Assessing Modularity using Co-Change Clusters

Luciana Lourdes Silva
Marco Tulio Valente
Department of Computer Science,
Federal University of Minas Gerais
{luciana.lourdes,mtov}@dcc.ufmg.br

Marcelo de A. Maia
Faculty of Computing,
Federal University of Uberlândia
marcmaia@facom.ufu.br

Abstract

The traditional modular structure defined by the package hierarchy suffers from the dominant decomposition problem and it is widely accepted that alternative forms of modularization are necessary to increase developer's productivity. In this paper, we propose an alternative form to understand and assess package modularity based on co-change clusters, which are highly inter-related classes considering co-change relations. We evaluate how co-change clusters relate to the package decomposition of three real-world systems. The results show that the projection of co-change clusters to packages follow different patterns in each system. Therefore, we claim that modular views based on co-change clusters can improve developers' understanding on how well-modularized are their systems, considering that modularity is the ability to confine changes and evolve components in parallel.

Categories and Subject Descriptors D.1.5 [*Software*]: Programming Techniques - Object-Oriented Programming; D.2.2 [*Software*]: Software Engineering - Design Tools and Techniques; D.2.7 [*Software*]: Software Engineering - Distribution, Maintenance, and Enhancement; D.2.8 [*Software*]: Software Engineering - Metrics; H.3.3 [*Information Storage and Retrieval*]: Information Search and Retrieval

Keywords Modularity, software change, version control systems, co-change graphs, co-change clusters, Chameleon graph partitioning algorithm

1. INTRODUCTION

Modularity is the key concept to embrace when designing complex software systems [3]. The central idea is that modules should hide important design decisions or decisions that are likely to change [27]. In this way, modularity contributes to improve productivity both during initial development and maintenance phases. Particularly, well-modularized systems are easier to maintain and evolve, because their modules can be understood and changed independently from each other.

For this reason, it is fundamental to consider modularity when assessing the internal quality of software systems [21, 24]. Typically, the standard approach to assess modularity is based on coupling and cohesion, calculated using the structural dependencies established between the modules of a system (coupling) and between the internal elements from each module (cohesion) [7, 35]. However, typical cohesion and coupling metrics measure a single dimension of the software implementation (the static-structural dimension). On the other hand, it is widely accepted that traditional modular structures and metrics suffer from the dominant decomposition problem and tend to hinder different facets that developers may be interested in [20, 30, 31]. Therefore, to improve current modularity views, it is important to investigate the impact of design decisions concerning modularity in other dimensions of a software system, as the evolutionary dimension.

Specifically, we propose a novel approach for assessing modularity, based on co-change graphs [5]. The approach is directly inspired by the common criteria used to decompose systems in modules, i.e., modules should confine implementation decisions that are likely to change together [27]. We first extract co-change graphs from the history of changes in software systems. In such graphs, the nodes are classes and the edges link classes that were modified together in the same commits. After that, co-change graphs are automatically processed to produce a new modular facet: co-change clusters, which abstract out common changes made to a system, as stored in version control platforms. Therefore, co-change clusters represent sets of classes that changed together in the past.

Our approach relies on distribution maps [12]—a well-known visualization technique—to reason about the projection of the extracted clusters in the traditional decomposition of a system in packages. We then rely on a set of metrics defined for distribution maps to characterize the extracted co-change clusters. Particularly, we describe some recurrent distribution patterns of co-change clusters, including patterns denoting well-modularized and crosscutting clusters. Moreover, we also evaluate the meaning of the obtained clusters using information retrieval techniques. The goal in this particular case is to understand how similar are the issues whose commits were clustered together. We used our approach to assess the modularity of three real-world systems (Geronimo, Lucene, and Eclipse JDT Core) and observed different patterns of co-change modularity in such systems.

Our main contributions are threefold. First, we propose a methodology for extracting co-change graphs and co-change clusters, including several pre and post-processing filters to avoid noise in the generated clusters. This methodology relies on a graph clustering algorithm designed for sparse graphs, as is the case of co-change graphs, that was capable to identify high density clusters. Second, we propose a methodology to contrast the co-change modularity with the standard package decomposition. This methodology includes metrics to detect both well-modularized and crosscutting co-change clusters. Third, we found that the generated clusters not only are dense in terms of co-changes, but they also have high

MODULARITY '14, April 22–26, 2014, Lugano, Switzerland.

http://dx.doi.org/10.1145/10.1145/2577080.2577086

similarity from the point of view of the meaning of the maintenance issues that originated the respective commits.

The paper is organized as follows. Section 2 presents the methodology to extract co-change graphs and co-change clusters from version control systems. Section 3 presents the results of co-change clustering, when applied to three systems. Section 4 analyzes the modularity of such systems under the light of co-change clusters. Section 5 analyzes the semantic similarity within the set of issues related to the extracted clusters. Section 6 discusses our results and presents threats to validity. Section 7 describes related work and finally Section 8 concludes the paper.

2. METHODOLOGY

This section presents the methodology we followed for retrieving co-change graphs and then for extracting the co-change clusters.

2.1 Extracting Co-Change Graphs

As proposed by Beyer et al. [5], a co-change graph is an abstraction for a version control system (VCS). Suppose a set of change transactions (commits) in a VCS, defined as $T = \{T_1, T_2, \ldots, T_n\}$, where each transaction T_i changes a set of classes. Conceptually, a co-change graph is an undirected graph $G = \{V, E\}$, where V is a set of classes and E is a set of edges. An edge (C_i, C_j) is defined between classes (vertices) C_i and C_j whenever there is a transaction T_k, such that $C_i, C_j \in T_k$, for $i \neq j$. Finally, each edge has a weight that represents the number of transactions changing the connected classes.

2.1.1 Pre-processing Tasks

When extracting co-change graphs, it is fundamental to preprocess the considered commits to filter out commits that may pollute the graph with noise. More specifically, we propose the following preprocessing tasks:

Removing commits not associated to maintenance issues: In early implementation stages, commits can denote partial implementations of programming tasks, since the system is under construction [25]. When such commits are performed multiple times, they generate noise in the edges' weights. For this reason, we consider just commits associated to maintenance issues. More specifically, we consider as maintenance issues those that are registered in an issue tracking system, such as Bugzilla, Jira, etc. Moreover, we only considered issues labeled as *bug correction*, *new feature*, or *improvement*. We followed the usual procedure to associate commits to maintenance issues: a commit is related to an issue when its textual description includes a substring that represents a valid Issue-ID in the system's bug tracking system [10, 34, 40].

Removing commits not changing classes: The co-changes considered by our approach are defined for classes. However, there are commits that only change artifacts like configuration files, documentation, script files, etc. Therefore, we discard such commits in order to only consider commits that change at least one class. Finally, we eliminate unit testing classes from commits because co-changes between functional classes and their respective testing classes are usually common and therefore may dominate the relations expressed in co-change graphs.

Merging commits related to the same maintenance issue: When there are multiple commits referring to the same Issue-ID, we merge all of them—including the changed classes—in a single commit. For instance, the issue `GERONIMO-3003` from Geronimo[1] is handled by four commits producing revisions `918360`, `798794`, `799023`, and `799037`. In this case, a single change set is generated for the four commits, including 13 classes. Therefore, in the co-change graph an edge is created for each pair of classes in this merged change set. In this way, we will have edges connecting classes modified in different commits (but referring to the same maintenance issue).

Removing commits associated to multiple maintenance issues: We remove commits that report changes related to more than one maintenance issue, which are usually called tangled code changes [15]. Basically, such commits are discarded because otherwise they would generate edges connecting classes modified to implement semantically unrelated maintenance tasks (which were included in the same commit just by convenience, for example). For instance, the revision `565397` of Geronimo includes changes to attend the following six issues: 3254, and 3394 to 3398.

Removing highly scattered commits: We remove commits representing highly scattered code changes, i.e., commits that modify a massive number of classes. Typically, such commits are associated to refactorings (like rename method) and other software quality improving tasks (like dead code removal), implementation of new features, or minor syntactical fixes (like changes to comment styles) [38]. For instance, the commit associated to revision `1355069` of Lucene changed 251 classes, located in 80 packages. In this revision, redundant throws clauses were refactored.

