Co-Change Patterns: A Large Scale Empirical Study

Luciana L. Silva[a,*], Marco Tulio Valente[a], Marcelo de Almeida Maia[b]

[a]Department of Computer Science, UFMG, Brazil
[b]Faculty of Computer Science, UFU, Brazil

Abstract

Co-Change Clustering is a modularity assessment technique that reveals how often changes are localized in modules and whether a change propagation represents design problems. This technique is centered on co-change clusters, which are highly inter-related source code files considering co-change relations. In this paper, we conduct a series of empirical analysis in a large corpus of 133 popular software projects on GitHub. We describe six co-change patterns by projecting them over the directory structure. We mine 1,802 co-change clusters and 1,719 co-change clusters (95%) are covered by the six co-change patterns. In this study, we aim to answer two central questions: (i) Are co-change patterns detected in different programming languages? (ii) How do different co-change patterns relate to rippling, activity density, ownership, and team diversity on clusters? We conclude that Encapsulated and Well-Confined clusters (Wrapped) implement well-defined and confined concerns. Octopus clusters are proportionally numerous regarding to other patterns. They relate significantly with ripple effect, activity, ownership, and diversity in development teams. Although Crosscutting are scattered over directories, they implement well-defined concerns. Despite they present higher activity compared to Wrapped clusters, it is not necessarily easy to get rid of them, suggesting that support tools may play a crucial role.

Keywords: Modularity, co-change clusters, co-change patterns

1. Introduction

Parnas developed the criteria that *"modules should hide decisions or decisions that are likely to change* (Parnas, 1972). In addition, according to Parnas, a module represents a responsibility assignment. However, the initial planned modules may suffer changes overtime to adapt to their new responsibilities, otherwise their modularity tends to decay. During software development, changes are performed constantly in tasks related to new features, code refactoring, and bug fixing. In a modular software, when those tasks are required, they should change a single module with minimal—if possible none—impact in other modules (Aggarwal and Singh, 2005). In contrast, improper modularization may cause ripple effects during software maintenance. For such situation, it would be beneficial whether the system expert could semi-automatically analyze modularity by retrieving set of classes that usually change together to compare and contrast co-change clusters with directory structure.

The importance of modular systems has motivated researchers and practitioners to investigate different dimensions to assess system modularity. Despite modularity being an essential principle in software development, effective approaches to assess modularity still remain an open problem. Typically, the standard approach is based on the analysis of structural measures, e.g., coupling and cohesion (Chidamber and Kemerer, 1991; Stevens et al., 1974). Over the years, several alternative attempts have been proposed, such as semantic approaches, which analyze the source code vocabulary using Information Retrieval techniques (Santos et al., 2014; Maletic and Marcus, 2000), co-change approaches, which mine historical data to detect software artifacts that usually change together (Zimmermann et al., 2007; Alali et al., 2013), or hybrid approaches, which combine these types of information (Kagdi et al., 2013; Bavota et al., 2014).

Recently, we proposed the Co-Change Clustering technique to assess modularity by capturing logical modules from history of software changes (Silva et al., 2014, 2015a). Co-change clusters consist of source code artifacts that frequently change together between themselves but not with others artifacts in different clusters. Later, we reported a study with experts of six object-oriented systems to investigate the developers' perception of co-change clusters (Silva et al., 2015b). These clusters were classified in three patterns to represent common instances of co-change clusters, regarding their projection over the directory structure: Encapsulated Clusters (clusters that match the directory structure, i.e., they dominate all co-change classes in the directory structures they touch), Crosscutting Clusters (clusters whose co-change classes are scattered across several directory structures), and Octopus Clusters (most co-change classes are confined in one directory structure with some arms—or "tentacles"—in others). Our proposed co-change patterns could cover 52% of the 102 mined clusters. Although unitary changes are preferable than co-changes, indeed, those co-change pat-

*Corresponding author at Departamento de Ciência da Computação, Av. Antônio Carlos, 6627 - Pampulha CEP: 31270-010, UFMG, Belo Horizonte, Brazil. Tel.: +55 31 3409-5865.

Email addresses:
luciana.lourdes@gmail.com,luciana.lourdes.silva@ifmg.edu.br (Luciana L. Silva), mtov@dcc.ufmg.br (Marco Tulio Valente), marcelo.maia@ufu.br (Marcelo de Almeida Maia)

terns with propagation and scattering revealed to have different perspectives on the impact they may cause in software maintenance. In summary, our first results indicate that: (i) Encapsulated Clusters tend to be more controllable; (ii) around 50% of the Crosscutting Clusters were associated to design anomalies; (iii) Octopus Clusters represent expected class distributions, which are difficult to implement in an encapsulated way.

Nonetheless, we did not evaluate whether programming languages, application domains, thresholds, and commit density (number of commits divided by the number of source code files) impact co-change pattern's detection. We also did not analyze whether co-change patterns relate to ripple-effect, activity density, ownership, and team diversity on clusters.

In this paper, we extend our previous work to answer these aforementioned open questions in five directions:

1. We conduct a new study in a large corpus of 133 popular projects hosted in GitHub. We consider projects in different languages (C/C++, Java, PHP, Ruby, JavaScript, and Python) and application domains.
2. We investigate the threshold used to group commits (applied by the same developer in a period of time during the data preprocessing) and the process of co-change cluster extraction. In this analysis, we intend to evaluate the effectiveness of co-change clusters.
3. We propose three new co-change patterns (Section 3): (i) Well-Confined Clusters (clusters which touch a single directory structure but do not dominate it); (ii) Black Sheep Clusters (similar to Crosscutting, however, they touch very few code files in each one directory structure); and (iii) Squid Clusters (similar to Octopus, but they have smaller bodies and arms). We mine 1,802 co-change clusters from version histories of such projects, which were then categorized in six patterns regarding their projection over the directory structure. From the initially computed clusters, 1,719 co-change clusters (95%) are covered by the proposed co-change patterns.
4. We conduct a series of statistical analysis to evaluate the categorized co-change clusters on the 133 projects (Section 5.4). This study aims to answer two central questions: (i) Are co-change patterns detected in different programming languages? (ii) How do different co-change patterns relate to rippling, activity density, ownership, and team diversity on clusters?
5. We conduct a qualitative analysis by investigating underlying natural language topics in commit messages on classified co-change clusters. Specifically, we analyze clusters which have massively changed source code files to understand the rationale behind the proposed co-change patterns and how they evolve overtime.

The remainder of this paper is organized as follows. First, we summarize our technique for extracting co-change clusters (Section 2) and present the co-change patterns used in this paper (Section 3). Then, we describe our study design, including the research questions that guide this work in Section 4. In Section 5 we present and discuss our achieved results. In Section 6 we conduct a qualitative and semantic analysis and Section **??** we discuss our findings. We discuss threats to validity in Section 7 and related work in Section 8. Finally, we conclude the paper in Section 9.

2. Co-Change Clustering

The ultimate goal of our technique is to support developers on modularity assessment using co-change relations. The technique relies on historical information to generate co-change graphs and then mine co-change clusters. As illustrated in Figure 1, we propose three phases to extract co-change clusters. In the first phase, we extract commit transactions and apply preprocessing steps to build co-change graphs. After that, a post-processing phase is applied on co-change graphs to prune edges with small weights. Finally, in the last phase, the co-change graph is clustered several times to selected the best clustering. A detailed description of this section can be found in previous work (Silva et al., 2014, 2015a).

2.1. Co-Change Graph

Beyer and Noack (2005) proposed co-change graphs, an abstraction to represent commit transactions from version control system (VCS). Conceptually, a co-change graph is an undirected and weighted graph {V, E}, where V is a set of source code files and E is a set of edges. If there is an edge between two vertices (source code files) F_i and F_j, then at least one commit in the VCS changes F_i and F_j, where $i \neq j$. Finally, we conduct some experiments with relative edge weights to define the weighting measure. We concluded that co-change modifications are usually much less frequent than single changes in a source code file. For example, in our previous work (Silva et al., 2014, 2015a), we extract co-change clusters for Lucene project[1] to present the concept of co-change clustering. We observe that the maximal edges' weight is seven and the maximum size is 27. For this reason, we do not adopt a relative edge weights on co-change graphs. Instead, the edges' weights represent how many commits changed the connected source code files simultaneously represent.

Before extracting co-change graphs, we perform Phase 1 (Figure 1) to preprocess commits extracted from version history. This phase consists of the following three steps:

1. We discard commits which do not change source code files, e.g., documentation and configuration files. In addition, testing files are eliminated because co-changes between testing files and their respective functional code files do not provide relevant information from the point of view of modularity assessment of the functional code. Testing files could be considered if the goal was overall change impact analysis.
2. We merge commits whose textual descriptions are associated to the same issue-ID in a single commit. Nonetheless, there are a significant number of commits not linked to maintenance issues (Silva et al., 2014; Couto et al., 2014). For

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

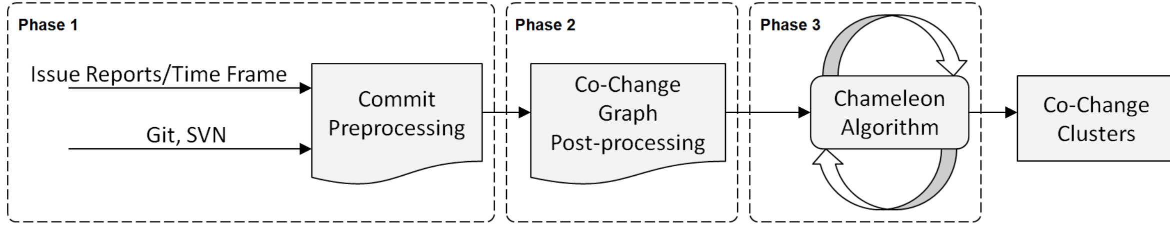

Figure 1: Phases proposed to extract co-change clusters for modularity assessment.

this reason, we merge commits done by the same developer under a time frame. Our goal here is to handle cases when the developer applies sequences of commits associated to the same maintenance task in a short period of time.

3. We remove commits with highly scattered code changes, i.e., commits that change an expressive number of code files in different directories. These commits usually are not related to recurrent maintenance tasks, e.g., rename methods, dead code removal, or comment changes on license agreements. Recent research showed that scattering in commits tends to follow heavy- tailed distributions (Walker et al., 2012), so massively scattering commits cannot be neglected due to the very large deviation between the number of classes changed by them and by the remaining commits in the system. For instance, in Lucene 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. In order, to remove highly scattered commits, a threshold of 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 (Zimmermann et al., 2005; Adams et al., 2010). We adopted a conservative approach working at package level because excessive pruning would remove relevant information, so only a few outliers are removed.

Finally, we compute co-change graphs and apply a post-processing step to pruning edges with weights less than a given support threshold. As our purpose is to generate a co-change graph that models recurrent maintenance tasks, edges assigning small weight are not relevant.

2.2. Co-Change Clusters

Co-change clusters are sets of source code artifacts in a co-change graph that frequently changed together. Co-change clusters are mined automatically using a graph clustering algorithm designed to handle sparse graphs, as is typically the case of co-change graphs (Beyer and Noack, 2005; Ball et al., 1997; Silva et al., 2014). Specifically, we use Chameleon algorithm (Karypis et al., 1999), an agglomerative and hierarchical clustering algorithm robust to sparse graphs. This algorithm consists of two phases. In the first phase, the algorithm divides the vertices into small clusters. After that, the algorithm merges the clusters retrieved in the first phase. Chameleon maximizes the internal similarity ($ISIM$) among vertices into a cluster (i.e., evaluates how close the objects are in a cluster) and minimizes the external similarity (ESIM) of vertices among clusters.

This algorithm requires an input parameter in the first phase to define the initial number of clusters M. For this reason, we run Chameleon several times varying M's value to mine good clusters (as illustrated in Figure 1). In addition, after each clustering of the co-change graph, the clusters smaller than a minimum threshold are discarded and the following clustering quality function is computed:

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

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

The clustering function $coefficient(M)$ combines cluster cohesion (dense subgraphs) and cluster separation (clusters extremely separated among each other). Similarly, Coverage metric Almeida et al. (2011) also combines cohesion and separation concepts. However, it takes into account number of edges while Coefficient considers edges's weight. For this reason, we decided to use Coefficient metric because it searches for co-change relation.

This measure searches for group of code files that may be used as alternative modular views. Therefore, it is not reasonable to consider clusters with a small number of files. A detailed description of these measures is out of the scope of this paper and can be found elsewhere (Silva et al., 2014, 2015a).

3. Co-Change Patterns

In this section, we describe six co-change patterns aiming to represent common instances of co-change clusters. The patterns are defined by projecting clusters over the directory structure, using distribution maps (Ducasse et al., 2006).

