Co-change Clusters: Extraction and Application on Assessing Software Modularity

Luciana Lourdes Silva[1,3], Marco Tulio Valente[1], and Marcelo de A. Maia[2]

[1] Department of Computer Science,
Federal University of Minas Gerais,
{luciana.lourdes,mtov}@dcc.ufmg.br
[2] Faculty of Computing,
Federal University of Uberlândia,
marcmaia@facom.ufu.br
[3] Federal Institute of the Triângulo Mineiro, Brazil

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 four real-world systems. The results show that the projection of co-change clusters to packages follows 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.

Keywords: Modularity, software changes, version control systems, co-change graphs, co-change clusters, agglomerative hierarchical clustering 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 [33]. In this way, modularity contributes to improve productivity 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 [28, 23]. 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) [9, 45]. However,

typical cohesion and coupling metrics measure a single dimension of the software implementation (the static-structural dimension). For this reason, 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 [22, 38, 39]. 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 [33]. 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. Basically, co-change clusters are sets of classes that frequently changed together in the past.

Our approach relies on distribution maps [14]—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 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 four real-world systems (Geronimo, Lucene, Eclipse JDT Core, and Camel).

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 usually the case of co-change graphs. 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 are not only dense in terms of co-changes, but they also have high similarity from the point of view of the meaning of the maintenance issues that originated the respective commits.

This paper is a revised and extended version of our work that appeared in [43]. The key additions in this new version are as follows. First, Section II describes several concepts needed to comprehend our methodology and analysis. Second, we improved Section III by adding examples and new information concerning the extraction of co-change clusters. Third, this paper contains new experimental results in Sections V and VI, including the extraction of co-change clusters for

a new system (Camel). Finally, we improved the discussion on using association rules to retrieve co-change relations in Section 7.2.

The paper is organized as follows. Section 2 presents background on clustering techniques and Latent Semantic Analysis. Section 3 presents the methodology to extract co-change graphs and co-change clusters from version control systems. Section 4 presents the results of co-change clustering, when applied to four systems. Section 5 analyzes the modularity of such systems under the light of co-change clusters. Section 6 analyzes the semantic similarity regarding the set of issues related to the extracted clusters. Section 7 discusses our results and presents threats to validity. Section 8 describes related work and finally Section 9 concludes the paper.

2 Background

This section provides background on clustering and information retrieval techniques. We use clustering techniques to retrieve clusters of software artifacts that changed together in the past, which are called co-change clusters. We also use linguistic preprocessing, Latent Semantic Analysis (LSA), and cosine similarity techniques to evaluate the meaning of issue reports related to software maintenance activities.

2.1 Clustering Techniques

Clustering techniques are broadly classified in partitioning and hierarchical. On one hand, a partitioning approach divides the set of data objects into K clusters such that each data object is in exactly one cluster. On the other hand, hierarchical clustering yields a tree of clusters, known as a dendrogram. It is further subdivided into agglomerative (bottom-up) and divisive (top-down). An agglomerative clustering starts with each data object being a single cluster and repeatedly merges two, or more, most appropriate clusters. A divisive clustering starts with a single cluster containing all data objects and repeatedly splits the most appropriate cluster. The process continues until a stop criterion is achieved, usually the requested number of K clusters.

2.2 Chameleon Clustering Algorithm

Chameleon [20] is an agglomerative hierarchical clustering algorithm designed to handle sparse graphs in which nodes represent data objects, and weighted edges represent similarities among these objects.

Input. Chameleon requires as input a matrix whose entries represent the similarity between data objects. A sparse graph representation is created following a k-nearest-neighbor graph algorithm. Each vertex represents a data object and there is an edge between two vertices u and v if v is among the k most similar points to u or vice-versa.

Constructing a Sparse Graph. In this step, data objects that are far away are disconnected in order to reduce noise. As Chameleon operates on a sparse graph, each cluster is a subgraph of the sparse graph. The sparse graph allows Chameleon to deal with large data sets and to successfully use data sets in similarity space.

First Phase (Partitioning the Graph): A min-cut graph partitioning algorithm is used to partition the k-nearest-neighbor graph into a pre-defined number of subclusters M. If the graph contains a single connected component, then the algorithm returns k subclusters. Otherwise, the number of subclusters after this phase is M plus C, where C is the number of connected components. Since each edge represents similarity among objects, a min-cut partitioning is able to minimize the connection among data objects through the partitions.

Second Phase (Merging Partitions): This phase uses an agglomerative hierarchical algorithm to merge the small clusters, created by the first phase, 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). Chameleon models similarity based on the Relative Interconnectivity (RI) and Relative Closeness (RC) of the clusters. A pair of clusters C_i and C_j is selected to be merge when both RI and RC are high, suggesting that they are highly interconnected as well as close together. Chameleon's authors provide a software package, named Cluto[4], which supports different agglomerative merging schemes in this second phase. As our goal is to find co-change clusters in a sparse graph, we are interested in functions that do not cluster all data objects (in our case, classes). The artifacts discarded by Chameleon are considered noisy data because they do not share any edges with the rest of the artifacts.

2.3 Latent Semantic Analysis

The discussion on Latent Semantic Analysis (LSA) [13] is relevant to our approach, since we evaluate the semantic similarity of issue reports that are related to a specific cluster in order to improve our understanding of the clusters' meaning. LSA is a statistical approach for extracting and representing the meaning of words. The semantic information is retrieved from a word-document co-occurrence matrix, where words and documents are considered as points in an Euclidean space. LSA is based on the Vector Space Model (VSM), an algebraic representation of documents frequently used in information retrieval [41]. The vector space of a text collection is constructed by representing each document as a vector with the frequencies of its words. The document vectors add to a term-by-document matrix representing the full text collection. First, the vocabulary of terms is determined using feature selection techniques such as tokenization, stop words removal, domain vocabulary, case-folding, stemming, and weighting

[4] http://glaros.dtc.umn.edu/gkhome/views/cluto

schemes (TF-IDF, binary weight) before representing the textual data in a numerical form. Moreover, LSA applies singular value decomposition (SVD) to the term-by-document matrix as a way of factor analysis. In SVD, a rectangular matrix is decomposed into the product of three other matrices — an orthogonal matrix U, a diagonal matrix Σ, and the transpose of an orthogonal matrix V. Suppose an original term-document matrix $C_{M\times N}$, where M is the number of terms and N is the number of documents. The matrix C is then decomposed via SVD into the term vector matrix U, the document vector matrix V, and the diagonal matrix Σ (consisting of eigenvalues) as follows:

$$C_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^T_{n\times n}$$

where $U^TU = I$ and $V^TV = I$. The columns of U are the orthogonal eigenvectors of CC^T and I is the identity matrix. The columns of V are the orthogonal eigenvectors of C^TC, and Σ is a diagonal matrix containing the square roots of eigenvalues from U or V in descending order.

