Unveiling and Reasoning about Co-change Dependencies

Marcos César de Oliveira
Federal Budget Secretary
Ministry of Planning (MPOG)
Brasília, Brazil

Rodrigo Bonifácio
Guilherme N. Ramos
Department of Computer Science
University of Brasília
Brasília, Brazil

Márcio Ribeiro
Computing Institute
Federal University of Alagoas
Maceió, Brazil

Abstract

Product flexibility is one of the expected benefits of a modular design, and thus "it should be possible to make drastic changes to a module without changing others." Accordingly, the data available on version control systems might help software architects to reason about some quality attributes of the modular decomposition of a system. In this paper we investigate the impact of co-change dependencies into system stability, that is, the potential ripple effect that might occur during maintenance tasks. Here we use (a) Design Structure Matrices (DSMs) for visualizing dependencies motivated by assets' co-change and (b) two metrics for estimating system stability: *Propagation Cost of Changes* and *Clustered Cost of a Decomposition*. We conducted a comprehensive study about co-change dependencies and their effects on system stability, considering the change history of six open-source Java systems: Derby, Eclipse UI, Eclipse JDT, Hadoop, Geronimo, and Lucene; and one relevant financial systems of the Brazilian Government (SIOP). We evaluated two distinct situations: first considering only the static dependencies of each system and then considering both static and co-change dependencies of each system. There is a significant impact of the co-change dependencies on the stability measurements for Derby, Hadoop, Lucene, and SIOP. This result suggests that the modular decomposition of these systems does not resemble their change history. Accordingly, our findings provide empirical evidence that the common approach for reasoning about the modular decomposition, which often uses only static dependencies, hides important details about the costs of maintenance tasks.

MODULARITY'16, March 14–17, 2016, Málaga, Spain
© 2016 ACM. 978-1-4503-3995-7/16/03...$15.00
http://dx.doi.org/10.1145/2889443.2889450

Categories and Subject Descriptors D.1.5 [*Software*]: Programming Techniques - Object-Oriented Programming; D.2.2 [*Software*]: Software Engineering - Design Tools and Techniques; D.2.8 [*Software*]: Software Engineering - Metrics

Keywords Modularity, software change, version control systems, co-change dependencies, co-change clusters, Design Structure Matrices.

1. Introduction

In a seminal paper about the criteria to decompose systems into modules [19], David Parnas relates "module" as a work assignment unit and states the expected benefits of a modular design. These are currently well known and encourage, for each module, parallel design, support for independent reasoning, and the flexibility to change one without the need to change others. Though this notion of modularity relates to work assignment, instead of the decomposition of a software in terms of language constructs, such as Java packages, classes, and interfaces, it is expected that the design structure of a system should resemble the modularization in terms of work assignments.

The issue about how far goes the design structure's resemblance to the work assignments has been investigated before, particularly through the mining of evolutionary data available in version control systems (VCS). For instance, Murphy et al. reported that more than 90% of the changes committed to the Eclipse and Mozilla source code repositories involved changes to more than one file, which suggests a typical crosscutting pattern that might compromise software modularity [17]. Likewise, Zimmerman et al. recommend the use of fine-grained co-change dependencies, in terms of syntactical entities, instead of files or modules, to reason about software modularity [25]. In a more recent work, Silva et al. use software clustering techniques based on the co-change of coarse-grained software elements, such as packages and classes, to analyze software architectures [21]. As a result, they suggest that it might be worth to restructure the package decomposition according to the results of co-change analysis. The aforementioned works use the source code history to unveil dependencies that could not be inferred from static analysis, which are based on typical usage relation between software elements, such as method calls or attribute access. However, the existing works do not empirically investigate the effect of co-changes on the system stability, a quality attribute that indicates "the possible ripple effect" of a design during maintenance tasks [3, 24].

In this paper we reason about the stability of a system considering a special kind of dependency, named *hidden dependency*, that is motivated by a set of co-changes between two software elements, assuming there is no static dependency between them. In this way, we say that two entities are *co-change dependents* when they frequently change together, and, according to a criterion we will introduce later, this leads to a hidden dependency between them. Our definition of hidden dependency is similar to that presented by Gall et al. [9], and to the definition of *evolutionary coupling* as presented by Zimmermann et al. [25], except that hidden dependencies (as considered here) and static dependencies are mutually exclusive. Accordingly, we quantitatively investigate the impact of hidden dependencies on software architecture, using a novel approach that allows us to answer a general research question: *To what extent do the hidden dependencies induced by the co-change of components impact the system stability?* Answering this research question serves as a predictor about how appropriate is the software decomposition when accommodating its typical maintenance tasks. We also investigate whether co-change clusters might support future efforts of software reconstruction, as suggested in [21], and thus investigate a second research question: *With respect to system stability, what are the benefits of organizing a system according to a decomposition based on co-change clusters?* The contributions presented in this paper are the following.

- A novel approach for reasoning about the hidden dependencies induced by the co-change of software assets. Differently from previous works, our approach extends well known tools and metrics for reasoning about modularity.
- A comprehensive study about the impact of co-change dependencies into the system stability. This study considered six leading open-source Java systems and one proprietary Java system from the financial domain.
- A report that the typical analysis of *changing propagation*, which often considers only usage relations, might mislead software architects to underestimate the costs related to the change of a specific module.

In addition, our experience in this research revealed that changing the architecture of a system, according to a specific decomposition based on co-change clusters, might not improve the system stability. ***Road-map.*** Next section introduces the fundamental concepts related to our research. Section 3 details the methodology we use for building co-change clusters, for deriving DSMs from static and hidden dependencies, and for computing the stability metrics we use in this paper. Sections 4, 5, and 6 present the study settings, main results, and threats of our research. Finally, Section 7 relate our paper to existing works and Section 8 present some final considerations.

2. Background

Our work builds on the Baldwin and Clark general theory of modularity [5] in order to investigate the impact of hidden dependencies. Hereafter, a hidden dependency is a co-change dependency between two entities (A and B) given that a static dependency between A and B does not exist. Two entities are co-change dependent when they frequently change together and comply with a criterion detailed in Section 4, and a static dependency corresponds to a typical usage relation among source code entities, such as a method call.

The Baldwin and Clark theory considers modularity as being essential for promoting innovation in different domains. The evolution of the computer industry, for instance, may be explained through the *modular design of computers* in the nineteen sixties, when IBM launched the System/360 computer family [5]. The original theory relies on two main components: design structure matrices for reasoning about the architecture of a system, and six modular operators that describe how a system might evolve towards a modular design.