Distribution map is a visualization technique can be used to compare two different partitions of entities. In our case, the entities are files, the first partition D is the directory structure, and second partition C is composed by the co-change clusters. Moreover, files are represented as small squares and the partition D (directory structure) groups such squares into large rectangles (directories). In the directory structure, we only consider files that are members of co-change clusters, in order to improve the maps visualization. Finally, partition C (co-change

clusters) is used to color the files (all files in a cluster have the same color).

In addition to visualization, we quantify the *focus* of a given cluster $c \in C$ in relation to the partition D (directory structure), as follows:

$$focus(c, D) = \sum_{d_i \in D} touch(c, d_i) * touch(d_i, c)$$

where

$$touch(d, c) = \frac{|d \cap c|}{|c|}$$

For instance, if all co-change files in a directory are touched by a single cluster, then its focus is one.

There is also a second metric that measures how *spread* is a cluster c in D, i.e., the number of directories touched by c.

We rely on focus and spread measures to detect patterns of co-change clusters which describe different kinds of shapes. The cluster shapes are intended to have a meaning that can be mapped to meaningful situations. The first two clear shapes that we have observed in our semainal study (Silva et al., 2014) are opposite to each other, representing the best and the worst possible situations, the *Encapsulated* and *Crosscutting* shapes, respectively. The *Encapsulated* cluster touches only one directory, and completely touches it, meaning that changes were really encapsulated in that module. The opposite situation is the *Crosscutting* shape where many packages are slight touched by the same cluster, meaning that changes in the files of that cluster typically require changes in several files on different directories, which seems to be an undesirable modular organization.

However, those two shapes represent just a limited part of clusters. We proposed the *Octopus* shape that stay in between and has an intuitive semantics (Silva et al., 2015b). It is mostly, well-encapsulated but still have just a few touching points in other packages. However, with those patterns only 52% of clusters could be classified. So, in this study, we define other shapes for clusters aiming at categorizing most of them. Following, we define thresholds to capture shapes that could be clearly distinguished between themselves. Obviously, since it is a matter of design, one could design other set of shapes, or include another shape in our proposed set. Following we present our proposed design for patterns of cluster shapes aims at being sufficiently discriminative, and that those patterns can be mapped to a few metaphors that covers a significant portion of the found clusters.

Encapsulated: Conceptually, the cluster q dominates entirely the directories it touches. A co-change cluster q is categorized as *Encapsulated* if:

$$Encapsulated(q), \text{if } focus(q) == 1.0$$

Figure 2 shows two examples of Encapsulated Clusters[2]. All classes in cluster yellow are in the same directory and this directory is only touched by this cluster. Similarly, the cluster green dominates the directory it touches.

[2]All examples presented in this section are real instances of co-change clusters, extracted from projects hosted in GitHub used in this paper, see Section 4

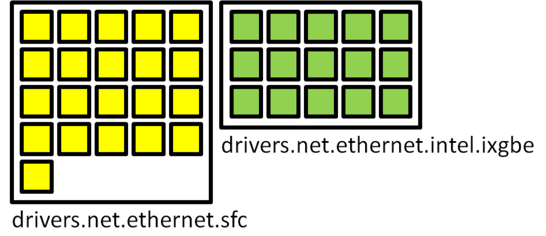

Figure 2: Encapsulated clusters (Linux)

Well-Confined: Conceptually, the cluster q touches a single directory and does not dominate it. A co-change cluster q is categorized as *Well-Confined* if:

$$WellConfined(q), \text{if } focus(q) < 1.0 \ \ and \ \ spread(q) == 1$$

Figure 3 shows a Well-Confined Cluster, The cluster touches a single directory and shares the directory it touches with other clusters. This situation indicates that at least for those classes in the cluster there is co-changes outside the current module.

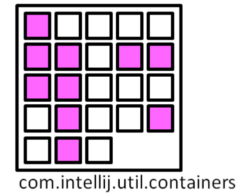

Figure 3: Well-Confined cluster (Intellij-Community)

Crosscutting: When a cluster is spread over several directories but touching few code files in each one, it is classified as Crosscutting Cluster. This situation seems to be the worst one since changes are spread across several directories, which would indicate difficulties during maintenance process. The following thresholds are set to describe a Crosscutting Cluster:

$$Crosscutting(q), \quad \text{if } spread(q) \geq 4 \quad \wedge \quad focus(q) \leq 0.30$$

Figure 4 shows an example of Crosscutting Cluster. The cluster is spread over five directories but it touches few files in each one.

Black Sheep: whether a cluster is spread over some directories but touching very few code files in each one, it is classified as Black Sheep Cluster. The following thresholds are set to represent a Black Sheep Cluster:

$$BlackSheep(q) = \text{if} \quad \begin{aligned} & spread(q) > 1 \wedge \\ & spread(q) < 4 \wedge \\ & focus(q) \leq 0.10 \end{aligned}$$

Figure 5 shows an example of Black Sheep Cluster. The cluster red is spread over three directories and touch very few files in each one.

We define 3 co-change patterns with similar behavior. Basically, a cluster q has a body B and a set of arms T. Most code

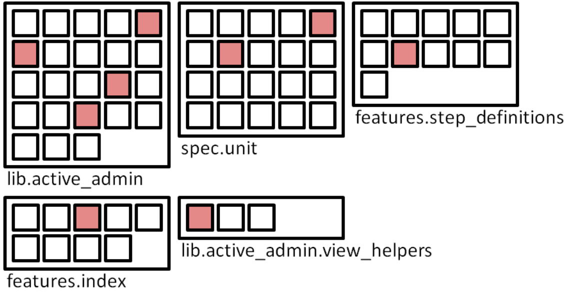

Figure 4: Crosscutting cluster (Active Admin)

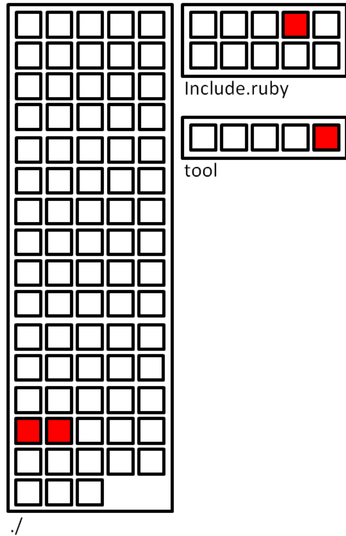

Figure 5: Black Sheep cluster (Ruby)

files are confined in the body and the arms have few files, i.e., very low focus. The following thresholds are set to represent a cluster q with these properties:

$$\begin{aligned} Octopus(q, B, T) = \text{if} \quad & touch(B, q) > 0.60 \wedge \\ & focus(T) \leq 0.25 \wedge \\ & focus(q) > 0.30 \end{aligned}$$

Figure 6 shows an example of Octopus cluster (dark blue). The body has 13 source code files confined in a single directory. Furthermore, the cluster has three tentacles. When these tentacles are considered as different sub-cluster, the tentacle focus is 0.04. Finally, this cluster has focus 0.72, which avoids categorized it as Crosscutting or Black Sheep.

$$\begin{aligned} Squid(q, B, T) = \text{if} \quad & touch(B, q) > 0.30 \wedge \\ & touch(B, q) \leq 0.50 \wedge \\ & focus(T) \leq 0.25 \wedge \\ & focus(q) > 0.3 \end{aligned}$$

Figure 7 shows a Squid cluster (light blue). The body has two files confined in a single package and the cluster has one tentacle. The touch of the body is 0.5 and the tentacle has focus 0.11.

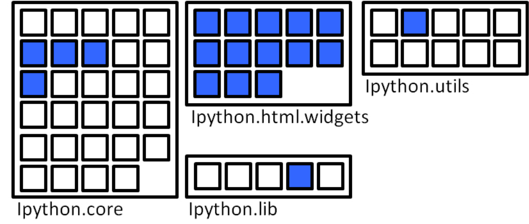

Figure 6: Octopus cluster (Python)

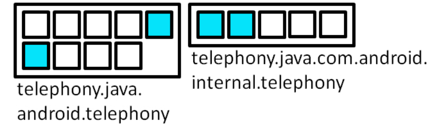

Figure 7: Squid cluster (Platform Frameworks)

Finally, this cluster has focus 0.31, which avoids categorized it as Crosscutting or Black Sheep.

We defined $focus(q) > 0.3$ to ensure a cluster does not be classified as Crosscutting and Octopus, or Squid, simultaneously.

4. Study Design

In this section we present the research questions (Section 4.1) and the criteria followed to select the projects used in this study (Section 4.2). We also present the criteria used to select the threshold used for preprocessing commits and extracting co-change clusters (Section 4.3).

4.1. Research Questions

In this section, our goal is to evaluate whether the proposed co-change patterns have different impacts on some software engineering output of interest (e.g., co-changeness level, the level of activity on clusters, the number of developers working on clusters, and the level of ownership on clusters). Specifically, we aim to answer four research questions:

RQ #1: How do the different patterns relate to co-change bursts?

The level of co-change bursts (bursts of commits used to create co-change graphs) can be interpreted as the level of ripple effect. The term ripple effect was first used by (Haney, 1972) to describe changes in one module that require changes in any another module. When assessing modularity using co-change clustering, all clusters emerge due to co-changes, and therefore, they express *ripple effects* in the system. Nonetheless, we plan to investigate in this RQ, if co-change patterns relate differently to the number of co-change bursts.

RQ #2: How do the different patterns relate to density of activity on clusters?

The density of activity on clusters may be measured by the number of commits performed on them controlled by clusters'

size or by the number of developers that perform commits in the clusters' classes. In other words, the higher the number of commits performed in a cluster, the higher the density of activity in this cluster. In this RQ, we plan to investigate whether the different cluster patterns relate to different activity levels.

RQ #3: How do specific patterns of co-change clusters relate to the number of developers per cluster?

The number of developers who work on the classes of a cluster may indicate different points, e.g., the heterogeneity of the team on a piece of the systems or the effort (manpower).

RQ #4. How do specific patterns of clusters relate to different number of commits of the cluster's owner?

To answer this RQ, we first compute the clusters' owners. The owner of a cluster `A` is the developer with the highest number of commits performed on classes in `A`. Our hypothesis is that the work of a clusters' owner may indicate different points. For example, the clusters with a dominant owner may be more error-prone for other programmers (Bird et al., 2011).

4.2. Dataset

We describe the projects hosted in GitHub that we collected, the selection criteria to generate the corpus and the analysis methods we use to answer our research questions. We conduct a large scale experiment on systems implemented in six popular languages as follows: C/C++, Java, JavaScript, Ruby, PHP, and Python. First, we rank the top-100 most popular projects in each language concerning their number of stars (a GitHub feature that allows users to favorite a repository). For each language, we analyze all the selected systems considering the first quartile of the distribution according to three measures: number of commits, number of files, and number of developers. Second, we choose the systems which are not in any measures of the first quartiles as presented in Figure 8. The first quartile of the distributions measures for number of commits range from 241 (Java systems) to 788 (Ruby systems), number of files ranges from 39 (Python systems) to 133 (C/C++ systems), and number of developers ranges from 18 (Java systems) to 72 (Ruby systems). Therefore, our goal in this step is to select large projects with large number of commits and a significant number of developers.

We also discard projects migrated to GitHub from a different version control system (VCS). Specifically, we discarded systems whose initial commits (around 20) include more than 50% of their files. This scenario suggests that more than half of development life of the system was implemented in another VCS. Finally, all selected systems were inspected manually on their respective GitHub page. We observed that `raspberrypi/linux` project is very similar to `torvalds/linux` project. To avoid redundancy, we removed this project.

We included 133 systems in our dataset. Table 1 presents a summary of the selected systems after the discarding step described previously. In this table, we presented C/C++ projects separately. For each language, we selected 18 (C/C++), 17 (PHP), 21 (Java), 22 (JavaScript and Python), and 33 (Ruby) systems. As a result, we considered in our experiment more than 2 million commit transactions. The overall sizes of the systems in number of files and line of code are 373K files and 41 MLOC, respectively.

Table 1: Projects in GitHub organized by language. The table describes The language, number of projects (Proj.), number of commits, number of developers (Dev.), number of files, and line of codes (LOC).

Language	Proj.	Commits	Dev.	Files	LOC
C	4	650,953	18,408	69,979	14,448,147
C++	14	196,914	2,631	37,485	5,467,169
Java	21	418,003	4,499	140,871	10,672,918
JavaScript	22	108,080	5,740	24,688	3,661,722
PHP	17	125,626	3,329	31,221	2,215,972
Python	22	276,174	8,627	35,315	2,237,930
Ruby	33	307,603	19,960	33,556	2,612,503
Total	133	2,083,353	63,194	373,115	41,316,361

Preprocessing files. In our technique we only consider co-change files that represent the source code. For this reason, we discard documentation, images, files associated to tests, and vendored files. We used Linguist tool[3] to generate language breakdown graphs. Linguist is a tool used by GitHub to compute the percentage of files by programming language in a repository. We follow Linguist's suggestions to remove files from our dataset. Linguist classified automatically 129,455 files (34%), including image, `xml`, `c`, `txt`, `js`, and `php` files. Finally, we inspected manually the first two top-level directories for each system searching for vendored libraries and documentation files not detected by Linguist. In this last step, we discarded 10,450 files (3%).