Text Pre-processing Tasks When analyzing text documents, an adequate pre-processing step is crucial to achieve good results [27]. After collecting the documents to be analyzed, some steps are usually performed as follows:

Tokenization. The tokenization process is applied on the text chopping character streams into tokens, discarding special characters, such as punctuation and numbers. Furthermore, when dealing with software artifacts, CamelCase identifiers are also split into tokens.

Stop words removal. In this step, common words that are irrelevant when selecting documents matching an end-user needs are removed from the vocabulary. For example, words such as *a*, *an*, *and*, *was*, and *were*.

Case-folding. It is a common strategy by reducing all letters to lower case.

Stemming. Due to grammatical reasons, documents usually contain different forms of a word, such as *run*, *runs*, and *running*. The goal of stemming is to reduce the possible inflectional forms of a word to a common base form.

Cosine Similarity Since we consider the semantic similarity between two issue reports, i and j, cosine similarity measures the cosine of the angle between the vectors $\overrightarrow{v_i}$ and $\overrightarrow{v_j}$ corresponding to the issue reports d_i and d_j in the semantic space constructed by LSA:

$$sim(\overrightarrow{v_i}, \overrightarrow{v_j}) = \frac{\overrightarrow{v_i} \bullet \overrightarrow{v_j}}{|\overrightarrow{v_i}| \times |\overrightarrow{v_j}|}$$

where $\overrightarrow{v}$ is the vector norm and $\bullet$ is the vector internal product operator and $sim(\overrightarrow{v_i}, \overrightarrow{v_j}) \in [-1, 1]$.

3 Methodology

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

3.1 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.

Our approach relies on two inputs: issue reports available in issue trackers, e.g., Jira, Bugzilla, and Tigris; and commit transactions retrieved from version control repositories (SVN or GIT). In a further step, several processing tasks are applied and then a co-change graph is build. Finally, sets of classes that frequently change together are retrieved, called co-change clusters.

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. 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 [29]. 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. Moreover, we only consider 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 [12], [44], [51].

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. Figure 1 presents an example for the Geronimo system.[5] The figure shows the short description of four commits related to the issue `GERONIMO-3003`. In this case, a single change set is generated for the four commits, including 13 classes. In the co-change graph, an edge is created for each pair of classes in this merged change set. In this way, it is possible to have edges connecting classes modified in different commits, but referring to the same maintenance issue.

```
------------------------------------
Revision: 918360
Date: Wed Mar 03 05:07:00 BRT 2010
Short Description: GERONIMO-3003 create karaf command wrpaper
for encryptCommand
Changed Classes: [1 class]
------------------------------------
Revision: 798794
Date: Wed Jul 29 03:54:50 BRT 2009
Short Description: GERONIMO-3003 Encrypt poassoreds and morked
attributes in serialized gbeans and config.xml. Modified from
patch by [developer name], many thanks.
Changed Classes: [9 new classes]
------------------------------------
Revision: 799023
Date: Wed Jul 29 16:13:02 BRT 2009
Short Description: GERONIMO-3003 Encrypt poassoreds and morked
attributes in serialized gbeans and config.xml. Modified from
patch by [developer name], many thanks. 2nd half of patch.missed
adding one file and several geronimo-system changes earlier.
Changed Classes: [3 new classes]
------------------------------------
Revision: 799037
Date: Wed Jul 29 16:49:52 BRT 2009
Short Description: GERONIMO-3003 Use idea from [developer name]
to encrypt config.xml attributes that are encryptable but reset
to plaintext by users
Changed Classes: [1 class, also modified in revision 799023]
```

Fig. 1. Multiple commits for the issue GERONIMO-3003

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 [17]. Basically, such commits are discarded because otherwise they would result on edges connecting classes modified to implement semantically unrelated maintenance tasks (which are included in the same commit just by convenience, for example). Figure 2 presents a tangled code change for the Geronimo system.

Removing highly scattered commits: We remove commits representing highly scattered code changes, i.e., commits that modify a massive number of classes.

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

```
Revision: 565397
Date: Mon Aug 13 13:21:44 BRT 2007
Short Description: GERONIMO-3254 Admin Console Wizard to auto
generate geronimo-web.xml and dependencies GERONIMO-3394,
GERONIMO-3395, GERONIMO-3396, GERONIMO-3397,
GERONIMO-3398
- First commit of "Create Plan" portlet code. ....
Changed Classes: [25 classes]
```

Fig. 2. Single commit handling multiple issues (3254, 3394 to 3398)

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) [50]. Figure 3 illustrates a highly scattered commit in Lucene. This commit changes 251 classes, located in 80 packages. Basically, in this commit redundant throws clauses are refactored.

```
Revision: 1355069
Date: Thu Jun 28 13:39:25 BRT 2012
Short Description: LUCENE-4172: clean up redundant throws clauses
Changed Classes: [251 classes]
```

Fig. 3. Highly scattered commit (251 changed classes)

Recent research shows that scattering in commits tends to follow heavy-tailed distributions [50]. 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 4 illustrates this fact by showing a histogram with the number of packages changed by commits made to the Lucene system.[6] As we can observe, 1,310 commits (62%) change 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 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 4, we define and explain the values for thresholds in our method.

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

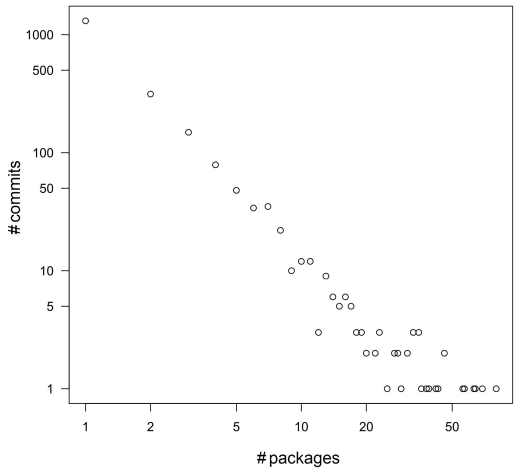

Fig. 4. Packages changed by commits in the Lucene system

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 occurred 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 [52].

3.2 Extracting Co-Change Clusters

After extracting the co-change graph, 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, which is an agglomerative and hierarchical clustering algorithm recommended to sparse graphs.

As reported in Section 2.2, there are several clustering criterion functions that can be applied in the agglomerative phase available in Cluto package. We conducted pre-experiments with those functions to find which one produces clusters with higher internal similarity, lower external similarity and higher density. The function i2 (Cohesion based on graphs) is the one that best fitted to our

goal. We observed that other functions retrieved several clusters with low density. The i2 function searches for subclusters to combine, maximizing the similarity by evaluating how close are the objects in a cluster, as follows:

$$maximize \sum_{i=1}^{k} \sqrt{\sum_{v,u \in S_i} sim(v,u)}$$

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 consider 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 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 the small ones.

The proposed $coefficient(M)$ 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 it represents how often the classes changed together in a cluster.