More recently, several research works investigated properties of software architectures using concepts related to this theory, in particular Design Structure Matrices [11, 12, 22]. However, the aforementioned works only consider typical usage relations to compute static dependencies, they do not consider the effect of hidden dependencies on the modular decomposition of a system. Complementing the Baldwin and Clark theory, MacCormack et al. [13] introduce specific metrics based on DSMs for reasoning about software modularity. Accordingly, we use the visual representation of DSMs, as well as the architectural metrics that can be computed from them, to answer our research questions introduced in the previous section.

2.1 Design Structure Matrix

A DSM is a technique for representing the modules and their dependencies in a system. This representation, which helps architects design, develop, and manage complex systems [8], has been considered as one of the most promising techniques for reasoning and measuring modularity [13]. A DSM is often represented as an $N \times N$ matrix, where each row and each column represent a module of a system [8].

Figure 1 shows an example of a DSM. The labels E_1 to E_4 identify the modules. Graphically, we use the '$\times$' symbol in a cell at row i and column j when the module i depends on the module j, and the '$-$' symbol when there is no dependency between them. To define the architecture of a system,

			1	2	3	4
E_1		1	·	–	–	–
$E_2 - E_3$	E_2	2	×	·	×	–
	E_3	3	×	×	·	–
E_4		4	–	–	×	·

Figure 1. Example of DSM.

			1	2	3	4
E_1		1	·	–	–	–
$E_2 - E_3$	E_2	2	×	·	×	–
	E_3	3	×	×	·	–
E_4		4	×	×	×	·

Figure 2. Example of a transitive closure.

it is usual to have a set of partitions of modules accompanying the DSM, as indicated by the thick borders in Figure 1. In Section 3 we discuss two hierarchical partition strategies used in this paper: (a) the structure of the source-code packages, and (b) the clusters based on co-change dependencies.

We use DSMs to analyze the design of existing software considering classes as modules, instead of considering the source file as a module [13]. In addition, we represent two kinds of dependencies: (a) static dependencies, and (b) hidden dependencies induced by the co-change of the software entities.

2.2 Metrics

To investigate the impact of hidden dependencies on system stability, we use two metrics based on the design structure of a system: *Propagation Cost* and *Clustered Cost* [13].

We chose these metrics due to several reasons. First, both metrics use an abstract notion of *dependency*, and thus we can compute these metrics considering a static dependency between two modules or an evolutive dependency based on co-change analysis. Second, we found a strong correlation between these metrics and other ones. For instance, the *Propagation Cost* metric presents a strong correlation with the *Net Option Value* metric, which also relies on an abstract notion of dependency. Moreover, the *Clustered Cost* has the distinctive characteristic of assigning a different cost to a dependency between two elements, depending on the location of these elements in the software decomposition. To the best of our knowledge, no other software stability metric takes the system decomposition into account. Finally, these metrics have been used in a number of research studies [1, 13, 16] and several full-fledged tools compute these metrics (e.g., Lattix's *System Stability* metric corresponds to the *Propagation Cost* metric used in our research).

There are other metrics for assessing system (in)stability but these are tied to a specific notion of dependency and, thus, are not suitable for assessing both the static and evolutive scenarios as are *Propagation Cost* and *Clustered Cost* metrics. For instance, the *Program Design Stability* metric [24] is only useful for a static situation since it is defined in terms of module invocation and the visible interfaces of the modules. The same is true for other metrics, such as *System Design Instability* [2, 10], *Reference for a Class* [7], *Coupling Between Objects* [7], and others. In the remaining of this section we present more details of the chosen metrics.

2.2.1 Propagation Cost

The *Propagation Cost* (PC) measures the degree of "coupling" of a design, captured by the degree to which a change to any single module causes a potential change to other modules in the system [13]. Formally, it is the ratio of the number of direct or indirect dependencies, and the maximum number of dependencies, and lower values indicate lower coupling and, thus, better stability. In this paper, a direct dependency is either a hidden dependency or a static dependency between modules A and B. An indirect dependency is a transitive dependency as follows: given the direct dependencies between A and B, and between B and another module C, we can say that there is an indirect dependency between A and C.

To know how many indirect dependencies exist, it is necessary to build a transitive closure of the original matrix which represents the DSM. Specifically, for the matrix $D = [d_{ij}]$, a transitive closure is another DSM $closure(D) = [d'_{ij}]$ where, $d'_{ij} = \times$ if a path between i and j in D exists. Such path exists if a exists a sequence

$$d_{k_0k_1}, d_{k_1k_2}, \cdots, d_{k_{n-1}k_n},$$

where each $d_{k_{l-1}k_l} = \times$, and $k_0 = i, k_n = j$.

Figure 2 shows the transitive closure computed from the DSM of Figure 1. Note that the cells d_{41} and d_{42}, in red, were marked '×' because E_4 depends on E_3, which depends on both E_1 and E_2. As a result, E_4 depends transitively on E_1 and E_2.

Given a DSM D, the PC metric is defined as:

$$PC = \frac{1}{|M|^2} \sum_{i,j=1}^{|M|} 1, \text{ if } d'_{i,j} = \times,$$

where $|M|$ is the total number of modules. The PC metric, which ranges from 0 to 1, counts all dependencies in the transitive closure since $[d'_{i,j}] = closure(D)$. For the example in Figure 2, this metric is $\frac{1}{4^2}(0 + 2 + 2 + 3) = 0.4375$.

2.2.2 Clustered Cost

To reason about the partitions strategies, either package based or co-change clustered based, we use the *Clustered*

Cost (CC) metric [13]. Differently from *Propagation Cost*, the *Clustered Cost* metric assigns different costs to each dependency according to the location of the corresponding modules. A smaller cost is assigned to a dependency when the two dependent modules are in the same partition. An even smaller cost is assigned to dependencies when one of the dependent modules correspond to a *vertical bus*, that is, a module with many dependent modules. Given a DSM D, a module j is a *vertical bus* if:

$$\left(DepRatio(j) = \frac{\sum_{i=1}^{|M|} 1, \text{ if } d_{i,j} = \times}{|M|} \right) > bus\ threshold$$

Then, given the set of modules M and the set of dependencies $D \subseteq M \times M$, the *Clustered Cost* of a DSM is the sum of the *Dependency Costs* (DC) of each $(M_i, M_j) \in D$, as defined below:

$$DC(i,j) = \begin{cases} 1, & \text{if } j \text{ is a vertical bus,} \\ n^{\lambda}, & \text{if } i, j \text{ in same partition,} \\ |M|^{\lambda}, & \text{otherwise.} \end{cases}$$

where n is the size of the partition, and λ is a user-defined parameter Here we use a $bus\ threshold = 0.1$ and $\lambda = 2$, as suggested by MacCormack et al. [13].