4.3. Threshold Selection

First, we preprocess the commit transactions of each system to compute co-change graphs. Table 2 shows the thresholds considered for this evaluation. We use the same thresholds of our previous experiences with Co-Change Clustering, where they were justified (Silva et al., 2014, 2015b). One exception is the time window threshold to merge commits. This threshold aims to group commits by the same developer that happen more than once in some period of time. This concept also called *"change bursts"* is applied in the literature (Nagappan et al., 2010). We applied this threshold to reduce the number of unitary commits because they cannot be used to evaluate a system in terms of co-change relations. Figure 9 shows the reduction rate of unitary commits after applying the time frame threshold to merge commits. We range this threshold from five to fifteen minutes. The figure presents only three thresholds to ease the analysis. While the threshold set in five minutes, the mean is 0.14 and the median is 0.15, for 10 minutes the mean and median slightly increase to 0.19 and 0.2, respectively. Moreover, the threshold set for thirteen and fifteen minutes, we observed significant reduction rate compared to others, e.g., the reduction rate ranged from 45% to 60% of unitary commit. Furthermore, in some cases we observed low reduction rate compared

[3] `https://github.com/github/linguist`

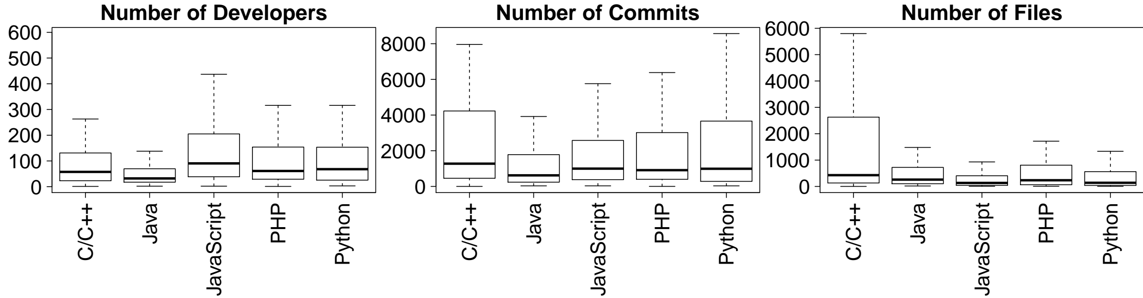

Figure 8: The overall number of commits and number of files by language

Table 2: Thresholds setup to extract co-change clusters

Metric	Value	Description
Maximum Scattering	10	We discard commits impacting more than ten packages
Minimum Weight	2	We remove edges with weight equal to one for not reflecting co-change relation (Beyer and Noack, 2005)
Minimum Cluster Size	4	After clustering we remove clusters with less than four classes
External Similarity	10^{-2}	We select clusters whose average external similarity tend to zero
Time Frame	10	Time window threshold to merge commits

five and ten minutes. In addition, we also noted an increase of unitary commit (reduction rate lower than zero) for three projects and this goes against our focus—reduce the number of unitary commits but only merging commits associated to the same task. For the above reasons, in our study we set up the time frame threshold to **ten minutes** because it does not cause great impact. Moreover, we preferred to adopt a conservative method to not have commit merging of different tasks. Finally, we randomly selected 100 change sets to check manually log messages whether the changes concern the same maintenance task. The first author confirmed that all sets refer to unique programming task.

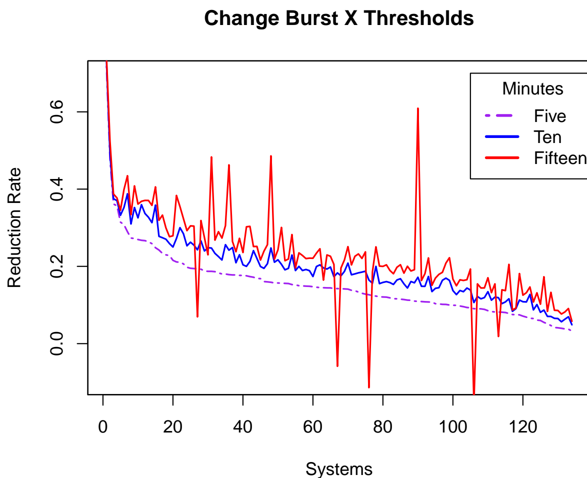

Figure 9: Commit reduction rate for 5, 10, and 15 minutes

In summary, ten systems do not provide enough commits to mine co-change clusters and they are discarded, thus, we extract co-change clusters for 123 projects. Our selection includes well-known systems, such as `ruby/ruby`, `torvalds/linux`, `php/php − src`, `webscalesql/webscalesql − 5.6`, and `rails/rails`.

5. Results

5.1. Clustering Quality

In this section, we analyze the quality of the extracted co-change clusters. In other words, we evaluate the process of co-change clusters' extraction. To increase the strength of our experiment, we use a sliding time window of three years for each of the 123 projects in GitHub. We defined three years because is too costly conduct the experiment for all projects in shorter period. Moreover, the projects' age differ from each other, e.g., as commits in `torvalds /linux` project started in 2002, then we have time frames like 2002—2004, 2003—2005, 2004—2006, 2005—2007, ..., 2012-2014, 2013-2014, 2014 (13 time windows, where each time window contributes with one clustering). Thus, for `torvalds/linux` project we have 13 clusterings. If we consider all projects, the outcome of this experiment consists of 600 final clusterings (600 time windows).

The goal in considering the sliding time window is to analyze whether co-change clusters are stable concerning their commit densities, i.e., number of commits divided by the number of source code files. As some projects are older than others, they do not have all time windows. We considered a range to have better uniformity, i.e., to consider old and recent projects in the same way. Thus, we analyzed the four most recent time windows from 2009—2011 to 2012—2014, resulting in 325 out of 600 clusterings. For this range, we extracted 5,229 co-change

clusters and only 241 clusters (5%) have not matched any of the six proposed patterns. Figure 10 shows the evolution of commit densities per co-change pattern. Note that for all patterns, there is no significant difference of commit density.

5.1.1. Analysis of the Co-Change Cluster Extraction

In our previous work we proposed the function *Coefficient* (Silva et al., 2014) to select the best number of partitions that is created in the first phase of the Chameleon algorithm. The goal of this quality function is to find the best clustering, in our case, combining cohesion and separation. However, selecting an adequate clustering metric to evaluate clusters is a well-known challenging issue (Almeida et al., 2011). For this reason, in this section we evaluate the stability of co-change clusters extracted from the evaluated projects.

We analyze the clustering quality using a sliding time window of three years for each of the 123 projects—600 time windows (clusterings) for all projects. To conduct this experiment, we run again the Chameleon algorithm for each system in our dataset to define the best number of partitions but this time using another metric, called *Coverage* metric (Almeida et al., 2011). Similarly to *Coefficient*, Coverage values also ranges from 0 to 1. However, higher values mean that there are more edges inside the clusters than edges linking distinct clusters. In summary, the difference between Coefficient and Coverage measures is that the former takes into account edges' weight and the latter number of edges.

We use MoJoFM (Wen and Tzerpos, 2004), a metric based on MoJo distance, to compare and evaluate the effectiveness of co-change clusters.[4] MoJo distance compares two clusterings of the same software system as the minimum number of *Move* or *Join* operations needed to transform a clustering A into B or vice versa. When A and B are very similar, there are few moves and joins. For instance, if two clusterings are identical, MoJo yields a quality of 100%. In this study, our clusterings are obtained from Coefficient and Coverage measures.

Figure 11 shows the MoJoFM values for all 600 clusterings. As we can observe, 471 clusterings (78.5%) have a match of 100%, i.e., they are identical. Furthermore, 553 clusterings (92%) have MoJoFM values greater than 90% and 581 (97%) greater than 80%. The mean value is 98% and the median value is 100%. This result shows that the co-change clusters are in most case well-defined sub-graphs in co-change graphs. In other words, most of co-change clusters are stable ones, i.e., their shapes are easily detected by Chameleon because the inter-clusters edges are minimized. Nonetheless, there are some systems with clusters' boundaries difficult to identify. While Coefficient considers the frequency of co-changing, Coverage measure does not. As we deal with recurrent maintenance task, Coefficient is more appropriated because it seeks for set of classes that frequently change together.

[4] To calculate MoJoFM, we relied on the MoJo 2.0, `http://www.cs.yorku.ca/~bil /downloads/`.

5.2. Classifying Co-Change Clusters

We classify the extracted co-change clusters into six patterns: Encapsulated, Well-Confined, Crosscutting, Black-Sheep, Squid, and Octopus Clusters. Table 3 presents the co-change clusters by pattern. In summary, the six co-change patterns cover 1,719 out of 1,802 (95%) of the extracted clusters. In contrast, in our previous study (Silva et al., 2015b) we used three co-change patterns (Encapsulated, Crosscutting, and Octopus) to categorize the extracted clusters and they covered around 50% of the clusters. Therefore, we can observe that the six co-change patterns increased substantially the percentage of categorized clusters (95%). By contrast, we could not assign a specific form to the remaining clusters, given the proposed rules. Typically, these clusters stay between Tentacled and Scattered clusters for having more sensitive definition concerning the thresholds, opposed to Wrapped clusters which are more sharply defined. For this reason, we decide to exclude those 5% of the study as a small consequence on the global picture that we aim with the statistical analysis.

Table 3: Number and percentage of categorized co-change clusters

Pattern	# Systems	# Clusters
Encapsulated	76 (61.8%)	464 (25.7%)
Well-Confined	56 (45.5%)	227 (12.6%)
Crosscutting	51 (41.5%)	106 (5.9%)
Black-Sheep	18 (14.6%)	51 (2.8%)
Octopus	114 (92.7%)	805 (44.7%)
Squid	44 (35.8%)	66 (3.7%)
No Pattern	38 (31%)	83 (4.6%)
Total Coverage		1,719 (95.39%)
Total of Extracted Clusters		1,802 (100%)

Table 4: Number and percentage of categorized co-change clusters grouped in three major patterns

Pattern	# Systems	# Clusters
Wrapped	95 (77%)	691 (38.35%)
Scattered	57 (46%)	157 (8.71%)
Tentacled	114 (93%)	871 (48.33%)
No Pattern	38 (31%)	83 (4.61%)
Total Coverage		1,719 (95.39%)
Total of Extracted Clusters		1,802 (100%)

To conduct our study, we group the categorized clusters to ease the analysis, as follows: (i) clusters with localized changes: Encapsulated and Well-Confined Clusters, (ii) clusters which present crosscutting behavior: Crosscutting and Black-Sheep Clusters, (iii) clusters with body and arms: Squid and Octopus Clusters. Table 4 summarizes the co-change clusters as grouped in these three major patterns: clusters with confined changes are named Wrapped (Encapsulated and Well-Confined), clusters with crosscutting behavior as Scattered (Crosscutting and Black-Sheep), and clusters with body and arms as Tentacled (Squid and Octopus). As we can observe, instances of Tentacled Clusters are quite common, since they are present on

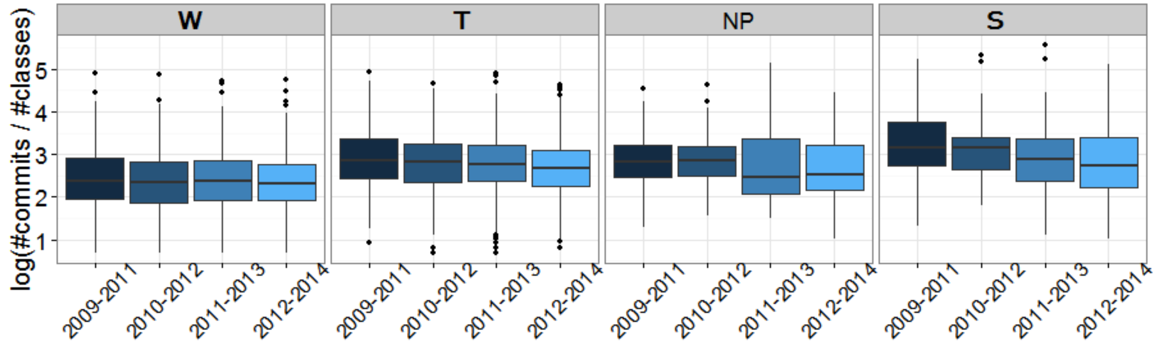

Figure 10: Evolution of commit density (source code files) per co-change pattern

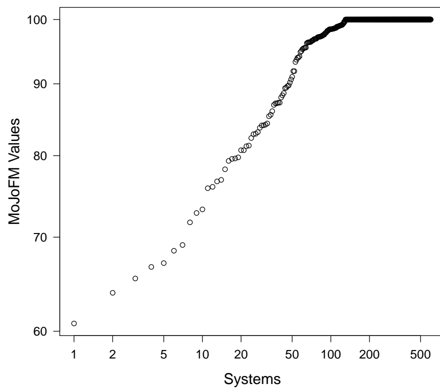

Figure 11: MoJoFM values for 600 clusterings

114 systems (93%), representing 48.33% of all co-change clusters. Furthermore, 38.35% of the clusters are Wrapped Clusters, which cover 95 systems (77%). By contrast, Scattered Clusters are detected in 57 systems (46%) and they represent only 8.71% of the co-change clusters.