Table 1. Target systems (size metrics)

System	Description	Release	LOC	NOP	NOC
Geronimo	Web application server	3.0	234,086	424	2,740
Lucene	Text search library	4.3	572,051	263	4,323
JDT Core	Eclipse Java infrastructure	3.7	249,471	74	1,829
Camel	Integration framework	2.13.1	964,938	828	11,395

Table 2. Initial commits sample

System	Commits	Period
Geronimo	9,829	08/20/2003 - 06/04/2013 (9.75 years)
Lucene	8,991	01/01/2003 - 07/06/2013 (10.5 years)
JDT Core	24,315	08/15/2002 - 08/21/2013 (10 years)
Camel	13,769	04/18/2007 - 06/14/2014 (7 years)

4 Co-Change Clustering Results

In this section, we report the results we achieved after following the methodology described in Section 3 to extract co-change clusters for four systems.

4.1 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). Table 2 shows the number of commits extracted for each system and the time frame used in this extraction.

In order to run the approach, we define 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 changes in comments formatting and rename method [52], [1]. However, excessive pruning is also undesirable, so we adopted a conservative approach working at package level.
- $MIN_WEIGHT = 2$ co-changes, i.e., we discard edges connecting classes with less 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* tends to be low because subgraphs that are not well connected are grouped in the same cluster. We performed several experiments varying $M's$ value, and observed that whenever M is high, the clustering tends to have clusters of unbalanced size.

– *M_DECREMENT* = 1 class, i.e., after each clustering execution, we decrement the value of M by 1.
– *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 [50]. 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 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 [47, 46]. In our four 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.

4.2 Co-Change Graph Extraction

We start by characterizing the extracted co-change graphs. Table 3 shows the percentage of commits in our sample, after applying the preprocessing filters described in Section 3.1: removal of commits not associated to maintenance issues (Pre #1), removal of commits not changing classes and also referring to 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 3. Percentage of unitary commits (i.e., changing a single class) discarded in the first phase and commits discarded after each preprocessing filters

System	Pre #1	Unitary Commits	Pre #2	Pre #3	Pre #4	Pre #5
Geronimo	32.6	39.6	25.2	17.3	16.1	14.3
Lucene	39.2	35.3	34.6	23.6	23.3	22.4
JDT Core	38.4	58.1	32.8	21.7	20.3	20.1
Camel	45.0	44.5	39.7	25.7	21.7	21.3

As can be observed in Table 3, our initial sample for the Geronimo, Lucene, JDT Core, and Camel systems was reduced to 14.3%, 22.4%, 20.1% and 21.3% of its original size, respectively. The most significant reduction was due to the first preprocessing task. Basically, only 32.6%, 39.2%, 38.4%, and 45.0% of the commits in Geronimo, Lucene, JDT Core, and Camel are associated to maintenance issues (as stored in the systems issue tracking platforms). Moreover, we analyzed the commits discarded in first preprocessing task. We observed a substantial number of commits changing a single class, 39.6% of Geronimo's, 35.3%

of Lucene's, 58.1% of JDT's, and 44.5% of Camel's commits. These unitary commits may contain configuration or/and script files, for instance. In addition, as some of these commits are not linked to issue reports, we cannot assume they represent a complete maintenance task. We also had not analyzed if these unitary commits are partial implementations of maintenance tasks. This could be done by inspecting their time frame, for instance. However, these unitary commits are not useful anyway to evaluate a system in terms of co-changes. There are also significant reductions after filtering out commits that do not change classes or that only change testing classes (preprocessing task #2) and after merging commits related to the same maintenance issue (preprocessing task #3). Finally, a reduction affecting 3% of the Geronimo's commits, 4% of the Camel's commits, and nearly 1% of the commits of the other systems is observed after 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 3.1, to remove edges with unitary weights. Table 4 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 4. 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
Camel	3,033	42,336	0.01	1,498	15,404	0.01

By observing the results in Table 4, 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 [20]. Second, most edges in the initial co-change graphs have weight equal to one (more precisely, around 81%, 70%, 66%, and 64% of the edges for Geronimo, Lucene, JTD Core, and Camel graphs, respectively). Therefore, these edges connect classes that changed together in just one commit and for this reason they were removed after the post-processing task. As result, the number of vertices after post-processing is reduced to 33.1% (Geronimo), 50.5% (Lucene), 68.5% (JDT Core), and 49.4% (Camel) of their initial value.

4.3 Co-Change Clustering

We executed the Chameleon algorithm having as input the co-change graphs created for each system (after applying the pre-processing and post-processing filters).[7] Table 5 shows the value of M that generated the best clusters, according to the clustering selection criteria defined in Section 3.2. 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 less 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 5. 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
Camel	251	130	47	0.06

For example, for Geronimo, 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 4.1. 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 systems is concentrated in few classes.

Table 6 shows standard descriptive statistics measurements regarding the size of the extracted co-change clusters, in terms of number of classes. The extracted clusters have 8.8 ± 4.7 classes, 11.7 ± 7.0 classes, 14 ± 10.4 classes, and 10.2 ± 11.48 (average $\pm$ standard deviation) in the Geronimo, Lucene, JDT Core, and Camel systems, respectively. Moreover, the biggest cluster has a considerable number of classes: 20 classes (Geronimo), 27 classes (Lucene), 43 classes (JDT Core), and 74 classes (Camel).

Table 7 presents standard descriptive statistics measurements regarding the density of the extracted co-change clusters. The clusters have density of 0.80 ± 0.24 (Geronimo), 0.68 ± 0.25 (Lucene), 0.54 ± 0.29 (JDT Core), and 0.77 ± 0.25 (Camel). The median density is 0.90 (Geronimo), 0.71 (Lucene), 0.49 (JDT Core), and 0.83 (Camel). Therefore, although co-change graphs are sparse graphs,

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

Table 6. 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	14.0	10.4
Camel	4	74	10.2	11.5

the results in Table 7 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.

Table 7. 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
Camel	0.16	1.0	0.77	0.25	0.83

Table 8 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, Lucene, and Camel. Whereas, in the JDT Core it is about four.

Table 8. Cluster average edges' weight

System	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
Camel	2	5	2.6	0.8	2.3

5 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 [14], which are typically used to compare two partitions P and Q of the entities of 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 1 means that the cluster q dominates the packages that it touches, i.e., it is well-encapsulated in such packages. In contrast, 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.

$$spread(q, P) = \sum_{p_i \in P} \begin{cases} 1, touch(q, p_i) > 0 \\ 0, touch(q, p_i) = 0 \end{cases}$$

Tables 9 and 10 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 and Camel have a higher focus than in Lucene and JDT Core. For example, the median focus in Geronimo and Camel is 1.00, against 0.55 and 0.30 in Lucene and JDT Core, respectively. Regarding spread, Camel has a lower value than the others, on average the spread is 2.96 against 3.50 (Geronimo), 3.35 (Lucene), and 3.83 (JDT Core). Figure 5 shows a scatterplot with the values of focus (horizontal axis) and spread (vertical axis) for each co-change cluster. In Geronimo and Camel, 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.