Therefore,

$$CC = \sum_{i,j=1}^{|M|} DC(i,j).$$

For the DSM D in Figure 1, we have:

$$DepRatio(1) = \tfrac{2}{4} = 0.5,\ DepRatio(2) = \tfrac{1}{4} = 0.25,$$
$$DepRatio(3) = \tfrac{2}{4} = 0.5,\ DepRatio(4) = \tfrac{0}{4} = 0.$$

Thus, if we assume a $bus\ threshold = 0.3$ and $\lambda = 2$, we have the following *dependency costs*:

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 1 & 0 \\ 3 & 1 & 4 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 \end{array}$$

leading to $CC = 8$.

The range of CC metric is from 0 to the maximum cost of a DSM, which is $|M|^{2+\lambda} = |M|^2 \cdot |M|^{\lambda}$, where $|M|^2$ is the maximum number of dependencies, and $|M|^{\lambda}$ is the maximum cost of a dependency. For this metric, the lower values indicates lower coupling w.r.t. the partitioning and, this, better stability.

3. Methodology

This section describes the methodology we use to investigate our research questions, as illustrated in Figure 3. In summary, in the first step we compute the co-change dependencies and co-change clusters using the data from version control systems. In the second step, we compute the static dependencies according to the usage relations between Java classes. Next, based on the co-change and static dependencies, we build the different DSMs (third step) that are necessary for reasoning about the impact of hidden dependencies on system stability and, finally, we compute the *Propagation Cost* and *Clustered Cost* metrics (fourth step). We now present more details on these steps.

3.1 Extracting Co-Change Clusters

Software clustering is a typical approach for discovering groups of artifacts based on their mutual dependencies. In general, before applying a clustering technique, it is first necessary to build a *Module Dependency Graph* (MDG), a directed graph that contains source code artifacts as vertexes and dependencies between them as edges. In our approach, Java classes are the vertexes of the MDGs; whereas the edges of the graph correspond to the co-change dependencies. Figure 4 illustrates the process for obtaining a co-change cluster.

To compute a MDG from the source code history, we iterate over the change sets from VCS and their respective artifacts (Figure 4a). A VCS repository contains the sequence of change sets applied to the software artifacts, and we consider that a *commit* is a change set that contains one or more artifacts. It is important to note that we followed some guidance about how to build MDGs from source code history, in order to enhance the quality of the clusters, so not all change sets are used when building the graph. First, we only consider commits explicitly related to at least one maintenance issue, registered in the *Issues Tracking System* (ITS) of the software system, which might indicate a certain degree of relevance of the commit [21]. In this case, a MDG (Figure 4b) is a co-change graph $G = (V, E)$, where V is a set of code artifacts, in our case Java classes, and E is a set of pairs $(V_i, V_j) \in V \times V$, such that there is an issue associated with one or more commits containing both V_i and V_j.

In addition, we follow the recommendation for excluding commits when the number of related entities exceeds a given threshold [26]. We also use both *support count* and *confidence* metrics for pruning co-change dependencies [25]. *Support count* is the number of issues in common between two artifacts and a MDG must only contain edges with a minimum *support count*. To improve the quality of our MDGs, the value assigned to the *support count* metric equals the weight of the corresponding edge [21]. In turn, the *confidence metric* measures the strength of a dependency between two artifacts, and is defined as:

$$confidence(V_i, V_j) = \frac{support\_count(V_i, V_j)}{support\_count(V_i, V_i)}$$

The term $support\_count(V_i, V_i)$ means the total of issues associated with V_i. Thus, *confidence* is the proportion of is-

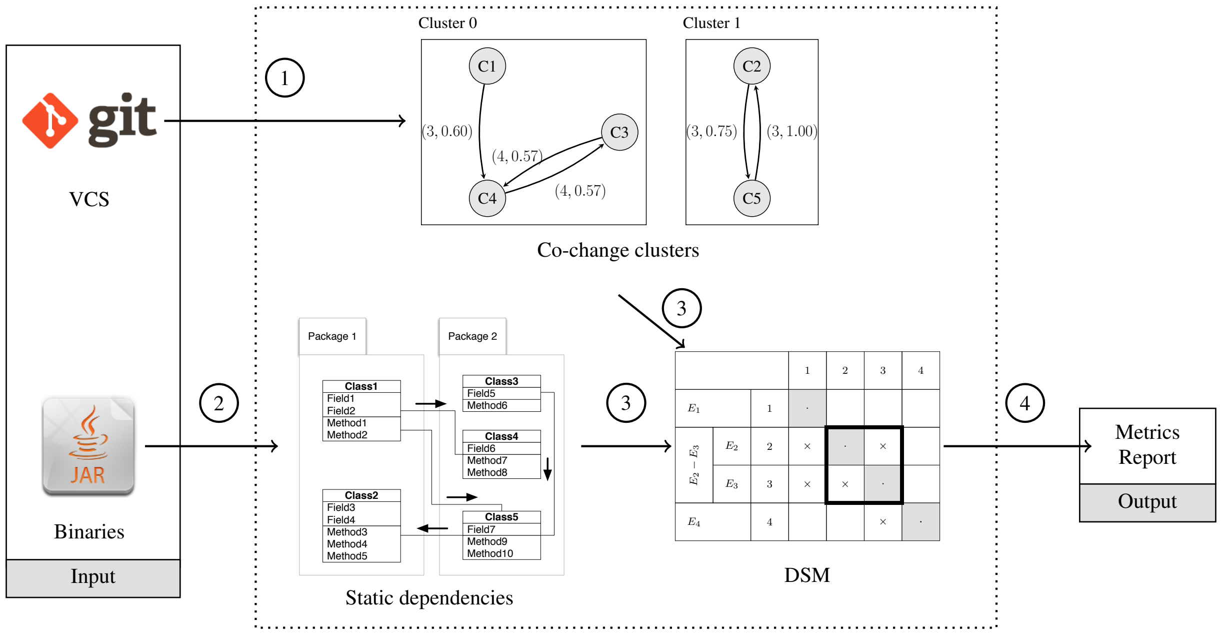
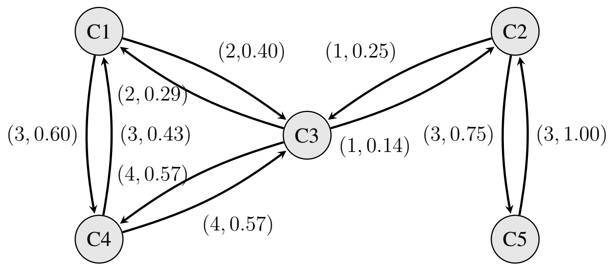


Figure 3. Metrics Extraction Process (numbered circles represent the steps).