Figure 12 depicts the percentage of co-change clusters by system in stacked bar plots. The clusters are grouped by pattern (dashed bars) or pattern absence (solid purple bars). We found instances of co-change patterns in all 123 systems. Specifically, all systems have Wrapped or Tentacled Clusters. Tentacled are present in most systems and the average percentage of such clusters by system is 57% (dashed red bars). The number of systems with Wrapped Clusters is smaller than Tentacled but it is significant, 30% on average (dashed green bars). By contrast, co-change clusters that present Scattered behavior are relatively rare (dashed blue bars).

We identify eight systems that have all co-change clusters categorized as Wrapped (dashed green bars). Furthermore, all clusters in 10 systems are Tentacled (dashed red bars). In addition, 85 projects have all co-change clusters categorized and 54 projects have only Wrapped and/or Tentacled Patterns. In contrast, Scattered Clusters do not dominate any system, e.g., 58% of the systems with clusters categorized as Scattered have only one cluster and 72% have two clusters. `JetBrains/intellij − community` and `torvalds/linux` are the top two systems concerning absolute number of Scattered Clusters, with 13 and 29 clusters, respectively. Scattered Clusters in `Intellij − Community` represent 9% of extracted clusters, while in Linux they represent only 8%.

Programming Language. From another perspective, Figure 13 depicts the percentage of co-change clusters by programming language. As can be observed, the three patterns are detected independently of implementation language. The results also show that few clusters match the Scattered Pattern. For instance, PHP and C++ projects have their clusters categorized as Scattered Clusters, 9% and 12% respectively. Conversely, Tentacled Pattern is very common. The percentage of Tentacled Clusters is more than 50% on average, with exception for C projects (28%). Wrapped is also common in all languages, they represent 36% of the clusters on average. Interestingly, C systems have the highest proportion of Wrapped clusters. In general, C systems have simpler organization in terms of folders, C systems have simpler organization in terms of folders, and they have proportionally less files per line of code, which seems to have favored changes to be wrapped.

Application Domain. We adopted the classification proposed by Borges and Valente (2018) to assign systems to a particular domain. Then, we manually classified all projects in our dataset, as follows:

1. *Application software.* Systems implemented to end-users, such as browsers, text editors (e.g., `drupal/drupal`).
2. *System software.* Systems that provide services and infrastructure, such as operating systems and databases (e.g., `webscalesql/webscalesql − 5.6`).
3. *Web libraries and frameworks.* Systems used to implement web application interfaces (e.g., `django/django`).
4. *Non-web libraries and frameworks.* Systems used to implement components for an application (e.g., `scikit − learn/scikit − learn`).
5. *Software tools.* Systems that support development tasks, such as IDEs and compilers (e.g., `git/git`).

Figure 14 shows characterizes the prevalence of clusters within different domains. All systems were classified into Application Software (App), Non-web Libraries and Frameworks (Lib), System Software (System), Software Tools (Tools), and Web Libraries and Frameworks (Weblib). We also observe no

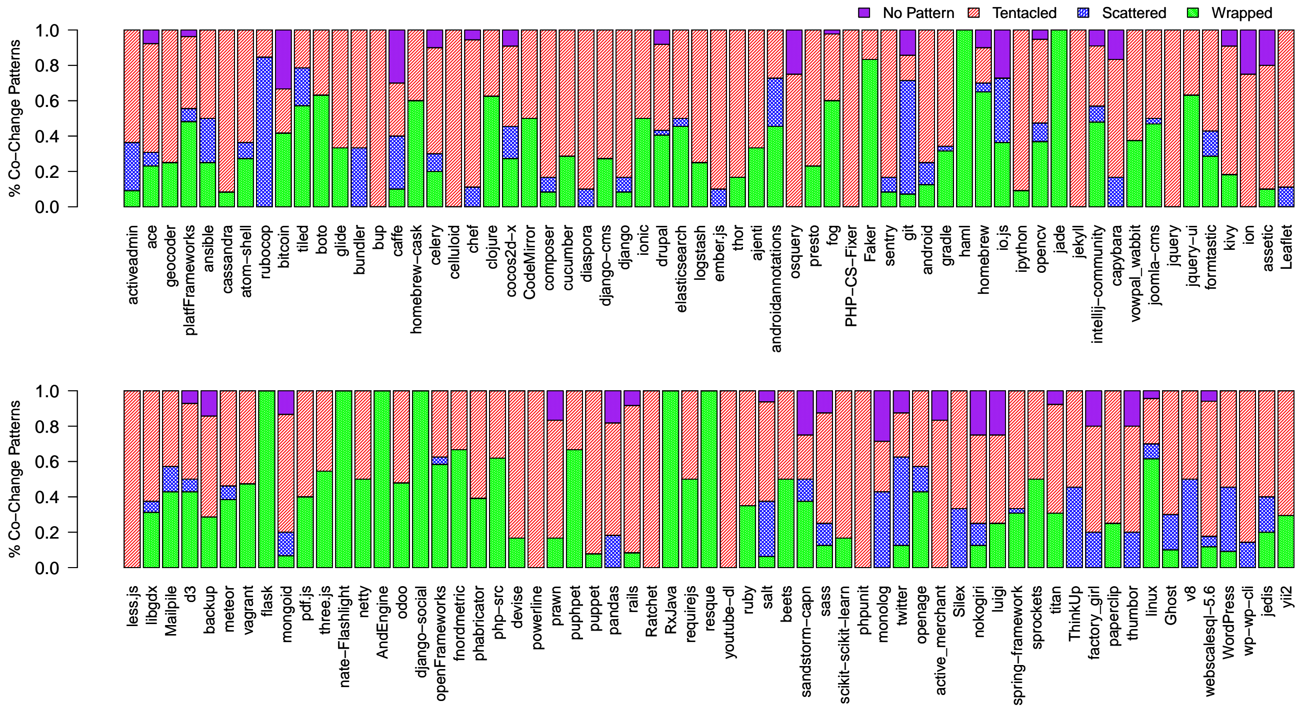

Figure 12: Relative number of identified and classified clusters for each system

major difference on the proportion of co-change patterns for the distinct domains, except the System Software domain that has more Wrapped than Tentacled clusters.

5.3. Statistical Methods

We use regression modeling to describe the relationship of a set of predictors against a response. Regression modeling is specially recommend in multi-variable analysis to understand the effect of several variables in only one model. To answer the research questions proposed in Section 5.4, we model either the number of co-change commits or the number of commits in clusters against other factors. We used the R statistical software. Because this kind of count data is over-dispersed, negative binomial regression (NBR) is used to fit the models, which is a type of generalized linear model used to model count responses. We used the function *glm.nb* function available in package MASS. NBR can also handle over-dispersion (Cohen et al., 2003). For example, there are cases where the response variance is greater than the mean. We verify if the main predictor really is a statistically significant predictor comparing the model with the predictor and without the predictor using a ANOVA with Chi Square test to verify whether reduction in the residual sum of squares are statistically significant or not. Moreover, we use the deviance table of the fitted models to understand the relative importance of the studied factors over the control factors, reporting on the explained variance of the studied factors.

Moreover, we control for several factors that are likely to influence the outcome. Specifically, NBR models the log of the expected count as a function of the predictor variables. We can interpret the NBR coefficient as follows: for a one unit change

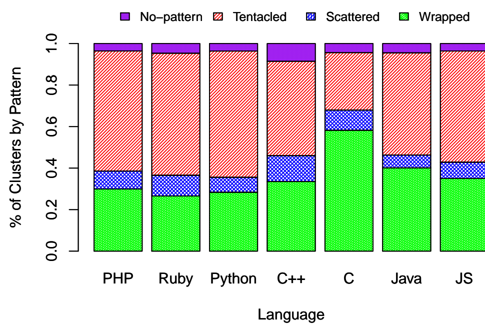

Figure 13: Relative Co-Change Pattern coverage by Programming Language

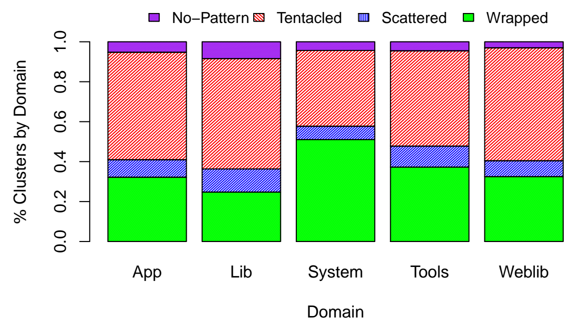

Figure 14: Relative Co-Change Pattern coverage by Domain

in the predictor variable, for a coefficient β_i, a one unit change in β_i yields an expected change in the response of e_i^{β}, given the other predictor variables in the model are held constant. We also measure how much a predictor variable accounts for the total explained deviance.

Absence of multi-collinearity is one of the assumption of negative binomial models, i.e., there should be no perfect linear relationship between two or more of the predictors. So, the predictor variables should not correlate too highly. To check whether excessive multi-collinearity is an issue, we compute the variance inflation factor (VIF) of each dependent variable in all models. We used the function *vif* available in the package *car*. Although there is no particular value of VIF that is always considered excessive, our VIF values do not exceed 5 which is a widely acceptable cut-off (Cohen et al., 2003).

To help the interpretation of the NBR models for a wider audience, we also provide Fox's effect plots (Fox, 2003), available in the R package *effects*. In effect plots, predictors in a term are allowed to range over their combinations of values, while other predictors in the model are held to "typical" values, then the fitted value of the response and standard error of the effects are computed for each combination of regression values.

5.4. Analysis of the Results

Before analyzing the rationale behind a co-change pattern classification, we begin with four research questions to investigate co-change clusters quantitatively.

RQ #1: How do the different patterns relate to co-change bursts?

To answer this RQ, we analyze the number of categorized clusters to investigate whether the patterns are associated differently to the number of co-change bursts (bursts of commits that spans several modules). Table 5 shows the NBR model for number of co-change bursts per system. We include two variables as controls for factors that influence co-change bursts: 1) the number of source code files (*nFiles*) in a project is included because size may induce a greater number of co-change relations. 2) the project age (*nMonths*) may also impact the analysis because older projects would have more evolutionary data available. The number of co-change clusters by pattern allows us to investigate the ripple effect level in each pattern. For instance, `torvalds/linux` has 65,433 co-change bursts, 48,948 source code files, 161 months of commits, 211 Wrapped, 29 Scattered, and 88 Tentacled Clusters.

Table 6 shows that all variables are significant, with the exception of Wrapped clusters (*nWrapped*), i.e., those factors account for some of the variance in the ripple effect. Although, there *nWrapped* is significant as predictor as shown in Table 5, it is negligible how much it explains deviance. The number of source code files in a project accounts for the majority explained deviance (61%), i.e., *nFiles* divided by the sum of the *Deviance* column. In the analysis of this deviance table, we can also see that the next closest predictor is project age which accounts for 20%. The number of *Tentacled* Clusters (*nTentacled*) accounts for 11% of the total explained deviance, and the number of *Scattered* clusters (*nScattered*) accounts for 7% of the deviance. Therefore, these relationships—*Tentacled* and *Scattered*—are statistically significant with an important effect.

Table 5: NBR model for number of co-change bursts per system.

	Estimate	Std. Error	z value	Pr(> |z|)	
(Intercept)	4.967e+00	1.408e-01	35.286	< 2e-16	***
nFiles	6.173e-05	1.653e-05	3.734	0.000188	***
nMonths	1.159e-02	1.784e-03	6.500	8.03e-11	***
nWrapped	-3.549e-02	1.615e-02	-2.197	0.028008	*
nTentacled	1.154e-01	1.587e-02	7.271	3.58e-13	***
nScattering	1.602e-01	3.523e-02	4.546	5.46e-06	***

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, . $p < 0.1$

Table 6: Deviance table for NBR model on the number of co-change bursts per system.