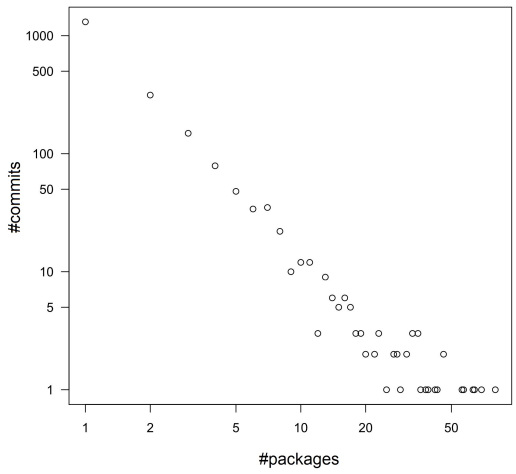

Figure 1. Packages changed by commits in the Lucene system

Recent research showed that scattering in commits tends to follow heavy-tailed distributions [38]. Therefore, the existence of massively scattering commits cannot be neglected. Particularly, such commits may have a major impact when considered in co-change graphs, due to the very large deviation between the number of classes changed by them and by the remaining commits in the system. Figure 1 illustrates this fact by showing a histogram with the number of packages changed by commits made to the Lucene system [2]. As we can observe, 1,310 commits (62%) changed classes in a single package. Despite this fact, the mean value of this distribution is 51.2, due to the existence of commits changing for example, more than 10 packages.

Considering that our goal is to model recurrent maintenance tasks and considering that highly scattered commits typically do not

[1] Geronimo is an application server, `http://geronimo.apache.org`

[2] An information retrieval library, http://lucene.apache.org.

present this characteristic, we decided to remove them during the co-change graph creation. For this purpose, we define that a package pkg is changed by a commit cmt if at least one of the classes modified by cmt are located in pkg. Using this definition, we ignore commits that change more than *MAX_SCATTERING* packages. In Section 3, we define and explain the values for thresholds in our method.

2.1.2 Post-processing Task

In co-change graphs, the edges' weights represent the number of commits changing the connected classes. However, co-change graphs typically have many edges with small weights, i.e., edges representing co-changes that happened very few times. Such co-changes are not relevant considering that our goal is to model recurrent maintenance tasks. For this reason, there is a post-processing phase after extracting a first co-change graph. In this phase, edges with weights less than a *MIN_WEIGHT* threshold are removed. In fact, this threshold is analogous to the *support* threshold used by co-change mining approaches based on association rules [39].

2.2 Extracting Co-Change Clusters

After extracting the co-change graphs, our goal is to retrieve sets of classes that frequently change together, which we call co-change clusters. We propose to extract co-change clusters automatically, using a graph clustering algorithm designed to handle sparse graphs, as is typically the case of co-change graphs [5]. More specifically, we decided to use the Chameleon clustering algorithm [18], which is an agglomerative and hierarchical clustering algorithm recommended to sparse graphs.

The algorithm operates in two phases:

- *First Phase:* a nearest-neighbor graph is extracted and a min-cut graph partitioning algorithm is used to partition the data items (classes in our case) into a pre-defined number of subclusters M. The number of clusters after this phase is M plus C, where C is the number of connected components in the graph.
- *Second Phase:* Chameleon combines such smaller clusters repeatedly. Clusters are combined to maximize the number of links within a cluster (internal similarity) and to minimize the number of links between clusters (external similarity).

2.2.1 Defining the Number of Clusters

A critical decision when applying Chameleon—and many other clustering algorithms—is to define the number of partitions M that should be created in the first phase of the algorithm. To define the "best value" for M we execute Chameleon multiple times, with different values of M, starting with a *M_INITIAL* value. Furthermore, in the subsequent executions, the previous tested value is decremented by a *M_DECREMENT* constant.

After each execution, we discard small clusters, as defined by a *MIN_CLUSTER_SZ* threshold. Considering that our goal is to extract groups of classes that may be used as alternative modular views, it is not reasonable to have clusters with only two or three classes. If we accept such small clusters, we may eventually generate a decomposition of the system with hundreds of clusters.

For each execution, the algorithm provides two important statistics to evaluate the quality of each cluster:

- $ESim$ - The average similarity of the classes of each cluster and the remaining classes (average *external similarity*). This value must tend to zero because minimizing inter-cluster connections is important to support modular reasoning.
- $ISim$ - The average similarity between the classes of each cluster (average *internal similarity*).

After pruning the small clusters, the following clustering quality function is applied to the remaining clusters:

$$coefficient(M) = \frac{1}{k} * \sum_{i=1}^{k} \frac{ISim_{C_i} - ESim_{C_i}}{max(ISim_{C_i}, ESim_{C_i})}$$

where k is the number of clusters after pruning small clusters.

The measure $coefficient(M)$ combines the concepts of cluster cohesion (tight co-change clusters) and cluster separation (highly separated co-change clusters). The $coefficients$ ranges from [-1; 1], where -1 indicates a very poor round and 1 an excellent round.

The selected M value is the one with the highest $coefficient(M)$. If the highest $coefficient(M)$ is the same for more than one value of M, then the highest $mean(ISim)$ is used as a tiebreaker. Clearly, internal similarity is relevant because maintainers are interested in clusters containing classes that frequently change together.

3. CO-CHANGE CLUSTERING RESULTS

In this section, we report the results we achieved after following the methodology described in Section 2 to extract co-change clusters for three systems.

3.1 Target Systems and Thresholds Selection

Table 1 describes the systems considered in our study, including information on their function, number of lines of code (LOC), number of packages (NOP), and number of classes (NOC), the number of commits extracted for each system, and the time frame used in this extraction. In the study, we considered the following thresholds:

- *MAX_SCATTERING* = 10 packages, i.e., we discard commits changing classes located in more than ten packages. We based on the hypothesis that large transactions typically correspond to noisy data, such as comments formatting and rename method [1, 39]. Excessive pruning is undesirable, so we adopted a conservative approach working at package level.
- *MIN_WEIGHT* = 2 co-changes, i.e., we discard edges connecting classes with fewer than two co-changes because an unitary weight does not reflect how often two classes usually change together [5].
- *M_INITIAL* = $NOC_G * 0.20$, i.e., the first phase of the clustering algorithm creates a number of partitions that is one-fifth of the number of classes in the co-change graph (NOC_G). The higher the M, the higher the final clusters' size because the second phase of the algorithm works by aggregating the partitions. In this case, the ISim tend to be lower because subgraphs that are not well connected are grouped in the same cluster. We made several experiments varying $M's$ value, and observed that whenever M is high, the clustering tend to have clusters of unbalanced size.
- *M_DECREMENT* = 1 class, i.e., after each clustering execution, we decrement the value of M by 1, meaning that no value for *M* is discarded from one iteration to another.
- *MIN_CLUSTER_SZ* = 4 classes, i.e., after each clustering execution, we remove clusters with less than 4 classes.

We defined the thresholds after some preliminary experiments with the target systems. We also based this selection on previous empirical studies reported in the literature. For example, Walker showed that only 5.93% of the patches in the Mozilla system change more than 11 files [38]. Therefore, we claim that commits changing more than 10 packages are in the last quantiles of the heavy-tailed distributions that normally characterize the degree of

Table 1. Target systems (size metrics and initial commits sample)

System	Description	Release	LOC	NOP	NOC	Commits	Period
Geronimo	Web application server	3.0	234,086	424	2,740	9,829	08/20/2003 - 06/04/2013 (9.75 years)
Lucene	Text search library	4.3	572,051	263	4,323	8,991	01/01/2003 - 07/06/2013 (10.5 years)
JDT Core	Eclipse Java infrastructure	3.7	249,471	74	1,829	24,315	08/15/2002 - 08/21/2013 (10 years)

scattering in commits. As another example, in the systems included in the Qualitas Corpus—a well-known dataset of Java programs—the packages on average have 12.24 classes [36, 37]. In our three target systems, the packages have on average 15.87 classes. Therefore, we claim that clusters with less than four classes can be characterized as small clusters.