Table 9. 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
Camel	0.23	1.00	0.87	0.20	1.00

Table 10. 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
Camel	1	13	2.96	2.52	2	1

In the following sections, we analyze examples of well-encapsulated and crosscutting clusters, using distribution maps.[8] Section 5.1 emphasizes well-encapsulated clusters, since they are common in Geronimo. On the other hand, Section 5.2 emphasizes crosscutting concerns, which are most common in Lucene. Section 5.3 reports on both types of clusters in Eclipse JDT. Finally, Section 5.4 emphasizes well-encapsulated and partially encapsulated clusters, since they are common in Camel.

5.1 Distribution Map for Geronimo

Figure 6 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:

[8] To extract and visualize distribution maps, we used the Topic Viewer tool [42], available at `https://code.google.com/p/topic-viewer`).

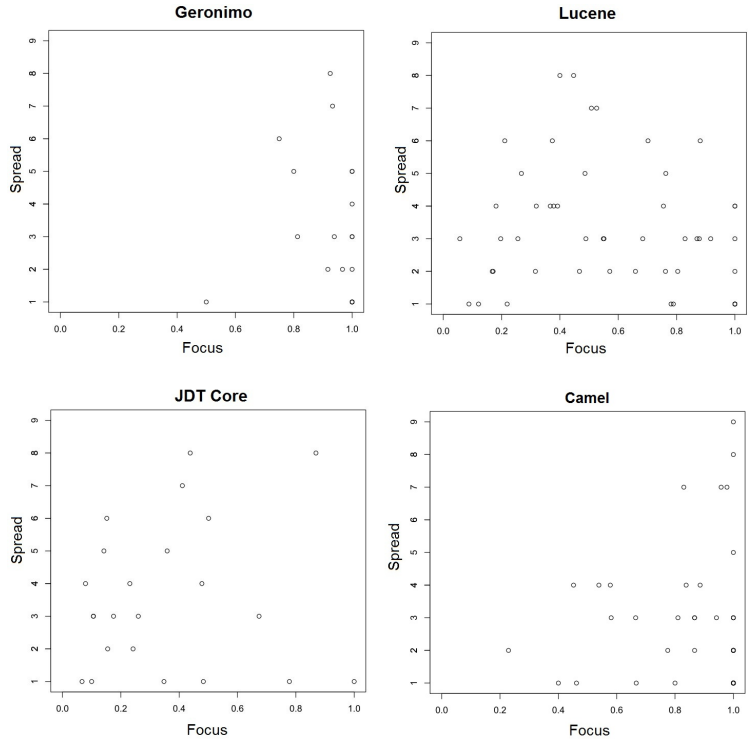

Fig. 5. Focus versus Spread

– *Clusters 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 2 in the map, column 3). 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 2 in the map, column 4) and Cluster 11 (package `security.remoting.jmx`, line 6, column 3).
– *Clusters 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 2, columns 1 and 2; line 6, column 6; and line 8, 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

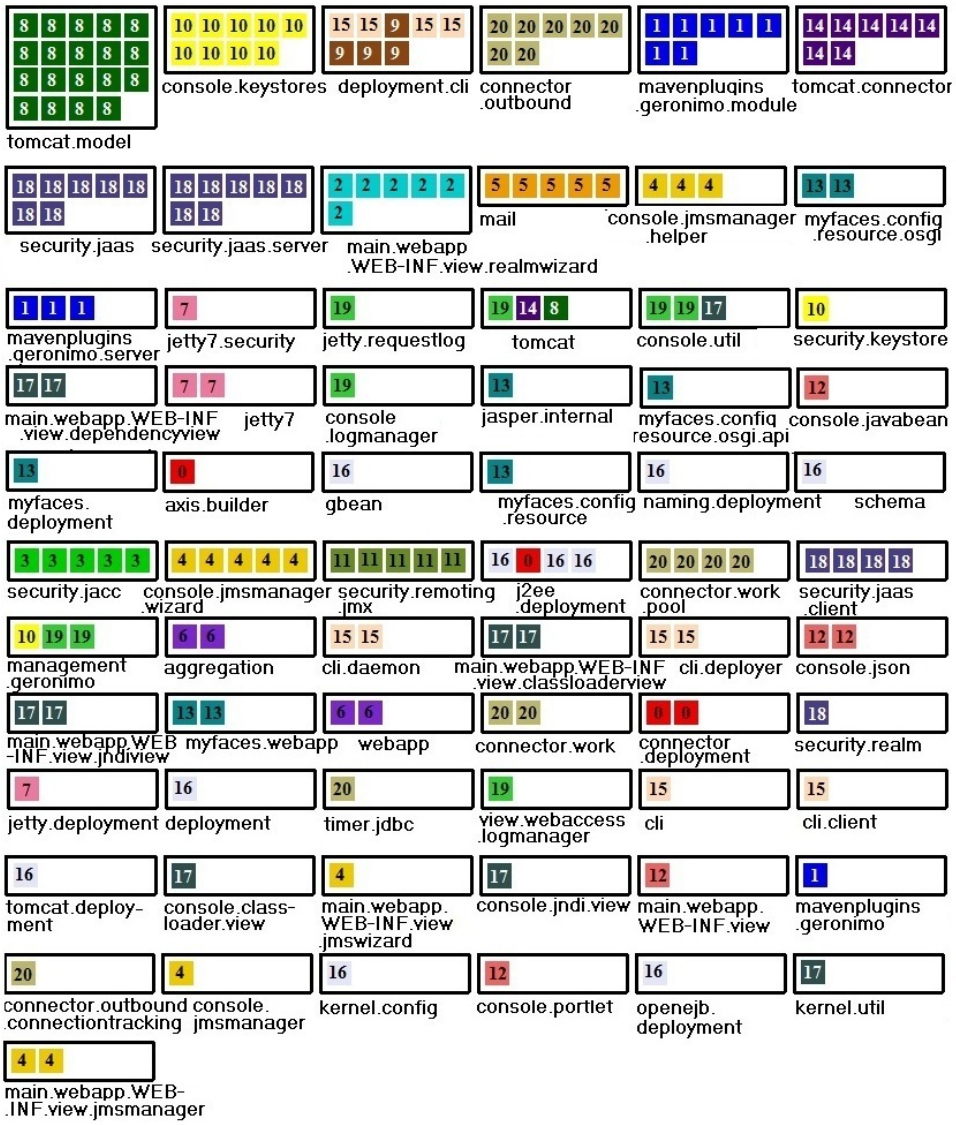

Fig. 6. Distribution map for Geronimo

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

6, column 5; line 8, column 4; line 9, column 3; line 11 and column 1). These packages implement EJB connectors for message exchange.

- *Clusters partially encapsulated (focus ≈ 1.0) and touching classes in other packages (spread > 1).*[9] As an example, 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 3, column 4). 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 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 3, column 4). This "tentacle" in a single class from another package suggests again an instability in the configuration interface provided by the underlying web server.

