution radar," *Transactions on Software Engineering*, vol. 35, no. 5, pp. 720–735, 2009.

[53] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *International Conference on Software Engineering*, San Francisco, CA, USA, 2013, pp. 692–701.

[54] R. Robbes, D. Pollet, and M. Lanza, "Logical coupling based on fine-grained change information," in *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008*, 2008, pp. 42–46.

[55] F. Bantelay, M. B. Zanjani, and H. H. Kagdi, "Comparing and combining evolutionary couplings from interactions and commits," in *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, 2013, pp. 311–320.