3.2 Co-Change Graph Extraction

We start by characterizing the extracted co-change graphs. Table 2 shows the percentage of commits in our sample, after applying the preprocessing filters described in Section 2.1.1): removal of commits not associated to maintenance issues (Pre #1), removal of commits not changing classes and also testing classes (Pre #2), merging commits associated to the same maintenance issue (Pre #3), removal of commits denoting tangled code changes (Pre #4), and removal of highly scattering commits (Pre #5).

Table 2. Percentage of commits after each preprocessing filters

System	Pre #1	Pre #2	Pre #3	Pre #4	Pre #5
Geronimo	32.6	25.2	17.3	16.1	14.3
Lucene	39.2	34.6	23.6	23.3	22.4
JDT Core	38.4	32.8	21.7	20.30	20.1

As can be observed in Table 2, our initial sample for the Geronimo, Lucene, and JDT Core systems was reduced to 14.3%, 22.4%, and 20.1% of its original, respectively. The most significant reduction was due to the first preprocessing task. Basically, only 32.6%, 39.2%, and 38.4% of the commits in the Geronimo, Lucene, and JDT Core systems are explicitly associated to maintenance issues (as stored in the systems issue tracking platforms). There were also significant reductions after filtering out commits that do not change classes or that only change testing classes (preprocessing task #2) and after merging multiple commits due to the same maintenance issue (preprocessing task #3). Finally, a reduction affecting 3% of the Geronimo's commits and nearly 1% of the commits of the other systems was achieved by the last two preprocessing tasks.

After applying the preprocessing filters, we extracted a first co-change graph for each system. We then applied the post-processing filter defined in Section 2.1.2, to remove edges with unitary weights. Table 3 shows the number of vertices ($|V|$) and the number of edges ($|E|$) in the co-change graphs, before and after this post-processing task. The table also presents the graph's density (column D).

Table 3. Number of vertices ($|V|$), edges ($|E|$) and co-change graphs' density (D) before and after the post-processing filter

System	Post-Processing					
	Before			After		
	|V|	|E|	D	|V|	|E|	D
Geronimo	2,099	24,815	0.01	695	4,608	0.02
Lucene	2,679	63,075	0.02	1,353	18,784	0.02
JDT Core	1,201	75,006	0.01	823	25,144	0.04

By observing the results in Table 3, two conclusions can be drawn. First, co-change graphs are clearly sparse graphs, having density close to zero in the evaluated systems. This fact reinforces our choice to use Chameleon as the clustering algorithm, since this algorithm is particularly well-suited to handle sparse graphs [18]. Second, most edges in the initial co-change graphs have weight equal to one (more precisely, around 81%, 70%, and 66% of the edges for Geronimo, Lucene, and JTD Core graphs, respectively). Therefore, they connect classes that changed together in just one commit and for this reason were removed by the post-processing task. As result, the number of vertices after post-processing was reduced to 33% (Geronimo), 50% (Lucene), and 68.5% (JDT Core) of their initial value.

3.3 Co-Change Clustering

We executed the Chameleon graph clustering algorithm having as input the co-change graphs created for each system (after applying the pre-processing and post-processing filters).[3] Table 4 shows the value of M that generated the best clusters, according to the clustering selection criteria defined in Section 2.2.1. The table also reports the initial number of co-change clusters generated by Chameleon and the number of clusters after eliminating the small clusters, i.e., clusters with fewer than four classes, as defined by the *MIN_CLUSTER_SZ* threshold. Finally, the table shows the ratio between the final number of clusters and the number of packages in each system (column %NOP).

Table 4. Number of co-change clusters

System	M	# clusters		%NOP
		All	$\|V\| \geq 4$	
Geronimo	108	46	21	0.05
Lucene	68	98	49	0.19
JDT Core	100	35	24	0.32

For example, for the Geronimo system, we achieved the "best clusters" for $M = 108$, i.e., the co-change graph was initially partitioned into 108 clusters, in the first phase of the algorithm. In the second phase (agglomerative clustering), the initial clusters were successively merged, stopping with a configuration of 46 clusters. However, only 21 clusters have four or more classes ($|V| \geq 4$) and the others were discarded, since they represent "small modules", as defined in Section 3.1. We can also observe that the number of clusters is considerably smaller than the number of packages. Basically, this fact is an indication that the maintenance activity in the system is concentrated in few classes.

For the Lucene system, we achieved the best clusters for $M = 68$, since the number of clusters returned in the first phase is M plus the number of connected components.

Table 5 shows standard descriptive statistics measurements regarding the size of the extracted co-change clusters, in terms of number of classes. As we can observe, the extracted clusters have 8.8 ± 4.7 classes, 11.7 ± 7.0 classes, and 14 ± 10.4 classes (average $\pm$ standard deviation) in the Geronimo, Lucene, and JDT Core systems, respectively. Moreover, the biggest cluster has a considerable number of classes: 20 classes (Geronimo), 27 classes (Lucene), and 43 classes (JDT Core).

[3] To execute Chameleon, we relied on the CLUTO clustering package, `http://glaros.dtc.umn.edu/gkhome/cluto/cluto/overview`.

Table 5. Co-change clusters size (in number of classes)

System	Cluster size			
	Min	Max	Avg	Std
Geronimo	4	20	8.8	4.7
Lucene	4	27	11.7	7.0
JDT Core	4	43	14	10.4

Table 6 presents standard descriptive statistics measurements regarding the density of the extracted co-change clusters. The clusters have a density of 0.80 ± 0.24 (Geronimo), 0.68 ± 0.25 (Lucene), and 0.54 ± 0.29 (JDT Core). The median density is 0.90 (Geronimo), 0.71 (Lucene), and 0.49 (JDT Core). Therefore, although co-change graphs are heavily sparse graphs, the results in Table 6 show they have dense subgraphs with a considerable size (at least four classes). Density is a central property in co-change clusters, because it assures that there is a high probability of co-changes between each pair of classes in the cluster. In other words, high-density co-change clusters can be viewed as related program units, at least under evolutionary terms.

Table 6. Co-change clusters density

System	Cluster density				
	Min	Max	Avg	Std	Median
Geronimo	0.31	1.0	0.80	0.24	0.90
Lucene	0.17	1.0	0.68	0.25	0.71
JDT Core	0.18	1.0	0.54	0.29	0.49

Table 7 presents standard descriptive statistics measurements regarding the average weight of the edges in the extracted co-change clusters. For a given co-change cluster, we define this average as the sum of the weights of all edges divided by the number of edges in the cluster. We can observe that the median edges' weight is not high, being slightly greater than two in Geronimo and Lucene, and four in the JDT Core. However, it is important to mention that after applying the preprocessing filters we only considered a small sample of the initial commits to create the co-change graphs (14.3% of the commits in Geronimo, 22.4% of the commits in Lucene, and 20.1% in JDT Core).

Table 7. Average edges' weight

System	Cluster average edges weight				
	Min	Max	Avg	Std	Median
Geronimo	2	5.5	2.4	0.8	2.1
Lucene	2	7.1	2.7	1.0	2.4
JDT Core	2	7.6	4.3	1.5	3.8

4. MODULARITY ANALYSIS

In this section, we investigate the application of co-change clusters to assess the quality of a system's package decomposition. Particularly, we investigate the distribution of the co-change clusters over the package structure. For this purpose, we rely on distribution maps [12], which are typically used to compare two partitions P and Q of the entities from a system S. In our case, the entities are classes, the partition P is the package structure, and Q is composed by the co-change clusters. Moreover, entities (classes) are represented as small squares and the partition P (package structure) groups such squares into large rectangles (packages). In the package structure, we only consider classes that are members of co-change clusters, in order to improve the maps visualization. Finally, partition Q (co-change clusters) is used to color the classes (all classes in a cluster have the same color).