	Df	Deviance	Res. Df	Res. Dev.	Pr(>Chi)	
NULL			121	466.55		
nFiles	1	204.974	120	261.58	< 2.2e-16	***
nMonths	1	67.733	119	193.85	< 2.2e-16	***
nWrapped	1	2.559	118	191.29	0.1097	.
nTentacled	1	37.385	117	153.90	9.698e-10	***
nScattered	1	23.584	116	130.32	1.195e-06	***

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, . $p < 0.1$

The column Estimate in the model presented in Table 5 relates the predictors to the result. The coefficients are compared among the respective variables to the different co-change patterns. Note that all intercepts are positive, with exception of *nWrapped* variable. A significant result is that for each added *Tentacled* and *Scattered*, the increase is 1.12 ($e^{1.154e-01}$) and 1.17 ($e^{1.602e-01}$), respectively. This outcome can also be noted in Figure 15, which presents the effects on co-change bursts for all variables included in the NBR model.

> There is a moderate and significant relationship between the frequency of a given cluster pattern in the system and the number of co-change bursts, controlled by system size and age. An increase in the number of Scattered and Tentacled Clusters is associated with an increase in the number of co-change bursts. In contrast, the association with Wrapped Clusters is not significant. An implication of these findings is that classes in Wrapped clusters, although co-changing, do not increase co-change bursts in the system as classes in Scattered and Tentacled clusters. Therefore, if developers work to decrease the number of Scattered and Tentacled clusters in the system, co-change bursts would likely decrease.

RQ #2: How do the different patterns relate to density of activity on clusters?

To answer this RQ, we analyze clusters with different patterns and their activity levels (number of commits per cluster). Table 7 details the NBR model for number of commits per cluster—variable of response. The lines in the table represent clusters and each cluster has a type and its respective factor represents this type (Scattered, Tentacled, or No Pattern). The type *Wrapped* is not listed in the table, because the results shows the

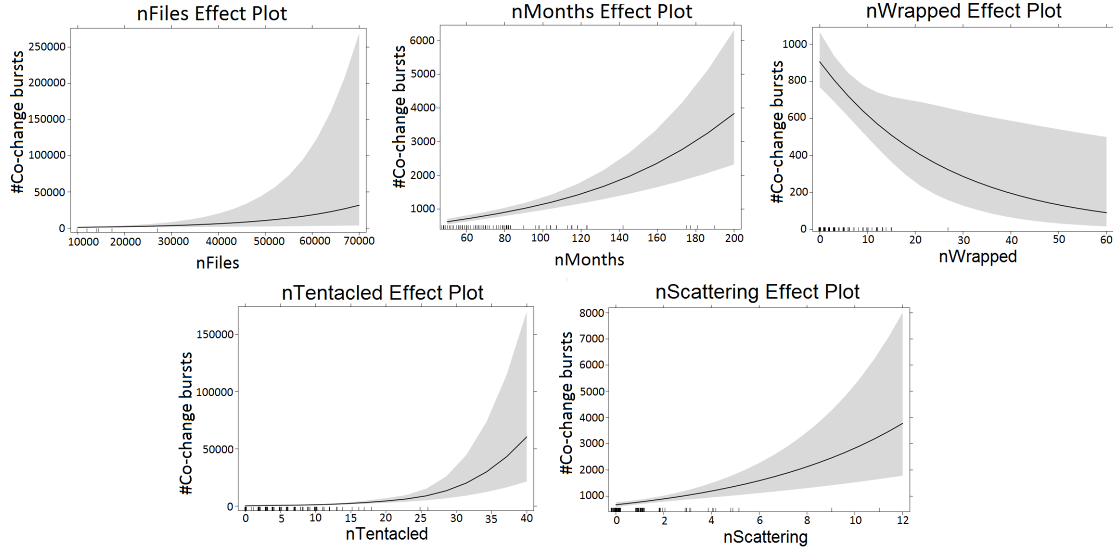

Figure 15: Effects on Co-change Bursts

coefficients of other types compared to the *Wrapped*. We decided to compared with *Wrapped* cluster because they are, by hypothesis, considered to have the lowest level of activity, so we compare the others with *Wrapped* to identify and quantify the increase in activity, if any. The idea is to analyze which type of commit has more impact on the variable of response (activity). As clusters with greater sizes tend to have more activity, we included the *size* variable, as a control variable in the model because some co-change patterns tend to follow a particular structure concerning their size. For instance, Tentacled Clusters have more source code files than the others. Note that in this model, we do not include age, because the the unit of experimentation is the cluster, and not the system as in the previous question.

Table 7: NBR model for number of commits per cluster. S - Scattered Clusters, NP - Clusters with no Pattern, T - Tentacled Clusters

	Estimate	Std. Error	z value	Pr(> $\|z\|$)	
(Intercept)	4.7181253	0.0426703	110.572	< 2e-16	***
size	0.0212170	0.0002399	88.448	< 2e-16	***
factor(S)	0.9521920	0.0989228	9.626	< 2e-16	***
factor(NP)	0.4957564	0.1300395	3.812	0.000138	***
factor(T)	1.0119010	0.0580520	17.431	< 2e-16	***

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, . $p < 0.1$

The result shows that compared to the *Wrapped* Clusters, the other clusters have higher activity level. For instance, the Tentacled Clusters increase the intercept (Wrapped Clusters) in 1.0119010 and the Scattered Clusters in 0.9521920. This means that the intercept for Wrapped Clusters is $e^{4.7181253} \approx 112$; for Tentacled Clusters is $e^{4.7181253+1.0119010} \approx 308$, and for Scattered Clusters is $e^{5.67} \approx 290$. Figure 16 shows that *Wrapped* Clusters have much lower level of activity than the other kind of clusters.

Moreover, in Figure 17, we observe some variability between clusters activity in different programming languages. However, ANOVA shows no statistically significant difference in the effect that would result in the programming language being a significant factor. We can also observe that activity is mostly stable across different domains, although being less stable in Scattered Clusters, which indeed have lower prevalence. In addition, ANOVA shows no evidence for the overall effect of the domain on the clusters activity. Interestingly, Wrapped Clusters consistently have lower activity level compared to *Tentacled* and *Scattered* Clusters, as already shown in Table 7. For Java and Ruby, *Scattered* and *Wrapped* present lower difference, which is also explained by the higher variability and lower prevalence of Scattered Clusters.

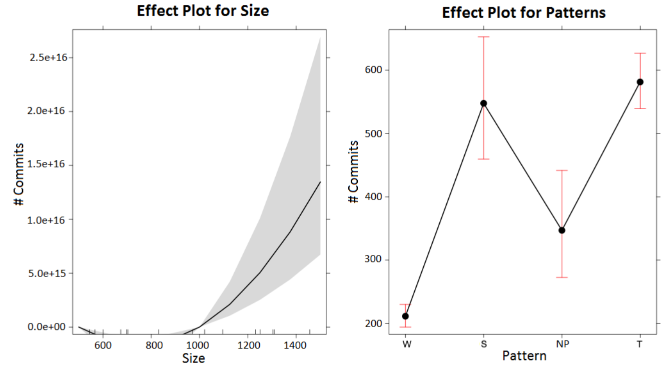

Figure 16: Effects on Activity of Clusters. W - Wrapped, S - Scattered, T - Tentacled, and NP - Clusters with no Pattern

Table 8 shows that *size* variable accounts for the majority explained deviance. In the analysis of deviance, we see that the factor pattern of clusters on the system accounts for 9.11% of the total explained deviance, i.e., *pattern* divided by the sum of the Deviance columns. Therefore, the results presented in Table 7 show that Tentacled and Scattered Clusters are statistically significant with an important effect.

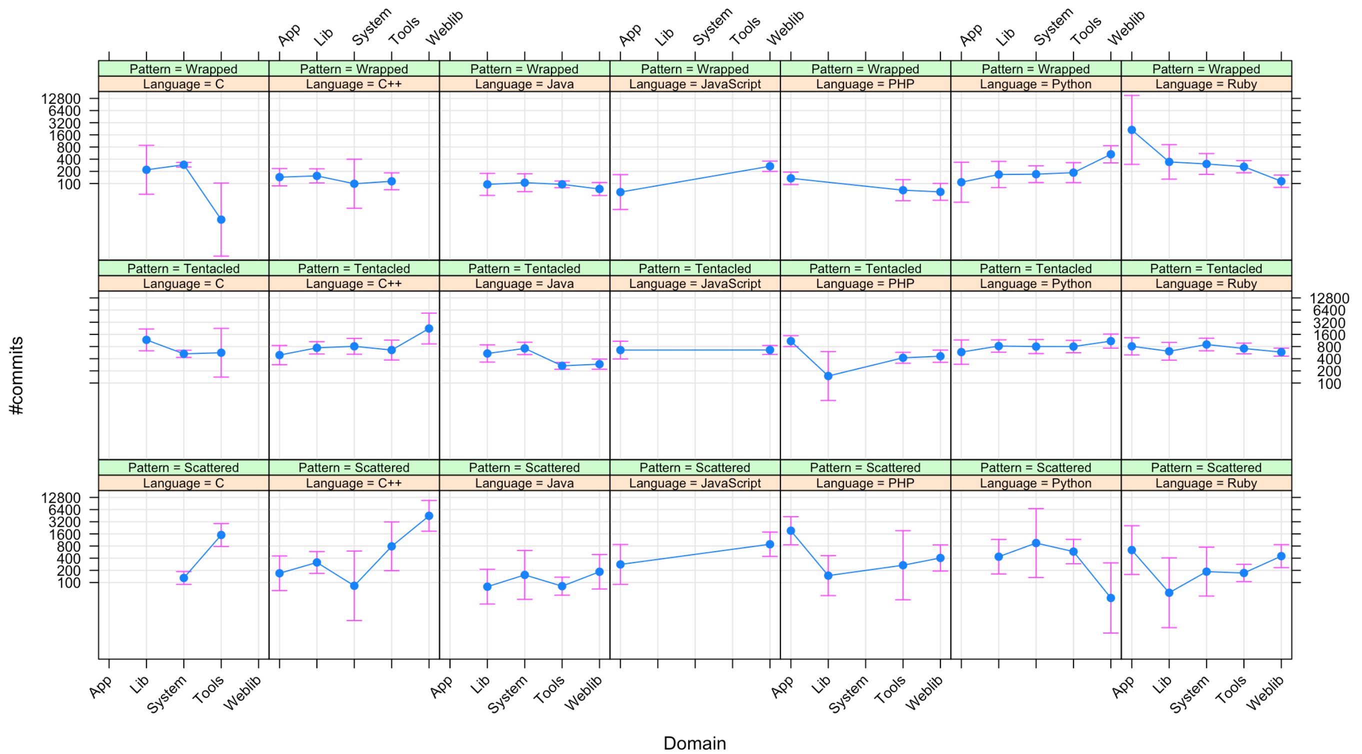

Figure 17: Effects of Domain and Language on Activity of Clusters.

There is a moderate and significant relationship between clusters of a given co-change pattern and the level of activity on the clusters. Tentacled Clusters have a greater association with the activity level than Scattered Clusters, i.e., Tentacled have more changes than Scattered. Tentacled and Scattered Clusters have the activity level substantially higher than Wrapped Clusters, with the exception for Scattered Clusters in Java and Ruby. In other words, Wrapped Clusters are the ones with the lowest level of changes. These findings is similar to the ones in the previous RQ. If developers work to limit the occurrences of Tentacled and Scattered clusters, they would likely promote less commits in the system. This somehow converge with the idea that if commits are more localized in classes in Wrapped packages one would perform more complete changes avoiding commits to fix ripple effects.

Table 8: Deviance table for NBR model on the number of commits per cluster

	Df	Deviance	Res. Df	Res. Dev.	Pr(>Chi)	
NULL			1801	5008.3		
size	1	2616.93	1800	2391.4	$< 2.2e{-16}$	***
pattern	3	262.42	1797	2128.9	$< 2.2e{-16}$	***

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, . $p < 0.1$

RQ #3: How do specific patterns of co-change clusters relate to the number of developers per cluster?