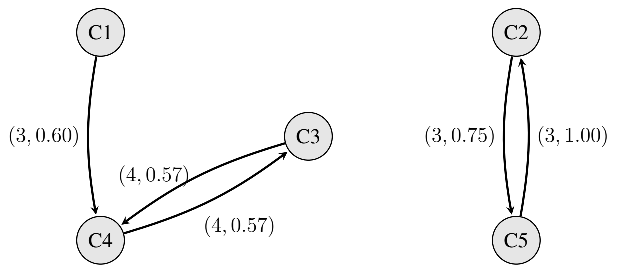
```
public class C1 { /* ... */ }
public class C2 { /* ... */ }
public class C3 { /* ... */ }
public class C4 { /* ... */ }
public class C5 { /* ... */ }
```

Commit	Description	Entities
028a98d	Issue #1	C1, C3
d8fd425	Issue #2	C1, C3
c90c352	Issue #3	C1, C4
ad3f78a	Issue #4	C1, C4
cd5e305	Issue #5	C1, C4
7de2d7b	Issue #6	C3, C2
83850f6	Issue #7	C4, C3
59561f2	Issue #8	C4, C3
b8e3afd	Issue #9	C4, C3
3bed650	Issue #10	C4, C3
5afa3bb	Issue #11	C5, C2
121192e	Issue #12	C5, C2
44b80e9	Issue #13	C5, C2

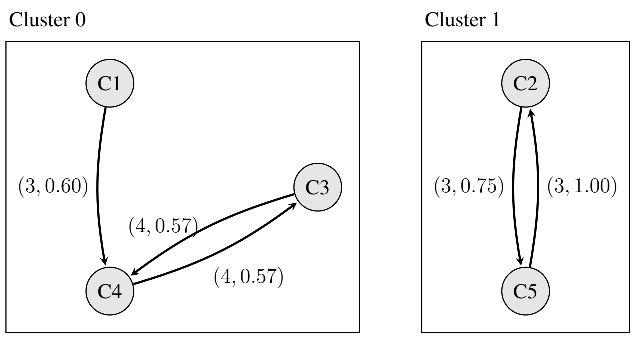
(a) Current source-code and respective commits log



(b) Co-change graph (MDG)



(c) Pruned co-change graph using *minimum support* equals 2 and *minimum confidence* equals 0.5



(d) Co-change clusters

Figure 4. Example of co-change cluster extraction (the edges' labels specify *support count* and *confidence* respectively).

sues where V_i and V_j participates in relation to the total of issues that V_i participates. Again, only edges with a minimum *confidence* will be used (see Figure 4c).

Given that criteria above, it is necessary to choose a suitable combination of thresholds for our goals, fortunately existing works present some advice on this issue. For instance, Beck and Diehl [6] state that the values for minimum *support* equals 1 and minimum *confidence* equals 0.4 produced the best results in their experiments. They also recommend to exclude commits with more than 50 related entities. Silva et al. [21] use a different set of thresholds, considering the minimum *support* equals 2.

Regarding automation, there are several algorithms, methods, and tools for clustering software artifacts. We chose `Bunch` because of its superior performance on clustering software [15]. `Bunch` applies a heuristic search algorithm based on random elements and increases the reliability of the results by carrying out the clustering process several times. More specifically, `Bunch` receives a MDG as input and outputs the same graph plus a set of clusters grouping similar artifacts (Figure 4d). In our analysis, we configured `Bunch` for *Agglomerative Clustering* with a *Hill Climbing* strategy, as in [6].

3.2 Extracting Static Dependencies

After computing the co-change dependencies and co-change clusters, we extract the static dependencies (second step of Figure 3) using the *Dependency Finder* tool[1], as Beck and Diehl [6]. This tool reads a set of binary class files and outputs the static dependencies based on usage relations between classes, which are then imported into our dataset.

3.3 Building DSMs

To answer the first research question, we create a DSM with static dependencies only and another comprising both static and hidden dependencies (third step of Figure 3). The modules of both are the *Java classes* of the subject system, and the partition strategy is based on the original *Java packages* in both DSMs. These two DSMs are then compared, using both visual representation and architectural metrics described, enabling us to reason about the impact of revealing the hidden dependencies.

To investigate if co-change clusters are a better partition strategy, we create two DSMs: one that uses the typical *Java packages* partition strategy; and another that uses the co-change clusters, each containing both static and hidden dependencies. This way, we can compare the *Clustered Cost* between DSMs and find which is the best partitioning strategy according to this metric.

3.4 Computing Metrics

We collect the architectural metrics using a custom script, which receives a DSM as input and outputs a set of values for the measures. Note that the original definition of the *Clustered Cost* metric uses as partitions a set of clusters that are built using an iterative search-based algorithm [13]. The intention in that case is to reflect in the metric only the amount and the patterns of dependencies in a DSM — not the quality of a particular partitioning. Differently, in this paper we measure the quality of a given partitioning based on co-change clusters, and thus we calculate the metric using the partitions based on packages and the co-change clusters.

[1] `http://depfind.sourceforge.net`

4. Study Settings

We selected seven Java projects from different domains as the subject systems for our study (Table 1 summarizes some basic metrics about them). All projects use either Git or SVN as version control systems, and an expressive number of commits are related to issues, ranging from 38% for SIOP to 98% for Hadoop. Six of the target systems are open-source projects; while SIOP is a fundamental financial system of the Brazilian Government, which the first author of this paper has contributed to. All systems have large code bases and at least four years of development history.

For each subject system, we consider the release version of the respective Java Archive (`JAR`) file as the end of the period of the change history— obtained from the VCSs. That is, we compute the dependencies between elements contained in the code base at a specific point in time, using two perspectives: static and evolutionary. The computation of static dependencies only uses the code as it appears in the `JAR` release date. In contrast, it is necessary to accumulate data from the change history to compute the co-change dependencies. In order to enable this study to be reproduced, we populate a relational database with the results of the first three steps of our methodology (Figure 3). Moreover, besides the use of existing tools (such as Dependency Finder and Bunch), we implemented several auxiliary scripts. All scripts and datasets are available on-line.[2]

Regarding the thresholds from the previous section, we carried out this research using the following configuration: *maximum number of entities per issue* of 50, *minimum support* of 2, following the recommendations detailed in [6, 21], and a *minimum confidence* of 0.5, a more conservative scenario than the value of 0.4 proposed by Beck and Diehl [6]). While we did not explore the parameter space regarding the thresholds in this study, further discussion on this subject can be found in [6, 21, 25, 26].