In addition to visualization, distribution maps can be used to quantify the *focus* of a given cluster q in relation to the partition P (package structure), as follows:

$$focus(q, P) = \sum_{p_i \in P} touch(q, p_i) * touch(p_i, q)$$

where

$$touch(p, q) = \frac{|p \cap q|}{|q|}$$

In this definition, $touch(q, p_i)$ is the number of classes of cluster q located in the package p_i divided by the number of classes in p_i that are included in at least a co-change cluster. Similarly, $touch(p_i, q)$ is the number of classes in p_i included in the cluster q divided by the number of classes in q. Focus ranges between 0 and 1, where the value one means that the cluster q dominates the packages that it touches, i.e., it is well-encapsulated in such packages. On the other hand, when co-change clusters crosscut many packages, but touching few classes in each of them, their focus is low. There is also a second metric that measures how *spread* is a cluster q in P, i.e., the number of packages touched by q.

Tables 8 and 9 show the standard descriptive statistics measurements regarding respectively the focus and spread of the co-change clusters. We can observe that the co-change clusters in Geronimo have a higher focus than in Lucene and JDT Core. For example, the median focus in Geronimo is 1.00, against 0.55 and 0.30 in Lucene and JDT Core, respectively. Regarding spread, the values in both systems are similar, on average the spread is 3.50 (Geronimo), 3.35 (Lucene), and 3.83 (JDT Core). Figure 2 shows a scatterplot with the values of focus (horizontal axis) and spread (vertical axis) for each co-change cluster. In Geronimo, we can see that there is a concentration of clusters with high focus. On the other hand, for Lucene, the clusters are much more dispersed along the two axis. Eclipse JDT tends to have lower focus, but also lower spread, even if the maximum spread is the largest of the three systems.

Table 8. Focus

System	Focus				
	Min	Max	Avg	Std	Median
Geronimo	0.50	1.00	0.93	0.12	1.00
Lucene	0.06	1.00	0.57	0.30	0.55
JDT Core	0.07	1.00	0.36	0.26	0.30

Table 9. Spread

System	Spread					
	Min	Max	Avg	Std	Median	Mode
Geronimo	1	8	3.50	2.10	3	1
Lucene	1	8	3.35	1.90	3	3
JDT Core	1	10	3.83	2.60	3	1

In the following sections, we analyze examples of well-encapsulated and crosscutting clusters, using distribution maps,[4] in the Geronimo system (Section 4.1), in the Lucene System (Section 4.2), and the Eclipse JDT (Section 4.3). Section 4.1 emphasizes well-encapsulated clusters, since they are common in Geronimo. On the other hand, Section 4.2 emphasizes crosscutting concerns, which are most common in Lucene. Section 4.3 on Eclipse JDT reports an analysis on both types of clusters.

[4] To extract and visualize distribution maps, we used the Topic Viewer tool [33], available at `https://code.google.com/p/topic-viewer`).

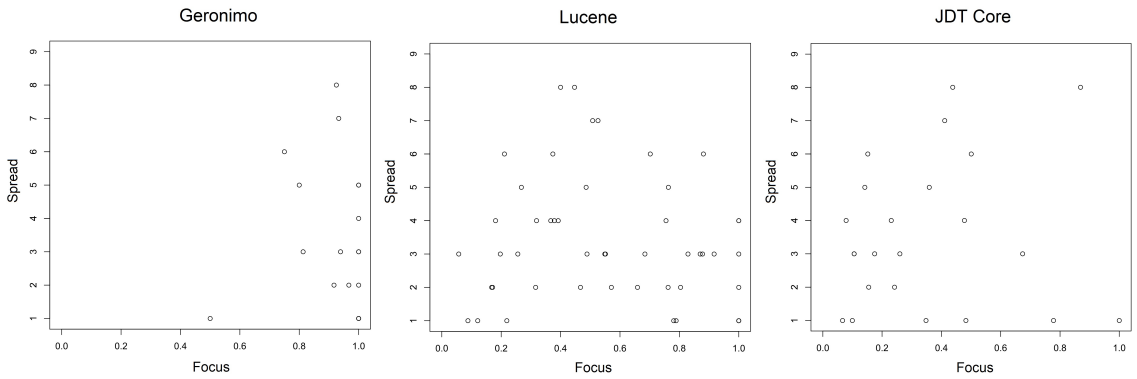

Figure 2. Focus versus Spread

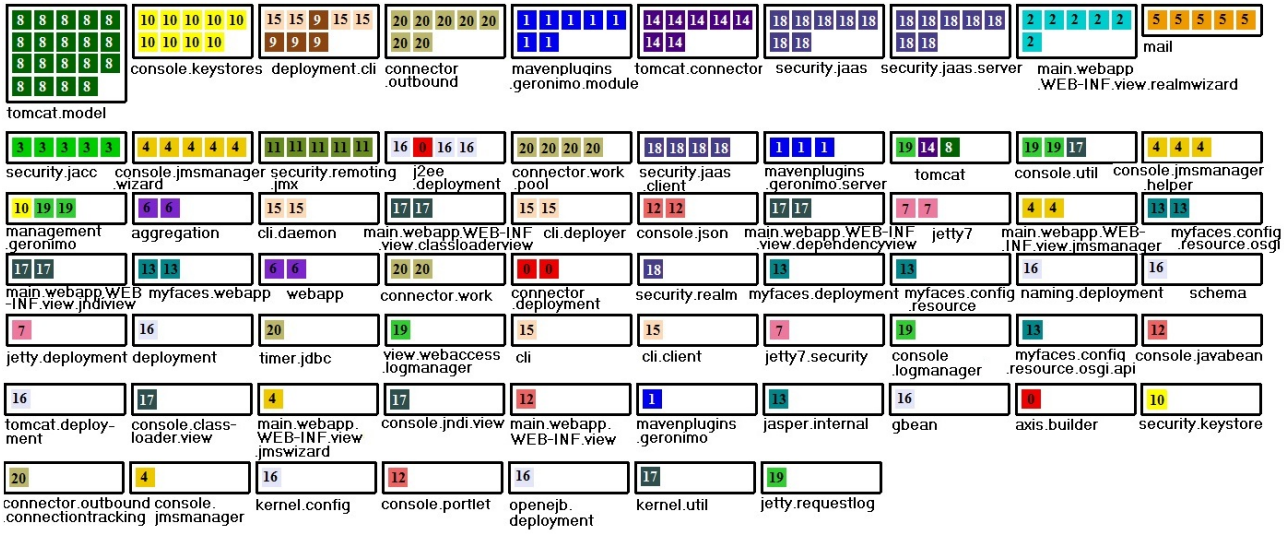

Figure 3. Distribution map for Geronimo

4.1 Distribution Map for Geronimo

Figure 3 shows the distribution map for Geronimo. To improve the visualization, besides background colors, we use a number in each class (small squares) to indicate their respective clusters. The large boxes are the packages and the text below is the package name.

Considering the clusters that are *well-encapsulated* (high focus) in Geronimo, we found three package distribution patterns:

- *Clusters well-encapsulated (focus = 1.0) in a single package (spread = 1).* Four clusters have this behavior. As an example, we have Cluster 2, which dominates the co-change classes in the package `main.webapp.WEBINF.view.realmwizard` (line 1 in the map, column 9). This package implements a wizard to configure or create security domains. Therefore, since it implements a specific functional concern, maintenance is confined in the package. As another example, we have Cluster 5 (package `mail`, line 1 in the map, column 10) and Cluster 11 (package `security.remoting.jmx`, line 1, column 3).
- *Clusters well-encapsulated (focus = 1.0) in more than one package (spread > 1).* We counted eight clusters with this behavior. As an example, we have Cluster 18 (*spread = 4*), which touches all co-change classes in the following packages: `security.jaas.server`, `security.jaas.client`, `security.jaas`, and `security.realm` (displayed respectively in line 1, columns 7 and 8; line 2, column 6; and line 4, column 6). As suggested by their names, these packages are related to security concerns, implemented using the Java Authentication and Authorization Service (JAAS) framework. Therefore, the packages are conceptually related and their spread should not be regarded as a design problem. In fact, the spread in this case is probably due to a decision to organize the source code in sub-packages.