Table 9: NBR model for number of developers per cluster

	Estimate	Std. Error	z value	Pr(> |z|)	
(Intercept)	2.6240082	0.0387607	67.698	$< 2e{-16}$	***
size	0.0074887	0.0002114	35.420	$< 2e{-16}$	***
factor(S)	0.6793003	0.0887683	7.653	1.97e-14	***
factor(NP)	0.4141105	0.1170532	3.538	0.000403	***
factor(T)	0.7448908	0.0523115	14.240	$< 2e{-16}$	***

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, . $p < 0.1$

To answer this RQ, we analyzed the number of developers who changed classes in each cluster. Table 9 details the NBR model for number of developers per cluster. The *size* variable is included as a control factor in the model for the same reason as in RQ #2. The result shows that compared to the *Wrapped* Clusters, the other clusters have more developers working on. For instance, Tentacled Clusters increase the intercept (Wrapped Clusters) in 0.7448908. This means that the intercept for Wrapped Clusters is $e^{2.6240082} \approx 14$ and the intercept for Tentacled Clusters is $e^{2.6240082+0.7448908} \approx 29$. As another example, the intercept for Scattered Clusters is $e^{3.303} \approx 27$. Figure 18 depicts the difference among the cluster patterns concerning number of developers. *Wrapped* Clusters usually have much less developers than in other patterns.

Similar to RQ #2, Table 10 shows that the *size* variable accounts for the majority of explained deviance. Furthermore, the *pattern* factor of clusters accounts for 10.73% of the total explained deviance, i.e., *pattern* divided by the sum of the Deviance column. Therefore, Tentacled and Scattered Clusters are

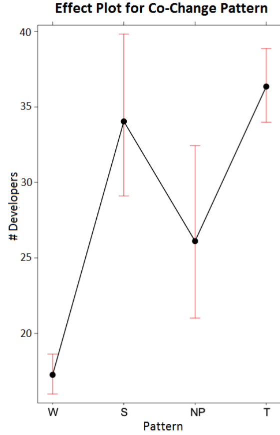

Figure 18: Effects on Number of Developers of Clusters. W - Wrapped, S - Scattered, T - Tentacled, and NP - No Pattern

Table 10: Deviance table for NBR model for number of developers per cluster

	Df	Deviance	Res. Df	Res. Dev.	Pr(>Chi)	
NULL			1801	3636.7		
#classes	1	1458.17	1800	2178.5	< 2.2e-16	***
pattern	3	175.21	1797	2003.3	< 2.2e-16	***

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, . $p < 0.1$

statistically significant with important effect.

> There is a moderate and significant relationship between co-change pattern and number of developers per cluster. Tentacled Clusters usually have more developers working on them than Scattered Clusters. Wrapped Clusters are usually the ones with the lowest number of developers. This finding reinforces the idea that because Tentacled clusters have more activity, they require more people working on them.

RQ #4. How do specific patterns of clusters relate to different number of commits of the cluster's owner?

We also analyze owners of the co-change clusters per pattern. Table 11 details the NBR model for number of commits related to cluster's owner. We include two control variables: number of commits and number of developers. Number of commits provide information concerning the frequency of changes in a cluster, i.e., the more commits in a cluster, the more its owner is likely to commit. Additionally, number of developers in a cluster may have some influence on the likelihood of the owner to commit. They have shown to explain most of the variance in the model as shown in Table 12. As we can observe in Table 11, in *Tentacled* Clusters the clusters' owner are more important, i.e., concentrate more commits than *Wrapped* Clusters. Interestingly, *Scattered* Clusters do not show a significant difference from *Wrapped* Clusters, defined by non-significant p-value. Tentacled Clusters increase the intercept (Wrapped Clusters) in 0.847 and Scattered Clusters increase the intercept only in 0.152. This means that the intercept for Wrapped Clusters is $e^{3.005} \approx 20$; the intercept for Scattered Clusters is $e^{3.005+0.152} \approx 23.5$, and the intercept for Tentacled Clusters is $e^{3.005+0.8472} \approx 47$. In other words, the commits performed by the owner have higher importance in Tentacled Clusters than in Wrapped Clusters, also illustrated in Figure 19. To define the notion of ownership, we rely on the committer. The committer is the person who last applied the patch, whereas the author is the person who originally wrote the patch. So, it may be the case that core developers are committers for other authors. In our data, we tend to credit core developers to be the owners, even though, they maybe were not actually the authors.

Table 11: NBR model for commit number of cluster's owner

	Estimate	Std. Error	z value	Pr(> |z|)	
(Intercept)	3.005e+00	5.778e-02	52.006	<2e-16	***
#commits	1.413e-04	5.393e-06	26.198	<2e-16	***
log #devs	4.772e-01	2.081e-02	22.933	<2e-16	***
factor(S)	1.524e-01	8.545e-02	1.784	0.0745	.
factor(NP)	1.158e-01	1.116e-01	1.038	0.2991	
factor(T)	8.472e-01	5.227e-02	16.207	<2e-16	***

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, . $p < 0.1$

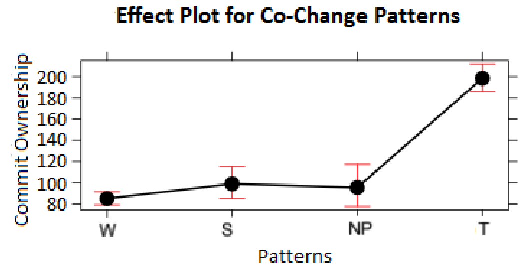

Figure 19: Effects on Commits of Owners. W - Wrapped, S - Scattered, T - Tentacled, and NP - Clusters with no Pattern

Table 12 shows that variables and *pattern* factor are significant. The number of commits in a cluster accounts for the majority of explained deviance. The *pattern* factor of clusters on the system accounts for 10.61% of the total explained deviance, i.e., *pattern* divided by the Deviance column.

> There is a moderate and significant relationship between co-change patterns and commits performed by the clusters' owner. Tentacled Clusters usually have more commits related to cluster's owner than the remaining clusters. In contrast, Wrapped and Scattered Clusters have no significant difference concerning number of commits. A possible rationale on these numbers is that Tentacled clusters tend to require a broader touching in the system because

Table 12: Deviance table for NBR model on the commit number of cluster's owner

	Df	Deviance	Res. Df	Res. Dev.	Pr(>Chi)	
NULL			1801	4662.8		
#commits	1	1695.39	1800	2967.4	< 2.2e-16	***
log #devs	1	654.89	1799	2312.5	< 2.2e-16	***
pattern	3	274.83	1796	2037.6	< 2.2e-16	***

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, . $p < 0.1$

of the tentacles. Thus, owners play an important in coping with the dependencies on those tentacles. Crosscutting would also display such behavior, but maybe because of their low prevalence, simple crosscutting commits, such as simple refactoring, can be carried by non-owners.

6. Qualitative and Semantic Analysis

In this part of our work we focus on co-change cluster analysis in a qualitative perspective. Our goal is to investigate clusters which have source code files massively changed, i.e., co-change clusters with high level of activity. We analyze natural language topics in log messages of commits aiming at retrieving conceptual information to describe activities frequently performed on co-change clusters and key words that may reveal their concerns. For this purpose, we apply topic model technique to automatically infer the rationale behind a co-change pattern classification. Specifically, we use topic modeling (Blei et al., 2003; Griffiths and Steyvers, 2002; Rosen-Zvi et al., 2004) to gain sense of what semantic meaning co-change clusters present and what maintenance activity type had been applied frequently. We intend to comprehend whether the maintenance tasks differ in clusters with distinct patterns.

6.1. Cluster Selection Process

For each pattern, we rank the 10 most massively changed clusters detected for 123 projects in GitHub. Table 13 describes the systems which contains such clusters. We apply some pre-processing steps, such as lowercase, tokenize, stopwords removal, and punctuation removal. To search the number of topics threshold, we explore the range of values from 5 to 45 and analyze how the text messages break down.[5] The number of topics which better organizes the text messages ranges from 20 to 40. Small number of topics end up in generic topics, i.e, they do not describe enough detailed information to comprehend cluster's semantics. In contrast, large number of topics result in null topics and very similar ones.

Table 13: The projects with the most massively changed clusters

System	Description
Haml	Markup Language
Git	Version Control System
Odoo	ERP and CRM System
Pandas	Python Data Analysis Lib.
Homebrew	Apple App Installation
Celery	Distributed Task Queue
PHP-src	Scripting Language
Twitter	Ruby Interface to the Twitter API
Linux	Operational System
Webscalesql-5.6	Relational Database Management System
V8	JavaScript Compiler
Ruby	Programming Language
WordPress	Content Management System
JQuery	JavaScript Library
io.js	NPM Compatible Platform
Beets	Music Media Organizer

6.2. Topic Extraction

Wrapped Clusters: Table 14 shows the 10 most massively changed clusters classified as Wrapped. We rely on the number of commits per file and cluster size to rank such clusters, i.e., #commits divided by cluster size. The column *score* in the table shows the ranking of the clusters, e.g., the highest score is the first top ten. The column Co-change Pattern describes the fine-grained pattern for each Wrapped Cluster, i.e., clusters classified as Encapsulated or Well-Confined. We can observe that 80% of the clusters are Encapsulated. The table also reports key words retrieved from topics to describe clusters semantically. To comprehend the meaning of these clusters, we analyzed the summary of the topics and inspected their distribution maps. We could easily detect the concerns implemented by the clusters. For example, the Haml cluster describes engines of templates (`Haml` and `Sass`) for `HTML` and `CSS` documents. As another example, in Odoo, the listed key words for the cluster describes user interface requirements of the system associated to Kanban view and web graph. Moreover, Homebrew's cluster describes formulas for database management systems and programming language, such as Postgresql and Python. Similarly, the cluster PHP (score 117) is encapsulated in the directory `ext.phar`. The `phar` is an extension in the PHP system responsible to wrap entire PHP projects into a single `phar` file (PHP archive) for helping in the distribution and installation. Furthermore, the source code files in this directory describe compressing format, such as `zip` and `tar`.

Scattered Clusters: Table 15 shows the 10 most massively changed clusters with crosscutting behavior. The table also reports the number of commits per file, cluster size, co-change pattern, and key words retrieved from topics to describe clusters semantically. The column *score* in the table shows the ranking of the clusters, e.g., the highest score is the first top ten. The column Co-change Pattern describes the fine-grained pattern for each Scattered Cluster, i.e., clusters classified as Crosscutting or Black Sheep. We can observe that 90% of the clusters are Crosscutting. As can be observed in Table 15, V8 system contains four Scattered Clusters in the ranking list. This system compiles JavaScript to specific machine code, such as `arm` and `MIPS`. Specifically, the first three clusters in the table present similar behavior. They touch directories containing machine-independent (`src`) and machine-dependent (`arm`, `arm64`, `x64`, `ia32`, and `mips`) source code files. We analyzed distribution maps of these clusters, extracted topics, and inspected the source code files to ease the comprehension about their concepts. For example, the first V8 cluster has source files to generate code in different platforms. The second V8 cluster has files of the machine-independent optimizer (`hydrogen`) and the low-level machine-dependent optimizer (`lithium`). Similarly, the third V8 cluster contains files responsible for code stub generation. Particularly, the specific architecture directories confine source code files concerning `lithium` optimizer and generators of stub and code. In contrast, the directory

[5] To extract and evaluate topics, we relied on the Mallet topic model package and guidelines, `http://mallet.cs.umass.edu/topics.php`.

Table 14: Ranking of Wrapped Clusters

Score	System	Co-change Pattern	# Commits per File	Size	Topic Summary
148	Haml	Encapsulated	1,477	10	rails sass haml html engine
129	Odoo	Encapsulated	7,733	60	kanban view addon web graph
126	Homebrew	Well-Confined	884	7	pypy python mongodb postgresql formula
117	PHP-src	Encapsulated	2,346	20	phar zip zlib tar stream
112	PHP-src	Encapsulated	5,935	53	mysql mysqli mysqlnd libmysql ext
101	Linux	Encapsulated	811	8	net emulex driver ethtool pci
100	Homebrew	Well-Confined	697	7	mysql mariadb pcre lua tools
85	PHP-src	Encapsulated	340	4	pcntl process control support signal
84	Linux	Encapsulated	839	10	staging xgi chipsets driver video
80	PHP-src	Encapsulated	5,872	73	extension library libgd ibase odbc xmlwriter

Table 15: Ranking of Scattered Clusters

Score	System	Co-change Pattern	# Commits per File	Size	Topic Summary
334	V8	Crosscutting	4,009	12	mips arm ast code generator
253	V8	Crosscutting	9,877	39	arm lithium allocation hydrogen instructions
194	V8	Crosscutting	3,107	16	generate code stub arm hydrogen
158	WordPress	Crosscutting	4,596	29	theme admin user customizer props
143	io.js	Crosscutting	4,446	31	node stream child process dns
112	Git	Crosscutting	8,106	72	sha gitweb git gui daemon
104	Twitter	Crosscutting	1,452	14	middleware development dependency gemspec version
92	V8	Black Sheep	1,746	19	log cpu profiler generator utils
90	Pandas	Crosscutting	454	5	plot series boxplot dataframe hist
89	Celery	Crosscutting	627	7	platform canvas log built task

`src` contains source code concerning `hydrogen`, `lithium`, and generators. Apparently, the design decision to decompose the system in directories by hardware architecture scattered these concerns over the directories. Thus, if requirements related to these concerns change, the several directories may have to be updated. Instead, if concerns were centralized in their respective directories, the change would be confined in one location.