5. Results and Discussion

In this section we present the results from our investigation. We first introduce an exploratory data analysis that characterizes the target systems according to the scattering of both issues and commits throughout their fine-grained entities. We then answer the fundamental research questions of this study.

5.1 Exploratory Analysis of the Impact of Commits and Issues on Fine-grained Entities

Figure 5 presents a characterization of the subject systems according to the effect of issues and commits on fine-grained

[2] `https://goo.gl/kOzSKR` and `http://github.com/mcesar/mpca`

Table 1. Basic metrics about the target systems. #C is the number of classes and interfaces; #SD is the number of static dependencies between entities; and #I is the number of issues.

Name	Description	Version	Period	#C	#SD	#I
SIOP	Brazilian Planning and Budget System	1.26.0	2009/05/14-2014/05/15	4,542	126,192	2,949
Derby	Relational database	10.11.1.1	2004/08/11-2014/09/10	1,597	55,447	3,245
Hadoop	Large data sets processor	2.5.2	2009/05/19-2014/09/10	10,351	70,877	5,331
Eclipse UI	Eclipse main UI	4.5M2	2001/05/23-2014/09/08	7,443	75,354	9,510
Eclipse JDT	Eclipse Java core tools	4.5M2	2001/06/05-2014/10/21	1,954	87,356	5,002
Geronimo	Java EE application server	3.0.1	2006/08/24-2013/03/21	3,101	26,118	1,511
Lucene	Text search engine	4.9.1	2010/03/23-2014/10/27	4,097	20,236	3,272

software entities, namely, attributes, constructors, and methods. Note that, considering all systems, each commit affects, on average, 14.51 attributes, 5.12 constructors, and 17.29 methods. Besides that, Geronimo, Hadoop, and Lucene present a significant scattering of commits throughout fine-grained entities. In particular, each commit of these three target systems affects, on average, more than 20 methods. Regarding the impact of issues on the fine-grained entities, each issue of the subject systems affects, on average, 18.38 attributes, 6.48 constructors, and 27.90 methods. Once more, Geronimo, Hadoop, and Lucene present a high degree of scattering between issues and the fine-grained entities.

Considering all target systems, each issue requires, on average, 1.59 commits. This quantitatively supports a previous assumption that, in open-source projects, there is almost a one-to-one mapping between commits and work assignments [17]. We also consolidated this analysis considering the impact of commits and issues into coarse-grained entities, and we found that, on average, each commit affects 5.1 Java classes or interfaces; and each issue affects 8.81 Java classes or interfaces. These numbers show that the evolution of the target systems' source code, in terms of work assignments, is spread over several code entities. Thus, it might be worthwhile to envision the architecture of those systems considering their evolutionary history available in a version control system. We investigate this issue by empirically assessing the impact of the hidden dependencies motivated by the co-change of coarse-grained entities into the architecture of the systems.

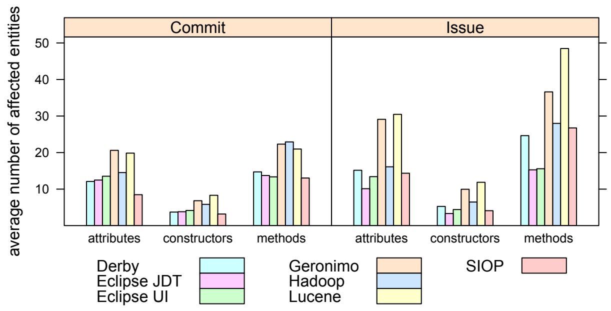
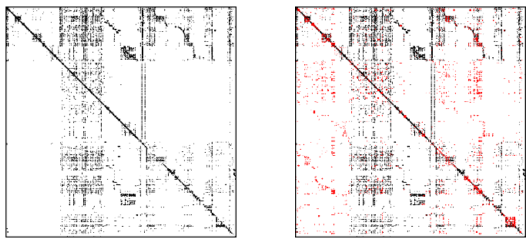


Figure 5. Characterization of the systems with respect to the impact of the commits and issues into the fine-grained entities.



(a) Static (b) Static and Co-Change

Figure 6. SIOP DSMs

5.2 On the Impact of Hidden Dependencies on System Stability

As discussed in Section 3, to answer our first research question we built two DSMs for each target system, using classes as modules and Java packages as the partitioning strategy. Figures 6 to 12 show the DSMs of the target systems. For all of them, the first sub-figure contains only static dependencies; the second contains both static (black) and hidden (red) dependencies.

Considering that in each DSM the rows and columns were sorted by the qualified class name, which is formed by package name plus class name, a modular design should concentrate most of the dependencies along the main diagonal and vertical buses. Comparing the resulting DSMs, different patterns emerge. For instance, Derby and SIOP present a high degree of scattering of both static and hidden dependencies, which might suggest an inadequate decomposition of these systems into modules. In the case of SIOP, we can also conjecture that this lack of modularity reflects the existing coupling within the organizational structure where SIOP has been developed [13].

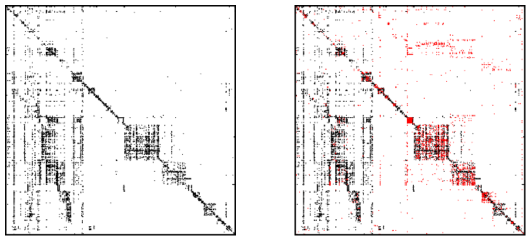

(a) Static (b) Static and Co-Change

Figure 7. Derby DSMs

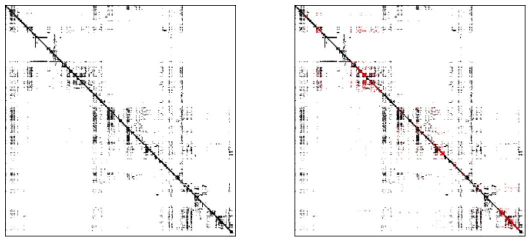

(a) Static (b) Static and Co-Change

Figure 8. Hadoop DSMs

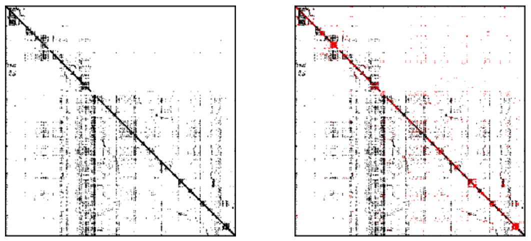

(a) Static (b) Static and Co-Change

Figure 9. Eclipse UI DSMs

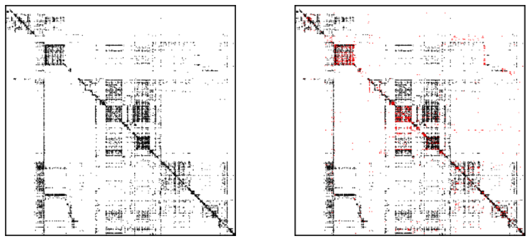

(a) Static (b) Static and Co-Change

Figure 10. Eclipse JDT DSMs

According to our observation that a modular design concentrates most of the dependencies along the main diagonal, we also conclude that Geronimo presents a reasonable decomposition. Indeed, after introducing the hidden dependencies, we can see a widespread presence of static dependencies in Geronimo (Figure 11b). For this system, the hidden