  As another example, we have Cluster 20 (*spread* = 5), which touches all classes in `connector.outbound`, `connector.work.pool`, `connector.work`, `connector.outbound.connectiontracking`, and `timer.jdbc` (displayed respectively in line 1, column 4; line 2, column 5; line 4, column 4; line 7, column 1; line 5 and column 3). These packages implement EJB connectors for message exchange.
- *Clusters partially encapsulated (focus ≈ 1.0), but touching classes in other packages (spread > 1).*[5] As an example,

[5] These clusters are called octopus, because they have a body centered on a single package and tentacles in other packages [12].

we have Cluster 8 (*focus* = 0.97, *spread* = 2), which dominates the co-change classes in the package `tomcat.model` (line 1 and column 1 in the map), but also touches the class `TomcatServerGBean` from package `tomcat` (line 2, column 8). This class is responsible for configuring the web server used by Geronimo (Tomcat). Therefore, this particular co-change instance suggests an instability in the interface provided by the web server. In theory, Geronimo should only call this interface to configure the web server, but in fact the co-change cluster shows that maintenance in the `model` package sometimes has a ripple effect on this class, or vice-versa.

As another example, we have Cluster 14 (*focus* = 0.92 and *spread* = 2), which dominates the package `tomcat.connector` (line 1 and column 6 in the map) but also touches the class `TomcatServerConfigManager` from package `tomcat` (line 2, column 8). This "tentacle" in a single class from another package suggests again an instability in the configuration interface provided by the underlying web server.

4.2 Distribution Map for Lucene

We selected for analysis clusters that are *crosscutting* (focus $\approx$ 0.0), since they are much more common in Lucene. More specifically, we selected the three clusters in Lucene with the lowest focus and a spread greater than two. Figure 4 shows a fragment of the distribution map for Lucene, containing the following clusters:

- *Cluster 12 (focus = 0.06 and spread = 3)* with co-change classes in the following packages: `index`, `analysis`, and `store`. Since the cluster crosscuts packages that provide very different services (indexing, analysis, and storing), we claim that it reveals a modularization flaw in the package decomposition followed by Lucene. For example, a package like `store` that supports binary `I/O` services should hide its implementation from other packages. However, the existence of recurring maintenance tasks crosscutting `store` shows that the package fails to hide its main design decisions from other packages in the system.
- *Cluster 13 (focus = 0.2 and spread = 3)*, with co-change classes in the following packages: `search`, `search.spans`, and `search.function`. In this case, we claim that crosscutting causes less harm to modularity, because the packages are related to a single service (searching).
- *Cluster 28 (focus = 0.21 and spread = 6)*, with co-change classes in the following packages: `index`, `search`, `search.function`, `index.memory`, `search.highlight`, and `store.instantiated`. These packages are responsible for important services in Lucene, like indexing, searching, and storing. Therefore, as in the case of Cluster 12, the crosscutting behavior of this cluster suggests a modularization flaw in the system.

We also analyzed the maintenance issues associated to the commits responsible for the co-changes in Cluster 28. Particularly, we retrieved 37 maintenance issues related to this cluster. We then manually read and analyzed the short description of each issue, and classified them in three groups: (a) maintenance related to functional concerns in Lucene's domain (like searching, indexing, etc); (b) maintenance related to non-functional concerns (like logging, persistence, exception handling, etc); (c) maintenance related to refactorings. Table 10 shows the number of issues in each category. As can be observed, the crosscutting behavior of Cluster 28 is more due to issues related to functional concerns (59.5%) than to traditional non-functional concerns (8%). Moreover, changes motivated by refactorings (32.5%) are more common than changes in non-functional concerns.

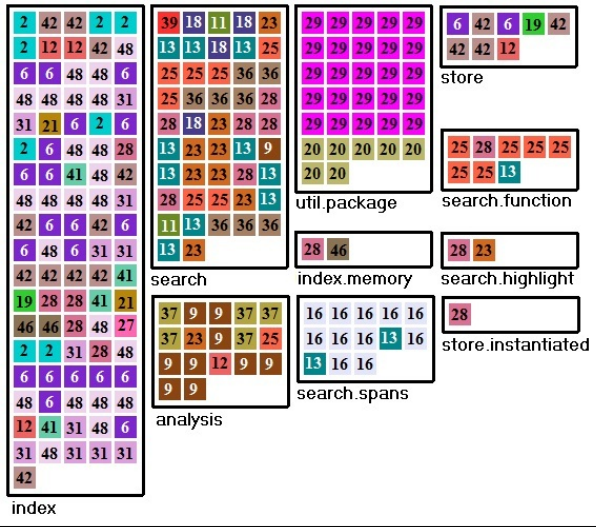

Figure 4. Part of the Distribution map for Lucene

Finally, we detected a distribution pattern in Lucene that represents neither well-encapsulated nor crosscutting clusters, but that might be relevant for analysis:

- *Clusters well-confined in packages (spread = 1).* Although restricted to a single package, these clusters do not dominate the colors in this package. But when considered as a single cluster, they dominate their package. As a concrete example, we have Cluster 20 (*focus = 0.22*) and Cluster 29 (*focus = 0.78*) that are both confined in package `util.packed` (line 1, column 3). Therefore, in this case a refactoring that splits the package in sub-packages can be considered, in order to improve the focus of the respective clusters.

Table 10. Maintenance issues in Cluster 28

Maintenance Type	# issues	% issues
Functional concerns	22	59.50%
Non-functional concerns	3	8.00%
Refactoring	12	32.50%

4.3 Distribution Map for JDT Core

Figure 5 shows the distribution map for JDT Core. We selected three distinct types of clusters for analysis: a *crosscutting* cluster (focus $\approx$ 0.0 and spread $>=$ 3), a clusters confined in a single package with (spread = 1), and a cluster with high spread.

- *Clusters with crosscutting behavior*. We have Cluster 4 (focus = 0.08 and spread = 4) with co-change classes in the following packages: `core.dom`, `internal.core`, `internal.compiler.lookup`, and `internal.core.util`. The `core.util` package provides a set of tools and utilities for manipulating `.class` files and Java model elements. Since the cluster crosscuts packages providing very different services (document structure, files and elements manipulation, population of the model, compiler infrastructure), we claim that it reveals a modularization flaw in the system.
- *Clusters well-confined in packages (spread = 1)*. We have Cluster 0 (*focus = 0.48*), Cluster 5 (*focus = 0.35*), and Cluster 6 (*focus = 0.07*) in the `core.dom` package (line 1, column 1).

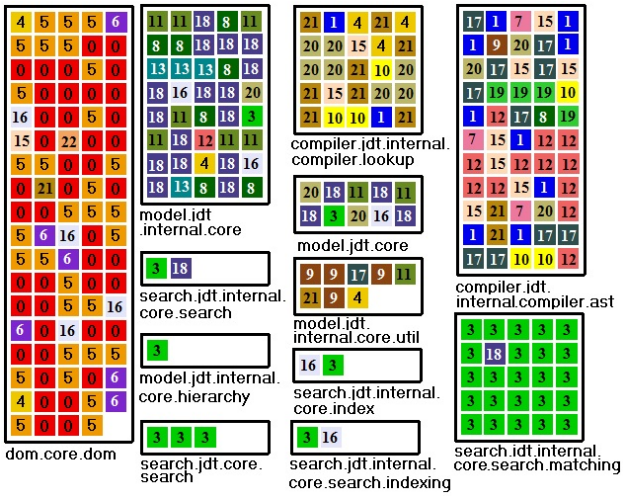

Figure 5. Part of the Distribution map for JDT Core

- *Clusters partially encapsulated (focus ≈ 1.0), but touching classes in other packages (spread > 1).* We have Cluster 3 (focus = 0.87 and spread = 8), which dominates the co-change classes in the packages `search.jdt.internal.core.search.matching` and `search.jdt.core.search`. These packages provide support for searching the workspace for Java elements matching a particular description. Their spread should not be regarded as a design problem, because the packages are related to a single service (searching). However, the cluster also touches classes in other packages. For example, the class `core.index.Index` maps document names to their referenced words in various categories.