Tentacled Clusters: Table 16 shows the 10 most massively changed clusters classified as Tentacled. The table also reports the number of commits per file, cluster size, co-change pattern, and key words retrieved from topics to describe clusters semantically. The column *score* in the table shows the ranking of the clusters. The column Co-change Pattern describes the fine-grained pattern for each Tentacled Cluster, i.e., clusters categorized as Octopus or Squid. We can observe that all the top ten clusters are categorized as Octopus. The top one Tentacled Cluster was detected in WebscaleSQL system and the second and third positions are WordPress' clusters. We analyzed their distribution maps, the extracted topics, source code, and documentation to understand which concerns these clusters implements. Their concerns are presented as follow:

- In WebscaleSQL's cluster most part of its body is centered on the core folder (`sql`). The body implements low level functionality, such as the parser, statement routines, global schema lock for `ndb` and `ndbcluster` in `mysqld` (MySQL embedded), binary log, and the optimizer code. The tentacles touch low level routines for file access, performance schema (private interface for the server), MySQL binary log (file reading), and client-server protocol (`libmysql`).
- WordPress's cluster (score 226) the body is the core which implements the WordPress frontend (themes, comments, post) and the tentacles are utility and admin functions.
- WordPress's cluster (score 215) the body defines plugins and themes. Specifically, the body implements the default theme for WordPress in 2015 (Twenty Fifteen theme) and the tentacles are administration APIs (post, template, scheme, dashboard widget, media).

6.3. Historical Analysis

We also analyze the evolution of the top one cluster for each pattern in terms of maintenance activities from 2008 to 2014 (6.5 years). The three clusters contain a collection of 1,168 (Haml), 1,602 (V8), and 16,737 (Webscalesql-5.6) commits. We consider the time frame of six months for all three clusters and extract the most frequent topic (mode statistics measurement) in each semester. This allows us to observe which maintenance activities are most common in each cluster and how they evolve over time.

Wrapped Cluster (or Encapsulated)- Haml project. Table 17 shows the timeline of the Wrapped Cluster with the most frequent topics by semester. In this co-change cluster, the source code files had improvement tasks, such as dead code removal, testing, and updating. As this cluster contains engines of template, we can observe in Table 17 topics describing the cluster concern and maintenance tasks. The topic *"Haml docs yard fix sass"* appears in three semesters as the most frequent topic in the whole year 2009 and in 1-2010. We inspected the log messages and changes applied concerning this topic to understand whether the key word *fix* is associated with bug fixing. Only 4% of topic's commits applied changes which modify the system behavior. The remaining (96%) just change comments in source code files, e.g, the log message *"Fix a minor anchor error in the docs"*.

Scattered Cluster (or Crosscutting) - V8 project. Table 18 shows the timeline of the Scattered Cluster with the most frequent topics by semester. In this co-change cluster, the source

Table 16: Ranking of Tentacled Clusters

Score	System	Co-change Pattern	# Commits per File	Size	Topic Summary
255	Webscalesql-5.6	Octopus	42,680	167	ndb ndbcluster binlog table mysql
226	WordPress	Octopus	10,184	45	blog comment theme login post
215	WordPress	Octopus	14,021	65	twenty fifteen theme css menu
210	Ruby	Octopus	24,570	117	bignum time strftime sprintf encoding
189	JQuery	Octopus	3,036	16	ajax xhr attribute css core
176	PHP	Octopus	20,839	118	zend api library zval class
144	V8	Octopus	9,217	64	regexp bit assembler debug simulator
138	Beets	Octopus	2,913	21	album art fetchart lastgenre logging
127	Pandas	Octopus	8,636	68	timedelta dataframe series groupby sql
123	PHP	Octopus	22,910	186	ext zend openssl pcre zlib libmagic stream

Table 17: Timeline for the top one Wrapped (Encapsulated) Cluster

Period	Wrapped Topics	(Freq %)
2-2008	Method Add Option Util	16
1-2009	Haml docs Yard Fix Sass	34
2-2009	Haml docs Yard Fix Sass	16
1-2010	Rails Make Test	21
2-2010	Haml docs Yard Fix Sass	12
1-2011	Rails Make Test	17
2-2011	Sass Rails Support Update Add	29
1-2012	Remove Code Unused Dead	28
2-2012	Method Add Option Util	20
1-2013	Version Bump Beta	4
2-2013	Rails Preserve Check Find Automatically	22
1-2014	Method Add Option Util	38
2-2014	HTML Escape Strings foo Character	17

Table 18: Timeline for the top one Scattered (Crosscutting) Cluster

Period	Scattered Topics	(Freq %)
2-2008	fix bug error	17
1-2009	Code review chromium	27
2-2009	ast node expression	27
1-2010	Code review chromium	21
2-2010	Code review chromium	18
1-2011	Fix bug error	10
2-2011	Compile mips crankshaft	11
1-2012	Remove mips profiler	10
2-2012	Remove array descriptor	12
1-2013	Add arm type feedback	11
2-2013	Revert mode stub	9
1-2014	Fix type feedback	8
2-2014	Add mips support	21

code files had different maintenance tasks, such as code review, new feature, and removal functionality. This cluster had frequent commits associated to bugs only in its initial semester (2-2008) and in (1-2011). However the absolute number of commits fixing bugs in these two semesters is quite small, only 27 commits. Finally, we analyzed the log messages related to the topic *"Fix type feedback"* to check if the term *fix* was related to some commits associated to bug fixing. We could observe there are no commits associated to bug correction.

Tentacled Cluster (or Octopus) - Webscalesql-5.6 project. Specifically, the topic model technique used in this work extracted 30 topics from all cluster's log messages and 19 topics describe bug fixing. An amount of 12,742 (76%) commits in this cluster were classified to topics concerning bug fixing. We grouped these topics by semester to comprehend the concentration of bug fixing tasks. Table 19 shows the timeline of the Tentacled Cluster with the most frequent topics by semester. In contrast to the other clusters, topics which dominate the timeline of the most changed cluster are concerning to bugs (10 out of 13 semesters). Our results suggested that concerns implemented by Tentacled Clusters tend to be difficult to localize changes. To comprehend the findings, a deep investigation is needed to analyze whether the high occurrence of bugs comes from the complexity of these concerns and their implementation. More specifically, if their complexities increase the difficult to maintain and evolve, consequently, increasing the chances to insert bugs. A possible solution to answer this question, it would be to define a method for detecting whether these bugs are concerning changes applied between body and tentacles or between tentacles.

Table 19: Timeline for the top one Tentacled (Octopus) Cluster

Period	Tentacled Topics	(Freq %)
2-2008	fix bug warning build	12
1-2009	fix bug warning build	15
2-2009	mysql backport revno timestamp	15
1-2010	join table outer pushed	11
2-2010	fix bug warning build	10
1-2011	ndb ndbcluster remove binlog	13
2-2011	fix merge bug post	10
1-2012	bug log binlog transaction	9
2-2012	bug select result fix	9
1-2013	bug select result fix	16
2-2013	slave log master bug	11
1-2014	slave log master bug	15
2-2014	slave log master bug	19

7. Threats to Validity

First, we evaluated 123 distinct projects implemented in different languages, with a large variety regarding size and domains. Despite attempts to cover several variables which may impact our conclusions, we may not generalize to other systems even for those implemented in programming languages considered in our study (external validity). Second, there are some factors that could influence our results and they are directly related to the threshold settings used in the experiment (internal

validity). For co-change clusters and patterns, we reused thresholds defined in our previous work due to the extensive investigation to define them. As another internal threat to validity is the threshold set to define the number of topics. To tackle this problem, we followed the guidelines suggested by Mallet tool's documentation[6]. Third, there are also some possible threats due to imprecision of the co-change relation measurements performed in our study (construct validity). More specifically, our technique relies on pre and pos processing steps of commits to build co-change graphs. For the time window frame parameter, we performed the calibration in all projects used in our study to compute co-change bursts (see Section 4.3).

8. Related Work

In a previous work, we investigate experts' perception of six oriented-object systems about co-change clusters (Silva et al., 2015b). However, we limited our study on three co-change patterns that cover around 52% of the clusters (Encapsulated, Crosscutting, and Octopus). In this study, we classify co-change clusters into six patterns and conduct a series of empirical studies in a large corpus with different languages. In summary, with additional co-change patterns presented in this work, we could increase the coverage to 95% of clusters.

Co-change relations are mined from version history to support on several aspects in software evolution and maintenance such as, bad smell detection (Palomba et al., 2013), improvement of defect detection techniques (D'Ambros et al., 2009b), program omprehension (Beyer and Noack, 2005; D'Ambros et al., 2009a), logical dependence detection (Alali et al., 2013; Oliva et al., 2011), and recommendation of changes (Zimmermann et al., 2005; Robillard and Dagenais, 2010).

Ball et al. (1997) introduce co-change graphs and later, (Beyer and Noack, 2005) propose a visualization technique which reveals clusters of co-change artifacts. Similar to our work, software artifacts are represented by vertices in the graphs. While they cluster all software artifacts, we follow a cleaning step to select the relevant source code files during dataset setting, co-change graph building, and co-change cluster detection phases. Finally, our goal is not propose software visualization but detecting co-change patterns from clusters to support modularity assessment.

Mondal et al. (2013) and Gîrba et al. (2007) define co-change patterns to investigate software evolution. Similar to our goal, both use co-change patterns to detect hidden dependencies among different parts of the system. Specifically, Mondal et al. detect a single method co-change pattern to support on the identification of methods logically coupled with several other methods. They apply constraints on association rules mined from version history to identify the pattern. Despite the idea of using co-change patterns is similar to ours, their work is centered on association rules at method-level while we focus on clustering of code files. Gîrba et al. use concept analysis to identify co-change patterns which affect several entities in the same time. However, their proposed patterns differ from ours because they extract complexity of methods, shotgun surgery bad-smell, and number of children in the classes. In contrast to both approaches, we detect patterns which represent common instances from co-change clusters concerning encapsulation, ripple-effect, and crosscutting behavior.

Vanya et al. (2008) use co-change clusters to decrease coupling between parts of a system. First, they recover information from version history on a higher abstraction level (directories) for clustering phase. Then, co-change clusters are associated with the current partition of the software. Although their goal on the usage of co-change clusters is similar to ours, we can highlight substantial differences: (a) we apply several preprocessing and post-processing phases and after extracting co-change clusters to filter out noises; (b) we focus on file level instead of directories; (c) we aim to guide developers on modularity analysis according to co-change patterns.

Beck et al. (2016) introduce a visualization approach centered on clustering for structuring and re-modularizing software. The authors consider different data such as, structural dependencies, semantic information, and logical information to use coupling concepts associated with distinct modularization criteria (Beck and Diehl, 2011). In summary, their approach compares the current modularization in package to a set of clustering results. As opposed to our work, their visualization enables users to identify modularization patterns that may suggest the criteria applied for a module construction, while co-change patterns can reveal design anomalies.

Zimmermann et al. (2005) proposed an approach that relies on association rule mining on version histories to suggest possible future changes. 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 modified too). On the other hand, we do not aim to recommend co-changes, but to assess modularity, using distribution maps to compare and contrast co-change clusters with the current package decomposition of a system.

In a recent work, Moonen et al. (2016) extract association rules from change histories to derive practical guidelines for change recommendation. Specifically, they assess how different parameters of the mining algorithm and characteristics of software changes impact the quality of change recommendations. Palomba et al. (2013) also mine association rules from versioning systems but they focus on source code smell detection. They identify five smells and define different heuristics for which historical data can support in the discovery process. There are two key differences between our study and those works. First, co-change clusters are coarser-grained structures than set of classes in association rules. Thus, developers can reduce the effort on program comprehension activities. Secondly, we aim to detect co-change patterns for supporting on change propagation analysis among modules.

Nguyen et al. (2016) propose TasC, a technique centered on task context to recommend source code changes. TasC performs LDA on changed code fragments extracted from version

[6] `http://programminghistorian.org/lessons/topic-modeling-and-mallet`.