5.2 Distribution Map for Lucene

We selected for analysis clusters that are *crosscutting* (focus ≈ 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 7 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 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 is less harmful 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

[9] These clusters are called octopus, because they have a body centered on a single package and tentacles in other packages [14].

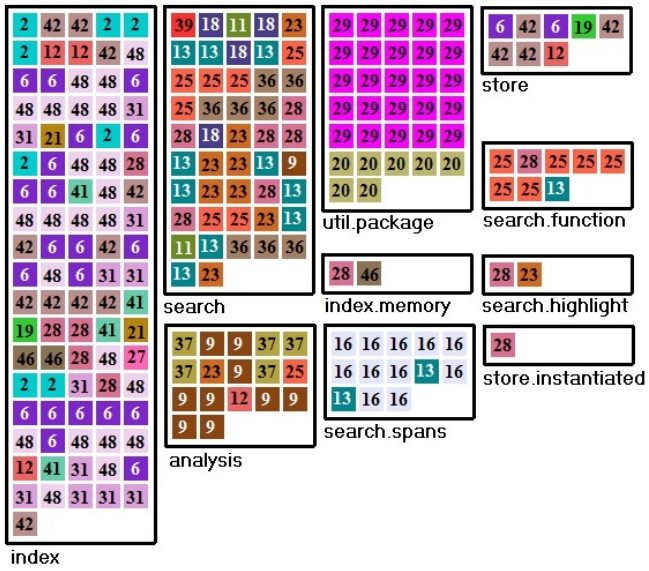

Fig. 7. Part of the Distribution map for Lucene

important services in Lucene, like indexing, searching, and storing. Therefore, as in the case of Cluster 12, this crosscutting behavior 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 11 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 non-functional concerns (8%). Moreover, changes motivated by refactorings (32.5%) are more common than changes in non-functional concerns.

Table 11. Maintenance issues in Cluster 28

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

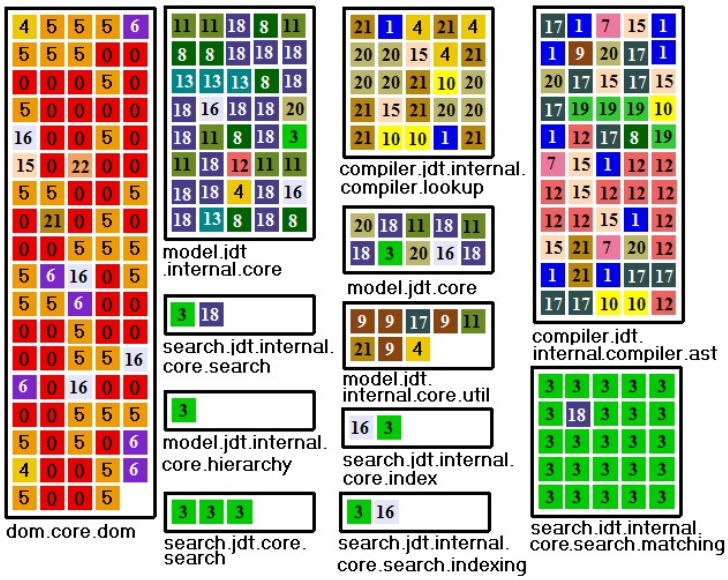

Fig. 8. Part of the Distribution map for JDT Core

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

– *Clusters confined in a single package (spread = 1).* Although restricted to a single package, these clusters do not dominate the colors in this package. But if merged in a single cluster, they dominate their package. As an 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, a refactoring that splits this package in sub-packages can be considered, in order to improve the focus of the respective clusters.

5.3 Distribution Map for JDT Core

Figure 8 shows the distribution map for JDT Core. We selected three distinct types of clusters for analysis: a *crosscutting* cluster (focus ≈ 0.0 and spread >= 3), clusters confined in a single package, 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: `internal.compiler.lookup`, `internal.core`, `core.dom`, 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

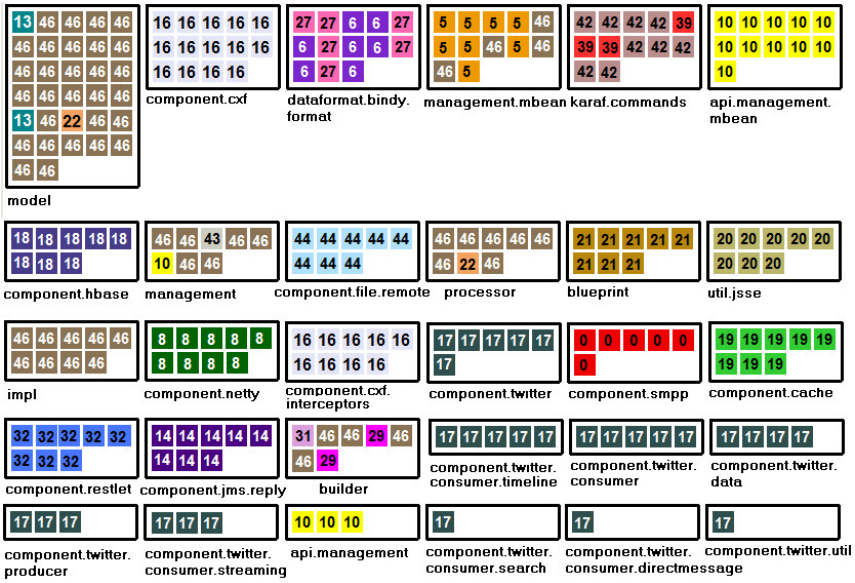

Fig. 9. Distribution map for Camel

packages providing 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 confined in a single package (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).
- *Clusters partially encapsulated (focus ≈ 1.0) and 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. Therefore, a maintenance in this of the cluster usually has a ripple effect in classes like that.

5.4 Distribution Map for Camel

Figure 9 shows the distribution map for Camel. We selected two types of cluster patterns for analysis: a well-encapsulated cluster (focus = 1.0) and a partially encapsulated cluster (focus ≈ 1.0).

- *Clusters encapsulated (focus = 1.0) in a single package (spread = 1).* Twelve clusters have this behavior. As an example, we have Cluster 0, which dominates the co-change classes in the package `component.smpp` (line 3 in the

map, column 5). This package provides access to a short message service. Therefore, since it implements a particular functional concern, the maintenance is confined in the package.

- *Clusters encapsulated (focus = 1.0) in more than one package (spread > 1).* We counted 13 clusters with this behavior. As an example, we have Cluster 17 (*spread = 9*), which touches all co-change classes in the following packages: `component.twitter`, `component.twitter.data`, `component.twitter.producer`, `component.twitter.consumer.streaming`, `component.twitter.consumer.directmessage`, `component.twitter.consumer.search`, `component.twitter.util`, `component.twitter.consumer`, and `component.twitter.consumer.timeline`. As revealed by their names, these packages are related to the Twitter API. Therefore, these 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 code in sub-packages.
- *Clusters partially encapsulated (focus ≈ 1.0) and touching classes in other packages (spread > 1).* We counted 11 clusters with this behavior. As an example, we have Cluster 10 (*focus* = 0.94, *spread* = 3), which dominates the co-change classes in the packages `api.management` and `api.management.mbean` (line 5, column 3; line 1, column 6 in the map), but also touches the class `MBeanInfoAssembler` from package `management` (line 2, columns 2). This class is responsible for reading details from different annotations, such as `ManagedResource` and `ManagedAttribute`. This co-change cluster shows that maintenance in `api.management` and `api.management.mbean` packages sometimes have a ripple effect on this class, or vice-versa.