5. SEMANTIC SIMILARITY ANALYSIS

The previous section has shown that the package modular structure of Geronimo has more adherence to co-change clusters than Lucene's and JDT Core's. We also observed that patterns followed by the relation clusters vs. packages can help to assess the modularity of systems. This section aims at evaluating the semantic similarity of the issues that are related to a specific cluster in order to improve our understanding of the clusters' meaning. We consider that if the issues related to a cluster have high semantic similarity, then the classes within that cluster are also semantically related and the cluster is semantically cohesive. We assume that an issue is related to a cluster if the change set of the issue contains at least a pair of classes from that cluster, not necessarily linked with an edge. In our strategy to evaluate the similarity of the issues related to a cluster, we consider each short description of a issue as a document and the collection of documents is obtained from the collection of issues related to a cluster. We will use Latent Semantic Analysis - LSA [11] to evaluate the similarity among the collection of documents related to a cluster because it is a well-known method used in other studies concerning similarity among issues and other software artifacts [28, 29].

5.1 Pre-processing Issue Description

When analyzing text documents with Information Retrieval techniques, an adequate pre-processing of the text is important to achieve good results. We determined a domain vocabulary of terms based on words found in commits of the target system. The first step is stemming the terms. Next, the stop-words were removed. The final step produces a term-document matrix, where the cells have value 1 if the term occurs in the document and 0 otherwise. This decision was taken after some qualitative experimentation, in which we observed that different weighting mechanisms based on the frequency of terms, such as td-idf [23], did not improved the quality of the similarity matrix.

5.2 Latent Semantic Analysis

The LSA algorithm is applied to the binary term-document matrix and produces another similarity matrix among the documents (issues) with values ranging from -1 (no similarity) to 1 (maximum similarity). The LSA matrix should have high values to denote a collection of issues that are all related among them. However, not all pairs of issues have the same similarity level, so it is necessary to analyze the degree of similarity between the issues to evaluate the overall similarity within a cluster. We used heat maps to visualize the similarity between issues that are related to a cluster. Figures 6 shows examples of similarity within specific clusters. We show for each system the two best clusters in terms of similarity to the left, and the two clusters with several pairs of issues with low similarity to the right. The white cells represent that the issues do not have any word in common, blue cells represent very low similarity, and yellow cells denote the maximum similarity between the issues.

We can observe that even for the cluster with more blue cells, there is still a dominance of higher similarity cells. The white cells in JDT's clusters suggest that there are issues with no similarity between the others in their respective cluster.

5.3 Scoring clusters

We propose the following metric to evaluate the overall similarity of a cluster c:

$$similarity\ score(c) = \frac{\sum_{\substack{0<i,j<n-1 \\ j<i}} similar(i,j)}{(\frac{n^2}{2} - n)}$$

where

$$similar(i,j) = \begin{cases} 0, & \text{if } LSA\_Cosine(i,j) < SIM\_THRS \\ 1, & \text{if } LSA\_Cosine(i,j) \geq SIM\_THRS \end{cases}$$

n = number of issues related to cluster c
$SIM\_THRS = 0.4$

The meaning of the *similarity score* of a cluster is defined upon the percentage of similar pair of issues related to that cluster. So, a cluster with score = 0.5, means that 50% of pairs of issues related to that cluster are similar to each other.

In this work, we had to define a threshold to evaluate if two issues are similar or not. We consider the semantic similarity between two issue reports, i and j, as the cosine between the vectors corresponding to i and j in the semantic space created by LSA. After experimental testing, we observed that pairs of issues (i,j) that had $LSA\_Cosine(i,j) \geq 0.4$ had a meaningful degree of similarity. Nonetheless, we agree that this fixed threshold cannot be free of imprecision. Similar to our study, Poshyvanyk and Marcus [29] used LSA to analyze the coherence of the user comments in bug reports. The system's developers classified as high/very high similar, the comments with average similarity greater than 0.33, so our more conservative approach seems to be quite adequate.

Moreover, because our goal is to have an overall evaluation of the whole collection of co-change clusters, some imprecision in the characterization of similarity between two issues would not affect significantly our analysis of the distribution of clusters' scores. Figures 7 shows the distribution of score values for Geronimo's, Lucene's, and JDT's clusters.

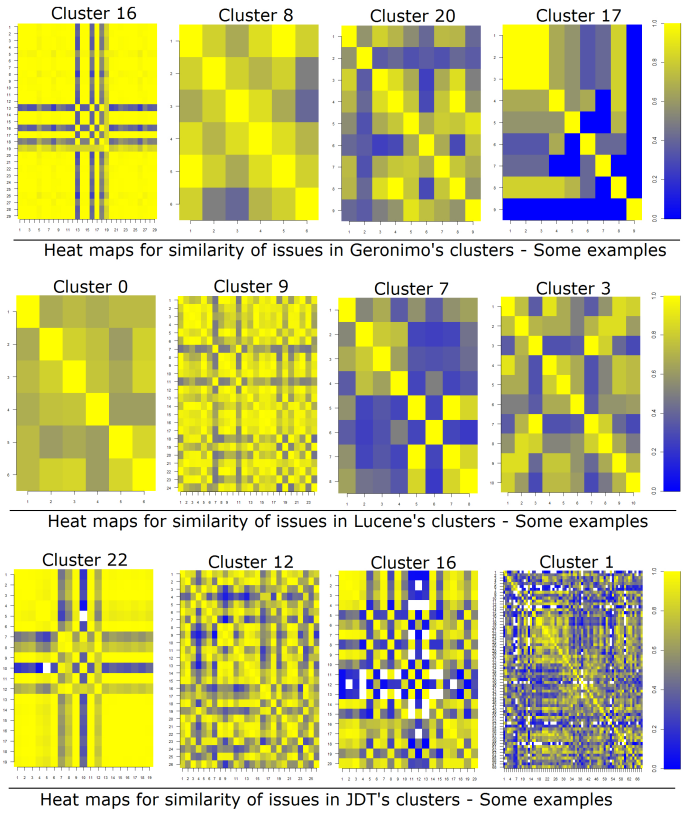

Figure 6. Examples of heat maps for similarity of issues

We can observe that the systems' clusters follow a similar pattern of scoring, with 100% (for Lucene and JDT) and more than 90% (for Geronimo) of clusters having more than half pairs of issues similar to each other.

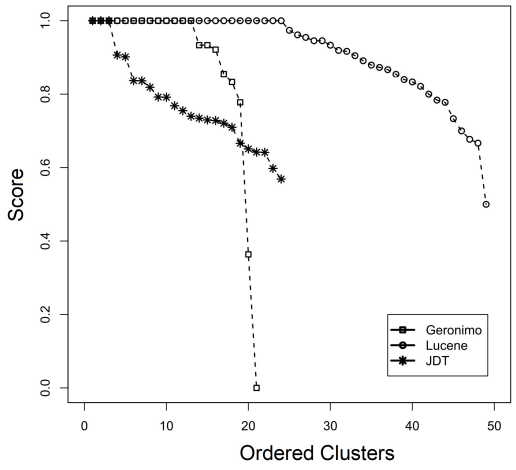

Figure 7. Distribution of the clusters' score

5.4 Correlating Similarity, Focus, and Spread

Another analysis that we carried out with clusters' scores was to evaluate the degree of correlation between the score, focus and spread. Table 11 shows the results obtained by applying the Spearman correlation test. For Geronimo, we observed a strong negative correlation between spread and score. In other words, the higher is the number of similar issues in a cluster, the higher is the capacity of the cluster to encompass a whole package in Geronimo. Interestingly, Lucene does not present the same behavior as Geronimo. We observe a weak correlation between focus and score, but we encounter no significant correlation between spread and score. In the case of Lucene, the higher is the number of similar issues in a cluster, the lower is the number of packages touched by the cluster. In the case of Eclipse JDT Core, there is no significant correlation between focus and score. Although, there is a moderate negative correlation between spread and score, it is only significant at p-value 0.074. Considering that the clusters of the analyzed systems followed a similar pattern of similarity, this result suggests that the reasonable similarity between co-change induces different properties in the clusters, either in spread or in focus.