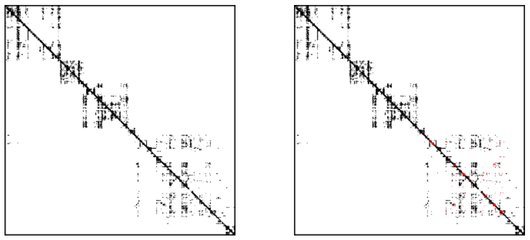

(a) Static (b) Static and Co-Change

Figure 11. Geronimo DSMs

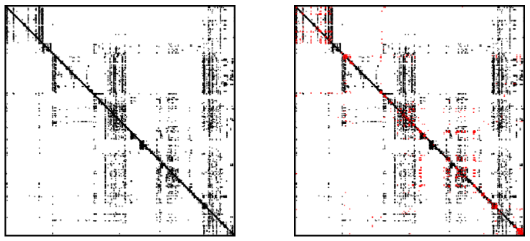

(a) Static (b) Static and Co-Change

Figure 12. Lucene DSMs

dependencies only increase the total number of dependencies by a factor of 0.05 (see the fourth column of Table 2). In contrast, the introduction of hidden dependencies for the other subject systems increases the total number of dependencies significantly (by a factor between 0.16 and 0.5).

Nevertheless, the distribution of hidden dependencies is not uniform, and the visual DSM representation reveals that most of those dependencies are along the main diagonal in the cases of Hadoop, Eclipse JDT, and Lucene. In addition, the hidden dependencies are prevalent in the cases of SIOP and Derby—in SIOP they increase the total number of dependencies by a factor of 0.5 (see the 4th column of Table 2). This result indicates that the decomposition of these systems does not accommodate their typical maintenance tasks, since **a significant number of hidden dependencies reveals that some classes which frequently change together do not present any usage relation between them, and, thus, some kind of semantic relation must exist between these components.**

We collected the *Propagation Cost* and *Clustered Cost* metrics from the DSMs in Figures 6 to 12. Table 2 presents the growth ratio of these metrics after introducing the hidden dependencies. A small growth denotes that entities that change together often present a static dependency or are confined in the same module. As a consequence, we consider that the lower the variation, the higher the stability of a system. Note that (a) most of the Geronimo static dependencies (Figures 11a and 11b) are within the main diagonal, and (b) Geronimo is more resilient to the introduction of the hidden dependencies—there is a small change on the *propaga-*

Table 2. Target System's Metrics Growth (%) after revealing hidden dependencies. 'D' means dependency

System	PC Growth	CC Growth	D Growth
SIOP	607	45	50
Derby	386	29	42
Hadoop	329	15	16
Eclipse UI	88	13	22
Eclipse JDT	87	15	20
Geronimo	22	5	5
Lucene	322	12	19

tion cost and *clustered cost* metrics of this system after considering them (see Table 2). Differently, Derby and SIOP present a high degree of scattering of the static dependencies, they are the less resilient subject systems with respect to the computed metrics. For instance, the introduction of the hidden dependencies increased the *clustered cost* by 29% and 45% in Derby and SIOP, respectively. Altogether, the results of Table 2 indicate a lack of stability for SIOP, Derby, Hadoop, and Lucene; and suggest that *changing propagation* cannot be completely explained by static dependencies, though most of the existing approaches use static dependencies for reasoning about it.

It is important to note that the impact on the metrics after introducing the hidden dependencies is higher on two subject systems: SIOP and Derby. Although Hadoop and Lucene also present a considerable growth on the *propagation cost* metric, they are more stable than SIOP and Derby when we consider the *clustered cost* metric. This is likely to occur because the hidden dependencies of Hadoop and Lucene often appear between classes of the same package. Moreover, we found a small correlation between the number of dependencies (*before* and *after* considering the hidden dependencies) and the metrics we use in this research (see Table 3). **As a consequence, the impact on these metrics might not be justified by the number of dependencies between software assets, but instead we can conclude that the corresponding lack of resilience is due to the modular organization of the systems.**

Accordingly, revealing the hidden dependencies caused by the co-change of system components brings new perspectives about the software stability, which we measured using both *Propagation Cost* and *Clustered Cost* metrics. For this reason, we consider that the hidden dependencies are relevant from an architectural perspective, and must be considered when deciding about restructuring a software. One question that arises at this point is: *what would happen to a system's stability if its decomposition was organized in terms of co-change clusters?* (instead of the original Java packages).

Table 3. Correlation between dependency count and growth of architectural metrics

Metric	PC Growth	CC Growth
D(S)	-21.23	-28.43
D(S, E)	-3.42	-7.28

Table 4. Clustered Costs growth (%) after restructuring the systems based on the co-change clusters. #P means Number of Original Packages. #C means the Number of Resulting Clusters

System	#P	#C	CC Growth
SIOP	142	88	−5
Derby	120	83	41
Hadoop	273	316	−2
Eclipse UI	217	386	−6
Eclipse JDT	521	407	39
Geronimo	263	109	9
Lucene	168	136	47

5.3 On the Benefits of Restructuring a System in Terms of Co-change Clusters

After analyzing the impact of hidden dependencies in the stability metrics of the target systems, we investigate the effect of a hypothetical re-modularization driven by co-change clusters. To this end, we built two additional DSMs for each project, both using Java classes as modules. The first DSM uses the original and typical Java package-based partitioning, while the second DSM uses a cluster-based partitioning strategy, which could be seen as a new arrangement of classes that change together.