6 Semantic Similarity Analysis

The previous section showed that the package structure of Geronimo and Camel 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 associated to a specific cluster in order to improve our understanding of the clusters' meaning. We hypothesize 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 an issue as a document and the collection of documents is obtained from the collection of issues related to a cluster. We use Latent Semantic Analysis - LSA [13] 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 [36], [35].

6.1 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 define 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 are 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 [27], did not improved the quality of the similarity matrix.

6.2 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 related to a cluster. Figure 10 shows examples of similarity within specific clusters. We show for each system the two best clusters in terms of similarity in the left, and the two clusters with several pairs of issues with low similarity in the right. The white cells represent issues that 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.

6.3 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

$$similar(i,j) = \begin{cases} 0, \text{ if } LSA_Cosine(i,j) < SIM_THRS \\ 1, \text{ Otherwise} \end{cases}$$

n = number of issues related to cluster c

$SIM_THRS = 0.4$

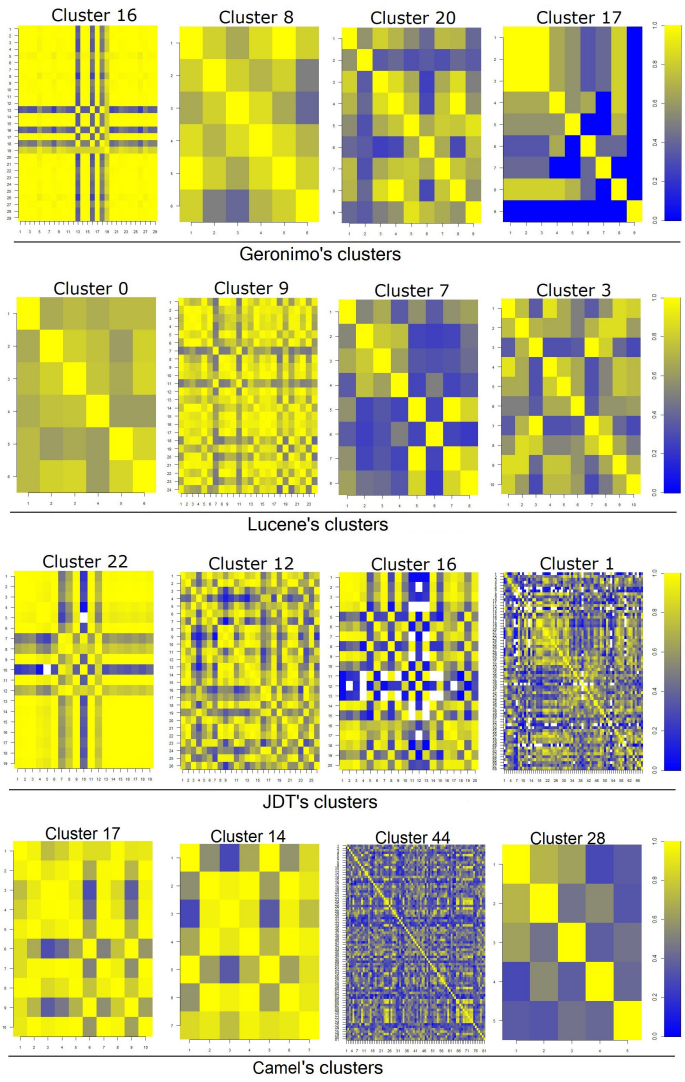

Fig. 10. Examples of heat maps for similarity of issues

The meaning of the *similarity score* of a cluster is defined upon the percentage of similar pair of issues. Therefore, 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 defined 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 is not free of imprecision. Similar to our study, Poshyvanyk and Marcus [36] 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. For this reason, our more conservative approach seems to be adequate. Moreover, because our goal is to provide 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.

Figure 11 shows the distribution of score values for Geronimo's, Lucene's, JDT's, and Camel's clusters. We can observe that the systems' clusters follow a similar pattern of scoring, with 100% (for Lucene, JDT, and Camel) and more than 90% (for Geronimo) of clusters having more than half pairs of issues similar to each other. Only two Camel's clusters have score less than 50% of similarity. Interestingly, one of these two clusters have 226 issue reports and their similarity is very low.

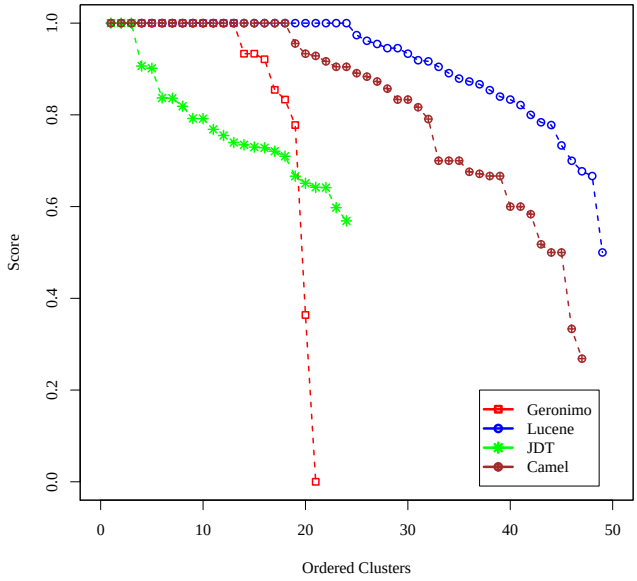

Fig. 11. Distribution of the clusters' score

6.4 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 12 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. For Camel, we observed a moderate negative correlation between spread and score. Similar to Geronimo, the higher is the number of similar issues in a cluster, the higher is the capacity of the cluster to enfold a whole package in Camel. Considering that the clusters of the analyzed systems follow a similar pattern of similarity, this result suggests that the similarity between co-changes induces different properties in the clusters, either in spread or in focus.

Table 12. Correlation between score, focus and spread of clusters for Geronimo, Lucene, JDT Core, and Camel

Correlation Coefficient *p-value*	**Score Geronimo**	**Score Lucene**	**Score JDT**	**Score Camel**
Focus	0.264	0.308	−0.015	0.067
	0.131	0.016	0.473	0.327
Spread	−0.720	−0.178	−0.304	−0.337
	0.000	0.111	0.074	0.010

7 Discussion

7.1 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 and Camel's clusters, 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 improve modularity. Particularly, there are two patterns of clusters that may suggest modularity flaws. The first pattern denotes clusters with crosscutting behavior (focus $\approx$ 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, Lucene, and Camel, we detected four, five, and eleven clusters related to this pattern, respectively.