Table 11. Correlation between score, focus and spread of clusters for Geronimo, Lucene, and JDT Core

Correlation Coefficient *p-value*	**Score Geronimo**	**Score Lucene**	**Score JDT**
Focus	0.264 0.131	0.308 0.016	−0.015 0.473
Spread	−0.720 1.71×10^{-4}	−0.178 0.111	−0.304 0.074

6. DISCUSSION

6.1 Practical Implications

Software architects can rely on the approach proposed in this paper to assess modularity under an evolutionary dimension. More specifically, we claim that our approach helps to reveal the following patterns of co-change behavior:

- When the package structure is adherent to the cluster structure, as in Geronimo's clusters (43% well encapsulated), then localized co-changes are likely to occur.
- When there is not a clear adherence between co-change clusters and packages, a restructuring of the package decomposition may be necessary to improve modularity. Particularly, there are two patterns of clusters that may suggest modularity flaws. The first pattern denotes clusters with crosscutting behavior (focus ≈ 0 and high spread). For example, in Lucene and JDT Core we detected 12 and 10 clusters related to this pattern, respectively. The second pattern is the octopus cluster that suggest a possible ripple effect during maintenance tasks. In Geronimo and Lucene, we detected four and five clusters related to this pattern, respectively.

On the other hand, modular designs usually demand well-trained, skilled, and experienced software architects. Nonetheless, we have no evidence that the proposed co-change clusters may fully replace traditional modular decompositions. Indeed, a first obstacle to this proposal is the fact that co-change clusters do not cover the whole population of classes in a system. On the other hand, we believe that they can be used as an alternative modular view during program comprehension tasks. For example, they may provide a better context during maintenance tasks (similar for example to the task context automatically inferred by tools like Mylyn [20]).

6.2 Clustering vs Association Rules Mining

Our approach is centered on the Chameleon hierarchical clustering algorithm, since this algorithm was designed to handle sparse graphs [18]. In our case studies, for example, the co-change graphs have densities ranging from 2% (Geronimo and Lucene) to 4% (Eclipse JDT Core). Particularly, in traditional clustering algorithms, like K-Means [22], the mapping of data items to clusters is a total function, i.e., each data item is allocated to a specific cluster. Likewise K-Means, Chameleon tries to cluster all data items (vertices). However, when some vertices do not share any edge with the rest of the vertices, the number of clustered vertices is fewer than the total number of initial vertices.

An alternative to retrieve co-change relations is to rely on association rules mining [2]. In the context of evolutionary coupling, an association rule $C_{ant} \Rightarrow C_{cons}$ express that commit transactions changing the classes C_{ant} (antecedent term) also change C_{cons} classes (consequent term), with a given probability.

However, hundreds of thousands of association rules can be easily retrieved from version histories. For example, we executed the Apriori algorithm [2] to retrieve association rules for the Lucene system. By defining a minimum support threshold of four transactions, a minimum confidence of 50%, and limiting the size of the rules to 10 classes, we mined 976,572 association rules, with an average size of 8.14 classes. We repeated this experiment with the confidence threshold of 90%. In this case, we mined 831,795 association rules, with an average size of 8.23 classes. This explosion in the number of rules is an important limitation for using association rules to assess modularity, which ultimately is a task that requires careful judgment and analysis by software developers and maintainers.

6.3 Threats to Validity

In this section, we discuss possible threats to validity, following the usual classification in threats to internal, external, and construct validity:

Threats to External Validity: There are some threats that limit our ability to generalize our findings. The use of Geronimo, Lucene, and JDT Core may not be representative to capture co-change patterns present in other systems. However, it is important to note that we do not aim to propose general co-change patterns, but instead we just claim that the patterns founded in the target systems show the feasibility of using co-change clusters to evaluate modularity under a new dimension.

Threats to Construct Validity: A possible design threat to construct validity is that developers might not adequately link commit with issues, as pointed out by Herzing and Zeller [15]. Moreover, we also found a high number of commits not associated to maintenance issues. Thus, our results are subjected to missing and to incorrect links between commits and issues. However, we claim at least that we followed the approach commonly used in other studies that map issues to commits [8–10, 40]. We also filtered out situations like commits associated to multiple maintenance issues and highly scattered commits. Another possible construction threat concerns the time frame used to collect the issues. We considered activity in a period of approximately ten years, which is certainly a large time frame. However, we did not evaluate how the co-change clusters evolved during this time frame or whether the systems' architecture substantially changed.

Finally, our approach only handles co-changes related to source code artifacts (.java files). However, the systems we evaluated have other types of artifacts, like XML configuration files. Geronimo for example has 177 Javascript files, 1004 XML configuration files, 19 configuration files, and 105 image files. Therefore, it is possible that we missed some co-change relations among non-Java based artifacts or between non-Java and Java-based artifacts. On the other hand, considering only source code artifacts makes possible the projection of co-change clusters to distribution maps, using the package structure as the main partition in the maps.

Threats to Internal Validity: Our approach relies on filters to select the commits used by the co-change graphs and clusters. Those filters are based on thresholds that could be defined differently, despite of our careful pre-experimentation. We also calibrated the semantic similarity analysis with parameters that define the dimensionality reduction in the case of LSA, and with a threshold in the case of the LSA_Cosine coefficient that defines when a pair of issues is similar. Although this calibration has some degree of uncertainty, it was not proposed to get better results favoring one system instead of the other. We defined the parameters and constants so that coherent results were achieved in all systems. Moreover, we observed that variations in the parameters' values would affect the results for all systems in a similar way.

7. RELATED WORK

In this section, we discuss work related to our approach. The discussion is organized in three sections: concern mapping, co-change mining, and aspect mining.

7.1 Concern Mapping

Several approaches have been proposed to help developers and maintainers to manage concerns and features. For example, concern graphs model the subset of a software system associated with a specific concern [30, 31]. The main purpose is to provide developers

with an abstract view of the program fragments related to a concern. FEAT is a tool that supports the concern graph approach by enabling developers to build concern graphs interactively, as result of program investigation tasks. Aspect Browser [14] and JQuery [16] are other tools that rely on lexical or logic queries to find and document code fragments related to a certain concern. ConcernMapper [32] is an Eclipse Plug-in to organize and view concerns using an hierarchical structure similar to the package structure. However, in such approaches, the concern model is created manually or based on explicit input information provided by developers. Moreover, the relations between concerns are typically only syntactical and structural. On the other hand, in the approach proposed in this paper, the elements and relationships are obtained by mining the version history.

7.2 Co-change Mining

Zimmermann et al. proposed an approach that uses association rule mining on version histories to suggest possible future changes [39]. Their approach differs from ours because they rely on association rules to recommend further changes (e.g., if class `A` usually co-changes with `B`, and a commit only changes `A`, a warning is given suggesting to check whether `B` should not be changed too). On the other hand, we use co-change graphs to retrieve clusters semantically related to a target system's concern. A co-change graph is created from the selection of commits linked to their respective maintenance issues. Furthermore, our goal is not to recommend future changes, but to assess modularity, using distribution maps to compare and contrast co-change clusters with the system's packages.

Beyer and Noack introduced the concept of co-change graphs and proposed a visualization of such graphs to reveal clusters of frequently co-changed artifacts [5]. Their approach clusters all co-change artifacts (code, configuration scripts, documentation, etc), representing files as co-change graphs' vertices. These vertices are displayed as circles and their area is proportional to the frequency that the file was changed. The vertex color represents its respective cluster. However, they do not define pre-processing and post-processing filters. In contrast, we prune many classes during the pre-processing and post-processing phases and after clustering the co-change graphs. Finally, our central goal is not directly related with improving the visualization of co-change clusters, but on using them to assess modularity.

Oliva et al. mined version histories to extract logical dependencies between software artifacts to identify their origins [26]. They conducted a manual investigation of the origins of logical dependencies by reading revision comments and analyzing code diffs. Beck and Diehl combined evolutionary dependencies with syntactic dependencies to retrieve the modular structure of a system [4]. However, they clustered all classes in a system, since their original goal was to compare both approaches to software clustering. On the other hand, since our goal is to assess modularity, we consider only high-density co-change clusters.