In both cases (package and cluster-based partitioning) we only considered DSMs comprising static and co-change dependencies. Moreover, it is important to note that `Bunch` generates hierarchical clusters with different levels of details, and thus we decided to select the alternative that leads to a number of clusters closest to the number of packages of the original decomposition of the systems. Therefore, this approach conducts an automatic search for a new arrangement of classes based on the co-change clusters that tries to preserve the original number of packages of the system. Many other arrangements would also be possible, though we believe that this decision will produce, on the average, clusters having a similar number of classes than the original package of the systems. We then computed the *Clustered Cost* metric for both partitioning strategies— as discussed in Section 2, differently from the *Propagation Cost* metric, this metric is sensitive to the partitioning strategy.

Accordingly, Table 4 presents the resulting *Clustered Cost* measurements. It is possible to note that changing the organization of the system from Java packages to the se-

Table 5. Characterization of the selected partitions

System	Cluster Density		Number of Entities	
	Avg.	Std.	Avg.	Std.
SIOP	0.19	0.14	26.71	19.07
Derby	0.04	0.02	31.39	9.75
Hadoop	0.47	0.32	7.04	7.05
Eclipse UI	0.50	0.30	6.18	6.15
Eclipse JDT	0.44	0.29	5.34	4.12
Geronimo	0.70	0.30	4.18	3.34
Lucene	0.38	0.30	5.85	4.62

lected co-change partitioning strategy does not improve the stability of the systems in terms of the *Clustered Cost* metric.

Actually, such reorganization would increase the *Clustered Cost* of a system for Derby, Eclipse JDT, Geronimo, and Lucene. For these cases, there is a respective increasing of 41%, 39%, 9%, and 47% on the *Clustered Cost* metric. Instead, changing the organization of SIOP, Hadoop, and Eclipse UI according to the clustering partition would lead to a small reduction of the *Clustered Cost* metric. **Based on these results, organizing the system in terms of the selected co-change partitioning does not improve the system stability, since the costs of a change in the new decomposition have increased in most of the systems**. Note that an alternative partitioning strategy could have improved the system stability, though other properties of the system could have been compromised. For instance, a single partition strategy comprising all classes of the system would improve the system stability, though this leads to an unacceptable decomposition with small cohesion.

Table 5 presents some numbers that characterize the selected partitions for the subject systems, considering that *cluster density* is the ratio of the actual number of edges in a cluster to the maximum possible number of edges of a cluster. Note that the selected partition for SIOP and Derby leads to clusters that are more sparse than the clusters of the other systems. Moreover, we can realize that, on the average, there is a greater number of entities per cluster for the selected partition of these systems. Although we do not have enough empirical evidence, we conjecture that this result was motivated because both SIOP and Derby present the highest scattering of hidden dependencies. Nevertheless, we did not find any correlation between cluster density and the growth of the Cluster Cost metric reported in Table 4.

This result might appear frustrating, since it suggests that the co-change clusters do not help developers to find a better decomposition of a system. Nevertheless, we consider that co-change clusters offer a relevant perspective that complements the traditional package based view of the systems. Moreover, it is an open question whether a different strategy for selecting a partition based on co-change clusters would provide results that better conciliate system stability with other quality attributes of a system.

6. Threats to Validity

In this section we discuss some questions that might threat the validity of our work. As explained in the previous sections, our approach leads to co-change clusters that contain only a fraction of the whole set of code entities and dependencies. It is important to note that the same limitation has been reported by previous studies that use co-change clusters [6].

Two technical decisions contributed to reducing the number of entities considered in our analysis. First, we only considered commits explicitly related to issues, and we prune issues that require modifications in more than 50 Java classes or interfaces. This requirement reduced the number of commits considered in our analysis, ranging from 38% (in the case of SIOP) to 98% of all commits being considered (in the case of Hadoop). Therefore, every entity present in the co-change analysis must have been modified by a commit related to an issue. Second, we only considered co-change dependencies that fulfilled the specified thresholds for the *confidence* and *support* metrics. However, we set these thresholds according to the recommendations of previous studies [6, 21, 25].

Table 6 presents some numbers related to the completeness of our analysis. Note that 55.87% of the Geronimo commits are related to issues, although only 5.00% of the entities present a co-change dependency. As discussed in the previous section, this situation suggests that, for this case study, there is a small number of entities that frequently change together and that do not have a corresponding static dependency. In contrast, 38.31% of SIOP commits are related to issues, though 33.00% of its entities present a co-change dependency. Therefore, the reduced number of co-change entities might be better explained as a consequence of the modular decomposition of systems like Geronimo, Lucene, and Hadoop, which better accommodate the maintenance tasks in terms of localized changes.

Another issue is related to the metrics we chose to estimate stability, that is the *Propagation Cost* and *Clustered Cost* metrics. The reasons for choosing both metrics were given in Section 2, but it is important to emphasize that they have been used in previous research efforts [13, 16] to investigate the same property of other systems. One important question that might arise is that the *Propagation Cost* metric estimates the worst case for the cost of a change—that is, it considers that a change in a component will propagate to other parts of a system which either directly or indirectly depend on it. It is beyond the scope of this paper to investigate whether static (or transitive) dependencies imply co-changes between the entities of a system. Nevertheless, we mitigate the threats related to this issue by reasoning about both metrics considering two distinct situations, one for DSMs with static dependencies only and another with both static and co-change dependencies between components that do not have a corresponding static dependency. In this way, we reveal that

Table 6. Survival analysis of the entities considered in the research

Subject System	Total Number of Commits	Commits Related to Issues	Entities with Co-Change Dependencies
SIOP	12 061	38.31%	31.10%
Derby	6656	90.49%	55.00%
Hadoop	6864	98.75%	15.00%
Eclipse UI	21 263	61.98%	28.00%
Eclipse JDT	16 846	39.59%	59.00%
Geronimo	4142	55.87%	5.00%
Lucene	8854	71.56%	14.00%

propagation (which must reflect in co-evolution) can not be only explained by static dependencies. Note that, to support this conclusion, the actual cost of a change is not relevant.

Finally, we selected a relatively small set of *Java* projects for this study. While this can potentially limit the generalization of our results, we chose a broad range of applications, not limited to open-source ones, and all projects have a large code base with a long history of maintenance tasks. Thus, we expect that our findings provide a reasonable perspective which should be compatible with the analysis of other projects as well.

7. Related Work

This section relates our study to previous research works. First, our research relates to the use of information present in the evolution history to reason about quality properties of a software and to investigate a possible reorganization of a system. To the best of our knowledge, Gall et al. [9] were the first to explore the information from version history repositories. Similarly to us, they argue that this kind of information is relevant to reveal new perspectives about software dependencies. However, our work is more comprehensive, considering a broader number of systems and the entire evolution history of the subject systems in our analysis.