Nonetheless, we have no evidence that the proposed co-change clusters can 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. However, 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 [22]).

7.2 Clustering vs Association Rules Mining

Our approach is centered on the Chameleon hierarchical clustering algorithm, which is an algorithm designed to handle sparse graphs [20]. In our case studies, for example, the co-change graphs have densities ranging from 1% (Camel) to 4% (Eclipse JDT Core).

Particularly, in traditional clustering algorithms, like K-Means [26], 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. However, it is possible that some vertices are not allocated to any cluster. This may happen when some vertices do not share any edge with the rest of the vertices or when a vertice share edges to other vertices that belong to different clusters with no significant discrepancy among weights.

We also performed an algorithm to detect communities (clusters) [6], provided by the Gephi Tool[10], to compare with our co-change clusters. Similar to K-Means, this algorithm also allocates each vertice to a particular cluster. Thus, vertices with few or even one edge are assigned to a cluster leading these clusters to have lower density than Chameleon's. Nonetheless, we could define a pos-processing task to prune such vertices from clusters detected by the community algorithm to increase their densities. In spite of clusters with lower density, the result suggested the same pattern we presented in this paper, e.g., for Geronimo the package structure is adherent to the cluster structure and for Lucene, there is not a clear adherence between co-change clusters and packages.

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

[10] http://gephi.github.io/

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 on Lucene's pre-processed dataset. 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. Another attempt to reduce the number of rules is to select the more interesting ones. There are several alternative measures available to complement the support and confidence measures [34], [31]. One of the most well-known is the lift [8]. However, if the rules present high values of lift, it is very hard to make a precise selection. Another way to reduce the number of rules is to combine association rules and clustering [25].

7.3 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, JDT Core, and Camel 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 [17]. 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 that we followed the approach commonly used in other studies that map issues to commits [12],[51], [11], [10]. 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 maintenance activity during 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 has 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, 1,004

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. However, 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 can 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 are achieved in all systems. Moreover, we observed that variations in the parameters' values would affect the results for all systems in a similar way.

8 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.

8.1 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 [38, 39]. 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 [16] and JQuery [18] are other tools that rely on lexical or logic queries to find and document code fragments related to a certain concern. ConcernMapper [40] 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. By contrast, in the approach proposed in this paper, the elements and relationships are obtained by mining the version history.

8.2 Co-change Mining

Kouroshfar investigated the impact of co-change dispersion on software quality [24]. His results revealed that co-changes localized in the same subsystem involve fewer bugs than co-changes crosscutting distinct subsystems.

Zimmermann et al. propose an approach that uses association rule mining on version histories to suggest possible future changes [52]. 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 be changed too). Differently from their approach, we use co-change graphs to retrieve clusters semantically related to a target system's concern. Robillard and Barthélémy evaluated on seven open-source systems whether change clusters can support developers in their investigation of a software [37]. Similar to one of our pre-processing tasks, their approach discards commit transactions that contain too few or too many changed elements before clustering. Furthermore, in order to select clusters, like we do by using the *MIN_CLUSTER_SZ* threshold, they retrieved clusters that matched to a query. A quantitative analysis revealed that less than one in five tasks overlapped with a change cluster. The qualitative analysis of the recommended clusters showed that only 13% of the recommendation for applicable change tasks are useful. However, our goal is not to recommend future changes, but to assess modularity, using distribution maps to compare and contrast co-change clusters with packages.

Beyer and Noack introduce 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. Vanya et al. used co-change clusters to support the partitioning of system, reducing the coupling between its parts [49]. Their approach detects co-change clusters in which a group of files from one part of the system often changes with a group from another part. After the clustering step, they prune clusters containing files from the same part of the system. However, our central goal is not directly related with improving the visualization of co-change clusters as proposed by Beyer and Noack, but on using them to assess modularity. Finally, we do not prune clusters, such as Vanya et al., but we use them to visualize and to understand the system' package decomposition.

Oliva et al. mine version histories to extract logical dependencies between software artifacts to identify their origins [30]. 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 cluster all classes in a system, since their original goal is 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. present an approach that combines conceptual and evolutionary couplings for impact analysis in source code [19], using information retrieval and version history mining techniques. Gethers et al. propose an impact analysis that adapts to the specific maintenance scenario using information retrieval, historical

data mining, and dynamic analysis techniques [15]. However, they do not use maintenance issues reports to discard noisy commits.

Palomba et al. propose HIST, an approach that uses association rule mining on version histories to detect code smells [32]. For each smell, they define a heuristics that relies on association rules for detecting bad smells. However, our goal is not to detect code smells but to assess modularity using co-change clusters.

A recent study by Negara et al. reveals that the use of data from version history presents many threats when investigating source code properties [29]. In this work, we proposed five pre-processing tasks and one post-processing task to tackle some of such threats.

8.3 Aspect Mining

Breu and Zimmermann proposed an approach (HAM) based on version history to detect crosscutting concerns in an object-oriented program to guide its migration to an aspect-oriented program [7]. They define the notion of transaction, which is the set of methods inserted by the developer to complete a single development task. They also consider 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 are 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. propose 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 concerns. 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.

9 Conclusion

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 four real software systems, Geronimo, Lucene, JDT Core, and Camel. 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 are 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 are 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 are specially useful 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] and clustering algorithms based on density property. 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 [21] and CIDE+ tools [48].

Acknowledgments

This work is supported by FAPEMIG, CAPES, and CNPq.

References

1. Adams, B., Jiang, Z.M., Hassan, A.E.: Identifying crosscutting concerns using historical code changes. In: 32nd International Conference on Software Engineering (ICSE). pp. 305–314 (2010)
2. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: 20th International Conference on Very Large Data Bases (VLDB). pp. 487–499 (1994)
3. Baldwin, C.Y., Clark, K.B.: Design Rules: The Power of Modularity. MIT Press (2003)
4. Beck, F., Diehl, S.: Evaluating the impact of software evolution on software clustering. In: 17th Working Conference on Reverse Engineering (WCRE). pp. 99–108 (2010)
5. Beyer, D., Noack, A.: Clustering software artifacts based on frequent common changes. In: 13th International Workshop on Program Comprehension (IWPC). pp. 259–268 (2005)
6. Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment 2008(10), P10008 (2008)
7. Breu, S., Zimmermann, T.: Mining aspects from version history. In: 21st Automated Software Engineering Conference (ASE). pp. 221–230 (2006)
8. Brin, S., Motwani, R., Ullman, J.D., Tsur, S.: Dynamic itemset counting and implication rules for market basket data. In: International Conference on Management of Data (SIGMOD). pp. 255–264 (1997)
9. Chidamber, S., Kemerer, C.: Towards a metrics suite for object oriented design. In: 6th Object-oriented Programming Systems, Languages, and Applications Conference (OOPSLA). pp. 197–211 (1991)