Huzefa et al. presented an approach that combines conceptual and evolutionary couplings for impact analysis in source code [17], using information retrieval and version history mining techniques. Gethers et al. proposed an impact analysis that adapts to the specific maintenance scenario using information retrieval, historical data mining, and dynamic analysis techniques [13]. However, they did not use maintenance issues reports to discard noisy commits.

A recent study by Negara et al. revealed that the use of data from version history presents many threats when investigating source code properties [25]. For example, developers usually fix failing tests by changing the test themselves, commit tasks without testing, commit the same code fragment multiple times (in different commits), or take days to commit changes containing several types of tasks. In this work, we proposed five pre-processing tasks and one post-processing task to tackle some of such threats.

7.3 Aspect Mining

Breu and Zimmermann proposed an approach (HAM) based on version history to detect cross-cutting concerns in an object-oriented program to guide its migration to an aspect-oriented program [6]. They defined the notion of transaction, which is the set of methods inserted by the developer to complete a single development task. They also considered that method calls inserted in eight or more locations (method bodies) define aspect candidates. One important difference from their work and ours is that they consider not only methods that were changed together, but also those changes that were the same. Moreover, they rely on a fine-grained notion of change that is interested in finding methods calls to define aspect candidates.

Adams et al. proposed a mining technique (COMMIT) to identify concerns from functions, variables, types, and macros that were changed together [1]. Similarly to HAM, COMMIT is based on the idea that similar calls and references that are added or removed into different parts of the program are candidates to refer to the same concern. This information produces several seed graphs which are concern candidates because nodes in the graph represent program entities to which calls or references have been co-added or co-removed. Their approach differs from ours because they generate independent seed graphs, while we are centered on a unique graph.

8. CONCLUDING REMARKS

In this work, we proposed a method to extract an alternative view to the package decomposition based on co-change clusters. We applied our method to three real software systems, Geronimo, Lucene, and JDT Core, that have approximately ten years of committed changes. Our results show that meaningful co-change clusters can be extracted using the information available in version control systems. Although co-change graphs extracted from repositories are sparse, the co-change clusters were dense and have high internal similarity concerning co-changes and semantic similarity concerning their originating issues. We have shown that co-change clusters and their associated metrics were useful to assess the hierarchical modular decomposition of the target systems. Even if in some cases co-change clusters may be used to restructure the original package decomposition, we suggest that they can also be use as an alternative view during maintenance tasks to improve the developer's understanding of the change impact.

We still need to investigate the reasons that induce co-change clusters and to identify the eventual patterns that produce those clusters, which would contribute in early modularization decisions. We plan to investigate and to compare our approach with other clustering algorithms for sparse graphs, like the approach proposed by Beyer et al. [5]. We also plan to consider co-changes at a finer-granularity level, more specifically among methods, and also including non-source code artifacts, like XML configuration files. Finally, we plan to investigate whether co-change clusters can be used as an alternative to the Package Explorer, supporting a mechanism for the virtual separation of concerns, inspired on the CIDE tool [19]. However, CIDE supports the virtual separation of features whose implementation physically crosscuts many classes. On the other hand, our goal is to support the virtual separation of features with a strong temporal relationship, in terms of co-changes.

Acknowledgments

This work was partially supported by FAPEMIG (grants CEX-APQ-2086-11, PPM-00388-13) and CNPq (grants 475519/2012-4, 304897/2011-6).

References

[1] B. Adams, Z. M. Jiang, and A. E. Hassan. Identifying crosscutting concerns using historical code changes. In *32nd International Conference on Software Engineering*, pages 305–314. ACM, 2010.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[3] C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity*. MIT Press, 2003.

[4] F. Beck and S. Diehl. Evaluating the impact of software evolution on software clustering. In *17th Working Conference on Reverse Engineering (WCRE)*, pages 99–108, 2010.

[5] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *13th International Workshop on Program Comprehension (IWPC)*, pages 259–268, 2005.

[6] S. Breu and T. Zimmermann. Mining aspects from version history. In *21st Automated Software Engineering Conference (ASE)*, pages 221–230, 2006.

[7] S. Chidamber and C. Kemerer. Towards a metrics suite for object oriented design. In *6th Object-oriented programming systems, languages, and applications Conference (OOPSLA)*, pages 197–211, 1991.

[8] C. Couto, C. Silva, M. T. Valente, R. Bigonha, and N. Anquetil. Uncovering causal relationships between software metrics and bugs. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 223–232, 2012.

[9] C. Couto, P. Pires, M. T. Valente, R. Bigonha, and N. Anquetil. Predicting software defects with causality tests. *Journal of Systems and Software*, pages 1–38, 2014.

[10] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *7th Working Conference on Mining Software Repositories (MSR)*, pages 31–41, 2010.

[11] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science,*, 41:391–407, 1990.

[12] S. Ducasse, T. Gîrba, and A. Kuhn. Distribution map. In *22nd IEEE International Conference on Software Maintenance (ICSM)*, pages 203–212, 2006.

[13] M. Gethers, H. Kagdi, B. Dit, and D. Poshyvanyk. An adaptive approach to impact analysis from change requests to source code. In *26th Automated Software Engineering Conference (ASE)*, pages 540–543, 2011.

[14] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *23rd International Conference on Software Engineering (ICSE)*, pages 265–274, 2001.

[15] K. Herzing and A. Zeller. The impact of tangled code changes. In *10th Working Conference on Mining Software Repositories (MSR)*, pages 121–130, 2013.

[16] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *2nd International Conference on Aspect-oriented Software Development (AOSD)*, pages 178–187, 2003.

[17] H. Kagdi, M. Gethers, and D. Poshyvanyk. Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering (EMSE)*, 2013.

[18] G. Karypis, E.-H. S. Han, and V. Kumar. Chameleon: hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.

[19] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *30th International Conference on Software Engineering (ICSE)*, pages 311–320, 2008.

[20] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *14th International Symposium on Foundations of Software Engineering (FSE)*, pages 1–11, 2006.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[22] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[23] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[24] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2000.

[25] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *26th European conference on Object-Oriented Programming (ECOOP)*, pages 79–103, 2012.

[26] G. A. Oliva, F. W. Santana, M. A. Gerosa, and C. R. B. de Souza. Towards a classification of logical dependencies origins: a case study. In *12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution (EVOL/IWPSE)*, pages 31–40, 2011.

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

[28] D. Poshyvanyk and A. Marcus. Using information retrieval to support design of incremental change of software. In *22th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 563–566, 2007.

[29] D. Poshyvanyk and A. Marcus. Measuring the semantic similarity of comments in bug reports. In *1st International ICPC2008 Workshop on Semantic Technologies in System Maintenance (STSM)*, pages 265–280, 2008.

[30] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *24th International Conference on Software Engineering (ICSE)*, pages 406–416, 2002.

[31] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):1–38, 2007.

[32] M. P. Robillard and F. Weigand-Warr. Concernmapper: simple view-based separation of scattered concerns. In *OOPSLA workshop on Eclipse technology eXchange*, eclipse '05, pages 65–69, 2005.

[33] G. Santos, M. T. Valente, and N. Anquetil. Remodularization analysis using semantic clustering. In *1st CSMR-WCRE Software Evolution Week*, pages 224–233, 2014.

[34] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *2nd Working Conference on Mining Software Repositories (MSR)*, pages 1–5, 2005.

[35] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, June 1974.

[36] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *Asia Pacific Software Engineering Conference (APSEC)*, pages 336–345, 2010.

[37] R. Terra, L. F. Miranda, M. T. Valente, and R. S. Bigonha. Qualitas.class corpus: A compiled version of the qualitas corpus. *Software Engineering Notes*, pages 1–4, 2013.

[38] R. J. Walker, S. Rawal, and J. Sillito. Do crosscutting concerns cause modularity problems? In *20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 1–11, 2012.

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

[40] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. In *3rd International Workshop on Predictor Models in Software Engineering*, page 9, 2007.