Table 20: Comparison with related work

Technique	Study	Goal	Results
Semantic clustering	Santos et al. (2014)	Software remodularization	Distribution maps of semantic clusters
Co-change clustering	Beyer and Noack (2005)	Software visualization	Clusters of co-change artifacts
	Vanya et al. (2008)	Software remodularization	Logical coupling between modules
Change clustering	Robillard and Dagenais (2010)	Support change tasks	Clusters matching a query (e.g., method name)
Association rules	Zimmermann et al. (2005)	Predict co-changes	Warn about unchanged files
	D'Ambros et al. (2009a)	Co-change visualization	Logical coupling between modules
	Palomba et al. (2013)	Identify code smells	Detection of five types of code smells
	Gîrba et al. (2007)	Detect bad smells and hidden depend.	Co-change patterns
	Mondal et al. (2013)	Detect logically coupled methods	Detection of one method co-change pattern
RTM	Bavota et al. (2014)	Software remodularization	Candidates to *move class* refactoring
Scattering changes	Nucci et al. (2018)	Improve bug prediction models	Structural and semantic scattering changes

histories and outputs a ranked candidate list. In contrast, our goal is not change recommendation but using co-change patterns to support modularity analysis.

Closely related to a part of our experiment, Kouroshfar (2013) investigates the impact of co-change dispersion on software quality. They conduct a small experiment and present some evidence that co-changes localized in the same subsystem involve fewer bugs than co-changes crosscutting distinct subsystems. Such result also arises in our findings. Specifically, we observe that changes with body and arms—Squid and Octopus—have the number of bug fixing tasks significantly higher than changes confined in modules.

Beck and Diehl (2010, 2013) compare and combine the evolutionary and structural dependencies to recover modular designs. They perform clustering experiments to extract the architecture of Java projects. Their result reveal that clustering approaches based on logical dependencies succeed only when substantial data is available. In this work, we conduct a large scale study and consider more than 2 million of commits.

Kawrykow and Robillard (2011) investigate the frequency of non-essential changes, which are changes that affect the source code of an entity without changing its behaviour, such as renaming a local variable or even adding or removing whitespaces. By evaluating seven software systems, they report that up to 15.5% of the changes to methods are non-essential. However, these changes are not a major threat to our co-change clustering approach, since not necessarily all detected non-essential changes are part of co-change relations. In other words, the effect of non-essential changes in our results is less than 15.5%.

Nucci et al. (2018) use the scattering of changes performed by developers to improve bug prediction models. Therefore, instead of assessing modularity, they intend to evaluate whether developers responsible for scattered changes tend to introduce more bugs. They rely on two specific measures of change scattering: *structural scattering* (the number of structural dependencies that connect two changed components) and *semantic scattering* (the semantic similarity of the vocabulary used in the source code of two changed components).

Finally, Table 20 summarizes the aforementioned works to make their differences more clear regarding the approach evaluated in this paper for assessing software modularity using co-change clusters.

9. Conclusion

In this paper, we conduct a large-scale study with GitHub projects to evaluate their modularity using the co-change clustering technique (Silva et al., 2014, 2015a). We studied projects implemented in different languages, size, and domains, aiming to increase the generalization of our findings. In summary, our findings and the implications of our study are as follows:

- **Wrapped Clusters** (i.e., Encapsulated and Well-confined) tend to implement well-defined concerns, which confirms the results of our previous work (Silva et al., 2015b). Furthermore, the most frequent maintenance activities in such clusters are associated to improvements and new features. Therefore, the number and proportion of wrapped clusters tends to be a proxy for well-modularized systems (*Implication #1*).

- **Scattered Clusters** (i.e., Crosscutting and Black Sheep) also tend to implement a single concern, but this implementation crosscuts many directories. In addition, most activities performed by developers in such clusters are also related to improvements and new features. However, it is not straightforward to move these files to a single directory. For example, the first V8 cluster in Table 15 includes classes that support code generation in different platforms (each platform has its classes in a directory). As a second example, the second V8 cluster is responsible for compiler optimization techniques (each technique is implemented n a different directory). We found that changes that crosscut such directories are common, leading to scattered clusters. Therefore, tools and techniques for physical (e.g., aspect-oriented programming (Kiczales et al., 1997)) or virtual separation of concerns (e.g., CIDE (Kästner et al., 2008)) can help developers to reason and understand these clusters, before performing maintenance tasks (*Implication #2*).

- **Tentacled Clusters** are associated to rippling effects, density of activity, diversity in the development teams, and ownership. Moreover, they also differ from the other clusters in terms of maintenance activities. We observed in the timeline of the most changed Tentacled Clusters

that they are generated by bug fixing tasks (76% of commits). Therefore, we claim that quality assurance practices (e.g., testing, code reviews, static analyzers etc) can help to avoid the appearance of such clusters (*Implication #3*).

As future work, we plan to perform a follow up study with GitHub developers, similar to our previous study with Pharo and Java developers (Silva et al., 2015b), but including other programming languages and more systems, as we have analyzed in this paper.

Acknowledgments

This research is supported by grants from FAPEMIG, CAPES, and CNPq.

References

Adams, B., Jiang, Z. M., Hassan, A. E., 2010. Identifying crosscutting concerns using historical code changes. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1. ICSE '10. ACM, pp. 305–314.

Aggarwal, K., Singh, Y., 2005. Software Engineering. New Age International.

Alali, A., Bartman, B., Newman, C. D., Maletic, J. I., 2013. A preliminary investigation of using age and distance measures in the detection of evolutionary couplings. In: 10th Working Conference on Mining Software Repositories (MSR). pp. 169–172.

Almeida, H., Guedes, D., Meira, W., Zaki, M. J., 2011. Is there a best quality metric for graph clusters? In: European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part I. pp. 44–59.

Ball, T., Porter, J. K. A. A., Siy, H. P., 1997. If your version control system could talk ... In: ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering.

Bavota, G., Gethers, M., Oliveto, R., Poshyvanyk, D., de Lucia, A., 2014. Improving software modularization via automated analysis of latent topics and dependencies. ACM Transactions on Software Engineering and Methodology (TOSEM) 23 (1), 1–33.

Beck, F., Diehl, S., 2010. Evaluating the impact of software evolution on software clustering. In: 17th Working Conference on Reverse Engineering (WCRE). pp. 99–108.

Beck, F., Diehl, S., 2011. On the congruence of modularity and code coupling. In: 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE). pp. 354–364.

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

Beck, F., Melcher, J., Weiskopf, D., 2016. Identifying modularization patterns by visual comparison of multiple hierarchies. In: 24th International Conference on Program Comprehension (ICPC). pp. 1–10.

Beyer, D., Noack, A., 2005. Clustering software artifacts based on frequent common changes. In: 13th International Workshop on Program Comprehension (IWPC). pp. 259–268.

Bird, C., Nagappan, N., Murphy, B., Gall, H., Devanbu, P., 2011. Don't touch my code!: Examining the effects of ownership on software quality. In: 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE). pp. 4–14.

Blei, D. M., Ng, A. Y., Jordan, M. I., Mar. 2003. Latent dirichlet allocation. The Journal of Machine Learning Research 3, 993–1022.

Borges, H., Valente, M. T., 2018. What's in a github star? understanding repository starring practices in a social coding platform. Journal of Systems and Software 146, 112–129.

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

Cohen, J., Cohen, P., West, S. G., Aiken, L. S., 2003. Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences. Lawrence Erlbaum.

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

D'Ambros, M., Lanza, M., Lungu, M., 2009a. Visualizing co-change information with the evolution radar. IEEE Transactions on Software Engineering 35 (5), 720–735.

D'Ambros, M., Lanza, M., Robbes, R., 2009b. On the relationship between change coupling and software defects. In: 16th Working Conference on Reverse Engineering (WCRE). pp. 135–144.

Ducasse, S., Gîrba, T., Kuhn, A., 2006. Distribution map. In: 22nd IEEE International Conference on Software Maintenance (ICSM). pp. 203–212.

Fox, J., 2003. Effect displays in r for generalised linear models. Journal of Statistical Software 8 (15), 1–27.

Gîrba, T., Ducasse, S., Kuhn, A., Marinescu, R., Daniel, R., 2007. Using concept analysis to detect co-change patterns. In: 9th International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting (IWPSE). pp. 83–89.

Griffiths, T., Steyvers, M., 2002. A probabilistic approach to semantic representation. In: 24th annual Conference of the Cognitive Science Society. pp. 381–386.

Haney, F. M., 1972. Module connection analysis: A tool for scheduling software debugging activities. In: Proceedings of the Fall Joint Computer Conference, Part I. AFIPS (Fall, part I). ACM, pp. 173–179.

Kagdi, H., Gethers, M., Poshyvanyk, D., 2013. Integrating conceptual and logical couplings for change impact analysis in software. Empirical Software Engineering (EMSE) 18 (5), 933–969.

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

Kästner, C., Apel, S., Kuhlemann, M., 2008. Granularity in software product lines. In: 30th International Conference on Software Engineering (ICSE). pp. 311–320.

Kawrykow, D., Robillard, M. P., 2011. Non-essential changes in version histories. In: 33rd International Conference on Software Engineering (ICSE). pp. 351–360.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J., 1997. Aspect-oriented programming. In: 11th European Conference on Object-Oriented Programming (ECOOP). Vol. 1241 of LNCS. Springer Verlag, pp. 220–242.

Kouroshfar, E., 2013. Studying the effect of co-change dispersion on software quality. In: 35th International Conference on Software Engineering (ICSE). pp. 1450–1452.

Maletic, J., Marcus, A., 2000. Using latent semantic analysis to identify similarities in source code to support program understanding. In: 12th IEEE International Conference on Tools with Artificial Intelligence. pp. 46–53.

Mondal, M., Roy, C. K., Schneider, K. A., 2013. Insight into a method cochange pattern to identify highly coupled methods: An empirical study. In: 21st International Conference on Program Comprehension (ICPC). pp. 103–112.

Moonen, L., Alesio, S. D., Binkley, D., Rolfsnes, T., 2016. Practical guidelines for change recommendation using association rule mining. In: 31st International Conference on Automated Software Engineering (ASE). pp. 732–743.

Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., Murphy, B., 2010. Change bursts as defect predictors. In: 21st International Symposium on Software Reliability Engineering (ISSRE). pp. 309–318.

Nguyen, H. A., Nguyen, A. T., Nguyen, T. N., 2016. Using topic model to suggest fine-grained source code changes. In: International Conference on Software Maintenance and Evolution (ICSME). pp. 200–210.

Nucci, D. D., Palomba, F., Rosa, G. D., Bavota, G., Oliveto, R., Lucia, A. D., 2018. A developer centered bug prediction model. IEEE Transactions on Software Engineering 44 (1), 5–24.

Oliva, G. A., Santana, F. W., Gerosa, M. A., de Souza, C. R. B., 2011. 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.

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., de Lucia, A., Poshyvanyk, D., 2013. Detecting bad smells in source code using change history information. In: 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 11–15.

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

Robillard, M. P., Dagenais, B., 2010. Recommending change clusters to support software investigation: An empirical study. Journal of Software Maintenance and Evolution: Research and Practice 22 (3), 143–164.

Rosen-Zvi, M., Griffiths, T., Steyvers, M., Smyth, P., 2004. The author-topic

model for authors and documents. In: 20th Conference on Uncertainty in Artificial Intelligence. pp. 487–494.
Santos, G., Valente, M. T., Anquetil, N., 2014. Remodularization analysis using semantic clustering. In: 1st CSMR-WCRE Software Evolution Week. pp. 224–233.
Silva, L. L., Valente, M. T., Maia, M., 2014. Assessing modularity using co-change clusters. In: 13th International Conference on Modularity. pp. 49–60.
Silva, L. L., Valente, M. T., Maia, M., 2015a. Co-change clusters: Extraction and application on assessing software modularity. Transactions on Aspect-Oriented Software Development (TAOSD), 1–37.
Silva, L. L., Valente, M. T., Maia, M., Anquetil, N., 2015b. Developers' perception of co-change patterns: An empirical study. In: 31st IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 21–30.
Stevens, W. P., Myers, G. J., Constantine, L. L., Jun. 1974. Structured design. IBM Systems Journal 13 (2), 115–139.
Vanya, A., Hofland, L., Klusener, S., van de Laar, P., van Vliet, H., 2008. Assessing software archives with evolutionary clusters. In: 16th IEEE International Conference on Program Comprehension (ICPC). pp. 192–201.
Walker, R. J., Rawal, S., Sillito, J., 2012. Do crosscutting concerns cause modularity problems? In: Proc. of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. FSE '12. ACM, New York, NY, USA, pp. 49:1–49:11.
Wen, Z., Tzerpos, V., 2004. An effectiveness measure for software clustering algorithms. In: 12th IEEE International Workshop on Program Comprehension. pp. 194–203.
Zimmermann, T., Premraj, R., Zeller, A., 2007. Predicting defects for Eclipse. In: 3rd International Workshop on Predictor Models in Software Engineering. p. 9.
Zimmermann, T., Weissgerber, P., Diehl, S., Zeller, A., 2005. Mining version histories to guide software changes. IEEE Transactions on Software Engineering 31 (6), 429–445.