10. Couto, C., Pires, P., Valente, M.T., Bigonha, R., Anquetil, N.: Predicting software defects with causality tests. Journal of Systems and Software pp. 24–41 (2014)
11. Couto, C., Silva, C., Valente, M.T., Bigonha, R., Anquetil, N.: Uncovering causal relationships between software metrics and bugs. In: 16th European Conference on Software Maintenance and Reengineering (CSMR). pp. 223–232 (2012)
12. D'Ambros, M., Lanza, M., Robbes, R.: An extensive comparison of bug prediction approaches. In: 7th Working Conference on Mining Software Repositories (MSR). pp. 31–41 (2010)
13. Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., Harshman, R.: Indexing by latent semantic analysis. Journal of the American Society for Information Science, 41, 391–407 (1990)
14. Ducasse, S., Gîrba, T., Kuhn, A.: Distribution map. In: 22nd IEEE International Conference on Software Maintenance (ICSM). pp. 203–212 (2006)
15. Gethers, M., Kagdi, H., Dit, B., Poshyvanyk, D.: An adaptive approach to impact analysis from change requests to source code. In: 26th Automated Software Engineering Conference (ASE). pp. 540–543 (2011)
16. Griswold, W.G., Yuan, J.J., Kato, Y.: Exploiting the map metaphor in a tool for software evolution. In: 23rd International Conference on Software Engineering (ICSE). pp. 265–274 (2001)
17. Herzing, K., Zeller, A.: The impact of tangled code changes. In: 10th Working Conference on Mining Software Repositories (MSR). pp. 121–130 (2013)
18. Janzen, D., Volder, K.D.: Navigating and querying code without getting lost. In: 2nd International Conference on Aspect-oriented Software Development (AOSD). pp. 178–187 (2003)
19. Kagdi, H., Gethers, M., Poshyvanyk, D.: Integrating conceptual and logical couplings for change impact analysis in software. Empirical Software Engineering (EMSE) 18(5), 933–969 (2013)
20. Karypis, G., Han, E.H.S., Kumar, V.: Chameleon: hierarchical clustering using dynamic modeling. Computer 32(8), 68–75 (1999)
21. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: 30th International Conference on Software Engineering (ICSE). pp. 311–320 (2008)
22. Kersten, M., Murphy, G.C.: Using task context to improve programmer productivity. In: 14th International Symposium on Foundations of Software Engineering (FSE). pp. 1–11 (2006)
23. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: 11th European Conference on Object-Oriented Programming (ECOOP). LNCS, vol. 1241, pp. 220–242. Springer Verlag (1997)
24. Kouroshfar, E.: Studying the effect of co-change dispersion on software quality. In: 35th International Conference on Software Engineering (ICSE). pp. 1450–1452 (2013)
25. Lent, B., Swami, A.N., Widom, J.: Clustering association rules. In: 13th International Conference on Data Engineering (ICDE). pp. 220–231 (1997)
26. MacQueen, J.B.: Some methods for classification and analysis of multivariate observations. In: 5th Berkeley Symposium on Mathematical Statistics and Probability. pp. 281–297 (1967)
27. Manning, C.D., Raghavan, P., Schtze, H.: Introduction to Information Retrieval. Cambridge University Press (2008)
28. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall (2000)

29. Negara, S., Vakilian, M., Chen, N., Johnson, R.E., Dig, D.: Is it dangerous to use version control histories to study source code evolution? In: 26th European conference on Object-Oriented Programming (ECOOP). pp. 79–103 (2012)
30. Oliva, G.A., Santana, F.W., Gerosa, M.A., de Souza, C.R.B.: 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). pp. 31–40 (2011)
31. Omiecinski, E.: Alternative interest measures for mining associations in databases. IEEE Transactions on Knowledge and Data Engineering 15(1), 57–69 (2003)
32. Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., de Lucia, A., Poshyvanyk, D.: Detecting bad smells in source code using change history information. In: 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 268–278 (2013)
33. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
34. Piatetsky-Shapiro, G.: Discovery, analysis and presentation of strong rules. In: Knowledge Discovery in Databases, pp. 229–248 (1991)
35. Poshyvanyk, D., Marcus, A.: Using information retrieval to support design of incremental change of software. In: 22th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 563–566 (2007)
36. Poshyvanyk, D., Marcus, A.: Measuring the semantic similarity of comments in bug reports. In: 1st International ICPC Workshop on Semantic Technologies in System Maintenance (STSM). pp. 265–280 (2008)
37. Robillard, M.P., Dagenais, B.: Recommending change clusters to support software investigation: An empirical study. Journal of Software Maintenance and Evolution: Research and Practice 22(3), 143–164 (2010)
38. Robillard, M.P., Murphy, G.C.: Concern graphs: finding and describing concerns using structural program dependencies. In: 24th International Conference on Software Engineering (ICSE). pp. 406–416 (2002)
39. Robillard, M.P., Murphy, G.C.: Representing concerns in source code. ACM Transactions on Software Engineering and Methodology 16(1), 1–38 (2007)
40. Robillard, M.P., Weigand-Warr, F.: ConcernMapper: simple view-based separation of scattered concerns. In: OOPSLA Workshop on Eclipse Technology eXchange. pp. 65–69 (2005)
41. Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Communications of the ACM 18(11), 613–620 (1975)
42. Santos, G., Valente, M.T., Anquetil, N.: Remodularization analysis using semantic clustering. In: 1st CSMR-WCRE Software Evolution Week. pp. 224–233 (2014)
43. Silva, L., Valente, M.T., Maia, M.: Assessing modularity using co-change clusters. In: 13th International Conference on Modularity. pp. 49–60 (2014)
44. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: 2nd Working Conference on Mining Software Repositories (MSR). pp. 1–5 (2005)
45. Stevens, W.P., Myers, G.J., Constantine, L.L.: Structured design. IBM Systems Journal 13(2), 115–139 (Jun 1974)
46. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: Qualitas corpus: A curated collection of Java code for empirical studies. In: Asia Pacific Software Engineering Conference (APSEC). pp. 336–345 (2010)
47. Terra, R., Miranda, L.F., Valente, M.T., Bigonha, R.S.: Qualitas.class corpus: A compiled version of the Qualitas Corpus. Software Engineering Notes pp. 1–4 (2013)

48. Valente, M., Borges, V., Passos, L.: A semi-automatic approach for extracting software product lines. IEEE Transactions on Software Engineering 38(4), 737–754 (2012)
49. Vanya, A., Hofland, L., Klusener, S., van de Laar, P., van Vliet, H.: Assessing software archives with evolutionary clusters. In: 16th IEEE International Conference on Program Comprehension (ICPC). pp. 192–201 (2008)
50. Walker, R.J., Rawal, S., Sillito, J.: Do crosscutting concerns cause modularity problems? In: 20th International Symposium on the Foundations of Software Engineering (FSE). pp. 1–11 (2012)
51. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for Eclipse. In: 3rd International Workshop on Predictor Models in Software Engineering. pp. 9– (2007)
52. Zimmermann, T., Weissgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. IEEE Transactions on Software Engineering 31(6), 429–445 (2005)