Zimmermann et al. [25] also investigate the evolution history as an alternative to find potential dependencies between program entities. Moreover, they introduce the support and confidence metrics we use to characterize a co-change dependency between two entities. Differently from our work that use coarse-grained entities (classes and interfaces), they use the evolution history present in version control systems to find dependencies between finer-grained entities and to reason about how an architecture accommodates the evolution of the systems [26]. The authors also argue that a lack of correspondence between the co-change dependencies and static dependencies might reveal weakness of the system architecture. Here we further investigate this issue using different tools (DSMs) and metrics (*Propagation Cost* and *Clustered Cost*), but focusing on the stability of the systems that we associate to the related costs of a change.

Some ideas for using software clustering as basis for system remodularization were first discussed by Wiggerts [23]. Anquetil and Lethbridge [4] compared different clustering algorithms to assist in software remodularization and Maqbool and Babri [14] assessed various hierarchical clustering algorithms for architecture recovering. More recently, Santos et al. [20] reported the use of semantic clustering to evaluate software remodularization. Their approach compares the values of conceptual metrics considering several versions of a system. Closely relate to our work, Silva et al. [21] proposed an approach to assess modularity using co-change clusters, which are further compared with the actual package decomposition. According to their work, mismatches between the co-change clusters and the package decomposition suggest new directions for restructuring the package hierarchy. This paper is different from the aforementioned works because our approach uses DSMs and metrics from the general modularity theory of Baldwin and Clark [5] for reasoning about the impact of hidden dependencies on system stability.

Therefore, our analysis rely on (a) the visual representation of DSMs and (b) the quantitative assessment of system stability using the *Propagation Cost* and *Clustered Cost* metrics. DSMs have been used to analyze the design properties of systems in a number of previous studies [11, 13, 18, 22]. Nevertheless, the aforementioned works only considered static dependencies between software entities when building the design structure matrices. Similarly, the metrics we use to quantify system stability have also been used before [13, 16], but without considering evolutionary dependencies.

8. Final Remarks

In this paper we presented an approach for reasoning about system stability, by considering hidden dependencies induced by the co-change of assets. Differently from existing works, our approach extends well known tools and metrics for assessing, both qualitative and quantitatively, the modular structure of a system. We investigated the impact of hidden dependencies using the change history of seven large and significant Java projects. Based on this, we presented quantitative evidence that the hidden dependencies of a system are relevant for understanding the costs related to a maintenance task. We found that, for some of the target systems, revealing hidden dependencies adds up to 50% of dependencies upon the usage relations between system components considered (Java classes and interfaces). That is, components do change together for reasons that are not explained by static dependencies, despite the fact that most existing approaches for reasoning about change propagation consider only such type of dependency. In addition, we investigate whether a specific decomposition based on co-change clusters would improve system stability. Such a decomposition was automatically generated by the clustering tool we use in our research and, at some extent, leads to a number of clusters close to the number of packages of the original system decomposition. As a conclusion, we found that this specific decomposition does not improve the system stability in terms of the related costs of changes. As a future work, we aim at investigating if a multi-objective approach could find a co-change based decomposition that better conciliates system stability with other quality attributes.

Acknowledgments

Alexander Serebrenik, Nicolas Anquetil, and the anonymous reviewers provided constructive comments on earlier versions of this paper.

References

[1] A. Almossawi. Visualizing ambiguity in an era of data abundance and very large software systems. In *New Challenges for Data Design*, pages 139–155. Springer, 2015.

[2] M. Alshayeb and W. Li. An empirical study of system design instability metric and design evolution in an agile software process. *Journal of Systems and Software*, 74(3):269 – 274, 2005.

[3] A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, and P. Avgeriou. The effect of gof design patterns on stability: A case study. *Software Engineering, IEEE Transactions on*, 41(8):781–802, Aug 2015.

[4] N. Anquetil and T. C. Lethbridge. Experiments with clustering as a software remodularization method. In *Reverse Engineering, Sixth Working Conference on*, pages 235–255. IEEE, 1999.

[5] C. Y. Baldwin and K. B. Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000.

[6] F. Beck and S. Diehl. On the impact of software evolution on software clustering. *Empirical Software Engineering*, 18(5):970–1004, 2013.

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

[8] S. D. Eppinger and T. R. Browning. *Design Structure Matrix Methods and Applications*. MIT Press, 2012.

[9] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, International Conference on*, pages 190–198. IEEE, 1998.

[10] W. Li, L. Etzkorn, C. Davis, and J. Talburt. An empirical study of object-oriented system evolution. *Information and Software Technology*, 42(6):373 – 381, 2000.

[11] C. V. Lopes and S. K. Bajracharya. An analysis of modularity in aspect oriented design. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 15–26. ACM, 2005.

[12] C. V. Lopes and S. K. Bajracharya. Assessing aspect modularizations using design structure matrix and net option value. In *Transactions on Aspect-Oriented Software Development I*, pages 1–35. Springer, 2006.

[13] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Manage. Sci.*, 52(7):1015–1030, 2006.

[14] O. Maqbool and H. A. Babri. Hierarchical clustering for software architecture recovery. *Software Engineering, IEEE Transactions on*, 33(11):759–780, 2007.

[15] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *Software Engineering, IEEE Transactions on*, 32(3):193–208, 2006.

[16] S. Muegge and R. Milev. Measuring modularity in open source code bases. *Open Source Business Resource*, 2009.

[17] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić. The emergent structure of development tasks. In *ECOOP Object-Oriented Programming*, pages 33–48. Springer, 2005.

[18] A. C. Neto, R. Bonifácio, M. Ribeiro, C. E. Pontual, P. Borba, and F. Castor. A design rule language for aspect-oriented programming. *Journal of Systems and Software*, 86(9):2333 – 2356, 2013.

[19] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[20] G. Santos, M. T. Valente, and N. Anquetil. Remodularization analysis using semantic clustering. In *Software Maintenance, Reengineering and Reverse Engineering, Conference on*, pages 224–233. IEEE, 2014.

[21] L. Silva, M. T. Valente, and M. Maia. Co-change clusters: Extraction and application on assessing software modularity. In *Transactions on Aspect-Oriented Software Development*, pages 1–37, 2015.

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

[23] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Reverse Engineering, Proceedings of the Fourth Working Conference on*, pages 33–43. IEEE, 1997.

[24] S. Yau and J. Collofello. Design stability measures for software maintenance. *Software Engineering, IEEE Transactions on*, SE-11(9):849–856, 1985.

[25] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Software Evolution, Sixth International Workshop on Principles of*, pages 73–83. IEEE, 2003.

[26] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, 2005.